Analysis, Design and Implementation Considerations of a Speech Coder Based on LPCNet

Group SPC-976
Aalborg University
Date: 4th of June 2020

AALBORG UNIVERSITY
DENMARK

Aalborg University
Department of electronic systems
Fredrik Bajers Vej 7
9220 Aalborg Øst
www.es.aau.dk

Title:

Analysis, Design and Implementation Considerations of a Speech Coder Based on LPCNet

Theme:

Signal Processing and Computing - Thesis

Project Period:

Autumn/Spring Semester 2019/2020
(50 ECTS)

Project Group:

SPC-976

Participants:

Christian Bekhoei Roskaer

Supervisor:

Peter Koch

Industry Partners:

Lars Soendergaard Bertelsen (RTX)
Christopher Meisner (RTX)

Pages: 138
Appendix: 12
Total: 157

Date of Completion: 4-June-2020

The contents of this report is freely accessible, however publication (with source references) is only allowed upon agreement with the authors.

Abstract:

In this project made in collaboration with RTX, we examine the newly proposed voice decoding neural network called LPCNet, with the intent of implementation on an embedded device. This work includes an analysis of the many methods and theories utilized by LPCNet. An error of the source code is corrected, and new models are trained using the TIMIT data set, yielding intelligible but unnatural sounding speech.
The part of the LPCNet with the highest computational complexity is selected, mainly consisting of two GRU layers. This sub-algorithm of the LPCNet is then further analysed, in order to gain insights of its inner workings. This analysis have among others resulted in the proposed Block Compressed Sparse Column (BCSC) format, as a means to address the block-sparse matrices utilized by the network.
In an FPGA based approach different data moment schemes where explored. In order to increase energy efficiency, while maintain the throughput of the solution. In a CPU based implementation different causes for processing stalls were investigated, and memory hierarchy was identified as the biggest reason for stalls of the sub-algorithm. Parallel processing have been emphasized in both approaches. Practical implementation results of these approaches have not been obtained within the time frame of the project.

Preface

This report is conducted by group SPC-976 attending the master thesis of the Signal Processing and Computing at Aalborg University.

Reading instructions:
This report is divided into chapters numbered by the order they appear. Sections and subsections are numbered as well the same way. Additionally, the report is divided into four main parts: Introduction, analysis of the neural network as an algorithm, Architectural considerations of the algorithm and lastly our final thoughts of these.

Figures, tables, equations, and algorithms are numbered by the chapter they appear in as well as the order. The fifth figure in chapter 4 will thereby be numbered as 4.5.

Appendices are found in the back of the report and will be named with the letter A and a given number, which will be referred to throughout the report.

The reference method will be a *harvard* style throughout the report. A list of the references is found at the end of the report, organized by author name reference in the report.

Example of a reference:

> "The embedding has earned - among other things - the function that converts the µ-law scale to linear" [Valin and Skoglund, 2019a]
>
> [Valin and Skoglund, 2019a] Valin, J.-M. and Skoglund, J. (2019a). LPCNET: Improving neuralspeech synthesis through linear prediction. In ICASSP 2019 - 2019 IEEE International Conferenceon Acoustics, Speech and Signal Processing (ICASSP). IEEE.

Acknowledgements

The author would like to thank Associate Professor Peter Koch for his supervision during the course of this project. The author would further like to thank RTX for their help and support during this project. Especially Lars Bertelsen which have provided many helpful discussions, and insights into the making of this report. Lastly would the author thank the Claaudia team, with the support given in regards to the training and simulation work performed with the research service.

Participants

Christian Bekhøi Roskær

Acronyms

AAU Aalborg University. 60
ACF Autocorrelation Function. 12, 13
ALU Arithmetic logic unit. 91, 113
APC All Partial sum Cleared. 102, 103
APE Accumulating Processing Elements. 101, 102, 103, 104
API Application Programming Interface. 59, 60

BCSC Block Compressed Sparse Column. ii, 85, 86, 88, 104, 106, 110, 122, 123, 124, 125, 127, 128, 130, 134, 135, 136, 137, 138
BFCC Bark Frequency Cepstral Coefficients. 152, 153
BPTT Back-Propagation Through Time. 26
BS Block Stationary. 96, 100, 104, 105

CNN Convolutional Neural Network. 92, 93
CORDIC Coordinate Rotation Digital Computer. 82, 107, 108
CPI Cycles Per Instruction. 113
CPU central processing unit. ii, 6, 60, 72, 73, 75, 76, 88, 89, 111, 112, 113, 114, 115, 117, 120, 125, 126, 127, 130, 131, 133, 134, 136, 137, 138, 154
CSC Compressed Sparse Column. 75, 84, 85, 86, 87, 121, 136
CSR Compressed Sparse Row. 38
CUDA Compute Unified Device Architecture. 60

DCT Discrete Cosine Transformation. 152, 153
DNN Deep Neural Network. 20, 72, 74, 75
DRAM Dynamic Random-Access Memory. 38, 72, 79, 91, 92, 93, 95, 98, 114
DSP Digital Signal Processor. 57, 88, 90, 112
DTFT Discrete-Time Fourier transform. 18

EOM End of Matrix. 102, 103
EOR End Of Row. 102, 103
EVD Eigenvalue Decomposition. 27
EWM Empty Weight Matrix. 102, 103

FC Fully connected. 47, 48, 49, 53, 54, 55, 56, 72, 73, 93
FFT Fast Fourier Transform. 74
FLOPS Floating Point Operations Per Second. 3, 4, 57, 72, 75, 113, 124, 135, 138

FPGA Field Programmable Gate Array. ii, 6, 57, 72, 73, 74, 75, 76, 88, 89, 90, 91, 97, 99, 100, 110, 111, 121, 122, 133, 134, 136, 137, 138, 154
FRN Frame Rate Network. 4, 63

GFLOPS Giga Floating Point Operations Per second. 3
GPU Graphic Processing Unit. 40, 60, 72, 73, 75, 88, 154
GRU Gated Recurrent Unit. ii, 26, 31, 32, 34, 35, 36, 37, 38, 43, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 63, 65, 73, 74, 75, 76, 77, 79, 80, 81, 82, 83, 85, 87, 88, 92, 93, 94, 95, 100, 103, 104, 106, 107, 108, 110, 112, 114, 115, 117, 118, 119, 121, 124, 125, 126, 127, 128, 129, 130, 134, 135, 136, 137

HPC High-Performance Computing. 60

IDFT Inverse Discrete Fourier Transform. 153
ITU-T International Telecommunication Union, Telecommunication standardization sector. 2, 62, 150

LPC Linear Predictive Coding. 3, 4, 7, 9, 16, 17, 18, 34, 42, 49, 57, 61, 72, 135, 147, 152, 153
LSTM Long Short-Term Memory. 27, 28, 29, 30, 31, 32, 58, 74, 80, 88
LTI Linear Time Invariant. 8, 18
LUT Look-Up Table. 37, 43, 50, 73, 78, 79, 108, 117

MAC Multiplier Accumulator. 57, 75, 90, 91, 93, 101, 102, 103, 113, 117, 118, 123, 124, 125, 127, 128, 130, 135, 136, 138
MFCC Mel Frequency Cepstral Coefficients. 153
MIMD Multiple instruction Multiple Data. 154
MLP Multilayer Perceptrons. 19, 26
MMU Memory Management Unit. 114, 124
MOS Mean Opininon Score. 3, 4, 62, 63, 69, 138
MSB most significant bit. 150
MSE Mean-Squared Error. 10

NINE Non-Inclusive Non-Exclusive. 114
NLR No Local Reuse. 91, 92, 93, 104, 105, 110
NNZ Number of Nonzero. 84

OS Output Stationary. 91, 92, 95, 96, 104, 105, 110

PCM Pulse-Code Modulation. 61, 62
PE Processing Elements. 73, 90, 91, 92, 93, 95, 96, 97, 98, 99, 100, 101, 104, 105, 108, 109, 110, 121
PESQ Perceptual Evaluation of Speech Quality. 62, 63
PSD Power Spectral Density. 153
PU Processing Unit. 154

ReLU Rectified Linear Unit. 20
RF Register File. 90, 91, 92, 93, 95, 96, 97, 99, 100, 102, 104, 108, 109
RISC Reduced Instruction Set Computing. 112
RNN Recurrent Neural Network. 3, 4, 6, 19, 23, 24, 25, 26, 27, 28, 29, 31, 32, 58, 73, 88, 92
RS Row Stationary. 91, 92, 93, 96
RTL Register Transfer Level. 90

SGD Stochastic Gradient Descent. 21, 22
SIMD Single Instruction Multiple Data. 72, 74, 75, 76, 88, 112, 115, 117, 127, 128, 134, 136, 137, 138, 154
SOX Sound eXchange. 61
SRAM Static Random-Access Memory. 38, 91, 95, 97
SRN Sample Rate Network. 4

WAV Waveform Audio File Format. 61
WMOPS Weighted Milion Operations Per Second. 3
WPE Weight Processing Elements. 100, 101, 102, 103, 104, 105, 106
WS Weight Stationary. 91, 92
WSS Wide Sense Stationary. 11, 13, 14

Contents

Part I

Introduction

Chapter 1

Introduction

1.1 Introduction

In the last decade we have seen a boom in use cases utilizing machine learning in forms of neural networks, including subjects such as image and audio recognition, user recommendation systems, financial predictions etc. One reason for this boom in machine learning is due to the growing computational power, allowing the general public to set up and train these neural networks. Which in some cases vastly outperforms traditional solutions, this generally comes at the cost of a higher complexity compared to traditional solutions.

While high complexity networks have become more manageable, it is not always possible or favorable. This is typically a concern in low energy systems or systems using batteries, where the amount of bits needed to be processed is linked to the power consumption of the given system.
One category of low power systems are wireless communication equipment. Often will these systems have a battery on board, and this battery's capacity along with the application running on the device sets the limit on time between recharges and how much power the application is allowed to draw at a given time.

One such subject used in low energy communication systems are voice coders (vocoders). "Vocoders is a category of voice codec that analyzes and synthesizes the human voice signal for audio data compression, multiplexing, voice encryption, voice transformation, etc"[Voc, 2019]. A significant part and issue of the vocoder is the compression of data. Since data compression may lead to loss of audio intelligibility. Examples of such audio codecs is the International Telecommunication Union, Telecommunication standardization sector (ITU-T) standards G719 and G722, which among other things compress data significantly, down to bit rate around 32 to 128 kbit/s compared to the some what standard 256 kbit/s for a mono system (16bit depth at 16 kHz). This compression have among others the advantage of requiring less bandwidth within the frequency band it operates, and hence allows more traffic within the band. Second does the reduced amount of data to be wireless transmitted, yield a lower power computation, as less information have to be conveyed. This comes at the cost of poorer audio quality (intelligibility) if lossy compression is used, and additional computational complexity, since these compression schemes are often decompressed once received.

With the introduction of neural network, neural speech synthesis algorithms have recently made it possible to both synthesize high-quality speech and code high quality speech at very low bit rates. The

usage of WaveNet [Aäron van den Oord, 2016] in [Kleijn et al., 2018] allowed to match the bit rate of the vocoders 'codec 2' and 'MELP' of 2.4 kbit/s while having a MOS score of 2.9, as compated to 2.7 and 2.9 by codec 2 and MELP respectively, the Mean Opininon Score (MOS) is a measure of a systems quality, ranging from 1 which is defined as 'bad' to 5 which is defined to be excellent'.

This comes at a cost though, it is estimated that the WaveRNN [Kalchbrenner et al., 2018] (a follow up to WaveNet), has an estimated complexity of around 10 Giga Floating Point Operations Per second (GFLOPS) [Valin and Skoglund, 2019a]. Comparing this to the G.719 codec which has a complexity of 15.397 to 21.000 Weighted Milion Operations Per Second (WMOPS) [Xie et al., 2009] - WMOPS is defined in ITU-T G.191, here the complexity of a given operation is weigted based on the number of cycles it takes to compute. The WaveRNN is significantly more complex, making it hard or unfeasible to run in real time on less powerful devices such as mobile phones or low energy system.

The LPCNet seeks to further improve the complexity of the WaveNet, by the means of introducing Linear Predictive Coding (LPC), pre/de-emphasis filters, sparse matrices and input embedding. Reaching a complexity of around 2.8 GFLOPS, low enough to achieve real-time synthesis can be achieved on a single core of an Apple A8 (iPhone 6) [Valin and Skoglund, 2019a]. Making it a candidate for implementation on other small devices.

1.2 Introduction to LPCNet

The LPCNet is based on layers of different neural networks, and draws inspirations from others. Thus we will outline the advancement of these networks, in order to give an overview of the development of how this networks came to be.

1.2.1 Overview of audio generative neural networks

The first of these neural networks capable of generating raw audio wave forms where the WaveNet [Aäron van den Oord, 2016]. It was inspired by PixelCNN [van den Oord et al., 2016] and [Jozefowicz et al., 2016], networks able to generate images or words. In WaveNet they used dilated convolutions and conditional probability in order to general wideband (16kHz samples per second) audio waveforms for text-to-speech purposes. The WaveNet achieved a MOS of 4.21 apporaching 8-bit μ-law encoded speech, which was reported to have a MOS of 4.46 [Aäron van den Oord, 2016].

In [Soroush Mehri, 2017] they instead make unconditional audio generation, based on Recurrent Neural Network (RNN) in a hierarchical structure name SampleRNN. They reported a better performance than the WaveNet by subjective comparison, and [Valin and Skoglund, 2019a] estimated the complexity to be around 50 GFLOPS.

In [Kleijn et al., 2018] they used the WaveNet to generate speech at 2.4 kb/s using the Codec 2 for parametric coding. They reported a MOS of 2.9 for their 2.4kb/s WaveNet coder, codec 2 and MELP were reported to have 2.7 and 2.9 respectively. On the subjective quality the reported to have a MUSHRA (ITU-R BS.1534-3) score of around 71 (of 100) compared to MELP's score of around 26 and Codec 2's around 20. Complexity measures have not been estimated.

In [Jin et al., 2018] they are inspired by the WaveNet. They instead made use of 1×1 convolutions so called *conv1x1*, in order to achieve an FFT like structure, but having resembles to the dilated convolution used in the WaveNet. The FFTNet furthermore used conditional sampling, like WaveNet, but differs on the fact that they no longer base the sampling on maximizing the log-likelihood as

WaveNet did. Instead they will randomly sample from the sample probability distribution generated by the last layer in FFTNet, furthermore do they distinguish between voice and unvoiced sounds. Additionally do they also perform noise injection during the training phase of the network. They report a MOS similar to that of WaveNet, and [Valin and Skoglund, 2019a] estimates their complexity to around 16GFLOPS.

WaveNet is then further improved in the work made WaveRNN [Kalchbrenner et al., 2018]. Here they make use of RNN, with a so called dual softmax layer, which separates a sample into coarse (highest 8-bits) and fine (lowest 8-bits) gain samples, and use weight pruning, which is a way to achieve sparse weight matrices. They report a performance close to similar to WaveNet with a MOS of 4.33 for their mobile version, compare to their trained WaveNet's MOS of 4.51. In [Valin and Skoglund, 2019a] they estimate that the sparse mobil version have a complexity of around 10 GFLOPS.

Lastly we have the LPCNet from [Valin and Skoglund, 2019a]. Here they combine knowledge gained from among others these prior networks, build on top of it with their own additions. This includes a the introduction of using Linear Predictive Coding (LPC) within the network, and other methods inspired by these ealier networks. They report a MUSHRA score better than the WaveRNN with all models used having equal size to WaveRNN, and estimates that the complexity of their network to be around 3 GFLOPS. Additional did they further develop a 1.6 kbit/s bit rate vocoder, based on the LPCNet and some clever frame package schemes, which they report to have higher scores than the MELP and Speex vocoders, which have a bit rate of 2.4 and 4 kbit/s respectively.

Thus we have seen the development, from a simpler yet very computational expensive networks which functioned un-parametric. Towards very complex parametric networks, with much lower complexity. With the networks 'history' known we will present the broad outline of the neural network that is the LPCNet.

1.2.2 Introduction to the LPCNet architecture

Here we will give a brief outline of the network and its broad functionality, theory and methods as well as in depth descriptions of this layers will be given in chapter 2, chapter 3 and chapter 4, the LPCNet can be seen in Figure 1.1 along with all its layers. LPCNet is the decoder part of a combined vocoder, it receives a total of 20 features, 18 of which are Bark-scale [Fastl and Zwicker, 2007] cepstral coefficients, and two pitch parameters.

These features is given in frames covering a duration of 10 ms, hence is 20 features received for each frame every 10 ms. The network consists of three parts; the Linear Predictive Coding (LPC), the Frame Rate Network (FRN) and Sample Rate Network (SRN). The FRN compute on multiple prior frames and the current frame and operates on frame basis, while the SRN operates at 16kHz, one sample at a time.

The FRN produces a vector $\mathbf{f}$ once every frame. A total of the latest 16 outputs is used by the LPC part of the network in order to produce a new prediction value $p^{(t)}$, for every sample.

The SRN produces wideband speech, hence it produces 16,000 samples every second. The SRN is given the vector $\mathbf{f}$ every frame, and the linear prediction $p^{(t)}$, prior output $S^{(t-1)}$ and exictation $e^{(t-1)}$ every time step. Hence do the SRN produce 160 samples (10 ms speech) for every vector $\mathbf{f}$, given by the FRN.

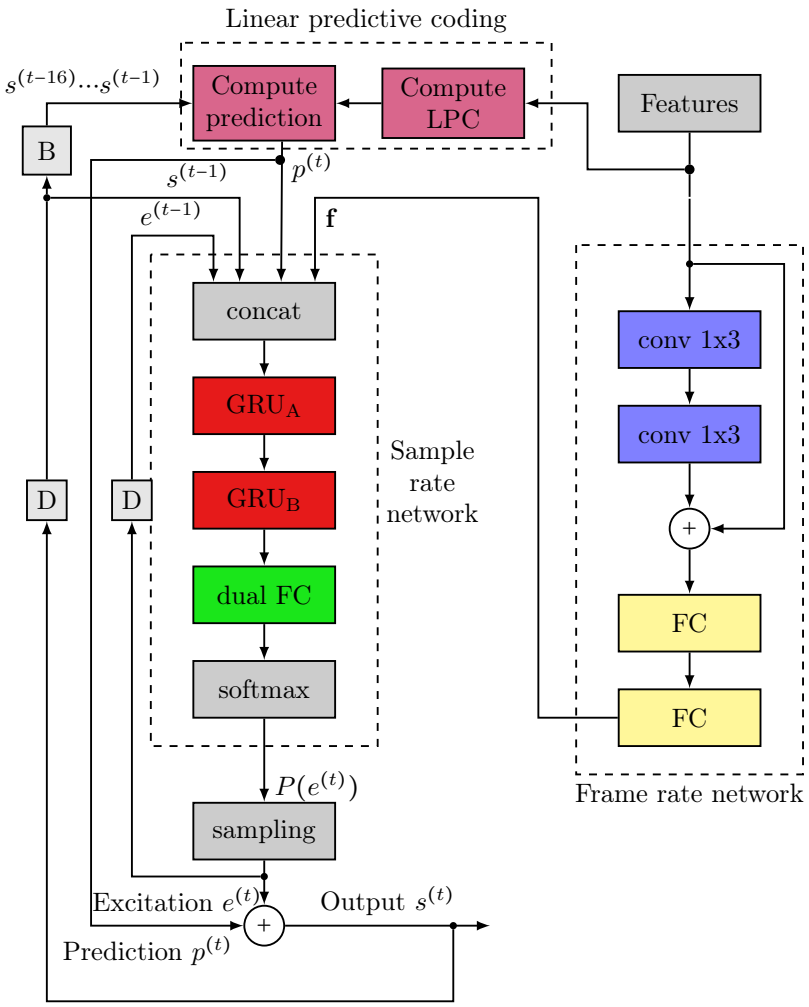


Figure 1.1: Overview of the LPCNet, modified figure of [Valin and Skoglund, 2019a]. 'D' denotes a delay of sample and 'B' denotes a buffer

Thus will the network handle 100 frames each second, and produce 16,000 samples as a result. Hence will it require a bit rate of 2000·BD per second, where BD denotes the bit depth of the features. For example if the features have a bit depth of 8 bits, will the network operate at a bit rate of 16 kbit/s.

Having a very sparse insight into the network, we will move on to the scope of this project, and return to theory and methods use within the network in the coming chapters.

1.2.3 Collaboration

This project is made in collaboration with the company RTX. RTX specializes in short range radio communications which are mainly focus on audio. This includes pro-audio, intercom systems, healthcare systems, audio codecs and wireless gaming equipment. Typically working in the 1.9, 2.4 and 5.8 GHz bands. Many of these systems are battery powered devices with requirements to the latency, in the ranges of microseconds to a few milliseconds. The ISM-bands is of cause one of the more crowded frequency bands to operate in, therefore can the LPCNet be of interest since its later development resulted in a newly proposed 1.6 kbit/s bit rate vocoder, based around the LPCNet in [Valin and Skoglund, 2019b].

RTX's main concern regarding the LPCNet is its very large complexity, compared to other more traditional solutions, and how to handle this on the typical low power hardware used in the voice encoding environment.

Thus this project will mainly be focused on investigating, what the LPCNet algorithm requires in terms of complexity and power consumption, and if optimizations are possible. Therefore metrics regarding numerical precision and area, are not first priority. They are by all means also of importance, but not if an implementation is infeasible complexity or power consumption wise.

1.3 Thesis

Having outline the background for RTX's interest in the LPCNet, along with a short description of the network it self. We will here describe our main points of interest regarding the network;

- What can the neural network 'LPCNet' achieve in regards to speech quality and what is the complexity of this network?
- Which hardware resources are suitable with considerations in regards to power consumption, size and similar?
- An implementation of the given LPCNet on suitable hardware with follow up tests.

Hence these will the areas of which we will try to clarify. Leading us the scope of this project.

1.4 Scope of project

This project has the purpose of conducting a feasibility and recommendations study of this newly proposed LPCNet speech decoder. This will include a theoretical walk-through of the methods used in the LPCNet. Thus the walk-through will include linear prediction and the Recurrent Neural Network (RNN), which is the type of neural network LPCNet and WaveRNN utilizes. Additional methods will be presented and examined when needed

Secondly we will analyse the different layers of the LPCNet and their functionality, including estimates of the networks computational complexity. Having understood the functionality and dataflow of the network, we set out to train out own network models based on the source code provided by the LPCNet proposers [Valin and Skoglund, 2019a]. This include the necessary pre- and post-processing of data, and performance evaluation of the trained models.

A part of the LPCNet will then be selected based on complexity, novelty and general re-usability of the selected part in other solutions. This part will be denoted as the sub-algorithm of the LPCNet. The sub-algorithm will serve as a demonstration of an implementation, and thus a deeper analysis of the sub-algorithm is to be made, in order to highlight information crucial for its implementation.

Based on this analysis of the sub-algorithm, will two approaches be described related to the implementation of the sub-algorithm on a CPU or FPGA based system. Based on describing these two approaches will result in recommendations for implementing this sub-algorithm and parts of the LPCNet sharing functional similarities, along the concluding remarks of this project.

Chapter 2

Linear Predictive Coding

In Linear Predictive Coding (LPC) the underlying basis for its derivation, is the assumption that for a given input signal v[n], it is possible through an LTI system with transfer function $H(z)$ followed by the inverse system $H^{-1}(z)$, to achieve the same output as input. Which in principal is true, but in reality will almost all ways lead to an approximation of the input. This is illustrated in Figure 2.1.

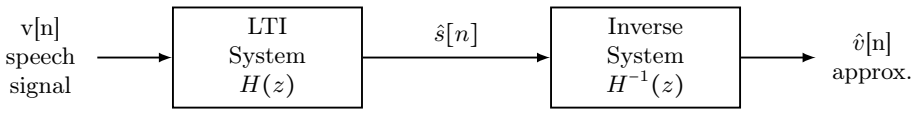


Figure 2.1: Underlying system

The reason for doing this had earlier among other things, been that through linear prediction it is possible to save significantly on the bit rate required to transmit audio and voice data. In short this is done by transmitting filter parameters instead of the actual audio samples. In LPCNet it is done such that the neural network is only required to model a spectrally flat excitation. Where in the opposite the neutral network has to model the spectral envelope [Valin and Skoglund, 2019a].

In this chapter we will examine some of the essential details of LPC. The examination of LPC relies heavily on chapter 11 in the educational book [Oppenheim and Schafer, 2013], since it gives an intuitive explanation of the subject of parametric signal modeling, and the main points of interest is shown here. Hence this chapter will give a similar presentation to that of 11 in [Oppenheim and Schafer, 2013]. Some additional points are taken from the well known article [Makhoul, 1975] on the subject of LPC. Any statements not from these sources is referenced within the text.

2.1 Parametric signal modeling

Using a parametric signal model, we make assumptions that leads to a signal model which can be represented based on a number of parameters. Hence we try to represent a given signal $s[n]$ by using the parameters, which will result in the models approximation of this signal $\hat{s}[n]$. We then wish to chosen these paramters such that we minimize the difference between true signal $s[n]$ and our model $\hat{s}[n]$. This is the initial part of figure Figure 2.1 as shown in Figure 2.2.

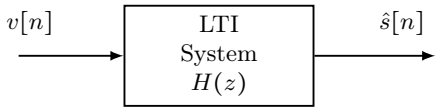


Figure 2.2: Linear system model - from [Oppenheim and Schafer, 2013].

For example in filter theory we often assume that a signal can be transformed by the usage of a Linear Time Invariant (LTI) system, with a tranferfunction consisting of polynomials in the form:

$$H(z) = \frac{\sum_{k=0}^{q} b_k z^{-k}}{1 - \sum_{k=1}^{p} a_k z^{-k}} \tag{2.1}$$

hence is the signal modeled by the transfer function $H(z)$ with belonging the poles and zeros. The frequency response of the system is of course depending on the model used. One model of speech signal used is the lossless tube discrete-time model [Rabiner and Schafer, 1978]. Here it is shown that speech can approximately be modeled in discrete-time by an all-pole filter. In this speech model the signal input $v[n]$ is assumed to be either an impulse δ[n] for deterministic signals or white noise for random signals.

The deterministic and white noise signals is used in the two following classes. The first class is so called voiced speech is used to model formants, these are typically linked to vowels and their harmonics, this is the deterministic input signal. The second class is unvoiced speech and is used to model sounds that are more noise like, these sounds a typical linked to consonant sounds such as 'sss' which is the white noise input signal. A simplistic speech production model based on [Rabiner and Schafer, 1978] is shown in Figure 2.3.

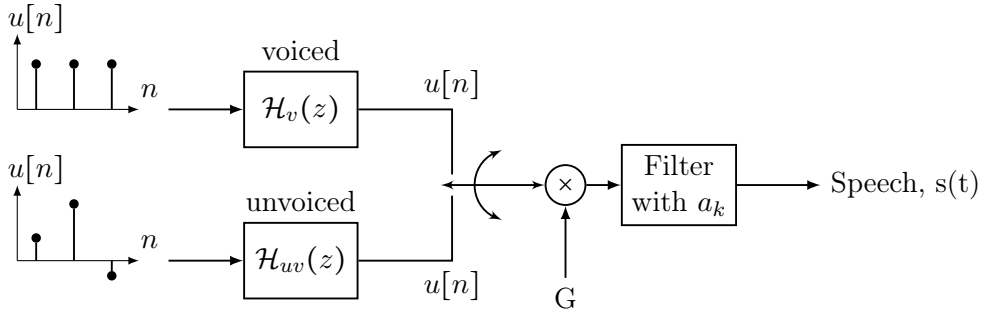


Figure 2.3: Simple speech production model.

The deterministic voiced speech signal is modeled as a train of impulses with a given period to produces periodic sounds. Unvoiced speech is modeled as white Gaussian noise to produces noise-like sounds. These are used as input to the time varying LTI filter which consists of a_k poles.

2.1.1 All-pole model of signal

In the all-pole model q in Equation 2.1 have been set to zero, and instead the zeros b_0 is replaced with letter G since it serves as a gain factor, hence we have that Equation 2.1 becomes.

$$H(z) = \frac{G}{1 - \sum_{k=1}^{p} a_k z^{-k}} \tag{2.2}$$

$$= \frac{G}{A(z)} \tag{2.3}$$

Typically is the order of the filter, chosen to be $p = 12$ for voice signals [Kay, 2012]. The difference equation of this filter can be seen in 2.4. This form of equations is also known as an autoregressive (AR) random process in statistical literature.

$$\hat{s}[n] = \sum_{k=1}^{p} a_k \hat{s}[n-k] + Gv[n] \tag{2.4}$$

From the equation it can be seen that the signal $\hat{s}[n]$ can be approximated as a linear combination of previous outputs and a scaled version of the current input. It will regularly be seen that this type of modeling a speech signal is referred to as Linear Predictive Coding (LPC). Since we try to predict the new output by linearly scaled values.

Thus the goal for this all-pole filter modeling is to determine whether the input $v[n]$ is voiced or unvoiced, the correct gain G and the coefficients a_k, such that the resulting output $\hat{s}[n]$ of the filter $H(z)$ is a close approximation of the desired output $s[n]$. Typically is $v[n]$ already specified in the means of whether its voiced or unvoiced. Thus the remaining task is to choose the gain G and the filter coefficients a_k, note that in linear prediction these coefficients are often referred to as *prediction coefficients*.

2.2 Least-squares inverse model

The inverse model build around the assumption that if we can make a filter $H(z)$ such that its output is indeed is $s[n]$, as shown in Figure 2.2. Then if we make an inverse filter assuming $H^{-1}(z) = A(z)$ and use $s[n]$ as input to this filter. The output of this inverse filter should be a scaled version of $v[n]$ by the gain factor G. Such that the output of the inverse filter $g[n]$ is equal to $Gv[n]$ as shown in Figure 2.4. Hence we have that Equation 2.4 becomes:

$$Gv[n] = \hat{s}[n] - \sum_{k=1}^{p} a_k \hat{s}[n-k] \tag{2.5}$$

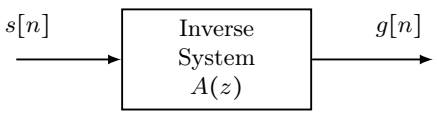


Figure 2.4: Inverse filter formulation for all-pole model - from [Oppenheim and Schafer, 2013].

again is the inverse filter of the all-pole filter $A(z)$ which then have the system function of.

$$A(z) = 1 - \sum_{k=1}^{p} a_k z^{-k} \tag{2.6}$$

Thus we seek the parameters of the inverse filter $A(z)$, which minimizes the Mean-Squared Error (MSE), between $g[n]$ and $Gv[n]$. Hence taking a close look at the difference equation for $g[n]$.

$$g[n] = s[n] - \sum_{k=1}^{p} a_k s[n-k] \tag{2.7}$$

It is now possible to define the modeling error $\hat{e}[n]$ as.

$$\hat{e}[n] = g[n] - Gv[n] \tag{2.8}$$

Thus $\hat{e}[n]$ is the error between the correct signal $s[n]$ and the linear prediction of the signal $\hat{s}[n]$ using the model parameters. The prediction error $e[n]$ can then be helpfully expressed as.

$$e[n] = s[n] - \sum_{k=1}^{p} a_k s[n-k] \tag{2.9}$$

Hence we see that the inverse filters output $g[n]$ is equal to the prediction error $e[n]$ by comparing Equation 2.7 and 2.9. For a signal which belongs to the all-pole model precisely, the modelling error $\hat{e}[n]$ would be equal to zero, meaning that the prediction error $e[n]$ (or $g[n]$) is equal to the scaled input $Gv[n]$.

$$\hat{e}[n] = e[n] - Gv[n] \tag{2.10}$$

Since $v[n]$ is assumed to be known and $e[n]$ can be computed based on the input signal $s[n]$, the coefficients a_k should be chosen to minimize the total energy in the modeling error $\hat{e}[n]$. As shown Equation 2.11.

$$\varepsilon = \left\langle |\hat{e}[n]|^2 \right\rangle \tag{2.11}$$

Where in [Oppenheim and Schafer, 2013] uses $\langle \cdot \rangle$ to denote a summing operation for finite energy deterministic signals hence:

$$\varepsilon_{det} = \sum_{n} \hat{e}^2[n] = \sum_{n} (s[n] - \sum_{k=1}^{p} a_k s[n-k] - Gv[n])^2 \tag{2.12}$$

and an ensemble averaging operation for random signals:

$$\varepsilon_{rnd} = \frac{1}{N} \sum_{n=1}^{N} \hat{e}^2[n] \tag{2.13}$$

and so Equation 2.11 minimizes either the total energy for a deterministic or the mean-squared value for random signal, in the modeling error. By determining the prediction coefficient which makes the prediction error equal to the scaled input. To find the optimal prediction values a_k and thus minimize ε in least squares sense. We differentiate w.r.t the i^{th} filter coefficient a_i and set the derivative equal to zero. We start by expanding Equation 2.11 by Equation 2.8.

$$\varepsilon = \left\langle (e[n] - Gv[n])^2 \right\rangle \tag{2.14}$$

$$= \left\langle e^2[n] + (Gv[n])^2 - 2Ge[n]v[n] \right\rangle \tag{2.15}$$

$$= \left\langle e^2[n] \right\rangle + G^2 \left\langle v^2[n] \right\rangle - 2G \left\langle e[n]v[n] \right\rangle \tag{2.16}$$

Using the linearity of this $\langle \cdot \rangle$ summing or averaging operator. We then take the derivative w.r.t the i^{th} filter coefficient a_i and set it equal to zero, we have:

$$\frac{\partial \varepsilon}{\partial a_i} = \frac{\partial}{\partial a_i} \left[\left\langle e^2[n] \right\rangle + G^2 \left\langle v^2[n] \right\rangle - 2G \left\langle e[n]v[n] \right\rangle \right] = 0 \tag{2.17}$$

we assume that both the gain G and input $v[n]$ are independent from these filter coefficients a_i, that is:

$$\frac{\partial}{\partial a_i} \left[G^2 \left\langle v^2[n] \right\rangle \right] = 0 \tag{2.18}$$

which leaves us with the following.

$$\frac{\partial \varepsilon}{\partial a_i} = \frac{\partial}{\partial a_i} \left[\left\langle e^2[n] \right\rangle - 2G \left\langle v[n]e[n] \right\rangle \right] = 0 \tag{2.19}$$

$$= \frac{\partial}{\partial a_i} \left[\left\langle e^2[n] \right\rangle - 2G \left\langle v[n]s[n-i] \right\rangle \right] = 0 \qquad i = 1, 2...p \tag{2.20}$$

In the speech production model we are interested in, it is assume that $v[n]$ will be an impulse if $s[n]$ is a causal finite-energy signal and white noise if $s[n]$ is a Wide Sense Stationary (WSS) random process.

Meaning that for the deterministic case, the product of $v[n]s[n-i] = 0$ for $i = 1, 2...p$. Since $s[n] = 0$ for $n < 0$, with $v[n]$ being an impulse.

In the random case is $\langle v[n]s[n-i] \rangle = 0$ for $i = 1, 2...p$ since the input is assumed to be white noise, thus the white noise input is uncorrelated with the output of the filter prior to time n, for a causal system. Thus Equation 2.19 is reduced to.

$$\frac{\partial \varepsilon}{\partial a_i} = \frac{\partial}{\partial a_i} \left\langle e^2[n] \right\rangle = 0 \qquad i = 1, 2...p \tag{2.21}$$

Is can be seen that selecting these prediction coefficients a_k to minimize the energy ε, which was equal to the averaged squared modeling error ($\langle \hat{e}^2[n] \rangle$) in Equation 2.11. Is the same as minimizing the $\langle e^2[n] \rangle$, i.e. the averaged squared prediction error for the signal which is of interest to us. If we expand this Equation 2.21 by inserting Equation 2.9. we have:

$$\frac{\partial}{\partial a_i} \left\langle \left(s[n] - \sum_{k=1}^{p} a_k s[n-k] \right)^2 \right\rangle = 0 i = 1, 2...p \tag{2.22}$$

which can further be expanded into:

$$\frac{\partial}{\partial a_i}\left\langle\left(s^2[n] - 2s[n]\sum_{k=1}^{p} a_k s[n-k] + \left(\sum_{k=1}^{p} a_k s[n-k]\right)^2\right)\right\rangle = 0 \tag{2.23}$$

Using the linearity of derivative and the averaging operator $\langle\cdot\rangle$, we then get:

$$\frac{\partial}{\partial a_i}\left\langle s^2[n]\right\rangle - \frac{\partial}{\partial a_i}\left\langle 2s[n]\sum_{k=1}^{p} a_k s[n-k]\right\rangle + \frac{\partial}{\partial a_i}\left\langle\left(\sum_{k=1}^{p} a_k s[n-k]\right)^2\right\rangle = 0 \tag{2.24}$$

Computing the derivative of this terms we have:

$$-\left\langle 2s[n]s[n-i]\right\rangle + \left\langle 2\sum_{k=1}^{p} a_k s[n-k]s[n-i]\right\rangle = 0 \tag{2.25}$$

Finally rearranging this yields:

$$\sum_{k=1}^{p} a_k \left\langle s[n-k]s[n-i]\right\rangle = \left\langle s[n]s[n-i]\right\rangle \tag{2.26}$$

Hence we have the Autocorrelation Function (ACF) of $s[n]$ on the right hand side of the equation. If we introduce the notation use in [Oppenheim and Schafer, 2013]:

$$\phi_{ss}[i,k] = \left\langle s[n-k]s[n-i]\right\rangle \tag{2.27}$$

we have that Equation 2.26 can be rewritten as.

$$\sum_{k=1}^{p} a_k \phi_{ss}[i,k] = \phi_{ss}[i,0] \tag{2.28}$$

Thus we have arrived at a set of p number of linear equations with p number of unknowns, which is sometimes expressed as $\Phi a = \Psi$. We can then solve these linear equations by using the known values of $\phi_{ss}[i,k]$ for both i and k. There exist different methods for solving this problem, with different assumptions regarding which range ε is minimized over, which we will return to in section 2.5.

To summarize we can now consider the prediction filter. Which is referred to as the p^{th} order *linear predictor* for the signal $s[n]$.

$$P(z) = \sum_{k=1}^{p} a_k z^{-k} \tag{2.29}$$

Its output is.

$$\tilde{s}[n] = \sum_{k=1}^{p} a_k s[n-k] \tag{2.30}$$

From here we have compute the difference between the prediction $\tilde{s}[n]$ and the actual signal $s[n]$ as $e[n] = s[n] - \tilde{s}[n]$. The difference is called the *prediction error residual* or just the *residual*. In this sense the coefficients a_k is called the *prediction coefficients*. In Figure 2.5 the relation between the linear predictor and the prediction error filter is depicted.

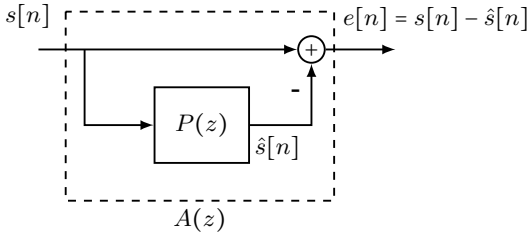


Figure 2.5: Prediction error filter - from [Oppenheim and Schafer, 2013]

2.3 Deterministic and random signal models

The reason to separate the models into two cases, a deterministic and a random, is because this affects the model differently. Yet it can be shown that with the assumption that in the case of a random signal, if its zero mean and WSS, then both cases can be solved by the Yule-Walker equations. In both cases we assume that we have knowledge of the signal over all time i.e $-\infty < n < \infty$ first let us consider the deterministic case.

2.3.1 Deterministic signals

For the case of a deterministic signal, we shall assume that the all-pole model is both causal and stable. We saw that with the use of the operator $\langle \cdot \rangle$ in Equation 2.11, which denoted a summing operation for a deterministic signal, we arrived at Equation 2.26 using the assumption that we have knowledge of the signal in time $-\infty < n < \infty$ Equation 2.27 becomes.

$$\phi_{ss}[i,k] = \sum_{n=-\infty}^{\infty} s[n-k]s[n-i] \tag{2.31}$$

$$= \sum_{n=-\infty}^{\infty} s[n]s[n-(i-k)] \tag{2.32}$$

$$= r_{ss}[i-k] \tag{2.33}$$

Which means that for real signals $s[n]$, the coefficients of $\phi_s[i,k]$ is the deterministic ACF.

$$r_{ss}[m] = \sum_{n=-\infty}^{\infty} s[n+m]s[n] \tag{2.34}$$

Using this in Equation 2.28 we get.

$$\sum_{k=1}^{p} a_k r_{ss}[i-k] = r_{ss}[i] \qquad i = 1,2...p \tag{2.35}$$

Thus we have the autocorrelation normal equation also known as Yule-Walker equations. Based on these we can compute the coefficients of a_k from the autocorrelation of the signal.

2.3.2 Random signals

In the case of a random signal, the input $s[n]$ to the filter $A(z)$ is assumed to be white noise. We therefore wish to make the output $g[n]$ of the inverse filter $A(z)$ as much like white noise as possible (instead of the impulse train used in the deterministic case) this filter is often referred to as a whitening filter. With the choice of $\langle\cdot\rangle$ operator denoting a mean-squared error, for the case of a random signal, Equation 2.11 becomes.

$$\varepsilon = E[(\hat{e}[n])^2] \tag{2.36}$$

Now if the signal $s[n]$ is assumed to be samples from a WSS process, then the coefficients of $\phi_{ss}[i,k]$ becomes the autocorrelation function i.e.

$$\phi_{ss}[i,k] = E[s[n-i]s[n-k]] = r_{ss}[i-k] \tag{2.37}$$

Meaning that Equation 2.28 again for the random signal becomes.

$$\sum_{k=1}^{p} a_k r_{ss}[i-k] = r_{ss}[i] \qquad i = 1,2...p \tag{2.38}$$

Then the parameters a_k can again be computed by the use of Yule-Walker equations, but in this case the autocorrelation function is based on the expected value of the signal.

2.3.3 Minimum mean-squared error

Hence when computing the minimal value of the prediction error $e[n]$, we take basis in Equation 2.28, and the solution to these depended on how the $\langle\cdot\rangle$ operator was used in the deterministic or random case. yet as we seen in Equation 2.35 and 2.38 results in the same equations, the difference being whether the autocorrelation is deterministic or that of a random process. In the general case this is equal to compute the minimal value of:

$$\varepsilon = \phi_{ss}[0,0] - \sum_{k=1}^{p} a_k \phi_{ss}[0,k] \tag{2.39}$$

and when using the averaging operation ($\langle\cdot\rangle$) as presented, Equation 2.39 becomes.

$$\varepsilon = r_{ss}[0] - \sum_{k=1}^{p} a_k r_{ss}[k] \tag{2.40}$$

2.4 Computation of gain

The auctocorrelation matching property [Makhoul, 1973], refers to the property that the solution for the deterministic signal Equation 2.35 and the random signal Equation 2.38 is the same, hence the autocorrelation coefficients of the output of an all-pole filter are the same. Thus we seek a gain, that in the deterministic case matches the total energy in the model output to the total energy of the signal that we are trying to model, and in the case of the random signal a gain that matches the average energy. In both cases it corresponds to choose a gain such that the autocorrelation of the signal $\hat{s}[n]$ becomes equal the autocorrelation of $s[n]$. It has been shown in [Makhoul, 1975] that the gain for both cases can be computed as:

$$G^2 = \varepsilon = r_{ss}[0] - \sum_{k=1}^{p} a_k r_{ss}[k] \tag{2.41}$$

Hence we see that the gain squared is equal to ε, the minimal total energy in the modeling error $|\hat{e}[n]|^2$ squared.

2.5 Correlation estimation and data windowing

As stated earlier, for the solution of the set of linear equations in Equation 2.28, we have to make an assumption regarding the range of the signal $s[n]$, this is also referred to as data windowing. In Generally two approaches (or windows) are used to compute the correlation functions $\phi_{ss}[i,k]$, these two methods are known as the covariance method and the autocorrelation method.

2.5.1 The autocorrelation method

In the autocorrelation method we assume that the signal have infinite range i.e. $-\infty < n < \infty$, and that signal samples outside the available range $0 \leq n \leq N$ is equal to zero. Thus the ε for the p^{th} order predictor becomes

$$\varepsilon^{(p)} = \left\langle e^2[n] \right\rangle = \sum_{n=\infty}^{\infty} e^2[n] = \sum_{n=0}^{N+p} e^2[n] \tag{2.42}$$

And then the autocorrelation values needed for the Yule-Walker equations can be computed as.

$$r_{ss}[|i-k|] = \sum_{n=0}^{N-|i-k|} s[n]s[n+|i-k|] \tag{2.43}$$

This estimation can be used for both the deterministic and random signal, but it is of course not equal to the autocorrelation of the infinite signal. This limits the accuracy of the linear prediction, and may lead to large errors in the first p samples, as the prediction filter tries to predict nonzero samples from zero samples, and likewise for the last p samples when the filter tries to predict zero samples from nonzero samples. This can bias the estimation of the predictor coefficients, and therefore a window function is often applied to the signal, such as a Hamming or Hanning window before the autocorrelation is computed, this is discussed further and further reference can be found in [Makhoul, 1975].

2.5.2 The covariance method

Using the covariance method we assume the the signal range is a finite interval $0 \leq n \leq N$ and then tries to make a prediction in the interval $p \leq n \leq N$. Thus the p^{th} order predictor becomes.

$$\varepsilon_{cov}^{(p)} = \left\langle e^2[n] \right\rangle = \sum_{n=p}^{N} e^2[n] \tag{2.44}$$

This means that the prediction is only over a finite interval, as opposed to the infinite interval used in the autocorrelation method.

2.5.3 Comparison of methods

The prediction error of the covariance method can become equal to zero, for n $>$ 0 if the original signal was generated by an all-pole model. The autocorrelation can, as earlier alluded, never becomes zero since the first and last p inputs to the autocorrelation leads to the inclusion of a bias. The minimum prediction error is instead $Gv[n]$, which can be seen in Equation 2.10, by setting the model error to zero and rearranging.

Computation of the prediction coefficients is done by solving the linear equations which was expressed as $\Phi a = \Psi$ and then obtain the coefficients by inversion $a = \Phi^{-1}\Psi$. In both methods the matrix Φ are symmetric and positive definite. This means that the system can be solved more efficiently than normal. Yet using the autocorrelation method the matrix Φ also becomes a Toeplitz matrix, which means that among others it can be computed by the Levinson-recursion or Levinson-Durbin algorithm, making it even faster. The Levinson-Durbin algorithm is briefly examined in section A.1

As for the filter stability is the autocorrelation method guaranteed to have poles within the unit circle and is therefore stable. The same guarantee can not be given in case of the covariance method in general, however [Makhoul, 1975] reports this not to be a problem in practice.

The LPC used in the LPCNet is based on the autocorrelation method, and its enablement of the Levinson-Durbin algorithm, as a means to compute the predictor.

2.6 Model order

Only using the covariance method it is possible to achieve a prediction error which is equal to zero. Using the autocorrelation method make do with an nonzero prediction error. It has been shown that the minimum error ε_i decreases or stays the same as the prediction order i increases [Levinson, 1947] such that $0 \leq \varepsilon_i \leq \varepsilon_{i-1}$ where $\varepsilon_0 = r[0]$. we then can also define was is called the normalized error $\mathcal{V}_i$ as.

$$\mathcal{V}_i^{(p)} = \frac{\varepsilon_i^{(p)}}{r[0]} \tag{2.45}$$

Usually at some $\mathcal{V}_i$ flattens out becoming more or less fixed, and this point is often selected as the model order (p). An example of this is shown in Figure 2.6, further information of this process can be found in section A.1.

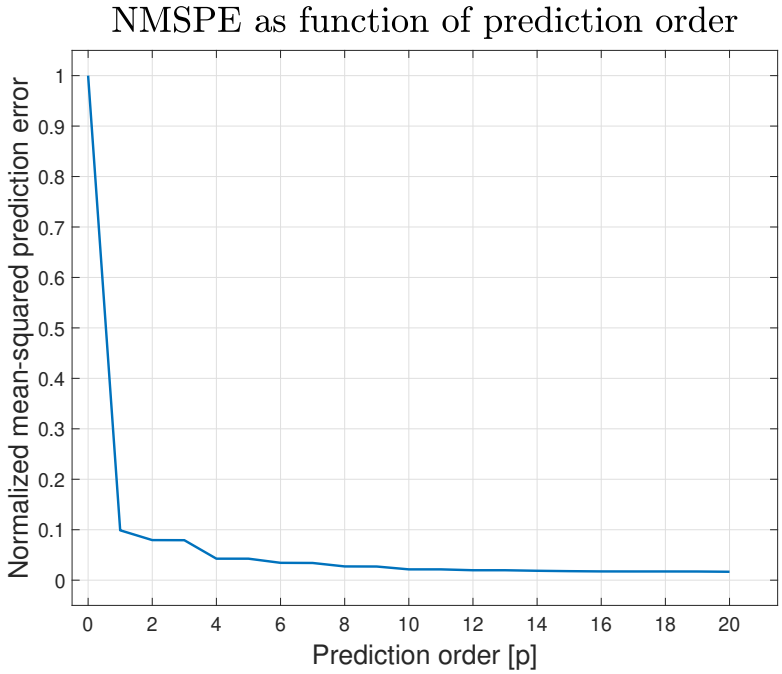


Figure 2.6: Example of normalized mean-squared prediction error (NMSPE) $\mathcal{V}^{(p)}$, as a function of model order p, using the autocorrelation method

In this example we can see how the normalized error rapidly decreases as we increase the order, hereafter it stagnates while approach a model error of zero around the 10'th to 12'th order. Since this example is based around the autocorrelation method, does the error never become equal to zero.

In speech analysis the model order can be based on the physical models for speech production, in [Rabiner and Schafer, 1978] they show that there will be around N/2 resonances per 1kHz sampling frequency, with a vocal tract of 17.5 cm, which is around the average of a human male [Rabiner and Schafer, 1978]. Thus with a sampling frequency of 8kHz, a typical sampling frequency for speech, the baseband will be 4kHz. Hence N would be equal to 8. Additionally it is described "a total of 3-4 poles is required to adequately represent the source excitation spectrum and the radiation load." [Rabiner and Schafer, 1978]. Therefore would a speech signal sampled at 8kHz require a order p of around 11-12. On the subject of LPC do we often see $p = 12$ as a form of "de facto standard" for the model order. LPCNet uses a model order of 16 and this choice is not described, yet it may be related to the network's wideband sampling frequency of 16kHz.

2.7 All-pole spectrum analysis

In the spectrum analysis of the all-pole model, the idea is that if the data provided fits the parametric signal model, then a finite section of the data is enough to determine the model parameters and therefore also the spectrum.

If we look at the spectrum for the case of a deterministic input, the estimated spectrum ($|\hat{S}(e^{j\omega})|^2$) of the signal s[n] is

$$|\hat{S}(e^{j\omega})|^2 = |H(e^{j\omega})|^2 |V(e^{j\omega})|^2 = |H(e^{j\omega})|^2 \tag{2.46}$$

Because the periodogram of the input $|V(e^{j\omega})|^2$ is equal to 1, since the input is an unit impulse excitation to the system. The same result occurs in the random case, for the power spectrum of the output of the model.

$$P_{\hat{s}\hat{s}}(e^{j\omega}) = |H(e^{j\omega})|^2 P_{ww}(e^{j\omega}) = |H(e^{j\omega})|^2 \tag{2.47}$$

Since the periodogram of the random process $P_{ww}(e^{jw})$ is equal to 1, when it is for a white noise input. Hence we can estimate the signal $s[n]$'s spectrum, by computing the squared magnitude of the frequency response of the system model $|H(e^{j\omega})|^2$, which works for both cases.

$$\text{Estimated spectrum} = |H(e^{j\omega})|^2 = \left|\frac{G}{A(e^{j\omega})}\right|^2 \tag{2.48}$$

Now if we look at the averaged prediction error in terms of the Discrete-Time Fourier transform (DTFT) of the signal $s[n]$, and remember that the prediction error is the convolution of the input signal and the prediction filters impulse response i.e. $e[n] = h_a[n] * s[n]$. Then the averaged prediction error, using the autocorrelation method and Parseval's Theorem, is.

$$\varepsilon = \sum_{n=0}^{N+p} e^2[n] = \frac{1}{2\pi}\int_{-\pi}^{\pi} |S(e^{j\omega})|^2 |A(e^{j\omega})|^2 d\omega \tag{2.49}$$

Additionally if we substitute the prediction filter, with its relation to the transfer function of the LTI system i.e. $A(z) = G/H(z)$ then Equation 2.49 becomes.

$$\varepsilon = \sum_{n=0}^{N+p} e^2[n] = \frac{G^2}{2\pi}\int_{-\pi}^{\pi} \frac{|S(e^{j\omega})|^2}{|H(e^{j\omega})|^2} d\omega \tag{2.50}$$

Hence we see that minimizing the total modeling error energy ε is the equivalent to minimizing the ratio of the spectrum of the signal $s[n]$ to the magnitude-squared of the frequency response of the all-pole model.

Which implies that the all-pole model spectrum seeks to approximate the energy of the signal spectrum, closer at frequencies where the energy of the signal is large. Since these make for a larger error, compared to frequencies with lower energy. Hence using this all-pole model prefers a better fit around energy peaks in the signal's spectrum.

2.8 Summary

In this chapter we have examined the subject of parametric signal modeling, which LPC is a part of. This exposition is again based on Chapter 11 in [Oppenheim and Schafer, 2013] and the article [Makhoul, 1975]. We have now described the underlying basis of LPC, how it works and how it can be computed. Thus we move on to describe the corresponding basis of machine learning and the type of neural network used in LPCNet.

Chapter 3

Introduction to Neural networks

In this chapter we will mainly examine the type of neural network used by the LPCNet, which is the RNN. Before that however is a brief and very shallow introduction to deep feedforward neural network, since these types of network make the foundation of many other types of neural networks including the RNN. Furthermore does the feedforward network give a more intuitive understanding, of the many functions and methods utilized in a network. After the *very* brief introduction to the feedforward network, the RNN will be described in greater detail, since this is the type of network LPCNet builds on. This chapter will mainly be based on [Goodfellow et al., 2016], additional arguments or sources of information will be given within the text.

3.1 Introduction to deep neural networks

In general does many neural networks address it self with the task of approximating an underlying function f. This function could for example be a classifier, which given an input x yield an output class y through the use of a function f and a set of parameters θ. The network is then task to find the function f through the parameters θ, which yields a suitable mapping, such that it can predict or classify y given x. One of the most basic neural networks are the feedforward neural network also called Multilayer Perceptrons (MLP). The "feedforward" term comes from the fact that information only "flows" forward through the network and in the end yields some output. When feedback is introduced into such a network they are generally based on the RNN architecture which will be described in 3.2. The "network" term comes form the fact that, multiple perceptrons are used in layers connected to each others, thus creating a form of network. Lastly does the "neural" term emerge of the loss inspiration from the neurons found in the brain.

The perceptron was an invention of Frank Rosenblatt [Rosenblatt, 1958, Rosenblatt, 1962]. The basic idea behind the perceptron is that, the perceptron receives a number of inputs and based on these inputs it calculates its own activation value. An illustration of a perceptron can be seen in Figure 3.1, where it is given a number of weighted inputs and a bias, which based on the perceptrons activation function yields an output.

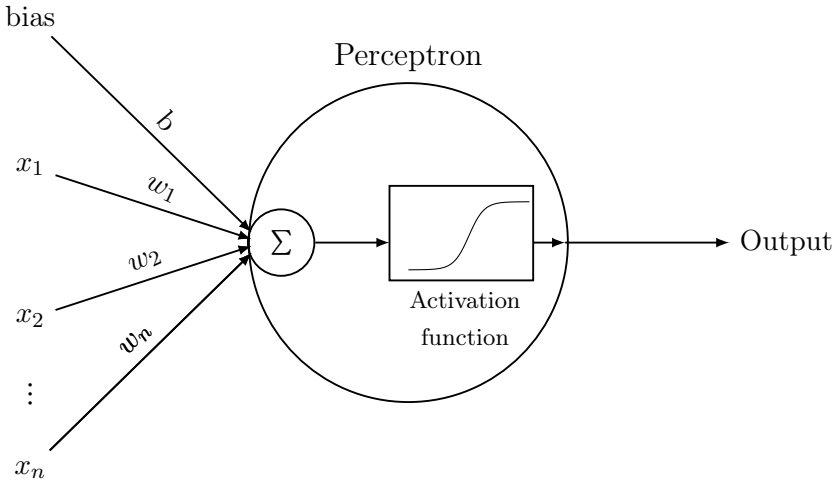


Figure 3.1: Illustration of a perceptron receiving a number of inputs **x** weighted by **w** and a bias **b**. It outputs an activation, based on the inputs and its activation function

The reason for introducing bias is such that the network can make affine functions i.e. $f(x) = \mathbf{x}^T\mathbf{W} + b$, Where W is the input weight matrix to the input x and b is a bias. The activation function serves the purpose of enabling the network, to make non-linear functions i.e. $f(x) = \phi\left(x^T W + b\right)$. Hence the activation function is nearly always non-linear. Without introducing this non-linear activation function the network cannot solve problems which are not linear separable, such as the XOR problem presented in [Minsky and Papert, 1969].

There exist many activation function, some of those we will encounter is the Rectified Linear Unit (ReLU), the sigmoid function which will be denoted by σ and the hyperbolic tangent function which will be denoted by τ. Which have the following fomulars:

$$\text{sigmoid} = \sigma = \frac{1}{1 + e^{-x}} \tag{3.1}$$

$$\tanh = \tau = \frac{e^{2x} - 1}{e^{2x} + 1} \tag{3.2}$$

$$\text{ReLU} = max(0, x) \tag{3.3}$$

Functions used as activation functions may also be referred to as squashing function, as they squashes the input into some given range e.g. will the sigmoid function squash any input into the range of (0,1).

These perceptrons will then be used in layers to form a network. In its simplest form, the network will contain an input layer, a hidden layer and an output layer. The term hidden layer is one or more layers which lies in between the input and output, these layers are hidden in the sense that they are not observed by other systems. Neither does training data fed into the network show the network how it should tune the layer or layers to achieve a desired output. If a network only contains a single hidden layer it is referred to as a shallow network, wheres if a network have multiple layers they are considered a Deep Neural Network (DNN). An illustration of a simple feedforward network is shown in Figure 3.2.

These networks of course needs to be trained, which is the whole point of machine learning. A common

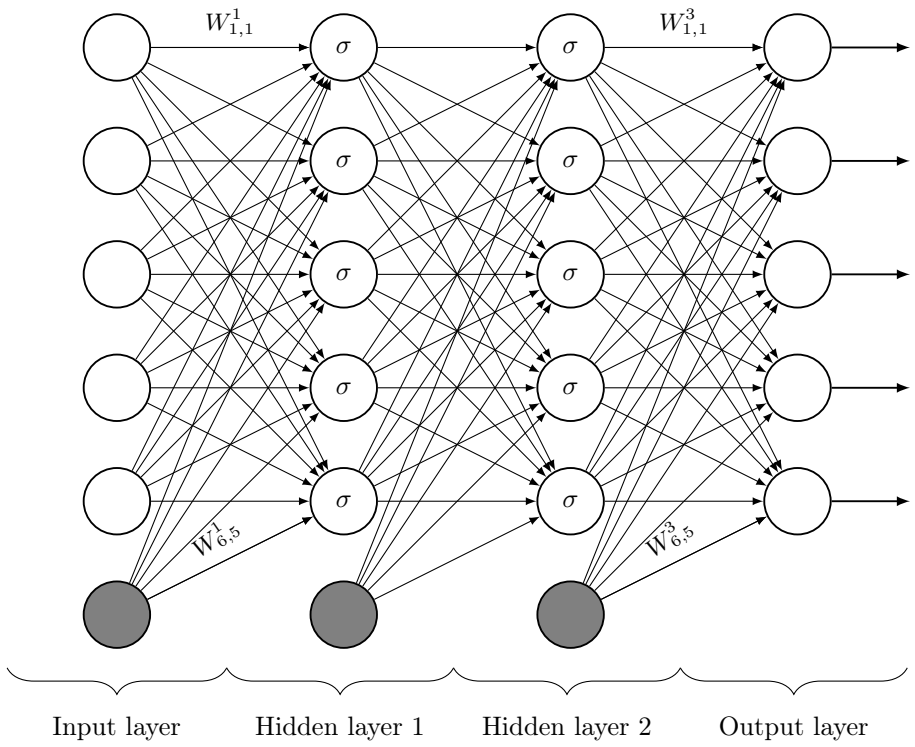


Figure 3.2: A deep feedforward neural network, with biases made implicit by gray nodes. σ denotes the use of sigmoid function. Each node outputs the same value, but they are weighted differently. The weighting scheme $W^a_{b,c}$ means from node b in layer a to node c in layer a+1

way to train these network is based on the gradient decent method or variants of it, for example the common Stochastic Gradient Descent (SGD), which we will return to shortly. The algorithm typically used for these gradient decent methods is the backpropagation (also called error backpropagation or backprop) first proposed by [Rumelhart et al., 1986]. In general one would like to compute the gradient of a cost function $\mathbf{J}$ w.r.t the parameters θ i.e. $\nabla_\theta \mathbf{J}(\theta)$. The network then needs to propagate this error back through the network, in order to update the parameters used in the network. Backpropagation utilizes the chain rule to achieve this. Suppose for example that we have the output y_i of a network, and this output is based on an input x_i which have been through some series of functions e.g:

$$
\begin{aligned}
y &= h(a) \\
a &= g(b) \\
b &= f(x) \\
y_i &= h(g(f(x_i)))
\end{aligned}
$$

This is shown in Figure 3.3. Then via the chain rule one can compute the derivatives of the series of functions as:

$$
\frac{dy_i}{dx_i} = \frac{dy_i}{da}\frac{da}{db}\frac{db}{dx_i} \tag{3.4}
$$

This can be generalized to both vector notation and tensors (multidimensional arrays), which is usually the case when working with neural networks. This needs to be done for the whole network i.e. based on the cost of a single forward propagation. The network have to backpropagate through all weights and biases all the way to the input, in order to compute the gradient for one single input. For the example shown in Figure 3.2 would this network have a total of 90 weights and biases (parameters). As

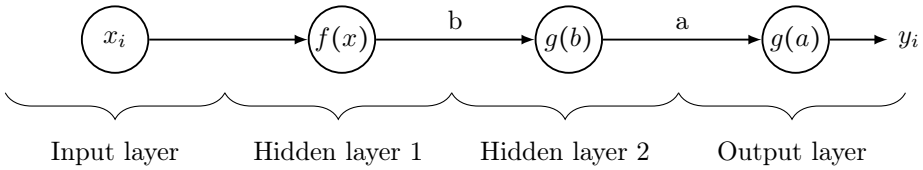


Figure 3.3: A part of the previous network, showing how the input x_i is transformed to y_i, and how the output y_i is a function of $y_i = h(g(f(x_i)))$ Weights and biases have been omitted for clarity

one could imagine when networks become very large, and have hundreds of thousands or even millions of parameters, this process of updating parameters becomes computationally heavy.

One of the common ways to reduce complexity is to work in what is called batches or mini batches. Here inputs equal to the mini batch number will run though the feedforward part of the network, and based on an average of all outputs computed from a mini batch, the network will then compute the cost function and the backpropagation. For example if the network works in mini batches of four. Then the network would compute the output of all four inputs, take an average of all the outputs. The average will then yield an error based on the difference between the networks output (its guess) and the ground truth. Based on the cost function of this error, the network will update its weights through backpropagation, this is illustrated in Figure 3.4. The inputs in the batch is randomly picked from the entire data set, hence with the introduction of mini batches, we work with a subset of the data set. This describes the essence of the SGD, allowing faster iterations through the data set, at the cost of lower convergence. The backpropagation method used in LPCNet is called AMSGrad [Reddi et al., 2018], which builds on top of multiple optimization methods which is based on SGD. These further developed optimization methods typically work with varying step sizes during training, which depends on previous steps sizes taken. These optimization methods is well beyond the scope of this project, and as such we do not go into deeper details of these.

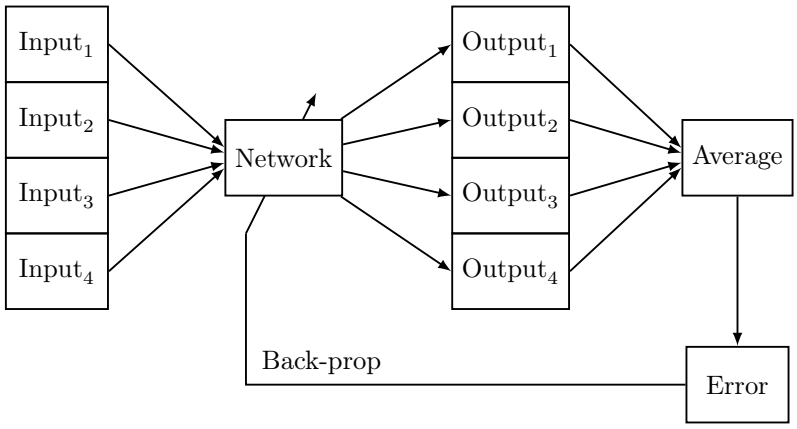


Figure 3.4: figure showing the idea behind mini batches, and how multiple of inputs/outputs are used each time the network's weights are changed through backpropagation

The cost function is computed based on a forward propagation through the network, and is typically based on the error between the networks guess $\hat{y}$ and the desired output y. Typically is the cost function based on the negative log-likelihood function. Because the log function can undo the transformation made by activation functions, and as we will see the softmax function, since these are based on the exponential function. This helps the optimization process since, many activation functions otherwise can saturate, and gradients vanish when this happens, making it hard for the network to know which

direction to move based on the gradient. As such the negative log-likelihood function may also help on the output unit. The output unit have the purpose of representing the data in a meaning full way.

Hence often it will transform the output to some form approaching a probability, and for probability distribution over a discrete variable the softmax function is often used. The softmax function have the following equation:

$$\text{softmax}(\text{y}_\text{i}) = \frac{\exp(y_i)}{\sum_j \exp(y_j)} \tag{3.5}$$

The softmax function can be thought as "soft arg max" since it emphasise the value y_i which is the largest significantly, in a softer differential manner. Thus pushing the largest value towards 1 and all other values of the output towards 0. Additionally does the softmax always sum to one, and hence its output can be thought of as a representation of a probability distribution.

When a neural network have finished its training, the back propagation is no longer needed, and only the feedforward part of the network is considered, this is often referred to as inference. The training phase of the network typically stops either when it has trained for a set amount of time, this time is typically measured by the number of times the network have been presented for the entire data set, which is referred to as epochs.

Alternatively is the network's training stopped when it begins to overfit to the training data. This overfitting can be identified by using a so called validation data set. This data set contains the same type of information as the training data set, but the network is not allowed to train on the validation set. A clear indicator of overfitting is when the network's accuracy on the training data continues to increase, while the the network's accuracy on the validation data set decreases. As such have the network stopped capturing the general trend the data holds, and rather it have sort of started to remember the training data set instead.

Lastly when the network's training phase is over, its performance is measured by inputting a new third type of data set, a so called verification or test data set. The network have never trained on this data set either, but it differs from the validation set. In the sense that this data set is only given to the network once, and the network's accuracy performance is measured based on this test data set. Thus we avoid as designers to indirectly overfit the network on the validation set, as part of the networks design choices.

Having scraped the surface of the feedforward neural network, some of the fundamental methods and terms have been presented. Next is to dive into the RNN. Additional information about the fundamental of deep neural network are given in both [Goodfellow et al., 2016] and [Bishop, 2006].

3.2 Recurrent neural networks

The recurrent neural network is a further development of the basic feedforward neural network, that in its simplest view allows the network to take dependencies within the inputs into considerations. It was proposed by [Rumelhart et al., 1986] and was used for symmetry detection and isomorphic family trees, since these tasks would require the network to remember earlier inputs in order to make a sensible decision. For example if the network knows a family tree, consisting of three person, say a mother, farther and child. Then if the network is given two persons it could tell who the third family member is, but only if it can 'remember' which two persons it has been given as input.

In the basic RNN the hidden unit (the equivalent to a single perceptron) used in the feedforward neural network have a weighted connection from the past hidden unit and to the future hidden unit. This allows the network to retain some form of weighted memory from past inputs, and the network can then be trained to know how much it should rely on the past inputs and the current in order to achieve a better result. This is shown in Figure 3.5

The similarity between a feedforward neural network and a RNN might not be apparent at first glance, only receiving a single input and yielding a single output as compared to the multiple input and multiple outputs imposed by the feedforward neural networks architecture. when the network is unfolded in regard to time it becomes evident that the RNN handles and yields multiple input and outputs.

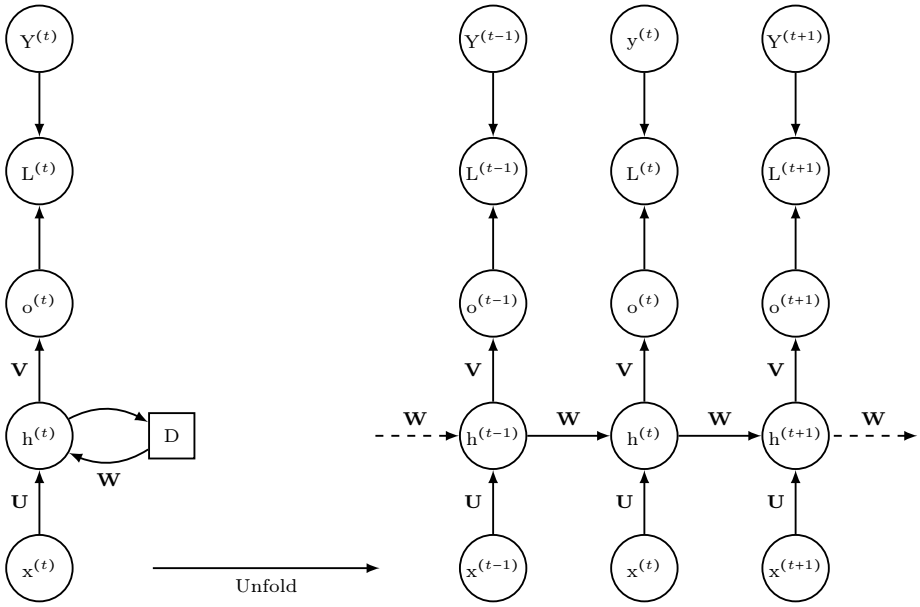


Figure 3.5: Illustration of one time step for an RNN and its unfolding

So when we consider the current state of the system it can be given as:

$$\mathbf{h}^{(t)} = f(\mathbf{h}^{(t-1)}, \mathbf{x}^{(t)}; \boldsymbol{\theta}) \tag{3.6}$$

We can see that the current state (hidden unit) is dependent on the current input $\mathbf{x}^{(t)}$ and the previous computed hidden state $\mathbf{h}^{(t-1)}$ and the parameters $\boldsymbol{\theta}$. Note that $\mathbf{h}^{(t-1)}$ thus has information on every previous input in some weighted form, so we can unfold the expression in order to arrive at an expression which has no recurrence, say for a given time step e.g. $\mathbf{h}^{(3)}$:

$$\mathbf{h}^{(3)} = f(\mathbf{h}^{(2)}, \mathbf{x}^{(3)}; \boldsymbol{\theta}) \tag{3.7}$$

$$= f(f(\mathbf{h}^{(1)}, \mathbf{x}^{(2)}; \boldsymbol{\theta}); \boldsymbol{\theta}) \tag{3.8}$$

From here we can see some properties of the RNN, that the model always have the same input size, no matter the length of the input sequence. Since the model is defined by the process of moving from state to state (i.e. from $\mathbf{h}^{(t)}$ to $\mathbf{h}^{(t+1)}$) and not the sequence of inputs. This allows the model to use same transition function f and the same parameters $\boldsymbol{\theta}$. Meaning that it is only required to have one transition function and the same number of parameters, regardless of the sequence length. This can be seen in Figure 3.5, where the weights $\mathbf{U}$,$\mathbf{V}$, $\mathbf{W}$ are used repetitively. $\mathbf{o}^{(t)}$ denotes the weighted output

of $\mathbf{h}^{(t)}$ as seen in Equation 3.10, $\mathrm{Y}^{(t)}$ denotes the ground truth at time t and $\mathrm{L}^{(t)}$ denotes the loss at time t.

Some of the fundamental design patterns of RNNs consist of networks mapping a certain number input to a certain number of outputs. Namely the designs which yield one output per input a many to many, those which handles one input and yields multiple outputs a 1 to many, those which given many inputs makes a single decision a many to 1. These are shown in Figure 3.6

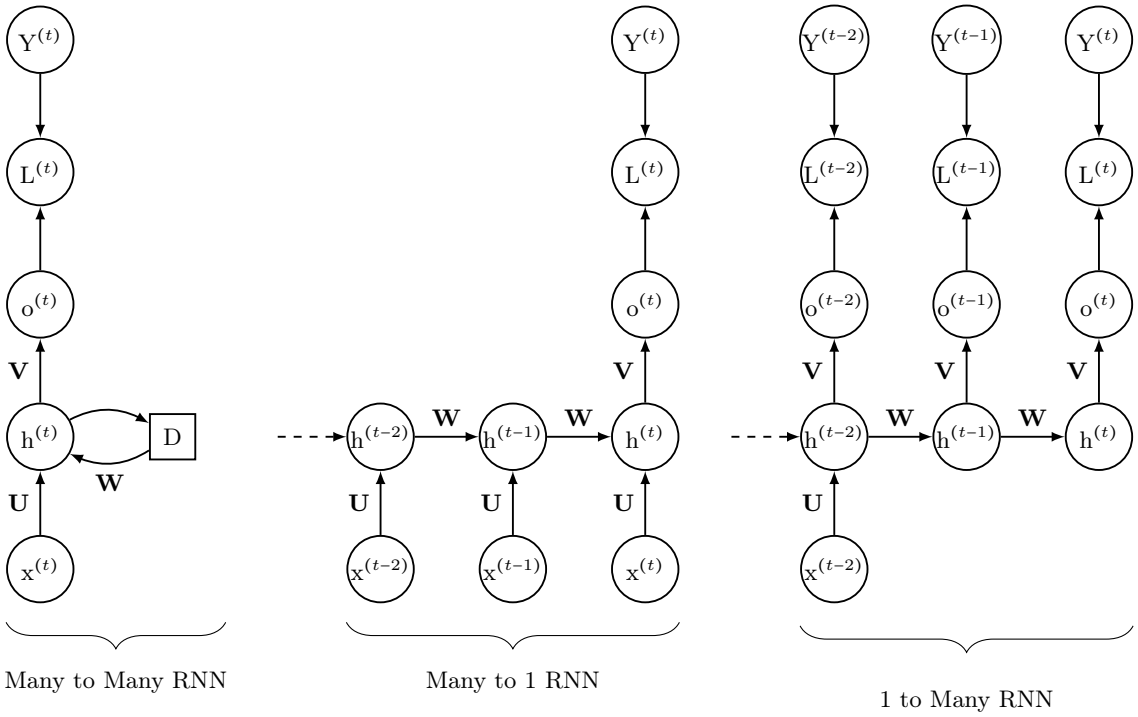


Figure 3.6: Comparison of fundamental designs of RNNs. A many to many RNN, which yields a single output for every single input (left). A many to 1 RNN which takes a whole input sequence, and gives one output (center). Last a 1 to many RNN, where a one input yields multiple outputs(right).

Examples of Many to 1 and 1 to Many is the networks used in [Cho et al., 2014a], where the task was to make a network which could translate from english to french. Here they used two RNNs to form an encoder-decoder bond between the networks. Allowing the two networks to handle "variable-length" inputs of sizes different from each other through a "fixed-length vector representation".

Now with some basic understanding of some structures of RNNs, we can address how the forward propagation of the Many to Many case is computed. We start at time instance $h^{(t)}$:

$$\mathbf{h}^{(t)} = \tau\left(\mathbf{b} + \mathbf{W}\mathbf{h}^{(t-1)} + \mathbf{U}\mathbf{x}^{(t)}\right) \tag{3.9}$$

$$\mathbf{o}^{(t)} = \mathbf{c} + \mathbf{V}\mathbf{h}^{(t)} \tag{3.10}$$

$$\hat{\mathbf{y}}^{(t)} = \mathrm{softmax}(\mathbf{o}^{(t)}) \tag{3.11}$$

Where τ denotes use of the *tanh* as activation function, $\mathbf{b}$ and $\mathbf{c}$ denotes biases in the hidden layer and output layer respectively, and lastly there is $\mathbf{W}$, $\mathbf{U}$ and $\mathbf{V}$ which are weights matrices for the previous hidden layer, the current input and the current hidden layer respectively.

The total loss L can then be computed as the sum of losses for a given sequence of input x and the corresponding truth y at every time step (up to τ).

$$L(\mathbf{X}|\mathbf{Y}) = L(\mathbf{x}^{(1)}, ..., \mathbf{x}^{(\tau)}|\mathbf{y}^{(1)}, ..., \mathbf{y}^{(\tau)}) \tag{3.12}$$

If $L^{(t)}$ for example is the negative log-likelihood of y at a given time step t, given the inputs prior to this time step $x^{(1)}, ..., x^{(t)}$, we have that:

$$L = \sum_t L^{(t)} \tag{3.13}$$

$$= -\sum_t \log\, p_{model}\left(y^{(t)}|\mathbf{x}^{(1)}, ..., \mathbf{x}^{(t)}\right) \tag{3.14}$$

$$= -\sum_t \log\, p\left(y^{(t)}|\hat{\mathbf{y}}^{(t)}\right) \tag{3.15}$$

With the loss computed, it is possible to calculate the gradient w.r.t the networks parameters. This requires a forward pass of the unfolded RNN, the equivalent of moving left through right in Figure 3.5, which is followed by a back propagation (moving right through left). This operation is expensive in the sense that this cannot be done in parallel, since that any given state of the network is dependent on the previous and there by sequential in nature, this requires $\mathcal{O}$(t) computations. Furthermore is it required to store all parameters of the forward pass, so that they can be reused in the back-propagation, which means that the memory cost is also $\mathcal{O}$(t). The algorithm which performs this task is called Back-Propagation Through Time (BPTT).

3.2.1 Deep RNNs

It has been suggested that it can be an advantage to introduce depth into the RNN by [Graves, 2013] and [Pascanu et al., 2013]. In general we have the three sets of parameters namely **U**,**W** and **V**. The weight matrices associated with going from input to hidden unit (**U**), going from the past to the present hidden unit **W** and going from hidden unit to the output **V**. One way to add depth into the network is to add MLPs in between the weights and an other is to multiple hidden states, examples of this is shown in Figure 3.7

The latter case where multiple hidden states is used in the network, is also what we see in the LPCNet where two stacks of Gated Recurrent Unit (GRU) have been stacked on top on each other, specifically "GRU_A" and "GRU_B". An explanation of what GRUs are, is given in 3.2.4.

3.2.2 The gradient problem of RNNs

Deep neural networks have a problem regarding long term dependencies. This occurs in deep feedforward networks with many layers, and also for RNNs. Since for both instances when the gradient is computed through back-propagation, the weights connecting the many hidden units or hidden states, makes the associated gradient either vanish or explode, i.e. dependencies from many layers or time steps ago becomes problematic. This is known as the vanishing or exploding gradient problem. An intuitive example is given by [Goodfellow et al., 2016], where the idea is that if an input move through a path, that consist of multiplying by a weight matrix **W** for every step of the path. The weight will then become a power of the number of steps taken (t) i.e. $\mathbf{W}^t$. Then suppose that the matrix **W** has

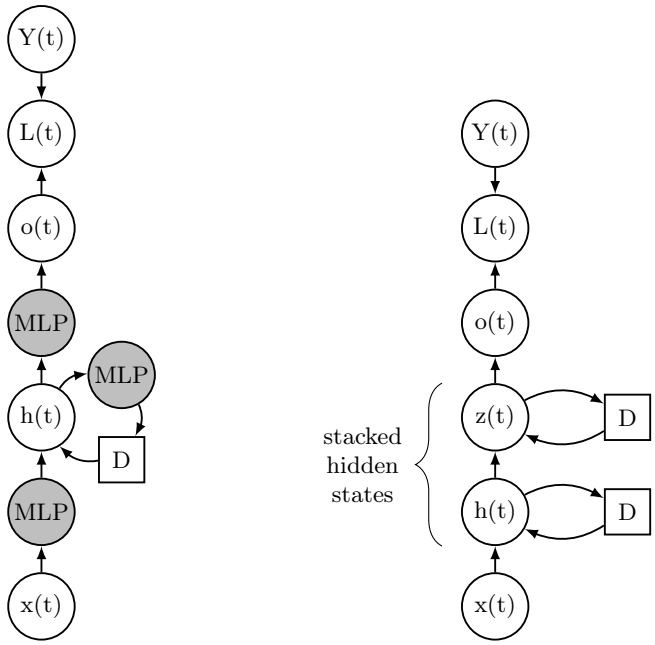


Figure 3.7: Comparison between deep RNNs with MLPs (left) or multiple hidden states (right).

an Eigenvalue Decomposition (EVD) as: $\mathbf{W} = Q\lambda Q^{-1}$, if we then take the matrix $\mathbf{W}$ to the power of t, the corresponding EVD becomes:

$$\mathbf{W}^t = (Q \text{ diag}(\lambda)Q^{-1})^t = Q \text{ diag}(\lambda)^t Q^{-1} \tag{3.16}$$

Hence if the eigenvalues are not close to one in absolute terms, the associate gradient will either explode, making the network unstable, for values above one. For values below one the gradient will like-wise vanish, as the steps increase.

The vanishing gradient makes gradient based optimization problematic, in the sense that it makes it hard for the optimization algorithm to know which direction it should move to minimize the cost. Thus slowing down the optimization process significantly. Feedforward neural networks can handle this by choosing weights between layers carefully [Sussillo, 2014]. RNNs on the other hand reuses the same weights over and over, and hence needs an other solution to this problem.

A further discussion of gradient problems in general is given in [Goodfellow et al., 2016]. Additionally is the vanishing/exploding gradient problem researched in [Hochreiter, 1991, Bengio et al., 1993, Bengio et al., 1994].

3.2.3 Long short term memory

One of the best known, and well used solution to the vanishing/exploding gradient problem is to introduce the Long Short-Term Memory (LSTM), made by [Hochreiter and Schmidhuber, 1997], into the RNN. The LSTM is integrated into the hidden state of the RNN, such that instead of a mere activation function to parameters from the past hidden state, input sample and bias as seen in Equation 3.9. The hidden state now contains what is called a memory cell along with three gates, and multiple additional parameters.

The idea is that the memory cell, can keep some memory within what is call the *cell state*. This cell state contains a self-loop which can keep information for significant amount of time steps. Hence

gradients can be, via the cell state, based on inputs which were given to the network many time steps ago. Solving the problem of having gradients vanish because of weights becoming insignificant, due to the recursive nature of the RNN.

The LSTM is shown in Figure 3.8 is a modified version of the original. This modification was proposed by [Gers et al., 1999], and this included an addition of the *forget gate*. Many additional designs of the LSTM have been made and are covered [Greff et al., 2017], where they give a good summation on LSTMs and their history, along with an empirically test of variants of LSTMs. They concluded that the different variations lead to no significant difference in performance, compared to the one presented here.

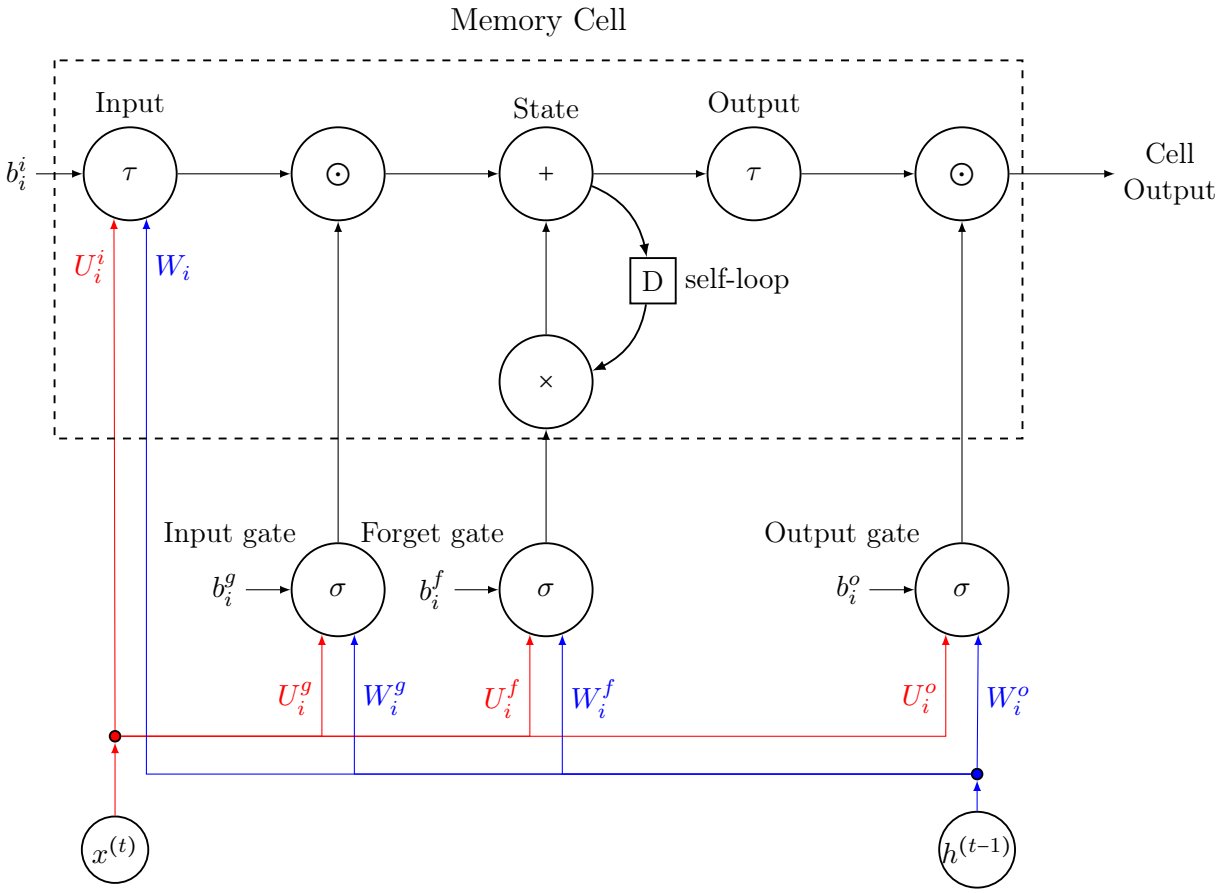


Figure 3.8: Illustration of the long short-term memory Block. Here the red colored lines denotes a weighted version of the input $x^{(t)}$ e.g. $U_i^i x^{(t)}$, and blue lines denotes a weighted version of the recurrent input $h^{(t)}$.

In the figure 'D' denotes a delay of one time step or sample and $\odot$ denotes entry-wise multiplication. Note that it is not strictly necessary to use tanh for the LSTM, any squashing function can be used, but tanh is the most typically used. The use of the sigmoid function is on the other hand always used for the gates [Greff et al., 2017].

The three gates; *input gate, forget gate and output gate* all serve a different purpose. The input gate enables the network to ignore new inputs, "protecting" the current memory it holds with the cell state. Like-wise can the output gate enable the network not to yield any output. The forget gate serves the purpose of giving the network the possibility to forget the information in the cell state, once its memory have become out of date.

Based on Figure 3.8 we can walk through the equations behind the LSTM unit, for the time step t and the LSTM cell i.

First the input $z_i^{(t)}$:

$$z_i^{(t)} = \tau\left(b_i^z + \sum_j U_{i,j}^z x_j^{(t)} + \sum_j W_{i,j}^z h_j^{(t-1)}\right) \tag{3.17}$$

where b, U and W, denotes bias, input weight and recurrent weight. The input is x at time step (t) and h is the past recurrent output from time (t-1). Which is similar to the formula of the hidden state of the simple RNN given in Equation 3.9. The input gate $g_i^{(t)}$ is computed in a similar manner:

$$g_i^{(t)} = \sigma\left(b_i^g + \sum_j U_{i,j}^g x_j^{(t)} + \sum_j W_{i,j}^g h_j^{(t-1)}\right) \tag{3.18}$$

As mentioned before does this allow the network to ignore any input $z_i^{(t)}$, for the given LSTM unit, based on $g_i^{(t)}$. The forget gate $f_i^{(t)}$ is computed the same way as the input gate:

$$f_i^{(t)} = \sigma\left(b_i^f + \sum_j U_{i,j}^f x_j^{(t)} + \sum_j W_{i,j}^f h_j^{(t-1)}\right) \tag{3.19}$$

Which allows the network to forget outdated information contained within the cell. As such we can now compute the state of the cell $s_i^{(t)}$ as:

$$s_i^{(t)} = f_i^{(t)} \odot s_i^{(t-1)} + z_i^{(t)} \odot g_i^{(t)} \tag{3.20}$$

where $\odot$ denotes an entry wise multiplication. Thus we can see that the current state of the memory cell in regards to a forward pass, is given by a weighted value of the previous cell state controlled by the forget gate, and the input $z_i^{(t)}$ controlled by the input gate $f_i^{(t)}$. For the output of the cell we have the output gate $q_i^{(t)}$, which is computed as usual:

$$q_i^{(t)} = \sigma\left(b_i^o + \sum_j U_{i,j}^o x_j^{(t)} + \sum_j W_{i,j}^o h_j^{(t-1)}\right) \tag{3.21}$$

The output gate controls whether or not an output leaves the LSTM cell $h_i^{(t)}$ by the following equation:

$$h_i^{(t)} = \tau\left(s_i^{(t)}\right) \odot q_i^{(t)} \tag{3.22}$$

To summarizes the LSTM cell have the following parameters:

- Input weights: $U_{i,j}^z$, $U_{i,j}^g$, $U_{i,j}^f$, $U_{i,j}^o$
- Recurrent weights: $W_{i,j}^z$, $W_{i,j}^g$, $W_{i,j}^f$, $W_{i,j}^o$
- Bias weights: b_z, b_i, b_f, b_o

Thus the LSTM covered here have a total of 12 parameters for a single LSTM cell in the hidden state. This covers a single LSTM cells equations for a forward pass. Yet is it rare to have only a single LSTM in a hidden state, usually one would have many LSTMs in parallel within the hidden state. In which case the size of the input weights $\mathbf{U}$ are $\mathbb{R}^{N\times M}$, the recurrent weights $\mathbf{W}$ are size $\mathbb{R}^{N\times N}$ and the bias $\mathbf{b}$ weights are size $\mathbb{R}^{N\times 1}$ [Greff et al., 2017]. Here N denotes the number of LSTM cells and M is the number of inputs per time step. A hidden state with multiple LSTMs in parallel is illustrated in Figure 3.9.

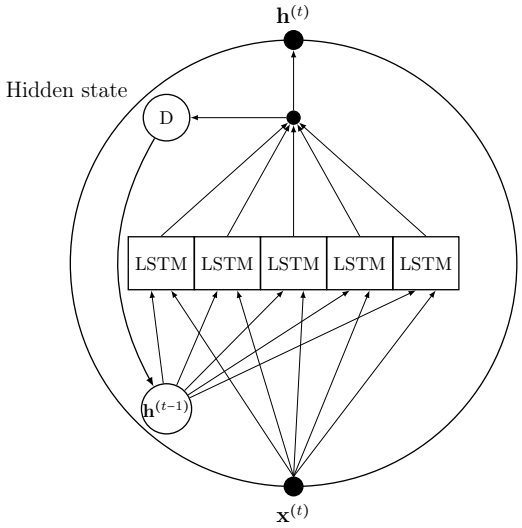


Figure 3.9: Figure of multiple LSTMs within a single hidden state. Note that biases, input and output weights are not shown for clarity

From the figure we can see that every LSTM are given the input sequence at time step t, and the past output of every LSTM $\mathbf{h}^{(t-1)}$. Given this data every LSTM produces one output, which make up the output vector $\mathbf{h}^{(t)}$. In the case where this whole hidden state, were input to a new hidden state with LSTMs i.e. stacking of LSTMs. The following LSTMs will receive all outputs i.e. $h^{(t)}$, the same way the LSTMs receive $h^{(t-1)}$. This process of having multiple LSTM layers stacted is shown in Figure 3.10.

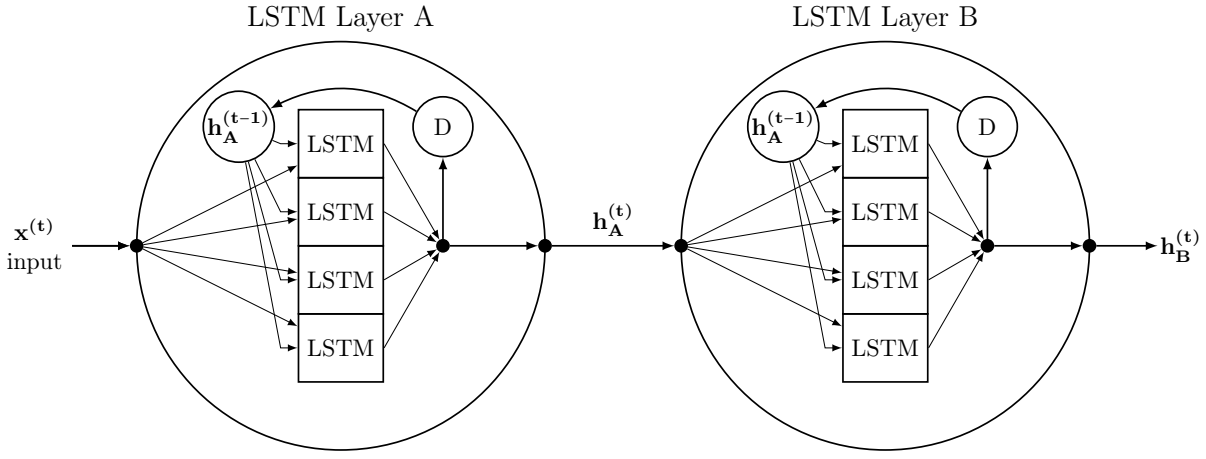


Figure 3.10: Illustration of two LSTM layers each containing four LSTM units within, biases and weights are not shown for clarity

With the understanding of the LSTM cell, we will now move on to the special type of LSTM unit

called Gated Recurrent Unit (GRU), which are the kind of units used in the LPCNet.

3.2.4 Gated recurrent unit

The GRU was first proposed in 2014 in [Cho et al., 2014b, Cho et al., 2014a] as a new type of hidden unit that can adaptively remember and forget elements of a sequence just like the LSTM. The GRU was motivated by the LSTM unit. Later the same year was this new hidden unit named as Gated Recurrent Unit in [Chung et al., 2014], where they performed preliminary experiments. Here they compared three different models namely the LSTM, GRU and *tanh* approach (i.e. a simple RNN), where the hidden state in the network is handled by these approaches respectively. In these preliminaries they found that both the GRU and LSTM outperformed the basic *tanh* approach, but could not make any conclusion of whether the GRU or LSTM performed best.

The GRU is simpler in design, but might be a bit harder to interpret. The idea is that the input gate and forget gate of the LSTM can be 'coupled' [Greff et al., 2017] into what is called an *update gate*. The update gate z^t have the following equation for time step t:

$$\mathbf{z}^{(t)} = \sigma(\mathbf{U}^z\mathbf{x}^{(t)} + \mathbf{W}^z\mathbf{h}^{(t-1)} + b^z) \tag{3.23}$$

Note that we are only looking at one single GRU, if we are to consider multiple GRUs like earlier case of the LSTMs equations, an superscript of i denotes which specific gated unit we are addressing. The update gate can be compared to the input and forget gate as $\mathbf{f}^{(t)} = 1 - \mathbf{g}^{(t)}$. The update gate is then used in the GRU's activation (output) in the follow manner:

$$\mathbf{h}^{(t)} = (1 - \mathbf{z}^{(t)}) \odot \tilde{\mathbf{h}}^{(t)} + \mathbf{z}^{(t)} \odot \mathbf{h}^{(t-1)} \tag{3.24}$$

where $\odot$ denotes an element wise multiplication and $\tilde{\mathbf{h}}$ is referred to as the candidate activation. Since $\mathbf{z}$ has a value between 0 and 1, the update gate controls the ratio of how much of the past output $\mathbf{h}^{(t-1)}$ and the candidate activation $\tilde{\mathbf{h}}^{(t-1)}$ should be used for the GRU's output $\mathbf{h}^{(t)}$ at the time step t. The candidate activation is computed as:

$$\tilde{\mathbf{h}}^{(t)} = \sigma(\mathbf{U}^{\tilde{h}}\mathbf{x}^{(t)} + \mathbf{W}^{\tilde{h}}(\mathbf{r}^{(t)} \odot \mathbf{h}^{(t-1)}) + b^{\tilde{h}}) \tag{3.25}$$

where r is called *reset gate*. We can see that through the use of the reset gate, it is possible to only base the new candidate activation on the current input sequence. The reset gate is computed as usual:

$$\mathbf{r}^{(t)} = \sigma(\mathbf{U}^r x^{(t)} + \mathbf{W}^r\mathbf{h}^{(t-1)} + b^r) \tag{3.26}$$

These equationa are illustrated in Figure 3.11. Here the block called '$1 - z$' denotes the operation of subtracting the output of the update gate $\mathbf{z}$ from 1.

If we again summarize the parameters used by a single GRU we have the following:

- Input weights: $U^z_{i,j}$, $U^{\tilde{h}}_{i,j}$, $U^r_{i,j}$
- Recurrent weights: $W^z_{i,j}$, $W^{\tilde{h}}_{i,j}$, $W^r_{i,j}$
- Bias weights: b^z_i, $b^{\tilde{h}}_i$, b^r_i

Hence a single GRU unit uses only 9 parameters as opposed to the LSTM's 12. In [Valin and Skoglund, 2019a] it is not described as to why LPCNet uses GRU based layers instead of LSTM since they typically obtains similar accuracy performance. In [Soroush Mehri, 2017] they however found GRUs to perform slightly better than LSTMs, and may have been inspired by this.

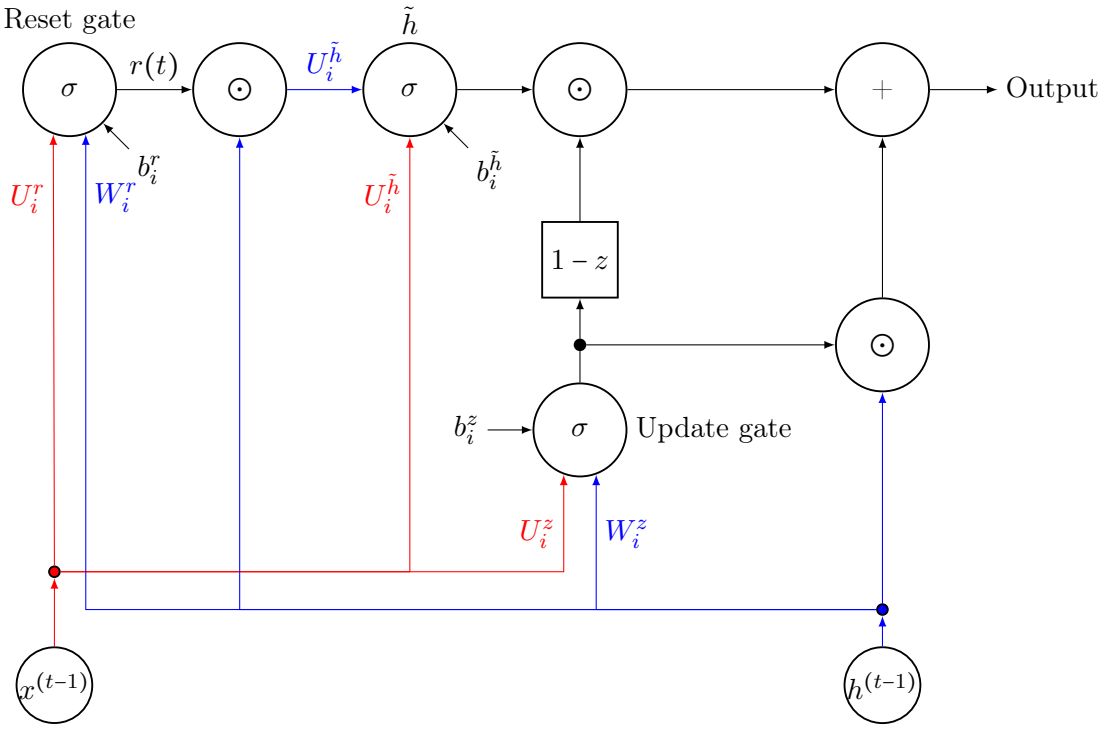


Figure 3.11: Illustration of the full gated recurrent unit

3.3 Summary

We have in this chapter outline the theoretical background of the machine learning principles used in the LPCNet. This includes the simple perceptron, multi layer perceptrons able to create a simple feedforward network. From this we further described a specific type of neural network namly the RNN, by introducing the simple RNN. This lead to the explanation of the gradient problem inherent to the simple RNN, and why gated units that is LSTMs and GRUs instead are used.

Having a theoretical basis for the principle of both the feedforward neural networks, RNN and the main unit used LPCNet the GRU, along with the theory of linear predictive coding introduced in chapter 2. We will now move on to the more specific part of the LPCNet network in chapter 4.

Part II

Algorithm

Chapter 4

Functional Description and analysis Description of LPCNet

Having described the basics of linear predictive coding and recurrent neutral networks, we now have the necessary knowledge to go through some of the more specific methods used in the LPCNet. In section 4.1 and 4.2 respectively the method of embedding and creating a sparse network (pruning) will be introduce. In section 4.3 the method of sampling from a probability distribution produced by the softmax function will be covered, and in section 4.4 we will cover how noise is injected during the training procedure. Lastly, we will examine the different layers of the network, in order to investigate the computational complexity of the network and its layers, this will be done in section 4.5.

First we will give an overview of the LPCNet as discussed in [Valin and Skoglund, 2019a], which can be seen in Figure 4.1. The network consists of three main parts; the sample rate network, Linear Predictive Coding (LPC) and the frame rate network. The network first receives a total of 20 features, 18 of which are Bark-scale [Fastl and Zwicker, 2007] cepstral coefficients and the last two is period and correlation. These features are used both in the frame rate network and to compute the LPC coefficients. We will return to how exactly the frame rate network acts in section 4.5.1, but the main point is that it produces what is called a conditioning vector in [Valin and Skoglund, 2019a] denoted **f**. This conditioning vector has a dimension of 128, it is computed once every frame and held constant through out the frame. It contains information of the 20 features which is also used to compute the LPC predictions. The conditioning vector is given to the sample rate network along with the one-step LPC prediction, the earlier excitation and output from the sample rate network. Hence the sample rate network, not only have information based on the LPC predictions, but also the initial features which yielded these predictions through the use of the conditioning vector. We will also return to the sample rate network in deeper detail in section 4.5.2.

All these inputs are concatenated and given to the first GRU layer, subsequently followed by the second GRU layer and two fully connected layers denoted 'dual FC'. The output of the dual fully connected layer used with a softmax activation function, which yield the probability $P(e^{(t)})$ of a sample being any of the possible excitation values $e^{(t)}$. Note that in this thesis we use (t) to describe a time step (sample) in discrete time, as opposed to the notation of [n] used in signal processing literature. It is done with the intent to keep the notation of the various machine learning related articles and materials.

Based on the sampling process described in section 4.3, the sample rate network gives an excitation

$e^{(t)}$. Lastly does the residual of the excitation and prediction yield the output $s^{(t)}$. Before describing the layers in greater detail, the methods used in the LPCNet will be covered first, starting with the use of embedding.

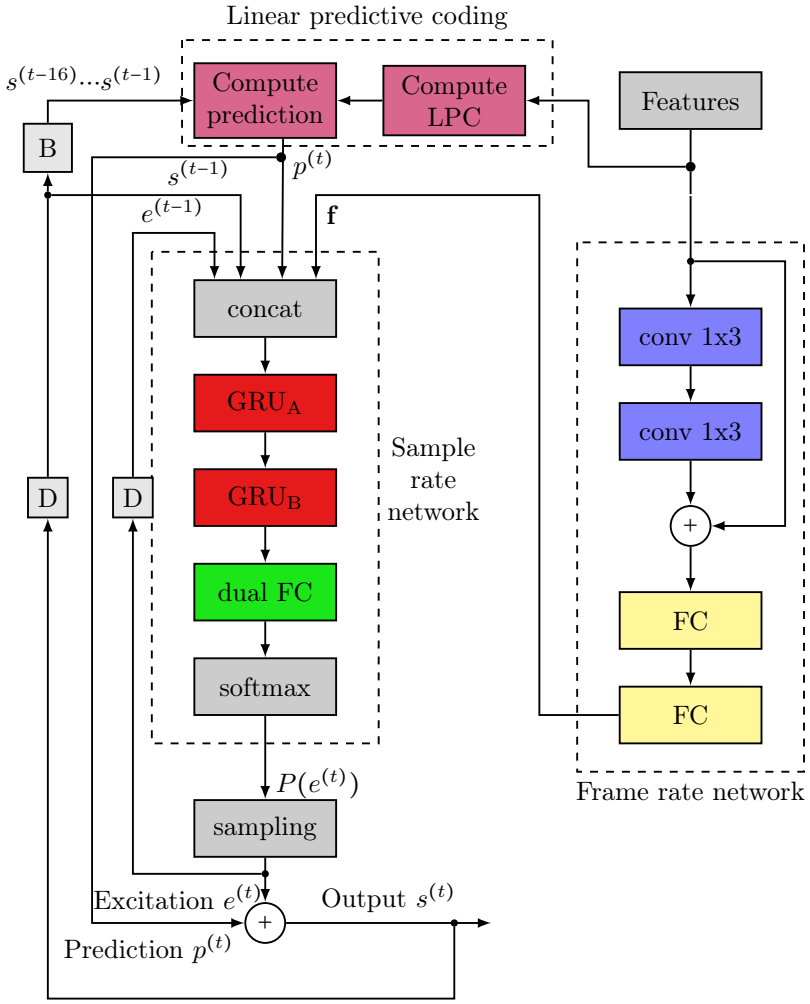


Figure 4.1: Overview of the LPCNet, modified figure of [Valin and Skoglund, 2019a]. 'D' denotes a delay of sample and 'B' denotes a buffer

Figure 4.1 differs only little compared to the figure given in [Valin and Skoglund, 2019a]. Here we have tried to make the delays and buffering of signals more explicit, by introducing buffers and delays. What is not as explicit is the minor layers, which performs some of the necessary pre-processing. In Figure 4.2 These layers have been made explicit. Note that in this figure we only concern our self with the path from the inputs to the Dual FC layer. For example, many of the signals are not present.

Starting at the frame rate network. We can see that it is given the 'Features' and 'Pitch' as input. The reason for separating the pitch periods for the remaining features, is that the pitch is input to a so called embedding layer. This embedding layer is described in section 4.5.1. The output of the embedded pitch is then reshaped such that it can be concatenated with the remaining features. These concatenated features are then given as input to the frame rate network as depicted in Figure 4.1, which yields the conditioning vector **f**. How the frame rate network computes this vector is also described in section 4.5.1. The Lambda layer's task is to hold the conditioning vector **f**, as a constant input to both the GRUs throughout the frame.

In the sample rate network, we see that in addition to the conditioning vector, the network is also given input excitation $e^{(t-1)}$, input speech signal $s^{(t-1)}$ and predictions $p^{(t)}$ as input. How these signals are made is discussed in Figure 4.6. These signals are also input to two embedding layers, these as described further in section 4.5.2 and section 4.5.2. The use of the "Enbedded signal" layer can yield a reduction complexity wise, this is discussed in section 4.1. Both the embedded signals are then reshaped and concatenated all together with the conditioning vector as input to the first of the two GRUs. The output of the first GRU is then concatenated with the conditioning vector again, and serves as input to the second GRU. From here after the path through the remaining layers is the same as in Figure 4.1.

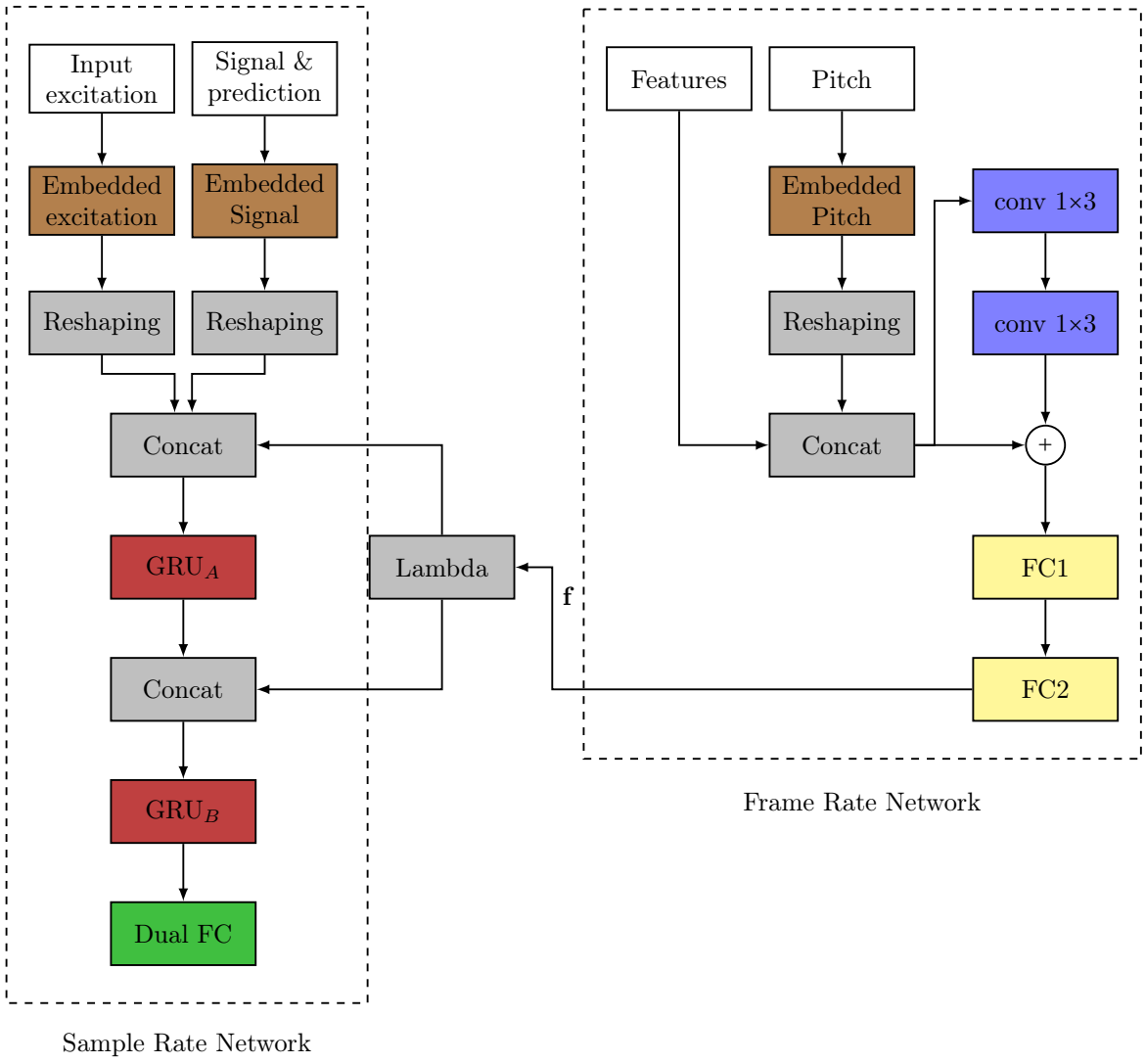


Figure 4.2: LPCNet overview with explicit layers. Note that only the frame rate network and sample rate network is show for the sake of clarity.

4.1 Embedding and Algebraic Simplifications

One way that LPCNet [Valin and Skoglund, 2019a] is trying to reduce the complexity of the network, is to introduce an embedding matrix which is denoted **E**. This embedding matrix is also part of the network, and is hence also part of the training. In LPCNet the embedding matrix **E** which is the equivalent to the "embedded signal" in Figure 4.2, it is in between the μ-law compressed predictions, excitation and past output, and the first GRU layer which is denoted GRU_A. For this embedding matrix it has been confirmed "that the embedding has earned - among other things - the function that converts the μ-law scale to linear" [Valin and Skoglund, 2019a] we revisit this statement in section 6.2.3. The Embedding layers does in essence function like a Look-Up Table (LUT), where the network can learn to tune the output of the LUT, we will discuss the embedding layers more in section 4.5.1. Hence one can save some computations by not having to scale the samples to a fixed range, before feeding them into the network, but instead let the network it self learn the necessary functions.

The idea is that during implementation of the trained network one can pre-compute the embedding and GRU product. We denote this product as $\mathbf{V} = \mathbf{UE}$, where $\mathbf{U}$ is the input weights of the GRU as discussed in Section 3.2.4. For example if we consider the update gate of a single GRU unit i, of the first GRU layer we had that:

$$\mathbf{z}_i^{(t)} = \sigma(\mathbf{U}_i^z \mathbf{x}^{(t)} + \mathbf{W}_i^z \mathbf{h}^{(t-1)} + b_i^z) \tag{4.1}$$

where the vector $\mathbf{x}^{(t)}$ is the inputs to any of the units in GRU_A, hence it contains the samples $p^{t-1}, e^{t-1}, s^{t-1}$ and $\mathbf{f}$. All inputs of $\mathbf{x}^{(t)}$ except $\mathbf{f}$, is outputs from embedding layers. Thus for the submatrix product of $\mathbf{U}^{(z,s)}$ and the embedding $\mathbf{E}$, which is the weights of the update gate to the input of the embedded version of the speech signal $s^{(t-1)}$ can be described as:

$$\mathbf{V}_{(z,s)} = \mathbf{U}_{(z,s)} \mathbf{E} \tag{4.2}$$

The same can be done for all the non recurrent inputs. Note that $\mathbf{f}$ only needs to be computed once per frame, this is denoted as $\mathbf{G} = \mathbf{Uf}$, we take a deeper look at this in section 6.2.1. Thus all non-recurrent input (and bias) to the GRU can be described as these embeddings and the Equation 3.23 through (3.26) can hence be written as:

$$\mathbf{z}^{(t)} = \sigma\left(\mathbf{V}_{(z,s)}^{s_{t-1}} + \mathbf{V}_{(z,p)}^{p_{t-1}} + \mathbf{V}_{(z,e)}^{e_{t-1}} + \mathbf{G}_{(z)} + \mathbf{W}^z \mathbf{h}^{(t-1)} + b^z\right) \tag{4.3}$$

$$\mathbf{r}^{(t)} = \sigma\left(\mathbf{V}_{(r,s)}^{s_{t-1}} + \mathbf{V}_{(r,p)}^{p_{t-1}} + \mathbf{V}_{(r,e)}^{e_{t-1}} + \mathbf{G}_{(r)} + \mathbf{W}^r \mathbf{h}^{(t-1)} + b^r\right) \tag{4.4}$$

$$\tilde{\mathbf{h}}^{(t)} = \tau\left(\mathbf{V}_{(\tilde{h},s)}^{s_{t-1}} + \mathbf{V}_{(\tilde{h},p)}^{p_{t-1}} + \mathbf{V}_{(\tilde{h},e)}^{e_{t-1}} + \mathbf{G}_{(\tilde{h})} + \mathbf{W}^{\tilde{h}}(\mathbf{r}^{(t)} \odot \mathbf{h}^{(t-1)}) + b^{\tilde{h}}\right) \tag{4.5}$$

$$\mathbf{h}^{(t)} = (1 - \mathbf{z}^{(t)})\mathbf{h}^{(t-1)} + \mathbf{z}^{(t)} \tilde{\mathbf{h}}^{(t)} \tag{4.6}$$

$$\tag{4.7}$$

Thus the use of the embedding matrix $\mathbf{E}$ yielding $\mathbf{V}$, one gains the functionality of the embedding matrix (the mapping form μ-law domain to linear), at the cost of a few additions and additional memory, more on this in section 6.2.1. We do look closer at how the embedding layer acts in practice, and give a small example in Section 4.5.1.

4.2 Sparse Matrices

When we are usually discussing a sparse matrix, we are thinking of a matrix in which most of the elements are zeros. Normally one would gather the information of a matrix into a few of its elements,

here by lowering the storage needed to store the values of a given matrix. In the case of LPCNet we are indeed trying to obtain a matrix which is sparse, more specifically the weight matrices used in the network. Yet the method for obtaining a sparse weight matrix is different from some of the common methods such as Compressed Sparse Row (CSR) [Golub and Loan, 2013].

In order to lower the complexity, LPCNet seeks to make the parameter matrix of the first GRU layer sparse since it is the largest of the two GRUs used in the network. The way of making the input matrix sparse is based on two approaches. The first method is to make the network (or parts of it, such as the GRU parameter matrix) gradually more sparse as training of the network progresses. To further reduce the complexity is the sparsity of parameters determined in blocks of parameters instead of considering them element-wise.

The LPCNet uses that same method as used in WaveRNN [Kalchbrenner et al., 2018] network, and works in blocks with the size of 16x1, what these block means will become clear soon. First lets consider the way of obtaining weight sparsity.

4.2.1 Pruning and weight sparsity

The method used in WaveRNN is based on the *pruning* techniques used in [Narang et al., 2017] and [Zhu and Gupta, 2017]. The term pruning is derived from the similarities of pruning a tree or plant to that of pruning a neural network. The first immediate usage of the method of pruning is described in [Cun et al., 1990]. In General the pruning process consists of forcing the smallest parameters/weights in an absolute term of the network to zero, thus 'removing' the connections between different nodes in the network.

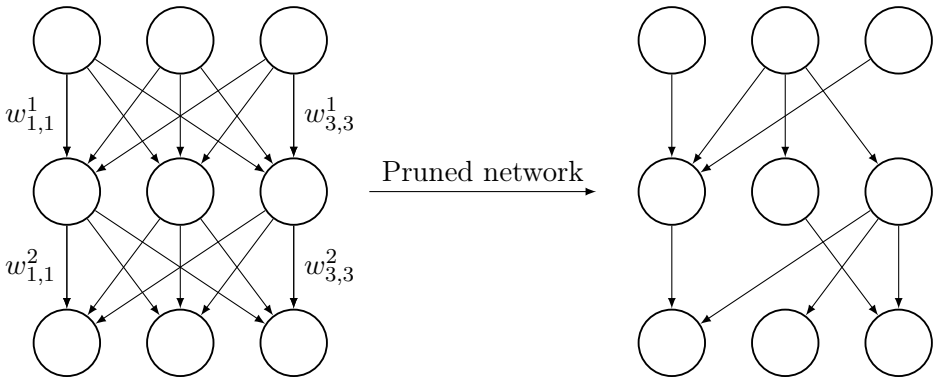


Figure 4.3: The basic idea of pruning, reducing complexity by removing connections between nodes

The reason for pruning a network is that it lowers complexity, since fewer parameters is needed to be handled. It can potentially reduce the energy consumption of the network, if the network can be handled in on-chip memory (SRAM) instead of of-chip memory (DRAM) since accessing on-chip memory requires less energy [Horowitz, 2014].

As mentioned before this is done gradually (iteratively), as a part of the networks training process. The reason for doing it iteratively and not just from the very start, is that the network itself will identify which connections contribute the least to obtain a given solution, and performing the pruning too early will mean that the network have not yet converged to a good solution [Zhu and Gupta, 2017].

The usage of pruning can be quite rewarding, as it can lower the parameter count by a factor of 10, without loosing significant accuracy depending on the network, as shown in [Han et al., 2015].

Here one might ask why not just use a smaller dense network, instead of a large network pruned to sparseness and avoid the pruning process all together. The reason is that a large and sparse network performs better that the small dense network even though they have approximately the same number of parameters [Narang et al., 2017, Zhu and Gupta, 2017]. With these benefits in mind lets consider the pruning method used by the WaveRNN.

The pruning process is achieved through what is called a binary mask, which contains a value for each and every parameter. Initially the mask contains only ones and gradually as the network is pruned, is the mask's values for the pruned parameters turned to zero, and this is done until some target sparsity is achieved. The deciding factor of whether the binary mask is one or zero is based on the parameter's absolute value and an increasing threshold. In [Narang et al., 2017] the increasing threshold is based on a set of heuristics which they call 'hyper-parameters'.

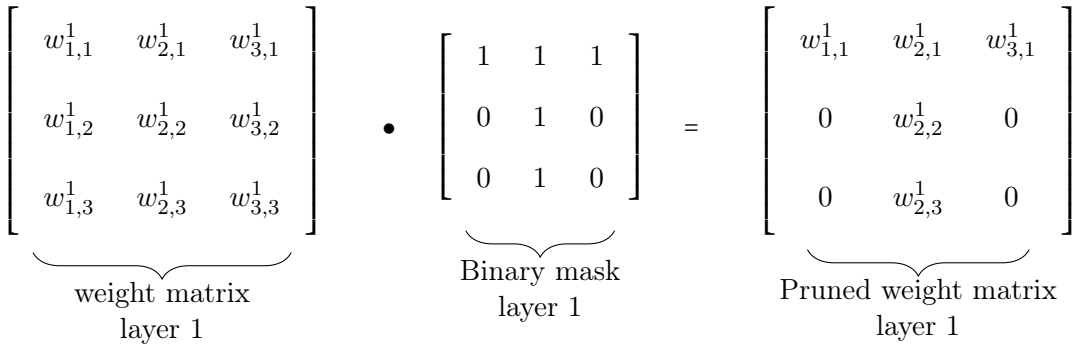


Figure 4.4: Pruning method by binary mask, yielding the weight matrix for the upper layer seen in Figure 4.3

While in [Zhu and Gupta, 2017] the target sparsity divided into so called pruning steps. Such that at pruning step zero no parameters have been forced to zero and at the last pruning step is the number of parameters forced to zero equal to the target sparsity - e.g. 90% of the parameters would have been forced to zero at the last pruning step. Hence is the threshold equal to some given percentile dictated by the target sparsity and the current pruning step. As for the WaveRNN it uses the threshold given by the following function.

$$z(t) = Z\left(1 - \left(1 - \frac{t - t_0}{S}\right)^3\right) \tag{4.8}$$

Where z is the number of parameters, Z is the target sparsity, $t_0 = 1000$ is at which training step the pruning begins and $S = 200e3$ is the number of training step during which the pruning process is active, and the WaveRNN is trained for a total of $500e3$ steps. During the pruning process k weights are zeroed out, where k is the fraction of z computed in Equation 4.8. A graph of the expression can be seen in Figure 4.5 here the target sparsity Z is set equal to 0.85.

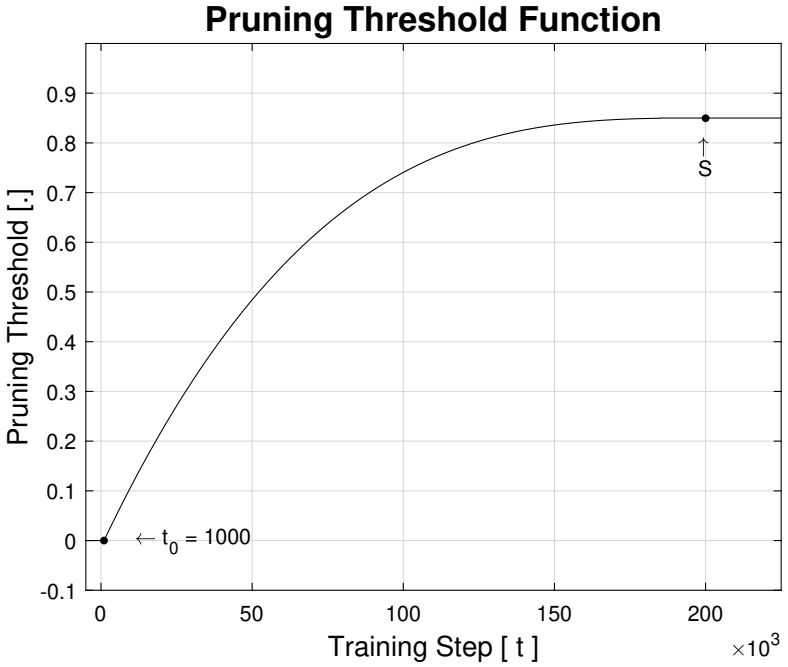


Figure 4.5: Pruning as a function of training steps based on equation (4.8), with Z = 0.85.

4.2.2 Block sparsity

In order to compute the sparsity mask more efficiently in the WaveRNN (and therefore also the LPCNet), is the sparsity mask encoded into blocks of a given size. In WaveRNN the parameters are considered in blocks with the size of $m = 16$ with the structure of 16x1 and 4x4 since these have been found to work well [Narang et al., 2018, Gray and Kingma, 2017], note that LPCNet uses a block size of 16x1 exclusively. This means that during pruning, a block of parameters have the average value of the parameters within the block. The binary mask then only needs to regard one block, hence is the memory overhead reduced. A deeper discussion of the memory overhead and how it can impact performance when computations is performed on a GPU is given in [Narang et al., 2018, Gray and Kingma, 2017]. In addition to addressing the weight matrices in blocks, do they also include all terms in the diagonal of the sparse version of the weight matrices. Since these entries have a high probability of being non-zero. As a result this helps [Valin and Skoglund, 2019a], to avoid the conflict of having only one non-zero value in the block, preventing the pruning of the given block. We return to this subject in Section 6.2.3.

4.3 Sampling from Probability Distribution

One method used in LPCNet to improve the sound quality, is to not sample directly from the probability distribution yielded by the softmax layer ($P(e_t)$). Instead they build on top of a method introduced by the 'FFTNet' [Jin et al., 2018]. In FFTNet they have the same number of possible values or categories 256, corresponding to the 8-bit μ-law compressed levels. A problem they have is when performing predictions in the case of unvoiced signals. The prediction which will yield the lowest training error is to use argmax. That is picking the argument that maximizes the probability of a lower training error. Hence the network will pick the mean of the noise distribution which is zero, making the network output no noise in its output synthesis.

To solve this [Jin et al., 2018] suggest a separate scheme for voice/unvoice sounds. In the case of unvoiced sounds they simply sample at random from the posterior probability distribution ($P(e_t)$). For the case of voiced sounds, they would take normalized logits (input to softmax layers) as input to the softmax layer. Here it should be carefully noted that logits in machine learning lingo, are not the same as in statistics. They denote inputs which are fed into a normalization function, such as the softmax function.

The logits will then be multiplied by a constant of c $>$ 1 (in FFTNet c = 2), before passing it to the softmax layer, and then lastly randomly sample from the posterior distribution.

In LPCNet, instead of having taking binary decision based on voice/unvoice sound, they rely on the pitch correlation of a frame to make a smoother decision. Hence they compute the constant multiplied to the logits as:

$$c = 1 + \max[0,\ 1.5g_p - 0.5] \tag{4.9}$$

where the pitch correlation g_p is $(0 < g_p < 1)$. Furthermore, a constant is subtracted from the probability distribution, in order to completely remove any probabilities below the constant denoted T. In LPCNet the constant T is set equal to 0.002 since it should "provide a good trade-off between reducing impulse", caused by the low probabilities, "and preserving the naturalness of the speech" [Valin and Skoglund, 2019a]. The probability distribution of the softmax layer therefore become:

$$P'(e_t) = \mathcal{R}(max[\mathcal{R}([P(e_t)]^c) - T,\ 0]) \tag{4.10}$$

Where $\mathcal{R}$ denotes the operation of re-normalizing the distribution to unity, both after raising P(e_t) to the power of c and after removing any probability lower than T.

4.4 Training Noise Injection

The second method used in the LPCNet to improve sound quality, is to inject noise into the signal during training, which was also suggested by [Jin et al., 2018]. The reason behind injecting noise into the training process is to make the network more robust, to the mismatch between the synthesized speech and training.

Because both the FFTNet and LPCNet uses the synthesized samples to generate the next samples, and since the network produces errors during training the network yields noisy samples. The process then consists of using noisy samples, as training inputs, which in return yields noisier samples meaning that the process can drift, leading to sound artifacts [Valin and Skoglund, 2019a].

So in order to make up for the noisy training samples, noise is injected into the samples used for training. We will return to what exactly this imply. For the LPCNet they [Valin and Skoglund, 2019a] found that injecting noise by the scheme shown by Figure 4.6, resulted in the network being able to minimize the error imposed by the noise originating from the training process.

Inspecting Figure 4.6, we can see that the input speech is first filtered by the pre-emphasis filter $E(z) = 1 - az^{-1}$ where $a = 0.85$, this produces the emphasised speech at time step t. The speech is then μ-law quantized and noise is added to the signal. The noisy speech signal then serves both as input to the network in the next time step as s_{t-1}, and as input to the prediction $P(z)$. Subtracting the prediction from the speech signal yields the excitation e_t. The excitation is then quantized, and serves both as the training target and as input to the network for the next time step (e_{t-1}). As can be seen all training inputs (s_{t-1}, p_t, e_{t-1}) are μ-law quantized.

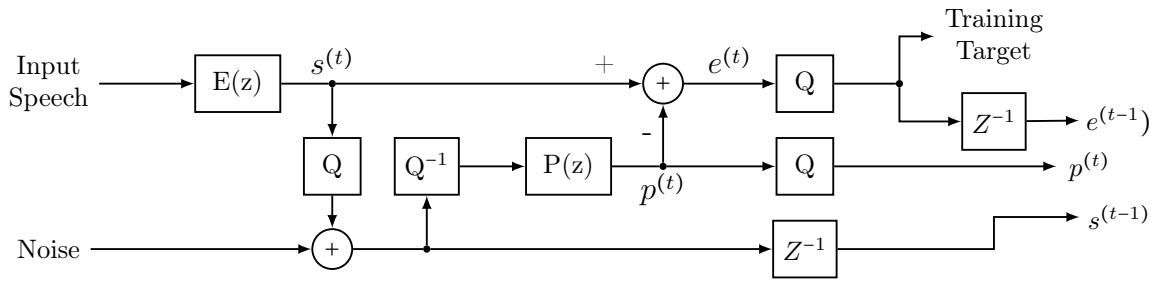


Figure 4.6: Noise injection during training, modified figure of [Valin and Skoglund, 2019a]. Note that Q denotes μ-law quantization and Q^{-1} is the reversing of the quantization. Z^{-1} denotes a one step delay. $E(z) = 1 - az^{-1}$ and $P(z) = \sum_{k=1}^{M} a_k z^{-k}$.

If we look at the figure and compare it to the overview of the LPCNet in Figure 4.1. Then we see that it is similar to start at the output s_t and then move backwards, hence the excitation e_t can be computed as $e_t + p_t = s_t \Leftrightarrow s_t - p_t = e_t$.

One should note that the input speech is not the speech produced by network, but rather the clean excitation, which is part of the pre-processing prior to the training.

Having reviewed the additional methods utilized in the LPCNet, we are now able to describe how each layer of the network interacts, and what is their individual purpose.

4.5 Layer Analysis

In this section we will go through the frame rate network, linear prediction and sample rate network respectively, and take a closer look at each of these, and their mathematical functionality. In order to gain insight of how the network's layers interact with each other, their computational complexity and their I/O relations. we will in this section only examine the computational complexity and data management in regards to inference of the network, since training would not be part of an implementation. The communication complexity will not be studied in this section, since it depends on the hardware architecture it is implemented on.

The number of weights will be accounted for and the computational complexity will be described in big O notation and subsequently by the amount of 'operations' it would take a layer to yield an output, based on the LPCNet's architecture [Valin and Skoglund, 2019a]. The term operations will in this section cover addition, subtraction, multiplication and division, hence it serves as way to count these arithmetic. Furthermore does many of the layers in LPCNet have activation function all of which requires to compute the exponential function, these will be assessed separately as these make up a more complex type of arithmetic.

Furthermore does this project not dive into the layers of computing the LPC coefficient, the prediction layer and the sampling process described in 4.3 - in regards to complexity and the associated expressions. Since the subject of LPC have been researched and described extensively in the past, hence we will make a short description of its relation to the remaining network but no deeper analysis will be given, other than the superficial exposition given in section A.3. As for the sampling process, it does only pose a minor part of the network complexity wise and at first glance is the inherent parallelism within this layer small.

4.5.1 Frame Rate Network

First we have the frame rate network. It consists of two convolutions, a 'residual' connection between the input features, and the output of the convolutions, and lastly two fully connected layers. The network receives a total of 20 features, 18 of which are"Bark-scale cepstral coefficients, and two pitch parameters (period and correlation) [Valin and Skoglund, 2019a].

Based on these inputs the frame rate network computes a 128-dimension 'conditioning' vector denoted **f**. This vector is computed for every frame i.e for every 160 samples. The frame rate network part of the LPCNet can be seen in Figure 4.7. This network have three functionalities applied to the features it is given, namely the one dimensional convolution, the fully connected layers and the embedding of the pitch.

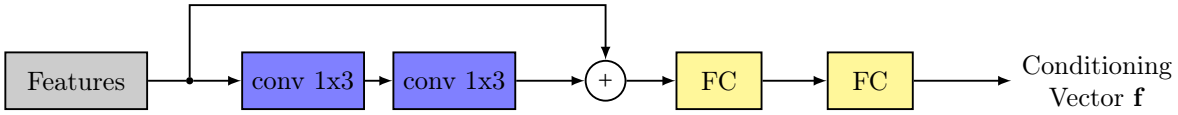


Figure 4.7: The frame rate network part of the LPCNet

Pitch embedding

The embedding of the pitch is based on the same layer that is used for embedding the gates for the first GRU (GRU_A), discussed in section 4.1. Using this embedding layer essential means, that the network will create a Look-Up Table (LUT) [Collobert and Weston, 2008]. Often these embedding are done in order to encode words, such that the words are mapped from categorical values into numerical values. It is also in these types of word related tasks that we find the use of embedding typically.

The way this embedding works by defining a vocabulary size i.e. how many different values the embedding layer will be shown. For example if the embedding layer are given the two sentences "he likes the color red" and "the color of the sky is blue" we would have a vocabulary equal to nine. In the LPCNet the vocabulary of the pitch embedding is 256, these values are often referred to as tokens.

For each of these tokens we define a number of weights, we want to represent the token. In LPCNet this number is set to 64, meaning that the embedding layer consists of a total of $256 \cdot 64 = 16\ 384$ parameters. All these weights are initialized to be a random value. The network can then tune these weights as a part of the training. This also means that some pre-processing is required in order to quantize the pitch down to only 256 different values/tokens. During inference this layer serves as a simple LUT, and therefore do not pose any arithmetically complexity. The embedding is illustrated in Figure 4.8.

From here it can be seen how if the pitch embedding layer where to receive an input of '3', the embedding will output a 64 dimensional vector containing the values $w_{3,1}, w_{3,2}...w3, 64$. Hence do the embedding layer produce more information available to the network, from just a single input. The embedded pitch is then concatenated with all 20 features and next transferred to the first convolution layer.

Convolution layers

The type of convolution performed in the two "1x3" convolution layers is so called one dimensional convolutions meaning that in the case of sound, only convolution is conducted in time. The one

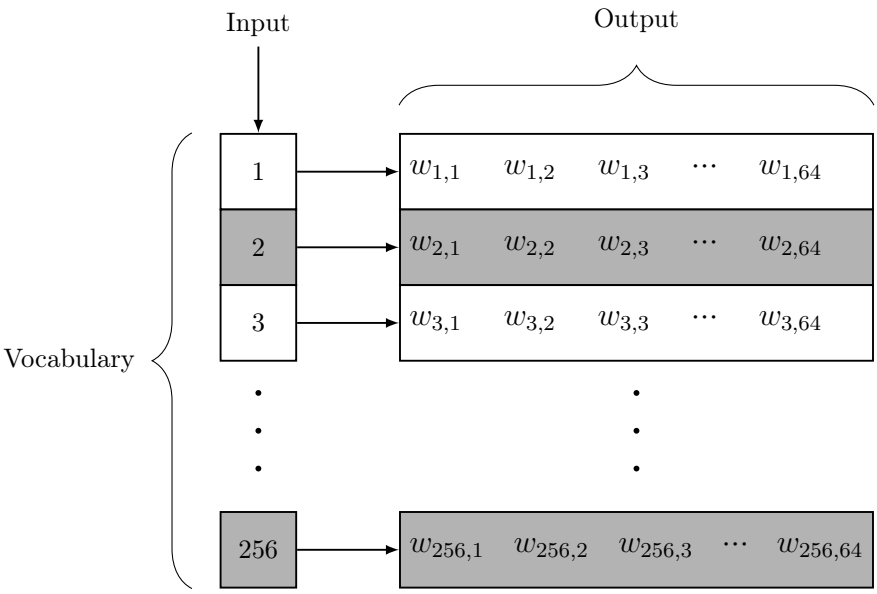


Figure 4.8: Figure visualizing how 256 possible inputs, each yields a 64-dimensional vector of weights

dimensional convolution can be thought of as sliding a kernel or filter across an input, computing the sum between the input and the kernel. One should note that it is actually the cross correlation that is computed, but it is still more often that not referred to as a convolution. The output of the convolution can be computed as:

$$O(i) = \sum_n \sum_m ((I(m, i+n) \cdot K(m,n))) + b \tag{4.11}$$

where the matrix I is the input, the matrix K is the kernel and b is the bias. As one can see, this expression handles an input with two dimensions, yet we stated that it was a one dimensional convolution. This is because the dimension is not related to the input, but rather in which dimension we move the kernel. Hence we can make convolutions on two dimensional inputs, One example of two dimensional data could be samples from multiple sensors. An example of convolution on a one dimensional input is shown in Figure 4.9.

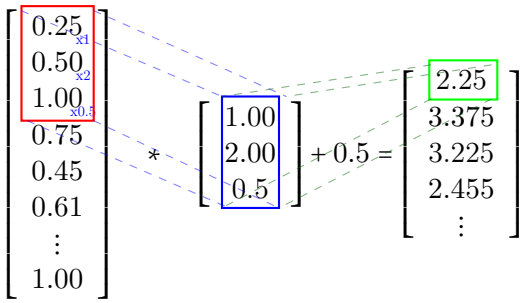


Figure 4.9: 1 dimension convolution, on a one dimensional input. The red box shows the kernel's/filter's location on the input. This is multiplied by the kernel in the blue box and a bias is added, which yield the result in the green box (output).

In Figure 4.9 the activation function of the output have been omitted for clarity, with an activation

function the output will instead become:

$$\mathbf{O}_\tau(i) = \tau\left(\sum_n \sum_m \left((\mathbf{I}(m, i+n) \cdot \mathbf{K}(m, n))\right) + \mathbf{b}\right) \tag{4.12}$$

where τ denotes the use of the hyperbolic tangent as activation function. By inspecting both the figure and the expression, we can see that computing the output of a one dimensional input at kernel position i, is no different from the method used by the perceptron in Figure 3.1, if we set $n = 3$ and the weights equal to those shown here. The scenario of course changes when the input dimension changes or when the kernel is moved, but fundamentally the two methods are very similar. In LPCNet the kernel size is equal to 3. The layers have 128 and 102 filters for the first and second convolution layers respectively, and both uses the hyperbolic tangent as activation function.

The number of (#) weights required for the convolutional layers can be computed as:

$$\#\text{weights} = F \cdot K \cdot O + B \tag{4.13}$$

Where F is the number of filters, K is the kernel size, O is the output size and B is the biases. In this expression the number of biases is equal to the number of filters used in each layer, since each filter have an associated bias. Like wise does each filter and bias produce one output, hence is the output size equal to the number of filters.

The input of the first convolution layer have the size of 102. Since the embedded pitch yields 64 values and it is concatenated with the original input features - here the reader might wounder how this adds up. This is because according to the source code, the number of features the network computes is actually 38. However 18 of these are zeroed before they are handed to the network, and thus no additional information is given to the network. None the less this causes unnecessary computations (at the very least during training), further discussion of this is given in section 5.4, but for the remainder of this chapter we will examine the network with the inclusion of the zeroed features as well. Thus the number of weights in the two convolutional layers are:

$$\mathrm{W}_{\mathrm{Conv1}} = 128 \cdot 3 \cdot 102 + 128 = 39\ 296 \tag{4.14}$$

$$\mathrm{W}_{\mathrm{Conv2}} = 102 \cdot 3 \cdot 128 + 102 = 39\ 270 \tag{4.15}$$

One important detail is how the 102 value input to the convolutional layer is handled. Because one could misinterpret, based on kernel size, how the kernel slides across the input. The data given to the first convolutional layer is a 102 valued vector, and with a kernel size of 3, the kernel is placed over three *vectors* not three of the 102 values. Hence the kernel does convolution over a total of $3 \cdot 102 = 306$ values at a time. Furthermore does the second layer also have a kernel size of 3, which means that it requires three vectors from the output of the first convolutional layer. Thus to gain one output of the second convolutional layer it needs a total of five frames, hence it has what is known as a 'receptive field' of five frames. This is illustrated in Figure 4.10. Therefore two things should be noted. First, does the frame rate network require five frames before it can yield any output. Second, when the input stream stops and the frame rate network no longer receives frames, it still needs frames to yield the last four outputs.

In [Valin and Skoglund, 2019a] this is solved by zero padding both of the inputs to the convolutional layers, yielding two vectors for each layer which contains of zeros. This can also solve the situation where one have to wait for frames to be received, if the zero padded frames is the first in a given sequence.

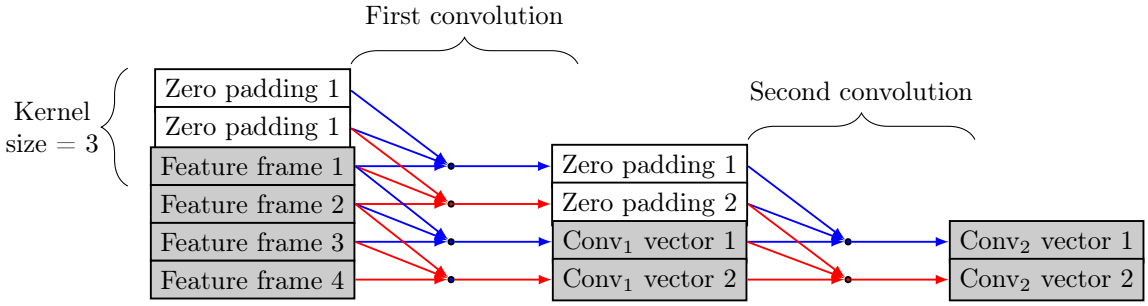


Figure 4.10: Illustration of how the inputs are convoluted into outputs of both the convolutional layers. Note that the 'feature frame' is not the actual frames recieved from the encoder, but vectors of information directly related to a given frame

Knowing how the kernels slides over the inputs frames, we can now decide on the computational complexity of the convolutional layers as:

$$\mathcal{O}_{conv} = F \cdot (K \cdot (2 \cdot I - 1)) + B + A_\tau \tag{4.16}$$
$$= 2 \cdot FKI - FK + B + A_\tau \tag{4.17}$$
$$= F \cdot (2KI - K) + B + A_\tau \tag{4.18}$$

Here A denotes the associated activation function and τ that it is the tanh activation function. I is the input size. Since the number of biases and activation is equal to the number of filters used in the layer, i.e. one activation and bias per filter, the expression can be reduced further:

$$\mathcal{O}_{conv} = F \cdot (2KI - K + 1 + 1) \tag{4.19}$$
$$= F \cdot (2KI - K + 2) \tag{4.20}$$

The input size of the first convolution layer is as mentioned before equal to 102. The activation function used in both convolution layers is the hyperbolic tangent as shown in Equation 3.2. Thus the activation be separated into three 'operations' (one sub, add and division) and one exponential function for computing the e^{2x} term.

The first convolutinal layer have an input size of 102 inputs, and yields outputs equal to the number of filters in the layer which is 128. The second layer hence have 128 inputs and yields 102 outputs. Thus the number of operations required to compute the output vector of the first convolution is:

$$\mathcal{C}_{conv} = F \cdot (K \cdot (2 \cdot I - 1)) + B + A_\tau \tag{4.21}$$
$$\mathcal{C}_{A_\tau} = 3 + \mathcal{C}_E \tag{4.22}$$
$$\mathcal{C}_{conv1} = 128 \cdot (3 \cdot (2 \cdot 102 - 1) + 1 + 3 + \mathcal{C}_E) = 78,464 + 128\mathcal{C}_E \tag{4.23}$$

where $\mathcal{C}_{conv1}$ denotes the number of operations required by the first convolutional layer (again in terms of adds, subs, multiplications and divisions), and $\mathcal{C}_E$ denotes the number of operations required to compute the exponential of e^{2x}. Like wise is $\mathcal{C}_A$ the number of operations required to compute the whole activation function (tanh).

The second convolutional layer is almost identical, with the exeption that it only have 102 filters, and an input size of 128, the number of operations is thus:

$$\mathcal{C}_{conv2} = 102 \cdot (3 \cdot (2 \cdot 128 - 1) + 4 + \mathcal{C}_E) = 78,438 + 102\mathcal{C}_E \tag{4.24}$$

the output vector is then added to the residual connection containing the earlier discussed 102 features i.e a vector-vector addition, as shown in Figure 4.7. Hence 102 additions are made and the output remains the same size, and its complexity is then equal to the number of outputs yield by the second convolutional layer, hence $\mathcal{O}_{ADD} = F$

Fully connected layers

The 102 value vector is then given to the two Fully connected (FC) layers. While we have not covered the FC layer yet, which is also called a dense layer, it is straight forward to understand. In essence is it a layer which consists of a number of nodes (or perceptrons) we decided. All perceptrons are then connected to all inputs i.e, "fully connected". This is shown in Figure 4.11. Both FC layers have 128 nodes and both use the tanh activation function. The equation to be solved by these layers is:

$$\mathbf{FC}_\tau(\mathbf{x}) = \tau(\mathbf{W}\mathbf{x} + \mathbf{b}) \tag{4.25}$$

where $\mathbf{x}$ is the input vector, $\mathbf{b}$ is the bias vector and $\mathbf{W}$ is the weight matrix. Since the both FC layers have 128 nodes, they will both output a vector of 128 values. The number of weights in these FC layers is:

$$\#\text{weights} = I \cdot N + B \tag{4.26}$$

Where I is the input size, N is the number of nodes in FC layer, and B is the associated bias. Each node have one bias associated, hence the number B is equal to the number N. The number of weights in the layers is:

$$\mathrm{W}_{FC1} = 102 \cdot 128 + 128 = 13,184 \tag{4.27}$$
$$\mathrm{W}_{FC2} = 128 \cdot 128 + 128 = 16,512 \tag{4.28}$$

The first FC layer is fed by the residual connection, hence this layer is given 102 inputs, whilst the second layer fed by the first layer. The second FC layer produce 128 outputs based on the its nodes. This output is the previously mentioned conditional vector **f**. The I/O relation of these layers is shown in Figure 4.11

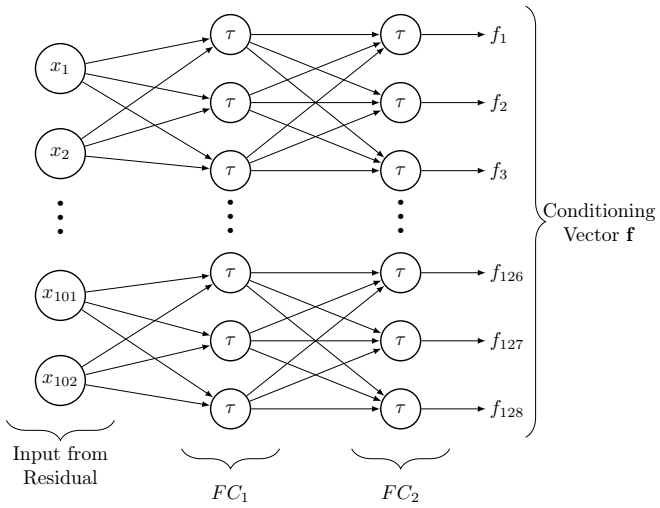


Figure 4.11: The two fully connected stage of the sample rate network, resulting in the conditioning vector

In regards to the computational complexity, we have a similar situation as with the convolution layers. For each weighted sum we add a bias and put the result through the tanh activation function. Hence the complexity can be computed as:

$$\mathcal{O}_{FC} = N \cdot (2 \cdot I - 1) + B + A_\tau \tag{4.29}$$
$$= N \cdot (2I + 1) \tag{4.30}$$

since the bias and activation function only occurs once per node in the FC layer, the expression of the complexity can be reduced to only be dependent on the input size and number of nodes.

If we compute the number of operations required by the layers in the same manner as the convolutional layers we have:

$$\mathcal{C}_{FC1} = 128 \cdot ((2 \cdot 102 - 1) + 4 + \mathcal{C}_E) \tag{4.31}$$
$$= 26,496 + 128\mathcal{C}_E \tag{4.32}$$
$$\mathcal{C}_{FC2} = 128 \cdot ((2 \cdot 128 - 1) + 4 + \mathcal{C}_E) \tag{4.33}$$
$$= 33,152 + 128\mathcal{C}_E \tag{4.34}$$

The conditional vector yielded by the second FC layer, is then given as one of the inputs to the sample rate network which we will evaluate next.

Frame rate network summary

The computational complexity, number of parameters/weights and number of operations have been summarized in Table 4.1. From the table it can be seen that this layer scales approximately linear, given that only one of the variables increases in size. Since the kernel size in the convolutional layers is relatively small, and it is typically kept that way. One could argue that the frame rate network, is scalable in terms of the initial input size, number of filters in the covolutional layer F and number of nodes in the FC layer N.

Layer	Parameter count	Complexity		
		Operations	Exponentials	$\mathcal{O}$
Embedding (pitch)	16,384	-	-	-
Convolution_1	39,296	78,464	128	$\sim 2F \cdot (KI + 1)$
Convolution_2	39,270	78,438	102	$\sim 2F \cdot (KI + 1)$
Add (residual)	-	102	-	F
FC_1	13,184	26,496	128	$\sim 2N \cdot (I + 1)$
FC_2	16,512	33,152	128	$\sim 2N \cdot (I + 1)$
Total	124,646	216,652	486	-

Table 4.1: Summary of the frame rate network's number of parameters and number of operations

4.5.2 Sample Rate Network

The sample rate network is given a total of four different types of inputs. First the conditioning vector **f** which is computed by the frame rate network. Then the prediction computed in LPC part of the network $p^{(t)}$, and lastly the time delayed excitation $e^{(t-1)}$ and the time delayed output $S^{(t-1)}$. An illustration of the sample rate network can be seen in Figure 4.12.

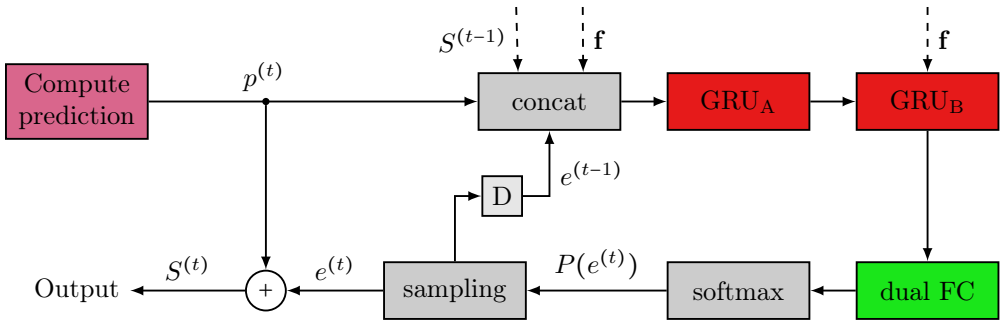


Figure 4.12: The sample rate network part of the LPCNet

All these parameters are concatenated and given to the GRUs, followed by the dual FC layer. The softmax function applied to the output of the dual FC yields the excitation probabilities $P(e^{(t)})$. These probabilities are then sampled as discussed in section 4.3, and based on the residual between the excitation and the prediction the network yields the output $S^{(t)}$. Additionally should the output be filtered by the de-empahsis filter, to reduce some of the quantization noise found at the higher frequencies [Valin and Skoglund, 2019a]:

$$D(z) = \frac{1}{1 - az^{-1}} \tag{4.35}$$

where a is equal to 0.85. This filter will reduce the power by 16 dB at the 8 kHz sampling frequency. Furthermore do we have the embedding discussed in section 4.1. The embedding discussed in this

section covered the quantized signal $s^{(t)}$, and the possibility to also embed the conditioning vector $\mathbf{f}$. In addition to these embeddings, we also have the delayed excitation $e^{(t-1)}$ and the prediction $p^{(t)}$ as input to an embedding layer.

To summarize the signals $s^{(t-1)}$, $e^{(t-1)}$ and $p^{(t)}$ are all inputs to embedding layers prior to the concatenation, and it was argued that the embedding matrix of the excitation signal $e^{(t-1)}$ can be multiplied onto the gates of the first GRU (GRU_A). The embeddings works like the pitch embedding discussed in section 4.5.1. That it is a LUT which takes the signal $s^{(t-1)}$, $e^{(t-1)}$ and $p^{(t)}$ as input and yields an output vector.

Speech sample embedding

The delayed speech signal $s^{(t-1)}$ is 8bit μ-law quantized as we saw in Figure 4.6. Hence the embedding of the signal have 256 different values to look up i.e. a vocabulary of 256, and in LPCNet each of these look ups result in a 128 value vector. Thus as discussed in section 4.5.1 this means that the layer have $256 \cdot 128 = 32,768$ parameters which the network can tune. Again since the embedding essentially is a LUT, it yields no additional computations - this embedding layer is the $\mathbf{E}$ matrix discussed in section 4.1.

Prediction and excitation embedding

The signals $p^{(t)}$ and $e^{(t-1)}$ are like the speech signal $s^{(t)}$ also 8bit μ-law quantized. Again meaning that the embedding layer have a vocabulary of 256. The difference of this embedding and the others, is that both signals $p^{(t)}$ and $e^{(t-1)}$ shares the same embedding layer. Hence this layer will also have $256 \cdot 128 = 32,768$ parameters, and it thus requires two look ups, one from each of the signals yielding two 128 value vectors.

First GRU layer

The two embedding layers outputs a total of three 128 valued vectors and the frame rate network also yielded a 128 valued vector. All these inputs are concatenated into an input vector of size 512. The conditioning vector is kept constant for the entire duration of each frame, but the other signals ($s^{(t-1)}$, $e^{(t-1)}$ and $p^{(t)}$) are all dependent on the current time. Since each frame is 160 samples, the sample rate network operates in 160 time steps. Meaning that GRU_A is given an 512 dimension input vector at every time step, where the 128 values from the vector $\mathbf{f}$ is constant, the remaining 384 values are not. Therefore we have an input vector $\mathbf{x}^t$ of size 512 for every time step t.

The GRU_A layer contains a total of have 384 GRU units, hence at each time step all units will produce an output, meaning that the output vector $\mathbf{h}^t$ contain 384 values. Thus we can now compute the number of parameters in the GRU_A layer. From section 3.2.4 we know that each GRU have 9 types of parameters:

- Input weights: $U^p_{i,j}$, $U^{\tilde{h}}_{i,j}$, $U^r_{i,j}$
- Recurrent weights: $W^p_{i,j}$, $W^{\tilde{h}}_{i,j}$, $W^r_{i,j}$
- Bias weights: b_p, $b_{\tilde{h}}$, b_r

where each input matrix $\mathbf{U}$ have dimensions $\mathbb{R}^{N \times M}$, the recurrent weights $\mathbf{W}$ are dimensions $\mathbb{R}^{N \times N}$ and the bias $\mathbf{b}$ weights are size $\mathbb{R}^{N \times 1}$. Here N is the number of units, and M is the number of inputs

in each time step. Hence the number of parameters can be computed as:

$$\#\text{Weights} = 3 \cdot N \cdot M + 3 \cdot N \cdot N + 3 \cdot N \cdot \mathbf{2} \tag{4.36}$$

$$\text{W}_{GRU_A} = 3 \cdot \left(N^2 + NM + 2 \cdot N\right) = 1,034,496 \tag{4.37}$$

The reason behind the highlighted **2**, is because in the implementation we actually have enabled, having two separated biases in the gates and candidate function. Hence we will have two biases for each of the gates (update and reset), and additional two for the candidate function, making it a total of six sets of biases i.e. $\mathbf{b} \in \mathbb{R}^{N \times 2}$

For the first GRU layer, N is equal to 384 and M is equal to 512, hence the layer holds a total of 384 GRU units and its input vector $\mathbf{x}^{(t)}$ have 512 elements. As we can see the layer holds a considerable amount of weights/parameters, it alone have over eight times the parameters of the frame rate network. Knowing the number of GRU units and inputs per time step, the sizes of the weight matrices and input vectors can be described. We have that $\mathbf{U} \in \mathbb{R}^{384 \times 512}$, $\mathbf{W} \in \mathbb{R}^{384 \times 384}$ and $\mathbf{b} \in \mathbb{R}^{384 \times 1}$. Going through the equations defining the GRU which where discussed in section 3.2.4, we have the output of a *single* GRU, which can be computed as:

$$h_i^{(t)} = (1 - p_i^{(t)}) \odot \tilde{h}_i^{(t)} + p_i^{(t)} \odot h_i^{(t-1)} \tag{4.38}$$

where $\odot$ denotes an element wise multiplication. The output is dependent on the inputs from the update gate $\mathbf{p}$, and candidate activation $\tilde{\mathbf{h}}$ which was defined as:

$$\tilde{h}_i^{(t)} = \tau\left(\sum_j \mathbf{U}_{i,j}^{\tilde{h}} \mathbf{x}_j^{(t)} + \sum_j \mathbf{W}_{i,j}^{\tilde{h}} \left(\mathbf{r}_j^{(t)} \odot \mathbf{h}_j^{(t-1)}\right) + b_i^{\tilde{h}}\right) \tag{4.39}$$

$$p_i^{(t)} = \sigma\left(\sum_j \mathbf{U}_{i,j}^{p} \mathbf{x}_j^{(t)} + \sum_j \mathbf{W}_{i,j}^{p} \mathbf{h}_j^{(t-1)} + b_i^{p}\right) \tag{4.40}$$

the candidate activation is dependent on the output of the reset gate $\mathbf{r}$:

$$r_i^{(t)} = \sigma\left(\sum_j \mathbf{U}_{i,j}^{r} \mathbf{x}_j^{(t)} + \sum_j \mathbf{W}_{i,j}^{r} \mathbf{h}_j^{(t-1)} + b_i^{r}\right) \tag{4.41}$$

however since we have separated biases the expressions changes into:

$$\tilde{h}_i^{(t)} = \tau\left(\left(\sum_j \mathbf{U}_{i,j}^{\tilde{h}} \mathbf{x}_j^{(t)} + b_{i,ip}^{\tilde{h}}\right) + \left(\sum_j \mathbf{W}_{i,j}^{\tilde{h}} \left(\mathbf{r}_j^{(t)} \odot \mathbf{h}_j^{(t-1)}\right) + b_{i,rec}^{\tilde{h}}\right)\right) \tag{4.42}$$

$$p_i^{(t)} = \sigma\left(\left(\sum_j \mathbf{U}_{i,j}^{p} \mathbf{x}_j^{(t)} + b_{i,ip}^{p}\right) + \left(\sum_j \mathbf{W}_{i,j}^{p} \mathbf{h}_j^{(t-1)}\right) + b_{i,rec}^{p}\right) \tag{4.43}$$

$$r_i^{(t)} = \sigma\left(\left(\sum_j \mathbf{U}_{i,j}^{r} \mathbf{x}_j^{(t)} + b_{i,ip}^{r}\right) + \left(\sum_j \mathbf{W}_{i,j}^{r} \mathbf{h}_j^{(t-1)} + b_{i,rec}^{r}\right)\right) \tag{4.44}$$

where 'ip' denotes the bias associated with the input sequence, and 'rec' denotes the bias associated with the recurrent sequence. These biases can be summed off-line if it is not necessary to keep them separated, but for now we consider them as separate. As it can be seen in order to compute the result

of a single GRU, one have to make multiple weighted sums. Additionally we have that the reset gate and update have an equal amount of computations required to yield a result. Starting with these the complexity of either of the gates for *all* N units within the layer are:

$$\begin{aligned}\mathcal{O}_{gates} &= N \cdot ((2 \cdot M - 1) + (2 \cdot N - 1) + 2 \cdot B + A_\sigma) && (4.45)\\ &= N \cdot ((2 \cdot M - 1) + (2 \cdot N - 1) + 2 + 1) && (4.46)\\ &= N \cdot (2M + 2N + 1) && (4.47)\end{aligned}$$

Hence we see that the computational complexity scales with the number of GRU units with in the layer, and the length of the input sequence. Hence the update and reset gates of the GRU actually scales quadratically with the number of units in the layer. Like before is one bias B and activation function A associated to each unit. The activation function of the gates is the sigmoid as shown in Equation 3.1, therefore in terms of our operation counting it consists of two operations (sub and division) and one exponential function

$$\begin{aligned}\mathcal{C}_{gates} &= N \cdot ((2 \cdot M - 1) + (2 \cdot N - 1) + 2 \cdot B + A_\sigma) && (4.48)\\ \mathcal{C}_{A_\sigma} &= 2 + \mathcal{C}_E && (4.49)\\ \mathcal{C}_{gates} &= N \cdot (2M + 2N + (2 + \mathcal{C}_E)) = 689,280 + 384\mathcal{C}_E && (4.50)\end{aligned}$$

Thus computing either of the gate vectors ($\mathbf{r}^{(t)}$ or $\mathbf{z}^{(t)}$) at time step t requires a total of 689,280 operations and 384 exponential function. The candidate function $\tilde{h}_i^{(t)}$ have almost the same expression as the gate functions except for one additional multiplication (which is element wise), and the difference in activation function which is tanh. Hence the complexity of the vector of candidate functions is:

$$\begin{aligned}\mathcal{O}_{\tilde{\mathbf{h}}^{(t)}} &= N \cdot ((2 \cdot M - 1) + (3 \cdot N - 1) + 2 \cdot B + A_\tau) && (4.51)\\ &= N \cdot (2M + 3N + 1) && (4.52)\end{aligned}$$

hence the candidate activation also scales quadratically with N. Continuing counting the operations, the candidate function have the following number of operations:

$$\begin{aligned}\mathcal{C}_{\tilde{\mathbf{h}}^{(t)}} &= N \cdot ((2 \cdot M - 1) + (3 \cdot N - 1) + 2 \cdot B + A_\tau) && (4.53)\\ \mathcal{C}_{A_\tau} &= 3 + \mathcal{C}_E && (4.54)\\ \mathcal{C}_{\tilde{\mathbf{h}}^{(t)}} &= N \cdot (2M + 3N + 3 + \mathcal{C}_E = 836,736 + 384\mathcal{C}_E && (4.55)\end{aligned}$$

lastly is the hidden state $h^{(t)}$, the expression is rather simple consisting only two multiplications, an addition and a subtraction, hence its complexity is:

$$\begin{aligned}\mathcal{O}_{\mathbf{h}^{(t)}} &= N \cdot 4 && (4.56)\\ & && (4.57)\end{aligned}$$

Hence the hidden state only scales linearly with N. In terms of number of operations it becomes $4N = 1536$. If we are considering the complexity of the entire GRU layer it becomes:

$$\mathcal{O}_{GRU_A} = \mathcal{O}_{\mathbf{h}^{(t)}} + \mathcal{O}_{\tilde{\mathbf{h}}^{(t)}} + 2 \cdot \mathcal{O}_{gates} \tag{4.58}$$
$$= (4N) + N \cdot (2M + 3N + 1) + 2 \cdot N \cdot (2M + 2N + 1) \tag{4.59}$$
$$= N \cdot [4 + 2M + 3N + 1 + 2 \cdot (2M + 2N + 1)] \tag{4.60}$$
$$= N \cdot [7N + 6M + 7] \approx 7N^2 \tag{4.61}$$

we hence see that the GRU layer as a whole scales quadratically with the number of units within the layer, and linearly with the length of the input sequence. This is in line with the fact that the "largest" type of operation performed in the GRU is matrix vector products. The total amount of operations of the GRU_A is:

$$\mathcal{C}(\text{GRU}_A) = \mathcal{C}(\mathbf{h}^{(t)}) + \mathcal{C}(\tilde{\mathbf{h}}^{(t)}) + 2 \cdot \mathcal{C}(gates) \tag{4.62}$$
$$= 2,216,064 + 1,152\mathcal{C}_E \tag{4.63}$$

The GRU_A layer thus requires a total of around 2.2 million operations and 1,152 exponential functions. One important detail is that this is the number of operations $\mathcal{C}(\text{GRU}_A)$ is only for a single time step, as mentioned earlier does every frame consists of 160 samples or time steps. Therefore is the actual number of operations per frame 355 million operations and 184 thousand exponential functions, which corresponds to 10 ms of speech!. This is discussed in section 4.6.

Second GRU layer

The output of the First GRU layer contains 384 outputs in the hidden state $h_A^{(t)}$ which is concatenated with the conditioning vector $\mathbf{f}$ (of 128 values) once again. This yields an input vector of 512 values to the second GRU layer called GRU_B. This layer only has 16 units, and hence has a vastly fewer operations. Using the same formula as in Equation 4.36, GRU_B has a total of 25,440 parameters.

The expression for the complexity of this layer is the same. Yet the difference of N = 16 instead means that the number of operations of the gate functions, candidate function and hidden state are respectively:

- gates = 16,928 + 16 $\mathcal{C}_E$
- candidate = 17,200 + 16 $\mathcal{C}_E$
- hidden state = 64
- total = 51,120 + 48 $\mathcal{C}_E$

Thus the number of operations in this layer is 51 thousand operations and 48 exponential functions per time step. Meaning that for a frame of 160 sample, the operations increases to 8 million and 7,7 thousand exponential functions.

Dual fully connected layer

Following the second GRU layer, we have what [Valin and Skoglund, 2019a] calls a dual fully connected layer, denoted dual FC. This layer is not the same as two sequential FC layer, but rather two FC layer

computed in parallel. Their result is multiplied with a weight vector and in the end summed. This is illustrated in Figure 4.13.

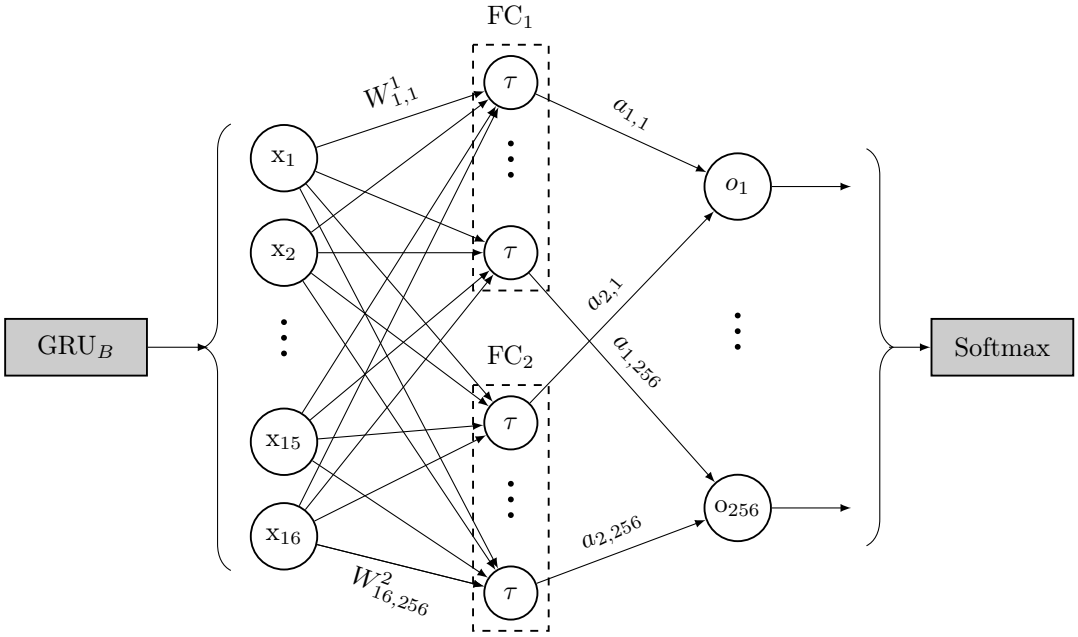


Figure 4.13: Illustration of the Dual FC layer. With GRU$_B$ as input to the layer and the softmax as the subsequent layer

The reason of having this dual FC layer is that it yielded a better result than a regular FC layer with an equal complexity [Valin and Skoglund, 2019a]. The layers expression is defined as [Valin and Skoglund, 2019a]:

$$\text{dual\_fc} = \mathbf{a_1} \odot \tau(\mathbf{W_1} \cdot \mathbf{x}) + \mathbf{a_2} \odot \tau(\mathbf{W_2} \cdot \mathbf{x}) \tag{4.64}$$

$\mathbf{W_1}$ and $\mathbf{W_2}$ are weight matrices and $\mathbf{a_1}$ and $\mathbf{a_2}$ are weight vectors. The weight matrices $\mathbf{W}$ have size $\mathcal{R}^{N \times M}$ where M is the number of inputs, that is the output of the prior GRU layer i.e. 16, and N is the number of nodes in each of the FC. The size of the weight vectors $\mathbf{a}$ is $\mathcal{R}^{M \times 1}$ where N is the number of nodes in the weight matrice. The number of parameters/weight required by this layer can be computed as:

$$\#\text{weights} = 2 \cdot (M \cdot N + B + M) \tag{4.65}$$

$$W_{Dual\ FC} = 2 \cdot (16 \cdot 256 + 256 + 256) = 2 \cdot (4,608) = 9,216 \tag{4.66}$$

Hence each of the FC have 4,608 parameters which adds up to a total of 9,216. Knowing the size of the matrices and vectors we can compute the complexity of the layer. Starting with one of the FC layers we have a matrix vector multiplication of M inputs, followed by the tanh activation function, and then an element-wise multiplication of the N (number of nodes) outputs. Thus the complexity of one of the FCs is:

$$\mathcal{O}_{single\ FC} = N \cdot ((2 \cdot M - 1) + B + A_\tau) + N \tag{4.67}$$
$$= N \cdot ((2M - 1) + 1 + A_\tau) + N \tag{4.68}$$
$$\approx N \cdot (2M + A_\tau + 1) \tag{4.69}$$
$$\mathcal{O}_{Dual\ FC} = 2 \cdot [(N \cdot (2M + A_\tau + 1)] + N) \tag{4.70}$$
$$= 2N \cdot (2M + A_\tau + 1) + N \approx 4NM \tag{4.71}$$

As always since there is N biases associated to each node in the layer, the bias provides N additions. Furthermore for the Dual FC layer as a whole, one now has two FC layers and an additional element-wise multiplication for combining both FC's output. We see that the dual FC layer scales linearly with the number of nodes in each FC and input length.

With the input to the dual FC layer being the 16 outputs of the GRU_B layer, and 256 nodes in each layer, the number of operations and exponential function is:

$$\mathcal{C}(dualFC) = 2N \cdot (2M + A_\tau + 1) + N \tag{4.72}$$
$$\mathcal{C}_{A_\tau} = 3 + \mathcal{C}_E \tag{4.73}$$
$$\mathcal{C}(dualFC) = 2N \cdot (2M + 4 + \mathcal{C}_E) + N = 18,688 + 512\mathcal{C}_E \tag{4.74}$$

This is the number of operations for every time step, and since there is 160 time steps per frame, the operations has to be multiplied by 160. Thus the total operations per frame becomes 2,990,080 operations and 81,920 power operations. The dual FC hence yields a 256 output vector which is given to the softmax layer.

Softmax layer

This layer contains simply the softmax function Equation 3.5 discussed in section 3.1. Recalling the equation of the softmax function, it compute a probability of each possible category, and with the dual FC layer, the number of categories N is 256. Thus one have to calculate the exponential of each of the N outputs yielded by the dual FC layer, sum them and compute the ratio of each N categories. The complexity of this layer can be computed as:

$$\mathcal{O}_{softmax} = 3N - 1 \tag{4.75}$$

Hence the softmax layer scales linearly with the number of categories, which is dictated by the number of nodes in the dual FC layer. Counting the number of operations done in this layer we have:

$$\mathcal{O}_{softmax} = 2N - 1 + N * \mathcal{C}_E \tag{4.76}$$
$$= 511 + 256\mathcal{C}_E \tag{4.77}$$

Thus the softmax layer requires 511 operations and 256 exponential functions. The number of operations for all time steps again increases by a factor 160, and the total number of operations is therefore 81,760 operations and 40,960 exponential functions. The softmax layer yields 256 probabilities ($P(e^{(t)})$) to the next layer which is the sampling process.

Summery of sample rate network

Here the layers of the same rate network have been summarized, as can be seen in Table 4.2 does the two GRUs have the largest computational complexity, while the dual FC and softmax layer pose less complexity. The biggest cause of increased complexity is the number of GRU units in the GRU layers and number of nodes in the dual FC layer (N), since it yields a quadratically scaling in the GRUs. Additionally does the length of the input sequence (M) also lead to increased complexity, it is however scaling linearly.

Layer	Parameter count	Complexity				
		Operations		Exponential functions		$\mathcal{O}$
		Per time step	Per frame	Per time step	Per frame	
Embedding ($s^{(t-1)}$)	32,768	-	-	-	-	-
Embedding (p_t, $e^{(t-1)}$)	32,768	-	-	-	-	-
GRU_A	1,034,496	2,216,064	354,570,240	1,152	184,320	$\sim 7N^2$
GRU_B	25,440	51,120	8,179,200	48	7,680	$\sim 7N^2$
Dual FC	9,216	18,688	2,990,080	512	81,920	$\sim 4NM$
Softmax	-	511	81,760	256	40,960	$\sim 3N$
Total	1,134,688	2,286,383	365,821,280	1,968	273,920	-

Table 4.2: Summary of the sample rate network's number of parameters and number of operations

4.6 Discussion

From the summaries of the frame rate networkand sample rate network, one can clearly see that the GRU_A layer is by far the layer with the highest complexity. It alone contributes to approximately 97 % of all operations, 47 % of all exponential functions and 82 % of the parameters. From here it is clear to see why it is beneficial to make the layer sparse, in particular, in order to make it feasible for battery driven systems.

In [Valin and Skoglund, 2019a] the weight matrices had been sparsified to a density of 0.1, that is 90% of the weight matrices entries, i.e. the layer's parameters, are zeros. If we assume that when a weight is zero, we do not perform any arithmetic. The number of operations in GRU_A becomes approximately 222 thousand operations per time step as opposed to 2.22 million and 103 thousand parameters compared to the 1.03 million of the dense GRU_A. The amount of exponential functions are only reduced if a complete row in both the **U** and **W** weight are zero along with the associated bias, thus it becomes difficult so say if the amount of exponential functions, introduced by the activation functions, will be reduced. As for now we will assume that, the number of exponential functions will remain the same.

If we make the above assumptio; the network have the computational complexity, which can be seen in Table 4.3 under 'sparse sample network'. Even though the sparse weight matrices in GRU_A makes

a significant reduction in complexity, the overall complexity of the sample rate network is still large.

As it turns out that the combined network requires roughly 4.7 GFLOPS and this exclude the exponential functions, sampling process, computation of LPC and predictions entirely. This is close to 2 GFLOPS (or ≈ 68%) more than reported in [Valin and Skoglund, 2019a].

The complexity given in FLOPS can be obtain, by multiplying the complexity of computing a single frame, by 100. Since each frames are equal to 10 ms of speech, and we thus require 100 frames to make an expression in FLOPS. An overview of the total complexity and number of parameters of both the sample rate and frame rate network can be seen in Table 4.3.

Even though GRU_A have been made sparse it still requires 3,545,702,400 FLOPS and make up approximately 76% of the total shown above, additionally does GRU_B require 817,920,000 FLOPS and make up approximately 17.5% of the total. These layers combined thus make up 93.5% for the networks number of operations (disregarding exponential functions). So if we are to reduce the computational complexity of the network, the GRUs and especially GRU_A, has the greatest potential for reducing the amount of operations done in the network.

A relative simple method that can lower the complexity is by using the multiply–accumulate operation. By using a Multiplier Accumulator (MAC) one can perform the operations of multiplying two values and then add the result to an accumulator as a single operation, halving the number of operations required otherwise. This does however require that the architecture supports the MAC, which is a rather common unit to have in a DSP or an FPGA.

If we assume that we supposedly have the multiply–accumulate operation available, and use this to solve the matrix-vector products and some of the cases of the element-wise vector-vector products in the network. We estimate that the computational complexity can be lowered to approximately 22.6 million operations per frame compared to 46,6 million.

Which means that the network would operate at 2.26 GFLOPS, again this number is in complete disregard to the exponential functions and remaining network parts (LPC predictions, sampling and de-emphasis filtering). This number is much closer to the estimated 2.3 GFLOPS described in [Valin and Skoglund, 2019a].

As stated in the beginning of this section, have this analysis completely disregarded the communicational complexity. The process of moving and storing data may very well pose a larger complexity than the computational complexity. This is further discussed during our implementation considerations.

Exactly how this network is trained, what efforts are done in order to reduce the complexity further and how this affects the quality of the networks capabilities is discussed in chapter 5.

Network part	Parameter count	Complexity			
		Operations		Exponential functions	
		Per time step	Per frame	Per time step	Per frame
Frame rate network	124,646	-	216,652	-	486
Sparse sample network	203,642	291,925	46,708,000	1,968	273,920
Total	328,288	-	46,924,652	-	274,406
Per second (100 frames)	-	4,692,465,200		27,440,600	
Per second (MAC)	-	≈ 2,250,000,000		27,440,600	

Table 4.3: Table of the parameters, number of operations and computational complexity of both the frame rate and sample rate network, in FLOPS (per second) and with the MAC assumption

4.7 Summary

In this chapter we have described the overview of the LPCNet and presented alternative ways to look at the network. We have covered methods used in LPCNet which lies beyond the general theory of the RNN and gated units (LSTMs and GRUs). Lastly have we examined the many layers of the network, and described each layers arithmetic, which layer feeds into the next and the specific complexity of each layer. Here we gained insight in which layers are the most computationally expensive, namely the two GRU layers.

In chapter 5 we will walk through the process of training the LPCNet, what it requires and how it performs. This includes looking into the hardware and software platforms, pre- and post-processing, objective quality of speech and model variations.

Chapter 5

Simulation models of LPCNet

In this chapter we will go through the steps necessary for training a working model. This includes the platform models have been trained on, both in regards to the software and hardware. The pre-processing of the training and testing data required to train and test a given model, and post-processing for producing speech files. The data set used and what method is used to measure the speech quality.

Having examined the network in depth as done in chapter 4 along with some of the basic theory behind neural networks as done in chapter 3. Allows not only to train a model, but also make meaningful augmentations of the original model. The results of both the reference model based on [Valin and Skoglund, 2019a] and augmented models, are examined in the later part of this chapter.

5.1 Training platform

In this section we shall go through the platform which LPCNet is trained on and tested on. We will cover both the software the source code is available on, and the key hardware responsible for the training.

5.1.1 Software

As can be read in [Valin and Skoglund, 2019a] is the network trained (and tested) using tensorflow[1] and keras[2] in the programming language python[3]. Tensorflow is a open-source software libary written for a wide range of programming languages, but it is only the Application Programming Interface (API) for python which have a 'stability promise'. Keras is a high level API which is made specifically for developing deep learning, it is build on top of other libraries and API such as tensorflow. Keras allows for easier use of tensorflow and generally faster prototyping, keras is written in python.

The source code for LPCNet is available at github [4], and the specific version of LPCNet presented in [Valin and Skoglund, 2019a] is commit '`0ddcda0`'. It contain the necessary code for anyone to train their own version of LPCNet or make adjustments as one see fit. Furthermore is the necessary code for "testing" LPCNet also included, that is the possibility of having the network synthesise speech based on the features described in [Valin and Skoglund, 2019a] and a trained network model. Some additional features which are not covered in the article, are seemingly still present in the code.

[1] https://www.tensorflow.org/
[2] https://keras.io/
[3] https://www.python.org/
[4] https://github.com/mozilla/LPCNet/

For faster training Nvidia[5] GPUs can be utilized. These are based on the Compute Unified Device Architecture (CUDA) developed by Nvidia. This covers both the parallel computing platform and the API for programming Nvidia GPUs. CUDA is supported by both tensorflow and keras, though sometimes for specific purposes. Utilizing Nvidia GPUs in the tensorflow/keras enviorment requires the CUDA tool kit[6] and cuDNN[7] library. In this project the following versions of the mentioned software have yielded successful training of the LPCNet:

- Tensorflow 1.14.0
- tensorflow-gpu 2.0.0
- Keras 2.3
- Nvidia CUDA tool kit 10.0
- Nvidia CuDNN 7.5.

It should be noted that minor parts of the code needs to be changed in order to meet compatibility.

5.1.2 Hardware

Initially the network was attempted trained on a common CPU, yet this showed to simply be infeasible. Since training of the network with the same batch size and number of epochs (number of times the data set have been trained on) as reported in [Valin and Skoglund, 2019a] took between approximately 50 days and six months, depending on the processing power of the CPU attempted to train the network on. It is recommended by the source code to use an Nvidia 'GTX 1080 ti' GPU or better, and its clear to see why.

It was initially attempted to perform the training on an available Nvidia 'GTX 1080', utilizing this GPU model yielded a training period of approximately seven days. This is indeed a substantial speed up, yet the bottleneck for this GPU turned out to be the lack of dedicated memory on the GPU.

So in order to train multiple versions of the network with different parameters and or layers, within the time frame of this project. It was decided to utilize the Aalborg University (AAU)'s 'Claaudia' research data service[8]. Claaudia's High-Performance Computing (HPC) solution is build around the Nvidia DGX-2 data center. The Nvidia DGX-2 is equipped with Nvidia's Tesla V100 series GPU. The use of a single Tesla V100 on the Nvidia DGX-2 platform yielded a training period of around 60 hours, which is a very significant time reduction compared to the initial approximately two to six months. It was furthermore attempted to utilize multiple GPUs for a single training instance, but this approach only provided a speed up of 5% to 10%. Instead one can train multiple models in parallel using multiple GPUs.

5.2 Data pre-/post-processing

Before the training of the LPCNet can commence, some data pre-processing is necessary, and likewise is some post-processing necessary in order to obtain the right format of the output of the network.

[5]https://www.nvidia.com
[6]https://developer.nvidia.com/cuda-toolkit
[7]https://developer.nvidia.com/cudnn
[8]https://www.claaudia.aau.dk/

5.2.1 Preprocessing

The data format used for training the network is pure PCM signals, 16 bit audio sampled at 16kHz. Since most speech data bases are not based on raw audio, one have to acquire the correct signal format, in addition to the bit depth and sample rate. In the case of this project, is the speech data based on 16-bit, 16 kHz sampled Waveform Audio File Format (WAV) files. Meaning that only the file format of the WAV file have to be removed in order to obtain the raw audio, of the correct sample rate and bit depth.

This is done with the Sound eXchange (SOX)[9] software. Which provides among other things utilities for format conversion, SOX is command line based. In this project we have used window's powershell platform, and written powershell scripts which can perform the necessary conversion of the sound data formats.

When the PCM speech signals have been acquired, one have to use the `dump_data` script provided in source code. This script makes data augmentation for the purpose of producing large amounts of training data. The augmentation consists of passing the data through different filters. The script can either provide training data or test data.

In the case of training data the script makes between two and four files depending on which commission the source code is based on. One of the files is consisting of a total of 38 features (according to the source code), 20 of which is used in LPCNet. The remaining file(s) contains the data necessary to produce the training targets and inputs, in accordance to the noise injection scheme discussed in 4.4. In this regard can the model overview shown in Figure 4.1 and [Valin and Skoglund, 2019a] seem a bit misleading. Since the reader might, based on the figure, think that it is the networks output $e^{(t)}$ and later $s^{(t)}$ which serves as a recurrent input to the network, however this is not the case during training. In reality is all the signals $e^{(t-1)}$,$s^{(t-16)}...s^{(t-1)}$ and $p^{(t)}$ pre-computed prior to the training phase.

In the case of testing, does the `dump_data` script only produce the features. Then based on the features, one can compute the LPC predictions and conditioning vector $\mathbf{f}$ i.e. the inputs to the sample rate network examined in 4.5.2. In the case of testing, we indeed have the scenario shown in overview of LPCNet in Figure 4.1. Here we use the time delayed excitation $e^{(t-1)}$, and output signals $s^{(t-16)}...s^{(t-1)}$.

5.2.2 Post processing

Using the included script from the source code, to perform the 'test' of the network. One obtains a file of the raw PCM de-empahsis filtered signals s^t (the filter is shown in Equation 4.35).

Here after we have again used the SOX software in another powershell script to convert the PCM signals to a WAV file. These WAV files can then be used to perform speech intelligibility or sound quality measurements discussed next.

5.3 Speech data and quality measures

In order to train a model one of course needs some data to train on. We will in short examine the chosen data set and its key specifications. Additionally in order to determine how well a given model performs, we will need some sort of quality measurements, which in this project is chosen to be PESQ.

[9] http://sox.sourceforge.net/

5.3.1 Speech data

The speech data set used for this project is the 'TIMIT Acoustic-Phonetic Continuous Speech Corpus'. It is a speech data set based on american english, consisting of 630 speakers with one of eight dialects. As earlier mentioned is this data set given in a sample rate of 16kHz with a bit depth of 16 bits in a single PCM channel (mono).

Since this project is focused on the algorithmic parts and implementation of the LPCNet, more emphasis have been given to the arithmetic and implementation aspects of the network. Hence this project have not tasked it self with finding a more suitable data set, in regards to cover a wider range of languages. Nor how the network responds to speech outside of the language it have been trained on, or how different encodings of the training affects the network.

5.3.2 PESQ

We wish to test the quality of multiple trained models, a subjective listening test would allow for a better understanding on the human perception of the produced speech and enable us to directly compare to other networks. Yet with the many models we set out to train, testing their produced speech will be very time consuming. So in order to save time and keep within the time frame given for the project, objective listening tests have been performed, more specifically have the Perceptual Evaluation of Speech Quality (PESQ) method been used.

PESQ is based on ITU-T recommendation P.862 and furthermore is the complementary recommendations P.862.1 and P.862.2 also used in this project, these extensions allows for MOS mapping and wideband speech (16kHz sample rate) respectively.

PESQ is an objective measurement method, which tries to estimate the subjective quality one could obtain from real listening tests, and while it is concluded that the correlation between subjective and objective score is quite high, PESQ cannot replace an actual subjective listening test. However for the purpose of this project we seek measurements, which allows us to compare tests of different models, either from multiple the models produced, future implementations of the network or similar. Hence for the sake of purely making comparisons between these and not other people's work, we see this approach as sufficient.

In this project we use PESQ with full reference, meaning that the algorithm compares a reference signal to the output of the network (degraded signal). The result of the PESQ algorithms measurement is a MOS which rates the sound quality between 1 (bad) and 5 (excellent).

The MOS is in principal the arithmetic mean of the rating given by multiple human's rating of a given stimuli, that is MOS= $1/N \sum_{n=1}^{N} S_n$, where S_n is the rating of the n'th stimuli.

The Type of MOS that PESQ produces is the so called MOS-LQO (MOS Listening Quality Objective) as defined in ITU-T .800.1. This type of MOS refers to a "The score is calculated by means of an objective model which aims at predicting the quality for a listening-only test situation." - ITU-T Recommendation P.800.1 (07/16). The recommendation state that these score can be obtained using PESQ i.e. ITU-T Recommendation P.862.1, P.862.2 and P.862.3.

PESQ (ITU-T recommendation P.862) produces a score between -0.5 and 4.5 to rate the quality of a given signal. Using the ITU-T recommendation P.862.1 describes the mapping function which maps from this rating to the MOS-LQO which allows for linear comparison with MOS. Using the ITU-T

recommendation P.862.2, we extend this mapping to take into account for a wideband signal i.e. going from a sample rating of 8kHz to 16kHz. Hence can we linearly compare the PESQ MOS presented in section 5.4.

5.4 Network models and results

For this project the LPCNet have been train on more than 3 hours of speech data from the TIMIT data, before the data augmentation performed by the `dump_data` script earlier described. When the model have been trained for er certain amount of time, the model is tested on a test file consisting of ten new speech sentences which have a combined playback time of around 30 seconds. The test file is once again converted into features by the `dump_data` script and given to the network, in order to have the network reconstruct the test file based on the features, yielding a degraded sound file. The test file and the degraded file is then compared using PESQ. The PESQ yielding a MOS based on these two files.

The first model established is based on the same training period and number of batches, i.e. a batch size of 64 and 120 epochs. This is used to establish a form of reference, for how well the network performs using the same settings as in [Valin and Skoglund, 2019a] with the TIMIT data set, this model have the name '`van_BS64E12o_L3d3`'. Thereafter have multiple models been trained with different variations compared to this reference, in order to gain knowledge on how these variations affects the performance of the network, some selected models is shown in 5.1. Different parameters have been adjusted using the aforementioned model for comparison.

First is the models with different settings in regards to batch size and increased training periods (epochs). None of these models performed better than the reference. The models which trained for longer did not improve the performance and a batch size of 32 resulted in models with different specific artifacts. For example would some yield synthesized speech which were whispering or stuttering. The models which trained for longer periods (epochs) also performed worse, thus over-fitting to the training set seems plausible.

Second is the models where some part of the architecture of the network have been changed. In general is the experience that changing the architecture by anything else than small adjustments, resulted in a worse performance. Some even to a catastrophic degree such as the 'sigmoid' model which would yield unintelligible noisy sounds.

We have trained models where we reduce the number of GRU units in the GRU_A layer, and as a result saw the score worsen. Hence it would seem that reducing the number of parameters, reduces the models capabilities to reproduce speech. In the model `van_BS32E90_grub32_L.3.33` we reduce the number of units in GRU_B from 64 to 32 and did not see as severe loss in performance, as when we reduces the units in GRU_A.

Injecting an additional layer between GRU_A and GRU_B, without being receiving the conditional vector $\mathbf{f}$, also reduces performance. Hence it would seem that the information gained from the FRN have significance for the performance.

One model which out performs the reference is the model in which the nulled features mentioned in section 4.5.1, is removed completely. Hence it would seem that the inclusion of 'features' with no information, is harmful for the performance of the network to some extend.

Model Name	Training parameters		Model Description	PESQ Score
	Batch Size	Epochs		
van_BS64E120_L3d3	64	120	Used for reference	2.1962
Networks with different batch size and epochs				
van_BS32E120_L3.41	32	120	Decreased batch size	1.2843
van_BS128E180_2_L3.29	128	180	Increased batch size and number of epochs	2.0901
van_BS256E180_L4d2	256	180	Increased batch size and number of epochs	1.7576
van_BS64E180_L3.29	64	180	Increased number of epochs	2.0535
Networks with different architecture				
edit_BS64E120_sigmoid_L3.32	64	120	Activation functions in the frame rate network have been changed to sigmoid	1.1132
edit_BS64E120_GRU1_256_L3.43	64	120	Number of units in GRU_A decreased to 256	1.5405
edit_BS64E120_GRU1_128U_L4.25	64	120	Number of units in GRU_A decreased to 128	1.0370
edit_sparse_GRUA384_AB64_BS64E120_L3.28	64	120	Injection of GRU layer in between GRU_A and GRU_B with 64 units	2.0597
edit_sparse_GRUA256_AB64_BS64E120_L3.33	64	120	Injection of GRU layer in between GRU_A and GRU_B with 64 units, GRU_A reduced to 256 units	1.5213
edit_BS64E120_GRUA128_GRUAB_64_L3.41	64	120	Injection of GRU layer in between GRU_A and GRU_B with 64 units, GRU_A reduced to 128 units	1.2434
edit_BS64E120_RNF_NEW_L3.26	64	120	Removal of unused features, instead of nulling them	2.3219
van_BS32E90_grub32_L.3.33	32	90	decreased batch size and number of epochs GRU_B has 32 Units	1.8092
edit_BS64E120_No_RNF_L3.27	64	120	No unused features and no zero padding in the convolutional layers	1.6297

Table 5.1: Overview of the trained models and their performance, with the sparsity error.

5.4.1 Source code error

All of the models in Table 5.1 does however have a critical error. The error lies in the process of making the first GRU layer (GRU_A) sparse, that is having weights within the layers weight matrices which is zero, as described in section 4.2.

The models trained this far have been based on a combination of newer versions of the LPCNet[10] and the network used in [Valin and Skoglund, 2019a] which is commit `0ddcda0` [11]. The networks architecture have not been changed from [Valin and Skoglund, 2019a], but smaller changes to the pre-processing and sparsity process have been adopted from later versions.

The error in the sparsity process have been consistent in all versions of the training models used so far, pre-trained models linked in commit `0ddcda0` [12] also have the same error. Additionally to the best of our knowledge, have this issue not been fixed yet [13].

The error in the sparsity process is made when the weights of the GRU layer is copied to a new variable. Immediately after the weights are copied to a the new variable, is the new variable indexed incorrectly. As a result the sparsity process is only made on the recurrent weights $\mathbf{W}$, but not on the biases $\mathbf{b}$ or weights related to the input sequence $\mathbf{U}$. This can be seen for both the model "van_BS64E120_L3d3" used as references for the training campaign in Figure A.7, and in trained models acquired at [12] in Figure A.8.

Efforts have been made to correct this error. By correctly indexing the variable possessing the copy of the layer's weights, and making the shapes of the remaining weight matrices compatible with the old code. This new addition yields weight matrices ($\mathbf{b}$,$\mathbf{U}$ and $\mathbf{W}$) with a combined sparsity of 89.8% as opposed to a total sparsity of 38.4% obtained by the old code. A weight distribution of a new model trained by the new addition to the code can be seen in Figure A.9.

5.4.2 Corrected models

As a consequence a new training campaign is performed, taking basis in the experiences made in the earlier training campaign. Hence the focus have been on training a new reference and a model which do not have the excess nulled features included, since this model performed the best. Two additional models have been trained as well. In these the sparsity of the weight matrices have been changed to approximately 95% and 80% respectively, again the new reference have a sparsity of approximately 90%. These are shown in Table 5.2.

It should be noted that for this second training campaign, a smaller error have been identified on the github forum where issues or different user experiences of the LPCNet is discussed [14]. The error consists in a relative loud clicking sound, which is caused by the sampling process, during which the largest positive amplitude is set to 2^{16} (16-bit depth). This seemly causes a problem for other programs or audio formats, since this value is equal to the largest negative value in two's complement. This mismatch between consensus of which value signifies the maximal amplitude would lead to this loud clicking sound. This issue have been fixed for this second training campaign, as a result we generally

[10] https://github.com/mozilla/LPCNet/ - last visited 25 February 2020
[11] https://github.com/mozilla/LPCNet/tree/0ddcda08d88dc52ceb89212ac62bbd91dd53d811 - last visited 25 February 2020
[12] found here https://jmvalin.ca/misc_stuff/lpcnet_models/ - last visited 25 February 2020
[13] Last inspection on 25 February 2020
[14] https://github.com/mozilla/LPCNet/issues - Last visited 20 May 2020

saw an increased performance of the models. The old reference model 'van_BS64E120_L3d3' PESQ score increased from 2.1962 to 2.2298 (1.5%).

If we compare the two reference models in the two tables, we can see that the change in sparsity only had little impact on the performance (around 3%), leading one to wounder if the network is over parameterized or if having a balanced sparity across the recurrent, input and bias weight matrices compensates for the loss, of the additionally 50% of the weights compare to the old reference. We will not investigate this matter any further, but only make this observation.

Since there was a suspicion about the models over fitting, we save models every 30 epoch during the training phase and tested those as well. The suspicion seems to be true as it can be seen that the reference model only trained for 60 epoch of the total 120 performs around 8.5% better. Hence the remaining models will also be test for every 30 epochs, and we will only cover the best.

From the Table 5.2 it can be seen that none of the trained models performs best after the entire training session of 120 epochs. Rather it would seem that the optimal training duration lies somewhere between 30 and 60 epochs.

This could be solved by incorporating a validation set during training, and registering when the training error begins to deviate from the validation error, which is usually done during training of other networks. The validation process have not been added to the training phase, since this was not part of the original source code and it is assumed that the addition of a validation process during training, will not change the remainder of this project, in regards of an implementation of the LPCNet. As such this can be part of some future works.

Changing the sparsity yields a model performing as expected, compared to the new reference model. Hence as seen, when the sparsity is increased the performance decreases and vice versa. It seems however that the sweet spot in regards to the sparsity of the models is around 90%, since increasing the sparsity to 95% gave a 20% worse performances, compared to the 'fix_BS64E60_L3.31' model, and decreasing the sparsity to 80% only yielded an increased performance of 2%.

Lastly do we have the model with no features that have been nulled. This model again outperforms the best reference model, thus we see no reason for not removing unused features. Thus we now have a corrected model, with little to no loss in performance compare to the first reference model.

This model (`fix_RNF_NEW_BS64E60_L3.30`) produces intelligible speech, but still struggles at reproducing the naturalness of the given speech. We analysed the models produced speech by comparing it to the reference. Here we are able to reaffirm the cause of the problem, compare to what can be head from the produced speech. The largest disruption in the speech comes at parts with voice sounds, hence during utterance of vocals this becomes very audible. A comparison between the reference and reproduced speech signal can be seen in Figure 5.1.

Model Name	Training parameters		Model Description	PESQ Score
	Batch Size	Epochs		
fix_BS64E120_L3.33	64	120	New reference model, with correct sparsity	2.1656
Reference model with different epochs				
fix_BS64E90_L3.32	64	90	90 epochs	2.3638
fix_BS64E60_L3.31	64	60	60 epochs	2.4209
fix_BS64E30_L3.31	64	30	30 epochs	2.3965
Networks with different sparsity (best only)				
fix_D005_BS64E30_L3.38	64	30	Sparsity of 95%	1.9324
fix_D020_BS64E30_L3.38	64	30	Sparsity of 80%	2.4701
Networks with no nulled features (best only)				
fix_RNF_NEW_BS64E60_L3.30	64	60	No nulled features	2.4351

Table 5.2: Overview of the new trained models with the corrected sparsity and their performance, only the best epoch of a model is included.

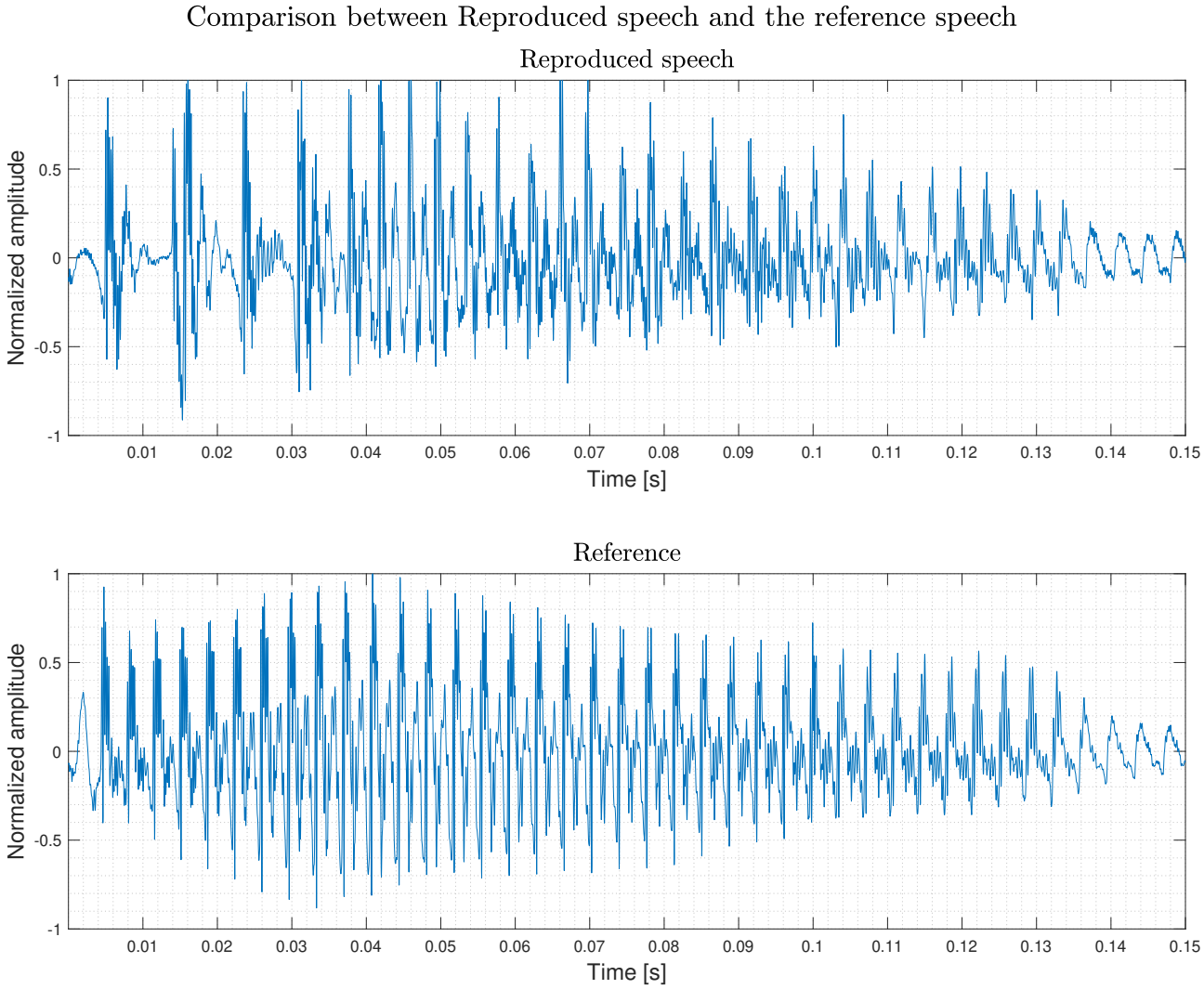


Figure 5.1: Comparison between the reference and reproduced speech signal. The model used for reproduction is `fix_RNF_NEW_BS64E60_L3.30`

The speech signal shown are from the utterance "..ha.." as in "shaving". As we can see does the model fail to correctly assess the pitch period. As a consequence does the speech sound stuttering and unnatural. This is a general problem for all models we have trained so far, and the problem happens often for voiced speech. While the issue is simple to identify, have we not been able to fix this by simple means. Hence in order to make further analysis towards implementation of this network, and not risk using the remaining time frame of the project on investigating and attempt to fix it. We will turn our focus towards algorithmic analysis, and leave the machine learning aspects for now.

5.5 Summary

In this chapter the training platform have been described both in terms of the necessary software, and how the hardware used to perform the training, can have a considerable effect on the training duration. The pre-and post-processing methods applied to the training and test data have also been examined.

A training campaign were described and a reference model was established, some alternations to the training duration were made and some changes to the general architecture were also made.

In general did these alternations to the original training duration and architecture described in [Valin and Skoglund, however one model yielded a slightly better result the 'RNF_NEW_BS64E60_L3.3' model MOS score is around 0.25 better.

A critical error was discovered in the process of inspecting the trained models, the error was corrected with some new additions to the existing code, and a new smaller training campaign was made on the basis of the old.

The second training campaign concluded with a new model, which performs better seemingly due to not over-fitting the model to the training set, at the very least not as much. It is also concluded that the exclusion of features not used in the model yielded better performance, and that the sparsity level of 90% the weight matrices, seems to be a sweet spot, in regards to the trade-off between performance and number of parameters used in the model.

Having a functional model with the correct sparsity. We will in the next chapter look into which part of the LPCNet to attempt to implement, and make a deeper analysis of the that sub-algorithm.

Part III

Architecture

Chapter 6

Sub-Algorithm selection & analysis

In the former chapters we have examined, some of the essential theory and methods used by LPCNet, the different layers of the network and most recently how a trained model have been obtained. Thus a fully functional model has now been made, able to decode speech based on the input features to comprehensible speech.

This chapter will consist of some preliminary discussion of which part of the LPCNet will be attempted to implement. Since the time frame of this project does not allow for the full implementation of the entire network.

Hereafter will the metrics of interest be outlined. It is followed by an analysis of the given part of the network, in order to discover structures of the part of the network to be implemented. Such structures contains subjects such as inherent parallelism and input output relations.

Having an overview of the sub-algorithm, allows for a discussion of which platform the algorithm can be implemented on and what impact it has on what will be in focus during an implementation.

6.1 Sub-algorithm selection

Due to the given time frame of this project, an in-depth analysis and implementation of the entire LPCNet is simply not possible. It is therefore necessary to limit the scope of this part of the project to some part of the network. To have a better basis for choosing which part to analyse, we see the necessity to cover some topics. Namely the two subjects of metrics and obtaining new knowledge. Discussing the metrics of the LPCNet helps us to identify which parts of the network, that have potential hurdles in regards to an implementation. Discussing which parts of the network we already have substantial knowledge about, helps us determine where this project can potentially convey new discoveries.

6.1.1 Metrics

Typically we emphasize four metrics for an algorithm and its implementation; accuracy of a given solution (numerical precision), how much power or energy it requires and uses, the latency and/or throughput of a given implementation and lastly the area and/or cost related to the implementation. As always tuning an algorithm to perform better on one metric, will at some point yield a worse performance on one or more of the other metrics. So there is always a trade-off to be made.

Some additional metrics which does not directly apply to the implementation, but may have influence on an implementation's design is topics such as; the possibility to protect the implementation's commercially secrets or similar, the time it takes to develop said implementation, options for updating or backwards compatibility, the structure of the algorithm, how its layout can be adjusted to existing platforms or building blocks and its scalability.

The purpose of the LPCNet is to use classical signal processing methods in conjunction with DNN. With the objective to gain the benefit of the superior accuracy of neural networks while applying handcrafted features and methods to lower the computational demands of the neural network. Having a high complexity, but relatively low compared to other neural networks, allows for the possibility of implementation on an embedded device. That is devices which typically have a limited power budget and processing power.

However the approximate 3 GFLOPS required to run the network in real-time stated in [Valin and Skoglund, 2019a], limits the potential to implement the network on embedded devices which are relatively powerful, such as a smart phone, having multiple powerful processors able to provide the necessary processing power.

Furthermore does the memory required by the network also pose a potential problem. Alone storing the approximate 330,000 weights of the sparse network would require around 1.3 Mb in single precision floating point. This may fit within the on-chip memory near the processor of more powerful embedded devices like a smart phones. But for smaller devices, one might have to store these weights on less energy efficient memory types such as DRAM. Thus affecting the energy consumption negatively, compared to less memory demanding algorithms.

Hence if the LPCNet should be implemented on smaller devices in terms of area and cost these two metrics pose the biggest restrictions. Since smaller or cheaper devices generally have less memory and processing power available, and a smaller power budget.

6.1.2 Preknowledge and potential new knowledge of the LPCNet

The LPC method utilized in the LPCNet is an old and well known method. This method have a thorough theoretical background as well as some thought out techniques to compute the necessary parameters. As such and as it has been mentioned before, does the LPC part of the network not present much new information, and we see little to no potential in this project expanding this subject.

Opposite is the situation of neural networks. The field of machine learning and DNNs does still continue to expand as resources, such as processing power and data sets sizes increases, and new software emerges - making the development of complex network architectures easier. While this field is continuously growing it is still relatively new, especially when it comes to the subjects of inference and consideration regarding how to implement networks in regards to complexity, memory and power efficiency.

The subjects of efficient implementation of DNNs is covered in [Sze et al., 2017] as a survey. Here the basics of DNNs are given as an overview along with the generic layers typically found in a neural network, namely the FC and covolutional layer. It covers two paradigms *temporal* and *spatial* architectures. The temporal architectures described in [Sze et al., 2017] are related to CPUs and GPUs similar to an Single Instruction Multiple Data (SIMD) configuration, while the spatial architectures are related to the use of FPGAs as accelerators - more on the SIMD architecture can be written in

section A.4 or [Flynn, 1972]. RNNs are not covered in this survey for two reasons; they argue that the majority of RNNs consists of weighted sums i.e. matrix vector products and more significantly, that as of the time the survey was performed, only "little attentions has been given to hardware acceleration specifically for RNNs" - [Sze et al., 2017], hence we see a larger opportunity expanding on this subject. For example as we have seen in the LPCNet, is it mainly composed of weighted sums, especially the FC and GRU layers. Therefore the necessary means for processing a RNN already exists, but maybe we can produce new knowledge on this subject.

Taking [Sze et al., 2017] as a starting point, we make a survey of what new development has been made on RNNs, in the time between the publication of [Sze et al., 2017] and now. i.e. the last two years.

6.1.3 Survey of new implementations

First we will describe some of the more general methods that have been used in order to improve on the efficiency of RNNs in some sense (e.g. improving storage efficiency, or through put). Second is the subject of hardware acceleration, these are typically linked to the usage of FPGA technology, which enables one to exploit some parts of the parameters or underlying structure of the network architecture.

General methods

In the general case of methods used to make the network more efficient, we generally see that in order to achieve better efficiency different works have given four types of methods. *Pruning* of network parameters, *quantization* of weights, inputs and/or activation functions, structured matrix based techniques and tensor decomposition.

Pruning have already been discussed in section 4.2, a new addition to this is load balance-aware pruning introduced in [Han et al., 2017]. Here they work with the realization that pruning leads to fragmented weight matrices. Hence given a number of Processing Elements (PE) (small processing units with access to their own small memory), one should balance the pruning of the weight matrices such that blocks of non-zero weights matches the number of PEs available. This however requires knowledge of the number of PEs available, and does as a whole rely on FPGA technology. All in all it resembles the block sparsity covered in 4.2.2, where the block sparsity is mostly geared towards general processing (CPUs and GPUs), and the load balanced pruning for any number of processing units.

Quantization is used to reduce the number of bits used to represent a value. The subject is already discussed in [Sze et al., 2017], and is a generally known method to reduce latency, power consumption and memory requirements. Methods used to quantize a network is to move from floating point to fixed point, and reduce number of bits from the typically used 32b to 16b or 8b for generally purpose platforms (CPUs and GPUs), or to something more fine-grained typically using an FPGA, e.g. does [Han et al., 2017] use 12b fixed point. Other means of quantization can be of approximating the non-linear activation functions such as the tanh or sigmoid by piece-wise linear functions [Wang et al., 2018] or LUT. The quantization can also be nonuniform such as the μ-law compression used in LPCNet, log domain quantization (example in [Sze et al., 2017]) or by weight sharing, which is forcing a number of weights to obtain the same weight, which for example can be done by hashing functions [Chen et al., 2015] or the k-means algorithm [Han et al., 2016]. A method used in [Wang et al., 2019] that lies somewhere in between quantization and pruning, is the 'clipped gating' method. In which the output of an activation function such as the sigmoid, is set equal to zero if its output is not above

a given threshold. This technique enables one to quantize an activation's output, since outputs lower than the threshold does not occur.

A **structured matrix** method that seems to have received some attention over the last three years, is the circulant matrix structure which is a special case of toeplitz matrices [Gray, 2005]. In a circulant matrix every row in the matrix is a right cyclic shift of the above row [Gray, 2005]. For the sake of compression and computing in neural networks it yields two things; first that if a weight matrix can be forced to attain a circulant property, the entire matrix can be described by a single row vector saving memory. Second due to the *circulant convolution theorem* the sub result of a matrix-vector (vector-vector) product

$$a_i = \sum_{j=1}^{q} \mathbf{U}_{i,j}\mathbf{x}_j \tag{6.1}$$

where $\mathbf{U}_{i,j}$ is a sub-matrix of $\mathbf{U}$, can be calculated as

$$a_i = \mathcal{F}^{-1}(\mathcal{F}(\mathbf{U_{i,j}}) \odot \mathcal{F}(\mathbf{x_j})) \tag{6.2}$$

thus can a matrix-vector product be computed with a complexity of $\mathcal{O}(pqk \log k)$ instead of $\mathcal{O}(pqk^2)$. Where each M × N weight matrix is partitioned into circulant blocks of size p × q, where these are computed as $p = M/k$ and $q = N/k$, with these block being a k × k block-circulant matrix [Wang et al., 2018, Li et al., 2019]. In these two works and in [Wang et al., 2017] were the weight matrices separated into blocks of circulant matrices of different sizes in order to achieve different compression ratios, and additionally using the FFT method above to make an FPGA accelerator for the LSTM and GRU. The block-circulant matrices have been produces as part of the training of the network [Wang et al., 2018] or additional optimization after the training of the network [Li et al., 2019]. Some of the first discussions of using the circulant weight matrices in neural networks are given in [Mathieu et al., 2013], [Cheng et al., 2015] and [Sindhwani et al., 2015].

A **tensor decomposition** was proposed by [Oseledets, 2011]. The decomposition consists of dividing a weight matrix into multiple tensors (multidimensional arrays) in such a way, that when multiplied in the correct series, it can yield any single element in the weight matrix. As a result one can achieve a significant compression of the number of parameters representing a weight matrix. For example did [Tjandra et al., 2017] accomplish a compression ratio between approximately 30 and 70 with little to no loss in accuracy. Other decomposition schemes have also been proposed such as [Thakker et al., 2019], using the Kronecker product to achieve compression. Like-wise we are aware that various matrix factorization schemes have been proposed in various literature, such as [Kuchaiev and Ginsburg, 2017] but we will not go into detail with this specific technique.

Hardware acceleration

Taking a look at the subject of hardware accelerating a DNN. Most approaches seems to be based around FPGA technology. For example as mentioned have [Wang et al., 2018] and [Li et al., 2019] used FPGAs to build an architecture which supports their FFT based scheme.

Overall the general theme for many FPGA based accelerations, is that of improving latency and energy efficiency during inference. This mainly stems from the situation that during training, the latency of a DNNs results often pose little to no problems, since the training it self rarely have an real-time application. As a consequence one can compute on batches of inputs, which makes the SIMD

architecture that many CPUs and especially GPUs utilize an effective way of computing a DNN. The training can also be performed off-line making energy efficiency less of a concern.

However during inference of a trained network in a real-time scenario the FPGA based architectures yields good results. Since for the real-time scenario the latency is critical, meaning that one can only collect little to no batches of inputs. In [Fowers et al., 2018] they propose a FPGA based topology for cloud based services. Here they show that their method vastly outperforms GPUs in terms of latency and energy efficiency as long as the batch size is sufficiently small.

In [Chen et al., 2019] they expand their 'eyeriss' accelerator, presented in [Chen et al., 2016], to their second generation. Here they present a new architecture enabling it to better scale when more computational hardware is available, as well as dealing with sparse DNN. Where non-zero elements are scattered across a given weight matrix, making SIMD like structure hard to fully utilize, this is done by introducing the Compressed Sparse Column (CSC) format shown in for example [Saad, 2003], which is a way of representing only the non-zero elements in a matrix, we will cover this in section 6.2.3. As a result [Chen et al., 2019] can skip computations where the input or weight is zero and clock-gate memory logic in order to save computational time and energy. This build on top of their earlier work in [Chen et al., 2016] where the row stationary dataflow was proposed. This dataflow aims to minimize the amount of data that have to be moved from memory to arithmetic units, and maximize data re-usage. Which is reported in [Sze et al., 2017] to use half the energy per MAC compared to having no local data reuse, we describe these deeper in 7.2.

Conclusion

Since this project is limited by a given time frame, we do not have the necessary time to both try out all these new methods, and make an implementation of the sub-algorithm which we choose. As such we try to argue which of the presented methods are the most interesting, would yield the biggest gains on some metric and which are possible to try out given the time frame.

Yet before we make this evaluation, we have to decide on the part of the network to focus on. The most significant part of the LPCNet is the two GRU layers. They have the largest computational complexity, and they process the largest amount of inputs, hence it seems plausible that these layers also have the biggest communicational complexity. They contain around 40% of the weights (after pruning) of the entire network, and they have the most complex data flow, caused by the many activation functions, element-wise operations and sequence results most be treated. Most of the remaining network also consists of matrix-vector products, hence much of the efforts of the implementing and analysing the GRU parts of the network, will likely overlap with other layers of the network. Thus by selecting the GRU layers, we cover close to 2.1 GFLOPS of complexity (assuming the sparse network and the use of MAC operations), which makes up around 92% of the complexity of the analysed network shown in Table 4.3.

Therefore we decide to focus the remaining of this project, to the analysis and implementation of the GRU layers. We will describe these deeper in section 6.2. Before that however we will consider which methods of the work we have surveyed, that we should try to integrate with the LPCNet.

In the subject of general methods, have we seen some interesting approaches primarily to compress weight matrices. The load balanced pruning scheme is not interesting as mentioned earlier, since the block sparse scheme used in the LPCNet already yield some benefits in this regard. Quantization should be tried and exploited, since it is a rather simple way to reduce all metrics except the accuracy.

Hence we should try to achieve some point where the quantization of weights and possibly computations yields this reduction, but with a sufficient accuracy.

The methods of structured matrix and tensor decomposition while able to yield significant performance gains, in regards to memory and computational complexity. they also require new training and/or optimization schemes, which we believe would take too much of the remaining time of this project, if an possible implementation of the sub-algorithm should be realized. Thus we leave these methods as possible future works.

In regards to the hardware acceleration methods, we have seen that the primary work that have been made is based on FPGA technology. The FPGA as a technology and platform is a possibility for this project, we see great potential in the works presented in [Chen et al., 2016] and [Chen et al., 2019]. This work is however targets general neural networks, and since we have knowledge of the specific network to be implemented it seems plausible to make some algorithm specific modifications. Which in return can make the implementation more efficient. These initiatives though requires that an implementation is made using FPGA technology.

Thus we envision both possibilities of an implementation on FPGA and CPU or similar microprocessors geared towards handling multiple data in parallel such as SIMD class of parallel computing, as we shall see that the sub-algorithm have great potential of parallelism.

We make the choice of platforms based on the discussion made in section 6.3, and the following analysis of the sub-algorithm in section 6.2.

6.2 Sub-algorithm analysis

Having chosen the two GRU layers as our sub-algorithm of interest, we will now make a deeper inspection of the I/O of the sub-algorithm, the dataflow, inherrent parallelism, weight matrices and the order of operations in both the GRU's main expressions (Equation 4.38 and 4.42-4.44) as well as the expressions internally, in order to better asses how the sub-algorithm should be implemented, starting the the I/O relations.

6.2.1 Input/output relations of Sub-algorithm

Extending Figure 4.12 in regards to the part of LPCNet we will be looking into for the remaining of the project we have Figure 6.1. In the figure we have the four inputs to the concatenation layer, which complies the four inputs to GRU_A into one vector of 512 inputs. The reason for considering the inputs prior to the concatenation, is that they affect how the input is computed. This can lead to some computational savings, we will address this in the section below.

The GRU_A layer takes the 512 input vector $\mathbf{x}^{(t)}$ and produce an output vector with length 384. The output vector is then again concatenated with $\mathbf{f}$, resulting in the input to GRU_B we denote it $\mathbf{y}^{(t)}$, it also have a length of 512. Based on the input $\mathbf{y}^{(t)}$ does GRU_B produce yet another output vector with a length of 16, we will referrer to this as $\mathbf{z}^{(t)}$.

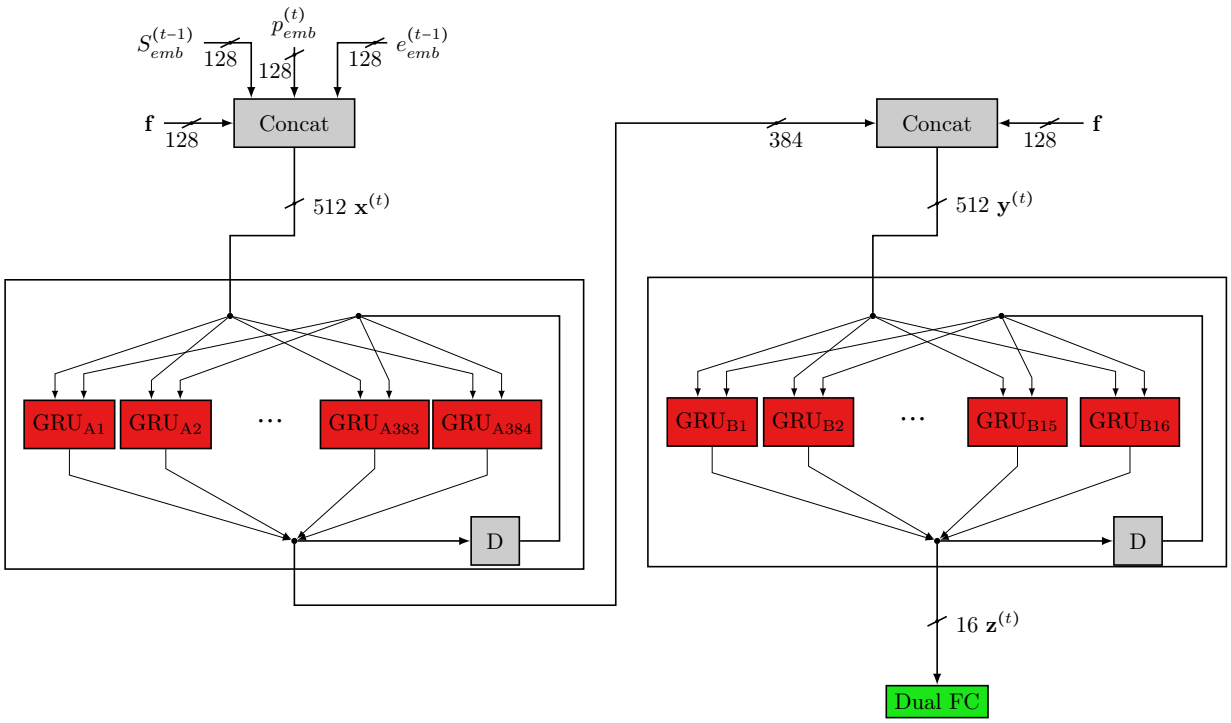


Figure 6.1: Visualization of the I/O relation and size of the two GRU layers, note that the numbers present in the figure is not the bit-width of the I/O, but rather the dimensions of I/O vectors passed through the two GRU layers.

Input to the first GRU layer

If we take a look at the matrix-vector product of:

$$\mathbf{I}(\mathbf{x}) = \mathcal{U}_A \mathbf{x} \tag{6.3}$$

where $\mathcal{U}_A$ denotes all input weights in the first GRU layer, that is U_A^z, $U_A^{\tilde{h}}$, U_A^r discussed in section 3.2.4. These input weight matrices have a dimension of $\mathcal{R}^{N\times M}$ as mentioned in section 4.5.2, hence if we concatenate these into $\mathcal{U}_A$ it will have a dimension of $\mathcal{R}^{3N\times M}$. The input $\mathbf{x}$ have the dimension of $\mathcal{R}^{M\times 1}$ and thus $\mathbf{I}$ will have the dimension of $\mathcal{R}^{3N\times 1}$. Where N is the number of GRU units within the layer and M is the length of the input vector, which is 384 and 512 respectively.

First of all does the conditioning vector $\mathbf{f}$ contain 128 values. This vector is kept constant for a whole frame that is 160 time steps. Currently this is done by repeatedly computing $\mathbf{f}$ as an input for every time step even though it will yield the same result of the entirety of a given frame. There is thus a significant amount of computations and data movement to be saved by only computing the matrix-vector product of $\mathbf{I}_A^f = \mathcal{U}_A^f \mathbf{x}^{f,(t)}$ once, where 'f' denotes the part of the input and associated weight sub-matrix related to the conditioning vector, as discussed in [Valin and Skoglund, 2019a].

Furthermore as discussed in section 4.1, do we have embedding layers between the GRU layer and the inputs $S^{(t-1)}, p^{(t)}$ and $e^{(t-1)}$. Meaning that we can precompute the matrix-vector product related to these inputs, that is $\mathbf{I}_A^{(S,p,e)} = \mathcal{U}_A^{(S,p,e)} \mathbf{x}^{(S,p,e),(t)}$, if we multiply the embedding matrices' weights with the weight of the input weight matrix $\mathcal{U}_A$. Hence to compute $\mathbf{I}$, one can simply, compute $\mathbf{I}_A^f$ once per frame, and the sum the pre-computed results of $\mathbf{I}_A^S, \mathbf{I}_A^e, \mathbf{I}_A^p$.

To summarize at the very first time step of a given frame is $\mathbf{I}_A^f = \mathcal{U}_A^f \mathbf{x}^{f,(t)}$ computed, remember that

$\mathbf{I}_A^f \in \mathcal{R}^{1152\times128}$ since we consider all input weight matrices and the 128 inputs yielded by $\mathbf{f}$. Once $\mathbf{I}_A^f$ is computed $\mathbf{I}_A$ can be computed as:

$$\mathbf{I}_A = \mathbf{I}_A^f + \mathbf{I}_A^S + \mathbf{I}_A^e + \mathbf{I}_A^p \tag{6.4}$$

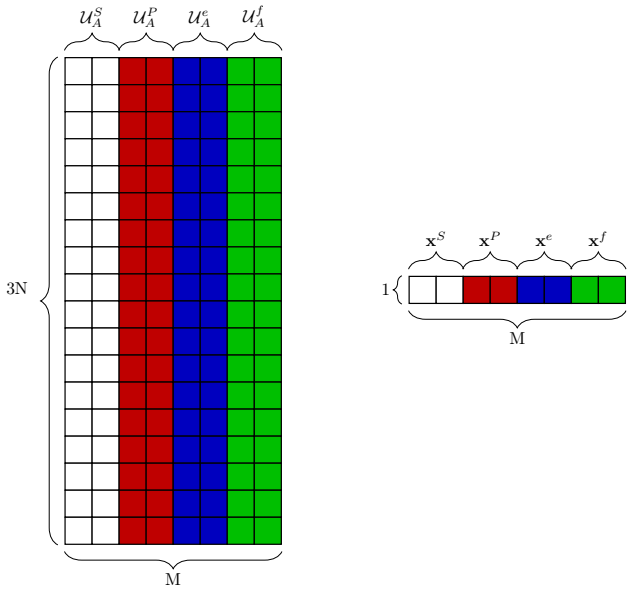


Figure 6.2: Visualization of the input matrix $\mathcal{U}_A$ and the input vector $x^{(t)}$, split into their respective parts (colors) based on the four input vectors. Note that the input vector is transposed.

This would mean that over the frame the complexity for computing the $\mathbf{I}$ before was:

$$\mathcal{C}_{\mathbf{I}} = 160 \cdot 3N \cdot (2M-1) = 960NM - 480N \tag{6.5}$$

$$\tag{6.6}$$

can now be reduced to:

$$\mathcal{C}_{\mathbf{I},emb} = \underbrace{3N \cdot (\frac{1}{4}2M-1)}_{\text{compute } \mathbf{f} \text{ once}} + \underbrace{160 \cdot 3 \cdot 3N}_{\text{Remaining frame}} = 1.5MN - 3N + 1,440N \tag{6.7}$$

As it can be seen this method reduces the computational complexity significantly, it should further be noted that the $1140N$ in the expression of $\mathcal{C}_{\mathbf{I},emb}$ are adds, and thus one can go from $160 \cdot 3N \cdot M = 480MN$ multiplications to $\frac{3NM}{4}$ multiplications, almost entirely avoding all previously necessary multiplications.

There is a considerable *but* to this discussion which we have not examined yet, and neither emphasized in [Valin and Skoglund, 2019a]. Each embedding layer is a LUT which based on one input yields 128 outputs ('embedded' versions of either $S^{(t-1)}, p^{(t)}$ or $e^{(t-1)}$). However each of these embedding layers have a vocabulary of 256, hence 256 possible permutations of the 128 outputs. Each of these

permutations can be possible inputs to the GRU_A, so if we want to achieve the benefit of less computational complexity, we have to make a LUT for each permutation of multiplying embedding layers with input weights in the GRU layer. It would as a consequence require that we made 256 permutations of $\mathcal{U}_A^{(S)}$,$\mathcal{U}_A^{(p)}$,$\mathcal{U}_A^{(e)}$, increasing the number of weights from:

$$3NM = 589,824 \tag{6.8}$$

to

$$\frac{1}{4}M \cdot N + 256 \cdot (3N \cdot \frac{3}{4}M) = 37,797,888 \tag{6.9}$$

Where N and M is equal to 384 and 512 respectively, as mentioned in section 4.5.2. As such one can decrease the computational complexity by a factor of around 225 times, by increasing the number of parameters (communicational complexity) by a factor of around 65, so there is a possible trade-off to be made. Though it might not be feasible to store these 38 million weights, as it would require around 150MB of memory in single precision floating point. Overall would it not be possible to store these weights in on-chip memory, as such it will be necessary to keep these at the slower and more energy consuming DRAM. Given that it is present on the given platform, which might not be the case on an embedded device.

It should be note that in this discussion, we have not considered the effects of the pruning process, and the fact that 90% of the weights in $\mathcal{U}_A$ are zeros. Thus it might be achievable to utilize the method above, or parts of it. This require that the weights and how the sparsity affects them are analysed.

Input to the second GRU layer

As for the GRU_B layer, it does not have any embedding layers prior to its input, hence non of the discussion above apply to this layer. Except for the $\mathbf{f}$ vector which is also fed into GRU_B. Therefore does the same considerations regarding this input apply. Namely that $\mathcal{U}_B^f \mathbf{y}^{f,(t)}$ should only be compute once per frame.

6.2.2 Inherent parallelism

Knowning the I/O relations of this sub-algorithm, we now examine the parallelism inherent to the two GRU layers, starting at inter layer parallelism, then 'intra GRU' and lastly we will look at the expressions defining the GRUs.

Inter layer parallelism

Already at this point is it possible to showcase some possible parallelism to exploit. In the case of GRU_A, it can start to calculate any of $\mathbf{I}_A^f, \mathbf{I}_A^S, \mathbf{I}_A^e$ or $\mathbf{I}_A^p$ as soon as inputs are available, likewise can the matrix-vector product of the recurrent part of the layer $\mathcal{W}_A h_A^{(t-1)}$ be computed as soon as GRU_A has yielded its output $h_A^{(t)}$. Here $\mathcal{W}_A$ denotes all recurrent weights in GRU_A (W_A^z, $W_A^{\tilde{h}}$, W_A^r).

In the case of GRU_B it can also compute $\mathcal{W}_B h_B^{(t-1)}$, when its output $h_B^{(t)}$ is ready. $\mathbf{I}_B^f$ can also be computed as soon as $\mathbf{f}$ is available. The remaining computations must wait for GRU_A's result ($h_A^{(t)}$).

It should be noted in both cases of computing the recurrent matrix-vector product of the GRU layers, there are some limitations as some of these have to wait for some intermediate results of the weighted input matrix-vector product.

Parallelism within a GRU

In Figure 6.3 we have modified Figure 3.11 to highlight the four sets of equations in the GRU unit. In this figure 'MV' denotes the matrix-vector products made within each GRU unit, that is either $Ux^{(t)}$ or $Wh^{(t-1)}$, biases have been omitted for the sake of simplicity. The term $1 - z_i^{(t)}$ means we subtract the output of the update gate from the number 1. It should be noted that in this figure we are looking at the i'th GRU unit, but when we consider all GRUs within a layer, we will subtract a vector of outputs (from the update gate) from a vector of ones, i.e. $\mathbf{1 - z^{(t)}}$.

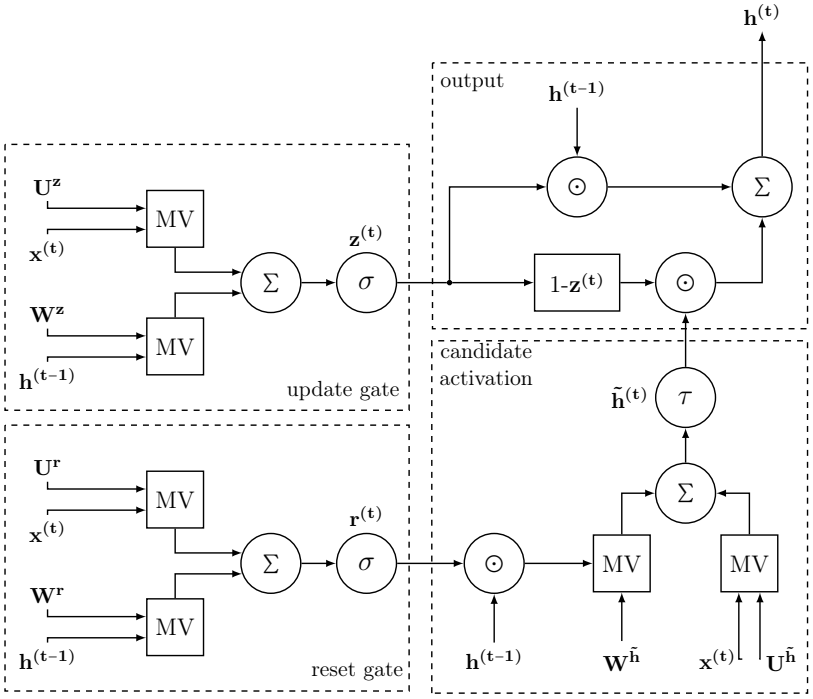


Figure 6.3: Illustration of the different parts (equations) i.e. a dataflow graph within a GRU layer.

One can see that the recurrent matrix-vector product related to the update and reset gates, can be computed immediately after $h^{(t-1)}$ is available. However in the candidate activation the recurrent matrix vector product of $W^{\tilde{h}}(h^{(t-1)} \odot r^{(t)})$ have to wait on the $(h^{(t-1)} \odot r^{(t)})$ part of the GRU.

We can make the dependencies of each of the blocks within the GRU more clear, if we make a precendece analysis on block basis, in similar fashion to [Fowers et al., 2018]s "critical path" for their analysis of the LSTM. This is shown in 6.4. Here it should be noted that these blocks that have to be processed, do not have the same complexity neither computationally nor in communications. In both Figure 6.4 and 6.3 the '$\sum$' denotes the summation of the input related matrix-vector product, the recurrent related or the summation of the candidate activation $(\tilde{h}^{(t)})$ and the old cell state of the GRU $(h^{(t-1)})$. The summations, activation functions (σ, τ) and the subtraction $(1 - z_i^{(t)})$ all work on the inputs vector-wise. Either by additions, subtractions or exponential functions, these vectors all have the size of $R^{N \times 1}$.

From these figures we can see that the critical path is through the reset gate, the candidate activation and last the output of a GRU $(h^{(t)})$. Hence if for some reason it should prove beneficial to compute on multiple of these blocks at a time. One should be aware of the order that the blocks is computed in, such that time and or energy is not wasted on waiting for a necessary sub result. However with the highly vectorized nature of the matrix-vector products, activation functions and so on. It might be possible that these operations can be parallelized in such a way, one can work on one block at a time,

and still be computationally bound in an energy efficient manner. We will examine these blocks next.

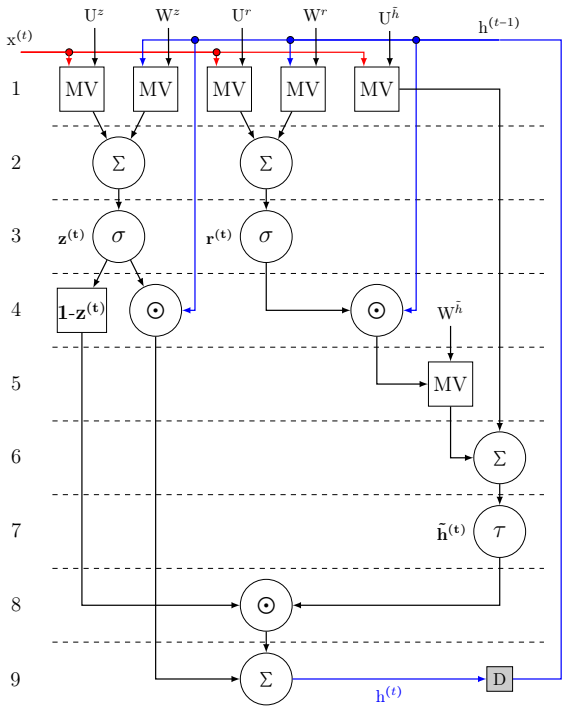


Figure 6.4: Precedence analysis of a GRU on block basis. Note that the different blocks are not comparable in computational complexity or amount of data they process/move. Hence we cannot relate the time it takes to compute a single block, but we gain insight on which blocks are dependt on results from other blocks.

Intra block parallelism

The GRU as presented in these last two figures have six different blocks as mentioned earlier. The summations is the additions of two vectors with each others, hence is each summation the addition of two vectors with $R^{N\times 1}$ elements. The same goes for the element-wise multiplications and the vector-wise subtraction.

Since none of these blocks have operations that depends on each others within the block it self, it is possible to make all of these operations (add/sub/mult) in parallel, should one have the necessary logic to do so, this is visualized in Figure 6.5 showing the precedence of these blocks. Here 'x' and 'w' can be any two operands such as an input and a weight. Each operation yield a single output 'o' which again is independent of any other outputs.

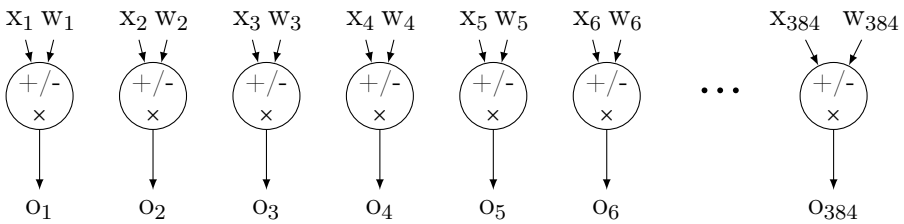


Figure 6.5: Precedence graph of the vector-wise addition, subtraction and multiplication

The next block to examine is the matrix-vectors product blocks. These also have a significant parallel potential, since it essentially can be split into a number of independent vector-vector products, equal to the numbers of rows in the matrix. The vector-vector product is visualized by the precedence graph

in Figure 6.6, here each input 'x' is multiplied with its corresponding weight 'w' and summed to yield the i'th output corresponding to the i'th row of the matrix. Every multiplication in the entire matrix-vector product can be made in parallel given that the operands x and w are present. Here after are the sub-results of the multiplication in each vector-vector product summed by the use of an adder tree.

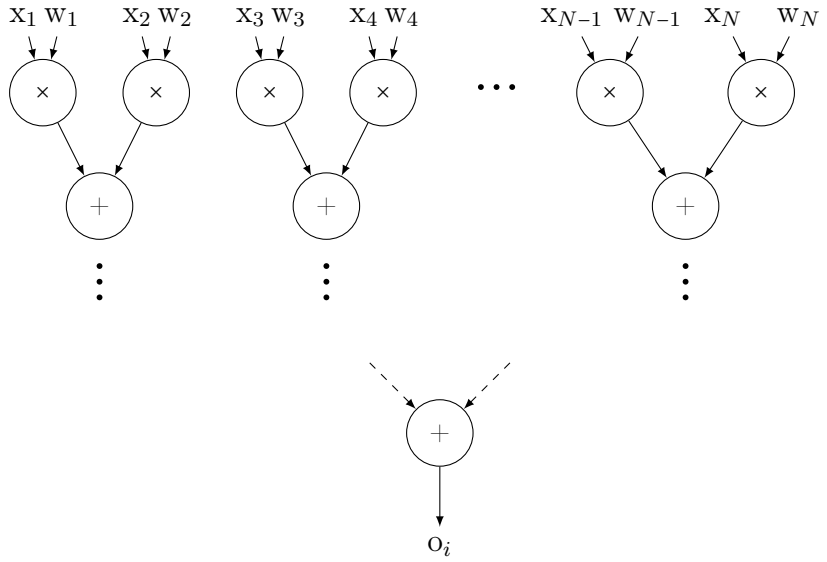


Figure 6.6: Precedence graph of a given vector-vector products within each 'MV' block, and the subsequent summation of their results.

Last we have the two activation functions tanh and sigmoid. These blocks consists of taking an input vector of the size $R^{n\times 1}$ and transform each element of the vector according to their respective expression (Equation 3.1 and 3.2), yielding an output vector of the same size. The transformation of each vector element is possible to do in parallel, since each of the elements are independent of each other.

Exactly how the exponential within the expression is computed depends on the selected method. For example can this be approximated by a number of piece-wise linear function, in [Wang et al., 2018] both the sigmoid and tanh is approximated by 22 segments. Another approximation is by use of the Coordinate Rotation Digital Computer (CORDIC) algorithm [Volder, 1959], and lastly one could compute it by the use of a power series.

6.2.3 Weight analysis

To gain a deep insight of the network we also look into the weights of the network. There is primarily two subjects that we want to confirm; first that the weights indeed are pruned in blocks of 16 values except of the diagonal, that is confirming that the method discussed in section 4.2 holds. Secondly we also want to reaffirm the discovery made in[Valin and Skoglund, 2019a]s, that the embedding layers have learned to convert from μlaw to linear.

Visualization of block-sparse weight matrices

An important factor of the LPCNet's relatively low complexity is due to the fact that GRU_A is pruned to achieve a sparsity of around 90%. As we have discussed multiple times, the pruning means that we have to store only the non-zero weights. The number of computations should like-wise be reduced, since we know any multiplications with an operand of zero, yields a result of zero. However if a given

implementation is not geared to exploit these zero weights, the potential of lowering the complexity may not be achieved. Having the weights grouped in blocks of 16 makes the task of identifying which weights to operate on much simpler. To confirm that the weight matrices of GRU_A indeed consist of blocks of 16 either non-zero or zero weight, we have visualized the weight matrices by making an image where white pixels corresponds to weights which are non-zero and black pixels to zero weights.

An example of this visualization can be seen in Figure 6.7, where we have shown the three input weight matrices concatenated that we have denoted $\mathcal{U}$ earlier. In the figure the x-axis represent the internal N GRU unit's three input weights with the GRU_A layer. The y-axis is the weights related to the input sequence $x^{(t)}$, hence it can be seen that the figure indeed have a dimension of $\mathcal{U} \in R^{3N \times M}$ (1152×512).

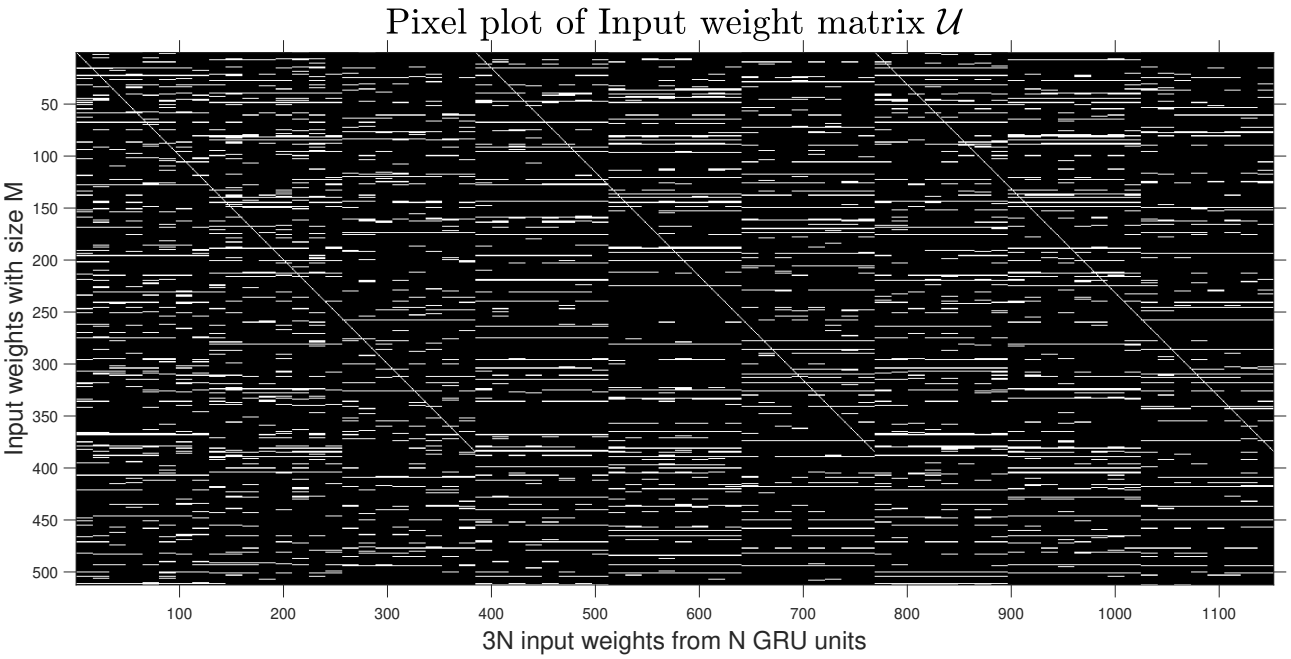


Figure 6.7: Transposed pixel plot of $\mathcal{U}$, with a white color denoting non-zero weights, and black denoting zero weights. Note that this pixel plot of the weight matrix does actually have a sparsity of around 90%, yet the visual impression may have a deceiving appearance.

While it can be difficult to see the individual weights, we can see a more general picture. First is the diagonal of each input weight matrix maintained as intended. Second the remaining weights, other than does in the diagonal, are seemingly distributed unstructured.

To better see the block distributed weights we look at upper 80×80 indices of the sub-matrix of $\mathcal{U}$ in Figure 6.8. Here the block-wise distribution of the weights is seen, and as before does the blocks seem to be distributed unstructured.

By visualizing different models we can further see, that from model to model there is no distinct pattern in where the blocks lie in the weight matrices. Thus in a given implementation we can rely on that the diagonal of the recurrent and input weight matrices always have weights. The remaining blocks of weights should be considered to be randomly distributed, hence we cannot rely on further heuristics. The biases are also distributed in blocks of 16, but being a vector there is no diagonal, and hence all weights are also distributed randomly.

With this information we can now better devise a plan for indexing the weight matrices correctly. The method for indexing might depend on the platform the network is implemented on, since different

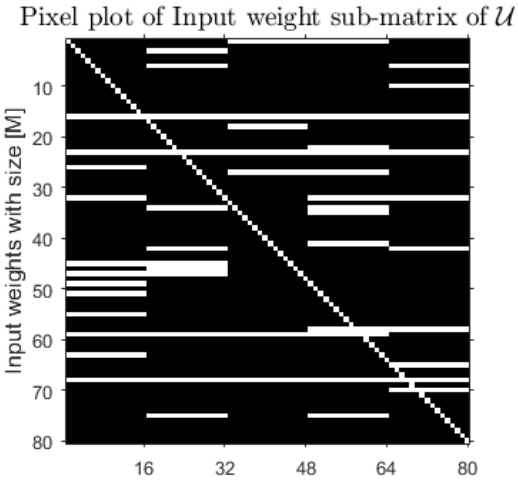


Figure 6.8: 80×80 submatrix of Figure 6.7

hardware architectures may have different way to store and handle these sparse matrices.

A method used in multiple works is the CSC matrix format as discussed earlier. One example of its implementation can be found in [Chen et al., 2019], an example of the CSC format is given in the same work, shown in Figure 6.9.

	c						**j**
a	d				**h**		k
b		**f**					l
	e			**g**	i		

CSC data compression

Data vector {**a**,b,**c**,d,e,**f**,**g**,**h**,i,**j**,k,l}
Count vector {1,0,0,0,1,2,3,1,1,0,0,0}
Address vector {0, 2, 5, 6, 6, 7, 9, 9, 12}

Figure 6.9: CSC format example given in [Chen et al., 2019]

In this example there is a total of 32 elements in the matrix, but only 12 values which is non-zero (a through l). The two vectors 'count vector' and 'address vector' signifies where non-zero data lies, and the 'data vector' what data can be found within the matrix.

Starting at the first element which is 'a'. The count vector tells one how many zero elements before non-zero data can be found at the start of each column. We see for the first column there is one, here after is there 0 non-zero elements before the element 'b'. From the address vector we can see, that we should expect two data elements before we move on to the next column.

In the second column a total of three non-zero elements hence the address vector is 5. In the case of the column with no data the address vector is simply repeated, as found in column four and seven.

In this example a total of 33 entries are used to represent 32 elements of data, however for large sparse matrices memory can be saved. For example if we had another row of only zero elements neither of the vectors would increase in size. One can achieve a reduction in memory using the CSC format, if the Number of Nonzero (NNZ) elements is smaller than:

$$\text{NNZ} < \frac{n \cdot (m-1) - 1}{2} \tag{6.10}$$

Where n is the number of columns and m is the number of rows, with a sparsity of 90% it is easy to see that this method can lead to memory savings. However, this method does not make any assumptions how the elements are structured in the matrices. Hence if we use the knowledge we have just obtained about the structure of the weight matrices, we should be able to improve the method further.

Thus in the next section will we present our thoughts on how we can further improve the memory savings, by taking into consideration the structure of the weight matrices of GRU_A.

Block compress sparse column

Knowing that the weights in GRU_A lies in either blocks of 16 or in the diagonal, can help us achieve a further reduction in memory compared to the standard CSC format. If we instead remove (set to zero) each diagonal of $\mathcal{W}$ and $\mathcal{U}$ from the weight matrices, such that $\mathcal{W}$ and $\mathcal{U}$ only contains blocks of weights, we denote them $\mathcal{W}_b$ and $\mathcal{U}_b$,Performing this on Figure 6.8 yields Figure 6.10.

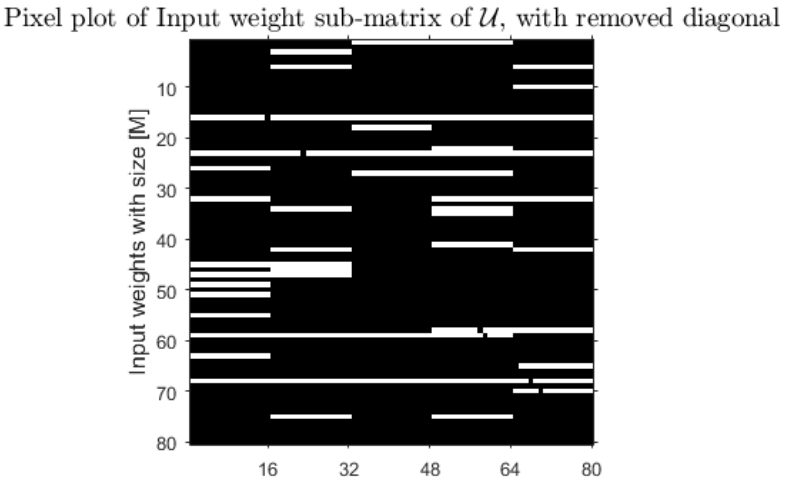


Figure 6.10: The submatrix of Figure 6.8, with no diagonal

Note that since $\mathcal{W}$ and $\mathcal{U}$ are three matrices concatenated they have a total of six diagonals, e.g. can we see three 'diagonals' in Figure 6.7. We furthermore save these diagonals as vectors, lets denote these $\mathbf{v}_\mathcal{W}$ and $\mathbf{v}_\mathcal{U}$, we can now recreate the matrix-vector products as:

$$\mathcal{U}x^{(t)} = \mathcal{U}_b x^{(t)} + \mathbf{v}_\mathcal{U} \odot x^{(t)} \tag{6.11}$$

$$\mathcal{W}h^{(t-1)} = \mathcal{W}_b h^{(t-1)} + \mathbf{v}_\mathcal{W} \odot h^{(t-1)} \tag{6.12}$$

Hence we achieve the same result as before, at the cost of 6N additional additions and multiplications. In return we can now reduce the count and address vectors of the CSC format by a factor of 16 for $\mathcal{W}_b$ and $\mathcal{U}_b$. Since we now only have to index for every 16'th column. A comparison of the storage required by the CSC format and Block Compressed Sparse Column (BCSC) can be see in Table 6.1

If we take the matrix $\mathcal{U}$ as an example it has a total of 589,824 entries (1152×512), where around 10% of these elements are nonzero. The storage required by the CSC format is then be around 119,000 elements a reduction of approximately 4.96 times. The BCSC format on the other hand would require around 63,900 elements a reduction of around 9.23 times. Hence using the BCSC format we nearly half the number of elements required to store the matrix $\mathcal{U}$. In Figure 6.11 we have made an example

Vectors	CSC format	BCSC format
Data vector	NNZ	NNZ
Count Vector	NNZ	NNZ/BS
Address vector	# columns + 1	# columns/BS +1
Diagonal vector	-	# rows
Total	2NNZ + # columns + 1	NNZ + (NNZ+# columns)/BS +1 + # rows

Table 6.1: Comparison between the two matrix formats, # denote 'number of' and BS denotes block size (i.e. 16)

of the BCSC format, excluding the diagonal vector ($\mathbf{v}_{\mathcal{U}}$ or $\mathbf{v}_{\mathcal{W}}$) for easier comparison to the former CSC example.

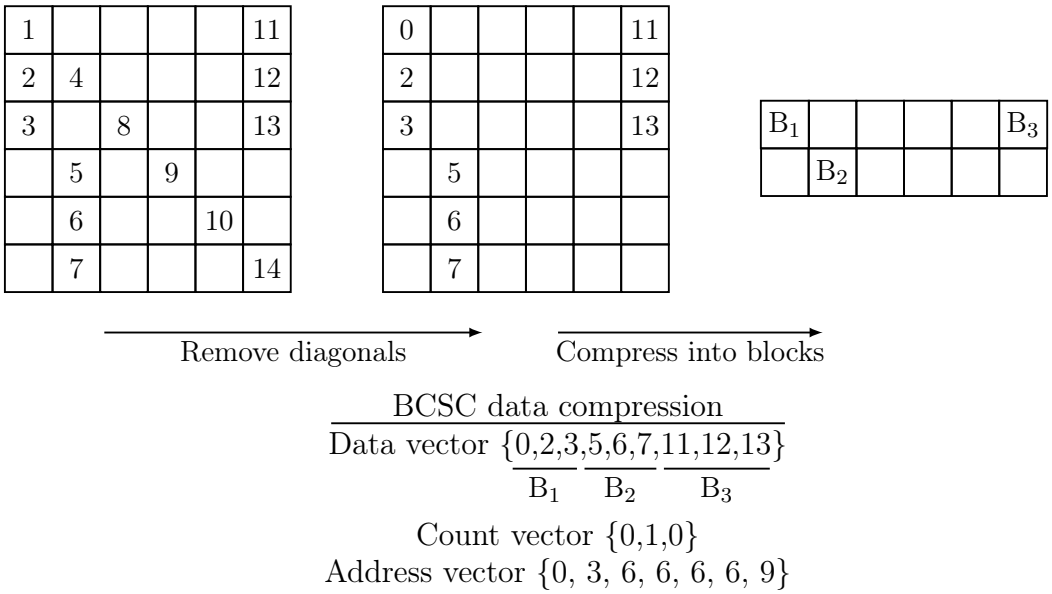


Figure 6.11: BCSC format example, note that for simplicity the block size in this example is set to 3.

In the case of the bias vectors associated to the input and recurrent weights, these have no diagonals but the weights are still in blocks of 16, thus BCSC format should still work as anticipate. In Table 6.2 we have collected the expected number of elements to be stored using the different formats of this section.

Matrix	Standard format	CSC format	BCSC format
$\mathcal{U}$	589,824	119,000	63,900
$\mathcal{W}$	442,368	89,000	48,200
$\mathcal{B}$	2,304	465	250
Total	1,034,496	208,465	112,350
Reduction	-	496%	921%

Table 6.2: Expected number of elements to be stored using different formats

Furthermore it is worth considering the fact that, while the weights in the data vector might need to be in float32 i.e. single precision, the values of the count and address vectors are mere positive integers. Hence they can be stored as unsigned integers with a size of 16bits, since neither of these vectors holds values above 2^{16}, requiring smaller bit representation.

Thus we have examined the weight matrices of the sparse GRU_A, and discussed methods for saving memory on these highly sparse matrices. Both covering methods already in use e.g. the CSC format utilized in [Chen et al., 2019], and additionally proposed our own thoughts on how this format can be further improved. If we make the (valid) assumption that elements of the sparse matrices, indeed are gathered in blocks of 16 elements at a time, leading to further improvement on memory savings.

Embedding layer weights

In [Valin and Skoglund, 2019a] they where able to visually confirm that the embedding layers have learned the function which converts from μ-law to linear, i.e. μ-law expansion. We want to reaffirm this, however it is not mentioned how the embedding layers were visually inspected. Purely inspecting the weights of the embedding layers, did not yield any meaningful link to μ-law expanding. However taking the mean of each output vector of the embedding layers, and plotting them gives a function-like expression. An example of a trained model's embedding layer weights can be seen in Figure 6.12.

In this figure we have plotted the input entries i.e. the vocabulary of the embedding matrices, against the mean of each corresponding output vector. We have shown a scaled version of the μ-law expansion for comparison, i.e. the function to be approximated in order to convert from μ-law to linear.

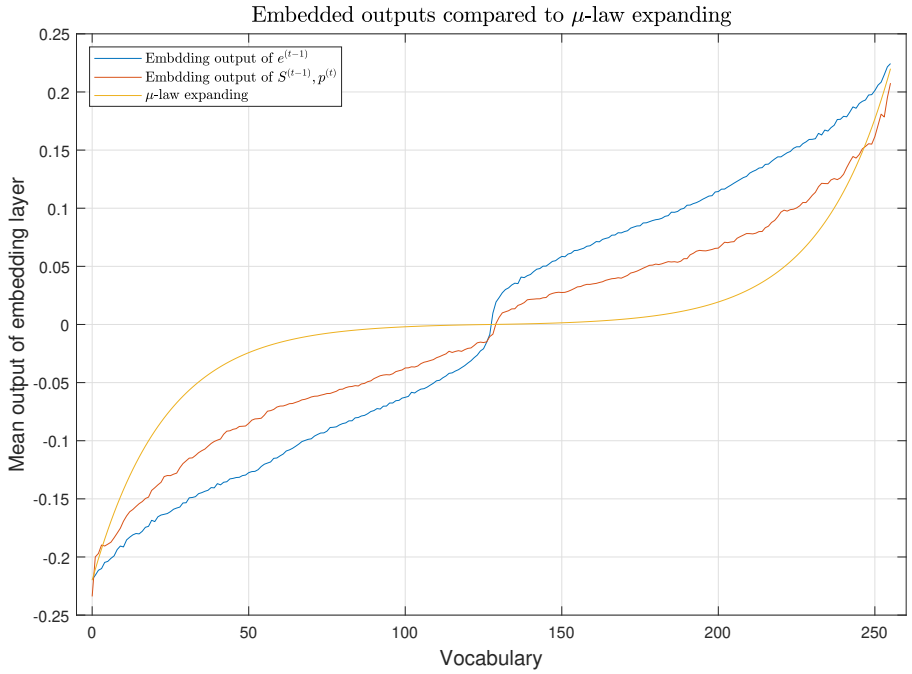


Figure 6.12: Comparison between the μ-law expansion function and the mean output of the embedding layers.

As we can see does neither of the embedding layers mean output matches the μ-law expansion function, yet they seem to make some approximations towards it but in a more linear manner. They do correctly change sign when the expansion function does, and seems to make a jump in order to make this change.

Yet we cannot truly confirm that the embedding layers have learned the expansion function, yet they seem to merely approximate it. A thing to keep in mind this is mean of the output vectors, and it is exceedingly difficult determine how the individual weights of the embedding layers interact as a weighted input to the GRU_A layer. So while we cannot completely reaffirm the claim made in [Valin and Skoglund, 2019a] it seems plausible.

6.3 Platform Choice

In the sub-algorithm selection, we concluded that the GRU layers posed as the biggest complexity of the network even when considering the effects of pruning. As such we have chosen the GRU layers for a deep inspection. We have also dived into some of the resent works in regards to implementing these RNNs and GRUs in an effective manner. Here we have described how methods have been proposed, some as a general way to compute the main contributor to computational complexity - the many matrix-vector product necessary for the GRU or LSTM variants of the RNN. Others have focused on ways to hardware accelerate exsisting networks, by the means of FPGA technology, in order to exploit the massive parallelism with these RNN methods, gaining latency, throughput and energy efficiency in scenarios where inputs cannot be collected and computed in batches.

We then took closer look at the chosen sub-algorithm, in order to better understand the necessary data that have to be processed and in which order. We looked into some of the statements made in [Valin and Skoglund, 2019a] on how to lower the computational complexity at the cost of memory. We also described the various degrees of parallelism which exists inter- and intra-layer of the GRU and the parallelism within the mathematical expression of the GRU itself.

Subsequently we visualised some of the weight matrices of interest namely the GRU_A's pruned weight matrices and the embedding layers. As a result we came up with the idea of BCSC as a method to exploit the structure within GRU_A's weight matrices in order to save memory. As can be seen having the specific algorithm that is the LPCNet instead of only considering the GRU as a general case, we are able to identify algorithm specific methods in order to possibly achieve high efficiency, for example the case of the BCSC.

We will look into possibilities for an implementation in two architectures FPGA technology and a CPU. The FPGA allows for more degrees of freedom and as have been described in 6.1.3 this have the potential to achieve higher efficiency in throughput, latency and energy consumption, than compared GPUs and CPUs. The choice of including a CPU, is based on the fact that the majority of embedded devices are based on CPUs or CPU like principals such as the DSP. It also represent the use of SIMD which many CPU like devices rely on.

In order to save time the analysis made in chapter 7 and 8, will be based on existing devices already in use. Hence we will limit the project to platforms used by RTX in order to demonstrate how their hardware will perform on a given implementation. We will try investigate the scalability of said implementation in a shallow manner. Yet we will not look into scenarios for example such as how the algorithm will perform, if we do assume infinite hardware and similar extremes.

In this chapter we have made a decision of, which part of the LPCNet the remaining of the project is limited to, this limitation is made in order to go more in-depth with the chosen sub-algorithm within the given time frame of this project. The choice of sub-algorithm is based on which metrics seems to be the most troublesome, for an implementation of the entire LPCNet, and additionally we have examined which methods already exists for making an effective implementation, and identified possible topics where this project might provide new knowledge.

Based on the chosen sub-algorithm an analysis is made, to study possible parallelism to be exploited, how layers interact with each other and lastly a weight analysis including how weights of the sparse GRU is structured. Leading to the proposed BCSC data format, as a means to further improve existing data formats for sparse matrices.

Finally we have discussed which platforms to study for an implementation of the sub-algorithm, which include both and FPGA based and a CPU based approach. Which is further limited to already existing platforms currently in use by RTX, in order to limited the assumptions on what a given platform can perform and what hardware-wise elements are available.

Chapter 7

FPGA based implementation of sub-algorithm

In this chapter we will examine the possibilities of an implementation of the LPCNet sub-algorithm on FPGA technology. We will take basis in the embedded hardware blocks available on the FPGA in order to create so called Processing Elements (PE). With these PEs we will outline the necessary functionalities, which we need to take care of in order to produce the correct output of this sub-algorithm, and we will provide considerations along the way. However we do not intent to describe all these on Register Transfer Level (RTL), instead we will have the discussion on an abstraction level somewhere between RTL and processor level. We do this in order to retain a clear link between hardware assignment, and the algorithm which we wise to run on this FPGA. Since our chosen sub-algorithm as described in section 6.2, is some what large and complex.

In order to relate this implementation to a platform already in use, and make appropriate assumptions of the hardware available we will basis in FPGAs used by RTX. Thus we will make our assumptions based on the IGLOO2 FPGAs.

7.1 Presentation of block topology

A central part of this block based functionality is where the computations are made, these are sometimes called Processing Elements (PE) and will typically contain a MAC unit, registers and some local memory with relative few Kbits called Register File (RF) available for each PE, i.e. the essentials for making standard arithmetics (additions, subtractions and multiply accumulate). Global buffers and routing network then provides the required data to these PEs, examples of PEs is shown in 7.1 and other examples can be found in [Chen et al., 2019, Sze et al., 2017, Beric et al., 2008]. In the example one can see that this topology build around a hierarchy of memory types and PEs to make the required computations.

In this chapter we will lean towards a similar topology, where we only consider these essential PE blocks and not dive into how these should be realized on the RTL. This choice is based on multiple earlier works such as [Chen et al., 2016, Sze et al., 2017], and the fact that many FPGAs have these embedded hardware blocks with the purpose to perform these standard arithmetics. Typically, these are based around a MAC unit and some necessary memory to feed this unit, i.e. the essentials to perform DSP functionality.

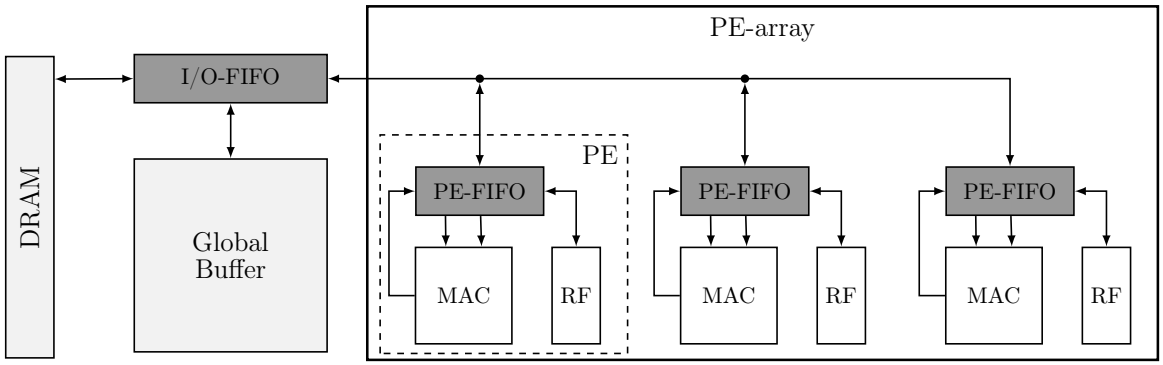


Figure 7.1: Example of memory hierarchy of Processing Elements in array from [Chen et al., 2016]

If we compare this topology to the hardware blocks available in the IGLOO2 FPGAs; we have the 'math' blocks which among other things are able to perform the MAC operation. So called 'micro SRAM' (μSRAM) blocks are able to assume the task of the RF, as there is one block per math block. Lastly is there the 'large SRAM' which holds more memory capacity, but less flexibility on how data is accessed compared to the μSRAM. Since the large SRAM blocks have to be accessed synchronously and in limited types of operations, while the μSRAM can be accessed both asynchronously and synchronously, and have a wider range of operations. The large SRAM blocks are suitable as the global buffer.

7.2 Data movement schemes

An advantage of having this hierarchical memory system, is that data can be processed near where the computations are being made. Moving data near these PEs have a clear advantage energy-wise, in [Chen et al., 2016] they compare the cost of moving data from four different levels of this hierarchy to an Arithmetic logic unit (ALU) within a PE, it can be seen in Table 7.1. The Global buffer and RF are based around on-chip SRAM technology, and these types of memory is typically faster and more energy efficient, but is also more expensive to produce, have less storage capacity per required area compared to the off-chip DRAM. Hence one have to make a trade-off between memory capacity or energy efficiency and latency.

Memory type	DRAM	Global buffer (100-500kB)	PE-array (inter-PE)	RF (0.5-1 Kb)
Normalized energy consumption	200	6	2	1

Table 7.1: Comparison of data movement at different memory levels normalized to a MAC operation, "extracted from a commercial 65 NM process" [Chen et al., 2016]. PE-array refers to moving data between two PEs, as seen in Figure 7.1.

Thus from an energy perspective, one should use the on-chip memory as much as possible, e.g. by reusing data as much as possible before reading new data from less energy efficient memory. In [Sze et al., 2017] four different schemes for data flow for neural networks is shown based on different earlier works. These four data handling schemes are Weight Stationary (WS), Output Stationary (OS), No Local Reuse (NLR) and Row Stationary (RS).

As the name suggest does the WS data scheme intent to keep the weights 'stationary', in the sense

that when read from higher level memory into the RF within the PE, one performs as many operation on that weight as possible. This scheme seeks to maximize the use of weights in convolutional layers, where a given weight is used multiple times on different inputs in a sequence or pictures, every time an input is given to a layer within the convolutional layer. LPCnet is a RNN, hence we only multiply the input and weights once per input, since the GRU layers only use matrix-vector products when it interacts with weights. Thus would this scheme seem ineffective compare to the other scheme will we present.

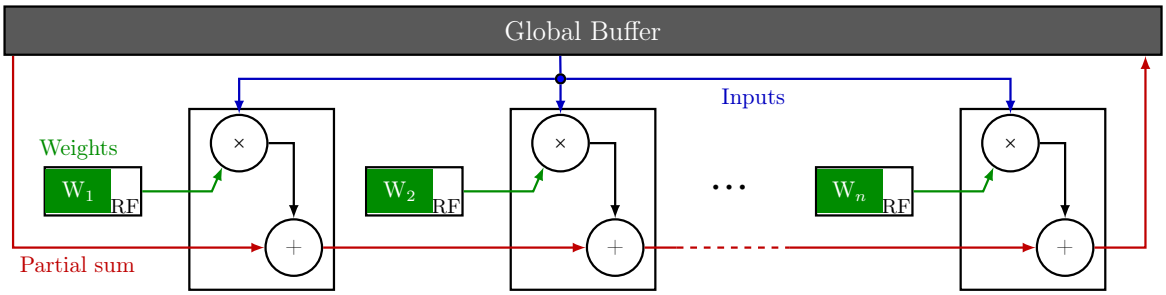


Figure 7.2: The weight stationary data scheme, weight remain stationary within each PE for a number of operations, inputs are read from higher level memory (e.g. a global buffer), the partial sums are accumulated across the PE array - modified version of example given in [Sze et al., 2017]

The OS scheme seeks to minimize energy consumption of moving the partial sum between PEs. Hence each PE keep their partial sum within the RF memory, the input can then be streamed through the PE array and weights can be broadcasted to the PE array. This approach also mainly seems to be related to convolutions, where a given input have to be multiplied by a plural of weights.

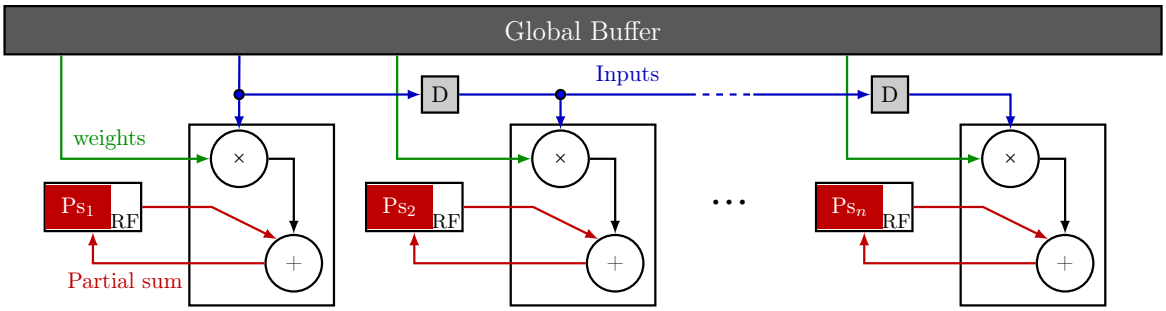


Figure 7.3: The output stationary data handling scheme. The partial sum is kept stationary in the RF of the PE, the inputs can be streamed to the PE array one by one, and all weights transmitted individually - modified version of example given in [Sze et al., 2017]

The NLR do not keep any data in the RF, but instead this memory is added to the global buffer. By using the NLR scheme one gains increased storage capacity, reducing the number of read necessary from less energy efficient memory types such as DRAM. Instead one have to broadcast inputs to all PEs, transmit the weight individually and possible accumulate the partial sum spatially as done in the WS, the scenario is shown in 7.4.

The RS dataflow scheme seeks to reuse at much data overall as possible, hence to reuse weights, inputs and partial sums as much as possible. This scheme is again immediate thought-out for the use in Convolutional Neural Network (CNN)s, since a PE can make a 1d-convolution of a filter in a vector with an input in an vector, and then utilize the RF to reuse these as much as possible, an example can be seen in 7.5

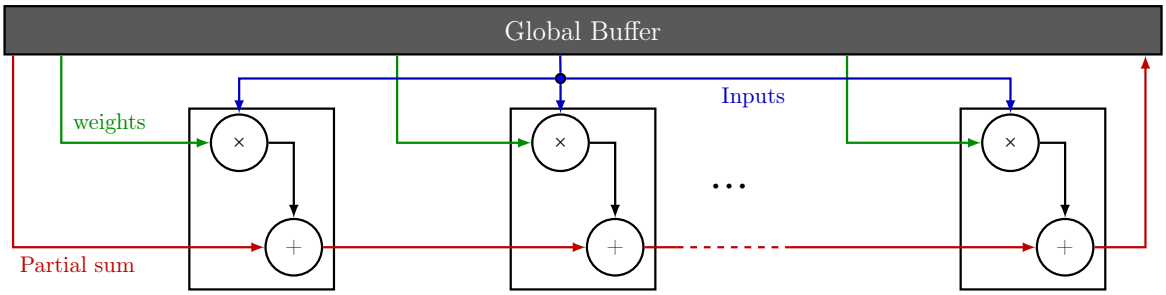


Figure 7.4: The no local reuse scheme. The RF is added to the global buffer to increase over all capacity. Inputs can be broadcasted, weight transmitted individually and the partial sum can be accumulated across the PE array - modified version of example given in [Sze et al., 2017]

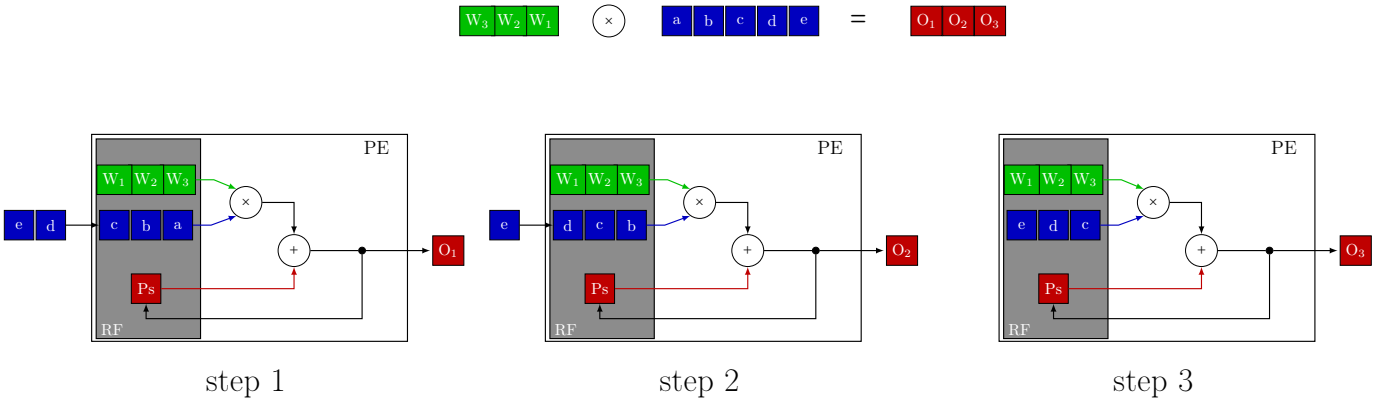


Figure 7.5: The Row stationary data handling. The RF holds a small number of weights, inputs and a partial sum. The example shows a 1D convolution of three weights with five inputs, resulting in 3 outputs, each new output requires 3 MAC operations - modified version of example given in [Sze et al., 2017]

While the RS does not focus specifically on any of the three inputs to the MAC process, the method of attempting to reuse most data overall leads to good results. In [Sze et al., 2017] a simulation is made based on the well known CNN "AlexNet" [Krizhevsky et al., 2012]. The RS data scheme performs best in regards to energy efficiency both for the network over all, but also for the FC layers which consist of matrix-vector products. The majority of our chosen sub-algorithm consists of GRUs as we have described in section 6.2, and these heavily uses matrix-vector products when the weighted sums are to be computed. In [Sze et al., 2017] the energy cost of moving data is normalized using a MAC operation as reference as shown in 7.1.

Seemingly because FC have poor potential for reuse weights, hence all of these schemes have to access the DRAM more often than in the convolution layer. Yet the RS scheme is optimized for reusing the input and partial sums, and thus overall performs better than the others.

The second best scheme in the FC layers is the NLR. Again since the relatively low data reuse the layer itself poses, contributes to the necessity of making many DRAM reads. Hence for the presented data handling schemes RS seems to have the biggest potential in regards to energy efficiency. Should the RS scheme not be possible or feasible due to unknown factors, the NLR is the second best option and its very simple design should make it possible to implement in any scenario. The exact numbers and methodology can be found in [Sze et al., 2017]. Having covered the basic topology will we use, and some consideration on how to make energy effective dataflows. We can now begin to develop on

an implementation which is tailored towards the chosen sub-algorithm.

7.3 Design of data flow

In this section we will try to make some considerations of how to implement the dataflow of the sub-algorithm. We will use the presented topology in order to achieve this and relate to the presented data handling scheme. Bear in mind that the objective of the remaining chapter, will not be that of a completed architecture on how implement the sub-algorithm, rather we seek to identify problems, structures an possible pitfalls. Which should enable the reader to make their own implementation, in a way they see fit based on the descriptions made here.

As we have discussed in 6.2 is the sub-algorithm composed of the two GRU layers and the data in and out of these layers. We also know the structure of the GRU layers, which parts can be computed in parallel and which cannot. From the various parts of this work we know that GRU_A and GRU_B works exactly the same, they do however have a difference of the weights layout, GRU_A is sparse while GRU_B is dense. Hence the computations to be made are the same for both GRUs but the way data should be handled is significantly different.

In the following sections we will examine six topics:

- Sparse matrices of GRU_A and matrix-vector product in subsection 7.3.1
- Additional on these weight matrices including the weight diagonal and biases in subsection 7.3.2
- Activation functions in subsection 7.3.3
- Vector-subtractions in subsection 7.3.5
- Element-wise multiplications in subsection 7.3.4 and 7.3.6
- Short on GRU_B in subsection 7.3.7

These block should cover the essentials of the necessary functionalities which the GRU layer holds. Of these blocks are the sparse matrices by far the most complex, have the most computations and data to handle, hence we will focus most of our efforts on this section.

The task to handle the sparse weight matrices of GRU_A is difficult, because we seek to minimize the memory required to store the weights of GRU_A, while also make computations in the correct order and minimizing the data movement by maximizing the data reuse. Ideally we would only have to store the non-zero weights of GRU_A, however as we have seen in 6.2.3 one have to spend additional memory for knowing the correct indexation of the sparse matrices. Opposite is the situation of GRU_B where one works with the weights in a more straight-out manner.

7.3.1 Sparse matrices of GRU_A

We will start this section of by examine the first and most difficult GRU layer. The issue is as earlier described the sparse weight matrices of this layer. Because while it is in theory possible to know, where a given block of non-zero weights is located within the sparse matrices at compile time, in practise it is very hard and a cumbersome task. This is further complicated by the fact that the non-zero weight block's location is difference from model to model. Hence it seems simpler to make an architecture capable of handling a general case, regardless of where these weight blocks are located within the sparse matrices. Following the graph inspired by precedence graph of a GRU in Figure 7.6 we start by inspecting the input weight matrices which we earlier have denoted $\mathcal{U}$.

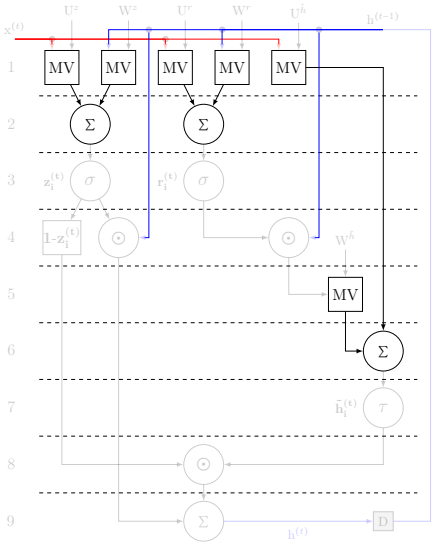


Figure 7.6: Currently we are addressing, matrix-vector products and subsequent summations.

Weight block distribution and initial thoughts

In Figure 7.7 we show the weights related to the update gate $\mathbf{U^z}$ of GRU_A to the input $\mathbf{x^{(t)}}$. As we know and as can be seen, does the matrix hold non-zero weights in blocks of 16 weights (vertical white lines) and in the diagonal (diagonal white line). To make the necessary matrix-vector product to compute the result of the update gate $\mathbf{U^z}$, one have to multiply the matrix with the input vector $\mathbf{x^{(t)}}$.

In simple terms we have to multiply each row of the weight matrix with the input vector and the sum the result of all products yielding a single value. Hence as we know the matrix-vector product of the update gate results in N outputs. Looking at the weights of the update gate for the input vector in Figure 7.7.

A matter is that each column in the weight matrix do get multiplied by only one of the 512 elements of the input vector, when we make the matrix vector product of $\mathbf{U^z x^{(t)}}$. Thus one could dissemble reusing the input to compute all the multiplication associated to that specific input, such that one would only have to give the PE that specific element of $\mathbf{x^{(t)}}$ once, thus reducing the number of times that value have to be moved between memory significantly i.e. 'input stationary', when we multiply one element of $\mathbf{x^{(t)}}$ with a column of $\mathbf{U^z}$.

However this would require that we have to compute on N different partial sums. It is simply not feasible to this case to have 384 partial sums stored within the RF, as a result one would have to constantly read and write the partial sums to less energy efficient memory types such as the global buffer or DRAM. Like-wise if we want to only work with one partial sum at a time i.e. OS. That is multiplying all elements of $\mathbf{x^{(t)}}$ with one row of $\mathbf{U^z}$, the partial sum could be stored within the RF of a PE, but it will be necessary to work with M different inputs - which again is not feasible to store within the same RF.

From here on out we will ignored the diagonal of the weight-matrix, and in the following work we will only consider the blocks of weights, we will address the diagonal weights later in section 7.3.2.

The micro SRAM only holds 1,152 bits worth of memory. Hence if we assume the weights, inputs

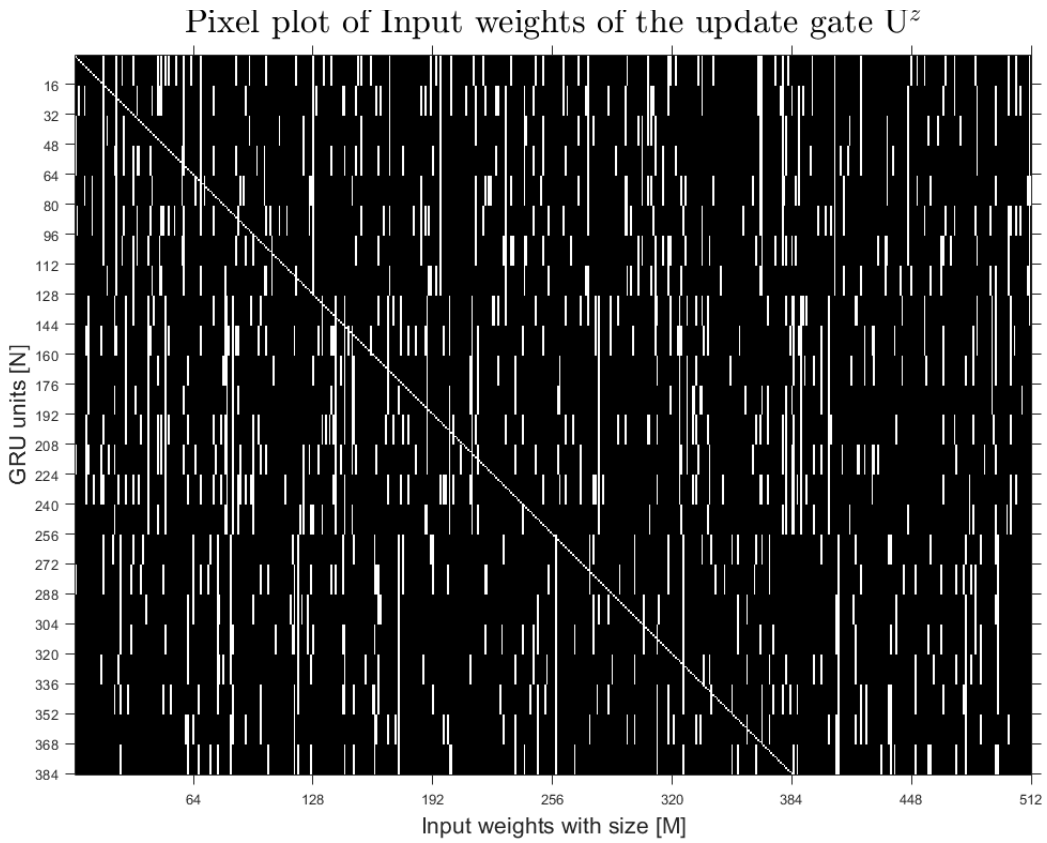


Figure 7.7: Pixel plot of the weights related to the update gate $\mathbf{U^z}$ of GRU_A, white cells denotes non-zero weights and black zero weights.

and partial sums all can be stored as single procession float i.e. float32. We require 32 bits per value and thus the RF can hold a total of 36 values in total. However as we have seen in numerous earlier works such as [Li et al., 2019, Wang et al., 2018, Chen et al., 2016, Han et al., 2017] the precision is typically reduced to 16 bits or lower. If we assume that all values can be represented by 16 bits, the RF can store a total of 72 values. Still it is neither large enough to store N (384) partial sums or M (512) different inputs.

Hence for these two described scenarios the trade-off lies between, constantly reading and writing N partial sums or M inputs values to and from the RF of the PE. Instead we will consider looking at one block of weights within the weight matrix at a time. These blocks consist of the 16×1 non zero weights, further we will denote any given set of 16 rows which the block resides in as a 'block-row'.

If a PE is working on one block at a time, it means that we can reuse the input 16 times i.e. number of weights in the block, at the cost of working on 16 different partial sums. As in any of the case weights are only used once hence it makes no sense to store these in the PE for reuse. This idea is shown in Figure 7.8, we will call this scheme Block Stationary (BS). This approach resembles the OS in the way we keep a block of partial sums stationary within the RF, and the RS since we keep the input or column stationary also for span the of a block.

Hence with the BS scheme we handle 16 rows of a given matrix at a time, by processing one block at a time. Thus for every block of weights in the matrix $\mathcal{U}$, we store one input of $\mathbf{x^{(t)}}$ and for every block-row we store 16 partial sums in the RF of the PE. Every weight have to be read from the global

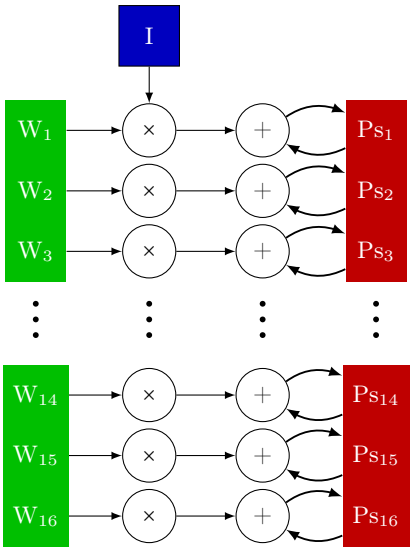


Figure 7.8: Block stationary data flow. For every input 16 MAC operations is done using 16 weights and partial sums, which is repeated until a block-row is fully computed.

buffer one at a time, in order to multiply and accumulate the partial sum related to that row of weights.

Having only need to store one input and 16 partial sums would be a lot more feasible. This of course depends on the size of the RF and number of bits representing a value, and hence the number of weights a PE is responsible for might need to be reduced in order to store these partial sums. The scenario of having 16 partial sums does however make addressing the weights simpler, since we now can work on a block at a time, and if we make either assumptions of the values bit precision (16- or 32-bits), they can be stored within the RF requiring 272 or 544 bits respectively of the 1,152 bit available.

If we examine the input weights alone, the matrix $\mathcal{U}$, it have in the case of the model `fix_RNF_NEW_BS64E60_L3.30` a total of 60,037 weights. Thus at the bare minimum it will be necessary to make that amount of multiplications, and in real-time these multiplications have to be made once per time step, that is once every 62.5 micro seconds. Hence to compute the input matrix-vector product alone one PE has to run at a clock speed of around 0.96 GHz (assuming we can perform one multiplication per clock-cycle), leaving no time to compute the remaining of the sub-algorithm or the remaining LPCNet. Hence stating the obvious, the sub-algorithm requires multiple PEs in order to complete the necessary computations in real-time.

Thus we will examine how scaling the numbers of PEs affects the dataflow. If we base the further scaling on the largest FPGA RTX currently uses the 'M2S025/M2GL025', we have a total of 34 math blocks and micro SRAM blocks available, along with 31 large SRAM blocks. Hence returning to the discussion of how to handle the sparse input matrix $\mathcal{U}$, we now have to determine how to best assign the additional computational power.

In Figure 7.9 the number of blocks with non-zero weights have been counted for any given rows. As can be seen does the number of blocks in a given row differ quite a lot, from 34 block in the smallest to 71 in the largest. Thus if we are to have multiple PEs process on the same set of rows, we will have to make a none general scheme to do so. A general scheme could for example be computing a specific number of blocks per PE.

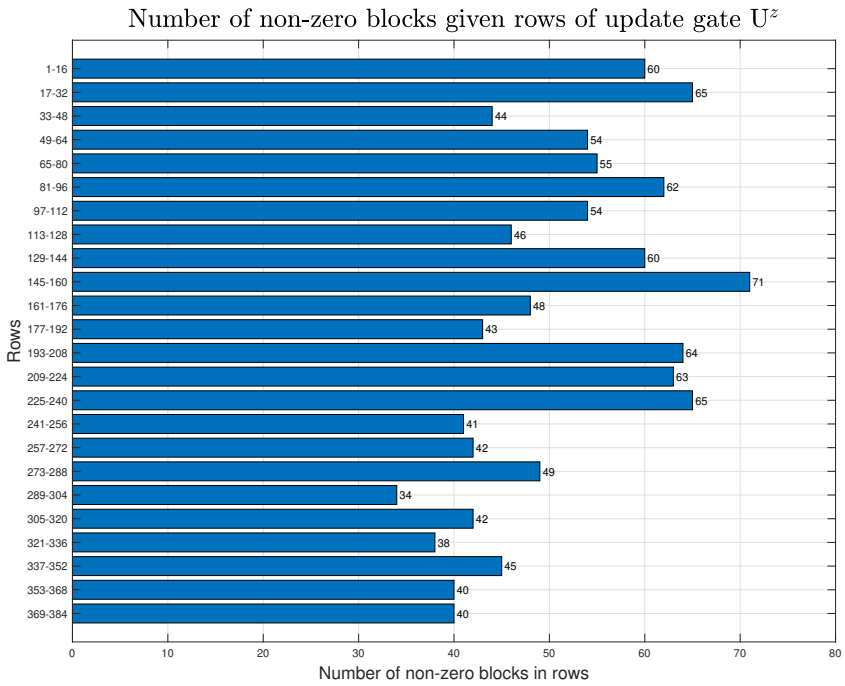


Figure 7.9: Number of non-zero blocks in a given interval of rows, in the input weight matrix related to the update gate $\mathbf{U^z}$

Like-wise if we want to assign one PE per rows, for example if PE_1 process all values given in the rows 1-16, and PE_2 process values in row 17 to 32. PE_2 have 5 blocks, that is 80 values, more to process than PE_1. Hence following this scheme of assigning one PE per block of rows like-wise requires, that we are able to communicate when a PE is done processing its current sets of rows, and this have to be done on an PE individual level. Since each PE will have different amount of data to be processes. Examining the weights column-wise yields a similar story as can be seen in Figure 7.10.

Hence we have to make an effort balance to the computationally load of these weights, otherwise we risk wasting time and energy by having the PE idle or storing unnecessary data leading to current leakage. Therefore since we cannot go with a general scheme efficiently, in the sense of giving a PE a set number of blocks to compute based on rows or columns of $\mathbf{U^z}$, we might as well investigate if there is some energy efficiency to be gained, while keeping the PEs busy, such that we can obtain a solution where the PEs runs at a lower clock frequency.

Processing of matrix-vector product

If we take basis in Table 7.1, as a guide line for how costly it is to move data between PE arrays and the global buffer. We can state that in order to consume less energy, we have to use data within the global buffer as little as possible, and keep data within the PE for as long as possible. Using the global buffer is of course much better that accessing DRAM, as it is an order of magnitude worse than accessing the global buffer.

Hence if we use multiple PEs on the same row of weights each working their own blocks, we have to accumulate the partial sums across PEs increasing the energy spent inter-PE. But we reduce the number of inputs we have to read from the global buffer, as we can store more inputs with each PE. In the opposite case if we assign multiple PE, by having one PE process an entire row of blocks, we decrease the number of partial sums which have to move between PEs. But increase the number of

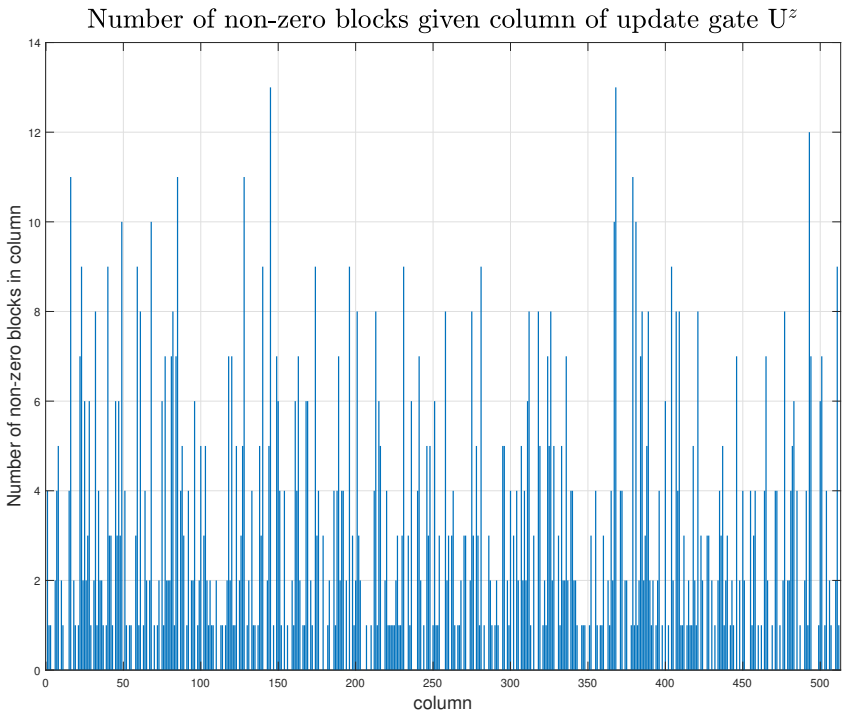


Figure 7.10: Number of non-zero blocks in a given column, in the input weight matrix related to the update gate U^z

inputs we have to read from either the global buffer or from other PEs.

If we were to use all PEs working on the same row, each on different blocks, then each PE will have to store 16 partial sums, leaving space for 20 inputs assuming each weight and partial sum requires 32bits. Thus we can actually store all 512 inputs from $\mathbf{x^{(t)}}$, by storing 15 to 16 inputs within each PE. As a result we have to read 512 inputs in to the PE from the global buffer, and then accumulate 34 partial sums for every row of blocks like the case shown in Figure 7.2 and 7.4. We will call this approach column partitioning. In the case of the update gates weight like in Figure 7.9 there is a total of 24 of such rows, meaning that a total of $16 \cdot 34 \cdot 24 = 13,056$ partial sums have to be moved in between the PEs, for the case of the weights related to the update gate. If we consider the entire matrix $\mathcal{U}$, this number is increased by 3 to a total of 39,168.

An alternative is to assign one PE per row of blocks, we again have to read 512 inputs from the global buffer and split these among the PEs. Thus we will not need to accumulate any partial sums across the PEs, instead we will have to read the correct inputs values from the various PEs, we will call this approach block-row partitioning. Assuming that each PE holds 15 input values, all PEs have to read 497 values from other PEs. Thus having a total of 72 rows of blocks in $\mathcal{U}$ and 34 PEs, each PE can reuse any given set of inputs $72/34 \approx 2.12$ times on average. Hence we have to read an approximate total of $479 \cdot 34 \cdot 2.12 \approx 35,800$ values to compute $\mathcal{U}$.

Thus we can save some energy by using this approach at first sight. But we will have to use additional logic to be able to read in between PEs, and we have to deal with the problem that we can only make so many read operations per RF, in the case of the 'M2S025/M2GL025' FPGA this number is two. Hence we only have two reading port available at any given time, and now have to avoid a situation where more than two PEs needs data that lies within the same PE. Therefore the first approach seems like an easier task to handle, given the current circumstances. A toy example of these approaches is

shown in Figure 7.11.

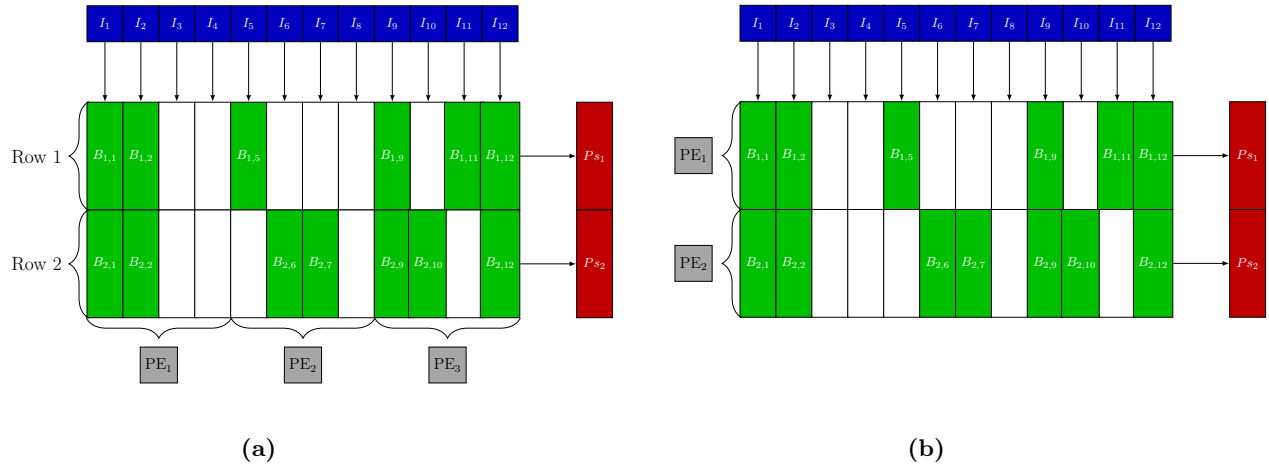


Figure 7.11: Toy example of the two presented approaches. **(a)** is an example on how PEs can be distributed column-wise, responsible for a set of inputs e.g. PE_1 will have input I_1 through I_4, the partial sum is accumulated across all PEs - column partitioning. **(b)** is an example where each PE is assigned to a specific row, each PE will responsible for a set of partial sums, yet inputs have to be shard among all PEs - block-row partitioning.

In any of the two presented approaches, there lies a hurdle in balancing the computational load equally between each PE. Since the weights to be processed is not evenly distributed, neither row or column wise as we can see in Figure 7.10 and 7.9, and this extends to all weight matrices of GRU_A. For example in Figure 7.11a does the PEs have to compute on four, three and six blocks respectively, like-wise does the PEs in Figure 7.11b have to compute six and seven blocks each.

Hence it seems exceedingly difficult to keep all PEs busy at all times by assigning them either way using the BS scheme. If we allow the implementation to have idling processing power, in order to circumvent this problem. A result is of course that we reduce the number of operations we can perform, and possibly waste energy by having idling PEs. In this case there is some energy to save by utilizing methods like clock-gating. When a given PE is assigned to compute a region of the weight matrix which has fewer weights, than what a different region assigned to a different PE.

Computing scheme of the sparse matrices

As we have discussed, it is highly difficult to keep all PEs busy at all times, while working on the basis of the block stationary scheme. Hence we allow a case where PEs might idle for a given time period. Doing so and continuing with the approach of assigning each PE to a set of inputs as depicted in Figure 7.11a, we will now describe our thoughts on how these sparse matrices can be computed.

In our thoughts assume that we have 34 PE available like in the case of the 'M2S025/M2GL025' FPGA, and we can store at least 16 partial sums and 16 inputs within the same RF of a PE.

If we use 32 PEs assign them 16 inputs each, we can hold all of the 512 inputs from $\mathbf{x^{(t)}}$, across these 16 PEs and thus we can reuse these inputs for entirety of the $\mathcal{U}x^{(t)}$ product, we will denote these as Weight Processing Elements (WPE). Hence each WPE will repetitively compute the same 16 inputs and their associated weights in this column partitioning approach. If we start with the case of computing a row of 'weight-blocks', each WPE will essentially process a 16 by 16 sub-matrix of $\mathcal{U}x^{(t)}$, therefore would a WPE in the worst case have to compute 256 products per row, if the given sub-matrix is completely dense. While the 32 WPEs compute on their individual sub-matrix, we can use the remaining two

PEs to accumulate the 16 different partial sums across these 32 WPEs simultaneously. Lets denote the two remaining PEs as Accumulating Processing Elements (APE).

If we work row-wise considering only one row, each WPE may compute on up to 16 weights using its 16 inputs i.e. all WPEs will process on the same sum. Thus after 16 MAC operations, all WPEs will have computed each of their fraction of the first sum. In Figure 7.12 we show a toy example of this. Where it is attempted to illustrated that PE_1 through PE_3 each compute on the first weights (light blue) within a given block of weights.

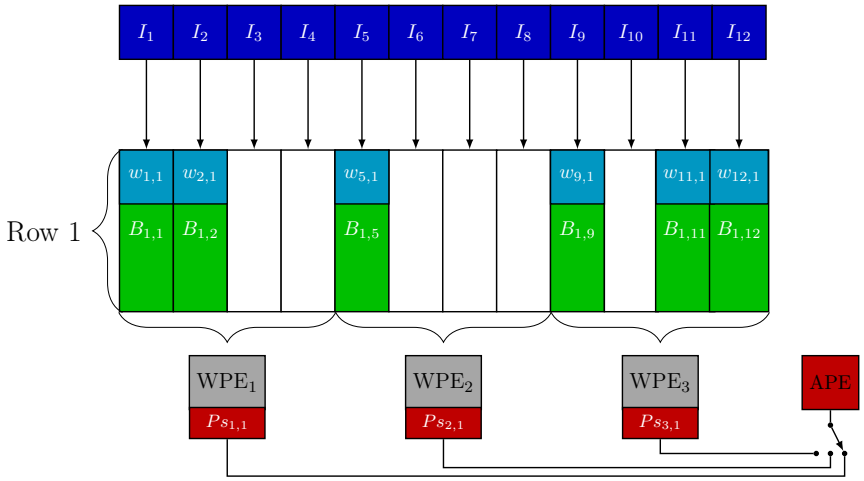


Figure 7.12: Column partitioning computational approach. The cyan blocks denotes weights within a given block of weights. PE_A denote an accumulating processing element

When the first partial sum is computed by PE_1 through PE_{32}, the remaining two APE can then accumulate their first partial sum, while the other 32 WPEs can work on the second partial sum of their individual sub-matrices. Each of the two APE will have accumulated one half each of the partial sum, after these 16 MAC operations.

At the end of the sub-matrix we are currently computing, each of the APE will hold 16 halves each of the 16 partial sums. Summing the remaining halves can be done over the span of eight MAC operations, by having each APE read eight of the 16 halves each from each other. when all partial sums are correctly accumulated each APE can then write their respective eight sums to the global buffer as shown in Figure 7.13.

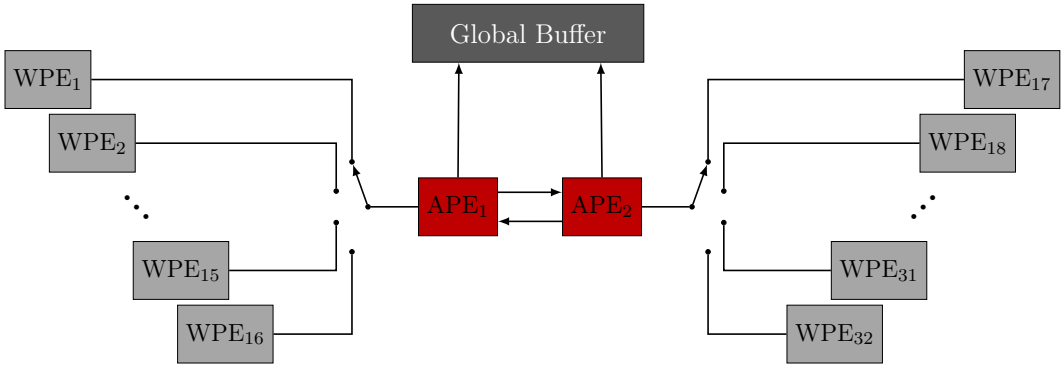


Figure 7.13: Accumulation across WPEs, using APEs

In principle we can thus have the 32 WPEs work simultaneously with the APEs, by having APEs

trailing one row behind the WPEs. Since each WPE in the worst case have 16 different blocks/inputs to process on and the APE each have 16 WPEs to accumulate partial sums from. They could in that regard work synchronously, with the exception at the end of the given sub-matrix, where the APEs have to combine their divide partial sums, so the WPEs do not get ahead for the APEs.

However by working in this predetermine scheme, we are either having the WPEs work on a product where the weights might be zero and therefore also their result or we could make it idle for that multiplication. In both scenarios we waste time by having the WPEs working on nothing of use, and possibly also energy. Wasting processing power is something we cannot afford if the LPCNet is to run in real-time, as we have discussed in chapter 4. Thus we have to make the WPEs skip computations that are unnecessary in order to compute on weights which are none-zero.

Considerations on control signals

In order to skip multiplications which will result in zeros based on the weights on and WPE individual level, we will need a plural of control signals. First we need a signal for every WPE, which define when a given WPE have reached the last weight in a given row, we will denote this signal End Of Row (EOR). This signal has two purposes first it tells when a given WPE can move on to the next row of weights and therefore also the next partial sum. Second it defines when one of the APEs can collect the partial sum of the given WPE.

A second signal is needed to define when a given WPE is at the end of its current sub-matrix. This signal serves the purpose of defining when all WPEs can move on to the next row of weight-blocks, we will denote this signal as End of Matrix (EOM). Additionally we will need a signal which denotes when all partial sums have been read of the given WPE, we will denote this All Partial sum Cleared (APC). Hence when all WPEs raises the EOM signal and the APC signal, we can move on to the next row of weight-blocks.

Meanwhile when a WPE have raised the EOM, we should clock-gate the MAC unit on that WPE in order to save power, and if the APC signal is also raised can we also clock-gate the RF on the WPE, since the APEs have read all the need partial sums the WPE holds. Lasty we may have a case where a WPE, have no weights in its current sub-matrix to compute on. Hence we need yet another signal to denote that the WPE should perform no computations for the entire row of weight-blocks, resulting in clock-gating the whole WPE doing this time, we will call this signal Empty Weight Matrix (EWM).

Signal name	Acronym	description
End Of Row	EOR	WPE's current partial sum is finished - APE can accumulate the sum
End Of Matrix	EOM	WPE have computed all partial sums - clock-gate MAC unit if no further computations are needed
All Partial sum Clear	APC	All a WPE's partial sums have been read - if all WPEs have raised APC and EOM, continue to next row - clock-gate RF if all partial sums have been read
Empty Weight Matrix	EWM	This WPE have no data to process during this row of weight-blocks - clock-gate entire WPE, until next row

Table 7.2: Overview of immediate control signals

Using Figure 7.14 as yet another toy example, we try to demonstrate this process. In this simple example, we only consider three WPEs, each "row" denotes weight-blocks hence each green block

corresponds to a single 16x1 vector, and each WPE are in this example responsible for four inputs each. As we can see, the loads for the WPEs are unbalanced, both in terms of number of weight-blocks, per row, per column and in total.

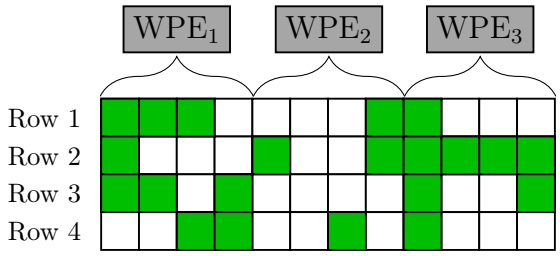


Figure 7.14: Toy example for processing a sparse weight matrix. Green blocks denotes a none-zero weight-block, white blocks denote a zero weight-block.

For the case of 'Row 1' both WPE_2 and WPE_3, will raise the EOR signal after one MAC operation, and they will thus move on to the second row of their respective weight-blocks. Therefore can the APE start to accumulate on these last two WPE. After three MAC operations does WPE_1 raise the EOR signal, and it moves on to its second partial sum.

After 16 MAC operations both WPE_2 and WPE_3 raises the EOM signal, and we will thus clock-gate each WPE's MAC units. After 48 MAC does WPE_1 also raise the EOM signal. After 48 MAC operations have the APE collected all partial sums from all WPE, and thus the both the APC and EOM signals have been raised for all WPE, denoting that we can move on to 'row 2'. In the 'row 3' we have case where WPE_2 has no weights to compute on, and thus it should raise the EWM signal, resulting in clock-gating the WPE completely during this row.

As we can see from this example does the computational efficiency greatly depend on the load balance of a given row of weight-blocks. For example in row 4 do we have a case where the WPE having the longest processing time is 32 MAC operations, yet the APE requires a total of 48 MAC operations, and therefore does the WPE wait between 16 and 32 MAC operations before they can continue. Thus it is hard to say anything about the computational speed in a analytical manner, since the weight-blocks are distributed randomly through out the sparse matrices, and this varies from model to model. Hence in order to get closer to an actual number on how fast the sub-algorithm can be computed, we can either try to simulate this approach or we can analyse the weights on a model basis.

For example in Figure 7.15 do we show how many weight-blocks each WPE will be responsible for, in the weight matrix of the update gate of GRU_A. Here a worst case scenario is in row 15, where WPE_6 is responsible for seven weight-blocks while WPE_7 has no data to work on. Thus the loads across the WPEs are very unbalanced, and consequently will the APE responsible for accumulating along the first 16 WPEs, have partial sums, that are computed by the WPEs far earlier than the APE can timely collect them. Therefore do we have a situation for similar characteristics as of 'Row 4' in our previous example just worse, since WPE_6 will have to perform 112 MAC operations, while the majority of the remaining WPEs have to make around 64 MAC operations or less.

So far we have only worked and described on the input weight matrix $\mathcal{U}$, but this method extends to the sparse recurrent weight matrix $\mathcal{W}$ also. This matrix and its sub-matrices have an input vector of size 384 instead of 512, and hence would each WPE be responsible for 12 columns instead of 16 if we allocate the same number of WPEs and APE.

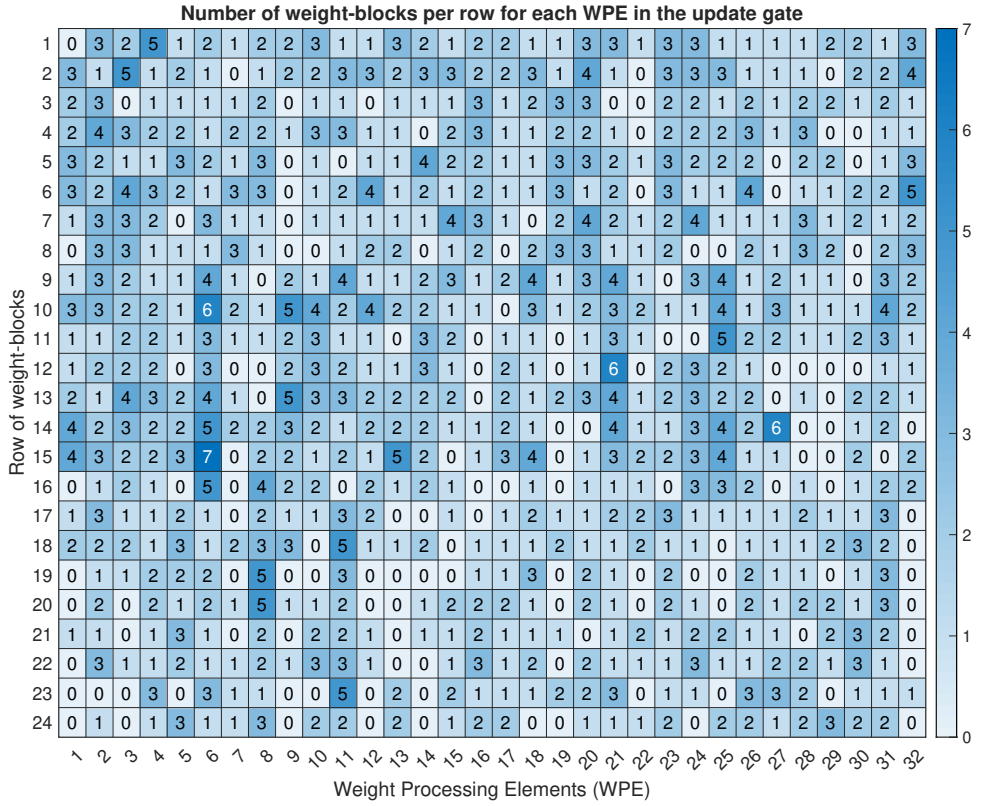


Figure 7.15: Heatmap showing the number of weight-blocks each WPE are to compute, for each row of the update gate weight matrix.

Alternatively can each WPE in the case of the recurrent weight matrices be responsible for 16 columns like before requiring a total of 24 PEs. We can thus assign the remaining 10 PEs as APEs. In either cases, we could use the same general approach as earlier described in this computation scheme.

Never the less is the point that it is exceedingly complicated, to have the PEs busy processing data at all times, while also working in an energy efficient manner posed by the BS data flow and handling the sparse weight matrices in GRU_A.

Simpler design scheme - simpler control

A much simple design would be to use the NLR together with the OS data schemes shown in Figure 7.16. Here one could have 16 PEs work on the same weight-block at a time, and thus would each PE only be responsible for one partial sum instead of 16, and the remaining space on the RF as additional capacity for the global buffer. The input to be multiplied with each PE's current weight is the same, and thus one could broad cast this input to all 16 PEs. An additional 16 PEs could simultaneously work on another weight-block in the same way. Then we could make use of the BCSC format presented in section 6.2.3 to correctly index the weight matrix, such that we multiply the correct weight with the correct input. This approach will make for a much easier design on all accounts, but it also comes at a cost, namely that one have to constantly write the inputs to the PEs from the global buffer. Possibly resulting in poorer energy efficiency, as we have to read the inputs a total of around $60,000/16 = 3,750$ times from the global buffer (one input per non zero weight), and additionally move these inputs 60,000 times to the individual PEs. Hence we again arrive at the same trade of between performance and energy consumption.

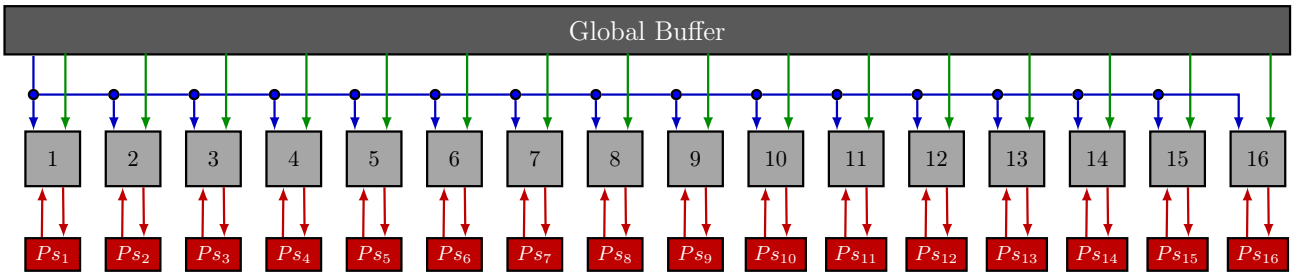


Figure 7.16: A combination between output stationary and no local reuse, where only the partial sum is reused, inputs are broadcasted to 16 processing elements (grey boxes), and each weight is sent individually.

Computing sparse matrices discussion

Thus we once again show the difficulty of dealing with the trade-off; between making the necessary computations energy efficiently by moving inputs and partial sums the least possible and increasing the computational performance by always keeping the PE busy, which indirectly requires lots of data movement.

In the above section for our described scheme for computing these sparse matrices, we attempt an energy efficient approach. By the introduction of the BS scheme, which we then extends to be able to handle the entire input weight matrix $\mathcal{U}$. Resulting in a design where the inputs to this matrix only have to be read form lower memory once, to compute the entire 1152×512 matrix and like-wise the recurrent weight matrix. Partial sums are also kept at an inter PE level until it is written to lower memory again. Hence we outline some of the general consideration, in order to energy efficiently compute these sparse matrices, both in terms of the data path and the necessary control signals.

This energy efficient approach comes at the cost computational performance, since we have some PE idling, while others completes their part of a partial sum. If we want to further improve the computationally performance, we can introduce more control signals, so that one can identify which WPEs are idling and have them help accumulating the remaining partial sums. This will in return make the design of such a scheme even more cumbersome than it already is, and one have to spend the extra energy to generate and act on these additional signals.

A simpler alternative is lastly presented, based around the NLR and OS data schemes. Here the task of keeping the PEs busy is much easier, but the design leads to poorer energy efficiency and hence we ultimately arrive at yet an other trade off. Hence it will be up to the designer to decide which is more valuable:

1. A simple design for high computational efficiency with poorer energy efficiency based on the NLR and OS schemes.
2. A more complex design using the BS data scheme, which should have better energy efficiency, but poorer computational efficiency
3. A very complex design requiring many control signals and like-wise complex control algorithm (state machines), which may yield a better result in terms of computational efficiency, compared to the second design.

7.3.2 Additional on weight matrices

So far have we only covered how to make the necessary matrix-vector products and necessary summation of sparse matrices within the GRU. While these matrix-vector products do indeed hold the vast majority of the computational complexity, the GRU holds some additional functionalities which we want to address. First of all, we have in our description of the computation of sparse metrics ignored the fact that the diagonal also holds some weights, second does the matrix-vector products also have some biases which we have not addressed yet. Additionally does the GRU also have the sigmoid and tanh activation function, element-wise multiplications and vector subtraction functionalities. Many of the considerations made here can be used directly for GRU_B aswell with some modification, since the amount of data to be handled is significantly smaller.

Diagonal of sparse matrices

As we have seen does the diagonal of the weight matrices also holds some weights, which we have ignored so far in order to focus on the weight-blocks. Our general thoughts of these diagonals is that in order to make a more stringent and easier processing pattern for the sparse weight matrices, and to utilize the proposed BCSC format. Should the diagonal be removed from the weight-matrices as described in the BCSC section. The now completely block-sparse matrices can then be computed based on the block structure of the matrices. The diagonal contribution can then simply be added to the partial sum after the remaining matrix is computed, and this can be done as an element-wise multiplication with the input and addition to the computed partial sums of the block-sparse matrices, as shown in Equation 6.11 and 6.12. This can be done in extension to what ever scheme already in place for the processing of the block-sparse matrices, e.g. with the column partitioning could one have the diagonal computed by the WPEs which holds the correct inputs to the associated weights.

Biases and partial sums for weight matrices

Another element which we have ignored so far is the biases associated with the matrix-vector products. As we have discussed earlier in chapter 4, the models currently have two sets of biases, one set for the recurrent weights and one for the input weights. These two sets can be added together offline, since they ultimately will be summed as a part of the different gates and candidate function through the GRU.

Now only having one set of biases we have a total of 3N×1 values, that is 1152 for GRU_A and 48 for GRU_B. Since the bias is a constant which will be added regardless of the respective inputs, the easiest way to include this in the sums which makes the different gates in the GRUs, is to begin the different partial sums with the biases value. That is instead of staring the partial sums with a value of zero, we will start with a value equal to the respective biases, this we save some additions and any potential discontinuities in the data flow. It only requires that the design for processing the matrix-vector products have these initial loads of the biases in mind.

This can be extended to include the partial sums computed from earlier partial sums. Such that when for example the matrix-vector products related to the inputs are computed, with the bias included as discussed above. We can use this as the new starting value in the partial sums related to the matrix-vector products related to the recurrent weights, i.e load $P_{sum,rec} = \mathcal{U}x^{(t)+B}$. Thus we can sum up the expression of the update and reset gate and the candidate activation 'for free', compared to having to add these at a later point.

7.3.3 Activation functions and subsequent layers

Next after the matrix-vector products and their summation are the activation functions. These may be the second largest contributors to the complexity of the GRUs depending on how they are computed. In Figure 7.17 one can see the current blocks in the GRU we are addressing.

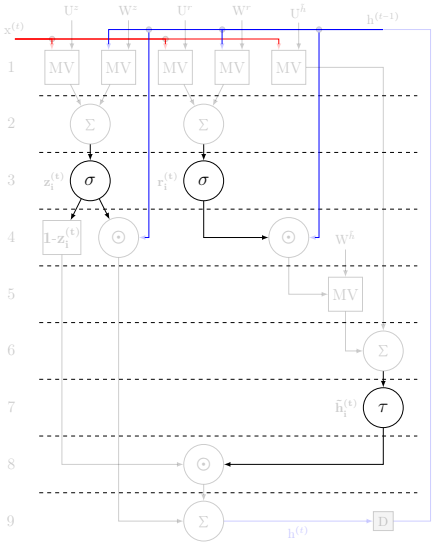


Figure 7.17: Currently we are addressing the activation functions, which have the sum of matrix-vector products as inputs and typically the element-wise multiplications as the following layer

Both the reset and update gate are subject to the sigmoid (σ) activation function, while the candidate activation is subject to the tanh (τ) function. These functions are given as:

$$\text{sigmoid}/\sigma(x) = \frac{1}{1+e^{-x}} \qquad \tanh/\tau(x) = \frac{e^{2x}-1}{e^{2x}+1} \tag{7.1}$$

The problem of these activation functions is to compute this exponential function, and here after compute the correct ratio which the activations make up. In terms of embedded devices we have two simple potential ways to compute these activation functions. Either a approximation by the use of piece-wise linear functions as done among others in [Wang et al., 2018] or by use of the CORDIC algorithm for hyperbolic functions [Walther, 1971], it should be noted that there exists several other ways to approximate the exponential function, but these two will be our focus.

First we have the CORDIC algorithm, in short by the use of the CORDIC algorithm one can approximate the values of $sinh(x)$ and $cosh(x)$, by making n iterations of add/sub and bit-shifts. These hyperbolic functions can then be used to approximate the exponential function as $exp(x) = sinh(x) + cosh(x)$ and tanh as $tanh = sinh(x)/cosh(x)$. The use of the CORDIC algorithm have the draw back compared to the the piece-wise linear functions, that we will eventually need to make division to compute either τ or σ. As we know does it take several cycles to compute a division between two variables, while in the other approach it only requires one multiplication and addition once the correct coefficients is identified, in order to compute the affine transformation of $y = a \cdot x + b$.

Thus if the accuracy of piece-wise linear solution allows for such an use, we think of this as the better option, for reference did [Wang et al., 2018]'s implementation of the piece-wise linear function

approximation result in an accuracy loss of less than 1%. Furthermore does the CORDIC come with its own limitations in regards to the accuracy it can produce, as it is decided by the number of iterations it uses. Hence for the remaining of this work we will assume the use of piece-wise linear functions in order to approximate the activation functions. Since reducing the activation function to simply reduced to identifying which linear piece to multiply with the input and then perform the multiplication, should yield a higher throughput compared to the CORDIC alternative.

We will not go into detail on how the correct coefficients should be identified, i.e. which of the piece-wise linear functions the input should be multiplied with. We assume this can be done relatively easy by the use of either a series of comparisons or some approach where the value it self can be translated to correct coefficients through logic and LUTs. Instead we will focus on the multiplication and addition to a given input, and have a look at I/O relations at this stage in the GRU.

First we have the sigmoid function. It comes after the reset $\mathbf{r^{(t)}}$ and update $\mathbf{z^{(t)}}$ gate have been computed. Meaning that each gate's output is a N-dimension (384) vector. If we assume that each PE can hold 24 different partial sums, a total of 16 PEs could then store the entire vector. By having a total of 32 PEs we could store both gate's output after they have been transformed by the sigmoid activation function, we will return to this shortly.

When we make the linear transformation of $y(x) = a \cdot x + b$, we can like in the case of the biases add the coefficient b by reading it as the beginning partial sum, as shown in Figure 7.18.

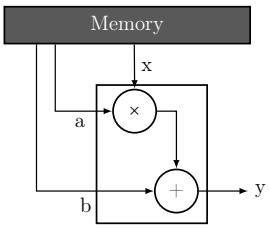


Figure 7.18: Making a linear transformation of the value x, by use of a MAC unit.

When the output of the reset and update gate has been computed i.e. the two 384-dimensional vectors. We will need to use these results directly after for new computations, as we can see in Figure 6.4. Hence subsequently do we have to compute the element-wise multiplication and the vector subtraction.

7.3.4 Element-wise multiplications - after sigmoid

Next we will address the element-wise multiplications, we devide this into two groups depending on the type of actication functions

For both the gate's outputs are followed by an element-wise multiplication with the recurrent input $\mathbf{h^{(t-1)}}$, that is:

$$\mathbf{z^{(t)}} \odot \mathbf{h^{(t-1)}} \qquad \text{and} \qquad \mathbf{r^{(t)}} \odot \mathbf{h^{(t-1)}} \tag{7.2}$$

Hence keeping the results of the gate's outputs within the RF of the PEs we have the multiplier, and thus only need to load the multiplicand that is the recurrent input. Furthermore since both the outputs have to be multiplied with the same input $\mathbf{h^{(t-1)}}$, we can match the PEs which requires the same input, so one only have to send this input once.

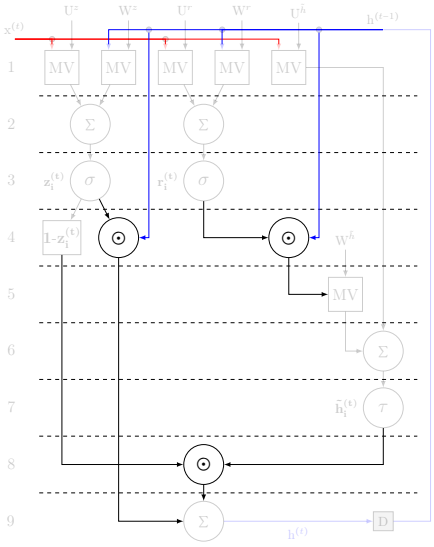


Figure 7.19: Currently we are addressing the element-wise multiplications, which comes after the two types of activation functions in the GRU.

For example if we assume we use the first 16 PEs to compute the update gate's output and the last 16 PEs to compute the reset gate's output. Then would PE_1 and PE_{17} require the same 24 multiplicands from $\mathbf{h^{(t-1)}}$, this is visualized in Figure 7.20. Thus we can save some time and energy by computing these element-wise multiplications in parallel.

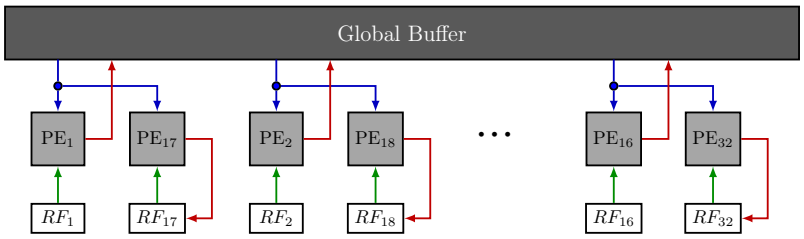


Figure 7.20: The result of the gate's output is already stored within the RF of their associated PEs (green). The inputs $\mathbf{h^{(t-1)}}$ (blue) can be sent to a pair of PEs at a time. The result (red) is then either written to lower memory or RF, depending on the layer to be computed next.

In the case of the result $\mathbf{z^{(t)}} \odot \mathbf{h^{(t-1)}}$, should be written to the global buffer, while the result of $\mathbf{r^{(t)}} \odot \mathbf{h^{(t-1)}}$ should be written to the RF within their respective PEs. Since this result is to be used in the following matrix-vector product of $\left(\mathbf{r^{(t)}} \odot \mathbf{h^{(t-1)}}\right) \cdot \mathbf{W^{\tilde{h}}}$ while we still need $\mathbf{z^{(t)}}$ in order to compute the vector subtraction of $\mathbf{1} - \mathbf{z^{(t)}}$ - as we can see in Figure 6.4.

7.3.5 Vector subtraction

The vector subtraction is rather simple as it is plainly to subtract all 384 values of $\mathbf{z^{(t)}}$ from the value one. Since the values of $\mathbf{z^{(t)}}$ is already present within the first 16 PEs, the task is merely to repeatedly make the subtraction for all values. The result of this should be written to the global buffer, in order to free up these PEs for the matrix-vector product of $\left(\mathbf{r^{(t)}} \odot \mathbf{h^{(t-1)}}\right) \cdot \mathbf{W^{\tilde{h}}}$.

7.3.6 Element-wise multiplications - after tanh

Like in the case of the two gates $\mathbf{z^{(t)}}$ and $\mathbf{r^{(t)}}$, we have a similar scenario when we have computed the candidate activation $\tilde{\mathbf{h}}^{\mathbf{(t)}}$ after performing the tanh transformation on the data, we have an output of

384 values which is used in the subsequent layer. Where an element-wise multiplication between the computed output of $\tilde{\mathbf{h}}^{(\mathbf{t})}$ and the vector subtraction is made, in order to compute $\tilde{\mathbf{h}}^{(\mathbf{t})} \odot \mathbf{1} - \mathbf{z}^{(\mathbf{t})}$. This again result in a vector of 384 values which yet again is used in the next layer when the GRUs output is computed as:

$$\mathbf{h}^{(\mathbf{t})} = \left(\mathbf{h}^{\tilde{(\mathbf{t})}} \odot \mathbf{1} - \mathbf{z}^{(\mathbf{t})}\right) + \left(\mathbf{z}^{(\mathbf{t})} \odot \mathbf{h}^{(\mathbf{t-1})}\right) \tag{7.3}$$

Thus we have the same scenario as before with the results of the gate's output, where results can be reused directly after they are computed, in order to save memory, time and energy. However this time we have more PEs available since we only handle one 384-dimension vector instead of two, which can be utilized in order to increase the computational performance.

7.3.7 Dataflow of GRU$_B$

As stated earlier does almost all considerations directly apply to GRU$_B$ as well. The difference is that the number N is decreased from 384 GRU units to 16, while the input size of $\mathbf{y}^{(\mathbf{t})}$ (the output of GRU$_A$ combined with $\mathbf{f}$) M remains the same size of 512 values. The dataflow is therefore alike but the size is smaller, and the PEs should be distributed accordingly.

One type of layer where the dataflow is different is in the matrix-vector products made, since GRU$_B$ have dense weight-matrices. Therefore is the task of distributing the weights, inputs and partial sums significantly easier. Hence we see both the column partitioning shown in Figure 7.11a without the extra control signal and BCSC format, and the combination of OS and NLR shown in Figure 7.16 as possible schemes to compute these matrix-vector products, with PEs distribute identically since the input size M remains the same.

7.4 Summary

In this chapter have we presented a topology, which allows us to explain out general thought on how the sub-algorithm can be computed. We have gone into great detail on how one can choose between a slower and more complex, but energy efficient implementation of the chosen sub-algorithm from chapter 6, and a more thought-put minded implementation, in regards to the hurdle it is to compute on the sparse matrices. Furthermore, we have broken the sub-algorithm down into its various layers, and highlighted the many possibilities to process the necessary data, in an energy efficient manner, by analysing which inputs, partial sums and results can be reused. If we store the data correctly and compute the different layers in the correct sequence.

The most difficult task of this sub-algorithm is by far the sparse weight matrices in GRU$_A$. Since this requires careful and complex planning of both data to be processed and keeping the processing elements busy as possible, while also matching the correct weights, inputs and partial sums. Which is made more burdensome, when we seek to keep data at the place of processing for as long time as possible, in order to gain energy efficiency, since the way the data is scattered counteracts this goal. We can conclude that it is possible to compute the sub-algorithm on an FPGA. Yet the sparse matrices requires a control system of varying complexity based on how energy efficient a data movment scheme one want to use.

While this work does not present the exact recipe for the implementation of the sub-algorithm. We do in a manner go beyond that, and give the reader the necessary knowledge to make their own decisions, on how this sub-algorithm implemented, while also proving out own thoughts.

Thus we conclude our thoughts on the FPGA as an implementation platform for the sub-algorithm. In the next chapter we will investigate and make considerations on how a CPU based platform can be used to compute the same sub-algorithm.

Chapter 8

CPU and SIMD based implementation of sub-algorithm

In this chapter we will present our thoughts on how the chosen sub-algorithm can be implemented on a CPU based architecture. This choice to include the CPU as an implementation option, is made because the CPU is a very popular platform for embedded devices and they are becoming increasingly powerful. Additionally have small and less throughput minded CPUs also seen a trend, where they have more options of parallel processing. Additionally do we think that the CPU as a platform, represent a software based implementation which many persons within the field can relate to, and our hope is that knowledge conveyed here can be adapted to similar platforms such as the DSP should readers see fit.

In order to describe this implementation in a more detailed manner, we take basis in a specific CPU which is in use by RTX. Therefore we will make assumptions based around this CPU. Hence in this chapter we will present the specific CPU, its features and considerations which can help us compute the sub-algorithm of chapter 6. This includes memory considerations, usage of the on board advanced SIMD extension, considerations on how to handle the process of converting the algorithm into code, considerations on how to prevent the CPU's cores from stalling and yet again how to handle the sparse matrices of GRU_A.

8.1 CPU description

The CPU chosen for our thoughts on an implementation is the ARM cortex A53 processor. ARM processors are based on Reduced Instruction Set Computing (RISC), and this processor is based on the ARMv8-A architecture and is a 64-bit processor. It supports the A32, T32 and A64 instruction sets with some possible extensions. The cortex A53 processor have between one to four cores within the processor, the specific processor we base this chapter on has four cores. Each core have its own L1 (level 1) cache, separated into a data-cache (d-cache) and instruction cache (i-cache) and a shared a unified instruction and data cache. Each of the L1 caches have a capacity of 32KB (both i- and d-cache) and the core's mutual L2 cache has a capacity of 512KB. One of the most interesting features of this processor, in regards to the implementation of the sub-algorithm, is ARM's advanced SIMD architecture extension which they call NEON, we will return to this later. A simple overview of the processor is given in Figure 8.1.

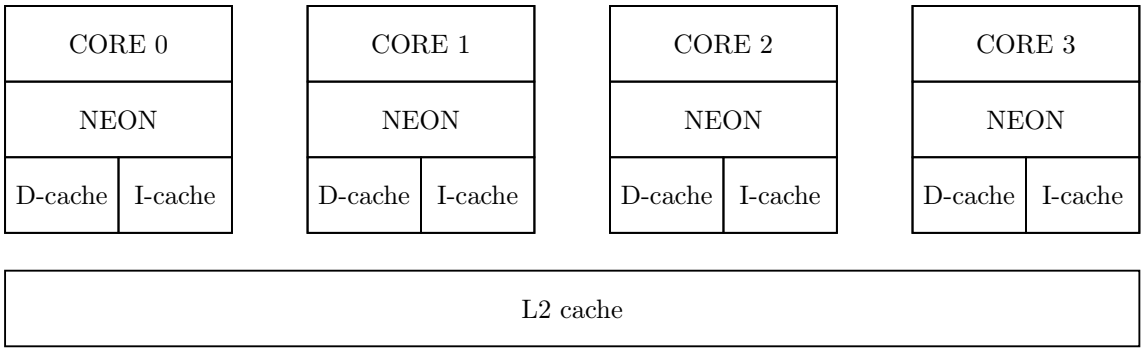


Figure 8.1: Simple block diagram of the A53 processor related to the data path.

Diving a little deeper into the specifications of the cortex-A53 processor. The target clock frequency is between 1.5 and 1.8 GHz for all four cores for the specific CPU, it has an 8-stages in-order 2-way super scalar pipeline, in theory making it possible to perform two instructions per clock. It has one multiply and divide unit along with two ALUs. Thus if we assume that each CPU core can make one MAC operations per clock, we would be able to perform $7.2 \cdot 10^9$ MAC operations per second ideally. However as we know is almost nothing ideal including this case. According to [Patterson and Hennessy, 2016] does the CPU have a Cycles Per Instruction (CPI) median of 1.3 using the SPECint 2006 benchmarks, which holds 12 different benchmarks, and the worst case is around 8.6 CPI - for a combinatorial optimization problem. Additionally for the median case is 60% of the stalls described to be due to pipeline hazards and the remaining 40% due to memory hierarchy stalls. Hence assuming the median of 1.3 CPI, it seems plausible to implement the LPCNet in real-time requiring around 3 GFLOPS, yet as we can see is it not a given, should we have something similar to the worst case. Additionally based on these assumptions is it clear that no matter how ideal our algorithm might run on the CPU, we do have to introduce multi core processing, if we are to have it running in real-time.

Hence a preliminary conclusion will be that the CPU have, with high probability, the necessary computational power to run the LPCNet (and our sub-algorithm) in real-time, provided that we use multiple of the CPU's processing cores and the compiler/cores avoids having too many stalls. While we cannot completely control the compiler if we are to program in a high level language, we can attempt to aid it by providing code which takes the factors which leads to stalls into account.

In [Patterson and Hennessy, 2016] these pipeline hazards are divided into three categories; structural, data and control hazards.

- Structural hazards happens because the hardware lack support for a given combination of the instructions one might want to run on the same clock cycle, the newer A64 instruction set from ARM should however be of some help.
- Data hazards occurs due to data dependencies, that is if we want to work on data which is not yet available, e.g. requesting a result which is not through the pipeline yet.
- Control hazards arises when we are required to take a decision and the further computations is dependent on this decision, hence the pipeline may have to stall while it waits for the decision to be made.

The compiler can do many things in order to avoid some of these scenarios, and as programmers we can further improve on these hazards if we are mindful of them - we will return to this subject of avoiding pipeline hazards later.

Continuing on the subject of stalls in [Patterson and Hennessy, 2016]. Memory stalls happens when the CPU have to wait for memory system either being ready to read from or write to. In [Patterson and Hennessy, 2013] it is eventually assumed that read and write stalls are mainly due to number of cache misses and the penalty of missing. Missing in this case refers to either not having the data within the upper layer of the memory hierarchy (the L1 cache in our case) for reading or writing, hence requesting or writing to memory not present in the L1 cache introduces a penalty in form of a memory-stall. The severity of this stall depends on how far the CPU have to go down in the memory hierarchy, therefore would data located in the L2 cache introduce less stall than data located in the main memory (DRAM). Furthermore do we consider a CPU with multiple cores, and one core will search for data within a second core's cache instead of going directly to L2 cache or the main memory, using the MOESI protocol.

Thus in order to better understand how we can avoid memory-stalls, we should attempt maximize the data reuse. Such that ideally we complete all processing on a piece of data, while the data is within the cache of the CPU. Hence only producing one read and write to and from main memory, in the given data's lifetime of the algorithm. To better understand how we can get closer to such an ideal scenario, we have to understand the cache of the CPU a little better.

8.1.1 Memory considerations

As already mentioned does the specific CPU we are considering, have a capacity of 32KB in both the i-cache and d-cache for the L1 cache, and an additional 512KB in the L2 for both data and instructions called a unified cache. While we do not wish to go over the entire description of the cache given in [ARM, 2015], we pick out the parts of which we find relevant for our implementation considerations.

The cortex-A53 have a cache line size of 64-byte for both data and instructions. That is the "smallest loadable unit of a cache, a block of contiguous words from main memory" [ARM, 2015]. Hence when data is loaded into the cache from the main memory, the cache is given a block of 64 bytes regardless if the Memory Management Unit (MMU) only need one byte or the whole block. Thus being aware of the size of the cache line's size, we may help the compiler by arranging data which is processed together in chucks of 64 bytes, or make data arrays dividable in such a way that the cache do not have to constantly read and write to lower levels of the memory hierarchy.

The CPU have what a is called a Non-Inclusive Non-Exclusive (NINE) cache policy. The instruction cache is inclusive, and hence instructions stored in the L1 cache also have to be in the L2 cache. While the data cache is exclusive, and thus the data will exclusively be found in only one of the two levels. This is interesting because this means, depending on how much of the L2 cache is used to store instructions, we might be able to store all data necessary for the sub-algorithm within the caches.

If we make an estimate of the immediate number of weights, inputs and outputs as done in Table 8.1 we get a better idea on how probable this is though. In this estimate we do not consider any additional information needed to correctly store and index in the sparse matrix of GRU_A, nor do we consider any additional variables. Note that the signal we have earlier have referred to as $y^{(t)}$ is the same as the output of GRU_A and $\mathbf{f}$ combined and $z^{(t)}$ is the same as the output of GRU_B. Furthermore we assume that all weights and signals are represented in float32 (4 bytes per variable).

If we assume that we use all four cores and L1 cache, we have a scenario where the 128KB i-cache would be stored both at the L1 and L2 levels since they have an inclusive cache policy. This would mean that the capacity for data is 128KB from the d-cache (L1) and 384KB in L2 cache yielding a

Input/layer	number of elements	number of bytes (float32)
$\mathbf{x^{(t)}}$	512	2048
GRU_A		
$\mathcal{U}_A$ input weights (sparse)	≈ 60,020	240,080
$\mathcal{W}_A$ recurrent weights (sparse)	≈ 45,275	181,100
$\mathcal{B}_A$ bias (sparse)	≈ 230	920
$h_A^{(t-1)}$ (GRU state) / $y^{(t)} - \mathbf{f}$	384	1,536
Sub-total	≈ 105,909	423,636
GRU_B		
$\mathcal{U}_B$ input weights	24,576	98,304
$\mathcal{W}_B$ recurrent weights	768	3,072
$\mathcal{B}_B$ bias (sparse)	96	376
$h_B^{(t-1)}$ (GRU state) / $z^{(t)}$	16	96
Sub-total	25,456	101,824
Total	≈ 131,877	527,508

Table 8.1: Estimation of memory required to store necessary inputs and coefficients of the sub-algorithm

total capacity of 524,288 bytes to be exact. Therefore we conclude that given this assumptions it is not possible to store the entire sub-algorithm within the cache of the CPU. It may however be plausible to store the necessary data of GRU_A and GRU_B separately, depending on the overhead necessary for storing partial sums and like-wise. The probability is further lessened if we consider that the CPU probably also have an operating system, and maybe even other tasks as well, this is however beyond the scope of this project.

In conclusion; in an effort to reduce memory stalls, we should try to reuse the data to be processed as much as possible, such that we avoid having to constantly be reading and writing data from lower levels of memory. We cannot store the entire sub-algorithm within the cache of the CPU and hence have to be mindful of how and when we access data from the main memory. We can reduce the need for multiple reads into cache, by having data which is processed together stored together. Such that when a cache line of 64 bytes is loaded into the cache, as much as possible of that data is used.

8.1.2 Advanced SIMD - NEON

We will do a quick outline on the SIMD NEON extension here based on [ARM, 2013] and then return to considerations using it in section 8.6.

NEON is part of ARM's advanced SIMD architecture. Each core of the cortex-A53 processor have a NEON unit, hence each core have the possibility to perform NEON tasks. Each NEON unit includes NEON registers, inter execute pipeline, floating-point execute pipeline and load/store and permute pipeline.

NEON registers

The unit have a total of 32 Q (Quadword) 128-bit registers or 64 D (Doubleword) 64-bit registers, which are seperate from the core arm registers. Each register is considered to be a vector of elements, of which each element must be the same data type. Each vector (register) is divided into n lanes (elements), and hence it is possible to perform n operations on n data elements in parallel. The Q

and D registers overlap, e.g. are the registers D0 and D1 the same as register Q0, where D0 holds the least 64 significant bits and D1 the 64 most significant bits. The NEON unit supports a plural of data types as shown in Table 8.2.

Each lane of elements are independent in the sense that there is no carry or overflow among elements, hence it is possible to make up to four single-precision operations in parallel.

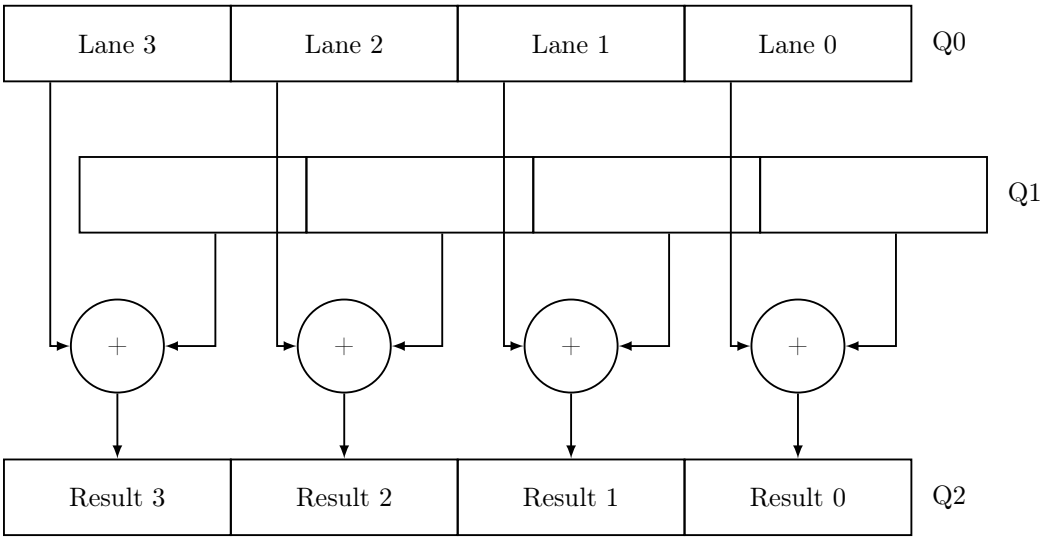


Figure 8.2: Example on multiple data operations using three 128-bit Q-registers - modified example of [ARM, 2013]

Programming NEON

A time saving feature of NEON is the 'NEON intrinsics'. Which allows the programmer to write C/C++ code normally, and then use these intrinsics when vectorized tasks present it self. These intrinsics serve as an option to NEON assembler, such that the programmer can write in a high level programming language and obtain some of the performance gains by using the NEON unit. The use of intrinsics will not yield the same performance as writing NEON assembler, but is much simpler to program and manage since, depending on the data type, one can make C variables which the compiler can map directly into the NEON registers, and hence the task of allocation of registers can be leaved to the compiler entirely.

Other alternatives to employ NEON is through neon-enabled libraries such as the Arm Compute Library, and Auto-vectorization features on the compiler may automatically optimize the code for NEON.

We will for the remaining of this chapter focus on the use of these intrinsics, as they present the simplest approach to obtain higher performance during computational extensive tasks throughout the sub-algorithm we are to cover.

Type / data size	8-bit	16-bit	32-bit	64-bit
Unsigned integer	U8	U16	U32	U64
Signed integer	S8	S16	S32	S64
Floating-point	N/A	F16	F32	F64

Table 8.2: Excerpt of supported data types by the NEON unit

8.2 From algorithm to code

We have now covered the topic on memory stalls and how we can deal with them, and the use of SIMD architecture to increase our computational performance. Having a better understanding of these options, the last subject to cover in order to improve our algorithm is that of pipeline hazards, but these are easier to address as we review the sub-algorithm w.r.t an CPU implementation. We will therefore begin to describe the process of converting the sub-algorithm towards programmable code.

8.2.1 Overview and discussion of sub-algorithm

Before reading this section the reader might want to take a short review of section 6.2, since we will build on this analysis and reference to information given here. We will present an overview similar to the one given in Figure 6.1, in a pseudo code manner this is shown in Algorithm 1 - we will return to it shortly, first we will discuss the input to the GRU layers.

Input to GRU layers and computational savings

As we recall does the algorithm in the broad picture include receiving the input to the first GRU layer, we have earlier denote this input $x^{(t)}$. This input is given by the three embedded inputs of $\mathbf{S^{(t-1)}}$, $\mathbf{p^{(t)}}$, $\mathbf{e^{(t)}}$ and the conditioning vector $\mathbf{f}$ given by the frame rate network. As we have discussed in section 6.2.1 is it possible to compute the matrix-vector product of $\mathcal{U}\mathbf{x^{(t)}}$, by having very large LUTs of the two embedding layers, and hence it is possible to reduce the computational cost significantly by increasing the memory usages correspondingly. As a result we have delimited this project from such an approach. However as the reader might recall, the conditioning vector $\mathbf{f}$ is reused for the entirety of a frame. Thus it is possible to compute the sub-matrix-vector product of $\mathbf{I}_A^f = \mathcal{U}_A^f \mathbf{x}^{f,(t)}$ once for every frame, and then reuse this product for the remaining frame. Therefore do we replace $159 \cdot 1/4M \cdot 3N$ (23,445,504) MAC operations with $159 \cdot 3N$ (183,168) additions. A significant reduction especially since additions generally is computational-wise less expensive.

Furthermore does the exact same situation unfold, when we are to compute GRU_B as its input is GRU_A's 384-dimension output (or state) and $\mathbf{f}$ again, this combined input was denoted as $y^{(t)}$. Thus we can do the same for GRU_B in order to compute $\mathbf{I}_B^f = \mathcal{U}_B^f \mathbf{y}^{f,(t)}$, and the computational expenditure is again reduced from $159 \cdot 1/4M \cdot 3N$ (976,896) MAC operations to $159 \cdot 3N$ (7,632) additions - remembering that GRU_B only have N = 16 GRU units.

If we translate this into a reduction in the real-time computational requirement, hence we go from:

$$159 \cdot 1/4M \cdot 3N \cdot 100 = 2,344,550,400 \tag{8.1}$$

MAC operations per second, since we have to process 100 10ms frames to compute per second, to a reduction of:

$$159 \cdot 3N \cdot 100 = 18,316,800 \tag{8.2}$$

additions per second. Here it is very important to note that in these MAC operations, we *have not* taken into account the sparsity of GRU_A. Ideally would this mean that the number of MAC operations is reduce by a factor equal to the sparsity, a factor 10. Thus would the required MAC operations to

compute $\mathbf{I}_A^f$ be reduce to around 235 $\cdot 10^6$ MAC operations per second, but the number of additions would most like remain the same. Since this would require that all weights in a row of the sub-matrix $\mathcal{U}_A^f$ is zero.

Sub-algorithm overview

Returning to the sub-algorithm its overview shown in Algorithm 1. We see that the sub-algorithm consists mainly of four functionalities, the two GRU layers and the preparation of the input to these. Additionally do we introduce the 'Reuse_flag', which indicates whether the GRU layers have to compute I_A^f and I_B^f or reuse it. The 'GRUA' and 'GRUB' functions takes a total of six inputs. The input $x^{(t)}/y^{(t)}$, the three types of weights, the reuse flag and the 'GRUA_state'/'GRUB_state', which is the earlier output of the same GRU layer ($\mathbf{h^{(t-1)}}$). The GRU functions yields the new GRU states ($\mathbf{h}^{(t)}$). The output of GRUB serves as the input to the Dual FC layer, we have earlier denoted this output as $\mathbf{z^{(t)}}$, note that this vector shares notation with the output of the update gate of a given GRU and these are not identical. It should however be clear from context which is which as we elaborate deeper on the GRU functions.

Algorithm 1 Overview of the sub-algorithm

```
/* Reuse_flag is used to determine if I_A^f and I_B^f is to be reused */
for i = 0:160 do
    ⋮
    if i == 0 then
        Reuse_flag = 0
    else
        Reuse_flag = 1
    /* Receive and prepare signal, Concatenate inputs in correct order */
    x^(t) = concatenate (S_emb^(t-1)[i], p_emb^(t)[i], e_emb^(t)[i], f)
    /* AB = bias, AIW = input weights, ARW = recurrent weights */
    GRUA_state = GRUA(x^(t), AB, AIW, ARW, GRUA_state,Reuse_flag)
    y^(t) = concatenate (GRUA_state, f)
    GRUB_state = GRUB(y^(t), BB, BIW, BRW, GRUB_state,Reuse_flag)
    z^(t) = GRUB_state
    ⋮
Done
```

The functions GRUA and GRUB performs the same operations to the given inputs, but differs on the size of these inputs and sparsity. Presented in this manner is our chosen sub-algorithm relatively simple. We transform the input $\mathbf{x^{(t)}}$ into $\mathbf{y^{(t)}}$ via GRU_A, and $\mathbf{y(t)}$ into $\mathbf{z^{(t)}}$ via GRU_B. This simplicity is however a bit more distant as we move onto the functionality of these GRU layers.

8.2.2 GRU layers

In Algorithm 2 we show the key functions necessary to produce the output of the GRU function. Here we suggest that the reader examine Figure 6.3 or 6.4, as these can be helpful to maintain an overview and path, as the input is transformed by the necessary functions.

In this algorithm we take basis on the first GRU layer, however since the two layers have the same functions, we will only show case this layer. Note that the size of a given result or variable is denoted in both the general term and the specific for the GRUA function. The size present a good option to

keep track of how results are transformed over the course of the GRU layer to yield the result in the correct size.

Note that the matrices and vectors are indexed multiple ways; either the whole matrix/vector, which is then denoted only by the variable name. If a specific part of the matrix/vector is index, it is indicated by ':', where it indicates 'start:stop' indexation, where the 'stop' value is not included. The 'all' denotes that all rows/columns are in use in a matrix.

For example in line **5**, we compute the first two thirds of the recurrent weight matrix, since the last third is related to the candidate activation $\tilde{\mathbf{h}}^{(\mathbf{t})}$ and cannot be computed yet. Hence we compute on the 768×384 sub-matrix-vector product of $\mathcal{W}$ and $h^{(t-1)}$, related to the update (z) and reset (r) gates.

Subsequently do we compute the matrix-vector product of $\mathcal{U}\mathbf{x}^{(\mathbf{t})}$. However based on the Reuse_flag we either reuse $\mathbf{I}_A^f$ and compute the remaining of the matrix-vector product (Reuse_flag == 1) or we compute both individually (Reuse_flag == 0). Either way are the results of these sub-matrix-vector products summed afterwards.

Following the summation we now have all variables necessary to compute both the update and reset gate, hence we sum these (line **17**) and both gates require the sigmoid activation function, hence we can apply the non-linear transformation to both gates simultaneous (line **19**).

Following the sigmoid transformation, we can separate the two gates output into two, for the sake of convenience. Hereafter we make the three element-wise operations, that is the element-wise multiplication of multiplying the GRU state with the two gates output, and the vector subtraction.

After these we can compute the last matrix-vector product, related to the recurrent weight matrix (line **29**). As before we sum this result with the corresponding weighted input and bias (line **31**), which is then transformed by the hyperbolic tangent function, yielding the candidate activation $\tilde{\mathbf{h}}^{(\mathbf{t})}$. This candidate is then element-wise multiplied by **zs**, which is lastly summed with **zh** to yield the new GRU state, that is returned at the end of this function.

Hereby is the GRU layers described. As stated before is the only difference between the GRUA and GRUB functions, the size of matrices and vectors, and the sparse weight matrices associated to GRU_A.

We now have the general overview of the necessary computations and functionality needed to compute the sub-algorithm. Thus we can begin to scrutinize which and how a given block or functionality can be addressed, in order to compute the sub-algorithm efficiently.

The effectiveness in this regard is to have the processor stall as little as possible. As we earlier have discussed are these stalls caused by either memory stalls or pipeline hazards. The memory stalls happens when a core searches for a piece of data within the highest level of memory and cannot find it (cache miss), resulting in the computations stalling while the core searches the lower levels of the memory hierarchy. The pipeline hazards covers three types namely, data dependency stalls, structural hazards, branching and/or branch mispredictions.

Thus we will investigate how we can avoid these as much as possible in the following sections. Here after will we attempt to address how we can handle the sparse matrices of GRU_A, which we have seen earlier is a bit of a hurdle. Lastly will look into how we can use the neon instruction set to help the core process multiple pieces of data at a time.

Algorithm 2 overview of the GRUs

/* Define the GRUA function inputs include GRU input,state and weight*/
Def **GRUA**($x^{(t)}$, **B**, $\mathcal{U}$, $\mathcal{W}$, $h^{(t-1)}$, Reuse_flag)
Begin
  /*Compute first 2N (of 3N) weighted recurrent matrix-vector product*/
  $\mathbf{W^{z,r}} = \mathcal{W}[\mathbf{0:768, all}] \cdot \mathbf{h^{(t-1)}}$ ▷ size = 2N×1 / 768×1
  /* Compute matrix-vector product related to **f** for later reuse
  Otherwise has $\mathbf{I^f_A}$ already been computed */
  if Reuse_flag==0 **then** $\mathbf{I^f_A} = \mathcal{U}[\mathbf{all, 384:512}] \cdot \mathbf{x^{(t)}}[\mathbf{384:512}]$ ▷ size = 3N×1 / 1152 × 1
  /* Compute remaining weighted input as matrix-vector product */
  $\mathbf{I'_A} = \mathcal{U}[\mathbf{all, 0:384}] \cdot \mathbf{x^{(t)}}[\mathbf{0:384}]$ ▷ size = 3N×1 / 1152 × 1
  /* sum into final weighted input */
  $\mathbf{I_A} = \mathbf{I'_A} + \mathbf{I^f_A}$ ▷ size = 3N×1 / 1152 × 1
  /* Compute input to update/reset gate, prior to non-linearity */
  $\mathbf{O'^{z,r}} = \mathbf{I_A}[\mathbf{0:768}] + \mathbf{W^{z,r}}[\mathbf{all}] + \mathbf{B}[\mathbf{0:768}]$ ▷ size = 2N×1 / 768 × 1
  /* Perform sigmoid non-linearity on all vector elements */
  $\mathbf{O} = \sigma(\mathbf{O'^{z,r}})$ ▷ size = 2N×1 / 768 × 1
  /* Separate into update/reset gate*/
  $\mathbf{r^{(t)}} = \mathbf{O}[384:768]$ ▷ size = N×1 / 384 × 1
  $\mathbf{z^{(t)}} = \mathbf{O}[0:384]$ ▷ size = N×1 / 384 × 1
  /* vector subtraction */
  $\mathbf{zs} = \mathbf{1} - \mathbf{z^{(t)}}$ ▷ size = N×1 / 384 × 1
  /* Element-wise multiplication */
  $\mathbf{zh} = \mathbf{z^{(t)}} \odot \mathbf{h^{(t-1)}}$ ▷ size = N×1 / 384 × 1
  $\mathbf{rh} = \mathbf{r^{(t)}} \odot \mathbf{h^{(t-1)}}$ ▷ size = N×1 / 384 × 1
  /*Compute last N (of 3N) weighted recurrent matrix-vector product*/
  $\mathbf{W^{\tilde{h}}} = \mathcal{W}[\mathbf{768:1152, all}] \cdot \mathbf{rh}$ ▷ size = N×1 / 384 × 1
  /* Compute input to candidate function, prior to non-linearity */
  $\mathbf{O'^{\tilde{h}}} = \mathbf{I_A}[\mathbf{768:1152}] + \mathbf{W^{\tilde{h}}}[\mathbf{all}] + \mathbf{B}[\mathbf{768:1152}]$ ▷ size = N×1 / 384 × 1
  /* Perform tanh non-linearity on all vector elements */
  $\tilde{\mathbf{h}}^{(\mathbf{t})} = \tau\left(\mathbf{O'^{\tilde{h}}}\right)$ ▷ size = N×1 / 384 × 1
  /* Element-wise multiplication of candidate function */
  $\tilde{\mathbf{h}}^{(\mathbf{t})}_{\text{scaled}} = \tilde{\mathbf{h}}^{(\mathbf{t})} \odot \mathbf{zs}$ ▷ size = N×1 / 384 × 1
  /* Compute and return new GRU state */
  $\mathbf{h^{(t)}} = \tilde{\mathbf{h}}^{(\mathbf{t})}_{\text{scaled}} + \mathbf{zh}$ ▷ size = N×1 / 384 × 1
Return $\mathbf{h^{(t)}}$

8.3 Cache considerations

We have briefly cover the topic of the cache on the Cortex-A53 CPU we have targeted in section 8.1.1. Here we mainly focused on the capacity of the two levels of cache available on the processor, and the so called cache line. An early conclusion is that, it is not possible to store the entire sub-algorithm within the two highest layers of memory, especially not if the processor have to handle other algorithms or tasks - such as an operating system. Instead we should focus on the smaller blocks of the sub-algorithm, and on how we can avoid reading data related to these blocks from lower levels of memory as much as possible.

The CPU only supports the Write-Back policy, which means that the cache will only overwrite data

in main memory, when the given data is evicted from the cache and only if the data within the cache have changed compared to that of the main memory. Hence we could have a situation similar to that of our FPGA based approach, where the updated partial sums only are written to the global buffer or lower level memory, once the PE writes the given piece of data to the global buffer.

Like with our FPGA based approach, a straightforward opportunity is to use a result directly after it is computed in the subsequent layer in the GRU. While the GRU holds many weights which requires more capacity than available in the L1 cache of a given block, the result of a given block is more often than not only a much smaller vector, which can indeed be stored in the L1 cache, ready for reuse. While we cannot control the compiler explicitly if we write in a high level programming language such as C, we can try to influence its decisions. For example by having one layer which uses a second earlier layers result directly after this second layer.

Here our precedence graph in Figure 6.4 can be helpful to identify which layers poses such opportunities. We have tried to make the pseudo code in Algorithm 2 build on this. For example following the summation of the two matrix-vector products, we have the two sigmoid activation functions yielding the update and reset gate's outputs. If we compute the output ($\mathbf{z^{(t)}}$) of the update gate last, we can use this result directly in the next to blocks to compute **zs** and **zh**. Additionally since **zh** and **rh** have the same input multiplied on them, we can reuse this input and computing **rh** last means, that we can reuse this result directly afterwards - as shown in line **21** through **29**. This is a very basic optimization and it plain verges on standard imperative programming, hence we can ask our self if there is more that can be done, in order to make the algorithm more cache friendly.

As earlier mentioned does the L1 have a cache line size of 64 bytes or 16 words and so does the L2 cache. Hence if we assume that weights, partial sums and values in general can be represented in float32, each cache line will indeed consists of 16 of such values. In order to read as few cache lines as possible, we have to try to make values which are used sequentially lie together, such that the whole cache line holds values which are to be used.

If we want to go into extremes with this approach we can separate the weights, inputs and results into arrays with a size of 16 values each. In order to influence the compiler to store these values together in high level language or directly make sure of it in low level programming. For example could the equation $\mathbf{zh} = \mathbf{r^{(t)}} \odot \mathbf{h^{(t)}}$, which consists of two N-dimensional vectors (384 values in GRU_A), be speared into 24 sets of vector pieces, which can individually and successive be multiplied together, resulting in 24 different sets of products.

If we consider these changes applied to the all values used in the GRU layers, the is change relative simple in terms of vector-vector operations and the non-linear transformation of the matrix-vector products. It is some what more challenging in the case of the dense matrix-vector products made in GRU_B, but yet again are the sparse matrices and their matrix-vector products the genuine hurdle in this approach. Since we have to address these, such that we only process non zero weight-input products, if the computational complexity is to remain feasible. Hence we will yet again turn our focus towards this issue.

8.4 Handling sparse matrices

In this project we have presented two methods in order to address the sparse matrices used in GRU_A discussed in section 6.2.3. First the already existing CSC which works for any given sparse matrix,

but only yields a memory reduction when it fulfills the expression given in Equation 6.10. Second we presented our own approach the BCSC format, which further reduces the memory required to store a sparse matrix. It is however required that the given matrix consists of blocks in a given block size. We will use this issue to further examine the BCSC approach as an option for block-sparse matrices.

8.4.1 Using the BCSC format

As we recall is the BCSC composed of three vectors; the data vector, holding the non zero elements. The Count vector which determines the successive distance between blocks column-wise. Lastly is the address vector which decides the number of blocks in a given column, indirectly signifying when we are addressing a new column of blocks. A side note is that we will have the diagonal removed from the weight matrices, and later have its contribution added as a dense vector to the correct partial sums.

If we are to elaborate further on this, and include our work with the matrix-vector products earlier in the FPGA approach. We can in a sense say that the count vector determines which partial sum we are currently working on, by adding one to the count value and multiplying by 16.

For example if we just had a block say the very first in the column, then we have worked on the partial sums associated to the rows 1 through 16. If the count vector holds the value '3' we know that the next rows, we will be processing are the former rows plus $(1 + 3) \cdot 16 = 64$, hence we will address row 65 to 80 next, we have shown this example in Figure 8.3.

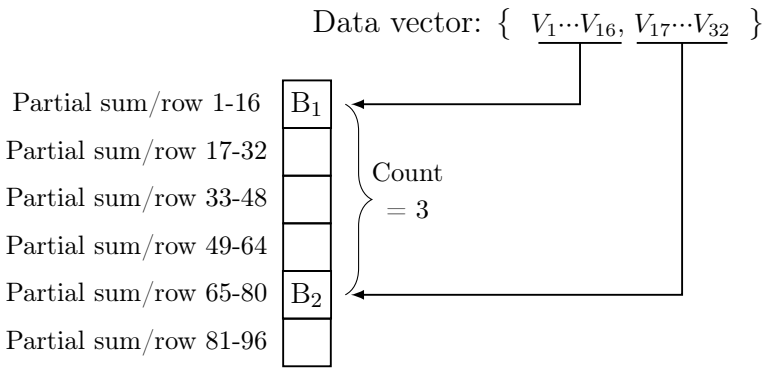


Figure 8.3: Example on how the count vector implicitly determine the partial sum currently being processed. 'V' denotes the values hold within the data vector.

It can like-wise be effortlessly seen, that the address vector in the BCSC format implicitly determines which column, and thereby which input, we are currently processing. Thus we can already now see that using the BCSC format results in some computational overhead, introduced when we have to compute which rows and inputs we are currently working on.

Initially in regards to the count vector, we would require to multiply the current value in the count vector by 16, and add this to the previous row we were working on. This overhead can be reduced slightly, by preemptively multiply the count vector's values by 16. Which means that the count vector no longer describes the block-wise distance between non zero blocks, but rather the number of rows we should add to the current row in order to process the next correct partial sum. Thus reducing the overhead to one add per block, we will denote the indexation to the count vector as 'blockCounter'.

The address vector will indirectly control a counting variable, which will have a value between 1 and 512 (M, for $\mathcal{U}\mathbf{x}^{(\mathbf{t})}$) (or 384 (N) for $\mathbf{h}^{(\mathbf{t-1})}$), which denotes what input we are currently multiplying to the

weights. Hence we have to compare, a second variable which counts how many data values have been processed, with the current value of the address vector, we denotes this variable as addressCompare. Once they are equal we know we have to begin on a new column, and hence we reset this second variable and row, and begin to iterate it and compare once again. An example of how a matrix-vector product is computed using the BCSC is shown in Algorithm 3.

Algorithm 3 Making matrix-vector products using the BCSC format

```
/* PS = partial sum
   cV = countVector
   bC = blockCounter
   aC = addressCompare
   aV = addressVector
   dV = dataVector
   pSI = paratialSumIndex */
while i ≤ K do                                   ▷ for 𝒰 K = 512, for 𝒲 K = 384
    /* Last block in the column, change input*/
    if aC == aV[i] then i = i + 1
    /**/
    pSI = cV[bC]
    /* Perform 16 MAC operations, multiplying 1 input with all 16 weights */
    PS[pSI:pSI+16] = PS[pSI:pSI+16] + Input[i] · dV[aC:aC+16]
    /* iterate the blockCounter */
    bC = bC + 1
    /* iterate the addressCompare */
    aC = aC + 16
End
PS[all] = PS[all] + removed_diagonals · Input
```

Thus we have a total overhead of three adds and one compare for every block within a sparse matrix, excluding the necessary MAC operation. Thus the total overhead can be a bit difficult to accurately compute, since the number of non-zero elements and location in the sparse matrices varies slightly from model to model, and when we are at the last block of a column, we do not add to the blockCounter and similarly at the very last column at the last weight block we neither add nor compare.

We should however encounter this overhead more than once per block, and hence make a conservative estimate equal to the number of blocks in the weight matrices, in the sense that we likely over estimate the complexity. Thus we have the overhead estimated to:

$$Total\ weights = 60,020 + 45,275 + 230 = 105,525 \tag{8.3}$$

$$Total\ blocks = \frac{105,525}{16} \approx 6,595 \tag{8.4}$$

$$Total\ overhead \approx (3\ adds + 1\ compare) \cdot 6,595 \tag{8.5}$$

$$\approx 19,785\ adds + 6,595\ compares \tag{8.6}$$

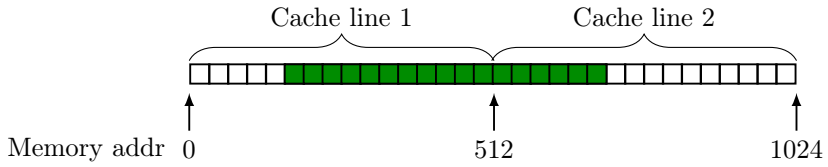
Which still is a significant reduction compared to doing one MAC operation for every zero valued weight (approximately 929 thousand), which needs to be computed every time we need to compute

a new GRU state. If we assume that adds and compares are one floating point operation each, the overhead will lead to a real-time computational cost of around 422 MFLOPS, a significant reduction to the alternative of making around $14{,}9{\cdot}10^9$ MAC operations per second. Note that these estimates do *not* assume reuse of I_A^f and I_B^f, which we approximate will yield a reducing of the computational cost in an order of around 10%, in both cases. Hence around 380 MFLOPS and $13{,}4 \cdot 10^9$ MAC operations per second respectively.

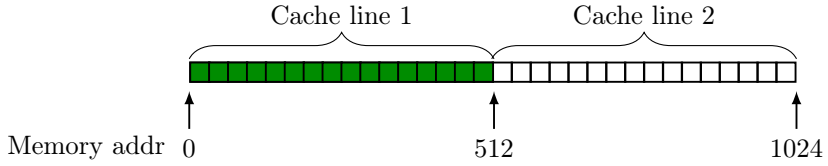
8.4.2 Cache/memory considerations

We have now shown how we can utilize the BCSC format, in order to gain the reductions in terms of both memory and computational requirements. We have shown that the use of the BCSC format comes with some overhead, but this overhead is well worth it compared to the alternative. However as we have discussed in section 8.3 we do see some benefits of having arrays which fit within a cache line (64 bytes). While it is obvious very few of the vectors within the sub-algorithm are a size of a cache line, we may be able to help the compiler out a bit.

First of all we should tell the compiler to store this dataVector of the BCSC format, in main memory such that it aligns with a cache line. Since a block of 16 weights in single precision requires 64 bytes, and we multiply the input on all values with this block. Hence aligning these weights in main memory such that they fit within a cache line, we avoid having to read two cache lines instead of one, reducing the risk of memory stalls. An example is shown in Figure 8.4.



(a) The block of weights (green) is mis-aligned, thus requiring reading both cache lines



(b) The block of weights (green) is aligned, thus requiring reading only one cache lines

Figure 8.4: Comparison of a weight blocks which is unaligned and aligned.

This consideration also apply to the vector holding the partial sums, since we address them in a block of 16 also. This consideration assumes that data stored within main memory, only can be accessed in fixed memory addresses. If the hardware can start a cache line read from any address within the main memory this consideration is needless.

Second if possible we could notify the compiler of which inputs are to be reused multiple times, for example the inputs as opposed to the weights. Such that these inputs are kept within the cache if possible, and the MMU does not accidentally evict these, just to have them loaded during the next multiplication of weight blocks.

Third during this project we have considered the entire blocks within the GRU, for example when we examine its precedence in Figure 6.4, as a result we often like to compute that whole block. However as we have seen does this approach often lead to handling vectors of size N (384 or 16). Vectors which

cannot also fit within the NEON register as presented in Section 8.1.2. Since the GRU units within the GRU layer are not dependent on each other only their old states ($\mathbf{h^{(t-1)}}$), hence one could make all the computations for one or more GRU units at a time i.e. setting N to any desired number. Thus doing periods where we use results from one computation subsequently in the next computation, we could keep the this result within the cache until the state of one or more GRU units are calculated.

Only working on a set number of GRU can be done for any given block with the GRU layer. Yet doing so when the matrix-vector products are computed, breaks the initial idea of using the BCSC format. Additionally do all GRU units have $\mathbf{x^{(t)}}, \mathbf{y^{(t)}}, \mathbf{h_A^{(t-1)}}$ or $\mathbf{h_A^{(t-1)}}$ as input having a size of 512, 512, 384 and 16 respectively. Therefore during these block we would have to read these inputs multiple times, if we work on a subsets of the GRU units within the GRU layer. Thus should one be mindful of computing subsets for these blocks, for all other blocks of the GRU we no such issues.

It should be noted that computing only a limited number of GRU units, which results will be needed for subsequent computations may have a drawback. Since such computations may lead to data dependencies and therefore pipeline stalls, we will return to this shortly.

8.5 Pipeline considerations

Having covered our thoughts on how we can address issues related to memory stalls, created primarily by cache miss, either by aiding the compiler write assembler such that we reduce the number of cache lines to be read. We now turn our focus shortly towards the aforementioned pipeline hazards, the stall they bring with them, and what we can do to reduce these stalls.

8.5.1 Data hazards

Data hazards happens because of data dependencies, for example making two different computations, where the second computation is dependent on the result of the first. This creates a pipeline stall, since the result of the first computation needs to be through the entire pipeline before it can be used. Alternatively is it sometimes possible for a processor to forward a result, before the result have been through the entire pipeline. It does however depend on the timing between when the result is computed, and when this result is then needed for subsequent calculations.

For our sub-algorithm this is generally not a big issue, assuming that the CPU can handle the MAC in a good way. Since it otherwise consists of a multiplication, where the result of this is added directly afterwards to a partial sum. The MAC is however an essential operation for many algorithms, and thus we do assume for the remaining of this section, that the use of MAC operations do not provoke data hazards.

Other scenarios where we in a sense encounter dependencies, are when for example in line **24** through **26** in Algorithm 2. Here we compute **rh** by multiplying N vector elements with N other elements. The result **rh** is then used consecutively, as we use it in the matrix-vector product of $\mathbf{W^{\tilde{h}}} = \mathcal{W}[\mathbf{768:1152, all}] \cdot \mathbf{rh}$. However should it be noted that in the case of GRU_B, which have the smallest vectors of size N = 16, we are still utilizing all 16 elements.

Thus we have a scenario where even though the result of the vector **rh**, is used consecutively, the first element of **rh** have been computed some time before the 16'th. Therefore if we keep the right order of these individual elements, we should have little to no data dependencies in this regard.

Returning to our earlier discussion for choosing the number of GRU units (N) to be computed at a time. We can now see that N should be chosen sufficiently large, such that we do not create pipeline stalls coursed by data dependencies. This number depends on the length of the given pipeline, and how either the core or NEON handles reading and writing inputs and results. Hence it may be hard to give an exact number theoretically, but in practice we may be able to measure if a specific number of N performs best.

With it we have shown that the sub-algorithm is not inherently problematized by data dependencies, causing the pipeline to stall, but we can course them our-self if we are not mindful.

8.5.2 Control hazards

As earlier described is control hazards caused by the need to make decisions whether the CPU should compute on one thing or the other, this is often referred to as branch hazards. Having a branch, is when an algorithm or a program have to make a decision to follow different paths depending on one or more inputs. The CPU does in the worst case have to stall, its computations until this decision have been made. Typically occurrences of such a branch is by 'if' statements, where based on some expression should the algorithm do something different.

In our sub-algorithm do we only have one such branch/if statement, which is related to the reuse_flag. This can be seen in Algorithm 2 in line **8**, where we ask whether the flag is set, and if not we should compute I_A^f and I_B^f once more. This happens every time the LPCNet receives a new frame, hence in one out of 160 cases.

To combat branch stalls most CPUs have branch prediction, i.e. the CPU tries to find patterns of when a given branch is taken, and based on this predict it continue to compute through the predicted branch. If the prediction is correct, then the CPU have successfully avoided a stall due to the branch hazard. If the prediction is incorrect, the current computation have to be discarded and new computations have to be made based on the branch's result. Making brach prediction based on patterns made at run-time is called dynamic Branch prediction.

A very simple strategy is to assume that a given branch or if statement is never taken or made, and continue through the program, as if nothing have happened, only computing the given branch should it eventually happen to be true - this is called static branch prediction. Thus in our case would the sub-algorithm only lead to a branch stall, in one of 160 cases, thus we see control or branch hazards as an negligible reason for stalls, and no we will examine this topic no further.

8.5.3 Structural hazards

As described earlier does structural hazards occur when we try to perform instructions which makes a conflict, or combinations of instructions which is not possible. Avoiding structural hazards thus requires good knowledge of the available instructions for the CPU and here after which are not compatible or can eventually lead to conflicts.

Since we have mostly considered an implementation by higher levels of programming, such as the use of C as programming language. It would in this case not be the job of the programmer to be mindful of these possible hazards but rather the compiler.

It is hard to say anything on a general basis on how to avoid structural hazards, but rather a very specific task and a time consuming one. As it would require a deep analysis of the *many* instructions

and combinations of instructions provided by the A64, A32, T32 and NEON instruction set. As such we delimit this work from a further investigation of this subject.

8.6 Neon considerations

As briefly touched earlier have we presented the use of the NEON extension to the cortex-A53 CPU. Since this represent ARM's option of using a SIMD structure to perform parallel computing, and since, as we have shown time and time again, both the sub-algorithm and LPCNet in general does posses possibilities to utilize parallel computing in abundances. It would be foolish not to exploit any given chance of utilizing parallel computing schemes.

As earlier described do the NEON extension to the core's pipeline include both registers and the necessary hardware to perform operations i parallel. These operations includes [ARM, 2013];

- Additions
- Pairwise additions
- Multiply
- Multiply accumulate
- Multiply subtract
- Shifts (left, right) and insert
- Logical operations (AND,OR,EOR,AND NOT, OR NOT)
- Operations to select minimum and maximum values
- Operations to count leading zeros, count signed bits, and count set bits

NEON does however not include division operations or square root operations. Instead [ARM, 2013] state the division should be made based on Newton-Raphson iteration.

Without the possibility to make divisions directly, it would be interesting to investigate which approach would yield the high throughput when computing the activation functions, which relies both on divisions and computing exponentials. That is whether making Newton-Raphson iteration and computing exponentials or the proposed use of piece-wise linear functions [Wang et al., 2018], which requires that we identify which of the pieces a given input falls within and a multiplication afterwards. The remaining operations required by the sub-algorithm is covered by the available operations.

If we assume that all values (inputs, coefficients and results) can be represented by float32, we saw earlier that with NEON we can make up to four operations in parallel (using Q-registers), requiring three of the 32 registers to do so. Hence we can have a total of 128 values into play within the NEON unit. Hence we have to be mindful of how we use these registers, as we will have to read and write from them a considerable amount, in order to compute many parts of our sub-algorithm. Here we would like to examine some key aspects of the sub-algorithm, starting with the sparse matrix-vector products.

8.6.1 Sparse matrices using NEON

We would like to highlight the sparse matrix-vector products of GRU_A once again, since these two holds the most weights, are responsible for the most computations and are generally the most complex task.

If we for the sake of simplicity take basis in the BCSC format presented in Algorithm 3. The part of the algorithm which is data heavy is when we perform the MAC operations. Other computations

performed are iterations, comparisons and possibly address calculations, these are however based on single variables and as such would not gain anything from computing in a SIMD structure.

Hence only the MAC operations will have something to gain from the SIMD structure. Since the operation to be made for the sparse matrix-vector product, is only a simple MAC. One may be lucky to have this functionality covered by an existing library function or maybe the compiler can recognise the operation and optimize the code automatically. Should this however not be the case, then we can use the NEON intrinsics to obtain this needed parallel performance.

Continuing of the use of the BCSC format and wanting to use the NEON extension. Then it would be useful to have a scheme to keep track of the NEON registers of use.

Since we assume the NEON registers can hold up to four values within each Q-register. In principle would three registers holding input, weights and partial sums be sufficient, but based on having a better overview it may be helpful to consider one block of weights at a time. From a performance point of view it is hard to tell whether using three or nine registers, yields a higher throughput.

Hence if we want to make the MAC operations as seen in Algorithm 3, it will require a total of nine registers (eight Q-registers and one D-register for the input). Since we need four registers to store a given block of weights and partial sums, and a single to hold the input to be multiplied with the block, over four iterations, as illustrated in Figure 8.5.

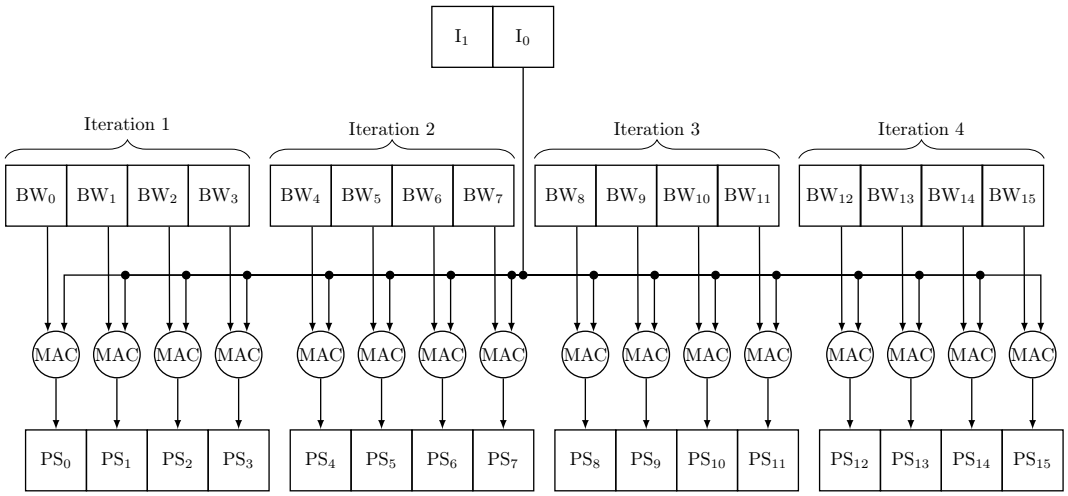


Figure 8.5: Compute a MAC operation performed on one input and one weight block (16 weights). BW denotes block weight and PS denote partial sums

After computing the MAC operations through four iterations, using the same input but changing which registers to be multiplied with, and which to be accumulated to after each iteration. Since neither the input vector nor the number of partial sums are sufficiently small to reside in these Q-registers, for either of the two sparse matrix-vector products made in GRU_A. Therefore would it not make much sense to use more registers than these nine during the computation of the products. Unless the NEON co-processor can load and store data to and from the cache faster than it is computed, in which case it would make sense to have a total of 18 registers, such that each register has a buffer register in order to keep the computational pipeline busy, thus avoiding memory stalls.

In Algorithm 4 we shown how the MAC operations in Algorithm 3 on line **14**, could be replaced by an intrinsics based function call.

Algorithm 4 Intrinsics example on MAC operation, based on BCSC format

```
1:  void _bcsc_mac_neon(float32_t *I, float32_t *B, float32_t *C){
2:    /* Register holding Input, use only D-register*/
3:    float32x2_t I0;
4:    /* Register holding block of Weights*/
5:    float32x4_t B0;
6:    float32x4_t B1;
7:    float32x4_t B2;
8:    float32x4_t B3;
9:    /* Register holding result (P-sums)*/
10:   float32x4_t C0;
11:   float32x4_t C1;
12:   float32x4_t C2;
13:   float32x4_t C3;
14:   /* Load Input from memory into registers*/
15:   I0 = vld1_f32(A);                          ▷ A is pointing at value within memory
16:   /* Load weights from memory into registers*/
17:   B0 = vld1q_f32(B);
18:   B1 = vld1q_f32(B+4);
19:   B2 = vld1q_f32(B+8);
20:   B3 = vld1q_f32(B+12);
21:   /* partial sums from memory into registers*/
22:   C0 = vld1q_f32(C);
23:   C1 = vld1q_f32(C+4);
24:   C2 = vld1q_f32(C+8);
25:   C3 = vld1q_f32(C+12);
26:   /* Make multiply accumulate*/
27:   C0 = vmlaq_lane_f32(C0,B0,I0,0);           ▷ multiply with I0 lane 0 to all B0 elemnents
28:   C1 = vmlaq_lane_f32(C1,B1,I0,0);
29:   C2 = vmlaq_lane_f32(C2,B2,I0,0);
30:   C3 = vmlaq_lane_f32(C3,B3,I0,0);
31:   /* Store the partial sums to memory */
32:   vst1q_f32(C,C0);                           ▷ Store values from register C0 into memory address C
33:   vst1q_f32(C+4,C1);
34:   vst1q_f32(C+8,C2);
35:   vst1q_f32(C+12,C3);
36: }
```

The intrinsics are C based and as such is the syntax as well. It should be noted that the vector indexations on line **14** in Algorithm 3, is now replaced with actual pointers.

8.6.2 Dense matrices using NEON

The sub algorithm also holds some dense matrix-vector products, when we are to compute GRU_B. The approach is much the same, in terms of the data which have to be processed by the NEON co-processor. Instead of using nine registers at a time, we can simply consider the bare minimum of three Q-registers; input values, weights and partial sums, much in line with Figure 8.5.

Something to note is that in both the cases of the matrix-vector products of $\mathcal{U}_B y^{(t)}$ and $\mathcal{W}_B h_b^{(t-1)}$, the

number of GRU units within the layer N is 16. Hence $\mathcal{U}_B$ is a 48×512 ($3N \times M$) matrix and $\mathcal{W}_B$ is a 48×16 ($3N \times N$) matrix.

We can thus for the $\mathcal{U}_B y^{(t)}$ matrix-vector product store all partial sums within the NEON-registers, require 12 of the 32 available registers. Hence we can actually avoid writing to cache, until this entire matrix-vector product is computed.

As for the $\mathcal{W}_B h_b^{(t-1)}$ matrix-vector produce, is it not possible to compute the whole matrix-vector product due to the precedence of the GRU. Hence we have to compute $W^{z,r} h_b^{(t-1)}$ and $W^{\tilde{h}} h_b^{(t-1)}$ separately. Never the less can we in the case of computing on the recurrent weight store both the partial sums (32 or 16 values) and the whole input (16 values), but the weights (512 or 256 values) will have to be read from the cache over multiple iterations as we perform the MAC operations.

In both cases of GRU_A and GRU_B we have the same opportunity regarding including the bias as presented in section 7.3.2. Thus we can save a few additions by loading the associated biases into the registers which will hold the partial sums. It is however more tedious for the sparse matrices, as we do not load and work on the partial sums in a straightforward manner.

8.6.3 Vector operations using NEON

The remaining of the GRU layers consists of either vector transformations by the activation functions or vector-vector operations by the element-wise multiplication or vector-subtraction. Hence they are less complex and easier to gain an overview on. These vectors are of size N, that is either 384 for GRU_A or 16 for GRU_B.

For GRU_B these vectors are sufficiently small, therefore we can hold all inputs, operands and result within the registers. As a result we can load all these once, perform the necessary computations and then write all results to the cache if wanted. It may however be better to write to cache iteratively, instead of one big chuck, depending on how the memory system and buses handles these scenarios.

8.7 Summary

In this chapter we have described the targeted CPU, of which the sub-algorithm might be implemented on. This description covers the most essential processing and memory systems. We furthermore dive into which issues might occur during computations, that might stall the processing pipeline and therefore reduce the throughput.

We describe some initial considerations regarding the memory system and how we can take advantage of the structure the sub-algorithm present, making parallel computations easy using the NEON extension provided by ARM within the CPU. We give an example on how the matrix-vector computations of the sparse matrices, could be handled using NEON intrinsics.

We describe the sub-algorithm in a pseudo code manner in order to gain overview and insight of the sub-algorithm. Followed by a discussion on how we can compute this sub-algorithm, when we are mindful of possible issues which causes stalls, and thus reductions of the throughput. This discussion includes considerations on how we can make cache friendly data arrays, how we can compute the sparse matrix-vector products using the BCSC format, pipeline stalls cause by data, control and structural hazards.

Thus we have now presented our thoughts on how the sub-algorithm could be implemented on the targeted CPU, using the available features the CPU offers. These thoughts and considerations should serve as the stepping stones for an actual implementation of the sub-algorithm.

Part IV

Final thoughts

Chapter 9

Evaluation

Thus we have now presented the reader with thoughts on an implementation of the chosen sub-algorithm from both an FPGA and a CPU based approach. We thus conclude our work with the LPCNet in this last chapter, where we first take a discussion of the two implementation approaches we have presented in this third part regarding architectures. We then follow up on this discussion by yielding some recommendations for implementation of this sub-algorithm and the LPCNet, based on our findings made in this third part of the project. Third we make our concluding remarks on this project, and lastly we look into possible future works based on this project. First we would like to give a summary of what this project have consisted of.

9.1 Summary of our work

In chapter 2 and 3 we present some of the core knowledge required to understand and analyse the neural network that is the LPCNet. This included topics such as the essentials of linear predictive coding, the basics of multilayer perceptrons, and later the recurrent neural networks with the inclusion of gated networks here among the gated recurrent unit used in the LPCNet.

Here after did we give a functional description and analysis of the LPCNET given i chapter 4, which included specific methods used in neural networks, which were beyond the basic theory of recurrent neural networks. These methods include embedding layers, pruning and sparse matrices, probability distribution sampling and training noise injection. Furthermore did the chapter consists of an exposition of the main layers that make up the LPCNet including an account of the number of weights and computational complexity of said layers.

Following this theory, analysis and examination of the network we set out to train our own model in chapter 5, based on the LPCNet's source code provided by [Valin and Skoglund, 2019a]. This included the necessary pre- and post processing of data, speech quality assessment and the Claaudia platform used to train the neural network. This concluded in a number of models with varying quality and complexity.

During our training and evaluation of our own LPCNet models, using the TIMIT data set for both. We found an error within the source code presented in [Valin and Skoglund, 2019a], which meant that the pruning only processed parts of the supposed weights within the network, hence would the complexity of the network trained with this faulty pruning been significantly higher than what is presented in

[Valin and Skoglund, 2019a]. The source code has links to pre-trained models, no longer available[1] at the link presented i chapter 5. These pre-trained models also had the same error in regards to the none pruned weights. We corrected the error and trained new models, based on experiences of our initial training campaign. From these experiences we concluded that the initial training period of 120 epochs lead to an over-fitted model, and a better training period lies within a range of 30 to 90 depending on the given model. We managed to marginally improve the performance of the original model by removing unused features, which seems to be a remnant from [Valin and Skoglund, 2019a]'s development of the LPCNet. From these new models objective audio quality scores were obtained, and seemingly where these not comparable to the quality presented in [Valin and Skoglund, 2019a].

The quality of the synthesized speech presented in this work have however been lackluster, both objectively and in our own subjective opinion. We have identified a main contributed to this as shown in chapter 5. Which means that the synthesized speech is intelligible in general, does it sound unnatural during voiced sequences of speech. While the pitch period problem have been identified, the solution to this have not been found in our work.

We then tasked our self with choosing a suitable part of the network for deeper analysis in chapter 6. We performed a survey of existing methods of implementation of neural networks, and based on this survey, interest and the complexity of the chosen part, an sub-algorithm of the LPCNet were chosen. This sub-algorithm consists of the two GRU layers within LPCNet.

The analysis lead to deeper knowledge of the inner workings of the chosen sub-algorithm, limitations, dataflow and inherent parallelism. During our analysis of the chosen sub-algorithm, we were able to both reaffirm some of the statements made, and shred light on the requirements of methods proposed in [Valin and Skoglund, 2019a], which was not highlighted in their work such as Section 6.2.1. We were also able to (to the best of our knowledge) present with our own addition to the computation of the block-sparse matrices, by making use of the structure present in these sparse matrices. We have however not be able to fully report on the results of the usage of this method, on any of our proposed implementation approaches (FPGA/CPU), due to the time frame of this project. As such we leave this as possible future work, where it would be exciting to see the result of this use.

Following the analysis we investigated two possible approaches to the implementation of this sub-algorithm, namely the use of an FPGA in chapter 7 and a CPU in chapter 8. On our take on the FPGA approach we examined existing data movement schemes designed to be energy efficient. Based on these data movement schemes we provided insight on the possibilities to compute the necessary layer within the sub-algorithm, and showed a general tendency. Where the more energy and computational efficiency the implementation is desired, the more complex the controls needs to be.

Lastly we investigated the approach of implementing the sub-algorithm on a CPU in chapter 8. Here we described the essential blocks required to compute the sub-algorithm and possible issues which could reduce the throughput of an solution. With this in mind we presented the sub-algorithm in terms of programming the sub-algorithm, while considering possible issues and gave ideas on how to minimize the effects of these. Additionally we showed how our proposed BCSC format could be a helpful tool, when making calculations on these sparse matrices. Lastly we showed how the CPU's Single Instruction Multiple Data based extension could take advantage of the massive inherent parallelism present in the sub-algorithm, in order to increase the CPU's throughput, marking the end of our work with the LPCNet and the sub-algorithm.

[1] As of 20 May 2020

In both of our implementation approaches, we have identified possible features or tools in order to compute the sub-algorithm. Furthermore have we presented some key considerations to make during an implementation, in order to better prepare a reader/designer for their own implementation of the sub-algorithm. Yet as earlier mentioned have we not been able to provide results of using these features or following the considerations, due to time constraints. It would be exciting to see how these considerations can affect an implementation, and the limitation they poses.

9.2 Discussion

In this project we have taken on the task of examining, reviewing and analyzing the neural network named LPCNet proposed by [Valin and Skoglund, 2019a]. As it is evident do the LPCNet utilize many different theories and methods. Combining both the use of classic signal processing method which is the linear predictive coding and the newer methods of neural network. In an effort to gain both the benefits of the low computational complexity inherent to LPC while also using the computational heavy neural network, which have been shown to achieve a high quality of speech at the given number of features [Valin and Skoglund, 2019a].

The LPCNet utilizes a considerable amount of theory and methods, and in this project we have analysed and highlighted the key aspects of this network, proposed implementation considerations and presented the stepping stones for an implementation of the sub-algorithm. We have not been able to present the result of these findings within the time frame of this project, and as such we leave this as future work.

9.2.1 Total computational complexity of LPCNet

It is difficult to say specifically the computational complexity of the LPCNet. In chapter 4 we estimated a complexity of 2.25 billion operations per second assuming that additions, subtraction, MACs and divisions require one operation. Additionally did we require to compute around 27,4 million exponential functions per second. We later discussed that these exponential functions made from activation functions could be computed as piece-wise linear functions instead, requiring one multiplication and one addition, thus the 27,4 million exponential functions and divisions needed for activation functions can be avoided.

In section 8.2 we showed that around 235 million MAC operations per second can be replaced with around 18.3 million additions per second, if we only compute matrix-vector products related to the conditional vector $\mathbf{f}$ once per frame. In section 8.4 we estimated that using the BCSC to compute the sparse matrices added an additional 380 MFLOPS of overhead, while also adding 6N (2,304) extra MAC operations every time GRU_A is computed, that is around 37 million MAC operations per second. If we again assume that one MAC operation requires one floating point operation, and add all these numbers together we have an estimated computational complexity of around 2.45 GFLOPS, in order to compute the frame and sample rate network. We expect that the LPC coefficients also computed in LPCNet have a negligible complexity compare to this number. Hence we estimate the computational complexity of the LPCNet to be around 2.5GFLOPS in total (including LPC coefficients, *mu*-law companding and similar).

9.2.2 FPGA or CPU based platforms

Not having presented results of the two investigated platforms for implementation, it is not possible to yield concrete evidence whether the FPGA is better than the CPU and vise visa. The particular FPGA we have highlighted have a total of 34 math blocks capable of performing MAC operations, however these math blocks only supports up to 18 bit × 18 bit signed multiplications. Thus if the inputs, weights and partial sums cannot be represented in less than single precision (32 bits). One have to use four of these math blocks jointly, and thus we can only make a total of 8 MAC operations in parallel.

In comparison can the NEON extension to the ARM CPU make four 32 bit MAC operations in parallel on each of its four cores, hence a total of 16 MAC operations in parallel. Thus can the CPU make many more MAC operations in parallel, and since the NEON extension also supports numbers represented in fewer bits (16 and 8), the CPU seems to be able to make a twice as many MACs at the typical bit depths (32bit, 16bit and 8bit), in parallel compared to the FPGA.

Which of the two platforms is faster, is hard to say without knowing how many clock cycles these operations take at the specific bit depths, and how data is accessed from the memory. Furthermore would it seem very likely that the CPU also requires an operating systems, which also will require hardware resources, while the FPGA may be able to avoid the use of such advanced controllers. It would seem that at first impressions the CPU based platform would be the most likely to achieve real time speech synthesis using the LPCNet, from a purely speculative standpoint.

9.3 Recommendations of implementation

Through our work with the chosen sub-algorithm of the LPCNet, we are able to make some general suggestions, if one where to implement the sub-algorithm. Suggestions which may also be transferable the remaining neural network, since many of the mathematical operations performed in the sub-algorithm are the same, especially when it comes to matrix-vector products.

- As with many neural network does the sub-algorithm and LPCNet hold potential for massive parallelism as both algorithms holds much inherent parallelism
  - Thus should all potential tools or features capable of performing parallel computations, of a given platform be examined, in order to identify potential acceleration of the algorithms. Especially SIMD based parallelism have proven to be useful.
- The architecture capability to make MAC operations is a necessity, if the computational complexity is to be hold at a reasonable level.
- The sparse weight matrices of GRU_A holds potential to greatly reduce memory and computational requirements, but are complex to handle.
  - These matrices do hold a significant amount of the weights and therefore also computations, thus should efforts be made to identify how to best compute handle these.
  - If the sparsity of these matrices are not addressed, the neglecting of this property will lead to a vastly increased complexity, making real-time synthesis infeasible for almost any platform.
  - The BCSC format can help handling these block-sparse matrices, an already existing alternative to this is the CSC format. Which may pose a slightly smaller computational complexity, but would require almost twice the memory (see section 6.2.3).
- Be mindful of alternative ways to fulfill certain functionality.

  - For example different approaches to compute activation functions, which is more aligned with the types of operations SIMD based computing can provide - e.g. the piece-wise linear approximation of activation functions.
- Be mindful of data movement and data reuse, since exploiting these may provide better energy efficiency and higher throughput.
- The use of FPGAs provides higher flexibility of the solution to be made, but the control of the data flow can be very complex, and can thus be very time consuming to implement - compared to the use of a CPU.
- When using a CPU as the platform of implementation, one should be mindful of the possible stalls the sub-algorithm poses. The sub-algorithm's biggest reason for stalls would be in regard to move data from lower levels of memory to the processing units of the CPU, and thus should this be the primary type of stalls to address.

9.4 Conclusion

In this project the neural network named LPCNet proposed by [Valin and Skoglund, 2019a] has been examined, evaluated and analysed with the intent for a possible implementation. Based on available source code provided by the proposers, have multiple models been trained and evaluated over the course of two campaigns.

During the first of the two, we discovered an error within source code, and within existing pre-trained models seemingly made by the proposers of LPCNet. This error was corrected and a second campaign were made. It was shown that the correction had little to no negative impact on the accuracy of the model. A model was made which saw a smaller gain in accuracy, compared to the base line model. In general does the models produce intelligible speech, but the naturalness of the speech is poor. One of the problems causing this poor naturalness have been identified as a pitch related error, but a resolution to this issue have not been made within the time frame of this project.

Based on the model and a wish to implement this on an embedded device. A survey of existing methods to efficiently implement such a network on embedded devices was made. The efficient implementation covers topics such as memory reduction, hardware acceleration, matrix transformations, matrix factorization and more. Based on this survey and novelty value it was decided to delimit the implementation and analysis of the LPCNet, to the most central and computational heavy part of the network, the so called sub-algorithm. Consisting of the two GRU layers within the LPCNet.

based on the sub-algorithm a deeper analysis was made on the weights, the inherent parallelism and inner workings of this sub-algorithm. From the analysis of the pruned weights of the sub-algorithm it was found that the structure of these sparse matrices, could be exploited in order to further reduce the memory required to store these matrices, outperforming existing solutions, hence the Block Compressed Sparse Column format was introduced. The analysis also provided insight to potential weaknesses with proposed computation saving methods in [Valin and Skoglund, 2019a] - i.e. the embedding layers discussed in Section 6.2.1.

Based on this analysis of the sub-algorithm two approaches were made in regards to the platform of its implementation, namely an FPGA based platform and a CPU based platform. On the basis of these platforms numerous, schemes, formats, dataflows and considerations where made. It was presented how the proposed Block Compressed Sparse Column format could be involved, as a way to deal with

the sparse matrices present in the sub-algorithm for both platforms. It has not been possible to display the results of this analysis and different considerations within the time frame of this project.

This project's contribution is the general analysis, explanations and considerations provided to interested readers. Further contributions include the discovery and correction of an error in the provided source code of LPCNet, identification of why the trained models have poor accuracy i.e. speech quality, survey of existing methods to make efficient implementation of neural network, the Block Compressed Sparse Column format as means to handle block-sparse matrices, and recommendations for implementing the sub-algorithm and to some extent the LPCNet.

With respect to the points of interest presented in section 1.3 we can conclude the following;

Our trained models achieve intelligible yet unnatural sound speech, with a MOS of 2.435 without increasing the complexity compared to the reference model using the `fix_RNF_NEW_BS64E60_L3.30` model and the TIMIT data set for training and testing. We estimate that the computational complexity of the LPCNet is around 2.5GFLOPS.

We cannot conclude if the presented FPGA and CPU based platforms are suitable for running the LPCNet or sub-algorithm in real-time. We think of the CPU based platform as the most plausible of achieving this. We conclude that SIMD and like-wise parallel computing options is a necessity and architectures capable of performing MAC operations are also more or less necessary.

9.5 Future works

Future works of this project includes improvements on the naturalness of the synthesised speech based on the discoveries made in chapter 5. Inclusion of the work made in [Valin and Skoglund, 2019b], in order to introduce low bit rate speech synthesis using the knowledge gained in this project. Investigation of the requirements and performance of the proposed Block Compressed Sparse Column format on embedded devices in a practical scenario. Implementation of the sub-algorithm and LPCNet on both proposed platforms, in order to gain further insights and performance results of the many design considerations made in this project.

Bibliography

[Voc, 2019] (2019). Vocoder wikipedia. Last accessed 6 September 2019.

[ARM, 2013] ARM (2013). Neon - progammer's guide.

[ARM, 2015] ARM (2015). Arm cortex-a series - programmer's guide for armv8-a.

[Aäron van den Oord, 2016] Aäron van den Oord, Sander Dieleman, H. Z. K. S. O. V. A. G. N. K. A. W. S. K. K. (2016). Wavenet : A generative model for raw audio. In *arXiv:1609.03499v2*. Google.

[Bengio et al., 1993] Bengio, Y., Frasconi, P., and Simard, P. (1993). The problem of learning long-term dependencies in recurrent networks. In *IEEE International Conference on Neural Networks*, pages 1183–1188 vol.3.

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

[Beric et al., 2008] Beric, A., van Meerbergen, J., de Haan, G., and Sethuraman, R. (2008). Memory-centric video processing. *IEEE Transactions on Circuits and Systems for Video Technology*, 18(4):439–452.

[Bishop, 2006] Bishop, C. M. (2006). *Pattern Recognition and Machine Learning*. springer.

[Chen et al., 2015] Chen, W., Wilson, J. T., Tyree, S., Weinberger, K. Q., and Chen, Y. (2015). Compressing neural networks with the hashing trick. In *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37*, ICML'15, page 2285–2294. JMLR.org.

[Chen et al., 2016] Chen, Y., Krishna, T., Emer, J. S., and Sze, V. (2016). Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE Journal of Solid-State Circuits*, 52(1):127–138.

[Chen et al., 2019] Chen, Y., Yang, T., Emer, J., and Sze, V. (2019). Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 9(2):292–308.

[Cheng et al., 2005] Cheng, O., Abdulla, W., and Salcic, Z. (2005). Performance evaluation of front-end processing for speech recognition systems. *The University of Auckland*.

[Cheng et al., 2015] Cheng, Y., Yu, F. X., Feris, R. S., Kumar, S., Choudhary, A., and Chang, S. (2015). An exploration of parameter redundancy in deep networks with circulant projections. In *2015 IEEE International Conference on Computer Vision (ICCV)*, pages 2857–2865.

[Cho et al., 2014a] Cho, K., van Merriënboer, B., Bahdanau, D., and Bengio, Y. (2014a). On the properties of neural machine translation: Encoder–decoder approaches. In *Proceedings of SSST-8, Eighth Workshop on Syntax, Semantics and Structure in Statistical Translation*, pages 103–111, Doha, Qatar. Association for Computational Linguistics.

[Cho et al., 2014b] Cho, K., van Merrienboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., and Bengio, Y. (2014b). Learning phrase representations using RNN encoder–decoder for statistical machine translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Association for Computational Linguistics.

[Chung et al., 2014] Chung, J., Gulcehre, C., Cho, K., and Bengio, Y. (2014). Empirical evaluation of gated recurrent neural networks on sequence modeling. In *NIPS 2014 Workshop on Deep Learning, December 2014*.

[Collobert and Weston, 2008] Collobert, R. and Weston, J. (2008). A unified architecture for natural language processing. In *Proceedings of the 25th international conference on Machine learning - ICML '08*. ACM Press.

[Cun et al., 1990] Cun, Y. L., Denker, J. S., and Solla, S. A. (1990). Optimal brain damage. In *Advances in Neural Information Processing Systems*, pages 598–605. Morgan Kaufmann.

[Fastl and Zwicker, 2007] Fastl, H. and Zwicker, E. (2007). *Psychoacoustics*. Springer Berlin Heidelberg.

[Flynn, 1972] Flynn, M. J. (1972). Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, C-21(9):948–960.

[Fowers et al., 2018] Fowers, J., Ovtcharov, K., Papamichael, M., Massengill, T., Liu, M., Lo, D., Alkalay, S., Haselman, M., Adams, L., Ghandi, M., Heil, S., Patel, P., Sapek, A., Weisz, G., Woods, L., Lanka, S., Reinhardt, S. K., Caulfield, A. M., Chung, E. S., and Burger, D. (2018). A configurable cloud-scale DNN processor for real-time AI. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE.

[Gang Min et al., 2015] Gang Min, Xiongwei Zhang, Jibin Yang, and Xia Zou (2015). Speech reconstruction from mel-frequency cepstral coefficients via l1-norm minimization. In *2015 IEEE 17th International Workshop on Multimedia Signal Processing (MMSP)*, pages 1–5.

[Gers et al., 1999] Gers, F. A., Schmidhuber, J., and Cummins, F. (1999). Learning to forget: continual prediction with lstm. In *1999 Ninth International Conference on Artificial Neural Networks ICANN 99. (Conf. Publ. No. 470)*, volume 2, pages 850–855 vol.2.

[Golub and Loan, 2013] Golub, G. H. and Loan, C. F. V. (2013). *matrix computations, 4th edition*. Johns Hopkins University Press.

[Goodfellow et al., 2016] Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep Learning*. MIT Press. `http://www.deeplearningbook.org`.

[Graves, 2013] Graves, A. (2013). Generating sequences with recurrent neural networks. *ArXiv*, abs/1308.0850.

[Gray and Kingma, 2017] Gray, S., R. A. and Kingma, D. B. (2017). block-sparse gpu kernels. Last accessed 17 October 2019.

[Gray, 2005] Gray, R. M. (2005). Toeplitz and circulant matrices: A review. *Foundations and Trends® in Communications and Information Theory*, 2(3):155–239.

[Greff et al., 2017] Greff, K., Srivastava, R. K., Koutník, J., Steunebrink, B. R., and Schmidhuber, J. (2017). Lstm: A search space odyssey. *IEEE Transactions on Neural Networks and Learning Systems*, 28(10):2222–2232.

[Gulzar et al., 2014] Gulzar, T., Singh, A., and Sharma, S. (2014). Comparative analysis of lpcc, mfcc and bfcc for the recognition of hindi words using artificial neural networks. *International Journal of Computer Applications*, 101:22–27.

[Han et al., 2017] Han, S., Kang, J., Mao, H., Hu, Y., Li, X., Li, Y., Xie, D., Luo, H., Yao, S., Wang, Y., and et al. (2017). Ese: Efficient speech recognition engine with sparse lstm on fpga. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '17, page 75–84, New York, NY, USA. Association for Computing Machinery.

[Han et al., 2016] Han, S., Mao, H., and Dally, W. J. (2016). Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding. *ICLR 2016*, abs/1510.00149.

[Han et al., 2015] Han, S., Pool, J., Tran, J., and Dally, W. (2015). Learning both weights and connections for efficient neural network. In Cortes, C., Lawrence, N. D., Lee, D. D., Sugiyama, M., and Garnett, R., editors, *Advances in Neural Information Processing Systems 28*, pages 1135–1143. Curran Associates, Inc.

[Hochreiter, 1991] Hochreiter, S. (1991). Untersuchungen zu dynamischen neuronalen netzen. diploma thesis.

[Hochreiter and Schmidhuber, 1997] Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural Comput.*, 9(8):1735–1780.

[Horowitz, 2014] Horowitz, M. (2014). 1.1 computing's energy problem (and what we can do about it). In *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pages 10–14.

[Jin et al., 2018] Jin, Z., Finkelstein, A., Mysore, G. J., and Lu, J. (2018). Fftnet: A real-time speaker-dependent neural vocoder. In *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 2251–2255.

[Jozefowicz et al., 2016] Jozefowicz, R., Vinyals, O., Schuster, M., Shazeer, N., and Wu, Y. (2016). Exploring the limits of language modeling.

[Kalchbrenner et al., 2018] Kalchbrenner, N., Elsen, E., Simonyan, K., Noury, S., Casagrande, N., Lockhart, E., Stimberg, F., van den Oord, A., Dieleman, S., and Kavukcuoglu, K. (2018). Efficient neural audio synthesis. In Dy, J. and Krause, A., editors, *Proceedings of the 35th International*

Conference on Machine Learning, volume 80 of *Proceedings of Machine Learning Research*, pages 2410–2419, Stockholmsmässan, Stockholm Sweden. PMLR.

[Kay, 2012] Kay, S. M. (2012). *Intuitive Probability and Random Processes using MATLAB®*. Springer, 4 edition.

[Kleijn et al., 2018] Kleijn, W. B., Lim, F. S. C., Luebs, A., Skoglund, J., Stimberg, F., Wang, Q., and Walters, T. C. (2018). Wavenet based low rate speech coding. In *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE.

[Krizhevsky et al., 2012] Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In Pereira, F., Burges, C. J. C., Bottou, L., and Weinberger, K. Q., editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc.

[Kuchaiev and Ginsburg, 2017] Kuchaiev, O. and Ginsburg, B. (2017). Factorization tricks for lstm networks. *ArXiv*, abs/1703.10722.

[Levinson, 1947] Levinson, N. (1947). The wiener rms (root mean square) error criterion in filter desgin and prediction. *Journal of Mathematical Physics*, 25(4):261–278.

[Li et al., 2019] Li, Z., Ding, C., Wang, S., Wen, W., Zhuo, Y., Liu, C., Qiu, Q., Xu, W., Lin, X., Qian, X., and Wang, Y. (2019). E-rnn: Design optimization for efficient recurrent neural networks in fpgas. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 69–80.

[Makhoul, 1973] Makhoul, J. (1973). Spectral analysis of speech by linear prediction. *IEEE Transactions on Audio and Electroacoustics*, 21(3):140–148.

[Makhoul, 1975] Makhoul, J. (1975). Linear prediction: A tutorial review. *Proceedings of the IEEE*, 63(4):561–580.

[Mathieu et al., 2013] Mathieu, M., Henaff, M., and Lecun, Y. (2013). Fast training of convolutional networks through ffts.

[Minsky and Papert, 1969] Minsky, M. and Papert, S. (1969). *Perceptrons: An Introduction to Computational Geometry*. MIT Press.

[Narang et al., 2017] Narang, S., Diamos, G., Sengupta, S., and Elsen, E. (2017). Exploring sparsity in recurrent neural networks.

[Narang et al., 2018] Narang, S., Undersander, E., and Diamos, G. (2018). Block-sparse recurrent neural networks.

[Oppenheim and Schafer, 2013] Oppenheim, A. V. and Schafer, R. W. (2013). *Discrete-Time Signal Processing*. Pearson Education Limited, 3 edition.

[Oseledets, 2011] Oseledets, I. V. (2011). Tensor-train decomposition. *SIAM Journal on Scientific Computing*, 33(5):2295–2317.

[Pascanu et al., 2013] Pascanu, R., Gulcehre, C., Cho, K., and Bengio, Y. (2013). How to construct deep recurrent neural networks, iclr 2014.

[Patterson and Hennessy, 2013] Patterson, D. and Hennessy, J. (2013). *Computer Organization and Design ARM Edition: The Hardware Software Interface*. Morgan Kaufmann Publishers In, 5 edition.

[Patterson and Hennessy, 2016] Patterson, D. and Hennessy, J. (2016). *Computer Organization and Design ARM Edition - The Hardware Software Interface: ARM edition*. Morgan Kaufmann Publishers In.

[Rabiner and Schafer, 1978] Rabiner, L. R. and Schafer, R. W. (1978). *Digital Processing of Speech Signals*. Prentice-Hall.

[Reddi et al., 2018] Reddi, S. J., Kale, S., and Kumar, S. (2018). On the convergence of adam and beyond. In *International Conference on Learning Representations*.

[Rosenblatt, 1958] Rosenblatt, F. (1958). The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386.

[Rosenblatt, 1962] Rosenblatt, F. (1962). *Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms*. Spartan.

[Rumelhart et al., 1986] Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986). Learning representations by back-propagating errors. *Nature*, 323(6088):533–536.

[Saad, 2003] Saad, Y. (2003). *Iterative Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics.

[Sindhwani et al., 2015] Sindhwani, V., Sainath, T., and Kumar, S. (2015). Structured transforms for small-footprint deep learning. In Cortes, C., Lawrence, N. D., Lee, D. D., Sugiyama, M., and Garnett, R., editors, *Advances in Neural Information Processing Systems 28*, pages 3088–3096. Curran Associates, Inc.

[Soroush Mehri, 2017] Soroush Mehri, Kundan Kumar, I. G. R. K. S. J. J. S. A. C. Y. B. (2017). Samplernn: An unconditional end-to-end neural audio generation model. ICLR 2017.

[Sussillo, 2014] Sussillo, D. (2014). Random walks: Training very deep nonlinear feed-forward networks with smart initialization. *ArXiv*, abs/1412.6558.

[Sze et al., 2017] Sze, V., Chen, Y., Yang, T., and Emer, J. S. (2017). Efficient processing of deep neural networks: A tutorial and survey. *Proceedings of the IEEE*, 105(12):2295–2329.

[Thakker et al., 2019] Thakker, U., Fedorov, I., Beu, J. G., Gope, D., Zhou, C., Dasika, G., and Mattina, M. (2019). Pushing the limits of rnn compression. *ArXiv*, abs/1910.02558.

[Tjandra et al., 2017] Tjandra, A., Sakti, S., and Nakamura, S. (2017). Compressing recurrent neural network with tensor train. In *2017 International Joint Conference on Neural Networks (IJCNN)*. IEEE.

[Valin and Skoglund, 2019a] Valin, J.-M. and Skoglund, J. (2019a). LPCNET: Improving neural speech synthesis through linear prediction. In *ICASSP 2019 - 2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE.

[Valin and Skoglund, 2019b] Valin, J.-M. and Skoglund, J. (2019b). A real-time wideband neural vocoder at 1.6 kb/s using lpcnet. In *arXiv:1903.12087v2*.

[van den Oord et al., 2016] van den Oord, A., Kalchbrenner, N., Vinyals, O., Espeholt, L., Graves, A., and Kavukcuoglu, K. (2016). Conditional image generation with pixelcnn decoders.

[Volder, 1959] Volder, J. (1959). The cordic computing technique. volume 1, page 257. National Joint Computer Committee.

[Walther, 1971] Walther, J. S. (1971). A unified algorithm for elementary functions. In *Proceedings of the May 18-20, 1971, Spring Joint Computer Conference*, AFIPS '71 (Spring), page 379–385, New York, NY, USA. Association for Computing Machinery.

[Wang et al., 2019] Wang, M., Wang, Z., Lu, J., Lin, J., and Wang, Z. (2019). E-lstm: An efficient hardware architecture for long short-term memory. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 9(2):280–291.

[Wang et al., 2018] Wang, S., Li, Z., Ding, C., Yuan, B., Qiu, Q., Wang, Y., and Liang, Y. (2018). C-lstm: Enabling efficient lstm using structured compression techniques on fpgas. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '18, page 11–20, New York, NY, USA. Association for Computing Machinery.

[Wang et al., 2017] Wang, Z., Lin, J., and Wang, Z. (2017). Accelerating recurrent neural networks: A memory-efficient approach. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 25(10):2763–2775.

[Xie et al., 2009] Xie, M., Chu, P., Taleb, A., and Briand, M. (2009). ITU-t g.719: A new low-complexity full-band (20 kHZ) audio coding standard for high-quality conversational applications. In *2009 IEEE Workshop on Applications of Signal Processing to Audio and Acoustics*. IEEE.

[Zhu and Gupta, 2017] Zhu, M. and Gupta, S. (2017). To prune, or not to prune: exploring the efficacy of pruning for model compression.

[Zwicker and Terhardt, 1980] Zwicker, E. and Terhardt, E. (1980). Analytical expressions for critical-band rate and critical bandwidth as a function of frequency. *The Journal of the Acoustical Society of America*, 68(5):1523–1525.

Appendix A

Appendix

A.1 Levinson-Durbin Algorithm

Here we will give a brief outline of the Levinson-Durbin algorithm. We will not derive it since one can use it without its proof, and since it is an algorithm already covered in many textbooks and other sources. We will base our outline of the algorithm, based on [Oppenheim and Schafer, 2013] and its presentation of it, as shown in 5.

Algorithm 5 Levinson-Durbin algorithm

1: $\varepsilon[0] = r_{ss}[0]$
2: **for** i = 1:p **do**
3: /* Compute reflection coefficient */
4: k[i] = $\left(r_{ss}[i] - \sum_{j=1}^{i-1}\left[a[j] \cdot r_{ss}[i-j]\right]\right)/\varepsilon[i-1]$
5: /* Define the i'th prediction coefficients as the reflection coefficient */
6: a[i] = k[i]
7: **for** j = 1:i-1 **do**
8: /* Re-compute all former prediction coefficients,
9: based on them self and the current reflection coefficient */
10: a[j] = a[j] - k[i] · a[i-j]
11: /* Compute the new mean-squard prediction error */
12: ε[i] = $\left(1 - k[i]^2\right) \cdot \varepsilon$[i-1]
End

In line **1** we state that the mean-squared prediction error ε is equal to the autocorrelation of the signal with a lag equal to zero. Here after we compute the so-called reflection coefficient in line **4**, the term is connected to transmission theory and it plays a similar role as impedance ratios do during transmission of a wave through two sections with different impedance [Makhoul, 1975]. It can be shown that the reflection coefficient always have a magnitude strictly less than one [Oppenheim and Schafer, 2013]. Often we see that the reflection coefficient settles in around zero as the prediction order is increased

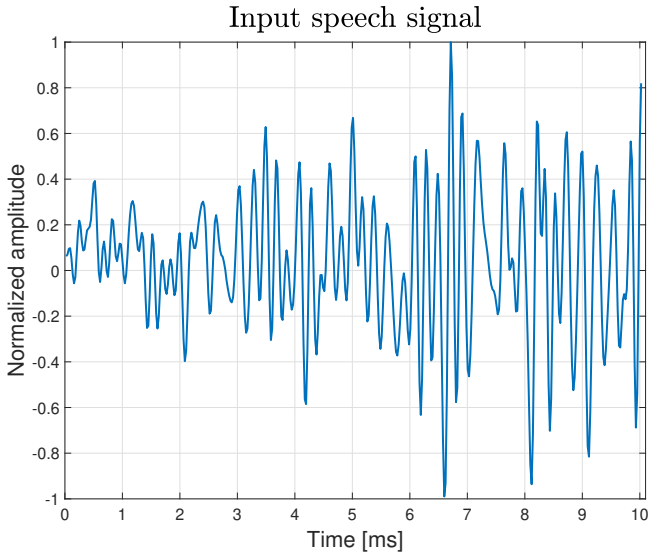
Based on the new reflection coefficient is the i'th prediction coefficient defined and the former prediction coefficients (1..i-1) are re-computed based on this new reflection coefficient in line **6** through **10**.

Lastly is the new prediction error computed, based on the reflection coefficient and the old prediction error in line **12**. It can be shown that the prediction error will come closer or remain the same, as the prediction order increases. Which is in agreement with our immediate line of thought, since as

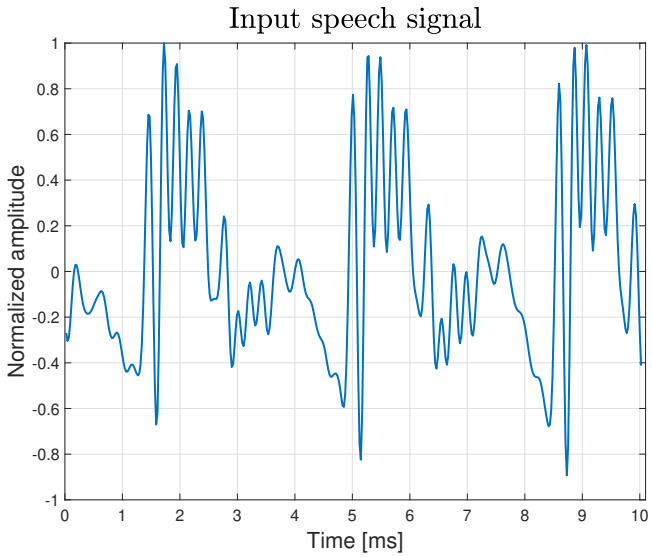
we increase the order, we come bring our approximation closer to the underlying model, which have produced the signal.

In the process of computing the p'th prediction coefficient, do one actually also compute all other predictors of orders smaller than p i.e. 1...p-1.

As an example on this algorithm, have we taken two speech inputs and perform this autocorrelation method on. In Figure A.1 we have sampled an unvoiced A.1a and a voiced A.1b speech section, a Hamming window have been applied to both signals, and their are both 10 ms long.



(a) Unvoiced speech signal multiplied with a Hamming window

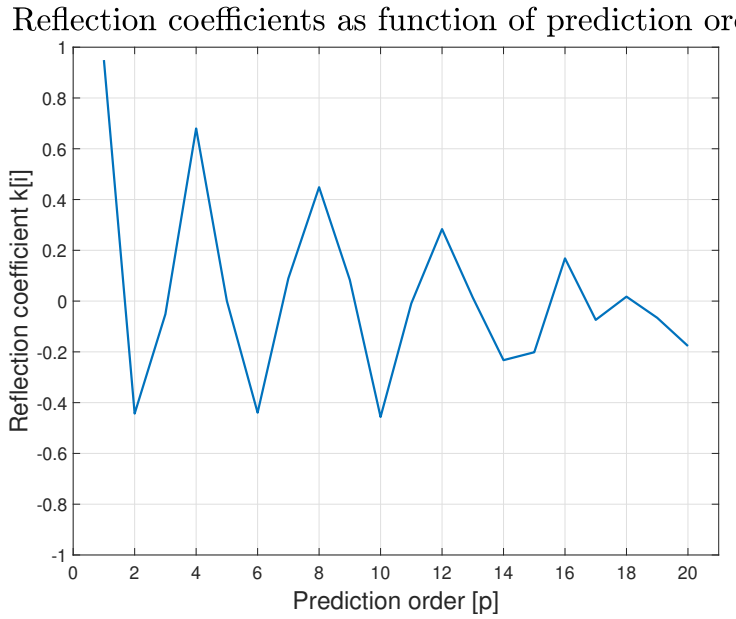


(b) Voiced speech signal multiplied with a Hamming window

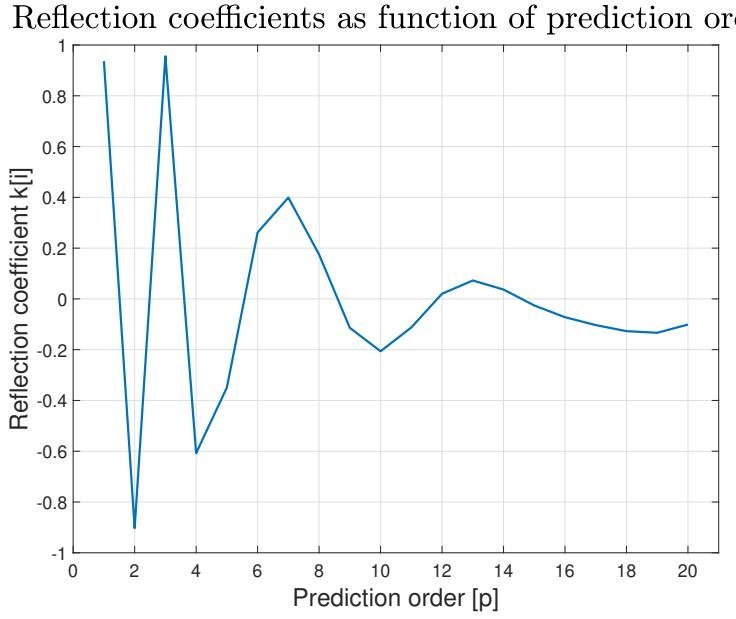
Figure A.1: Two speech signals, multiplied with a Hamming window

For both signals we compute the autocorrelations, and perform the Levinson-Durbin algorithm on these, in order to compute the prediction and reflection coefficients, and the mean-squared prediction error. In Figure A.2 we show the reflection coefficients as a function of the prediction order. It can be seen that the reflection coefficients never have a magnitude larger than one, and that is settles as

the prediction order increases. For voiced speech, the reflection coefficients are smoother and settles around zero earlier.



(a) Reflection of unvoiced speech



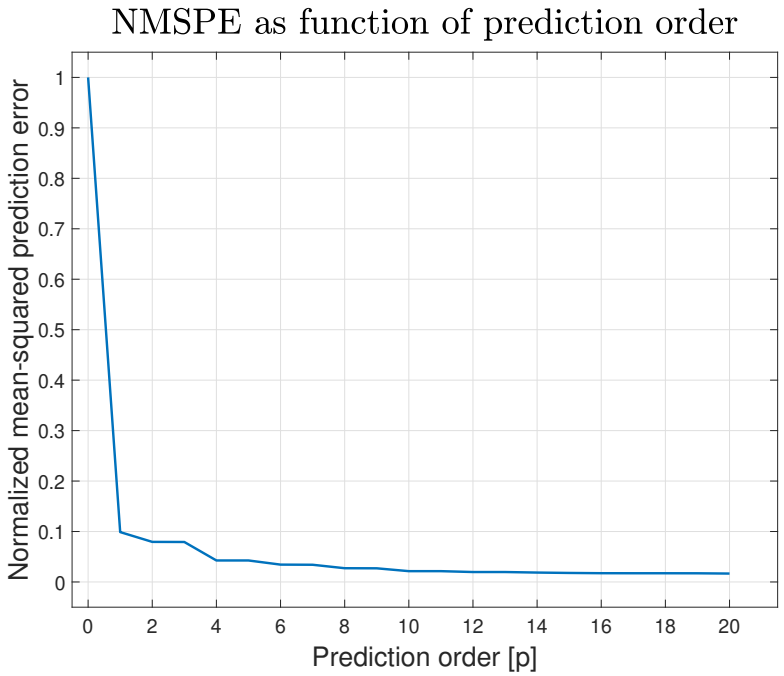
(b) Reflection of voiced speech

Figure A.2: Comparison of reflection coefficients of voiced and unvoiced speech.

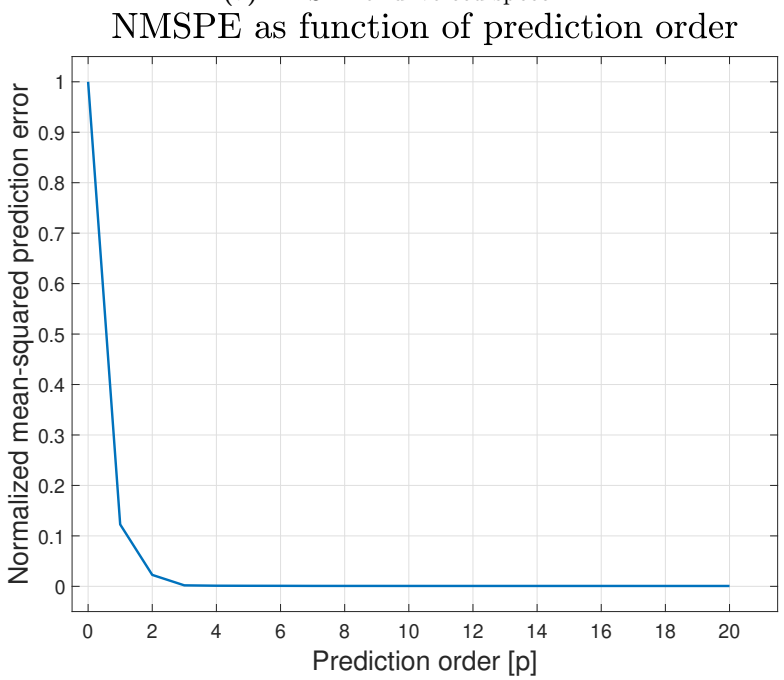
In Figure A.3 we show the normalized mean-squared prediction error ε as a function of the prediction order. From the reflection coefficients we get the impression that the unvoiced speech is harder to model, in the sense that it requires a higher prediction order before the reflection settles. This can easily be seen when we compare this prediction error. As the prediction error of voiced speech drops rapidly towards zero, in the means of a few orders and comes close to becoming zero. The unvoiced prediction requires a higher order and does not come as close to zero as the voice scenario.

Is concludes our brief outline of the Levinson-Durbin algorithm. More information on the algorithm and the LPC's relation to speech can be found in [Oppenheim and Schafer, 2013], along with the

covariance method and its effects on speech signals.



(a) NMSPE of unvoiced speech



(b) NMSPE of voiced speech

Figure A.3: Comparison of the normalize mean-squared prediction error, between voiced and unvoiced speech.

A.2 Quantization and μ-law in short

The basic of sampling quantization

Nyquest sampling theorem says that we should have a sampling rate, of at least the double frequency of the frequency we want to sample i.e.

$$f_s = 2 \cdot f_m \qquad (f_m \text{ is maximum signal frequency})$$

Typical sampling frequencies for speech is between 8kHz and 16 kHz and 44.1kHz or 48kHz for music. The quantization interval (or stepsize) Q, depends on the dynamic range of the signal amplitude and perceptual sensitivity. The interval Q and signal range D determines the bits/sample (bit depth) R in $2^R = D/Q$.

Typical values for speech is $R = 8$ bits and $R = 16$ bits for music. One can make a trade off between the sampling time T and the quantification interval Q. Lower bit depth requires higher sampling frequency and vice versa.

The basic quantization is uniform, meaning that the step size between any given values are equivalent to each other, hence the data range is divided into L equally sizes intervals. This is usable when the signal has a finite range.

$$Q = \frac{(f_{max} - f_{min})}{L}$$

Thus when a signal is quantized, the sample value is typically either rounded or floored to the nearest quantization level within the range of possible values.

μ-law quantization

For audio signals (speech and music) a problem arises as these signal is more concentrated near zero, and because the ears of humans have a higher sensitivity to quantization errors at smaller values, the usage of uniform quantization is problematic. A solution to this using non-linear quantization intervals, such that the quantization intervals are smaller at values near zero, and equivalently larger at values near the maximum of the dynamic range.

There are two major algorithm used to quantize speech name *mu*-law and A-law. These are used in telecommunications as a compander (compresser and expander). The idea of a compander is to first compress a given signal, such that bandwidth can be saved. The compressed signal is then transmitted and on the receiving end expanded again back to the original signal.

Both algorithm is based on logarithmic companding, based on human perception of sound intensity is logarithmic in nature. Since LPCNet only make use of μ-law we shall only examine it going forward. The μ-law compression have the following equation.

$$F(x) = sign(x)\frac{ln(1 + \mu \cdot |x|)}{ln(1 + \mu)} \qquad -1 \leq x \leq 1$$

Where $\mu = 255$. The subsequently expander for the compressed signal is.

$$F^{-1}(y) = sign(y) \cdot (\frac{1}{\mu}) \cdot ((1+\mu)^{|\mu|} - 1) \qquad -1 \leq y \leq 1$$

Both functions, along with the result of companding, is shown in Figure A.4

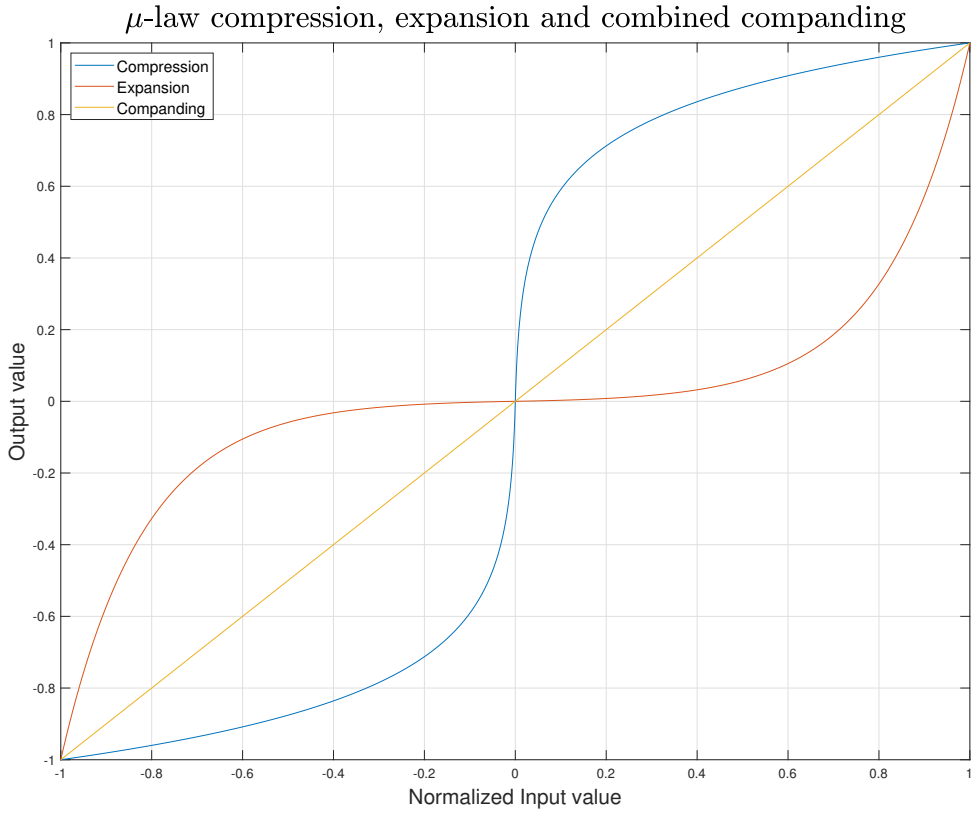


Figure A.4: μ-law companding

This logarithmic function is approximated into eight piece-wise linear functions. Since computing the logarithmic function can be computationally heavy compared to these linear function, and these straight lines is called a chord. Each chord is divided into equally sized quantization steps, and this steps size is doubled for each successive chord. Such that the first chords which handles the smallest parts of the dynamic range, have the smallest distance between each quantization step, where the human perception is most susceptible to quantization noise.

Thus the μ-law uses 8 bits to represent a given dynamic range. Where the most significant bit (MSB) is the sign bit, then three bits are used for representing chords and the remaining four for steps. In the standard μ-law used in among others the ITU-T G.711 standard, the μ-law algorithm receives an 14 bit-depth signal and compresses it into 8 bit. This compression of the input signal is known as a lossy compression since some of the data is discarded during compression, which cannot be accurately reproduced when the data is expanded after transmission. Instead is the least significant bits, which was discarded during compression, approximated by trailing a single 1 and subsequent zeros after the step values.

Additional technicality used in μ-law is that the sign bit for positive values is 1 and 0 for negative. As oppose to sign bits in the input signal which is in two's complement. Furthermore is the input to the

compressor bias by 33 (10.0001_b) for 14 bit input signals or 132 (100.0100_b) for 16 bit input signals. This is done such that the decision values for the chords is by power of two's.

After The sign bit is changed and the bias has been added, is the input encoded with the usage of the truth table shown in Table A.1. The truth table is for a 14-bit input signal, where the sign bit is omitted, values represented by x is input data which is discarded. Before the compressed data is transmitted, it is inverted by XOR'ing it by the code 0xFF, this is done to reduce the amount of 1's transmitted, since signals with low amplitude is more often seen than signal with large amplitudes.

Biased Input Values														Compressed Values							
															Chord			step			
bit	12	11	10	9	8	7	6	5	4	3	2	1	0	bit	6	5	4	3	2	1	0
	0	0	0	0	0	0	0	1	a	b	c	d	x		0	0	0	a	b	c	d
	0	0	0	0	0	0	1	a	b	c	d	x	x		0	0	1	a	b	c	d
	0	0	0	0	0	1	a	b	c	d	x	x	x		0	1	0	a	b	c	d
	0	0	0	0	1	a	b	c	d	x	x	x	x		0	1	1	a	b	c	d
	0	0	0	1	a	b	c	d	x	x	x	x	x		1	0	0	a	b	c	d
	0	0	1	a	b	c	d	x	x	x	x	x	x		1	0	1	a	b	c	d
	0	1	a	b	c	d	x	x	x	x	x	x	x		1	1	0	a	b	c	d
	1	a	b	c	d	x	x	x	x	x	x	x	x		1	1	1	a	b	c	d

Table A.1: μ-law compression encoding scheme

After the signal has been transmitted, it is inverted and decoded by the truth table shown in Table A.2. Finally is the sign bit restored and the bias is subtracted.

Compressed Values								Biased Output Values													
Chord				step																	
bit	6	5	4	3	2	1	0	bit	12	11	10	9	8	7	6	5	4	3	2	1	0
	0	0	0	a	b	c	d		0	0	0	0	0	0	0	1	a	b	c	d	1
	0	0	1	a	b	c	d		0	0	0	0	0	0	1	a	b	c	d	1	0
	0	1	0	a	b	c	d		0	0	0	0	0	1	a	b	c	d	1	0	0
	0	1	1	a	b	c	d		0	0	0	0	1	a	b	c	d	1	0	0	0
	1	0	0	a	b	c	d		0	0	0	1	a	b	c	d	1	0	0	0	0
	1	0	1	a	b	c	d		0	0	1	a	b	c	d	1	0	0	0	0	0
	1	1	0	a	b	c	d		0	1	a	b	c	d	1	0	0	0	0	0	0
	1	1	1	a	b	c	d		1	a	b	c	d	1	0	0	0	0	0	0	0

Table A.2: μ-law expanding encoding scheme

A.3 Bark scale to Linear Predictive Coding

As stated will we not dive deep into the LPC used in the LPCNet, as the subject is rather well described and since this part of the network do not pose a particular large part of its computational complexity.

For the sake of the broad comprehension of how the LPC coefficients are computed, we will present a brief exposition on how these coefficients are computed based on the input features.

The input features used to compute the LPC coefficients and thereby the prediction $p^{(t)}$ in the sample rate network, is the so call bark-scale cepstral coefficients. Cepstrum analysis and its properties is beyond the scope of this project, and hence we will only describe this subject superficial. Deeper description of cepstrum analysis and its applications can be found in [Rabiner and Schafer, 1978].

If we start on how these bark-scale cepstral coefficients is computed and then we will return to the LPC coefficients later. These bark-scale coefficients also goes under the name Bark Frequency Cepstral Coefficients (BFCC), computing these requires performing the sequence as shown in Figure A.5.

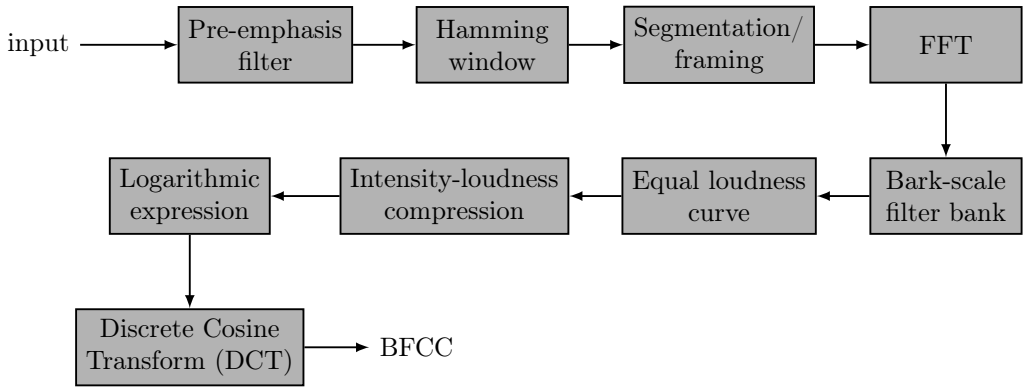


Figure A.5: Block diagram of the process for producing BFCC - modified of [Gulzar et al., 2014]

It can be seen from the figure that it is not a simple task to produce the BFCC's.

It require first that the input is applied a pre-emphasis filter to emphasise the higher frequencies of the audible spectrum. Then a hamming window is applied, the windowed signal is then divided into frames, of which an FFT is applied to. The spectrum is then multiplied with a bark-scale filter bank, each output of these filter banks weighted by the so called loudness curve, which seeks to approximate the sensitivity of the human ear. The intensity-loudness compression seeks to approximate our perception of loudness [Cheng et al., 2005].

Then one take the logarithmic of this expression and lastly one uses the Discrete Cosine Transformation (DCT) in order to obtain the BFCCs. The bark-scale and so called mel-scale cepstral coefficients is produces by almost the same process, expect for the loudness curve and intensity compression, as can be seen in [Gulzar et al., 2014].

First lets start with the bark-scale. The bark-scale is related to the critical bands. In short do these critical bands describe the bandwidth of sound signals where one tone can be masked by noise, if the noise spectrum lies near that tone. The audible frequency spectrum is divided into 24 of these critical bands based on the bark-scale given by [Zwicker and Terhardt, 1980]:

$$z = 13arctan(0.00076f) + 3.5arctan((f/7500)^2) \tag{A.1}$$

There is one "bark" between the center frequencies of these critical bands, a deeper description and a table of the critical bands can be found in [Fastl and Zwicker, 2007]. If one want to describe the first 5kHz of the audible spectrum it would require the first 18 critical bands or correspondingly the first 17 barks i.e. 18 bark-scale coefficients as exemplify in [Cheng et al., 2005]. Hence the bark-scale holds information on how we as human perceive the sounds we can hear. An alternative to the BFCC is the mel-scale based cepstal coefficients i.e. Mel Frequency Cepstral Coefficients (MFCC) which seemingly is a more popular perceptive frequency scale.

The cepstrum is described in detail in [Rabiner and Schafer, 1978], it can be used to gain insight on the pitch period and formants within a segment of a signal. The real cepstrum is defined as [Rabiner and Schafer, 1978]:

$$c(n) = \frac{1}{2\pi} \int_{-\pi}^{\pi} log|X(e^{j\omega})|e^{j\omega n} d\omega \tag{A.2}$$

That is the logarithmic inverse Fourier transformed of the Fourier transformed signal. It can be approximated with the Inverse Discrete Fourier Transform (IDFT) as [Rabiner and Schafer, 1978]:

$$c_p(n) = \frac{1}{N} \sum_{k=0}^{N-1} log|X_p(k)|e^{j\frac{2\pi}{N}kn} 0 \leq n \leq N-1 \tag{A.3}$$

Hence it would seem that taking the logarithmic of the bark-scaled input and applying the DCT, is close enough to the real cepstrum that people call this the bark-scale frequency cepstral coefficients or alternative the mel-scale frequency cepstral ceofficients.

It is these 18 BFCC that is used as input to LPC part of the network in order to produce the prediction coefficients.

In [Valin and Skoglund, 2019a] they describe how this is done by converting this bark-scale frequency cepstrum into a linear Power Spectral Density (PSD), of which they then apply the inverse Fourier transform to in order to obtain the autocorrelation, which is then used to obtain the prediction coefficients, as we have discussed in chapter 2.

It is not described how the [Valin and Skoglund, 2019a] convertes the bark-frequency cepstrum into a linear PSD, however we expect that it is similar to that of converting a MFCC to PSD which have been done in [Gang Min et al., 2015].

A.4 Single Instruction Multiple Data

Single Instruction Multiple Data (SIMD) are a part of Flynn's classification of computer architectures [Flynn, 1972]. It consists of the Single Instruction Single Data (SISD), Multiple Instruction Single Data (MISD), Single Instruction Multiple Data (SIMD) and Multiple instruction Multiple Data (MIMD).

In this project we primarily focus on the SIMD class of parallel computers. These types of computer architectures, are able to compute on multiple pieces of data at the same time, given that it is the same instruction is to be performed on the data pieces. An illustration of this is given in Figure A.6. Thus it requires that the data pieces, within the program or algorithm have to processed in the exact same manner, for this type of architecture to be effective. An example of an architecture which widely employs is try of processing are GPUs. A single GPU have hundreds if not thousands of Processing Unit (PU)s, which can perform operations of equally many pieces of data vectors or matrices.

The MIMD type of parallel processing is also faintly present in this project. The MIMD allows for the processing multiple pieces of data with multiple different instructions. The MIMD type of architectures can typically be found in multi-core CPUs and FPGAs. Hence these types of architectures are able to perform parallel data processing, on data which requires different instructions to be correctly processed. Thus if this architecture to be effective, it requires that the algorithm or program have data which is not dependent on each others results. Alternatively can a MIMD architecture process multiple programs or algorithms at the same time.

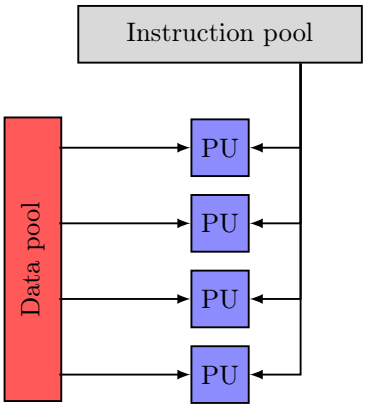
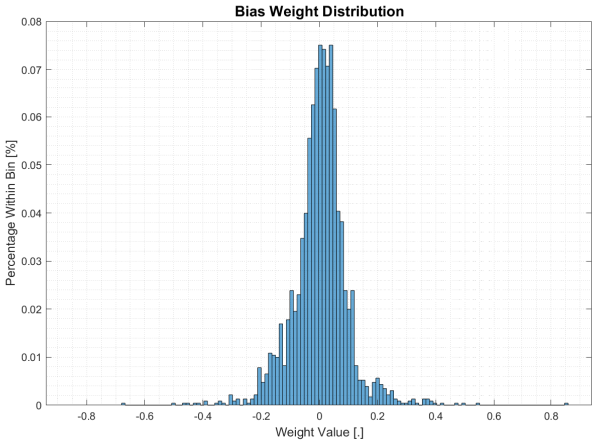
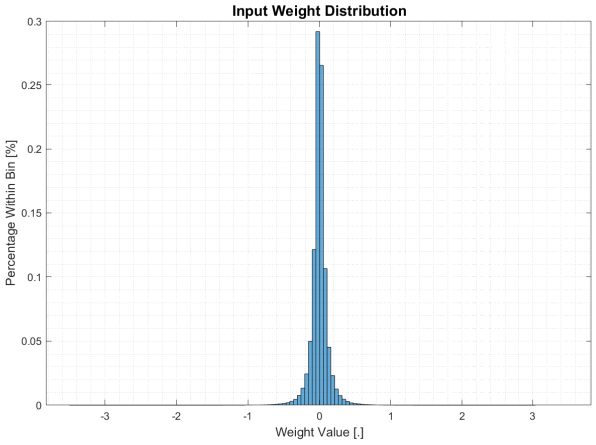


Figure A.6: Illustration of the SIMD, parallel processing scheme. PU denotes Processing Unit

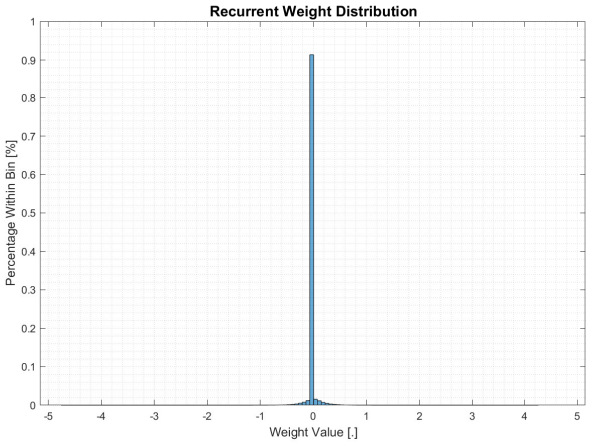
A.5 Weight distributions of some trained models



(a) Bias weights **b**

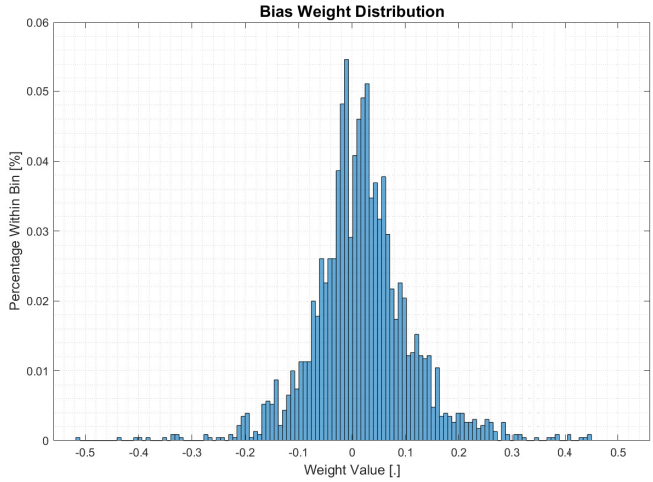


(b) Input weights **U**

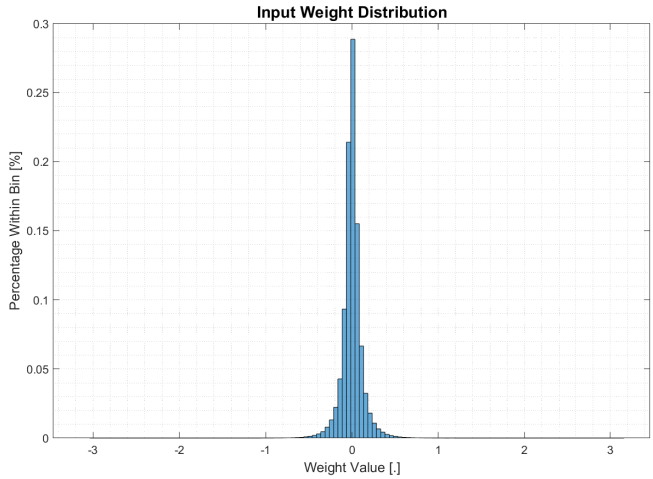


(c) Recurrent weigts **W**

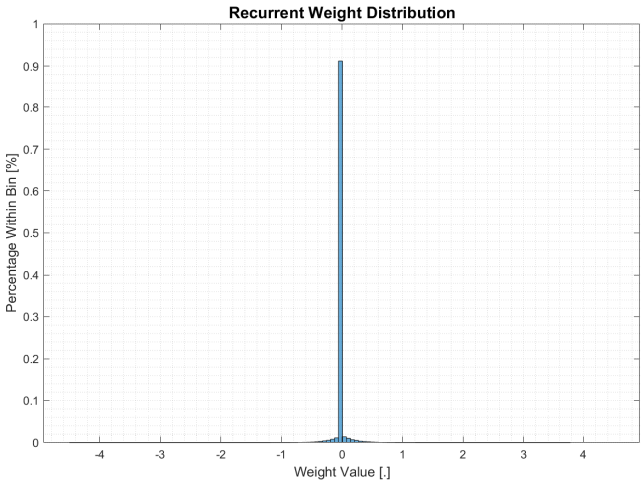
Figure A.7: Weight distribution of GRU_A's weight matrices, based on model 'van_BS64E120_L3d3'



(a) Bias weights **b**

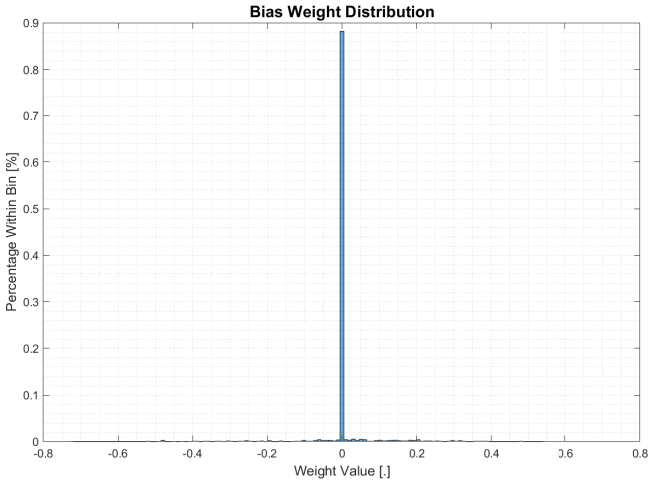


(b) Input weights **U**

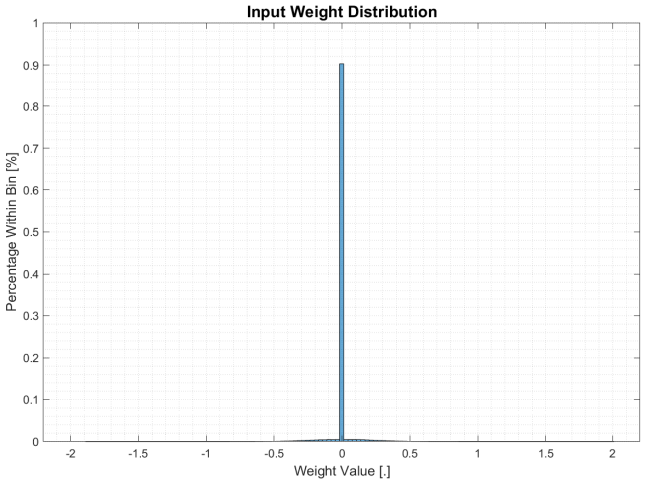


(c) Recurrent weigts **W**

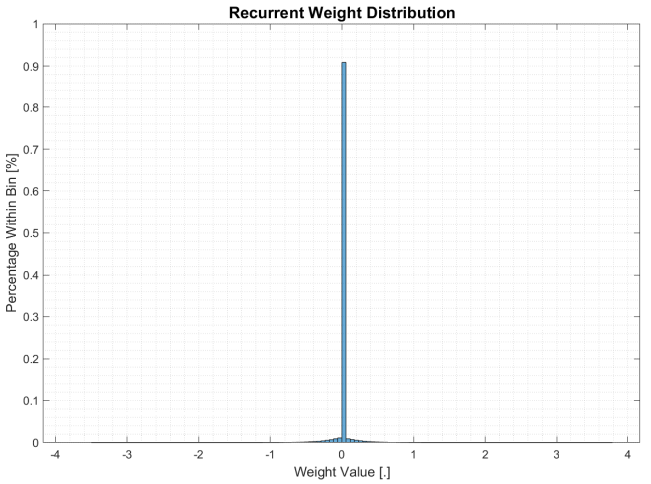
Figure A.8: Weight distribution of GRU_A's weight matrices, based on model 'lpcnet20c_384_10_G16_80' - found at https://jmvalin.ca/misc_stuff/lpcnet_models/ - Last visited 25 February 2020



(a) Bias weights **b**



(b) Input weights **U**



(c) Recurrent weigts **W**

Figure A.9: Weight distribution of GRU_A's weight matrices, based on model 'fixed_BS64E30_L3.32'