Forecasting Multivariate Time Series Data Using Neural Networks

Sigurd Øyen

Master of Science in Cybernetics and Robotics
Submission date: June 2018
Supervisor: Geir Mathisen, ITK

Norwegian University of Science and Technology
Department of Engineering Cybernetics

Summary

Over the last few years, neural networks have become extremely popular, and their usage is increasing rapidly. This project has investigated the use of neural networks for one-step time series forecasting on highly random data. *Multi-layer perceptron* (MLP), *convolutional neural networks* (CNN), *recurrent neural networks* (RNN), and *long short-term memory* (LSTM) cells are tested to see if they can give a binary classification accuracy above 50% using this data. The assignment focuses on designing a small embedded neural network with low latency.

The different neural network architectures are built using a *deep learning* library in *Python*, called *Keras*. This is a high-level software framework, built on top of either *Tensorflow* or *Theano*, for fast and easy prototyping of neural networks.

The conclusion of the study is that only the CNN satisfied the requirements of the assignment during the work of this thesis. None of the other architectures showed sign of learning generalized patterns and structures from the dataset in question. The CNN showed the most promising results, being able to extract information about the training set that increased the classification accuracy of the test. This leads the way for further development and an eventual hardware implementation of the inference phase reducing the run-time latency.

Sammendrag

I de senere årene har nevrale nettverk blitt ekstremt populære, og deres bruk øker stadig. Dette prosjektet har undersøkt bruken av nevrale nettverk for prediktering av tidsserie data hvor dynamikken i dataene er svært tilfeldige. *Multi-layer perceptron* (MLP), *convolutional neural network* (CNN), *recurrent neural network* (RNN) og *long short-term memory* (LSTM) topologier testes for å se om de kan gi en binær klassifisering med nøyaktighet over 50% på disse dataene. Oppgaven fokuserer på å designe et lite innvevd nevralt nettverk med lav forsinkelse.

De forskjellige nevrale nettverksarkitekturene er implementert ved bruk av et bibliotek for *deep learning* i *Python*, kalt *Keras*. Dette er et høynivå programmeringsrammeverk, bygget på toppen av enten *Tensorflow* eller *Theano*, for rask og enkel prototyping av nevrale nettverk.

Under arbeidet med denne oppgaven, konkluderes det med at CNN-arkitekturen tilfredsstiller kravene stilt av problemstillingen. Ingen av de andre arkitekturene viste tegn på å lære generelle mønstre og strukturer fra det aktuelle datasettet. CNN topologien viste de mest lovende resultatene. Modellen lærte generell informasjon fra treningssettet som bidro til økt klassifikasjonsnøyaktighet av testsettet. Dette peker ut veien for videre utvikling og en eventuell hardware implementasjon av *inference* delen for å redusere forsinkelsen fra input til klassifisering.

Preface

"We spend a great deal of time studying history, which, lets face it, is mostly the history of stupidity. So its a welcome change that people are studying instead the future of intelligence."
- Stephen Hawking

Acknowledgements

I would like to thank Geir Mathisen, from NTNU, for adequate guidance and support regarding the report and general information throughout the project.

I would also like to express my gratitude to Torgeir Trøite, from Embida AS, as the proprietor of the project and advisor.

Lastly, I would like to thank Steinar Øyen and Trude Støren for help with report structure, grammar and spell checking.

Assistance & design tools

I have been working on this project independently. There has been no involvement from the company or university on the code or other technical aspects of the thesis. The company, Embida AS, was involved in the planning phase of the project, and in taking superior choices, like directing the focus of the assignment. Administrative help and feedback on report structure etc. have been provided by the main supervisor, Geir Mathisen, at the Department of Engineering Cybernetics at NTNU.

All code in this project is written in Python. The neural networks are implemented using *Keras* [12], which is a deep learning library freely available. This is used with *Tensorflow* [1] as backend. The code is mostly based on *numpy* and *pandas*, using *Keras* for the neural network implementations. In addition to the *Keras* documentation, they also provide a long list of examples, available from their *GitHub* repository [43]. These examples and other online examples and guides [44] [7] [6] [64] have provided inspiration to the code.

Table of Contents

List of Tables

List of Figures

Abbreviations

SYMBOL	=	DEFINITION
AI	=	Artificial Intelligence
ANN	=	Artificial Neural Network
ASIC	=	Application Specific Integrated Circuit
BNN	=	Binarized Neural Network
CNN	=	Convolutional Neural Network
CPU	=	Central Processing Unit
DBN	=	Deep Belief Network
DNN	=	Deep Neural Network
DRNN	=	Deep Recurrent Neural Network
FP32	=	Floating Point 32
FPGA	=	Field-Programmable Gate Array
FSM	=	Finite-State Machine
GEMM	=	GEneral Matrix to matrix Multiplication
GPU	=	Graphics Processing Unit
HDL	=	Hardware Description Language
HLS	=	High-Level Synthesis
IP	=	Intellectual Property
LSTM	=	Long Short-Term Memory
ML	=	Machine Learning
MLP	=	Multi-Layer Perceptron
NLP	=	Natural Language Processing
NN	=	Neural Network
PReLU	=	Parametric Rectified Linear Unit
RBM	=	Restricted Boltzmann machine
ReLU	=	Rectified Linear Unit
RNN	=	Recurrent neural network
RTL	=	Register Transfer Level
SGD	=	Stochastic Gradient Descent
TFLOP	=	Teraflop
TNN	=	Ternary Neural Network
VHDL	=	VHSIC Hardware Description Language
VHSIC	=	Very High Speed Integrated Circuit

Part I

Background

Chapter 1

Introduction

This chapter gives a short motivation, describes the objectives, and presents the outcomes from this thesis. A part of this project is subject to confidentiality requirements. This is explained in a section 1.3. Finally, the chapter presents a brief overview of the report structure.

1.1 Motivation

Machine learning, and especially deep learning, has gotten a lot of attention lately due to the continuing increase in computational power and the availability of progressively bigger amounts of data. This enables machine learning, initially researched in the 1950s, to solve today's problems. In contrast to more traditional ways of solving problems, machine learning algorithms evolve from analyzing samples of data instead of accurately modelling all parts of a system from known models and equations. In deep learning, or so called *end-to-end* learning, all parameters of the network are trained from input data, eliminating the need for prior knowledge about the system's dynamics to build a model. Deep learning has proven to be very effective. Neural networks (NN) have become impressively accurate, even being as good as humans in tasks like image classification. We still see though, that humans perform better with degraded or distorted images, as discussed in [17] (Dodge, S. 2017) and [24] (Geirhos, R. 2017). Also, Google's AlphaZero AI won or played *remis* on all matches against the world champion chess program, Stockfish, in a 100-game match up according to the Guardian [68].

Much of the interest about deep neural networks (DNNs) has been the increasing accuracy, but this project will mainly focus on an embedded neural network. A smaller NN with focus on latency, size and accuracy. It is meant to forecast multivariate time series that contains highly random data, containing very little deterministic structures or patterns. Different NN architectures will be tested to see if they can extract any information about this type of data and thus increase the accuracy of a binary classification to over 50%.

Mainly GPUs have been used for machine learning tasks, but there has also been an increase of FPGA use, due to among other things, the potential of more computations per unit of power and reduced latency as discussed by Quoc V. Le (2011) [47]. Traditionally, programming of FPGAs requires knowledge about HDL (verilog of VHDL), and has significantly longer development time than programming of a regular CPU or GPU. This has led to an increased demand for higher levels of abstraction, speeding up a potentially slow and tedious development process. The different tools for higher level of abstraction can largely assist the developer, but at the cost of the detailed development at the Register-Transfer Level (RTL), which gives precise control over latency and throughput of the system.

1.2 Objectives

The problem of this assignment is to assess the ability of different neural network topologies to forecast highly random time series data. We will implement a one-step forecast, predicting the next single value of one time series based on the history of multiple time series or features. This is repeated, doing multiple one-step predictions without retraining the model. The data is highly random and will be predicted using no domain knowledge, meaning that the forecast is solely based on the time series itself. Other use cases, like predicting the weather or pollution can use domain knowledge, but that is not possible here since there is no such information available.

The focus of the assignment is to design small and compact neural networks with low inference latency, using a restricted number of parameters, well suited for FPGA implementation later on. Different architectures will be tested using a deep learning software framework. Since the data is highly random, containing little or maybe no structures and patterns, the goal is to test the different architectures to see if any of them can give a binary classification accuracy slightly above 50%.

The long-term goal for this work is to implement the network in an FPGA minimizing the inference latency, thus a focus on FPGA should be included in the literature review. The goal of this thesis is to give a *proof of concept* and explore which type of network will be most suited for further exploration and implementation in an FPGA. Testing and development in SW is much faster, thus all testing carried out during this project are done on CPUs and GPUs.

The objectives for this project will be the following:

1. Write a literature review and acquire necessary background knowledge: The background theory consists of, among other things, the basic understanding of deep learning, recurrent neural networks, and convolutional neural networks. It will also touch upon aspects like the opportunities of high-level synthesis and FPGA versus GPU. The literature review should also contain other NN implementations on

FPGAs and usage of NNs for time series forecasting. It should include a short evaluation at the end of the review.

2. Compose/design a theoretical suggestion of different neural network architectures based on the literature review.

3. Implementation of the typologies from 2 using a deep learning SW framework. This also includes acquiring and pre-processing of input data. Test and compare the results, and evaluate the work.

1.3 Confidentiality Requirements

The proprietor of this project is Embida AS. Due to confidentiality requirements, information about the content of the dataset is left out of the report. We will refer to this dataset as the ”mysteryset”. To ease the reader’s understanding of the report, the overall structure of the data is explained, without revealing any information about the content of the data. This is seen as sufficient for the purpose of this report. Hence this requirement, all work on acquiring data and preparations specific to the content of the ”mysteryset” is not further discussed. The code written for this project is also not included in this report due to the same confidentiality requirements.

The ”mysteryset” is composed of a couple of hundred so called *items*. Each item contains multivariate time series. There are 5-6 parameters in each item, and a few thousand samples for each parameter. A selection of parameters from different items are combined forming the time series, or *features*, of the ”mysteryset”. The selection of features is used to predict the rising or falling of the next value of one feature, i.e. a parameter in one of the items. The output or label of each time step is characterized as either *rising* or *falling*. There is not one specific feature that should be predicted. Several different features are used for prediction during the project. This is referred to as the *classification feature* later in the report.

1.4 Outcomes

The outcomes from this thesis are:

- A literature review of neural network basics, FPGA implementations, and neural networks used on time series.

- System for pre-processing and combining feature data used in this thesis.

- Training CNN models for binary classification of the dataset in question.

- Discussion about the use case and further development.

1.5 Report Structure

The report is divided into five different parts: *Background*, *Literature Review*, *Method*, *Results and Discussion*, and *Closing Remarks*.

The *Background* part gives a short motivation and introduces the problem and its objectives.

The *Literature Review* consists of three chapters; *Background Theory of Deep Learning*, *FPGA and Software Tools*, and *Review of Related Work*. The theory chapter starts from the basics of deep learning. This reflects the prior knowledge of the writer, as a cybernetics student, about the subject. It goes through the necessary theory about neural networks, including topics like basic properties of neurons, layers, multi-layer perceptron, activation functions, forward and backward propagation etc. Different network topologies can be used for time series prediction, and for that reason the chapter includes the two commonly used networks: *convolutional neural networks* and *recurrent neural networks*.

Hardware design on FPGAs is a time-consuming process, and higher level of abstraction and software frameworks are included in the *FPGA and Software Tools* chapter as a natural result of using FPGAs.

GPUs are the most commonly used platform for NNs today, the *Review of Related Work* chapter will explore the opportunities and advantages of using FPGAs instead. This chapter also covers the most famous network architectures and the techniques they introduced. Finally, related work of other implementations on FPGAs and time series forecasting using NN are reviewed.

The *Method* part includes the following chapters: *Functional Specification*, *Tools & Data Preparation*, *Design*, and *Implementation*. It gives a description of the system and briefly goes thought the tools used and shows how the datasets are prepared. The design and implementation chapters goes though the design process and the implementation of the different network topologies.

The *Results and Discussion* presents each implementation. The final discussion of the thesis has its own chapter at the end of this part.

Lastly, the *Closing Remarks* concludes the thesis and presents the future work.

Part II

Literature Review

Chapter 2

Background Theory of Deep Learning

This chapter introduces the basic terminology used in deep learning. It goes through the building blocks of the neural networks, and introduces the commonly used network topologies: *recurrent neural networks*, and *convolutional neural networks*.

2.1 Origin

Deep learning (DL) is a sub field of machine learning which again is a sub field of the much broader field of Artificial Intelligence (AI) [63, p. 4]. AI is deeply covered by other texts such as *Artificial intelligence: a modern approach* by Stuart Russell and Peter Norvig [67], and will not be covered in this thesis apart from pointing out that deep learning originates from AI.

The idea of deep learning is inspired by the biological behavior of the brain. Simplified, each neuron, the main component or building block of the brain, transmits information to other neurons forming a very large and complex network. Each node or neuron is stimulated by inputs and passes information or some part of the information on to other neurons.

In size, the artificial neural networks today are not even close to the human brain. Josh Patterson [63, p. 4], says that the brain is composed of approximately 86 billion neurons and more than 500 trillion connections between these neurons. The article *The Basics of Brain Development* [72] says more than 60 trillion connections. Big artificial NN today have billions of parameters. Examples are, *Building High-level Features Using Large Scale Unsupervised Learning* [48] with 1 billion parameters, and the company Digital Reasoning with 160 billion parameters in their neural networks [40]. The brain is very complex and there is much to be discovered before we understand the workings of the brain. A very big artificial neural network doesn't even resemble the complexity of the

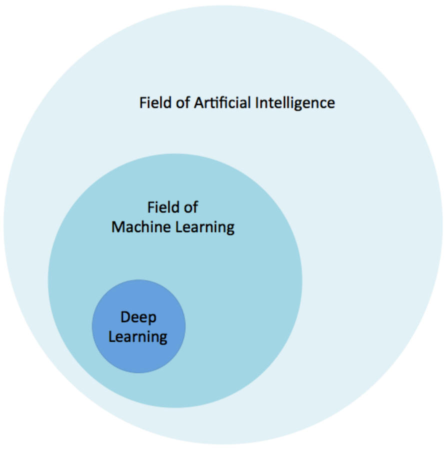


Figure 2.1: The fields of AI, ML and DL. Patterson, J.(2017) [63, p. 4]

human brain, but it's the idea behind neural networks.

2.2 Supervised and Unsupervised Learning

In machine learning there are two groups of algorithms that differ in the way they learn. These are classified as supervised and unsupervised learning. Supervised learning has samples of labeled data and learns to predict the output based on input data. Expressed in another way; the algorithm learns the function that maps the input data to the output data. This is the most popular method and the one in focus during this project. Supervised learning requires labeled data to work. Unsupervised learning on the other hand does not need labeled data. It basically learns structures in the data set. It can for example be used or categorize unlabeled data.

2.3 Classification and Regression

Classification and regression are two different types of machine learning algorithms. If a model is trained to recognize only dogs and cats, this would be a binary *classification*, since there are two outputs or *classes*. In time series forecasting this could be predicting if the next value of the series is higher or lower that the current one. The number of classes differs depending on the application. In regression the network outputs the actual predicted value instead of the probabilities for each class.

2.4 Biological Neuron

A simplified biological neuron is shown in Figure 2.2. In the most basic sense the biological neuron consists of a cell body with one axon and many dendrites. The connections between neurons are called synapses. This is the connection between the axon of one neuron and a dendrite of another. A single neuron has many dendrites and one axon, making every neuron a *multiple input single output* building block. Connecting many neurons together forms the network of the brain.

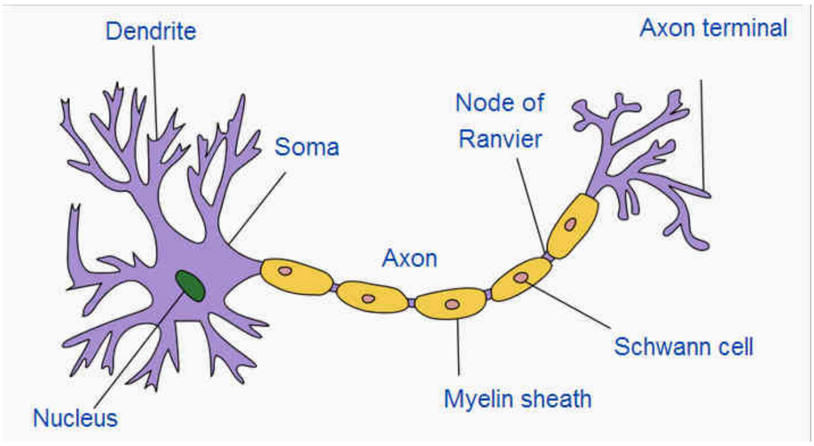


Figure 2.2: Simplified biological neuron. Harry Fairhead (2014) [33]

2.5 Artificial Neuron

The artificial neuron is based on the biological version. It consists of inputs, weights and a bias, a summation, an activation function, and the output as shown in Figure 2.3. The output of a neuron can be called the *activation* of a neuron.

The summation basically does a linear transformation on the inputs by its weights and bias as in equation 2.1. The non-linearity is introduced by the activation function which decides how much of the information from this sum to pass through to the output. There are diverse types of activation functions, also linear activation. Neural networks using only linear activations are essentially linear regression models.

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

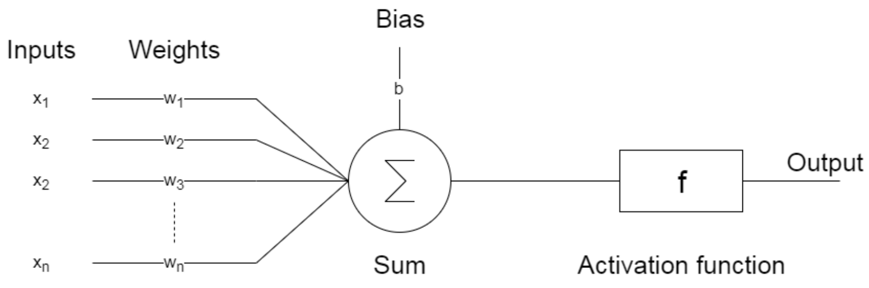


Figure 2.3: Artificial neuron

2.6 Activation Functions

Different activation functions are used for different problems. The following sub sections give a brief overview of the most common ones. Information about each function are taken from Patterson, J. (2017) [63, p. 65], and GUPTA, D. (2017) [31]. All plots of the following activation functions are made in Python.

2.6.1 Linear

Figure 2.4 shows the linear activation function $f(x) = x$. When using this activation function, the output is simply proportional to the input. It basically lets the signal through.

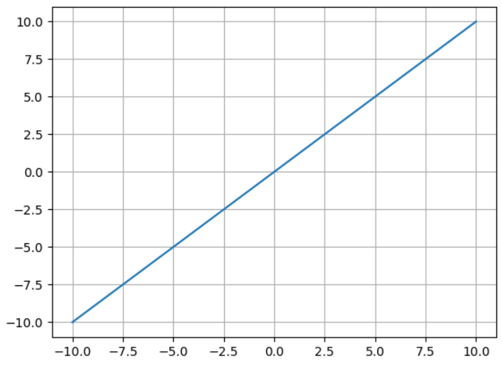


Figure 2.4: Linear Activation Function

2.6.2 Rectified Linear

The *Rectified Linear Unit (ReLU)* function shown in Equation 2.2, illustrated in Figure 2.5 is the most common activation function due to its simplicity and good results. A subset of neurons fire at the same time. This makes the network sparser, improving efficiency. With a uniform initialization of the weights, around 50% of the hidden neurons will fire according to Glorot Xavier (2011) [28]. Sparsity is discussed in more depth later in the literature review.

$$f(x) = max(0, x) \tag{2.2}$$

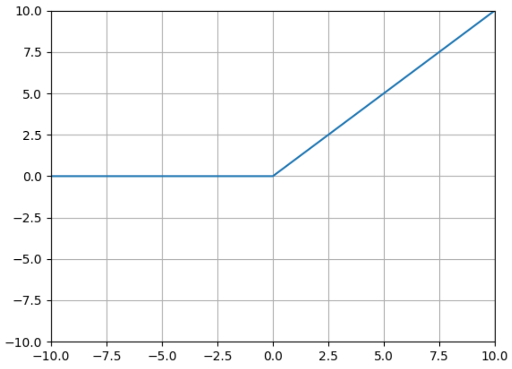


Figure 2.5: Rectified Linear Activation Function

There is also a *Leaky ReLU*. It is similar to the ReLU function except that when x is less than 0 the function has a small negative slope. *Dying ReLU* [63, p. 70] can be a problem with standard ReLU. *Leaky ReLU* prevents neurons from being totally inactive, or to have *dead neurons*, which means that the neurons are inactive for all the input samples. Solving *dead neurons* and other issues are discussed in *Solving internal covariate shift in deep learning with linked neurons* by Carles R. (2017) [65].

$$f(x) = \begin{cases} x : x > 0 \\ 0.01x : x \leq 0 \end{cases} \tag{2.3}$$

There is also a *Parameterised ReLU* function, shown in Equation 2.4. The *a*, in the equation, decides the slope for negative values of x. The added parameter is trained by the network. This activation function can be used when the *Leaky ReLU* does not solve the problem of *dead neurons*.

$$f(x) = \begin{cases} x : x > 0 \\ ax : x \leq 0 \end{cases} \tag{2.4}$$

2.6.3 Softplus

Figure 2.6 shows the *softplus* activation function. It is also a version of the ReLU function. The standard ReLU function is graphed with a red dotted line. In contrast to the ReLU this function is continuously differentiable.

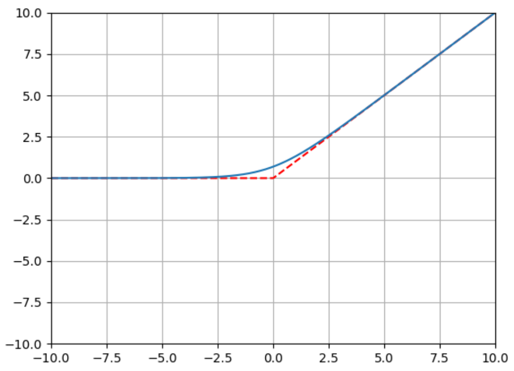


Figure 2.6: Softplus activation function

2.6.4 Sigmoid

The *sigmoid* function, in Equation 2.5, is also a very popular activation function. It squeezes the output between 0 and 1. It is continuously differentiable. The gradient of this function is highest around 0 and flattens out for higher or lower input values. Meaning that when the network falls into that region of the graph it learns slower and slower, i.e. the vanishing gradient problem. The sigmoid function is shown in Figure 2.7.

$$f(x) = \frac{1}{1 + e^{-x}} \tag{2.5}$$

2.6.5 Tanh

Tanh, shown in Figure 2.8 is similar to the sigmoid function, only that it outputs values from -1 to 1.

2.6.6 Softmax

The *softmax* function, shown in Equation 2.6 is also similar to the sigmoid function. It outputs continuous values from 0 to 1 and is often used at the output layer as a *classifier*, because it outputs the probabilities distributed over the number of classes. I.e summing up all the probabilities add up to 1 or 100%.

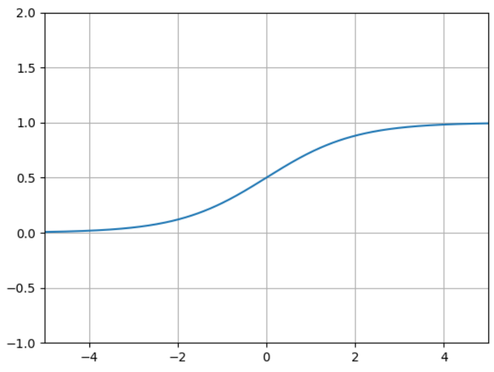


Figure 2.7: Sigmoid activation function

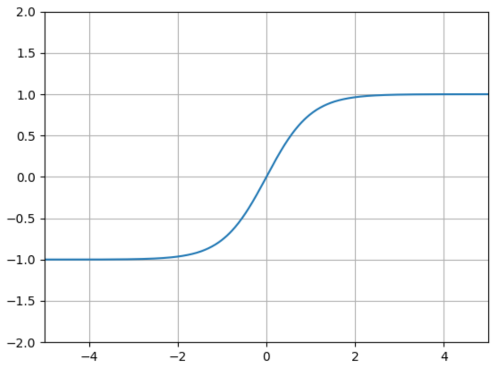


Figure 2.8: Tanh activation function

$$f(x_i) = \frac{e^{x_i}}{\sum_{j=0}^{n} e^{x_j}}, \quad for\ i = 0, 1, 2...n \tag{2.6}$$

2.6.7 Binary Step Function

The *binary step* function, shown in Equation 2.7, is basically just a threshold that says if the neuron should be active or not. It is shown in Figure 2.9.

$$f(x) = 1, \quad for\ x \geq 0 \tag{2.7}$$

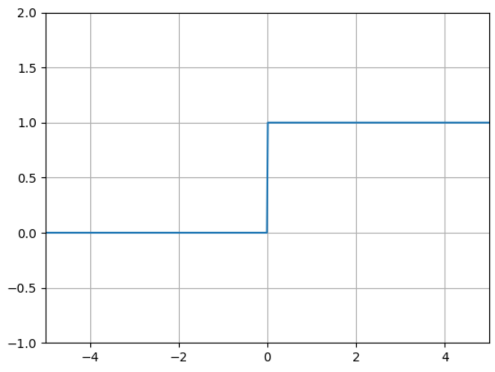


Figure 2.9: Binary Step function

2.7 Artificial Neural Networks (ANN)

Basic artificial neurons are the building block of an artificial neural network. An ANN consists of three different types of layers; the input layer, hidden layers, and the output layer [52]. There may be many hidden layers in the network. Figure 2.10 shows a basic ANN with one hidden layer with four neurons, three inputs, and three outputs. A network where each layer has multiple neurons and all the neurons in one layer are connected to the neurons in the next layer, is called a *fully connected network* or *multi-layer perceptron* (MLP).

A deep neural network, a NN with more than two layers, is to a considerable extent based on statistics and linear algebra. This review will not go in depth on these topics since they are broadly covered in other texts like Patterson, J. [63] and [55].

2.7.1 Forward Propagation

The neural network feeds (forward) information from the inputs through the hidden layers to the outputs. This movement of information through the network is called *forward propagation*.

2.7.2 Weights & Biases

Between layers in the NN, the output of neurons in one layer, or the activations of these neurons, are connected to the input of neurons in the next layer, each connection associated with a weight. These weights are the *tuning knobs* of the network. One can say that the weight is the strength of the connection between two neurons, or how much of the activation from one neuron that is carried through to the next. This can be illustrated using

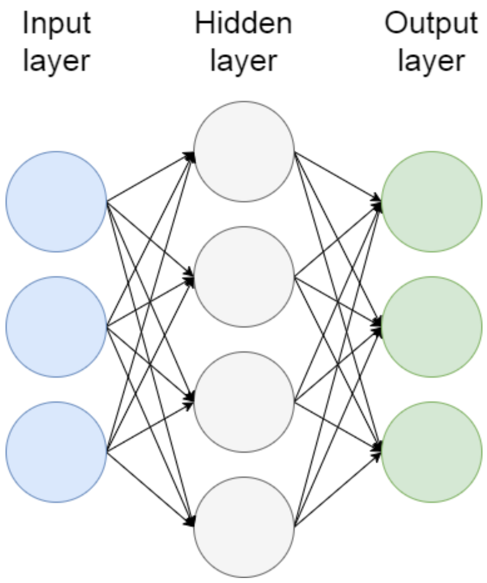


Figure 2.10: Basic Artificial Neural Network

different thickness of the connections like in Figure 2.11. The weights, and the bias that basically offsets the activation of the neuron, are the adjustable parameters of a NN.

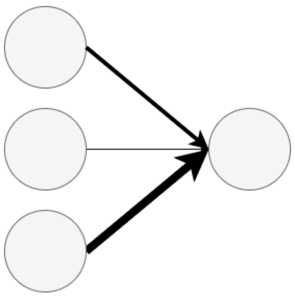


Figure 2.11: Weights between neuron i a NN

2.7.3 Parameter Optimization

The parameters, weights and biases, in a neural network are updated using a training data set. Initially, the parameters of the network can be assigned randomly. With more training data the model will more accurately resemble the real system. Machine learning (ML) finds a way to represent data based on the training set. It does not try to match the data to a mathematical model, i.e. it is not told what patterns to look for, but updates the parameters of the model based on a cost function which represents the differences between the desired values, i.e. the labels of the training data, and the actual output provided by the network.

The weights and biases are updated such that the average cost of all the training example are minimized the most.

As seen in Figure 2.12, using a simple linear function can be an under fit of the data as it in many cases does not represent the data very well. There is also a problem with overfitting in machine learning. Overfitting the model will give a very low error in the training data but does not provide a generalized solution to the problem. This can result in a significant decrease in accuracy on the test set, i.e. on new unseen inputs after training, as it also will account for noise and outliers in the training set.

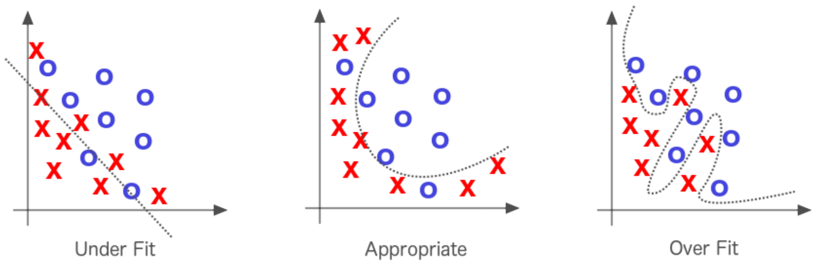


Figure 2.12: Underfitting & overfitting. Patterson, J (2017) [63, p. 27]

The process of updating the parameters of the model is called *parameter optimization*, and is basically adjusting the weights based on the cost function. The weights are adjusted such that the cost function decreases most efficiently. A popular method is the first order optimization using gradient descent as it is easy to use and less time consuming and computationally heavy than for example using the hessian for second order optimization.

2.7.4 Gradient Descent

Gradient descent is basically how the network learns. Training data is fed through the untrained network, and the weights and biases are adjusted. Afterwards the network is tested with unseen data. The way it works, is finding the minimum of a function. The strength or weight between the connections in the network are initially random and the network will perform terrible. The function to minimize is called the *cost function*. The squared of the differences between each of the outputs and desired value of the outputs are summed up. That is the cost of one training example. The inputs are the weights and biases and the cost function outputs a single number, the average cost of all the training samples. With one or two input parameters it is very easy to visualize moving a little bit in the direction that decreases the value of the function the most like shown in Figure 2.13. It becomes harder to visualize this with thousands of parameters, but the idea is still the same. The negative gradient of the cost function is a vector. There is some direction inside the high dimensional space that tell which nudges to all the parameters are going to cause the most rapid decrease of the cost function. The algorithm for computing the gradient is

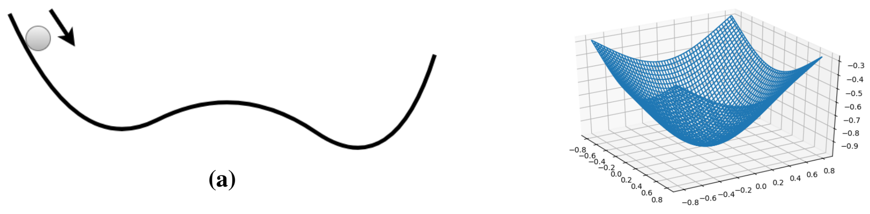


Figure 2.13: Illustrations of gradient descent

called *backpropagation*. The *learning rate* decides how much the parameters are updated per iteration. Setting it too big can make the function unable to settle on a good value making the network worse. A network with too low learning rate takes very long time to train. Choosing a learning rate proportional to the slope prevents it from overshooting.

2.7.5 Backpropagation

Each step takes the average cost of all the training samples. Each activation is a weighted sum of all the activations of the previous layer and a bias. I.e. the error is dependent of these weights, the bias and the activations from the last layer. Since the activations are dependent of the previous layer and cannot be directly altered, we back propagate though the network, adjusting the weights. Using every training sample for every gradient descent step takes a long time to compute. Stochastic gradient descent (SGD) is used to make this process faster. It basically randomizes the order of the input data and splits it up into mini batches. A step is computed according to the mini batch. This does not give exactly the correct *direction* in the high dimensional space to move in as it does not accord for the whole training, but using a subset gives a good approximation.

Batch size, Iterations, and Epochs

These terms are easily explained using an example: If there are 10000 training samples divided into 10 batches. The batch size is 1000, and there are 10 iterations for each epoch. The number of epochs represents the number of times the model has trained on all the training samples in the data set.

2.7.6 Training Phase and Inference

Normally the dataset is split into a training set, a validation set, and a test set. During the training phase, the training data is used to update the parameters of the network. The validation data is used during training to monitor the training process and to detect e.g. overfitting. *Inference* is when the trained model is tested with new unseen data. E.g. when a trained model is deployed and used in a live application.

2.7.7 Sparsity of NN

This sub section is mostly based on the paper from Xavier Glorot, Antoine Bordes, and Yoshua Bengio: *Deep Sparse Rectifier Neural Networks* [28]. Using activation functions like ReLU which outputs 0 for negative input values naturally makes the network sparse. This can have some advantages over a non-sparse network. *Pruning* is another technique to achieve sparsity. It identifies non-important neurons and sets them to zero.

"We argue here that if one is going to have fixed-size representations, then sparse representations are more efficient (than non-sparse ones) in an information-theoretic sense, allowing for varying the effective number of bits per example". Yoshua Bengio (2009) [4]. Sparse representations allow the network to vary the effective dimension and required precision of a given input. Using ReLU the output is a linear representation of the subset of active neurons.

Using a sparse NN, results in a less entanglement network making it easier to identify the factors explaining the variations in the data. Sparse NN gives a computational advantage in comparison to a dense network and it can contribute to reducing the problem of overfitting. Sparse networks are becoming more popular, as the accuracy of the NNs don't decrease significantly when introducing a sparser network. *"Maximum sparsity is obtained by exploiting both inter-channel and intra-channel redundancy, with a fine-tuning step that minimize the recognition loss caused by maximizing sparsity. This procedure zeros out more than 90% of parameters, with a drop of accuracy that is less than 1% on the ILSVRC2012 dataset"* Liu, B, (2015) [51].

2.7.8 Dropout

Dropout is a technique used under training to avoid overfitting. As the name suggest is drops out random neurons in the hidden layers. This means that the neurons are temporarily removed from the network. An illustration of dropout neurons is shown in Figure 2.14 taken from *Dropout: A Simple Way to Prevent Neural Networks from Overfitting* [71]. For each presentation of each training case a different *reduced* network is used. During the *inference* phase all neurons are active.

2.7.9 Data Augmentation

Having too few training samples is a frequent problem using neural networks, as they often need many samples to create a good generalization of the problem. *Data augmentation* is creating new input data from already given inputs increasing the number of samples. This is useful in application which has a restricted number of training samples available. Examples of data augmentation on images are mirroring, rotations, random cropping and color shifting.

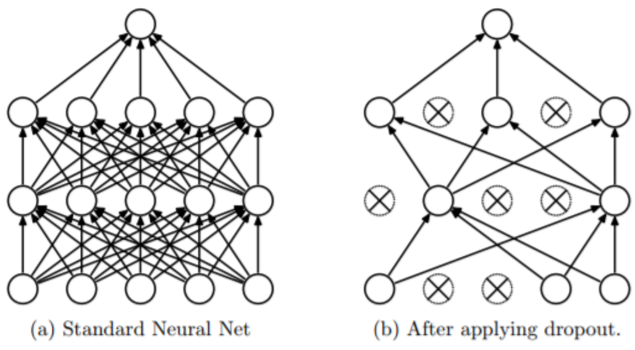


Figure 2.14: Illustration of dropout in a NN. Srivastava, N (2014) [71]

2.8 Convolutional Neural Network (CNN)

Convolutional neural networks are a type of neural network that has gained a lot of momentum lately partly due to its great ability to classify objects in images. It learns to recognize features through convolution. It utilizes that pixels closer together in an image are more related to each other than pixels far apart. For classifying images, MLPs does not scale very well. It takes the input as a one-dimensional vector and passes the data through the fully connected hidden layers. This is fine for small images. 10 pixels by 10 pixels image and 3 RGB channels will give 300 weights per neuron in the first hidden layer. A 640x480 pixels image and 3 RGB channels will on the other hand give 921600 weights per neuron in the first hidden layer.

The CNN basically consists of several types of layers stacked on top of each other. There is no given way to stack the different layers, that is up to the designer. Using object classification is a very intuitive example going through the basics of CNNs, but they can be used on other types of data like text or sound, they are even being used to make computers learn to play video games. Zhao Chen (2017) [10].

In the following sub sections different types of layers, *input layer*, *convolutional layer*, *pooling layer*, *fully connected layer*, and *batch normalization* will be described.

2.8.1 Input Layer

The input layer stores the raw input data. It is a three-dimensional input consisting of the width and height of the image and the depth is represented by the color channels, typically three for RGB.

2.8.2 Convolutional Layer

The convolution layer is the key layer of the CNN. It uses filters, or *kernels*, that basically is a smaller image than the input. Convolution is done with a part of the input and the kernel. This is done in a sliding window manner, ultimately covering the whole input image, as illustrated in Figure 2.15. It is done for every depth of the input. The output from this process is called a *feature map* or an *activation map*. The region of the input the feature map is looking at, is called the *receptive field*. Each filter results in a feature map. The activation map for each filter are stacked outputting a 3-dimensional tensor. As the filters are trained they learn to recognize edges and patterns, and deeper in the network they can recognize more advanced shapes. The input to a convolution layer is either the NN input or the feature map output from another convolution layer.

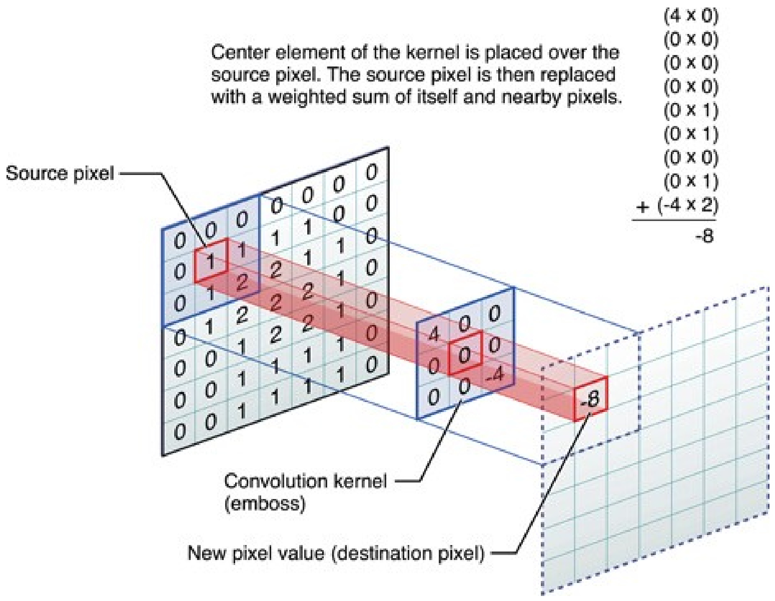

Center element of the kernel is placed over the source pixel. The source pixel is then replaced with a weighted sum of itself and nearby pixels.
(4 x 0)
(0 x 0)
(0 x 0)
(0 x 0)
(0 x 1)
(0 x 1)
(0 x 0)
(0 x 1)
+ (-4 x 2)
-8
Source pixel
Convolution kernel (emboss)
New pixel value (destination pixel)

Figure 2.15: Illustration of convolution Engineering, H (2015) [18]

Very commonly used in CNNs are the ReLU activation function. This layer basically takes all the negative inputs and sets them to zero. The ReLU layer has no hyperparameters, i.e. parameters that are chosen by the designer.

2.8.3 Pooling Layer

The pooling layer reduces the size of the data. The most common version is *max pooling* which outputs the maximum value of the given window size and ignores the rest. It does this operation over the whole input. The stride is chosen by the designer. With a common window size of 2x2 and a stride of 2 the reduction would be 75% of the original size like shown in Figure 2.16. Pooling doesn't care about where in that window the maximum value is which makes it a little less sensitive to the position and helps to control *overfitting*.

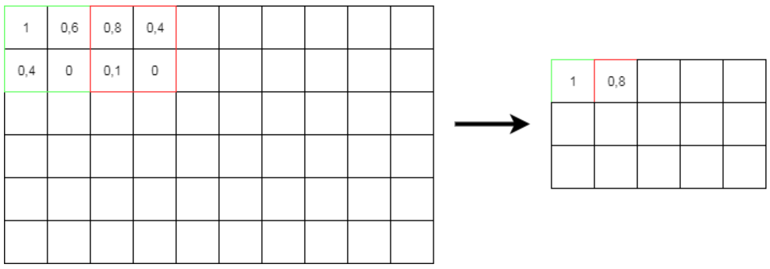


Figure 2.16: Max Pooling layer example

2.8.4 Fully Connected Layer

Typically, at the output, or classification of the CNN we have one or multiple fully connected layers. The classifier outputs probabilities for the different classes. Figure 2.17 shows an illustration of the famous CNN AlexNet developed by Alex Krizhevsky, Geoffrey Hinton, and Ilya Sutskever.

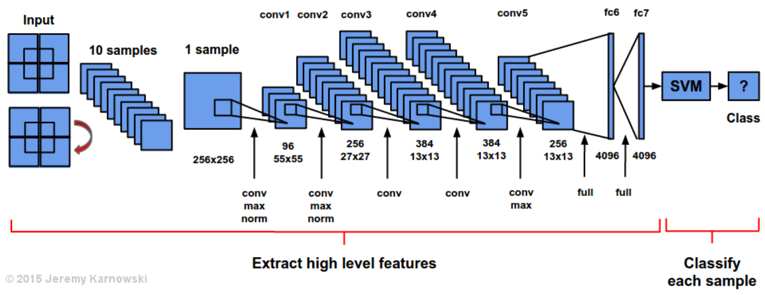


Figure 2.17: Illustration of AlexNet. KARNOWSKI, J. (2015) [42]

2.8.5 Batch Normalization

It is common to normalize the data before inputting it to the NN. *Batch normalization* normalizes the mean of the layer's output activation close to 0 and its standard deviation close to 1. This method is commonly used to accelerate the training of CNNs.

"The training is complicated by the fact that the inputs to each layer are affected by the parameters of all preceding layers so that small changes to the network parameters amplify as the network becomes deeper" Ioffe, S., Szegedy, C. T(2015) [39]. As the title says: *Batch Normalization: Accelerating Deep Network Training by Reducing Internal*

Covariate Shift, and the paper defines the internal covariate shift as the: *"change in the distribution of network activations due to the change in network parameters during training"*. As the problem becomes much more severe as the networks gets deeper, batch normalization layers are more needed in these cases. The two extra parameters introduced by batch normalization are also trained by the network.

2.9 Recurrent Neural Network (RNN)

Recurrent neural networks can be used for all sequential forms of data like video frames, text, music etc. The feed-forward networks input some value to the network and returns some value based on that input and the network parameters. A RNN has an internal state that is fed back to the input, illustrated in Figure 2.18. It uses the current information on the input and the prediction of the last input. The time steps of a recurrent neural network are often illustrated as in Figure 2.19.

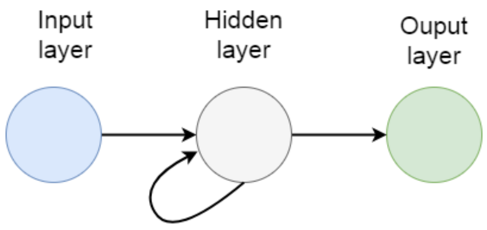


Figure 2.18: Illustration of RNN

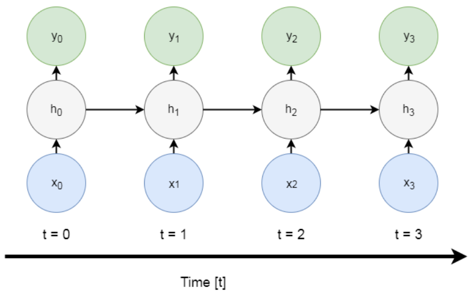


Figure 2.19: Illustration of the time steps of a RNN

Feed-forward networks have a fixed size input and output. For example, an image has a given number of pixels as inputs, and a number of classifiers as outputs. This is not the case with RNNs. They can have *one-to-many*, *many-to-one* or *many-to-many*. For example, *one-to-many* image classification could input an image and output a sequence of words. *Many-to-one* could be used for sentiment analysis, and *many-to-many* can for example be sentence translation from one language to another.

RNNs have a problem with *vanishing gradient descent*. This can happen when the gradient of the activation function becomes very small. When back-propagating through the network the gradient becomes smaller and smaller further back in the network. This makes it hard to model long dependencies. One way of getting around this is to use *long short-term memory* (LSTM), which is a variant of the RNN. The opposite of the vanishing gradient problem is the *exploding gradient problem* where the gradient gets to large.

2.9.1 Long Short-Term Memory (LSTM)

The LSTM block consists of three so called gates; the forget gate, the input gate, and the output gate, in addition to the input and output blocks and the memory cell. Figure 2.20 shows an illustration of the block. The vector formulas for the LSTM can be found in *LSTM: A Search Space Odyssey* [29].

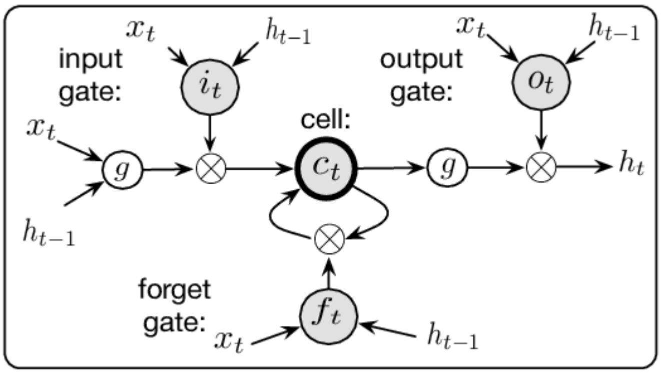


Figure 2.20: Illustration of a LSTM block. Chen, J. (2016) [9]

Forget Gate

The LSTMs lack of an effective way to reset itself was solved introducing the forget gate to the network [27]. The forget gate says how much of information from the input x_t and the last output h_{t-1} to keep. 1 is hold on to everything and 0 is forget everything.

Input Gate

The input gate says how much of the information that should be stored in the cell state. It prevents the cell from storing unnecessary data.

Output Gate

Lastly, the output gate decides how much of the content in the memory cell to expose to the block output.

Chapter 3

FPGA and Software Tools

This chapter gives a brief introduction for ways to increase the abstraction level in FPGA development and gives a brief review of the different software frameworks available.

3.1 High Level Synthesis (HLS)

Since the low-level design on the register transfer level (RTL) can be quite time consuming and prone the errors, tools providing a higher level of abstraction has lately become increasingly popular.

High-level synthesis (HLS) basically converts C-code to hardware description language (HDL) code. HLS is provided by both Xilinx, Intel FPGA and others. The HLS compiler inputs the code together with a specification which contains information about things like accuracy, speed and area. The main steps of the HLS design are shown in Figure 3.1.

The HLS tool executes the following tasks [15]:

- Compiles the specification
- Allocates hardware resources (functional units, storage components, buses, and so on)
- Schedules the operations to clock cycles
- Binds the operations to functional units
- Binds variables to storage elements
- Binds transfers to buses

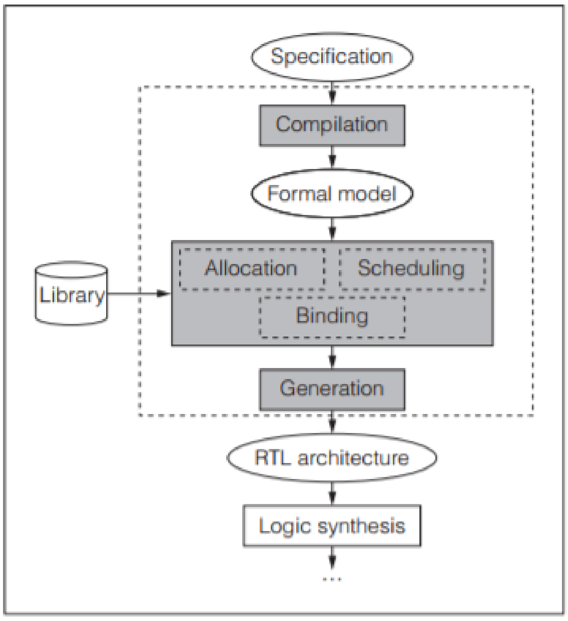


Figure 3.1: HLS design steps. Coussy, P. (2009) [15]

- Generates the RTL architecture

The paper from Philippe Coussy goes through all the steps in more detail. In short, the tool finds dependencies, unrolls loops, and utilizes pipelining to optimize the design. The tool identifies needed RTL components, and decides what the clocking scheme should look like; dependent of input specifications such as area, delay, and power. Operations can be chained or scheduled to execute in parallel provided there are no dependencies and sufficient amount of resources available. The output is the data flow and the control unit which typically is a finite-state machine (FSM).

HLS it typically not used on system level design but used to solve some function $f(x)$. The module can then be included in the rest of the design. The Intel HLS compiler creates an intellectual property (IP) core [21] of the design making it a self-contained component that can be used in Qsys [22] as any other IP core.

3.2 OpenCL

OpenCL [73] is an open source C-based programming language for writing parallel computational programs which can run on heterogeneous platforms like CPUs, GPU and FPGAs. It is a low level, high performance programming language. OpenCL can be used instead of writing RTL code for an FPGA or to have snippets of code independent of platform.

3.3 Software Frameworks

There are a lot of different deep learning software frameworks. This section will take a brief look at some of them, with focus on the ones with support for Python. This section is based on online articles [38] [80] [54] [56] and the frameworks' own documentation.

- *TensorFlow* [1] is an open source Python based machine learning library developed by Google. It uses data flow graphs to represent the computations. It is said to be easy to use.
- *Theano* [76] is a popular software framework for deep learning. It is a Python library developed at the University of Montreal, LISA Lab. It is said to be somewhat harder to use compared to e.g. TensorFlow.
- *Torch* [13] is a library based on Lua scripting language with C/Cuda implementation.
- *Keras* [12] is a higher level deep learning library used on top on either TensorFlow or Theano.
- *Caffe* [41] is primarily a C++ library. It is much used for implementing convolutional neural networks. It is developed at Berkeley Vision and Learning Center.
- *Neon* is also a Python library. It is said to be easy to use as well as very fast. It is developed by Intel.
- *CNTK* [69], developed by Microsoft is very fast and outperforms TensorFlow on performance, but it is somewhat harder to use. It has support for languages such as Python and C++.

Chapter 4

Review of Related Work

This chapter is built up of four sections:

- FPGA vs GPU
- Well Known Neural Network Architectures
- FPGA Implementations
- Neural networks in Time Series Forecasting

The first section reviews the usage of FPGA versus GPU in deep learning applications. Secondly, well known CNN architectures are presented. It focuses on the layers introduced by each model, and what makes them solve new problems improving performance. Further, neural network implementations on FPGA are reviewed. And finally, different neural network topologies used for time series forecasting are reviewed.

4.1 FPGA vs GPU

On the question of whether FPGAs are the new platform for deep learning, there are several online articles [3] [58] [23] that refer to the same paper, *Can FPGAs Beat GPUs in Accelerating Next-Generation Deep Neural Networks?* [60]. It reflects on the opportunities of the new FPGAs vs GPUs in deep neural networks.

GPUs are the most favored platform used in deep learning. FPGAs have proven to be energy efficient but has not provided the performance or TFLOP/s as the GPU. Traditional DNNs rely much on dense *GEneral Matrix to matrix Multiplication* (GEMM) on FP32 data type. This is very favorable for the GPU. It performs very well on regular parallelism and gives high floating point computational throughput. The GPU only supports a fixed set of native data types. In contrast, the FPGA offers extreme flexibility which makes it better for irregular parallelism and custom data types. With new FPGA technologies there are not much difference in performance on FP32 TFLOP/s either according to Nurvitadhi, E (2017) [60]: *”integration with high-bandwidth memories (up to 4x250GB/s/stack or 1TB/s), and improved frequency from the new HyperFlex technology, thereby leading to a peak 9.2 TFLOP/s in FP32 throughput. In comparison, the latest Nvidia Titan X Pascal GPU offers 11 TFLOPs in FP32 throughput”.*

Deeper neural networks have shown more accuracy. This increases the computational demand, and the focus on efficiency has thus increased. As discussed earlier, sparsity is one way of making the network more efficient. Another popular approach is to use more compact data types. *”Many researchers have shown (e.g., [6,7,10,11]) that it is possible to represent data in much less than 32-bits, demonstrating the use of 8-4 bits (depending on the network) leads to only a small reduction in accuracy compared to full precision.”* Nurvitadhi, E. (2017) [60]

Binarized neural networks (BNN) are very efficient at the expense of some accuracy. For most efficiency 1 bit is used for both neurons and weights. Another type of network is the ternary neural network (TNN). It uses 2 bits for the weights and fp32 for the neurons.

As discussed by Vivienne Sze (2017) [74] there are also techniques like *weight sharing*, meaning several weights share the same value, using varying size filters, *pruning*, and batch normalization. *GoogleNet* [75] for example uses 1x1 convolutional filters to increase the efficiency of the network.

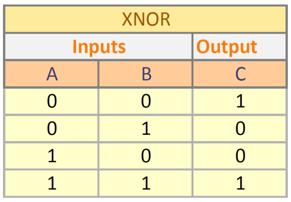
The increased use of sparse networks and compact data types might indicate that FPGA is the new platform for deep learning, but as of today GPUs are still the most widely used platform. Traditionally FPGAs have not been able to compete with GPUs, but devices like Intel’s Stratix 10 with 5000 floating-point units and over 28MB of on-chip RAM integrated with high-bandwidth memories, might level the playing field.

4.1.1 Acceleration of BNNs

Eriko Nurvitadhi’s paper (2016) *Accelerating Binarized Neural Networks: Comparison of FPGA, CPU, GPU, and ASIC* [59] discuses FPGAs vs GPUs on solely on the classification part, not the training of the network. They also compare FPGAs with ASICs.

With deeper nets, more storage capacity is needed. This is a limited resource which increases the need for optimization. For example, the weights between two fully connected layers with 5000 neurons each would result in 100 MB using 32-bit representations for each weigh. This would be reduced to 3.125 MB if we used a binary representation for the weights.

The activation functions in a BNN produces a 1-bit output, 1 or -1. Doing a multiplication of a 1-bit value with another 1-bit value does not require multiplication. This can be done simply with a xnor gate. The truth table is shown in Figure 4.1.



XNOR		
Inputs		Output
A	B	C
0	0	1
0	1	0
1	0	0
1	1	1

Figure 4.1: XNOR truth table

"Our evaluation results show that the proposed accelerator can deliver orders of magnitude improvements in performance and performance/watt over well optimized software on CPU and GPU." Eriko Nurvitadhi's paper (2016) [59].

When training BNN, only the activations and the weights are binary. The gradients of the weights are accumulated with higher precision. *"SGD explores the space of parameters in small and noisy steps, and that noise is averaged out by the stochastic gradient contributions accumulated in each weight. Therefore, it is important to keep sufficient resolution for these accumulators, which at first glance suggests that high precision is absolutely required."* Matthieu Courbariaux (2016) [14].

Intel's paper on acceleration of NN using binarization [57] uses higher precision for the accumulated gradients. *"Real valued gradients are required for SGD to work. The weights are stored in real valued accumulators and are binarized in each iteration for forward propagation and gradient computations."*

It's worth mentioning that some of the articles on the topic of FPGAs vs GPUs easily can be perceived as slightly biased against using one or the other platform. But in the near future, as the NN evolve, both the FPGA and be GPU will most likely coexist and outperform each other in different areas. There are various needs in form of accuracy, speed and power consumption. GPUs will probably be used in high precision, and dense NNs, while the FPGA is more suited for other types of irregularities and applications requiring lower power consumption.

4.2 Well known Neural Network Architectures

There is a vast amount of literature on the different NN architectures from the 90s until today. This section will only give a brief overview of some of the different architectures, and look at the different techniques introduces by each network for solving various problems.

Many of the most famous neural networks have participated in the *ImageNet* challenge, where the goal is to achieve the highest accuracy in image classification. Figure 4.2 from Alfredo Canziani & Eugenio Culurciello (2017) [8] gives an overview of the different architectures. The challenge is quite popular and can be used for comparison of accuracy between different NN architectures, but does not take any other measures, like speed or size.

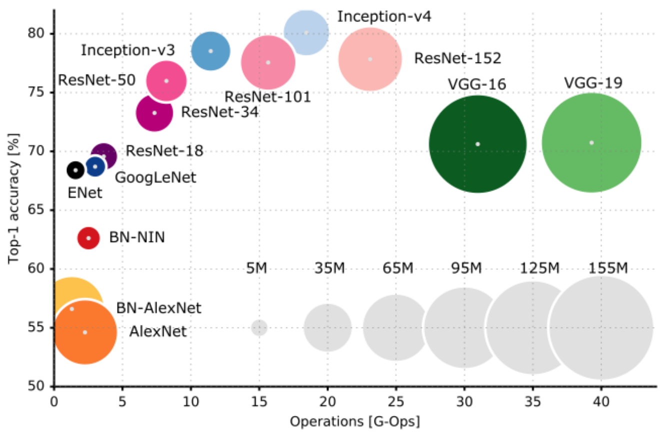


Figure 4.2: *”Top1 vs. operations, size parameters. Top-1 one-crop accuracy versus amount of operations required for a single forward pass. The size of the blobs is proportional to the number of network parameters; a legend is reported in the bottom right corner, spanning from 5106 to 155106 params. Both these figures share the same y-axis, and the grey dots highlight the centre of the blobs.”* Canziani, A., Paszke, A., Culurciello, E. (2016) [8]

4.2.1 LeNet5

The *LeNet5* [49] was one of the first convolutional neural networks, being much of the inspiration for the next generations of CNN to come. It did not use all the pixels as inputs to a large fully connected multi-layer network, but took advantage of the fact that pixels in an image are more closely related to neighboring pixels than pixels further away. The network consists of 7 layers and the architecture is shown in Figure 4.3. The activation functions used to introduce non-linearity were sigmoid or tanh.

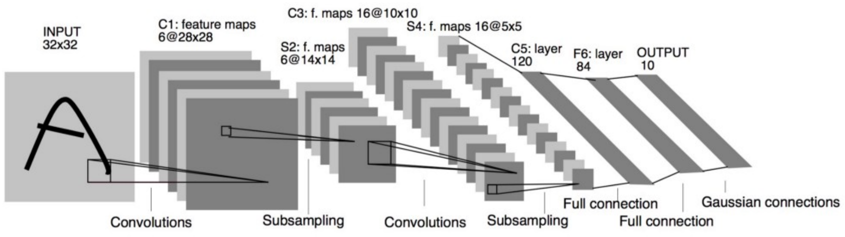


Figure 4.3: The architecture of LeNet5. Lecun, Y. (1998) [49]

4.2.2 AlexNet

AlexNet [45] is a bigger CNN than *LeNet5*. It reduced training time running on multiple GPUs. The activation function used was ReLU making it more effective. The pooling layers used max pooling and were overlapping, meaning that the pixels in a pooling unit overlaps with the previous adjacent pooling unit. *AlextNet* addresses the overfitting problem in several ways. It uses *data augmentation* utilizing both color shift and random cropping, in addition of using dropouts to reduce overfitting. The architecture of *AlexNet* is shown in Figure 4.4.

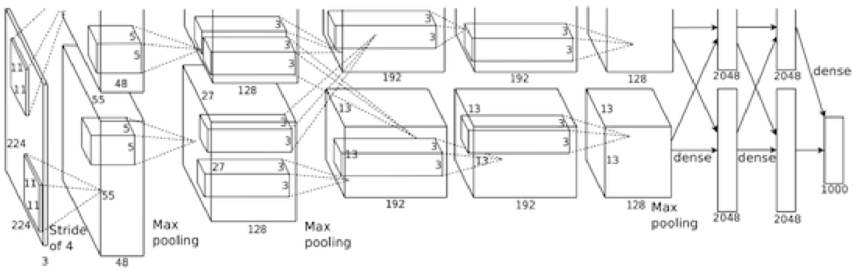


Figure 4.4: The architecture of AlexNet. Krizhevsky, A. (2012) [45]

4.2.3 VGG

VGG [70] is deeper and more accurate than AlexNet, but has a lot of parameters and takes a long time to train as illustrated by the size of the circle in Figure 4.2. It uses smaller filter sizes of 3x3, making the network more efficient. Stacking multiple smaller filters gives larger receptive field, similar to using a bigger filter. This reduces the number of parameters needed. Also using ReLU functions between the layers introduce more non linearities making the features more expressive.

4.2.4 GoogleNet

GoogleNet [75] made a deeper and more effective neural network. It introduces 1x1 convolutional layers and the *inception module*.

1x1 Convolutions

1x1 convolutions does mainly two things: it introduces more non-linearity to the network, and it reduces the dimension. Using a 1x1 on an input of 1 channel doesn't make much sense (for example 7x7x1), it would only result in a scaling of all the number in the input. Using more input channels, like 7x7x10, each pixel of the output would be a weighted sum from that same pixel position in the input from each channel. The output channel depth will be the same as the number of filters.

Inception Module

Instead of using either a 3x3 or a 5x5 for example, the *inception module* basically uses multiple filter sizes and a pooling layer and concatenates the outputs letting the network learn what to use instead of choosing one to begin with. The 1x1 convolutions before the 3x3 and 5x5 reduces the dimension of the input before these larger filters reducing the computational cost. Later, more versions of the inception module were introduced.

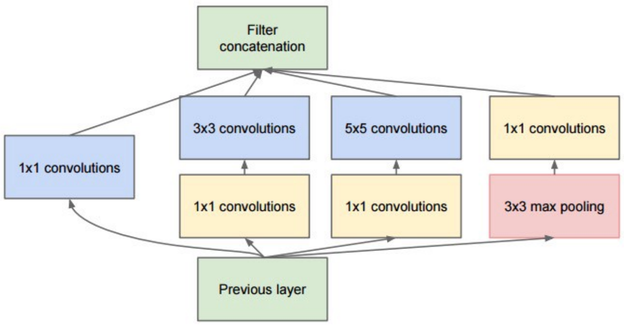


Figure 4.5: The inception module. Szegedy, C. (2014) [75]

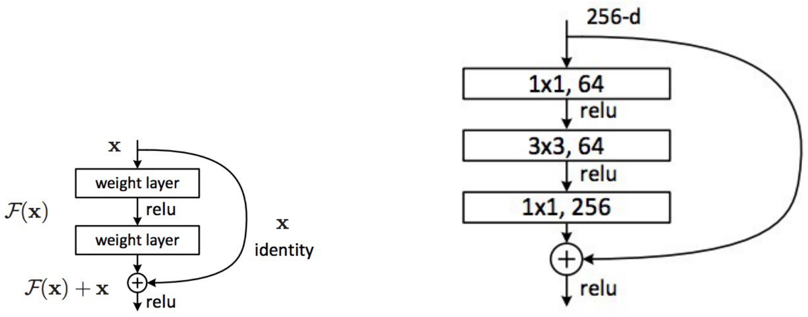
4.2.5 ResNet

ResNet [34] introduces the residual block shown in Figure 4.14(a). This block adds a *shortcut connection*. In plain networks without these shortcuts, a problem with training arises when the networks get deeper. The shortcuts make the optimization solver converge faster. With many layers they also started to use *bottleneck* layers, like shown in Figure 4.14(b). Figure 4.7 illustrates that the plain network struggles to converge when using more layers as the accuracy does not improve. In contrast, the ResNet improves when using a deeper network.

4.2.6 SqueezeNet

SQUEEZENET: ALEXNET-LEVEL ACCURACY WITH 50X FEWER PARAMETERS AND $\leq$ 0.5MB MODEL SIZE [37]. As the title says, SqueezeNet's goal was to achieve a certain point of accuracy with a much smaller model. The paper points out three advantages using fewer parameters; a more efficient distributed training process, less overhead when exporting new models to clients, and feasible FPGA and embedded deployment, due to the traditionally small amounts of on chip memory on the FPGA. To achieve fewer parameters without degrading the accuracy, the paper introduces three design strategies for the NN:

- Strategy 1. Replace 3x3 filters with 1x1 filters
- Strategy 2. Decrease the number of input channels to 3x3 filters. Reducing the inputs to the 3x3 filters are done using squeeze layers



(a) Residual block **(b)** Residual block with bottleneck

Figure 4.6: Residual blocks of ResNet. He, K. (2015) [34]

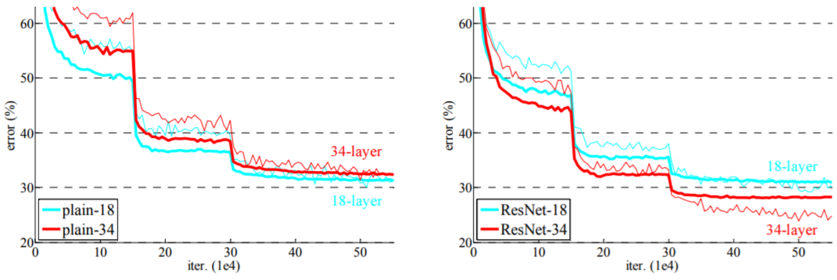


Figure 4.7: Error in ResNet vs plain network. Szegedy, C. (2014) [75]

- Strategy 3. Downsample late in the network so that convolution layers have large activation maps. This strategy is used to achieve high accuracy

SqueezeNet introduces the *fire module* shown in Figure 4.8. There are three hyperparameters in the fire module. The number of filters of 1x1 squeeze, 1x1 expand and 3x3 expand. The use of 1x1 reflects on strategy 1 and the squeeze also limits the number of inputs to the 3x3 filters as proposed in strategy 2. Strategy 3 are shown by the somewhat late positioning of the pooling layers in Figure 4.9.

4.2.7 ENet

ENet (efficient neural network) [62] was designed for *Semantic Segmentation* running on low power mobile devices. It is very small and still precise. The building blocks introduced by ENet is shown in Figure 4.10. Like many of the other more efficient networks, ENet factorizes filters, meaning decomposing bigger filters into multiple smaller ones. It downsamples early to reduce the module size, and it used PReLU (Parameterized ReLU) as activation function. This uses more parameters, but enables learning of the negative slope of the activation function. It also used *Dilated convolutions*. This adds one parameter to

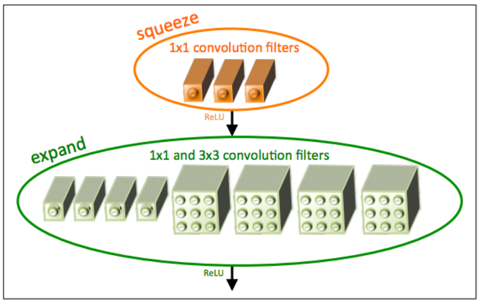


Figure 4.8: Fire Module. Iandola, F. (2016) [37]

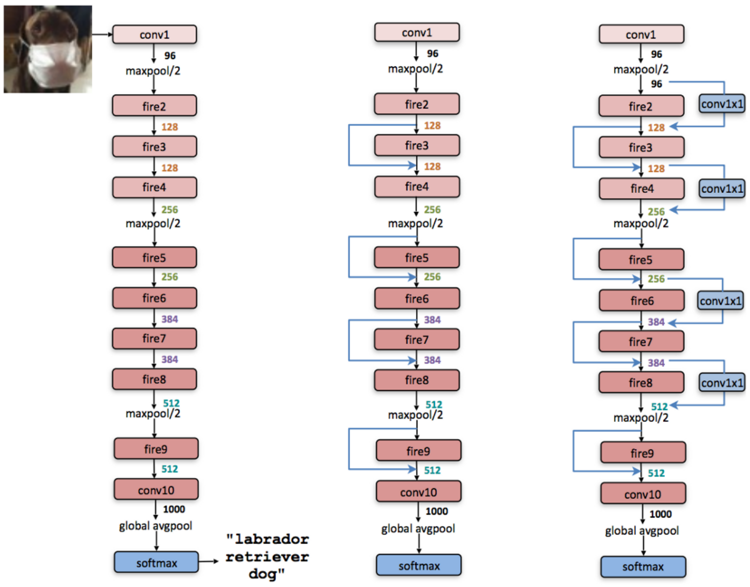


Figure 4.9: *Macroarchitectural view of our SqueezeNet architecture. Left: SqueezeNet; Middle: SqueezeNet with simple bypass; Right: SqueezeNet with complex bypass* Iandola, F (2016) [37]

the convolution. From the paper from Fisher Yu (2016) [83]: *”The dilated convolution operator can apply the same filter at different ranges using different dilation factors.”* It is basically an upsampling layer, which gives a exponential growth in the receptive field with the number of parameters growing linearly. Normal dropout used against overfitting showed little success, so *spatial dropout* was used instead. It drops out whole branches instead of random subset of neurons. Jonathan Tompson (2015) [77]: *”extend the dropout value across the entire feature map. Therefore, adjacent pixels in the dropped-out feature map are either all 0 (dropped-out) or all active”*

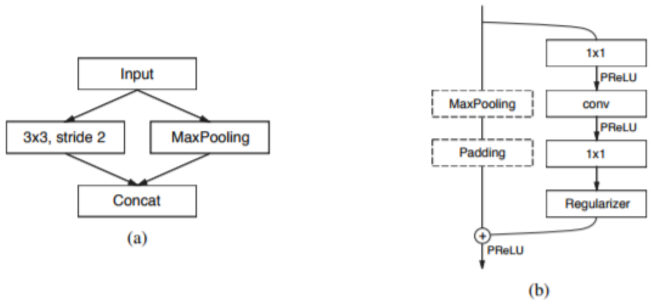


Figure 4.10: Blocks of ENet. Paszke, A. (2016) [62]

4.2.8 Xception

The *Xception* architecture is rather simple. It builds on inception, residual blocks. In the paper from Franois Chollet (2017) [11]: *"In short, the Xception architecture is a linear stack of depthwise separable convolution layers with residual connections."* The architecture is shown in Figure 4.11.

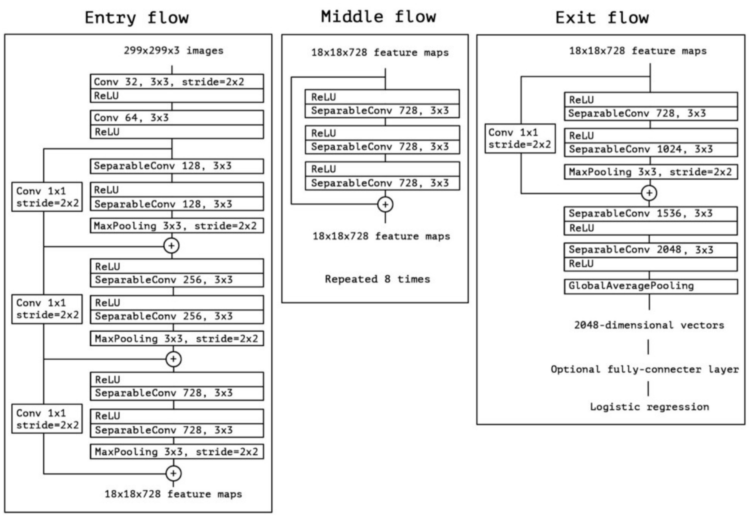


Figure 4.11: Architecture of Xception. Chollet, F. (2016) [11]

4.3 FPGA Implementations

Most implementations done on FPGAs focus on the inference part, i.e. not the training phase. This makes sense because it is in the inference phase the FPGA really shine providing very low latency and better power efficiency. This section will look at both implementation of training and inference but has most focus on the latter.

4.3.1 F-CNN: An FPGA-based Framework for Training Convolutional Neural Networks

F-CNN [84] presents a framework for training convolution neural networks. I.e. not accelerating the classification of the CNNs as most FPGA accelerators, but the training process. This has a more complicated workflow, requiring a more flexible network. Classification uses forward propagation on a pre-trained network, while training does forward propagation, error propagation and then updates the weights. F-CNN has been used to implement *AlexNet* and *LeNet5* on an FPGA showing higher performance than a CPU, and a more energy efficient implementation than a GPU.

The F-CNN paper proposes some design principles:

- Modularity: Since the FPGA has far from sufficient resources to implement the whole training process, a modular design is used. The training process is partitioned, and parameterized modules implements the three computational layers of the network: the convolution, pooling and MLP.
- All the modules support a unified data path. The data transferred between the blocks are 4-dimensional tensors.
- It uses runtime reconfiguration. There are not enough resources to pre-configure all modules before execution.

The architecture of F-CNN is shown in Figure 4.12. The *data controller* divides the input training data into smaller mini batches and loads them to DRAM.

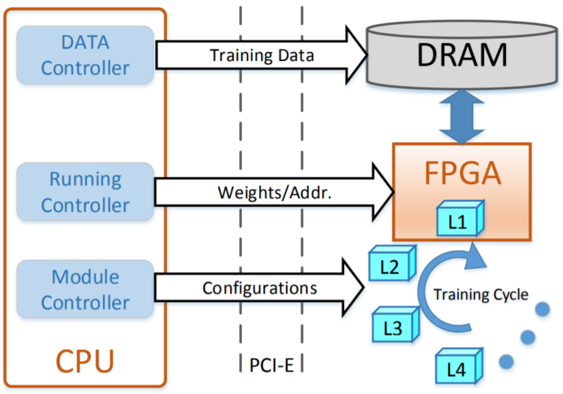


Figure 4.12: F-CNN architecture. Zhao, W. (2016) [84]

Which computational modules to use are controlled from the *module controller*. It specifies the configuration of the modules dependent of the training cycle as seen in Figure 4.12. There are implemented modules for each of the layers in the network.

The *running controller* are responsible to start the module to do the actual computation.

The paper sets up four steps for a training cycle:

1. Reconfigure a module into a FPGA card
2. Prepare data in DRAM
3. Call the module to do the computation, with parameters and weights
4. Read back the results and update weights in CPU (for back-propagation modules) and go to step 1

The basic architecture of the modules in F-CNN is shown in Figure 4.13. There can be one or more kernels running in parallel in each module. Four different kernels: forward convolutional kernel, backward convolutional kernel, pooling kernel, and MLP kernel.

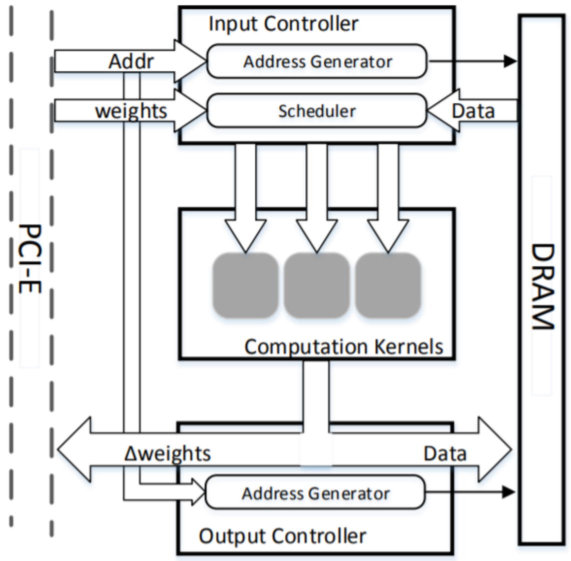


Figure 4.13: Model architecture in F-CNN. Zhao, W. (2016) [84]

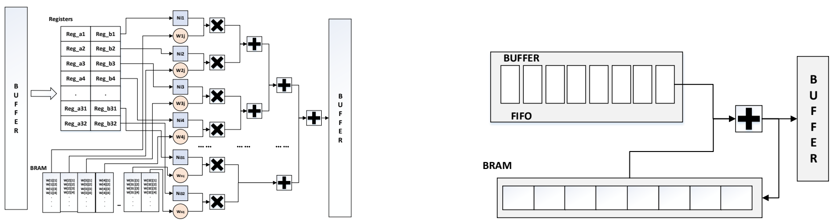
The hardware programming for this project was designed writing high-level code using the MaxCompiler [53] and the software results are obtained using the Caffe framework [41]. It takes some time to reconfigure the FPGA, so two FPGAs are used. One is executing while the other one is reconfigured. The results show that the performance using the FPGA is almost as good as the GPU. 4,3x speed up vs 4,7x on the GPU and the FPGA is 7,5 times more energy efficient.

4.3.2 DLAU: A Scalable Deep Learning Accelerator Unit on FPGA

DLAU [81] is a scalable and flexible deep learning accelerator implemented in an FPGA. It consists of three main blocks that is pipelined; the TMMU, the PSAU and the AFAU. The architecture has 3 key elements;

- Each element uses FIFO buffers to prevent loss of data in case of inconsistent data flow between units.
- It uses *tiled techniques*. It splits the input data into smaller pieces that is cached on the chip. The hardware is time-shared between the partitions of the data.
- There are streaming interfaces between the processing units transferring data.

The TMMA (Tiles Matrix Multiplication Unit) is the main block doing the multiplications and the additions like shown in Figure **??**. In the illustration, the tile size is set to 32, doing 32 multiplications for each tile. This is connected to the PSAU (Part Sum Accumulation like shown in Figure **??** before the data goes though the AFAU (Activation Function Acceleration Unit) which performs a sigmoid activation.



(a) TMMU schematic **(b)** PSAU schematic

Figure 4.14: DLAU schematics. Wang, C. (2016) [81]

The whole architecture is shown in Figure 4.15. As the paper illustrates; the speed of the network is proportional with the tile size as expected, since more multiplications would be executed in parallel. It also includes some discussion on resources and power utilization. There is not much focus on training, or other topics. The core of the paper is the acceleration unit itself, and its scalability.

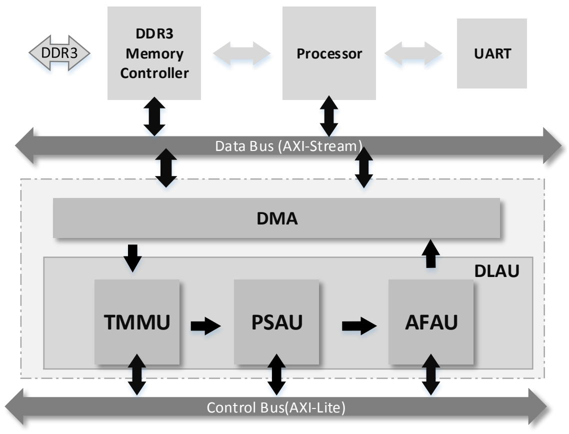


Figure 4.15: DLAU architecture. Wang, C. (2016) [81]

4.3.3 The implementation of a Deep Recurrent Neural Network Language Model on a Xilinx FPGA

Yufeng Hao (2017) [32] focuses on implementing an embedded deep recurrent neural network (DRNN) used for NLP (Natural Language Processing). It uses Python, with the Theano deep learning framework [76], for training and verification of the DRNN. To deal with the vanishing gradient problem it used LSTM blocks. The DRNN language model's program flow is shown in Figure 4.16.

The architecture of the DRNN is shown in Figure 4.17(a) and the main computational block that's implemented in logic is shown in Figure 4.17(b).

From the pynq website (`http://www.pynq.io/`): *"PYNQ is an open-source project from Xilinx that makes it easy to design embedded systems with Xilinx Zynq All Programmable Systems on Chips (APSoCs)."* This project utilizes high level of abstraction speeding up the process implementing the DRNN. The trained model on PYNQ is deployed through Jupyter. It shows that a pre-trained model on a CPU or GPU can be deployed to an FPGA SoC using high-level system tools.

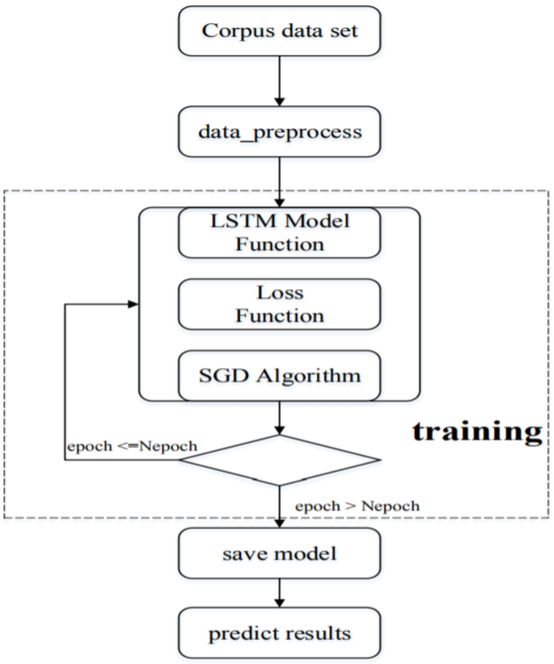
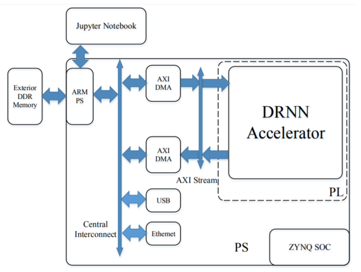
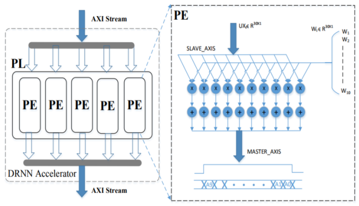


Figure 4.16: Program flow of the DRNN LM. Hao, Y. (2017) [32]



(a) Block diagram of the accelerator overlay data path diagram



(b) DRNN accelerator

Figure 4.17: Architecture of DRNN. Hao, Y (2017) [32]

4.3.4 FINN: A Framework for Fast, Scalable Binarized Neural Network Inference

FINN is an embedded neural network performing millions of classifications per second with very low latency. From the paper, Yaman Umuroglu* (2016), [78] *"On a ZC706 embedded FPGA platform drawing less than 25 W total system power, we demonstrate up to 12.3 million image classifications per second with 0.31 s latency on the MNIST dataset with 95.8% accuracy, and 21906 image classifications per second with 283 s latency on the CIFAR-10 and SVHN datasets with respectively 80.1% and 94.9% accuracy"*

The paper discusses, among other things, the tradeoffs between accuracy and network size. Interestingly the difference in accuracy between low precision and floating-point networks is reduced in larger networks like seen in Table 4.1. The lack in accuracy can be compensated, by making the network larger, and the speedup is greater than the increase in parameters. That indicates that BNNs, achieving the same accuracy as fixed-point networks, could be faster.

The streaming architecture and scheduling of FINN are shown in Figure 4.18. All the parameters of the NN are stored in on-chip memory, reducing the latency of the network.

For all the BNNs in this paper, the input and output activations and the weights are represented using 1 bit, -1 or +1 like shown in Equation 4.1. They also use batch normalization before the activation function:

$$Sign(x) = \begin{cases} +1 : x \geq 0 \\ -1 : x < 0 \end{cases} \tag{4.1}$$

The paper explains some optimizing techniques:

- *Popcount for Accumulation*: only counting value since its only +1 or -1.
- *Batch-norm activation as threshold*: the same output can be computed via thresholding.
- *Boolean OR for Max-pooling*: Max pooling after activation (on binary values)

Three topologies are presented:

Neurons/layer	Binary err. (%)	Float err. (%)	Params	Ops/frame
128	6.58	2.70	134,794	268,800
256	4.17	1.78	335,114	668,672
512	2.31	1.25	932,362	1,861,632
1024	1.60	1.13	2,913,290	5,820,416
2048	1.32	0.97	10,020,874	20,029,440
4096	1.17	0.91	36,818,954	73,613,312

Table 4.1: Accuracy results - BNN vs floating point NN. Umuroglu, Y. (2017) [78]

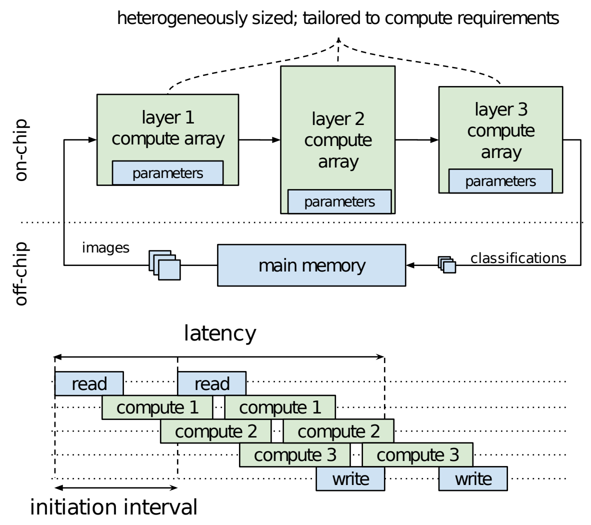


Figure 4.18: FINN streaming architecture and scheduling. Umuroglu, Y. (2016) [78]

Name	Thr.put (FPS)	Latency (us)	LUT	BRAM	P_{chip} (W)	P_{wall} (W)
SFC-max	12361 k	0.31	91131	4.5	7.3	21.2
LFC-max	1561 k	2.44	82988	396	8.8	22.6
CNV-max	21.9 k	283	46253	186	3.6	11.7
SFC-fix	12.2 k	240	5155	16	0.4	8.1
LFC-fix	12.2 k	282	5636	114.5	0.8	7.9
CNV-fix	11.6 k	550	29274	152.5	2.3	10

Table 4.2: Results from different topologies. Umuroglu, Y. (2016) [78]

- SFC: Three fully connected layer with 256 neurons in each layer. Used to classify the MNIST data set (28 x 28 handwritten digits)
- LFC: Three fully connected layer with 1024 neurons in each layer. Used to classify the MNIST data set (28 x 28 handwritten digits)
- CNV: It is a convolutional NN. From the paper: *”It contains a succession of (3x3 convolution, 3x3 convolution, 2x2 maxpool) layers repeated three times with 64-128-256 channels, followed by two fully connected layers of 512 neurons each.”* It is used to classify CIFAR-10 (32 32 color images in 10 categories) and SVHN (32 32 images of Street View House Numbers).

Results from the different topologies are shown in Table 4.2. The *max* and *fit* are either maximum performance or a fixed FPS.

The FINN framework outperforms the other topologies reviewed here and others [2] [19] [35] [61].

	ALSTM	RBM	SAE	Auto-LSTM
RMSE	0.011562	0.035586	0.030211	0.022520

Table 4.3: Power consumption: one-step performance. Hsu, D. (2017) [36]

4.4 Neural Networks in time series forecasting

Diverse types of network topologies have been used for time series forecasting. Takashi Kuremoto [46] used a deep belief network (DBN) in 2014 and later both CNNs and LSTMs have been used. This section will take a brief look at some of the different topologies used for time series forecasting.

4.4.1 Time Series Forecasting Using LSTMs

Due to the memory of LSTMs, it has the ability to use long time dependencies when forecasting time series. Daniel Hsu paper (2017) [36]: *Time Series Forecasting Based on Augmented Long Short-Term Memory*, presents an augmented long short-term memory (A-LSTM). This is a combination of an Auto Encoder and the LSTM. The inputs are encoded to latent variables. The latent variables are dependent on both the input and the hidden state of the LSTM. The predicted output is decoded from the hidden state of the LSTM and these latent variables. This idea comes from Felix A. Gers's paper (2002) [26]. It shows that an MPL outperformed the LSTM on prediction only using some of the recent inputs. *"LSTM learned to tune into the fundamental oscillation of each series but was unable to accurately follow the signal. The MLP, on the other hand, was able to capture some aspects of the chaotic behavior."* The paper also suggests using a hybrid of the two.

Daniel Hsu shows results using simulated data and real word data from household electric power consumption. The results from the A-LSTM are compared against: Restricted Boltzman Machine (RBM) [46], Stacked Denoising Auto-Encoders (SDAEs) [66] and autoencoder stacked on LSTM (Auto-LSTM) [25].

The SDAE is a bit like the RBM. In the unsupervised part of the learning, some noise is added to the inputs. Each layer then need to reconstruct a *clean* version of the inputs. Doing this attempt to extract higher level features from the input data, generalizing it in a better way. The Auto-LSTM is basically just as the name suggest an autoencoder stacked on a LSTM.

Figure 4.19 shows the one-step performance test of power consumption data. 1000 data points are used for training and about 6000 for testing. Table 4.3 show the root-mean-squared error (RMSE) for the architectures mentioned above.

4.4.2 Time Series Forecasting Using CNNs

Anastasia Borovykh's (2017) paper [5], *Conditional Time Series Forecasting with Convolutional Neural Networks* is based on the WaveNet architecture [79] which is used to

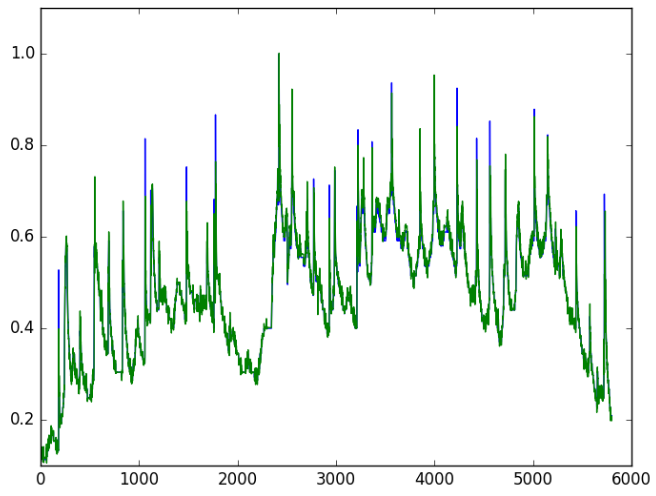


Figure 4.19: Predicted data (green). Actual values (blue). Hsu, D. (2017) [36]

generate audio from text. It uses dilated convolution to enable the network to learn connections between data points that are further away from each other. In the dilated convolution, the filter skips elements in the input vector. Figure 4.20 show a 1, 2 and 4-dilated convolution.

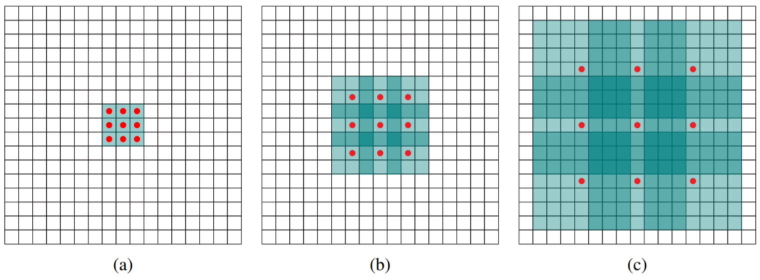


Figure 4.20: Dilated convolution. Yu, F. (2015) [83]

The network uses residual connections and the activation function used is ReLU. The architecture is shown in Figure 4.21. It uses conditional time series to help forecast the input time series. There can be multiple conditions, and the *parameterized skip connection* learns to skip connections that are not important for the forecast. According to the paper this network shows promising results in comparison to LSTMs, that is widely used on time series data.

Interestingly, Zhicheng Cui (2016) [16] explores the opportunity to use multiple trans-

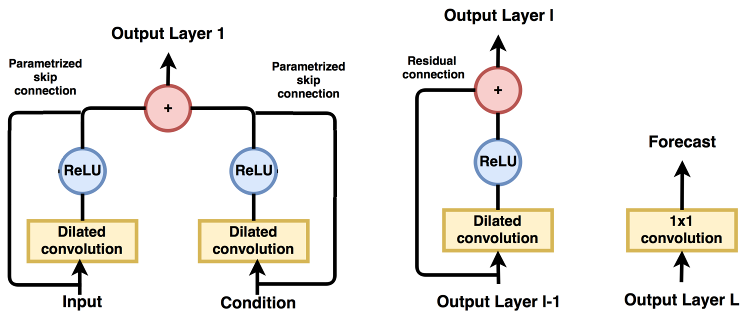


Figure 4.21: *"The network structure. In the first layer (L) the input and condition (with the zero padding) are convolved, passed through the non-linearity and summed with the parametrized skip connections. The result from this first layer is the input in the subsequent dilated convolution layer with a residual connection from the input to the output of the convolution. This is repeated for the other layers, until we obtain the output from layer L (M). This output is passed through a 11 convolution, resulting in the final output: the forecasted time series (R)."* Yu, F., Koltun, V. (2015) [83]

formation like time and frequency domain of the input data for time series classifications. This showed better results in most cases compared to using an original CNN with the same size. The architecture of the *Multi-Scale CNN* is shown in Figure 4.22.

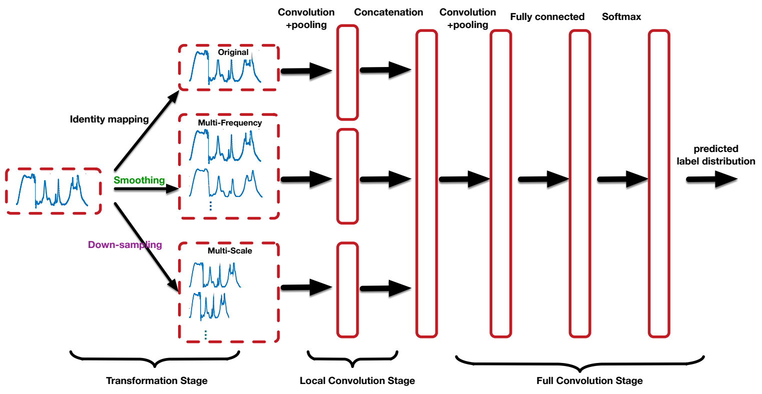


Figure 4.22: Architecture of multi-scale CNN. Cui, Z. (2016) [16]

4.4.3 Combining CNNs and RNNs

Wolfgang Groß's paper from 2017 [30] introduces something they call *Space-Time Convolutional and Recurrent Neural Network* (STaR). It is a combination of a CNN and a RNN or LSTM blocks. One of the key features in this paper is that it interprets the input as a time-space matrix, i.e. each channel of the multivariate input has its own column and the discrete time increases from top to bottom. The architecture of STaR and the arrangement of the input matrix is shown in Figure 4.23. This arrangement of the input matrix only makes sense if the neighboring input channels are related to each other. Combining the CNN with a RNN like this enables the network to learn features with longer time dependencies.

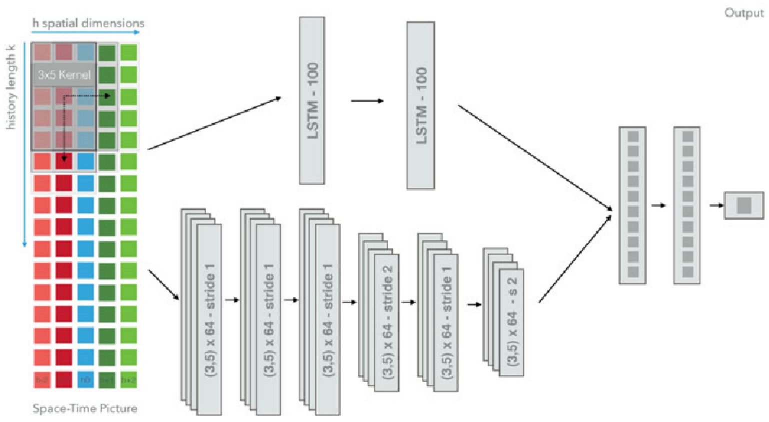


Figure 4.23: Space time arrangement and STaR architecture. Groß, W. (2017) [30]

The STaR architecture is tested against other architectures on the European power exchange trading power in form of contracts. There are three output classes from each network: the price stays the same within some margin, it rises, or it falls. Random selection would get approximately 33%. Table 4.4 shows the results from the test done in the paper. As we see the STaR network yields the best performance.

Time CNN	ST-CNN 6L-3Ch	ST-CNN 6L-5Ch	ST-CNN 9L-5Ch	ST-CNN 6L-7Ch
38.6 %	41.9 %	41.8 %	43.9 %	44.3 %
NN	RNN 1L	RNN 2L	STaR NN	STaR Linear
42.1 %	45.3 %	44.3 %	48.3 %	43.2 %

Table 4.4: Model comparison. Groß, W. (2017) [30]

Chapter 5

Evaluation of The Literature Review

This literature review had the purpose of acquiring general knowledge about deep learning, and explore related work on neural networks, FPGA implementations and NNs used for time series forecasting.

The review's background theory chapters are influenced by the writer's prior knowledge on the topic as it starts from the basics. It covers basic elements about deep learning and goes though some different architectures and optimization techniques. This relatively broad approach touches upon several important subjects, but at the expense of going very deep into any specific subject. It covers enough for the writer to start the implementation.

When discussing FPGA implementations and NN used for time series forecasting, the writer reviews a few chosen academic papers instead looking very broadly on the subject. In the case of deep learning it makes sense to use fewer, and stick to relatively new papers, rather than including older research, as the field is evolving very fast. The main difference between these implementations, and this project's problem, is the aspect of the highly random dataset. This makes it much harder to find clear patterns and structures in the data. Methods of pre-processing, feature selection (like explored by Jundong Li (2016) [50]) and handling of highly random data is not included in the review. In hindsight of testing, this should have been included and given more focus.

Part III

Method

Chapter 6

Functional Specification

This chapter will shortly describe the functional specification of the software that has been developed as a part of this project, including what tasks it should perform and the success criteria of the project.

6.1 System Description

The system consists of two sub systems: the pre-processing and application system shown in Figures 6.1 and 6.2, respectively.

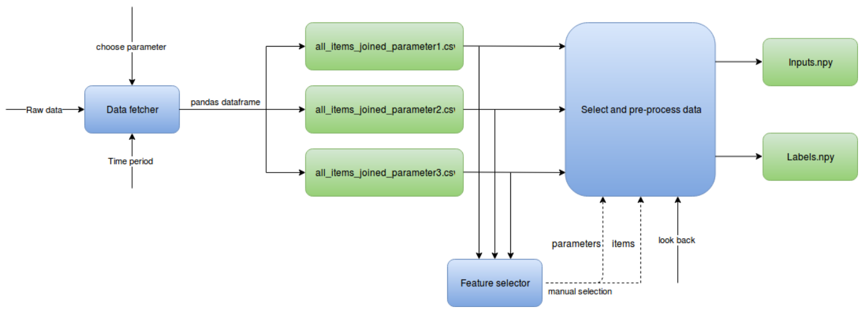


Figure 6.1: Pre-processing system

The pre-processing system in Figure 6.1 inputs the raw data. There is one file for each *item*, and about 500 items in total in the ”mysteryset”. These contain a few parameters each. All the parameters from these items makes up the total number of available features. The *Data fetcher* combines all the features represented by each parameter into big pandas *Dataframes* and saves them as .csv files. I.e. each parameter from all the items is stored in their own csv file. The structures of the data from separate files for each item into one

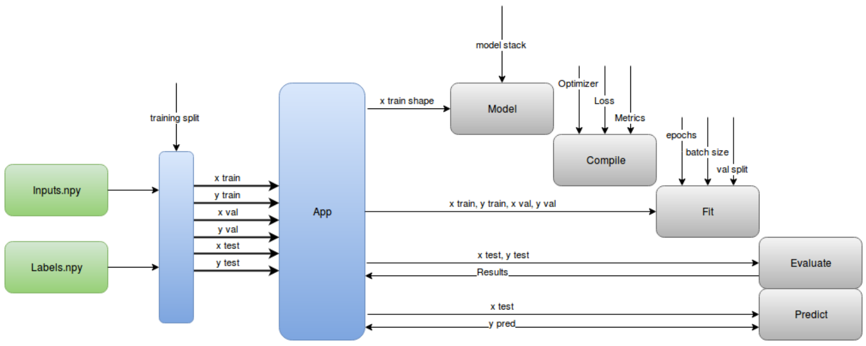


Figure 6.2: Application system

csv file for each chosen parameter is shown in Figure 6.3. The block diagram of the *Data fetcher* is shown in Figure 6.4. The pickled list of item names is manually put together from a much larger selection of items.

The application system in Figure 6.2 inputs the numpy arrays of inputs and labels. These are split into a training-, a validation- and a test-set. The model is built, compiled and fitted to the training data. Then the results are visualized and evaluated.

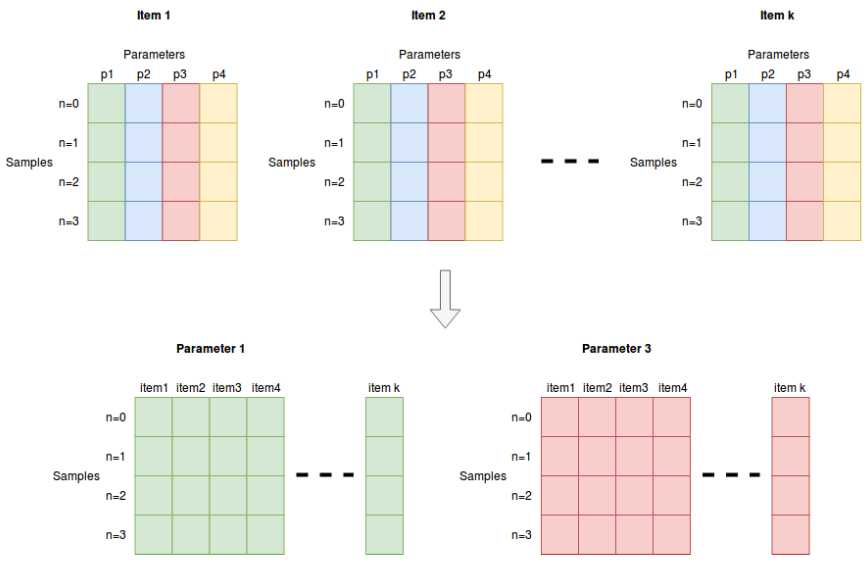


Figure 6.3: Illustration of the conversion from items to files for each parameter. This figure shows an example using only 4 samples and 2 selected parameters

The pre-processing system does some features extraction, i.e. choosing what items to keep, in each parameter file. All the selected features are scaled prior to saving the data as

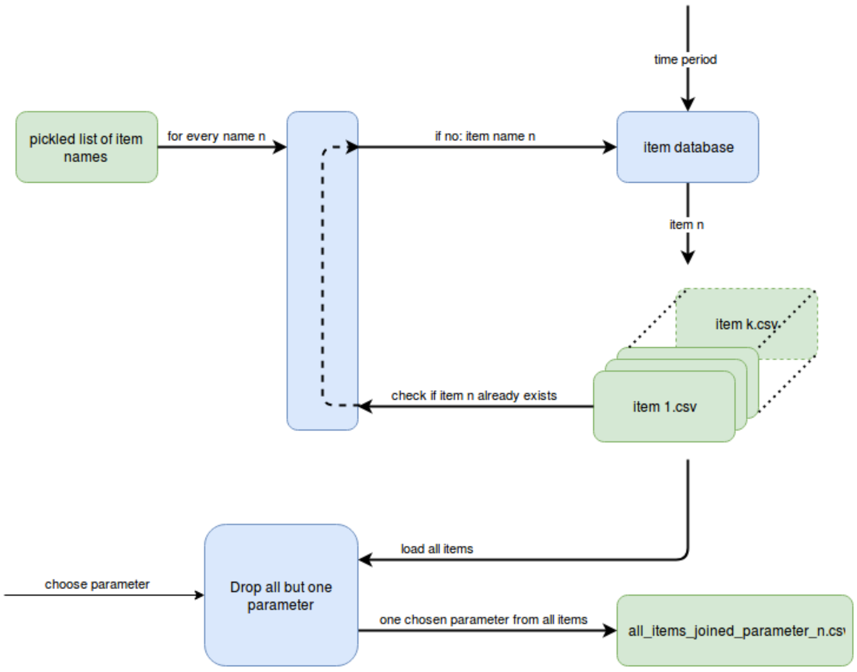

time period
pickled list of item names
for every name n
if no: item name n
item database
item n
item k.csv
item 1.csv
check if item n already exists
load all items
choose parameter
Drop all but one parameter
one chosen parameter from all items
all_items_joined_parameter_n.csv

Figure 6.4: Block diagram of the data fetcher

one set of *numpy* arrays of inputs and labels. There will thus be one set of inputs/features and outputs/labels files. A more detailed description of the feature selection is presented in Section 7.2.4.

6.2 Specification

The specification of the pre-processing is easily summarized by three simple bullet points:

- Inputs raw files. One for each feature containing a few parameters each
- Feature selection and scaling
- Outputs one set of inputs and labels as *numpy* arrays.

The neural networks should be small in size, and kept shallow minimizing the latency. Their specification is shown in Table 6.1. Only the common specification for all the different topologies are included here. This common specification is used to ease the comparison of performance between the different architectures. Other specifics like: number of epochs, batch size and optimizers might vary for the different topologies and are found mostly by trial and error in the design chapter.

input shape	: look back $\cdot$ number of features
output shape	: binary classification
number of features	: < 50
look back	: < 100
number of parameters	: $< 10\,000$
number of layers in depth	: < 10
number of training examples	: > 3000
number of test examples	: > 400
output metrics	: accuracy
baseline accuracy	: > 50 %

Table 6.1: Common NN specification

6.3 Success Criteria

The success criteria are simply to create a model that finds a generalization of the training data that achieves better than 50% accuracy on a binary classification on the test data, i.e. predicting if the next value of a given feature is rising or falling. This must be showed on multiple runs, using different features as the *classification feature*.

Chapter 7

Tools & Data Preparation

This chapter will briefly describe the software framework chosen for the neural network implementation, and some of the pre-processing done prior to fitting the data to the model.

7.1 Keras - Software Framework

Based on the selection of software frameworks in the literature review, *Keras* is chosen as the software framework for testing the different neural network architectures for predicting time series data. This software framework is chosen because of its high-level user interface, enabling faster creation of different models compared to using a more low-level framework - yet still with a lot of options for tweaking and modifications.

All information not cited elsewhere about the Keras functional API is taken from the official Keras Documentation [12]. Keras uses either *Tensorflow* or *Theano* as backend. In this project Tensorflow is used.

7.1.1 Building Models

Building of models is done using the *Sequential* function. The Sequential model is basically a stack of layers specified by the user. Only the first layer needs to specify the input shape of the data. When combining multiple models, the *Model* method is used.

Core Layers

A selection of the core layers in Keras taken directly from the documentation [12]:

- *Dense: Just your regular densely-connected NN layer.*
- *Activation: Applies an activation function to an output.*
- *Dropout: Applies Dropout to the input.*

- *Flatten: Flattens the input. Does not affect the batch size.* E.g. used between the convolutional layer and the classifier.

- *Input: Input() is used to instantiate a Keras tensor.* The input can be specified as an argument in the first layer.

Other layers like different convolutional layers, recurrent layers, and layers to merge multiple models etc. are found in the documentation.

7.1.2 Compilation

Configuration of the learning process is done with the *compile* method. The loss function and the optimizer must be specified. All the different optimizers and their arguments can be found in the Keras documentation. One example of categorical classification using Nesterov Adam optimizer with 0.002 learning rate is shown below.

```
model.compile(optimizer=Nadam(lr=0.002),
              loss='categorical_crossentropy',
              metrics=['accuracy'])
```

7.1.3 Training

The model is trained using the *fit* method. It uses *numpy* arrays for inputs and labels. The number of epochs and the batch size are specified here. There are also many other arguments that can be set. One example is shown below. The data is split into training and validation data. The validation data is not used for updating the weights of the network, but to keep track of the training process, and to detect overfitting.

```
history = model.fit(inputs_train, labels_train,
          nb_epoch=n_epochs,
          batch_size=batch_size,
          verbose=1,
          validation_data=(inputs_val, lables_val),
          shuffle=True)
```

7.1.4 Evaluation

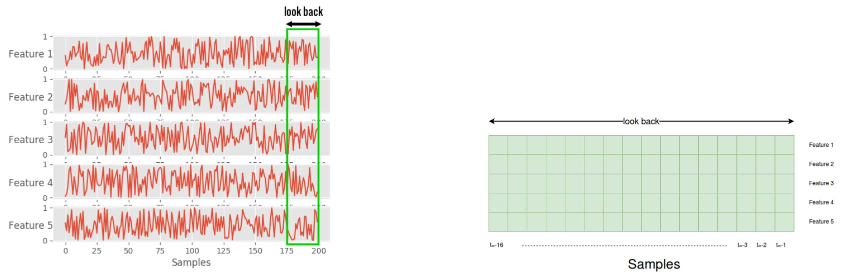
There is an *evaluate* method that returns the loss value and the metrics values. This provides the model accuracy for a given set of inputs and labels. There is also a method that predicts the output based on a given input. This method is used to predict future values of the time series. The evaluation is done on a separate (unseen) test set, which is not part of the training or validation set.

7.2 Pre Processing

7.2.1 Shape of Input Frame

The input frame is inspired by the space time arrangement from Wolfgang Groß's paper (2017) [30] in the literature review.

The input frame is made from a moving window over all the features in the dataset. Each input feature is a 1-dimensional array. The length of the array depends on how much history that is included in each input frame, denoted the *look back*. Combining all the 1-dimensional arrays of features form the input frame for each time stamp. Figure 7.1 illustrate the moving window which results i a new input frame for each sample. The final shape of the input frame x will be on the format: (samples, look back, number of features).



(a) Time series of 5 features

(b) One input frame from 5 features

Figure 7.1: The relationship between the input features and the input frame

In Groß's paper, neighboring features should be related to each other, since the convolution kernel has a specific size in the "spatial" dimension. This is not applied for this implementation. The spatial dimension of the paper, which contains the different features, can be seen as the number of *channels* in e.g. a standard RGB image with three channels. Here the number of features will represent the number of channels, i.e. the number of one dimensional vectors of the input frame.

7.2.2 Feature Scaling

The different features are not necessarily in scale when acquiring the dataset. Normalization or standardization is done per input frame as shown in Equations 7.1 and 7.2

$$x_{\text{standardization}} = \frac{x - \bar{x}}{\sigma} \tag{7.1}$$

where $\bar{x}$ is the mean of x and σ is the standard deviation of x.

$$x_{\text{normalization}} = \frac{x - min(x)}{max(x) - min(x)} \tag{7.2}$$

The whole dataset cannot be scaled before use. When predicting future values, these futures vales cannot be a part of the scaling. Here each input frame of *LOOK_BACK* length is scaled.

```
for i in range(len(dataset)):
    # Take out the input frame from the input features
    in1 = feature_1[i:i + LOOK_BACK]
    in2 = feature_2[i:i + LOOK_BACK]
    in3 = feature_3[i:i + LOOK_BACK]
    in4 = feature_4[i:i + LOOK_BACK]
    in5 = feature_5[i:i + LOOK_BACK]

    if feature_scaling == 'standardization':
        # Standardization for the input frame
        in1 = (np.array(in1) - np.mean(in1)) / np.std(in1)
        in2 = (np.array(in2) - np.mean(in2)) / np.std(in2)
        in3 = (np.array(in3) - np.mean(in3)) / np.std(in3)
        in4 = (np.array(in4) - np.mean(in4)) / np.std(in4)
        in5 = (np.array(in5) - np.mean(in5)) / np.std(in5)

    elif feature_scaling == 'normalization':
        # Normalization for the input frame
        in1 = (np.array(in1)-np.min(in1))/(np.max(in1)-np.min(in1))
        in2 = (np.array(in2)-np.min(in2))/(np.max(in2)-np.min(in2))
        in3 = (np.array(in3)-np.min(in3))/(np.max(in3)-np.min(in3))
        in4 = (np.array(in4)-np.min(in4))/(np.max(in4)-np.min(in4))
        in5 = (np.array(in5)-np.min(in5))/(np.max(in5)-np.min(in5))
```

The dataset is also made stationary, preventing the NN from only extracting long-term trends. The data could for example have an upward or downward going trend, making the NN prone to overdoing that trend by learning to choose only one classifier for all inputs.

7.2.3 Labeling

Supervised learning uses pre-labeled data to train the models. *Labels* are the outputs or the answers from the NN in response to the input data. Machine learning datasets may already be labeled, but not always. If the data is gathered e.g. by sensors they need to be labeled. In time series regression the labels are basically a time shifted version of the input feature that is to be predicted. From the example in Figure 7.1 the label would be sample t=0 for the feature or features in question. In standard classification with e.g. the *MNIST* dataset there are 10 output probabilities, one for each handwritten digit. Each number represents the model predictions for the specific classes. This can also be used for classification of time series data. The classifications could e.g. be rising or falling or any number of classifications within specified ranges.

Labeling e.g. if Feature 1 is rising or falling, is done by looking at the previous and current value like shown below.

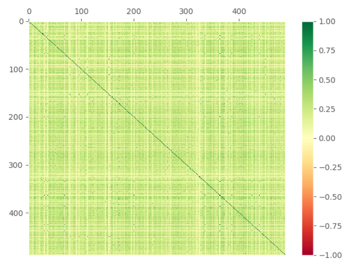
```
    if feature1_t0 < feature1_t-1:
        label = [1, 0]
    else:
        label = [0, 1]
```

7.2.4 Feature Selection

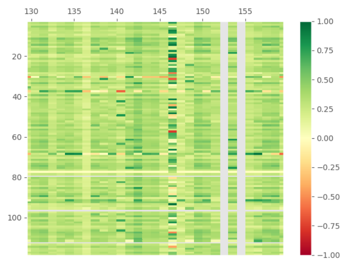
The ”mysteryset” studied in this project, contains a large number of features. All the features can in principle be used, but this would increase computational time a lot, making that approach very inconvenient for this study. Using all the features can also contribute to overfitting. Some of the features have missing data fields. These could be replaced by values from a neighboring feature, but since there are enough features in this case, the features with missing data are not used.

The dataset is highly random, containing little, if any, structure or repetitive patterns. This makes it hard to achieve any significant increase in classification accuracy. The goal here is to see if the NNs can pick up any structures giving a slightly better performance then 50% on a binary classification.

Multiple features can be used predict to the rise or fall of one feature. Correlation between features are used to pick out which features to be included in the dataset. Several different setups with different numbers of features will be tested. Using correlations of lagged time series are also tested. Figure 7.2 shows the correlation matrix of the differences in the time series. The diagonal only shows the features correlated with themselves giving ones on the diagonal. There are clearly some features that are more correlated to the other time series, and others that are almost totally uncorrelated to the other features. Including very correlated inputs would add less new information to the network than uncorrelated ones. Using groups of correlated features can make the network more prone to overfitting since one type of structure might be over represented in the inputs.



(a) All features



(b) Zoomed

Figure 7.2: Correlation matrix of features

7.2.5 Selecting the Number of Samples

The dynamics of the data can change over time. Using too much history can lead to worse results, making the network fit to older data which has little or nothing in common with newer data. On the other side, the NN needs a fair number of examples to make a good generalization of the problem. In this project, there is a finite number of samples available. Initially, and probably throughout the project, all available samples will be used.

7.3 FloydHub

FloydHub [20] is a service used to train deep learning model in the cloud. When the models get larger it takes a lot of time to train on a laptop, therefore FloydHub was used during this project. It is a similar service to *Google ML engine* or *Amazon AWS* and was chosen here for its simplicity. More information about *FloydHub* can be found on their website. Setting up projects and running them on GPUs using *FloydHub* are well documented and not included in this report.

Chapter 8

Design

This chapter will cover the initial design of different network architectures used during this project and some tuning of the hyperparameters.

Finding structures in highly random datasets, like the ”mysteryset”, is a challenging task. All the topologies are therefore first tested using a different dataset which has the same shape of the input frame, but contains much more structure, in order to get the feel of the dynamics of each topology. Then, using this experience, the ”mysteryset” is tested to see if any structures can be extracted.

There is not one specific way of designing a NN. It differs a lot depending on the problem. Much is based on trial and error, but there exists a lot of tips and tricks, like the guidelines for building a neural network explained in the article from *InfoWorld* (2015) [82].

The designs used in this project are kept simple, and are to a considerable extent based on trial and error. A small portion of the testing is included in this chapter. Some rules of thumb were applied; one obviously needs many input examples for each classification and there should not be too many parameters compared to the number of samples. Using too large networks would almost certainly result in overfitting.

The dataset is split into training, validation and a separate test set. It is a finite number of samples in the data set. The validation set is kept relatively small, using most of the data for training. 10% of the data is set aside as a separate test set. The smaller validation set will affect the validation graphs. There will be more variations or noise in the validation set, due to its small size, making it less informative. It is good enough to get some information about the generalization of the data not used in training, and to spot overfitting.

8.1 Neural Network Topologies

Intuitively, RNNs are a good start for predicting sequences of data. Replacing the RNN with LSTM cells would make the network able to learn longer dependencies. As seen in the literature review, CNN are also being used for forecasting. Here dilated convolutions, for example, can be used for enabling the network to exploit longer dependencies. Other papers also reveal good results using a combination of LSTMs and CNN. The following topologies - from the literature review - will be tested:

- MLP
- RNN
- LSTM
- CNN
- Combinations of the topologies listed above

All the networks will use a two neurons fully connected layer with *sigmoid* activation as the classifier. The other activations are ReLU if nothing else is mentioned. To make the comparison fair, each network should have approximately the same number of parameters. This number is kept low and limited to 10000 parameters. This equals 20 kB of data using 16-bit resolution. This is done to keep the parameters storage size low enough to fit inside the on-chip memory of an FPGA at a later stage. Larger networks would also result in overfitting. In addition, none of the networks should be very deep, since this will increase the inference latency.

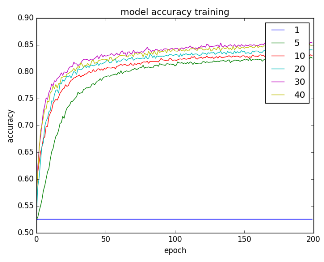
8.1.1 Multi-Layer Perceptron (MLP)

The number of parameters in a fully connected layer is basically the weights and the bias of each neuron, where the number of weights depends on the input size.

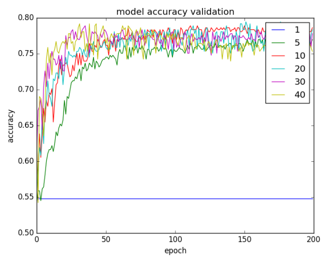
$$\text{parameters} = (\text{input size} + \text{bias}) \cdot \text{n neurons} \tag{8.1}$$

For the first layer the input size would be the number of features multiplied with the *look back* length, i.e. all the samples in the input frame shown in Figure 7.1(b). Visualized with an image analogy; all pixels in the input image are connected to each neuron in the first layer through an unique weight. For the other layers the input size is the number of outputs from the previous layer.

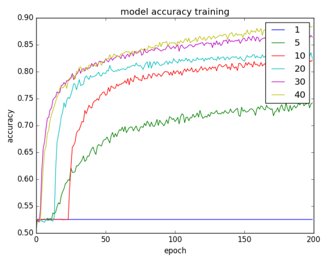
Using wide networks tend to result in overfitting. The idea that substructures are identified in different layers, and that more layers lead to more non-linearity often make networks grow in depth and not in width. Figure 8.1 shows two different depths of neural MLPs with equal number of neurons in each layer. Plot (a) and (b) represent a 2-layer MLP, and (c) and (d) represent a 12-layer MLP with dropout layer of 0.2 in between the layers to avoid overfitting. The deeper network is actually doing slightly worse in accuracy. In both networks it does not seem like adding width to the layer adds anything to the accuracy of the model. The shallow network reacts pretty much the same - increasing the number of neurons. Testing shows overfitting when increasing the number of neurons extensively. The deeper network can avoid overfitting keeping the width of each layer low.



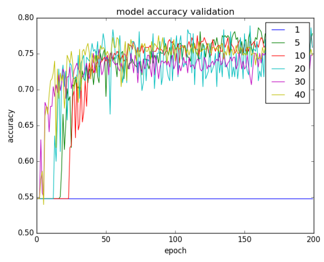
(a) Accuracy Training shallow (2-layer)



(b) Accuracy Validation shallow (2-layer)



(c) Accuracy Training deeper (12-layer)



(d) Accuracy Validation deeper (12-layer)

Figure 8.1: Shallow & deeper MLP with different layer width. The legend shows the number of neurons in each layer of the model.

8.1.2 Recurrent Neural Network (RNN)

In general, the number of parameters of a RNN layer with a given number of cells is shown in Equation 8.2. The simple RNN cell has only one hidden state. That state has a number of weights as inputs. If the RNN cell is the first layer, this equals the number of features in the dataset. In addition, it has a bias and the output state as shown in Figure 8.2.

$$\text{parameters} = \text{hidden states} \cdot \text{cells} \cdot ((\text{weights} + \text{bias}) + \text{outputs}) \tag{8.2}$$

A simple two-layer RNN network with different number of cells are shown in Figure 8.3. There are similar dynamics to the MLP example, where more cells result in more overfitting, and fewer cells achieve higher accuracy.

Figure 8.4 shows a three layered RNN with different dropout values. Overfitting is clearly reduced with increasing dropout rate, but in this case not increasing the overall accuracy. Overfitting is no big problem here, and using higher dropout rates only slows down the learning process. It seems appropriate to leave the dropout rate between 0.1 and 0.5 to achieve good accuracy and generalization of the dataset.

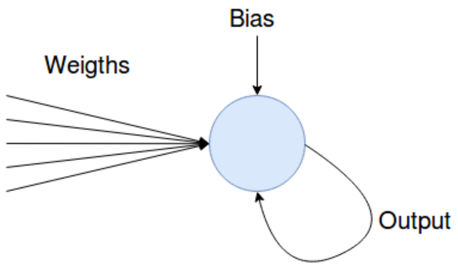
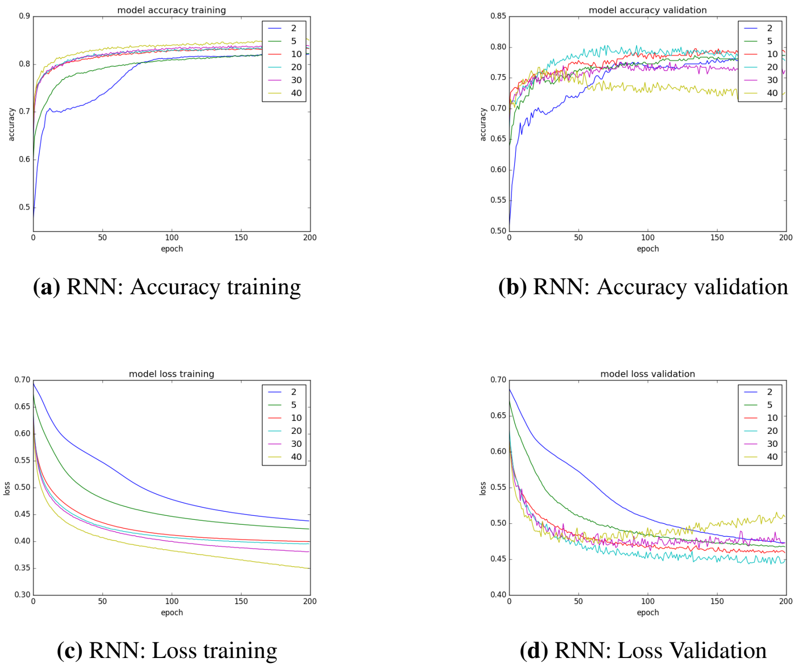


Figure 8.2: Illustration of a hidden state in a RNN layer



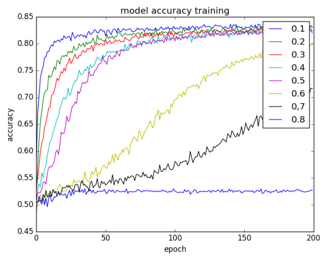
(a) RNN: Accuracy training

(b) RNN: Accuracy validation

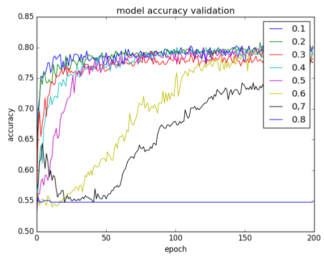
(c) RNN: Loss training

(d) RNN: Loss Validation

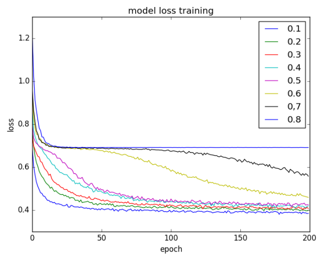
Figure 8.3: Two-layer RNN. The legend shows the number of cells used in each layer.



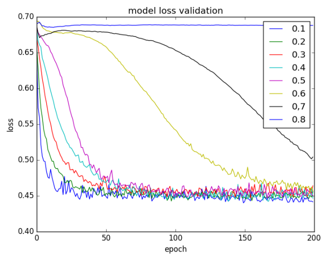
(a) RNN: Accuracy training



(b) RNN: Accuracy validation



(c) RNN: Loss training



(d) RNN: Loss Validation

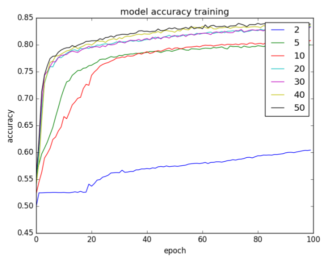
Figure 8.4: Three layer RNN with different dropout rates. The legend shows the dropout rates between each RNN layer.

8.1.3 Long Short-Term Memory (LSTM)

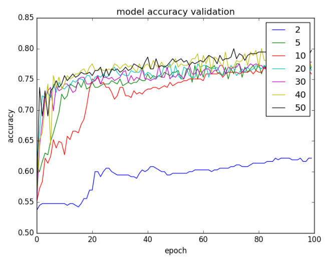
The number of parameters in a LSTM layer is similar to the RNN. It has four hidden states. The three gates and the cell, which makes one LSTM unit use four times as many parameters compared to a RNN as shown in Equation 8.3

$$\text{parameters} = \text{hidden states} \cdot \text{units} \cdot ((\text{weights} + \text{bias}) + \text{outputs}) \tag{8.3}$$

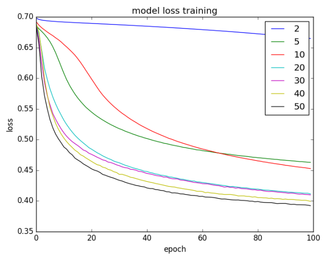
Figure 8.5 show a one-layer LSTM with different number of units. The LSTM seems less prone to overfitting, achieving higher accuracy with more units.



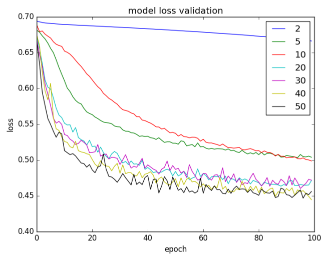
(a) LSTM: Accuracy training



(b) LSTM: Accuracy validation



(c) LSTM: Loss training



(d) LSTM: Loss Validation

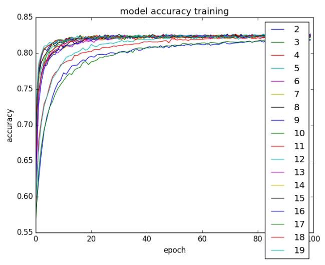
Figure 8.5: One-layered LSTM using different number of units. The legend shows the number of units.

8.1.4 Convolutional Neural Network (CNN)

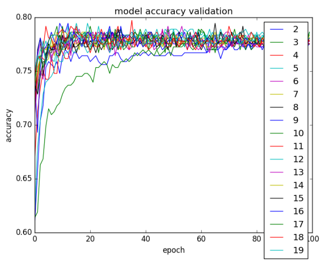
In a convolutional layer the number of parameters is basically the size of the convolutional kernel and the bias, multiplied with the number of kernels like shown in Equation 8.4

$$\text{parameters} = (m \cdot n + \text{bias}) \cdot \text{n kernels} \tag{8.4}$$

Figure 8.6 shows a model with one convolutional layer increasing in size and number of filters. The training set fits better as the size increases, but the optimal validation accuracy stops increasing after some epochs. Figure 8.7 shows different sizes and number of filters, adding a second convolutional layer.

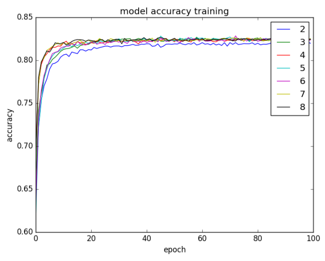


(a) CNN: Accuracy training

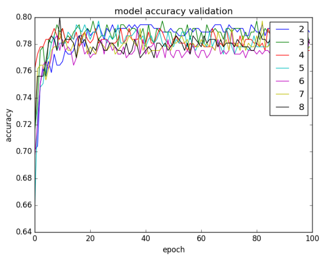


(b) CNN: Accuracy validation

Figure 8.6: CNN with one convolutional layer. The legend represents the size and number of kernels used. This is not the exact numbers but show the relative size. The bigger the legend number the bigger the size and number of kernels.



(a) CNN: Accuracy training



(b) CNN: Accuracy validation

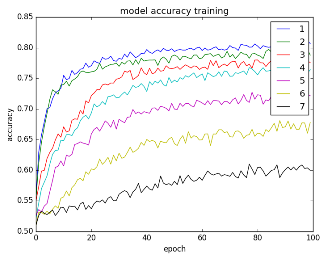
Figure 8.7: CNN with two convolutional layers. The legend represents the size and number of kernels used. This is not the exact numbers but show the relative size. The bigger the legend number the bigger the size and number of kernels.

Figure 8.8 shows the 2-layer convolutional model with added dropout layers using different rates. 8.8(c) 8.8(d) have batch normalization layers in addition, right before the non-linearity. The batch normalization layers reduce overfitting, improving the accuracy slightly in this dataset.

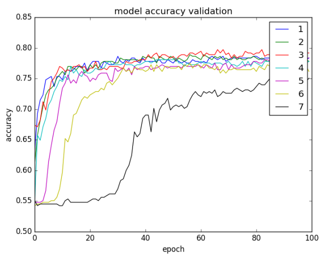
Some other adjustments were tested resulting in decreased or no significant increase in performance:

- Dilated convolution with different rates
- Different activation: LeakyReLU
- Different number and size of kernels
- Pooling layer

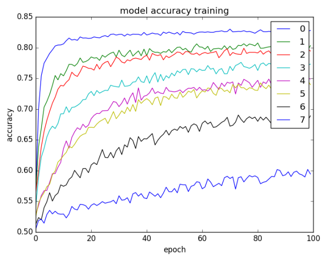
- Adding an extra *Dense* layer before the classifier



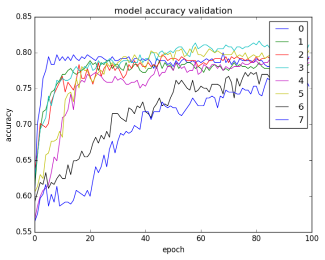
(a) With dropout



(b) With dropout



(c) With dropout and batch norm



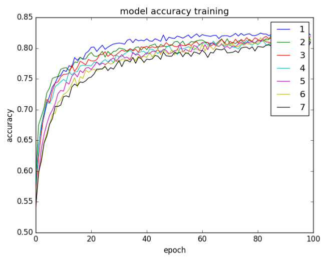
(d) With dropout and batch norm

Figure 8.8: Convolutional layers adding different dropout rates. The legend shows the dropout rates from 0.1 to 0.7.

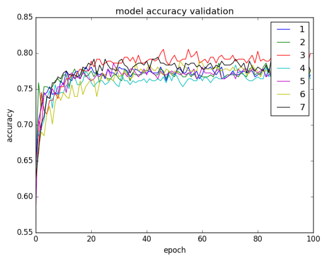
8.1.5 Combinations

Different combinations of LSTMs, CNNs and MLPs can be put together forming the desired topology. Layers can be added in series or parallel. In this project the latency should be kept to a minimum. Accordingly, the different models should preferably be kept in parallel, not building the network too deep.

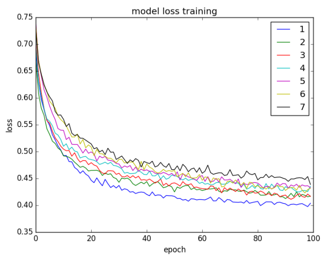
The MLP, LSTM and CNN models from earlier are concatenated making a three input, one output model. After the merging one dropout layer is added. Figure 8.9 shows the training of the model using different dropout rates. Using batch normalization on the combined model did not improve the performance on this dataset. It seems from this test run that some dropout rate would be useful, but multiple runs with batch normalization and dropout layer does not improve performance on this dataset.



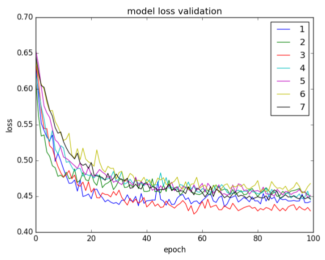
(a) Comb: Accuracy training



(b) Comb: Accuracy validation



(c) Comb: Loss training



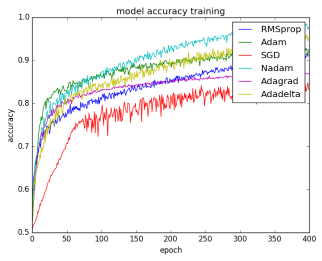
(d) Comb: Loss validation

Figure 8.9: Combination of MLP, LSTM, and CNN using different dropout rates after concatenation. The legend shows the different dropout rates starting from 0.1 to 0.7.

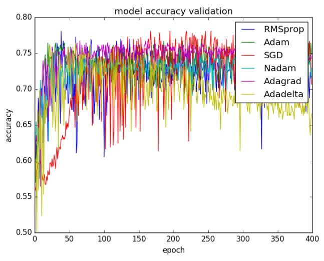
8.2 Choosing Optimizer

Keras provides a range of optimizers with different configuration parameters, though using most optimizers Keras recommend leaving the parameters at their default values. Each optimizer affects the learning of the model differently. Some learns faster, but may be less stable than others. Figure 8.10 shows 400 epochs with different optimizers used on a shallow MLP network with two layers and the classifier. All give similar results in validation accuracy except for Adadelta which performs worse. SGD is clearly the optimizer converging slowest but is the most stable. Adagrad is converging faster and seems stable. For most optimizes, the model accuracy during training and the loss validation does not converge against a value. They keep increasing for each epoch, overfitting the model more and more to the training data.

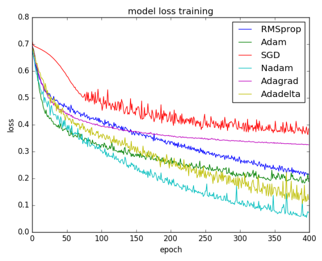
Figure 8.11 shows the different optimizers on a deeper MLP with 12 layers and the classifier. The deeper MLP seems to be less prone to overfitting. Also, in this case the SGD is converging slower than the other optimizers, and Nadam has the best accuracy. Both networks have approximately the same validation accuracy of about 70% on this dataset.



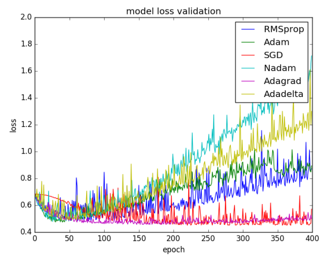
(a) Accuracy Training



(b) Accuracy Validation

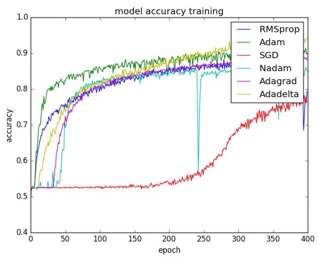


(c) Loss Training

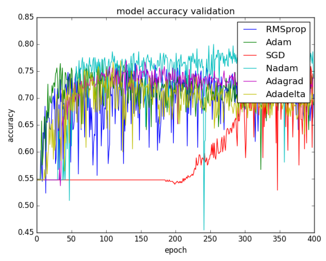


(d) Loss Validation

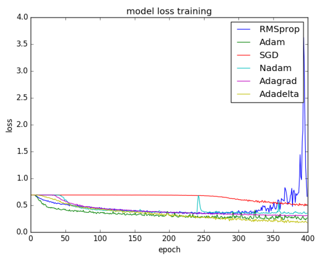
Figure 8.10: Shallow MLP using different optimizers. The legend shows the optimizers used.



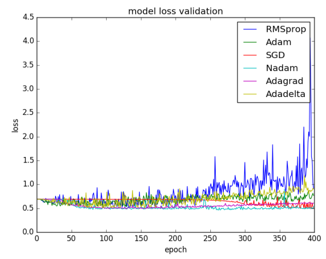
(a) Accuracy Training



(b) Accuracy Validation



(c) Loss Training

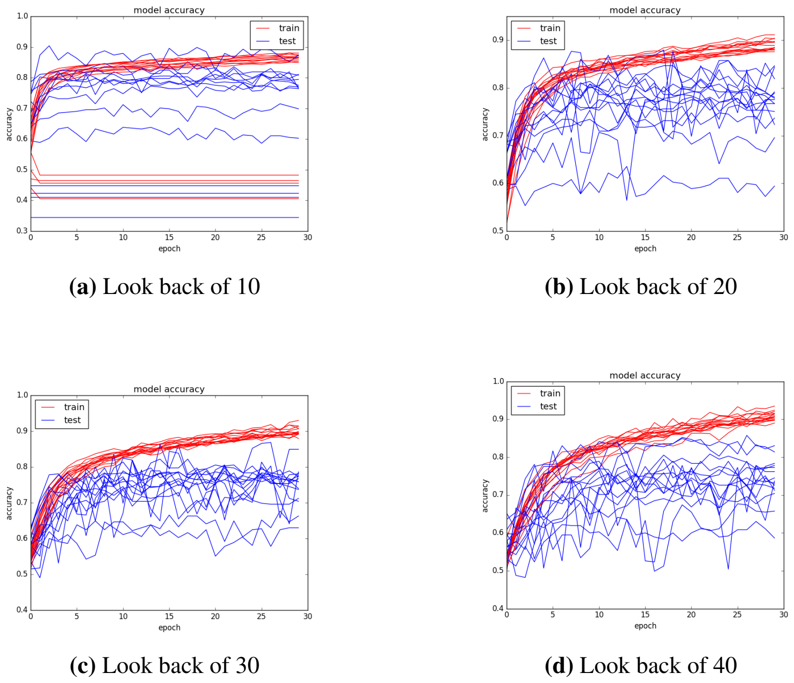


(d) Loss Validation

Figure 8.11: Deeper MLP using different optimizers. The legend shows the optimizers used.

8.3 Choosing Look Back

The model accuracy is dependent on how much history is included in the input frame. Also, the number of parameters, using a MLP model, increases with the size of the input frame, making the model more prone to overfitting. Figure 8.12 shows a MLP using a look back of 10 to 40. The highest accuracy is actually achieved using a fairly low look back using MLP. The different graphs in each plot are from runs using a different *classification feature*. In some cases the model does not seem to learn anything using too low look back. To avoid this, choosing a look back of about 20 seems appropriate.



(a) Look back of 10 **(b)** Look back of 20

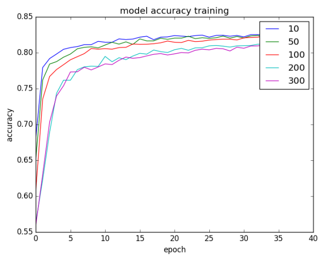
(c) Look back of 30 **(d)** Look back of 40

Figure 8.12: Training with different look back. Each Figure shows multiple runs using a different *classification feature*.

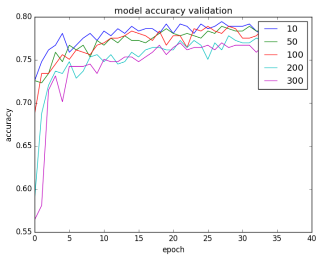
8.4 Choosing Batch Size

Since using all input samples for each iteration of updating the model is inefficient, the training samples are split into mini batches. The size of these batches will affect the performance and training time. As seen in Figure 8.13 using LSTM, best performance is achieved if the batch size is kept relatively low. A test with a simple MLP show less influence by changing the batch size on this dataset.

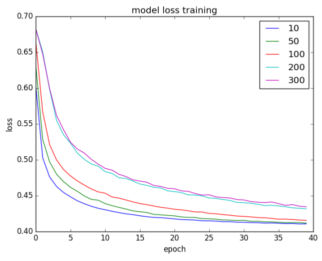
Using all the training samples for each iteration would give the perfect direction in the high dimensional space to update the model, but only the perfect direction to fit the training data. Mini-batches will take a small step in the direction, which minimize the loss function the most, for the samples included in that batch. This introduces some noise in comparison to taking the whole data set on each iteration. Either using a very low batch size like one, or taking the whole data set in one huge batch is also impractical, due to the increasing computation time.



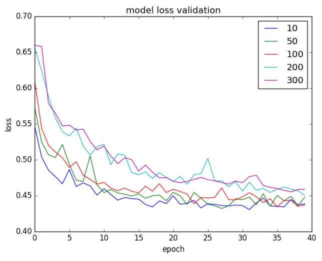
(a) Accuracy training



(b) Accuracy validation



(c) Loss training



(d) Loss validation

Figure 8.13: Training with different batch sizes. The legend shows the size of the training batches.

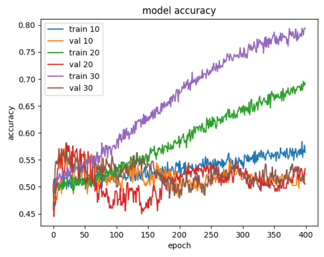
8.5 Analysis of The ”Mysteryset”

This section covers each topology on the highly random dataset, the ”mysteryset”.

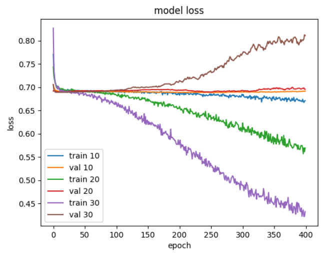
8.5.1 Multi-Layer Perceptron (MLP)

The tests shown in this subsection use two *Dense* layers with dropout.

Figure 8.14 shows different *look back* values using the ”mysteryset”. It looks like the accuracy is mostly unaffected by the different values, and that the network only overfits to the training data when the *look back* is increased. Figure 8.15 shows the accuracy and loss using wider layers. This also results in more overfitting. It looks like increasing the number of neurons increases accuracy slightly. Widening too much results in overfitting, and the separate test set reveals that the model is unable to make a generalization of the problem.

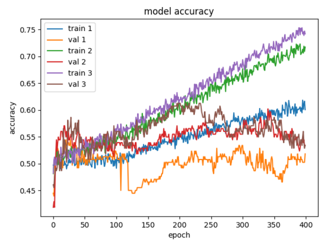


(a) MLP: model accuracy

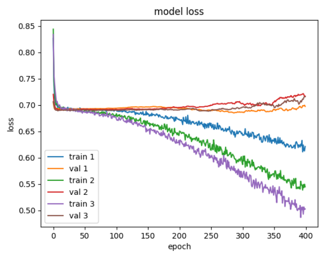


(b) MLP: model loss

Figure 8.14: Different look back values for MLP. The legend shows the training and validation using the different look back values:10, 20, and 30.



(a) MLP: model accuracy



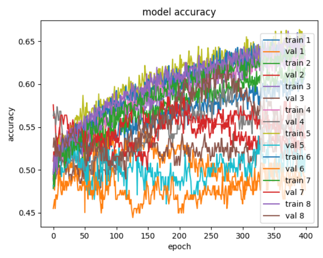
(b) MLP: model loss

Figure 8.15: Different widths of MLP. The legend shows the width of the MLP. train/val 1/2/3 means either training or validation with 10, 20, or 30 neurons in each layer.

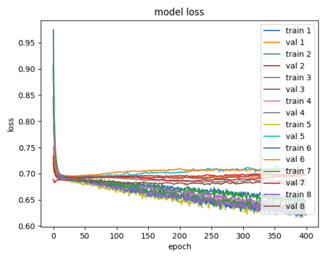
8.5.2 Convolutional Neural Network (CNN)

The tests shown in this subsection use two convolutional layers with batch normalization and dropout.

Figure 8.16 shows results from a simple CNN using different look back lengths from 10 to 80. The increased look back value does not seem to result in much overfitting using the CNN. In contrary to the MLP, this can be expected, since the number of parameters does not increase with the look back value. Though, it does not seem to find any clear structure in the data. When increasing the size and number of the kernels the model overfits to the training data as seen in Figure 8.17.

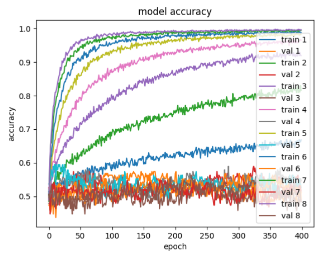


(a) CNN: model accuracy

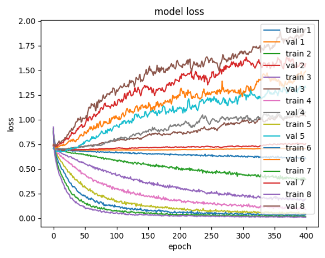


(b) CNN: model loss

Figure 8.16: CNN: Different look back values. The legend shows the different look back values from 10 to 80.



(a) CNN: model accuracy



(b) CNN: model loss

Figure 8.17: CNN: number of size of kernels. The legend represents the size and number of kernels used. This is not the exact numbers but show the relative size. The bigger the legend number, the bigger the size and number of kernels.

Figure 8.18 show some signs of learning structures of the training data giving about 55% correct classifications on the test set. But here, the model predicts an overweight of the rising class. It predicts about 80% rising and the test set contains about 55% rising and 45% falling. The model finds the slight increasing trend over time, but does not catch the inner dynamics of the dataset.

Figure 8.19 shows a CNN using 80 look back. At first glance the network seems unable to learn any significant pattern from the data. But a closer look at the results can show a slightly better performance than 50%. Average classification accuracy over multiple ex-

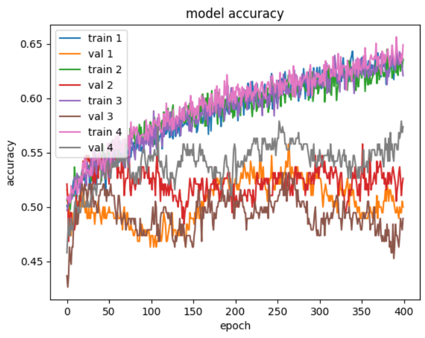
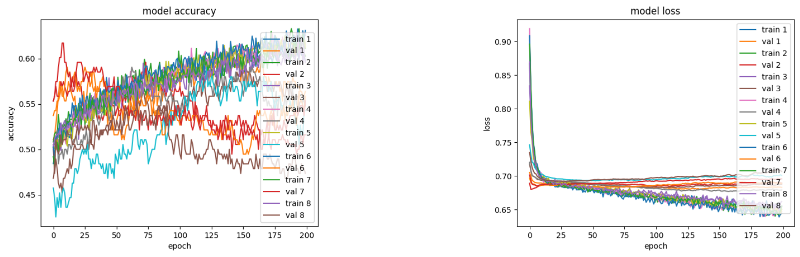


Figure 8.18: CNN learning positive trend. The legend only indicates different runs of the same code.

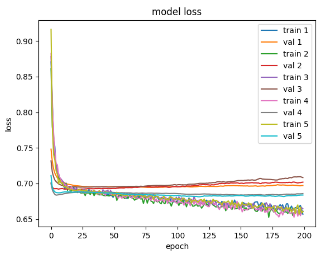
ecutions is about 52%. The interesting thing here is that the model predicts more falling, but the test set contains more rising labels. Even with this mismatch the model predicts above 50% correct. This is repeated with multiple runs using different features as the *classification feature*. After some tuning, the model gives relatively balanced classifications and is able to find a generalization of the data giving better than 50% accuracy. Figure 8.20 shows the model loss of two test runs. It finds some patterns after a few epochs, but does not lower the validation loss further after this initial improvement. Using the same CNN only adding dilated convolutions decrease the accuracy. With a dilation rate of 2 there is a slight decrease in accuracy, and with a dilation rate of 3 the accuracy is almost down to 50%.



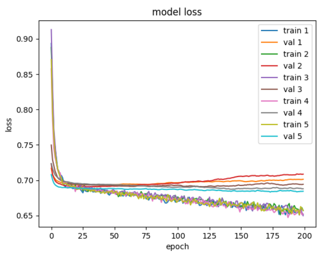
(a) CNN: model accuracy

(b) CNN: model loss

Figure 8.19: CNN with 80 look back. The legend only indicates different runs of the same code.



(a) Model loss example 0

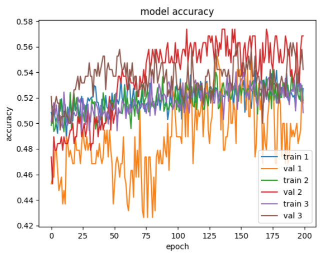


(b) Model loss example 1

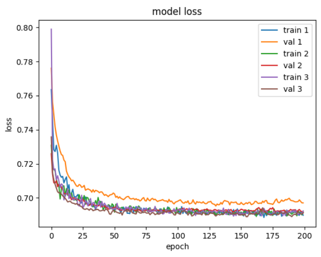
Figure 8.20: CNN: Model loss examples. The legend only indicates different runs of the same code.

8.5.3 Recurrent Neural Network (RNN & LSTM)

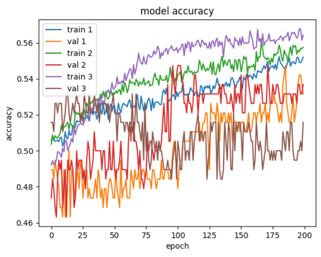
Figure 8.21 shows different runs with RNN and LSTM topologies. There are some similarities to the behavior of the MLP network. In some cases, it finds an upward going trend making very biased classifications, but it cannot find any good generalization of the data. It seems like the model learns some patterns, but the small validation set also makes it somewhat harder to make that assumption. The test data reveals that the model performs no better than random, i.e. 50%. Over multiple runs, the average accuracy on the test set can be 52-54%. Taking the biased classification into considerations, this is no better than random.



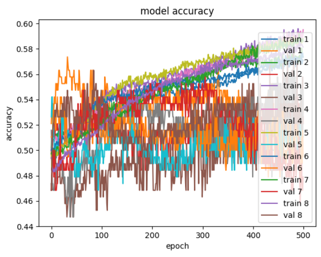
(a) Simple 2-layered RNN



(b) Simple 2-layered RNN



(c) 2-layered RNN. More units

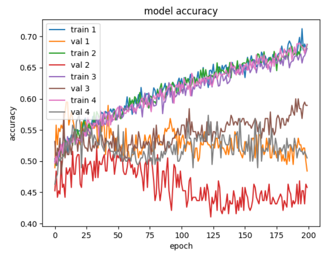


(d) 2-layered LSTM

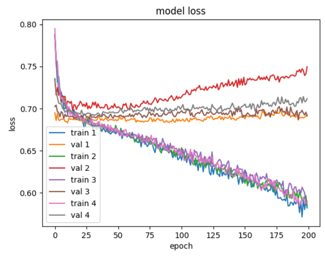
Figure 8.21: RNN & LSTM: Figure (a) and (b) show three runs from a simple 2-layered RNN model. Figure (c) has increased the number of units. This results in overfitting, and a more biased classification. Figure (d) shows multiple runs multiple runs with a 2-layered LSTM using the same code. The average test results give about 52% accuracy, but with very biased classification.

8.5.4 Combinations

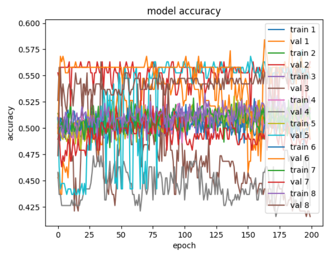
Both series and parallel combination of the MLP, LSTM, and CNN are tested, forming a more complex network. An attempt is also made to make use of several CNNs in parallel, concatenating them and using a MLP before the classifier. Figure 8.22 shows some of the training using these combinations. It seems like the model in some cases can extract information about the training set, making it perform slightly better than 50% on the test set. Widening the *Dense* layer after the concatenations of the CNNs, results in overfitting of the network.



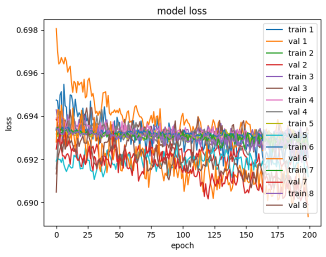
(a) Three CNNs in parallel



(b) Three CNNs in parallel



(c) LSTM, MLP, and CNN concatenated



(d) LSTM, MLP, and CNN concatenated

Figure 8.22: Figure (a) and (b) show the training using parallel CNNs. Figure (c) and (d) use a concatenation of the LSTM, MLP, and CNN. The legend only indicates different runs of the same code

Chapter 9

Implementation

This chapter shows all the different model stacks using the different architectures. All the activation functions except for the classifiers in the these model stacks are ReLU. Note that the activation functions can be a part of another layer, like dense_1 (Dense, ReLU), or as a stand-alone layer, activation_1 (ReLU).

9.1 Multi-Layer Perceptron (MLP)

The model stack of the MLP is shown in Table 9.1.

Layer (type)	Output shape	Param #
flatten_1 (Flatten)	(None, 96)	0
dense_1 (Dense, ReLU)	(None, 20)	1940
dropout_1 (Dropout 0.3)	(None, 20)	0
dense_2 (Dense, ReLU)	(None, 5)	105
dropout_2 (Dropout 0.3)	(None, 5)	0
dense_3 (Dense)	(None, 2)	12
activation_1 (softmax)	(None, 2)	0

Table 9.1: MLP model

Total params: 2,057
Trainable params: 2,057
Non-trainable params: 0

9.2 Convolutional Neural Network (CNN)

The model stack of the CNN is shown in Table 9.2.

Layer (type)	Output shape	Param #
conv_1 (Conv1D)	(None, 40, 12)	2316
nb_filter=12, filter_length=4,		
batch_normalization_1	(None, 40, 12)	48
activation_1 (ReLU)	(None, 40, 12)	0
dropout_1 (Dropout 0.3)	(None, 40, 12)	0
conv_2 (Conv1D)	(None, 40, 2)	26
nb_filter=2, filter_length=1,		
batch_normalization_2	(None, 40, 2)	8
activation_2 (ReLU)	(None, 40, 2)	0
dropout_2 (Dropout 0.3)	(None, 40, 2)	0
flatten_1 (Flatten)	(None, 80)	0
dense_1 (Dense)	(None, 2)	162
activation_3 (softmax)	(None, 2)	0

Table 9.2: CNN model

Total params: 2,560
Trainable params: 2,532
Non-trainable params: 28

9.3 Recurrent Neural Networks (RNN & LSTM)

The simple RNN was not able to find any generalization of the data. For that reason, the simple RNN is not included from here on out. The model stack of the LSTM is shown in Table 9.3.

Layer (type)	Output shape	Param #
lstm_1 (5 units)	(None, 40, 5)	760
lstm_2 (5 units)	(None, 5)	220
dense_1 (Dense)	(None, 2)	12
activation_1 (softmax)	(None, 2)	0

Table 9.3: LSTM model

Total params: 992
Trainable params: 992
Non-trainable params: 0

9.4 Combinations

The model stacks of three concatenated CNNs with a MLP before the classifier are shown in Table 9.4.

Layer (type)	Output shape	Param #	Connected to
input_1 (InputLayer)	(None, 40, 32)	0	
input_2 (InputLayer)	(None, 40, 32)	0	
input_3 (InputLayer)	(None, 40, 32)	0	
conv1d_1 (Conv1D)	(None, 40, 12)	2316	input_1[0][0]
conv1d_3 (Conv1D)	(None, 40, 12)	2316	input_2[0][0]
conv1d_5 (Conv1D)	(None, 40, 12)	2316	input_3[0][0]
batch_normalization_1	(None, 40, 12)	48	conv1d_1[0][0]
batch_normalization_3	(None, 40, 12)	48	conv1d_3[0][0]
batch_normalization_5	(None, 40, 12)	48	conv1d_5[0][0]
activation_1 (Activation)	(None, 40, 12)	0	batch_normalization_1[0][0]
activation_3 (Activation)	(None, 40, 12)	0	batch_normalization_3[0][0]
activation_5 (Activation)	(None, 40, 12)	0	batch_normalization_5[0][0]
dropout_1 (Dropout 0.3)	(None, 40, 12)	0	activation_1[0][0]
dropout_3 (Dropout 0.3)	(None, 40, 12)	0	activation_3[0][0]
dropout_5 (Dropout 0.3)	(None, 40, 12)	0	activation_5[0][0]
conv1d_2 (Conv1D)	(None, 40, 2)	26	dropout_1[0][0]
conv1d_4 (Conv1D)	(None, 40, 2)	26	dropout_3[0][0]
conv1d_6 (Conv1D)	(None, 40, 2)	26	dropout_5[0][0]
batch_normalization_2	(None, 40, 2)	8	conv1d_2[0][0]
batch_normalization_4	(None, 40, 2)	8	conv1d_4[0][0]
batch_normalization_6	(None, 40, 2)	8	conv1d_6[0][0]
activation_2 (Activation)	(None, 40, 2)	0	batch_normalization_2[0][0]
activation_4 (Activation)	(None, 40, 2)	0	batch_normalization_4[0][0]
activation_6 (Activation)	(None, 40, 2)	0	batch_normalization_6[0][0]
dropout_2 (Dropout 0.3)	(None, 40, 2)	0	activation_2[0][0]
dropout_4 (Dropout 0.3)	(None, 40, 2)	0	activation_4[0][0]
dropout_6 (Dropout 0.3)	(None, 40, 2)	0	activation_6[0][0]
concatenate_1 (Concatenate)	(None, 40, 6)	0	dropout_2[0][0]
			dropout_4[0][0]
			dropout_6[0][0]
flatten_1 (Flatten)	(None, 240)	0	concatenate_1[0][0]
dense_1 (Dense, ReLU)	(None, 10)	2410	flatten_1[0][0]
dropout_7 (Dropout 0.3)	(None, 10)	0	dense_1[0][0]
dense_2 (Dense, ReLU)	(None, 5)	55	dropout_7[0][0]
dropout_8 (Dropout 0.3)	(None, 5)	0	dense_2[0][0]
output_1 (Dense, softmax)	(None, 2)	12	dropout_8[0][0]

Table 9.4: Three CNNs concatenated with a simple MLP before the classifier.

```
Total params: 9,671
Trainable params: 9,587
Non-trainable params: 84
```

Part IV

Results and Discussion

Chapter 10

Results & Discussion

The results are presented for each topology in separate sections. Discussion about each topology's performance is also assessed in this chapter, while the overall discussion is left for the next chapter. Many of the models predicts a much higher amount of one class. This may give a false impression of performance when the test set contains more of the same class. For example: if there are 60% rising labels in the test set, predicting only that class would give 60% accuracy. To compensate for this, the performance for the random baseline will not be exactly 50%, but adjusted for how many predictions there are for each label. If the predictions are totally balanced with an equal amount of predictions for each classification, the baseline would be 50%. With 60% of one label in the test set and if 75% of the predictions are for that label, the random chance for these predictions would be 60% and 40% chance for the rest. The end baseline will be set to 55% in this case, which is the weighted sum of these accuracies as seen in Equation 10.1.

$$Random_{adjusted} = \frac{UP}{UP + DN} \cdot P(UP)[\%] \; + \frac{DN}{UP + DN} \cdot P(DN)[\%] \qquad (10.1)$$

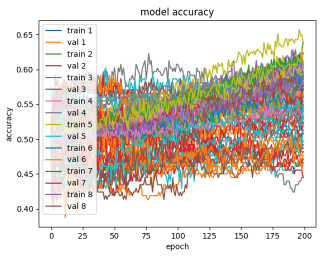
All results in the tables are from consecutive runs for each represented architecture. This it done to avoid a selection of specific runs. Even running identical code repeatedly gives natural variations in performance, since the weights are initiated randomly. The tables include the following columns:

- UP predictions: the number of rising prediction on the test set
- DN predictions: the number of falling prediction on the test set
- Random adjusted: the adjusted random baseline is calculated from Equation 10.1. The percentage of rising label in the test set, P(UP), are not given a dedicated column, but it is included in the calculation of the random adjusted value.
- Actual correct: the accuracy [%] predicted the model
- Diff: The difference between the random adjusted and the actual correct

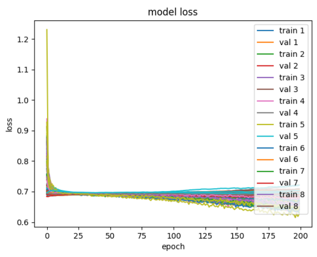
10.1 Multi-Layer Perceptron (MLP)

The MLP network does not seem to find any specific patterns in the dataset. In some cases it seems like the network is learning some patterns. Below is an example of the network giving about 53% accuracy on the validation set. The network predicts most falling, but there are more rising labels in the test set, and the network still achieves right over 50% on this separate test set. With a selection of runs there can be observed some behavior that could resemble learning, but running multiple times shows that this is only statistical variations. Over time the classification accuracy closes in on 50%. Figure 10.1 shows 50 runs of using the MLP.

```
val_acc: 0.53
acc: test data 50.12%
number of up predicitons:  198.0
number of down predicitons:  225.0
number of up in test:  239.0
number of down in test:  184.0
up procentage in test:  56.50
number of up in training:  1920.0
number of down in training:  1884.0
up procentage in training:  50.47
```



(a) MLP: model accuracy



(b) MLP: model loss

Figure 10.1: 50 runs using the MLP. Average accuracy of about 50%. The legend only indicates the 8 first runs.

Runs from classification of four different features are shown in Table 10.1. Only comparing to 50% accuracy gives a better performance in all cases. Especially the last 8 runs show a 4.5% better than 50% not considering the biased classification. Comparing to the adjusted random accuracy in Equation 10.1, the model has no improvement in performance. The average performance comparing to *adjusted random* is 0.1% worse than random.

# UP predictions	# DN predictions	Random adjusted [%]	Actual correct [%]	Diff[%]
109	316	46.74	46.82	0.08
294	131	52.57	52.71	0.14
347	78	54.24	57.18	2.94
379	46	55.25	55.76	0.51
238	187	50.80	49.41	-1.39
267	158	51.72	50.12	-1.60
274	151	51.94	51.29	-0.65
146	279	47.90	47.53	-0.37
387	38	55.50	57.18	1.68
385	40	55.44	54.82	-0.62
49	376	44.84	45.65	0.81
246	179	51.06	50.59	-0.47
384	41	55.41	54.82	-0.59
283	142	52.22	56.00	3.78
283	142	52.22	52.71	0.49
307	118	52.98	48.47	-4.51
131	294	47.43	45.65	-1.78
324	101	53.52	51.06	-2.46
396	29	55.79	57.18	1.39
359	66	54.62	54.59	-0.03
390	35	55.60	54.82	-0.78
349	76	54.30	54.59	0.29
325	100	53.55	53.65	0.10
288	137	52.38	51.06	-1.32
389	36	55.56	56.94	1.38
366	59	54.84	53.41	-1.43

Table 10.1: MLP: results

10.2 Convolutional Neural Network (CNN)

The CNN seems to find some structures in the data. It achieves to make a generalization of the data, resulting in more accurate model predictions on the separate test set. The classification here is so to speak balanced, i.e. the results are not significantly affected by adjusting the baseline. Table 10.2 show the results from multiple runs using different features for classification. The overall average accuracy here is 2.48 % over baseline. Comparing this against 50% would give 2.63% better than baseline.

# UP predictions	# DN predictions	Random adjusted [%]	Actual correct [%]	Diff[%]
196	225	49.55	47.74	-1.81
226	195	50.48	53.44	2.96
227	194	50.51	55.11	4.60
236	185	50.79	56.29	5.50
227	194	50.51	49.88	-0.63
199	222	49.64	52.73	3.09
198	223	49.61	51.54	1.93
208	213	49.92	49.64	-0.28
178	239	49.06	47.96	-1.10
182	235	49.19	55.16	5.97
195	222	49.59	51.08	1.49
181	236	49.16	52.04	2.88
165	252	48.66	54.44	5.78
192	225	49.49	47.96	-1.53
202	215	49.80	50.84	1.04
188	229	49.37	51.32	1.95
246	175	51.35	56.77	5.42
205	216	49.79	51.78	1.99
243	178	51.24	56.53	5.29
217	204	50.25	48.93	-1.32
228	193	50.67	51.54	0.87
240	181	50.64	53.92	3.28
229	192	50.40	54.16	3.76
250	171	50.86	59.14	8.28
266	155	51.21	54.39	3.18
259	162	51.06	51.78	0.72
170	251	50.62	50.36	-0.26
219	202	49.87	50.59	0.72
205	216	50.08	52.49	2.41
190	231	50.31	52.73	2.42
176	245	50.52	53.68	5.29

Table 10.2: CNN: results

10.3 Recurrent Neural Network (LSTM)

Some results from the LSTM model are shown in Table 10.3. The average accuracy is 0.17% worse than the adjusted baseline. It performs 2% better than 50% because of the skewed classifications. It has similar results as the MLP network.

# UP predictions	# DN predictions	Random adjusted [%]	Actual correct [%]	Diff[%]
246	175	51.10	49.64	-1.46
238	183	50.85	49.17	-1.68
200	221	49.68	48.69	-0.99
243	178	51.00	49.41	-1.59
225	196	50.45	50.83	0.38
305	116	52.92	52.73	-0.19
292	129	52.52	49.64	-2.88
260	161	51.53	52.02	0.49
242	179	50.97	48.69	-2.28
255	166	51.37	51.31	-0.06
219	202	50.26	53.68	3.42
290	131	52.45	52.49	0.04
272	149	51.90	53.44	1.54
218	203	50.23	49.64	-0.59
240	181	50.91	49.64	-1.27
222	199	50.36	49.64	-0.72
316	105	53.26	56.29	3.03
310	111	53.07	53.92	0.85
345	76	54.15	53.68	-0.47
310	111	53.07	52.49	-0.58
290	131	52.45	52.97	0.52
342	79	54.06	52.97	-1.09
384	37	55.36	55.34	-0.02
287	134	52.36	49.88	-2.48
302	119	52.83	51.07	-1.76
217	204	50.20	52.73	2.53
203	218	49.77	49.41	-0.36
281	140	52.18	53.68	1.50
317	104	53.29	55.11	1.82
347	74	54.21	52.26	-1.95
391	30	55.57	56.06	0.49
367	54	54.83	50.83	-4.00
200	221	49.68	48.69	-0.99
312	109	53.13	53.44	0.31
285	136	52.30	48.93	-3.37
332	89	53.75	53.92	0.17
320	101	53.38	52.97	-0.41
251	170	51.25	56.06	4.81
251	170	51.25	50.83	-0.42
367	54	54.83	57.48	2.65

Table 10.3: LSTM: results

10.4 Combinations

Creating different combinations of the previous networks is not believed to increase performance, when only the CNN shows promising results. But it can be hard to predict the exact performance of the network when making substantial changes to its architecture. Table 10.4 shows the results using three CNNs concatenated with a simple MLP before the classifier. The accuracy is 0.8% better than the adjusted baseline, while it is 2.66% better than 50%. This is reflected by the biased classification. This show some results, but way worse than using only the CNN.

# UP predictions	# DN predictions	Random adjusted [%]	Actual correct [%]	Diff[%]
371	50	54.96	56.53	1.57
363	58	54.71	56.06	1.35
255	166	51.37	50.83	-0.54
323	98	53.47	55.11	1.64
369	52	54.89	55.11	0.22
344	77	54.12	55.34	1.22
343	78	54.09	53.68	-0.41
314	107	53.20	54.39	1.19
326	95	53.57	52.97	-0.60
361	60	54.65	56.53	1.88
368	53	54.86	55.82	0.96
347	74	54.21	56.53	2.32
160	261	48.44	50.59	2.15
150	271	48.13	48.69	0.56
381	40	55.26	56.53	1.27
296	125	52.64	52.49	-0.15
326	95	53.57	55.34	1.77
155	266	48.29	46.08	-2.21
377	44	55.14	56.06	0.92
349	72	54.28	57.96	3.68
85	336	46.12	49.41	3.29
105	316	46.74	47.03	0.29
354	67	54.43	56.77	2.34
330	91	53.69	55.82	2.13
202	219	49.74	53.92	4.18
202	219	49.74	47.27	-2.47
162	259	48.50	48.22	-0.28
188	233	49.31	47.27	-2.04

Table 10.4: Three concatenated CNNs: results

Chapter 11

Overall Discussion

The success criteria from the functional description of achieving over 50% accuracy was only accomplished using the CNN, when the biased classifications are taken into considerations. Combinations of topologies gave some positive results, but was worse than only using the CNN. Keeping the size of the networks within the limit was no real issue. Using larger networks result in overfitting without having more training samples.

The Main goal of the project was achieved using the CNN. The results from this thesis indicate that there are structures and pattern to be found in the data.

It is very likely that the model performs better with further optimization. Not to mention, that there is probably a lot to gain in the pre-processing and feature selection phase, which is much based on trial and error during this project. There exist other methods, as explored by Jundong Li (2016) [50], that might give a better starting point, resulting in higher accuracy. From a better starting point, the other topologies which did not work in this case, might also do better, though this thesis's results indicate that CNNs probably are the best choice for the problem. Although the thesis does not provide an optimal solution, it gives the indented *proof of concept* and starting point for further research and development.

As LSTMs are able to find long term dependencies, dilated convolutions widen the receptive field enabling the CNN to utilize longer dependencies. During the design phase, dilated convolution was tested. The accuracy decreased with higher dilation rates. The model does not seem to benefit from these longer dependencies, but rather perform worse. This can also be some of the reasons LSTMs did not work on the ”mysteryset”. If samples further apart are less related, samples closer together, or values in between samples, might be more related. Reducing the filter size does not improve the results, but it would be interesting to see if increasing the sample rate would result in higher accuracy. This could not be done in this case since an up-sampled version of the dataset was not available. An up-sampled version of the data would also give more training examples. Furthermore, only acquiring and running tests using more samples with the same sampling frequency would be interesting. Taking advantage of that e.g. pixels closer together in an image are more related than pixels far away are exactly why CNNs are used a lot in image classification tasks. This also seems to apply for the problem in question, which can be a partial reason why the CNN worked much better than the MLP and LSTM.

The softmax classifier outputs the probabilities, predicted by the model, for each class. Using two classes, this results in two probabilities that represents how strongly the model predicts that the next value of the classification feature is bigger or smaller than the current value. There could be interesting to investigate the distribution of probabilities, to see if the model, more often, predicts the correct outcome in the cases when the predictions are stronger.

In addition to binary classification, one could increase the resolution of the output by adding more classes. E.g. using a three-way classifier for rising, falling, and staying within some limit. Regression can also be an option for future work, though higher performance would probably be needed to yield any usable results. This project assesses one-step forecasts. One might also do multi-step forecasts, with a probable decrease in accuracy, extending the horizon. Before trying with multiple steps, the accuracy on the one-step forecasts should be increased.

For the sake of this assignment, the use case indicates that there is no time to retrain the network after each sample. In a live application, somewhere down the line, the model has to retrain on the new arriving data. At which point the accuracy starts declining because the model is fitted to old, less representative data, is unknown. Live, this would probably be done using a moving window, retraining the model every given number of samples.

This project assesses the use of neural networks to predict the next value in the dataset in question. Also using other types of supervised machine learning algorithms could be tested.

The end goal is to implement the inference part of the network in hardware, lowering the latency, while still doing training on GPUs. In that context lowering the bit accuracy, or even trying binary representation for each weight as reviewed by the literature, might be considered, optimizing the network for size and latency.

Part V

Closing Remarks

Chapter 12

Conclusion

This chapter gives a short conclusion of the thesis including future work, repeated in with bullet points, already assessed in the discussion.

12.1 Overview

This thesis has investigated the application of different neural network topologies, used in one-step forecasting, on highly random data. This is meant as a *proof of concept* for further development and implementation in an FPGA. The development platform for testing the different topologies are built around a freely available deep learning library in Python, called *Keras*.

The literature review assesses the use of MLPs, RNNs, LSTMs and CNNs. Models of these, based on the literature review and trial and error have been tested. Only the CNN have achieved the success criteria of classifying features with an accuracy over 50% - about 2,5% better than the baseline. Most likely, more research and development would improve this performance further. In hindsight of testing, pre-processing, feature selection and handling of highly random data should have been included in the literature review.

Some of the papers reviewed in the literature use methods to increase the receptive field, making the networks able to utilize dependencies between samples over a larger time span. LSTMs are introduced to handle longer dependencies in recurrent networks, and dilated convolutions are used in CNNs. Adding dilated convolutions for the CNN decreased its performance. These findings suggest that adding ways of utilizing longer dependencies, decreases the model's prediction accuracy. Using an up-sampled version of the dataset might have stronger relations between samples increasing the performance.

Even though the CNN topology gave satisfying results, this does not exclude other approaches, machine learning algorithms or other procedures, not included in this thesis.

12.2 Future Work

Based on the final discussion in Chapter 11, the future work can be summed up to:

- Acquire dataset with higher sampling frequency
- Trying different pre-processing and feature selection methods
- Trying other supervised machine learning algorithms
- Multi-step forecasting
- Categorical classification and/or regression
- Refit model every n samples using a moving window
- Implementation in an FPGA
- Exploration with bit accuracy

Bibliography

[1] Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., Zheng, X., 2015. TensorFlow: Large-scale machine learning on heterogeneous systems. Software available from tensorflow.org.
URL `https://www.tensorflow.org/`

[2] Alemdar, H., Caldwell, N., Leroy, V., Prost-Boucle, A., Pétrot, F., 2016. Ternary neural networks for resource-efficient AI applications. CoRR abs/1609.00222.
URL `http://arxiv.org/abs/1609.00222`

[3] Barney, L., 2017. Can fpgas beat gpus in accelerating next-generation deep learning? Accessed: 2018-01-29.
URL `https://www.nextplatform.com/2017/03/21/can-fpgas-beat-gpus-accelerating-next-generation-deep-learning/`

[4] Bengio, Y., 2009. Learning deep architectures for ai. Foundations and Trends in Machine Learning 2 (1), 1–127.
URL `http://dx.doi.org/10.1561/2200000006`

[5] Borovykh, A., Bohte, S., Oosterlee, C. W., Mar. 2017. Conditional Time Series Forecasting with Convolutional Neural Networks. ArXiv e-prints.

[6] Brownlee, J., 2016. How to build multi-layer perceptron neural network models with keras. Accessed: 2018-05-23.
URL `https://machinelearningmastery.com/build-multi-layer-perceptron-neural-network-models-keras/`

[7] Brownlee, J., 2016. Time series prediction with lstm recurrent neural networks in python with keras. Accessed: 2018-05-23.

URL https://machinelearningmastery.com/time-series-prediction-lstm-recurrent-neural-networks-python-keras/

[8] Canziani, A., Paszke, A., Culurciello, E., 2016. An analysis of deep neural network models for practical applications. CoRR abs/1605.07678.
URL http://arxiv.org/abs/1605.07678

[9] Chen, J., Wang, D., 2016. Long short-term memory for speaker generalization in supervised speech separation. The Journal of the Acoustical Society of America 141 6, 4705.

[10] Chen, Z., Yi, D., 2017. The game imitation: Deep supervised convolutional networks for quick video game AI. CoRR abs/1702.05663.
URL http://arxiv.org/abs/1702.05663

[11] Chollet, F., 2016. Xception: Deep learning with depthwise separable convolutions. CoRR abs/1610.02357.
URL http://arxiv.org/abs/1610.02357

[12] Chollet, F., et al., 2015. Keras. https://keras.io.

[13] Collobert, R., Bengio, S., Marithoz, J., 2002. Torch: A modular machine learning software library.

[14] Courbariaux, M., Bengio, Y., 2016. Binarynet: Training deep neural networks with weights and activations constrained to +1 or -1. CoRR abs/1602.02830.
URL http://arxiv.org/abs/1602.02830

[15] Coussy, P., Gajski, D. D., Meredith, M., Takach, A., 2009. An introduction to high-level synthesis. IEEE Design & Test of Computers 26 (4), 8–17.
URL http://dblp.uni-trier.de/db/journals/dt/dt26.html#CoussyGMT09

[16] Cui, Z., Chen, W., Chen, Y., 2016. Multi-scale convolutional neural networks for time series classification. CoRR abs/1603.06995.
URL http://arxiv.org/abs/1603.06995

[17] Dodge, S., Karam, L., May 2017. A study and comparison of human and deep learning recognition performance under visual distortions.

[18] Engineering, H., 2015. Introduction to convolution neural networks. Accessed:2018-01-24.
URL https://engineering.huew.co/introduction-to-convolution-neural-networks-18981d1cd09a

[19] Esser, S. K., Merolla, P. A., Arthur, J. V., Cassidy, A. S., Appuswamy, R., Andreopoulos, A., Berg, D. J., McKinstry, J. L., Melano, T., Barch, D. R., di Nolfo, C., Datta, P., Amir, A., Taba, B., Flickner, M. D., Modha, D. S., 2016. Convolutional networks for fast, energy-efficient neuromorphic computing. CoRR abs/1603.08270.
URL http://arxiv.org/abs/1603.08270

[20] Floydhub, 2018. Floydhub home page. Accessed: 2018-04-18.
URL https://docs.floydhub.com/

[21] FPGA, I., 2017. Introduction to intel fpga ip cores. Accessed: 2018-01-28.
URL https://www.altera.com/documentation/mwh1409960636914.html

[22] FPGA, I., 2018. Platform designer (formerly qsys). Accessed: 2018-01-28.
URL https://www.altera.com/products/design-software/fpga-design/quartus-prime/features/qts-platform-designer.html

[23] Gao, X., 2017. Fpga 2017 (part 1): Fpgas versus gpus in deep learning. Accessed: 2018-01-29.
URL https://admk.github.io/2017/07/13/fpga-2017-part-1-fpgas-vs-gpus.html

[24] Geirhos, R., Janssen, D. H. J., Schtt, H. H., Rauber, J., Bethge, M., Wichmann, F. A., June 2017. Comparing deep neural networks against humans: object recognition when the signal gets weaker.

[25] Gensler, A., Henze, J., Sick, B., Raabe, N., Oct 2016. Deep learning for solar power forecasting x2014; an approach using autoencoder and lstm neural networks. In: 2016 IEEE International Conference on Systems, Man, and Cybernetics (SMC). pp. 002858–002865.

[26] Gers, F. A., Eck, D., Schmidhuber, J., 2002. Applying lstm to time series predictable through time-window approaches. In: Tagliaferri, R., Marinaro, M. (Eds.), Neural Nets WIRN Vietri-01. Springer London, London, pp. 193–200.

[27] Gers, F. A., Schmidhuber, J., Cummins, F., 1999. Learning to forget: Continual prediction with lstm. Neural Computation 12, 2451–2471.

[28] Glorot, X., Bordes, A., Bengio, Y., 11–13 Apr 2011. Deep sparse rectifier neural networks. In: Gordon, G., Dunson, D., Dudk, M. (Eds.), Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics. Vol. 15 of Proceedings of Machine Learning Research. PMLR, Fort Lauderdale, FL, USA, pp. 315–323.
URL http://proceedings.mlr.press/v15/glorot11a.html

[29] Greff, K., Srivastava, R. K., Koutnk, J., Steunebrink, B. R., Schmidhuber, J., March 2015. Lstm: A search space odyssey.

[30] Groß, W., Lange, S., Boedecker, J., Blum, M., 2017. Predicting time series with space-time convolutional and recurrent neural networks.

[31] GUPTA, D., 2017. Fundamentals of deep learning activation functions and when to use them? Accessed: 2018-01-23.
URL https://www.analyticsvidhya.com/blog/2017/10/fundamentals-deep-learning-activation-functions-when-to-use-them/

[32] Hao, Y., Quigley, S., 2017. The implementation of a deep recurrent neural network language model on a xilinx FPGA. CoRR abs/1710.10296.
URL `http://arxiv.org/abs/1710.10296`

[33] Harry Fairhead, i. c. Q. J., 2014. The mcculloch-pitts neuron. Accessed: 2018-01-19.
URL `http://www.i-programmer.info/babbages-bag/325-mcculloch-pitts-neural-networks.html`

[34] He, K., Zhang, X., Ren, S., Sun, J., 2015. Deep residual learning for image recognition. CoRR abs/1512.03385.
URL `http://arxiv.org/abs/1512.03385`

[35] Hegde, G., Siddhartha, Ramasamy, N., Kapre, N., Oct 2016. Caffepresso: An optimized library for deep learning on embedded accelerator-based platforms. In: 2016 International Conference on Compliers, Architectures, and Sythesis of Embedded Systems (CASES). pp. 1–10.

[36] Hsu, D., 2017. Time series forecasting based on augmented long short-term memory. CoRR abs/1707.00666.
URL `http://arxiv.org/abs/1707.00666`

[37] Iandola, F. N., Moskewicz, M. W., Ashraf, K., Han, S., Dally, W. J., Keutzer, K., 2016. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <1mb model size. CoRR abs/1602.07360.
URL `http://arxiv.org/abs/1602.07360`

[38] Imanuel, 2017. Top 15 deep learning software. Accessed: 2018-02-07.
URL `https://www.predictiveanalyticstoday.com/deep-learning-software-libraries/`

[39] Ioffe, S., Szegedy, C., 2015. Batch normalization: Accelerating deep network training by reducing internal covariate shift. pp. 448–456.
URL `http://jmlr.org/proceedings/papers/v37/ioffe15.pdf`

[40] Jeremy Hsu, i. s., 2015. Biggest neural network ever pushes ai deep learning. Accessed: 2018-01-19.
URL `https://spectrum.ieee.org/tech-talk/computing/software/biggest-neural-network-ever-pushes-ai-deep-learning`

[41] Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R., Guadarrama, S., Darrell, T., 2014. Caffe: Convolutional architecture for fast feature embedding. arXiv preprint arXiv:1408.5093.

[42] KARNOWSKI, J., 2015. Alexnet visualization. Accessed: 2018-01-24.
URL `https://jeremykarnowski.wordpress.com/2015/07/15/alexnet-visualization/`

[43] Keras-team, 2018. Keras git repository. Accessed: 2018-05-23.
URL `https://github.com/keras-team/keras/tree/master/examples`

[44] Kompella, R., 2017. Using lstms to forecast time-series. Accessed: 2018-05-23.
URL `https://towardsdatascience.com/using-lstms-to-forecast-time-series-4ab688386b1f`

[45] Krizhevsky, A., Sutskever, I., Hinton, G. E., 2012. Imagenet classification with deep convolutional neural networks. In: Pereira, F., Burges, C. J. C., Bottou, L., Weinberger, K. Q. (Eds.), Advances in Neural Information Processing Systems 25. Curran Associates, Inc., pp. 1097–1105.
URL `http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf`

[46] Kuremoto, T., Kimura, S., Kobayashi, K., Obayashi, M., 2014. Time series forecasting using a deep belief network with restricted boltzmann machines. Neurocomputing 137, 47 – 56, advanced Intelligent Computing Theories and Methodologies.
URL `http://www.sciencedirect.com/science/article/pii/S0925231213007388`

[47] Lacey, G., Taylor, G. W., Areibi, S., February 2016. Deep learning on fpgas: Past, present, and future.

[48] Le, Q. V., Ranzato, M., Monga, R., Devin, M., Chen, K., Corrado, G. S., Dean, J., Ng, A. Y., December 2011. Building high-level features using large scale unsupervised learning.

[49] Lecun, Y., Bottou, L., Bengio, Y., Haffner, P., 1998. Gradient-based learning applied to document recognition. In: Proceedings of the IEEE. pp. 2278–2324.

[50] Li, J., Cheng, K., Wang, S., Morstatter, F., Trevino, R. P., Tang, J., Liu, H., 2016. Feature selection: A data perspective. CoRR abs/1601.07996.
URL `http://arxiv.org/abs/1601.07996`

[51] Liu, B., Wang, M., Foroosh, H., Tappen, M. F., Pensky, M., 2015. Sparse convolutional neural networks. 2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 806–814.

[52] Maind, S., Wankar, P., 01 2014. Research paper on basic of artificial neural network 2, 96–100.

[53] maxeler technologies, 2011. Maxcompiler. Accessed: 2018-02-02.
URL `https://www.maxeler.com/media/documents/MaxelerWhitePaperMaxCompiler.pdf`

[54] Meyen, N., 2017. A survey of deep learning frameworks. Accessed: 2018-02-07.
URL https://towardsdatascience.com/a-survey-of-deep-learning-frameworks-43b88b11af34

[55] Mohamed, S., July 2015. A statistical view of deep learning combined pdf svdl.pdf.

[56] Murphy, J., 2016. Deep learning frameworks: A survey of tensorflow, torch, theano, caffe, neon, and the ibm machine learning stack. Accessed: 2018-02-07.
URL https://www.microway.com/hpc-tech-tips/deep-learning-frameworks-survey-tensorflow-torch-theano-caffe-neon-ibm-machine-learning-stack/

[57] Nagappan, S., 2016. Accelerating neural networks with binary arithmetic. Accessed: 2018-01-30.
URL https://software.intel.com/en-us/articles/accelerating-neural-networks-with-binary-arithmetic

[58] Nunes, N. E., 2017. Fpgas challenge gpus as a platform for deep learning. Accessed: 2018-01-29.
URL https://theintelligenceofinformation.wordpress.com/2017/04/03/fpga-chips-will-be-the-hardware-future-for-deep-leaning-and-ai/

[59] Nurvitadhi, E., Sheffield, D., Sim, J., Mishra, A. K., Venkatesh, G., Marr, D., 2016. Accelerating binarized neural networks: Comparison of fpga, cpu, gpu, and asic. 2016 International Conference on Field-Programmable Technology (FPT), 77–84.

[60] Nurvitadhi, E., Venkatesh, G., Sim, J., Marr, D., Huang, R., Ong Gee Hock, J., Liew, Y. T., Srivatsan, K., Moss, D., Subhaschandra, S., Boudoukh, G., 2017. Can fpgas beat gpus in accelerating next-generation deep neural networks? In: Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays. FPGA '17. ACM, New York, NY, USA, pp. 5–14.
URL http://doi.acm.org/10.1145/3020078.3021740

[61] Park, J., Sung, W., 2016. Fpga based implementation of deep neural networks using on-chip memory only. CoRR abs/1602.01616.
URL http://arxiv.org/abs/1602.01616

[62] Paszke, A., Chaurasia, A., Kim, S., Culurciello, E., 2016. Enet: A deep neural network architecture for real-time semantic segmentation. CoRR abs/1606.02147.
URL http://arxiv.org/abs/1606.02147

[63] Patterson, J., 2017. Deep learning : a practitioner's approach.

[64] Remy, P., 2016. Stateful lstm in keras. Accessed: 2018-05-23.
URL https://philipperemy.github.io/keras-stateful-lstm/

[65] Riera Molina, C. R., Pujol Vila, O., Dec. 2017. Solving internal covariate shift in deep learning with linked neurons. ArXiv e-prints.

[66] Romeu, P., Zamora-Martínez, F., Botella-Rocamora, P., Pardo, J., 2013. Time-series forecasting of indoor temperature using pre-trained deep neural networks. In: Mladenov, V., Koprinkova-Hristova, P., Palm, G., Villa, A. E. P., Appollini, B., Kasabov, N. (Eds.), Artificial Neural Networks and Machine Learning – ICANN 2013. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 451–458.

[67] Russell, S., 2016. Artificial intelligence : a modern approach.

[68] Samuel Gibbs, T. G., 2017. Alphazero ai beats champion chess program after teaching itself in four hours. Accessed: 2018-01-15.
URL `https://www.theguardian.com/technology/2017/dec/07/alphazero-google-deepmind-ai-beats-champion-program-teaching-itself-to-play-four-hours`

[69] Seide, F., Agarwal, A., 2016. Cntk: Microsoft's open-source deep-learning toolkit. In: Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. KDD '16. ACM, New York, NY, USA, pp. 2135–2135.
URL `http://doi.acm.org/10.1145/2939672.2945397`

[70] Simonyan, K., Zisserman, A., 2014. Very deep convolutional networks for large-scale image recognition. CoRR abs/1409.1556.
URL `http://arxiv.org/abs/1409.1556`

[71] Srivastava, N., Hinton, G. E., Krizhevsky, A., Sutskever, I., Salakhutdinov, R., 2014. Dropout: a simple way to prevent neural networks from overfitting. Journal of Machine Learning Research 15 (1), 1929–1958.
URL `http://www.cs.toronto.edu/~rsalakhu/papers/srivastava14a.pdf`

[72] Stiles, J., Jernigan, T., December 2010. The basics of brain development. Neuropsychology Review 20 (4), 327–348.

[73] Stone, J. E., Gohara, D., Shi, G., May 2010. Opencl: A parallel programming standard for heterogeneous computing systems. Computing in Science Engineering 12 (3), 66–73.

[74] Sze, V., Chen, Y., Yang, T., Emer, J. S., 2017. Efficient processing of deep neural networks: A tutorial and survey. CoRR abs/1703.09039.
URL `http://arxiv.org/abs/1703.09039`

[75] Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S. E., Anguelov, D., Erhan, D., Vanhoucke, V., Rabinovich, A., 2014. Going deeper with convolutions. CoRR abs/1409.4842.
URL `http://arxiv.org/abs/1409.4842`

[76] Theano Development Team, May 2016. Theano: A Python framework for fast computation of mathematical expressions. arXiv e-prints abs/1605.02688.
URL `http://arxiv.org/abs/1605.02688`

[77] Tompson, J., Goroshin, R., Jain, A., LeCun, Y., Bregler, C., 2014. Efficient object localization using convolutional networks. CoRR abs/1411.4280.
URL http://arxiv.org/abs/1411.4280

[78] Umuroglu, Y., Fraser, N. J., Gambardella, G., Blott, M., Leong, P. H. W., Jahre, M., Vissers, K. A., 2016. FINN: A framework for fast, scalable binarized neural network inference. CoRR abs/1612.07119.
URL http://arxiv.org/abs/1612.07119

[79] van den Oord, A., Dieleman, S., Zen, H., Simonyan, K., Vinyals, O., Graves, A., Kalchbrenner, N., Senior, A. W., Kavukcuoglu, K., 2016. Wavenet: A generative model for raw audio. CoRR abs/1609.03499.
URL http://arxiv.org/abs/1609.03499

[80] Varangaonkar, A., 2017. Top 15 deep learning frameworks. Accessed: 2018-02-07.
URL https://datahub.packtpub.com/deep-learning/top-10-deep-learning-frameworks/

[81] Wang, C., Yu, Q., Gong, L., Li, X., Xie, Y., Zhou, X., May 2016. Dlau: A scalable deep learning accelerator unit on fpga.

[82] world, I., 2018. 5 guidelines for building a neural network architecture. Accessed: 2018-04-24.
URL https://www.infoworld.com/article/3155052/technology-business/5-guidelines-for-building-a-neural-network-architecture.html

[83] Yu, F., Koltun, V., 2015. Multi-scale context aggregation by dilated convolutions. CoRR abs/1511.07122.
URL http://arxiv.org/abs/1511.07122

[84] Zhao, W., Fu, H., Luk, W., Yu, T., Wang, S., Feng, B., Ma, Y., Yang, G., July 2016. F-cnn: An fpga-based framework for training convolutional neural networks. In: 2016 IEEE 27th International Conference on Application-specific Systems, Architectures and Processors (ASAP). pp. 107–114.