Anders Nilsen

Accelerating keyword spotting neural networks on FPGAs using Intel OpenVINO and Xilinx DNNDK

Master's thesis in Electronic Systems Design
Supervisor: Kjetil Svarstad
June 2019

Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Electronic Systems

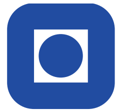
NTNU
Kunnskap for en bedre verden

Anders Nilsen

Accelerating keyword spotting neural networks on FPGAs using Intel OpenVINO and Xilinx DNNDK

Master’s thesis in Electronic Systems Design
Supervisor: Kjetil Svarstad
June 2019

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Electronic Systems



NTNU – Trondheim
Norwegian University of
Science and Technology

Accelerating keyword spotting neural networks on FPGAs using Intel OpenVINO and Xilinx DNNDK

Anders Nilsen

Master of Science in Electrical Engineering
Submission date: June 2019
Supervisor: Kjetil Svarstad, IET
Co-supervisor: Florian Bochud, Cisco Norway AS

Norwegian University of Science and Technology
Department of Electronic Systems

Project Assignment

Candidate name: Anders Nilsen
Assignment title: FPGA Implementation of the Google Speech recognition challenge

Assignment text

Background: Machine Learning has increased dramatically in popularity the last years and is now being used in various applications like web search, speech recognition, object detection, face recognition, etc.

Huge set of data are used to train a neural network (Learn), before the network can be used stand-alone to classify new patterns (inference).

Today's most common Machine Learning architecture is deep neural networks, which can be seen as layers of matrix multiplication. The network can have typically many layers with many weights (up to several 100k). Therefore, inference requires heavy processing resources, usually run on a GPU.

Both pre-processing of audio/video data and inference of neural networks is very well adapted for FPGA acceleration, due to its huge parallel-processing capabilities. This project will go through the step to achieve this accelerations.

In this project, the student will:
- Train a neural network for a speech recognition applications. Network like Convolutional neural network (CNN) or Long short term memory network (LSTM) should be trained with different parameters, ideally a in order to get a good classification of the data samples from the Google Speech recognition challenge: https://www.kaggle.com/c/tensorflow-speech-recognition-challenge/data
- Accelerate the network on a FPGA development kit (Altera or Xilinx)
- Implement the data pre-processing in the FPGA (sliding window FFT), using available IP and own coding. High-level-Synthesis (HLS) is recommended. Pre-processing: https://github.com/udacity/AIND-VUI-Capstone/blob/master/utils.py

- Implement a C++ application instantiating both the pre-processing acceleration and the neural network, preferable getting the input data from a laptop/desktop microphone.

To implement these networks, the student will use:

- Xilinx SDAccel and XFDNN (https://www.xilinx.com/applications/megatrends/machine-learning.html) on an Amazon AWS F1 cloud platform.
- Intel/Altera Deep Learning development toolkit (https://software.intel.com/en-us/computer-vision-sdk) running on an Intel development kit available in a NTNU server.

Assignment proposer/co-supervisor: Florian Bochud (flobochu@cisco.com)

Supervisor: Kjetil Svarstad (kjetil.svarstad@ntnu.no)

Abstract

Neural networks are used for almost all devices performing speech recognition, such as Amazon's Alexa and Apple's Siri. The networks have to process and understand large vocabularies while having a small of a processing time as possible. This requires substantial computational power, which is difficult to achieve in edge devices. The computational requirements have led to keyword spotting neural networks being implemented on these edge devices. These networks specialise in detecting a small vocabulary of words, establishing an uplink to a server if a target word is detected. Though small, the networks still require substantial amounts of computational power to perform keyword spotting in addition to the associated pre-processing.

Research is being made into accelerating these networks using Field-Programmable Gate Arrays (FPGAs), an attractive target due to their high performance-per-watt ratio and relatively small size. Traditionally, FPGAs have required specialised skill to be programmed. High-level Synthesis (HLS) tools aim to remove the need for these skills by generating Hardware Description Language (HDL) code from high-level languages such as C++. Additionally hardware manufacturers are developing toolkits to help developers accelerate neural networks on FPGAs.

This thesis examines the acceleration possibilities of using HLS for the acceleration of speech recognition pre-processing, as well as the FPGA-acceleration possibilities for Intel OpenVINO and Xilinx DNNDK for three types of neural networks: Convolutional Neural Network (CNN), Long Short-Term Memory (LSTM) and Attentive LSTM (att-LSTM).

Though pre-processing implementation was unsuccessful, it was estimated to be accelerated from 718.61 ms to 111.5 ms by running on an FPGA. Of the three networks, only the convolutional network was accelerated on an FPGA using OpenVINO, achieving a median classification time of 3.69 ms with 0.02% loss in classification accuracy on an Intel Arria 10 GX. Network acceleration using DNNDK was unsuccessful due to incompatibilities with non-image data, though initial testing shows promising possibilities for the Software Development Kit (SDK).

Keywords: *Neural networks*, *FPGA acceleration*, *Keyword spotting*, *High level synthesis*, *Speech recognition*

Sammendrag

Nevrale nettverk brukes i nesten alle enheter med talegjenkjenning, som blant annet Alexa fra Amazon og Siri fra Apple. Disse nettverkene må kunne klassifisere store ordforråd med så liten forsinkelse som mulig. Dette krever betydelige mengder prosesseringskraft, noe som er vanskelig å få til i såkalte "edge devices". Disse prosesseringskravene har ført til at man i stedet implementerer nevrale nettverk for søkeordsdeteksjon. De resulterende nettverkene har små ordforråd, og oppretter automatisk en forbindelse til en server dersom et av ordene oppdages. Selv om de er mindre krever disse nevrale nettverkene relativt store mengder prosesseringskraft for å utføre søkeordsdeteksjon i tillegg til den tilhørende forhåndsprosesseringen.

På grunn av høy ytelse-per-watt og relativt liten størrelse forskes det på hvordan man kan aksellerere nevrale nettverk på FPGAer. Tradisjonelt sett har FPGA-programmering krevd spesialisert kunnskap, og for å redusere kunnskapen som trengs prøver verktøy for høynivå-syntese å generere HDL-kode fra høynivå-programmeringsspråk som C++. I tillegg til FPGA-aksellerering av høynivå-kode lager maskinvareprodusenter verktøy for å gjøre det lettere å aksellerere nevrale nettverk på FPGA.

Denne oppgaven undersøker mulighetene for FPGA-aksellerering av forhåndsprosesseringen til nevrale nettverk for stemme- og nøkkelordsgjenkjenning, i tillegg til mulighetene for FPGA-aksellering av tre nevrale nettverkstopologier med Intel OpenVINO og Xilinx DNNDK: CNN, LSTM og att-LSTM.

Forhåndsprosesseringen ble ikke kjørt på FPGA, men ble estimert til å bli aksellerert fra en kjøretid på 718.61 ms til 111.5 ms. Av de tre nettverkene ble bare CNN-nettverket

aksellerert på FPGA med OpenVINO, og fikk en median-kjøretid på 3.69 ms med 0.02% tap i klassifiseringstreffsikkerhet på en Intel Arria 10 GX FPGA. Aksellerering av nevrale nettverk med DNNDK ga ingen resultater på grunn av begrensninger under nettverksoptimalisering, men sett bort fra dette virker DNNDK lovende for FPGA-aksellerering av nevrale nettverk.
Nøkkelord: *Nevrale nettverk*, *FPGA-aksellerering*, *Nøkkelorddeteksjon*, *Høynivå syntese*, *Stemmegjenkjenning*

Acknowledgements

I'd like to thank my supervisor at NTNU, Kjetil Svarstad, and my co-supervisor at Cisco, Florian Bochud, for their invaluable assistance throughout the thesis. I'd also like to thank Håkon Sandsmark at Cisco for helping me with problems related to neural networks and Adrian Sparrenborn and Graham McKenzie at Intel for technical assistance regarding OpenVINO. Finally, I'd like to thank my friends and family for supporting me throughout the studies.

Anders Nilsen
Trondheim, June 2019

Contents

1 Introduction **1**

1.1 Problem description 1
1.2 Own contributions 2
1.3 Method 3
1.4 Thesis structure 4

2 Background **5**

2.1 Machine learning 5
2.2 Speech recognition 6
2.3 Previous work 7
2.3.1 Google Speech Commands dataset 7
2.3.2 The Kaggle TensorFlow Speech Recognition Challenge 8
2.3.3 Kaggle Speech Challenge neural networks 8
2.3.4 Keras 10
2.3.5 Microsoft MMdnmn 10
2.3.6 Semester project 10
2.4 Neural Networks on FPGA 11
2.4.1 FPGA acceleration frameworks 12
2.5 High level synthesis 15
2.5.1 Vivado HLS 15

3 **Theoretical background** **17**
3.1 Neural networks 17
3.1.1 Neural network basics 17
3.1.2 Topologies 27
3.1.3 Neural network optimisations 28
3.2 High level synthesis 29
3.3 Speech recognition 30
3.3.1 Speech recognition fundamentals 30
3.3.2 Hidden Markov Models 30
3.4 Speech pre-processing 31
3.4.1 Fast Fourier Transform 31
3.4.2 Framing 33
3.4.3 Window function 35
3.4.4 Mel-frequency scaling 36
3.4.5 Mel-scale filterbank 37
3.4.6 Mel Frequency Cepstral Coefficients 37

4 **Implementation** **39**
4.1 Pre-processing 39
4.1.1 Testing using Xilinx FFT example code 39
4.1.2 Prototyping in C++ 41
4.1.3 Porting the code to Vivado HLS 44
4.1.4 Integrating the HLS IP with the Zynq Processing Unit 45
4.2 Neural network training 46
4.2.1 Deciding the network topology 46
4.2.2 Preparing the input data 47
4.2.3 Loading data into the network 48
4.2.4 Training the network 48
4.2.5 Verifying network functionality 49
4.3 FPGA acceleration 50
4.3.1 Intel OpenVINO 50

4.3.2 Xilinx DNNDK . . . 52

5 Results **55**
5.1 Pre-processing . . . 55
5.1.1 On Processing Unit . . . 55
5.1.2 On FPGA . . . 56
5.2 Neural networks . . . 58
5.2.1 Training results . . . 59
5.2.2 Verification results . . . 60
5.3 Neural network acceleration using OpenVINO . . . 64
5.3.1 On CPU . . . 64
5.3.2 On FPGA . . . 65

6 Discussion **73**
6.1 Pre-processing . . . 73
6.1.1 Comparing C++ and HLS . . . 73
6.1.2 Using Vivado HLS for FPGA acceleration . . . 75
6.2 Neural networks . . . 76
6.2.1 Accuracy evaluation . . . 76
6.2.2 Runtime comparison . . . 77
6.2.3 Using Keras for neural network development . . . 77
6.3 FPGA acceleration . . . 78
6.3.1 OpenVINO accuracy . . . 78
6.3.2 OpenVINO runtimes . . . 78
6.3.3 Using Xilinx DNNDK for FPGA acceleration . . . 79
6.3.4 Using OpenVINO for FPGA acceleration . . . 80
6.4 Error sources . . . 80
6.4.1 Classification accuracy . . . 80
6.4.2 Classification times . . . 81
6.5 Further work . . . 82
6.5.1 FPGA acceleration of pre-processing . . . 82
6.5.2 FPGA acceleration of the whole CNN using OpenVINO . . . 82

6.5.3 FPGA acceleration using DNNDK 82
6.5.4 FPGA acceleration of LSTM networks 83

7 Conclusion **85**

References **87**

A Inference Engine per-layer execution times **99**

B Precision and recall curves for the 12 classification words **101**

C Confusion matrices for FPGA-inferred CNN networks using OpenVINO115

List of Tables

3.1 List of common neural network activation functions 22

5.1 Execution times for each part of the C++ pre-processing code running on an Ultra96 56
5.2 Resource usage estimates for synthesised single-precision mel-log-filterbank energy pre-processing on Ultrascale+ ZU3EG A484 FPGA . 59
5.3 Execution time minimums, medians and averages for OpenVINO accelerated CNNs on different inference targets and precisions 66
5.4 Classification accuracies for OpenVINO accelerated CNNs on different inference targets and precisions 67

A.1 Execution times for each layer of the CNN on Central Processing Unit (CPU) and single-precision- and half-precision FPGA. Dashed entry means that the layer was not run on the inference target 100

List of Figures

2.1 Example of dilated convolutions, converting a 2D data matrix into a 3D data cube 9

2.2 Program flow for the Intel OpenVINO Inference Engine 14

2.3 Code example showing the use of pragmas in Vivado HLS 16

3.1 Perceptron model as described by Frank Rosenblatt [28]. The activation function is an adjustable threshold for deciding whether the result of the input function is 1 or 0 18

3.2 Types of artificial neural networks [30] 19

3.3 Convolutional neural network layer containing several layer types to form one deep convolution layer, starting at 2DConv 1 and ending at Dropout 1 21

3.4 Simple 2D-convolution of a 3x3 matrix with a 2x2 kernel producing a 2x2 output matrix 23

3.5 An LSTM layer for three discrete time steps [34] 23

3.6 Comparison of a Recurrent Neural Network (RNN) and Bidirectional Recurrent Neural Network (BRNN) layer functionality [36]. In an LSTM network, the LSTM layers are used for the squared circles of the figure 24

3.7 Hidden Markov Model (HMM) showing the probability of different activities occurring based on the current weather [49] 31

3.8 Discrete Fourier Transform (DFT) of the word "yes" with an Fast Fourier Transform (FFT) size of 16000 33
3.9 DFT with a length of 1024 for the word "yes" 34
3.10 Hamming window weighting with corresponding FFT [52] 35
3.11 Non-windowed and Hamming-windowed DFT frames for 1024 samples of the word "yes" 36
3.12 Mel-filterbank of 10 filters from 0 Hz to 8000 Hz [55] 37

4.1 Hot/cold Colormap-representation of log-mel-filterbank energies for the word "yes". Calculated using an FFT windows size of 1024, a shift length of 128 and 80 mel-filterbanks from 300 Hz to 8000 Hz 43
4.2 Settings struct used for the Xilinx FFT IP-core in Vivado HLS 44
4.3 Block diagram from Vivado IP Block Design tool for the communication between the FFT HLS IP block and the Zynq Processor System (PS) . 46
4.4 Layer execution times as reported by the Inference Engine 51
4.5 Execution times and throughputs reported at the end of Inference Engine-based prediction using OpenVINO 52
4.6 DECENT input images generated from log-mel-filterbank energies using the Pillow library for Python 53
4.7 Error message generated by DNNC upon kernel compilation 54

5.1 Comparison of power data calculated using 64-bit C++ and 16-bit HLS 57
5.2 Hot/cold Colormap-representation of log-mel-filterbank energies for the word "yes". Calculated using 16-bit HLS FFT with a window size of 1024, shift length of 128 and 80 mel-filterbanks from 300 Hz to 8000 Hz 58
5.3 Training and validation loss and accuracy for CNN 60
5.4 Training and validation loss and accuracy for LSTM network 61
5.5 Training and validation loss and accuracy for att-LSTM network . . . 62
5.6 Confusion matrix for CNN for 12 words with no thresholding 63
5.7 Precision and recall curve for CNN with thresholding from 1.00 to 0.00 with 0.05 decrements per step 64

5.8 Confusion matrix for Keyword Spotting (KWS) LSTM for 12 words with no thresholding . . . 65
5.9 Precision and recall curve for KWS LSTM network with thresholding from 1.00 to 0.00 with 0.05 decrements per step . . . 66
5.10 Confusion matrix for KWS att-LSTM network for 12 words with no thresholding . . . 67
5.11 Precision and recall curve for KWS att-LSTM network with thresholding from 1.00 to 0.00 with 0.05 decrements per step . . . 68
5.12 Confusion matrix for single-precision CPU-inferred KWS CNN . . . 69
5.13 Runtimes for single-precision CPU-inferred KWS CNN . . . 70
5.14 Execution times for different precision CPU/FPGA-deployed KWS CNNs using OpenVINO . . . 71

6.1 Classification times across 100 iterations with CPU and FPGA as inference targets using OpenVINO . . . 79

B.1 Precision and recall curve for the word "yes" for all networks . . . 102
B.2 Precision and recall curve for the word "no" for all networks . . . 103
B.3 Precision and recall curve for the word "up" for all networks . . . 104
B.4 Precision and recall curve for the word "down" for all networks . . . 105
B.5 Precision and recall curve for the word "left" for all networks . . . 106
B.6 Precision and recall curve for the word "right" for all networks . . . 107
B.7 Precision and recall curve for the word "on" for all networks . . . 108
B.8 Precision and recall curve for the word "off" for all networks . . . 109
B.9 Precision and recall curve for the word "stop" for all networks . . . 110
B.10 Precision and recall curve for the word "go" for all networks . . . 111
B.11 Precision and recall curve for the "silence" for all networks . . . 112
B.12 Precision and recall curve for the unknown word ("marvin)" for all networks . . . 113

C.1 Confusion matrix for single-precision KWS CNN inferred on half-precision bitstream . . . 116

C.2 Confusion matrix for single-precision KWS CNN inferred on 11-bit precision bitstream 117
C.3 Confusion matrix for half-precision KWS CNN inferred on half-precision bitstream 118
C.4 Confusion matrix for half-precision KWS CNN inferred on 11-bit precision bitstream 119

Chapter 1

Introduction

This chapter briefly explains the problem, the contributions made during the project and the report structure.

1.1 Problem description

Speech recognition is the process of recording, converting and interpreting spoken words and sentences. One device performing speech recognition is Amazon's Alexa, which takes user commands and responds based on these commands. Speech recognition requires large vocabularies on the receiving end, something which has traditionally been done using Hidden Markov Models (HMMs), though recently Deep Neural Networks (DNNs) have become the standard for such tasks. DNNs for large-vocabulary tasks can have several million parameters and computations, requiring large amounts of computational power, usually computed using powerful servers and computers. The power required makes these networks unsuitable for edge-deployment on devices such as Microcontroller Units (MCUs), small CPUs or FPGAs. To solve this, keyword detection networks are used, which specialise in recognising a subset of words. If the network detects one of the words in question, it establishes a link to a server which can classify the rest of the sentence and the words to come.

Keyword detection networks still require relatively large amounts of computational power when compared to traditional embedded systems such as smart-devices. To solve this, neural networks can be optimised and accelerated on FPGAs, exploiting the parallel computation capabilities of the device along with its low power consumption. This results in energy-efficient and accurate neural networks with relatively short classification times.

Previously, a keyword spotting neural network was accelerated on an Intel Arria 10 GX FPGA using OpenVINO as part of a semester project at NTNU in 2018. The network had been trained using the Caffe framework to detect the phrase "Hey Spark". A C++ program was developed to interface with OpenVINO's Inference Engine which also recorded live audio from a microphone and pre-processed the recording using a pre-compiled binary file from Cisco.

This thesis will focus on training and accelerating three different neural networks to perform KWS based on the Google Speech Commands dataset using Keras. The networks will then be accelerated on an FPGA using Intel OpenVINO or Xilinx DNNDK. The pre-processing required for the network will be programmed using C or C++ and also accelerated on an FPGA using Vivado HLS.

1.2 Own contributions

The following own contributions have been made to the thesis:

- A literary study of speech recognition neural networks in regards to history, theory and previous work. The previous work is mostly in regards to the Kaggle Speech Recognition challenge.

- A C++-implementation of the required pre-processing, converting .wav audio files into log-mel-filterbank energies.

- A synthesisable C++ implementation of the pre-processing, developed using Vivado HLS.

- A modified Vivado HLS example for performing pre-processing with 16-bit floating point precision on an FPGA and PS.
- Development of a Python program for training KWS neural networks using Keras.
- Development of a Python program for verifying the functionality of KWS neural networks using Keras.
- An updated C++-program for inferring an optimised neural network using Intel OpenVINO on an Intel Arria 10 GX or on a CPU. Also involves verification of the inferred network.
- An exploration into using Intel OpenVINO and Xilinx DNNDK for acceleration of KWS neural networks on FPGA.
- Recommendations for future works in regards to FPGA acceleration of KWS neural networks.

1.3 Method

To test the FPGA acceleration possibilities of Intel OpenVINO and Xilinx DNNDK, some neural networks have to be trained. This requires networks, either pre-designed or designed from scratch, and corresponding training data. Training data has to be created or generated for the purpose, and most likely require some pre-processing before being used by the network. This pre-processing requires theoretical knowledge in the field of signal processing.

After training data has been generated, the networks have to be trained and validated. Once trained, the networks can be accelerated on FPGAs using OpenVINO or DNNDK. This requires setting up the SDKs and programming the required programs using their respective Application Programming Interfaces (APIs), as well as preparing the networks as required. In addition, the pre-processing has to be accelerated on an FPGA using Vivado HLS. Using these tools and programs requires knowledge about how they work and how to use them.

To verify that the pre-processed data is correct, the generated training data, using the non-accelerated C++ code, is compared against the data generated by a known good implementation, and the FPGA-accelerated pre-processing is verified against the non-accelerated results. In the case of the network classifications, the classifications are verified by earmarking a part of the dataset for verification purposes prior to training.

The results are analysed and discussed to evaluate the performance of the SDKs, in terms of speedup, accuracy and usability, as well as the performance of the networks, to determine what network type is most suitable for FPGA acceleration and possibly deployment. If no direct result can be achieved, an estimate can be made if enough related results are available.

1.4 Thesis structure

This chapter, chapter 1, introduces the problem, the contributions made to the thesis, the method for solving the problem and the thesis structure. Chapter 2 presents some background information in regards to the history and concepts of machine learning and speech recognition, as well as some previous work in the field of speech recognition and keyword detection. Chapter 2 also presents some FPGA acceleration SDKs and HLS tools. In chapter 3, theory for neural networks, HLS, speech recognition and speech pre-processing is presented. Chapter 4 covers the development and implementation process of: Developing and accelerating speech recognition pre-processing code, using C++ and Vivado HLS; neural network training, using Keras; and FPGA acceleration, using Intel OpenVINO and Xilinx Deep Neural Network Development Kit (DNNDK). Next, chapter 5 presents the results of the training and acceleration, while chapter 6 discusses the results achieved. Chapter 7 presents the conclusion of the thesis.

Chapter 2

Background

This chapter presents a brief history of machine learning and speech recognition, as well as some related work in neural network speech recognition and FPGA acceleration. Additionally, this chapter presents some information on the Google Speech Commands dataset and the accompanying Kaggle Speech Challenge, some related works and examines some frameworks for FPGA acceleration of neural networks and HLS. Chapters 2.4.1.1 and 2.4.1.2 were initially written for the semester project.

2.1 Machine learning

Machine learning is a concept which dates back to 1959 when the term was coined by Arthur Samuel [1], though its origins can be traced back to Alan Turing [2] and his concept of the "learning computer". The concept revolves around using statistical techniques to make a computer "learn" an operation using sets of data instead of manually adjusting and writing the program to achieve the desired response. Initially, machine learning was restricted to relatively simple tasks such as playing checkers [3]. Though advances were being made, the field experienced periods of disinterest and reduced funding, caused mainly by unsatisfied expectations, notably in the Lighthill Report from 1973 [4]. Interest in machine learning remained dormant, increasing and

decreasing for small periods during the 1980s, until the 1990s when interest flourished. Development shifted from general-purpose A.I. to task-specific applications, leading to more clearly defined goals and less heightened expectations from the public. 2002 saw the release of Torch [5], a machine learning framework, to simplify the development of neural networks. Publicly available tools, lower cost of computational power and public interest saw the creation of other frameworks, such as free and open-source frameworks such as TensorFlow [6] and Caffe [7].

2.2 Speech recognition

The field of speech recognition can be traced back to Bell Labs' 1952 system "Audrey" [8], a primitive computer capable of recognising digits uttered by a single speaker. Early systems were limited to single-speaker, single word utterances and small vocabularies, but after researchers started using Hidden Markov Models (HMMs) for speech recognition in the 70's [9], vocabularies increased to the thousands, with IBM's 1984 typewriter system "Tangora" boasting a trainable vocabulary of 20.000 words [10]. Progress was limited by the cost of computers and storage space, but as technology improved, so did speech recognition. In 1992, AT&T started using speech recognition to route telephone calls [11] and for customer service [12]. As computational power increased and costs decreased, more advanced speech recognition systems were developed, mostly based on HMMs. The 2000s saw the rise of neural networks and were soon deployed in speech recognition. Early systems used HMMs alongside neural networks, but in recent years the focus has shifted to primarily using neural networks.

Today, speech recognition platforms have mostly shifted from HMMs to neural networks, using either Convolutional Neural Networks (CNNs) or Recurrent Neural Networks (RNNs). These models are often deployed on servers using Graphical Processing Units (GPUs) for acceleration, using edge-devices for detecting speech commands, also known as keyword spotting, to create an uplink when a keyword is detected.

While previously limited to specific devices, today, speech recognition is available on most computer platforms, from home computers to smart-phones. The most notable

example is Apple's Siri, recognising and answering questions from the user. Most major electronic brands have their own speech recognition system, such as Apple's Siri and Microsoft's Cortana. Speech recognition is still being used for customer service, voice-controlled applications and are a significant part of the "smart home" concept.

2.3 Previous work

2.3.1 Google Speech Commands dataset

The Google Speech Commands dataset is a dataset consisting of speech samples for up to 35 different words. Initially released in 2017, the dataset featured 30 different words, each word featuring 1600 to 4000 different variations [13], and has since been updated to include over 100.000 samples for 35 words. The dataset is targeted towards developers of edge-deployed Keyword Spotting (KWS) neural networks as a toolset to train and verify the functionality of the networks, while another target audience is hardware manufacturers. By having a common dataset, manufacturers can demonstrate their products by using networks trained with the toolkit, in addition to creating specialised hardware or optimisations to cater to KWS neural networks.

All recordings in the dataset have the same file properties: Recorded at 16 kHz with a bit-rate of 16 bits, all clips are 1 s long and stored as .wav-files. The recordings were selected based on specific criteria, such as intelligibility and loudness, initially using file size and average loudness levels to filter unsuitable samples, followed by manual verification to check that the pronounced word is the same as the label. The words in the dataset were chosen based on commonness, such as "yes" and "no", the number of phonemes, such as "Marvin" and "Sheila", or their similarity to other words in the dataset, such as "tree" and "three". The recordings were collected through a website where users could record themselves pronouncing words from the dataset. No specific guidelines were set, ensuring different recording qualities and background noises for the recorded samples, creating a more realistic dataset.

2.3.2 The Kaggle TensorFlow Speech Recognition Challenge

The Kaggle TensorFlow Speech Recognition Challenge was a neural network design competition issued by Google Brain in January 2018 on the Kaggle platform. The challenge consisted of designing and training a CNN using TensorFlow to recognise 12 words from the Google Speech Commands dataset, with a special extra competition which required the network to run on a Raspberry Pi 3 with specific requirements [14]. The Google Speech dataset consists of 30 different words, each recorded 2000 times resulting in a data set of 60.000 samples, in addition to 6 background noise samples. To reduce network size and scope, the Kaggle challenge selected 10 words from the dataset: *yes*, *no*, *up*, *down*, *left*, *right*, *on*, *off*, *stop*, *go*, *silence* and *unknown*. *Unknown* is one of the remaining words in the dataset, selected by the user, used to represent any word other than the ones listed, while *silence* represents no sound.

After designing and training the network, contestants could test their network on a test-dataset by uploading it to the Kaggle website and afterwards submitting it as an entry in the competition. The top three networks were able to achieve a classification accuracy of 91% using CNNs. Afterwards, the winning entries explained how they were able to achieve these results on the discussion pages for the challenge. The winning submission, achieving a classification accuracy of 91.06%, used log-mel-filterbanks as the input data to a CNN-based network, with several networks running simultaneously in ensemble-form to achieve higher accuracy.

2.3.3 Kaggle Speech Challenge neural networks

Though the techniques used by the winners were explained, the neural networks themselves were not released. Still, several papers have used the challenge and dataset as a metric for measuring the accuracy of their networks. McHahan et al. [15] used the dataset to evaluate pre-trained speech recognition models as well as "fresh" models, using two different input techniques. One set of models used regular input, a 2D-matrix of mel-filterbank data, and another using multiscaling. Multiscaling is the technique of performing several dilated convolutions on the same set of data. Dilation is a relatively new parameter in convolutions, introduced in 2015 by Yu et al. [16], in addition to

stride and kernel size, which increases the stride length by skipping spaces of data. Performing dilated convolutions 2D input data creates 3D output data, as illustrated by figure 2.1.

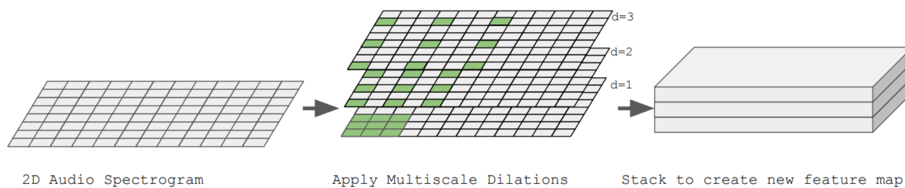


Figure 2.1: Example of dilated convolutions, converting a 2D data matrix into a 3D data cube

The networks using dilated convolutions achieved a classification accuracy of 82.22% and 85.52% for fresh and pre-trained, respectively, compared to 81.32% and 82.84% for non-multiscale networks for 20 words.

In 2018, de Andrade et al. [17] used the dataset to evaluate the classification accuracy of three networks: A CNN , an LSTM network and an Attentive LSTM (att-LSTM) network. The att-LSTM network used an attention model to detect parts of the data which were of interest, such as ignoring periods of silence and increasing attentiveness when speech information was present. Also illustrated by Bahdanau et al. [18] and Vaswani et al. [19], neural attention models increase performance on long sequence-to-sequence models, such as speech recognition. In addition to accuracy, de Andrade et al. [17] focused on developing network models with relatively few parameters, around 200K, to make the networks more suitable for edge deployment. Similar to McHahan et al. [15], the networks used 2D mel-filtered data as the input, though logarithmised. The final att-LSTM network had 202K trainable parameters, compared to 185K and 3060K for the LSTM network and CNN, respectively. The att-LSTM network achieved a classification accuracy of 94.5% for 20 words and 96.9% for 12 words, as used for the Kaggle Speech Challenge. The final network models were uploaded to GitHub under no specific license.[1]

[1] https://github.com/douglas125/SpeechCmdRecognition

2.3.4 Keras

Keras[2] is an open-source neural network library for Python. Originally released in March 2015, Keras aims at providing a high-level interface towards machine learning frameworks such as TensorFlow and MXNet and features GPU support along with the standard CPU support. Keras describes a neural network model as a sequential structure of layers, allowing for rapid prototyping and modularity.

2.3.5 Microsoft MMdnmn

MMdnn[3] is a toolkit developed by Microsoft to convert models between frameworks, i.e. convert TensorFlow-models to MXNet-models. As of 2019-06-03, the framework supports conversion between Caffe, Keras, TensorFlow, CNTK, MXNet, PyTorch, CoreML and ONNX. The toolkit converts between the frameworks by initially converting the model into an intermediate representation before converting it into the target framework. The toolkit also features a graph visualiser for visualising the neural network structure.

2.3.6 Semester project

During autumn 2018, a semester project was conducted examining FPGA acceleration of a KWS neural network using OpenVINO for Cisco [20]. The neural network was a binary classifier performing keyword spotting for the phrase "Hey Spark" on a 0.9 s input recording, either performed live using a microphone or a pre-recorded sample. The network was accelerated on a server at NTNU on an Intel Arria 10 GX Development Kit, achieving a classification time of 5.85 ms and 3.10 ms for CPU and FPGA, respectively. The network was inferred on the CPU/FPGA using a C++-program integrating the Inference Engine with the live audio recording or sample loading. The program was partially based on a master's thesis from Spring 2018, where the same network was accelerated using OpenCL and the Arria 10 GX Development Kit. The semester project C++ program used the same pre-processing, live recording technique

[2] https://keras.io/
[3] https://github.com/microsoft/MMdnn

and sample loading as the program from the master's thesis. The pre-processing was a pre-compiled binary file supplied by Cisco which generated Mel-Frequency Cepstrum Coefficient (MFCC) values for the input sample. The neural network was designed and trained using Caffe. The project report is available along with the associated code on GitHub[4].

2.4 Neural Networks on FPGA

FPGAs offer high performance per watt, making it a strong candidate for neural network computations and inference. Neural networks deployed on an FPGA can also be sped up when the inferred algorithm uses low numeric precision in calculations, e.g. using fixed point weighting and quantisation data instead of 32-bit floating point. These optimisations can provide substantial speedup while maintaining reasonable accuracy [21][22].

Due to this, FPGAs are a preferred platform for running inferred artificial neural networks. One of the main problems reducing the adoption rate is, and has been for many applications, how they are programmed. FPGAs are not programmed in the same way as MCUs or computer programs using a programming language such as C++ or C which is assembled into machine-level instructions. An FPGA is "programmed" by describing the functionality using a Hardware Description Language (HDL), such as VHDL or Verilog. The HDL code is then synthesised into a netlist which is mapped onto the FPGA. This way of programming differs from regular programming and increases the difficulty of writing effective and quality HDL-code, in most cases requiring specialised engineers. To reduce the difficulty of programming FPGAs, several tools exist to synthesise high-level programming languages, such as C, C++ and Python, into HDL code. This is called High-level Synthesis (HLS), and can be utilised in conjunction with Artificial Neural Networks (ANNs) to allow for inference of C++ code using OpenCV, TensorFlow, Caffe and other frameworks to FPGA without the need for the designer to write HDL-code.

[4]https://github.com/andernil/OpenVINO_project

2.4.1 FPGA acceleration frameworks

2.4.1.1 OpenCL

Open Computing Language (OpenCL) [23] is a platform heterogeneous framework for writing and running programs on several computing platforms, including CPUs, GPUs, FPGAs, Digital Signal Processors (DSPs) and other hardware accelerators. OpenCL was launched in 2009 by Apple to utilise the acceleration possibilities of on-board GPU. A collaborative group, the Khronos Compute Working Group, was created featuring representatives from several CPU, GPU, embedded-processing and software companies to maintain and improve the framework. As of this report, the newest version was 2.2, which incorporated more C++ features to the language.

The OpenCL framework is officially available for C and C++, but is unofficially available for Python, Java, Perl and .NET. An OpenCL implementation of a program is based around a host containing several compute devices, such as a CPU and a GPU, which is further divided into multiple processing elements. A function which is executed using OpenCL is called a kernel and can run in parallel on all processing elements. A programmer can utilise the acceleration capabilities available on a system by getting the device information from the computer the program is running on.

While OpenCL provides good possibilities for acceleration and resource usage, it is limited by its low-level nature. While it has functions for standard operations like FFT, neural networks have to be manually declared unless the frameworks used to generate the network have OpenCL-branches. Caffe has such a branch [24], but it is currently under development. TensorFlow has an OpenCL-branch on its roadmap. The lack of neural network framework support limits its adoption. A more supported and similar framework to OpenCL is Nvidia's CUDA, although this only runs on Nvidia GPUs.

2.4.1.2 Intel OpenVINO

The OpenVINO toolkit is Intel's solution for running neural networks on FPGAs, and aims to simplify the process compared to existing solutions. The OpenVINO toolkit was launched in 2018 by Intel, and allows users to program applications where neural networks can be accelerated on Intel processors, GPUs, FPGAs and Vision Processing

Units (VPUs) [25]. The toolkit is available for Windows 10, CentOS, Ubuntu and Yocto Project Poky Jethro, but compatibility with different inference targets varies between platforms. As of this report, FPGA acceleration with OpenVINO works on the Altera Arria 10 GX development kit and the Intel Vision Accelerator Design with Intel Arria 10, with a retail price of $4,495, while there is no publicly available retail price for the Vision Accelerator. These FPGAs have a PCI-Express connector which allow them to easily be integrated into a computer.

OpenVINO is mainly used for accelerating image recognition CNNs, but can be used for other purposes such as speech recognition. It supports frameworks such as Caffe and TensorFlow and deep learning architectures such as AlexNET and GoogleNET. It supports a set amount of layers for each framework out of the box, with custom layer support available for developers.

To use OpenVINO in a project, the neural network model is optimised using the model provided by the neural network framework, such as a .caffemodel (from Caffe), with the calculated weights with the Model Optimizer. The default model precision is single-precision floating point, while quantisation to half-precision floating point is available in the Optimizer. 8-bit integer quantisation is also available. The Optimizer provides an optimised intermediate representation which is loaded into the code using the Inference Engine API. The API prepares and infers the network to the target device and runs the network with the supplied input data. All pre- and post-processing is done in C++, so the only part which has to be replaced is the inference or prediction process. On FPGA, OpenVINO uses a pre-loaded bitstream programmed onto the FPGA to accelerate instructions. It does not utilise HLS, but uses the FPGA as a specialised processor for performing mathematical operations found in neural networks, such as convolutions and activations. The OpenVINO bitstreams utilise all available space on the FPGA, leaving no room for additional HDL code.

Figure 2.2 show the program flow when using the Inference Engine:

1. Load Plugin - The appropriate plugin for the deployment target is selected, such as FPGA and CPU, or for multiple targets, Hetero, which runs the network on the CPU if it is unsupported on the primary targeted device.

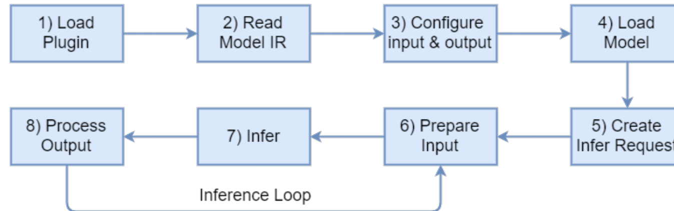


Figure 2.2: Program flow for the Intel OpenVINO Inference Engine

2. Read Model IR - The Intermediate Representation of the target neural network is loaded into the Inference Engine. If Hetero-execution is selected, the network layers are split between the target device and the CPU depending compatibility.

3. Configure input and output - The input and output accuracies are specified, as well as the input layout.

4. Load Model - The model is loaded from the IR into the system memory

5. Create Infer Request - An inference request is sent to the Inference Engine to ready for inference onto the target device(s).

6. Prepare input - The input data is loaded into the network.

7. Infer - The network is inferred onto the target device(s) along with the input data. The layers are executed in order and an output is generated.

8. Process Output - The output is loaded from the Inference Engine.

As illustrated by figure 2.2, it is recommended to repeat steps 6 through 8 when the network is used multiple times in one session.

Several network topologies are supported by OpenVINO, such as CNN and LSTM; however, LSTM support is limited to only the Kaldi-framework and partially for MXNet.

2.4.1.3 Xilinx DNNDK

To compete with OpenVINO, Xilinx acquired Chinese developer DeePhi in 2018 and their neural network FPGA acceleration SDK, the Deep Neural Network Development

Kit (DNNDK). The DNNDK SDK features model pruning, quantisation and deployment on Xilinx FPGA development kits such as the Xilinx ZCU102 ($2800), ZCU104 ($1000) and Avnet Ultra96 ($250) along with some of DeePhi's development kits. Along with FPGAs, the systems have embedded MCUs, on the Xilinx devices called Multi-Processor System-on-Chip (MPSoC), with FPGA as Programmable Logic and MCU as Processor System (PS). According to DeePhi, the SDK is capable of accelerating CNNs as well as RNNs, achieving a throughput speedup of 1.8x and 19x when compared to Application Specific Integrated Circuit (ASIC) and HLS-implementations of the same network, using 56x less power than the HLS implementation [26]. The SDK is divided into three main programs: DECENT, DNNC and the C++-API.

DeePhi's solution to running neural networks on an FPGA is to accelerate them using a soft-core processor, the Deep-learning Processor Unit (DPU). The DPU is designed to support and accelerate common neural network designs, such as AlexNET, SSD and SqueezeNet, as well as custom networks. In contrast to OpenVINO, the FPGA image does not occupy the whole FPGA, leaving space for custom HDL-code to run alongside the SDK.

DECENT performs quantisation on the weights and activations in the SDK, from 32-bit floating point to 8-bit signed integer precision, optimising mathematical functions and memory usage. DECENT also performs pruning, removing unnecessary connections and neurons.

DNNC splits the neural network into smaller kernels for either deployment on main parts of the MPSoC, the PS or FPGA/DPU, depending on the network type, which are then loaded and used using a C++-program to interface with the target devices.

2.5 High level synthesis

2.5.1 Vivado HLS

Vivado HLS is Xilinx' HLS tool for HDL synthesis of C, C++ and SystemC. Initially a paid upgrade to the standard Vivado package, Vivado HLS has since been included for free alongside the core Vivado package. The program allows for any of the supported

programming languages to be loaded into the program, simulated and synthesised into HDL and exported either as HDL or as a Vivado IP Core. The program has several features, such as specialised HLS libraries, IP cores such as FFT and FIR-filters, C-simulation and software/hardware co-simulation. Vivado HLS is integrated with other Xilinx programs, using Vivado XSim for HDL debugging and waveform analysis, in addition to Vivado's IP Block Design tool. Vivado HLS also features the use of pragma directives to perform optimisations to the synthesised code, such as loop unrolling, parallelisation and interface optimisation. Figure 2.3 shows an example of pragmas for specifying loop-unrolling and partitioning.

```
template<class TI, class TO, class TC, int SZ, win_fn_t FT, int UF>
void window_fn(TI *indata, TO *outdata)
{
   TC coeff_tab[SZ];
   init_coef_tab<TC,SZ,FT>(coeff_tab);
#pragma HLS ARRAY_PARTITION variable=coeff_tab cyclic factor=UF

apply_win_fn:
   for (unsigned i = 0; i < SZ; i++) {
#pragma HLS UNROLL factor=UF
#pragma HLS PIPELINE rewind
      outdata[i] = coeff_tab[i] * indata[i];
   }
}
```

Figure 2.3: Code example showing the use of pragmas in Vivado HLS

The code to be synthesised is defined by selecting a "top-function", the function from which all code within shall be synthesised. This requires all functionality of the program to be divided into subfunctions, and only one function can be selected as the top function. If several independent parts of the code are to be synthesised, the functions have to be synthesised separately.

Chapter 3

Theoretical background

This chapter explores some of the theory behind Artificial Neural Networks (ANNs), High-level Synthesis (HLS) and digital signal processing for speech recognition. Chapter 3.1.1 was originally written for the semester project.

3.1 Neural networks

3.1.1 Neural network basics

Machine learning uses Artificial Neural Networks (ANNs) to perform calculations and predict or classify an output based on a set of input data, attempting to emulate the functionality of the neural networks found in the human brain. Although computers are more efficient at performing mathematical computations and storing data, humans are better at learning tasks. Learning a new skill, recognising faces and voices and decision making are tasks which humans are better at, and which computers struggle with performing. These actions are, in humans, performed by a neural network in the brain. This neural network consists of approximately 10^{11} neurons which are interconnected into a decision tree [27]. The desire to mimic the human neural network motivated scientists to create a mathematical model of a neuron called a perceptron. A model of

the perceptron was described by Frank Rosenblatt in 1958.

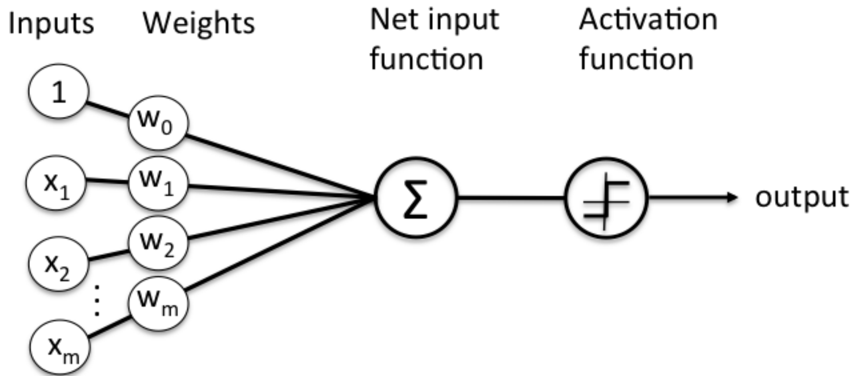


Figure 3.1: Perceptron model as described by Frank Rosenblatt [28]. The activation function is an adjustable threshold for deciding whether the result of the input function is 1 or 0

Figure 3.1 shows a block diagram of the model proposed by Rosenblatt. The diagram served as a basis for further computational models and was implemented in software by IBM in 1958 to perform image recognition, though limited to only one pattern. The limitation was due to the network having only one layer. The solution to this limitation was to implement a feed-forward, multi-layered neural network consisting of perceptrons, as suggested by Stephen Grossberg in 1973 [29]. This type of network is called a Deep Neural Network (DNN), while networks with a single hidden layer are traditionally called a Artificial Neural Network (ANN).

These new types of networks consist of several different types. The most common

network types are shown in figure 3.2. The network types can be divided into several categories: Feed-forward, recurrent, competitive and self-organising. Feed-forward networks, such as figure 3.2a and 3.2b, can be compared to combinatorial logic circuits. Recurrent networks, such as figure 3.2c and 3.2d, are more akin to sequential circuits with the recursive connections acting as a short-term memory. Competitive neural networks and self-organising networks, such as figure 3.2e and 3.2f, allow the networks to self-organise and build input feature maps, a feature used during network training.

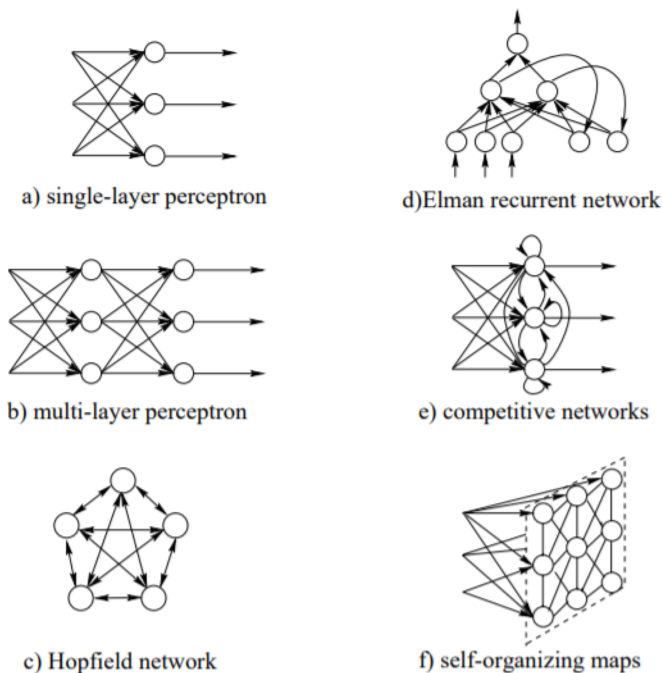


Figure 3.2: Types of artificial neural networks [30]

The neural network type to use depends on the task of the network. For image recognition, convolutional neural networks can be used, while for handwriting and speech recognition, recursive neural networks can be used. After selecting a network

type, the weighting of the connection between the nodes has to be calculated. This is performed by training the network.

3.1.1.1 Network design

Neural network design is a complex field, with several layer types to choose from to perform different functions. The only standard, go-to layers are the input and output layers. The input layer is usually some form of pre-processing, such as normalisation or re-ordering. The output layer represents the result of the network, often in one-hot encoding or integer values. Neural networks can be further divided into smaller "networks" for each layer, as they might contain convolutions, activations and poolings to perform a function, such as a deep convolution network layer as shown in figure 3.3. Several layers exist, such as:

Activation functions: Activation functions are used to perform gating and weighting on the input values of the function, adding a non-linear property to the neural networks. Several activation function types exist with different mathematical properties. Linear activation uses a linear scaling factor on the input value, while the step function is zero for negative values and one for positive values. Table 3.1 shows some common activation functions.

Pooling: Pooling layers are used to reduce the spatial dimension without reducing the depth of the data, e.g. if the data has the dimensions *H x W x Depth*, pooling reduces the dimensions to *H' x W' x Depth*. Reduction can be done in several ways, either downsampling using algorithms such as skipping or averaging, leading to several different variations of the pooling function. One such variation of pooling is max-pooling, where a window "slides" over an area of the input data and selects the highest value. Pooling reduces the amount of data in the neural network, increasing computation performance and decreasing the number of tunable parameters, further improving performance and reducing the risk of overfitting.

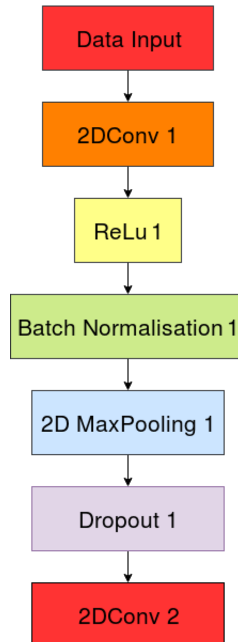


Figure 3.3: Convolutional neural network layer containing several layer types to form one deep convolution layer, starting at 2DConv 1 and ending at Dropout 1

Dense: Dense layers are fully connected layers, i.e. each input node is connected to each output node with adjustable weights for each connection. These layers are computationally heavy as they involve multiple Multiply–Accumulate Operations (MACs) for each connected node.

Dropout: Dropout is similar to dense, but will randomly "drop" a neuron, i.e. remove the neuron from the computation. Random neuron removals force the neural network to be more versatile and robust, reducing the chance of overfitting the neural network to a particular dataset.

Table 3.1: List of common neural network activation functions

Name	Function
Linear	$f(x) = ax$
Step	$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Sigmoid	$f(x) = \frac{1}{1+e^{-x}}$
Tanh	$f(x) = tanh(x) = \frac{2}{1+e^{-2x}} - 1$
ReLu	$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$

Convolution: Convolution layers perform convolution functions on the input data, sliding a fixed-size window across the 2D input data matrix, similar to pooling. The fixed-size window is called a kernel and contains adjustable weight values in the matrix which are multiplied with the input data. Mathematically, a convolution produces a function expressing how one function is modified by another. Figure 3.4 shows a simple 2D-convolution. Convolutional layers excel with matrix-shaped data, such as image data [31], but as a result, are computationally intensive with a large amount of MACs.

Batch normalisation: Batch normalisation is the process of normalising values between layers to reduce "internal covariate shift". Internal covariate shift is a by-product of adjusting weights during the training phase, where the input data of the next layer is affected to the prior layer, causing a varying offset during training. Batch normalisation deals with this by always normalising values before the next layer's input, resulting in increased learning rate and reducing the number of learning steps required [32].

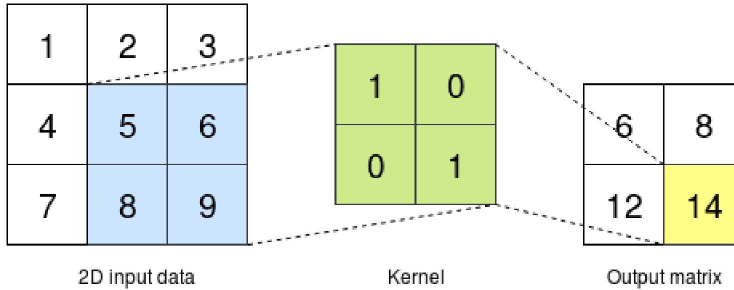


Figure 3.4: Simple 2D-convolution of a 3x3 matrix with a 2x2 kernel producing a 2x2 output matrix

Long short-term memory: Memory cells are used as temporary storage in neural networks, and can "remember" previous input values, i.e. in a video input, it can "remember" previous frames. This memory feature makes the networks recursive, which are classified as a RNN These form the building blocks for Long Short-Term Memory (LSTM) layers. LSTM layers consist of memory cells and activation functions combined in a single block [33], along with the ability to bypass the memory cells. Their depth, or memory length, can be defined by the task and requirements. Smaller memories have less trainable parameters and require less storage. LSTMs layers are one of the main components of recurrent neural networks.

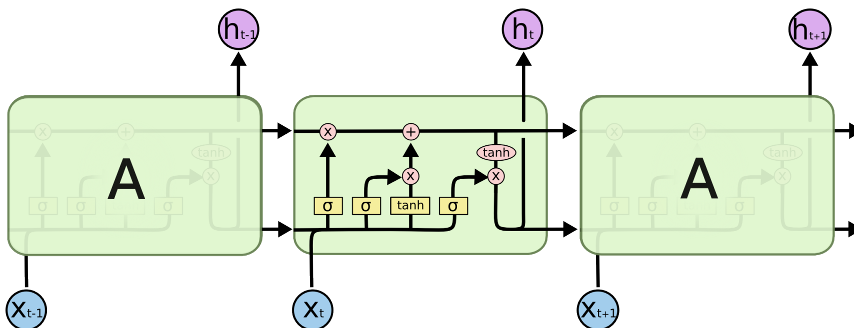


Figure 3.5: An LSTM layer for three discrete time steps [34]

Figure 3.5 shows an unrolled LSTM layer with three states. The key to the LSTM layer functionality is the top line shown in the figure, the cell state. It functions similarly to a conveyor belt, where each stage has the ability to modify the cell data passing through the stage. Whether to adjust or not is decided by the activation layers, illustrated as the four square boxes with sigmoid and tanh activation functions. These four activation layers are the trainable parameters of the LSTM block. These activation functions are based on the input data, X, which is multiplied with the cell state after the cell state has passed a static tanh activation function. The output, h, is passed to the next time the LSTM layer is called. The depth of the LSTM layer decides how many of the previous cell state and output values are kept in memory and used for the calculation of h.

LSTM layers can have an additional bidirectional property, allowing for future and past values to be used in computations [35]. This differs from traditional RNN and LSTM layers as they require future values to be delayed, shown in figure 3.6 comparing an RNN and a BRNN.

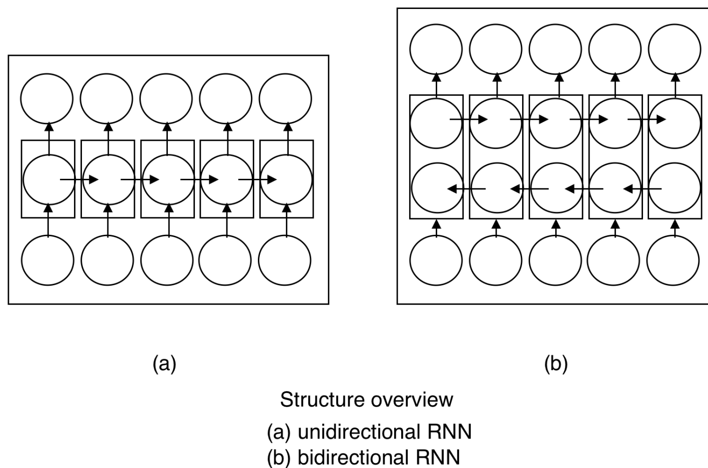


Figure 3.6: Comparison of a RNN and BRNN layer functionality [36]. In an LSTM network, the LSTM layers are used for the squared circles of the figure

3.1.1.2 Training

Neural network training can be performed in several ways but is generally performed by loading testing data, which has been manually classified, into the network and verifying the output. The prediction error is measured using a cost function, and the goal of training is to find the global minimum for this function. This is performed by tuning the weights of the network until this value is reached, or is as low as achievable. All weights in a layer can be expressed as a gradient, and the optimisation algorithms focus on tuning the values of the gradients for each layer to reduce loss. Gradient tuning can be done using several optimisation algorithms, such as gradient descent. Gradient descent is an optimisation algorithm which attempts to find the local minimum by finding the steepest descent for a function using backpropagation. Backpropagation is an algorithm which steps backwards through the different layers and gradients of the network. One cycle of sample input and weight adjustment is called an "epoch".

During training, callback functions can be used to perform specific actions after the end of an epoch. In theory, any function can be called, but these functions are usually reserved for training functions, such as:

- Model Checkpointer - Saves the neural network configuration with weights when a specified training metric improves.
- CSV Logger - Saves epoch results to a .csv-file.
- Early Stopping - Stops the training process if the specified training metric, such as validation accuracy, fails to improve for a specified number of epochs.
- Learning rate adjustment - Adjusts the learning rate if the specified training metric fails to improve for a specified number of epochs.
- Terminate on NaN - Terminates the training if the loss of the network is calculated to be NaN. Usually indicates an error in the training data.

Using callback functions helps reduce unnecessary time spent training, though some should be used with caution. Terminating training too early, due to a strict early

stopper, can result in a sub-par neural network as it is based on a local minimum.

Training a neural network is a resource intensive process, both in data and computational power. To properly train a neural network, training-, testing- and validation data is required, usually combined in a dataset. First, the dataset has to be related to the desired output of the data. To achieve good classification accuracy across different classification scenarios, the dataset has to be diverse and plentiful. Though the size of the dataset depends on the problem, research indicates that a data set of over 500 samples per class [37] is sufficient for some classification tasks, while others recommend datasets of 10 to 50 times the amount of weights in the network [38].

After finding or creating a varied and large dataset, the network can be trained. Training requires the calculation and adjustment of up to several million weights. Also, the dataset and the number of epochs required to achieve optimal loss can be high, leading to long training times. Therefore, training is often accelerated on GPUs, either locally or on servers.

3.1.1.3 Verification

To verify a trained neural network, verification data is loaded into the network, and the outputs analysed. The verification data is usually unused data from the dataset, split before the network training. The split-ratio can vary, but a common ratio is 60/30/10 for training, validation and verification, respectively. The output of the network is compared with the reference, and the results can be analysed in several ways, such as overall accuracy, precision and recall.

Overall accuracy is the most straightforward metric, illustrating how many of the predictions were correct. Overall accuracy, though, can be an inaccurate metric as many predictions might be to "large" labels, such as "unknown" words. Precision and recall is a more detailed metric, illustrating two similar aspects. Precision represents, for a given class, how many of the predicted labels for that category were correct, while recall represents, for a given label, how many of the possible labels were predicted. E.g., if class A was predicted 20 times out of a possible 40 times, but label A was used 80 times, class A would have a precision of 0.5 and a recall of 0.25. For multiclass predictions, such as the Kaggle challenge, the precision and recall of a model can be

found by calculating the average of the precision and recall for the individual words.

Precision and recall requires the definition of four classification categories: True positive, false positive, false negative and true negative. These can be explained using the following analogies: True positive, a pregnant woman is diagnosed as pregnant; false positive, a non-pregnant woman is diagnosed as pregnant; false negative, a pregnant woman is diagnosed as not pregnant; and true negative, a non-pregnant woman is diagnosed as not pregnant. These four categories are used to calculate precision based on equations 3.1 and 3.2.

$$Precision = \frac{tp}{tp + fp} \tag{3.1}$$

$$Recall = \frac{tp}{tp + fn} \tag{3.2}$$

Both of these metrics can be used to help increase the robustness of the network by serving as reference metrics when performing adjustments to the network, or when performing post-processing such as thresholding and weighting. By setting a threshold for prediction values, false positives can be reduced when the prediction value is low. Weighting can be performed on the prediction values to compensate for the uncertainty in the model's predictions.

3.1.2 Topologies

Though several topologies exist, the most common are Convolutional Neural Network (CNN) and Recurrent Neural Network (RNN).

3.1.2.1 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) mainly feature convolutional layers to perform classification and other machine learning tasks. As convolution layers work well with 2D-matrices, CNNs are commonly deployed for image and video recognition purposes; however they can serve as the basis for small and effective networks [39], CNNs usually have thousands to millions of parameters [40], making them susceptible to long training

times and high memory usage. Still, CNNs are the standard for image recognition, forming the backbone for popular neural network designs such as SqueezeNet [39] and YoloNet [41].

3.1.2.2 Recurrent Neural Networks

Recurrent Neural Networks (RNNs) are an alternative to CNNs, and instead of primarily using convolution layers, utilise memory cell layers, often in the form of an LSTM layer, to remember and sequence input data. The memory property enables the network to remember input sequential patterns, making it ideal for speech recognition, while having relatively few parameters [42] and of smaller sizes.

3.1.3 Neural network optimisations

After training, a neural network often has redundant layers and neurons which can be optimised away. Also, networks are often trained using floating point data types for weights and activations, which can be optimised to fixed-point representation. Together, these optimisations can reduce the size of the network and computational power required to run the network. These optimisations can be divided into two categories: Pruning and quantization. These functions are often run during the inference, the process of tuning and optimising the network to run on a target device, such as a phone or an edge device.

3.1.3.1 Pruning

Pruning is the process of removing unnecessary layers, neurons and connections between them to reduce network size and computational demand. The removal process can be done by using pruning algorithms which search through the network, removing unnecessary parts and connections [43]. Pruning might come with the cost of reduced network accuracy, but this is of negligible and in some cases none [44].

3.1.3.2 Quantisation

Quantisation converts weights and activation values from floating-point or fixed-point notations to smaller fixed-point notations and even binary values. Mathematical functions on computers and in processors are performed faster when the data width is decreased. Quantisation, therefore, reduces the computations required to calculate the output of a neural network.

3.1.3.3 Inference

Inference combines pruning and quantisation to optimise the network for deployment on a target device such as a phone or an edge device. The inference process can be done manually but is often performed using a neural network acceleration framework such as Intel OpenVINO.

3.2 High level synthesis

High-level Synthesis (HLS) is the term used to describe the process of converting and compiling high-level language code, such as C++ or SystemC, to Hardware Description Languages (HDLs) such as VHDL and Verilog. One of the main goals of HLS is to accelerate product development, allowing complex functions to be written and tested in C before being synthesised to HDL. HDL-programming has traditionally required specialised skill as the fundamentals are different from traditional object-oriented and functional programming. One of these fundamental differences is that HDL is a descriptive language, as the code is synthesised into logic gates and functions. Formal programming languages, on the other hand, are compiled into instructions for execution on a processor. In addition, HDL is executed in parallel on an FPGA while formal programming languages are sequential, though parallelisation is possible through the use of multi-core processors.

While HLS has been a concept for several decades [45], it has recently progressed to the point of being adopted and used in industry and academia [46], with languages such as SystemC and HLS-tools like Vivado HLS being popular choices.

3.3 Speech recognition

Speech recognition is usually performed on computers by sampling and converting a speech symbol from acoustic waves into uncompressed data, such as Pulse-Code Modulated (PCM) data, and then analysed in a neural network. Though raw PCM data can be used [47], the data is often processed in some form, such as filtering and Fourier-transformation, to reduce noise and extract the most relevant information.

3.3.1 Speech recognition fundamentals

Speech recognition is based on the analysis of speech signals, attempting to convert audio to text for further use in a computer program or system. It can be described as oral communication; the use of vowel and consonant sounds to generate words. As such, words, and the speech recognition used to detect them can be split into individual sounds. These individual sounds are called "phonemes", and one goal of speech recognition is to recognise the individual phonemes in a recorded phrase and assemble these into a word. This method, however, has proven challenging to utilise as phonemes can be hard to differentiate. The most common method is to divide the speech signal into smaller frames for analysis, mapping the analysed frames to vectors and inputting them into a neural network or an HMM.

3.3.2 Hidden Markov Models

A Hidden Markov Model (HMM) is a statistical system used to model sequences such as speech and handwriting by modelling them as hidden states connected by weighted connections. The weights describe the probability of the next member of the sequence. Figure 3.7 shows an example HMM for activities based on the current weather. HMMs were the standard for speech recognition until neural networks were adopted for the same purpose, though both have been used in combination to perform speech recognition [48].

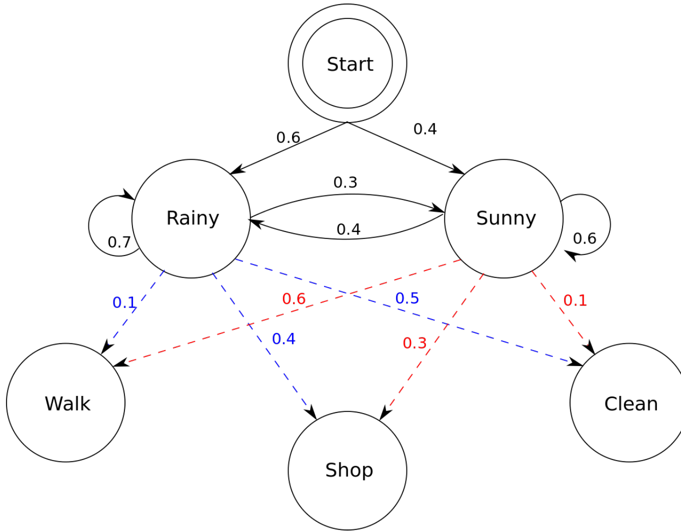


Figure 3.7: HMM showing the probability of different activities occurring based on the current weather [49]

3.4 Speech pre-processing

The speech signal is processed to reduce noise, remove unnecessary information and filter it for analysis. Pre-processing can be performed in several ways, but the most common method is to frame the signal into smaller frames, perform DFT and possibly calculate MFCC values.

3.4.1 Fast Fourier Transform

Though raw speech data can be used, the common way of analysing speech data is to perform a Fourier transform, usually using a Fast Fourier Transform (FFT) algorithm to perform a Discrete Fourier Transform (DFT). Performing a DFT on the speech signal produces a frequency representation which can be mapped to a particular word, such as figure 3.8 which shows the DFT of the word "yes". This FFT can be performed either on a CPU or an FPGA for higher throughput. Computing the FFT can be performed using

several different algorithms, such as the Cooley-Tukey algorithm with a complexity of $O(n \log n)$. The pseudocode for the Cooley-Tukey algorithm is shown in algorithm 1. The Cooley-Tukey algorithm is based on the fact that an FFT of composite size

Algorithm 1 Out-of-place Cooley-Tukey Radix-2 algorithm [50]

$X_{0,\ldots,N-1} \Leftarrow ditfft2(x, N, s)$: {x = data start, N = number of samples, s = stride}
if $N = 1$ **then**
  $X_0 \Leftarrow x_0$
else
  $X_{0,\ldots,N/2-1} \Leftarrow ditfft2(x, N/2, 2s)$
  $X_{N/2,\ldots,N-1} \Leftarrow ditfft2(x + s, N/2, 2s)$
  for $k = 0$ to $N/2 - 1$ **do**
    $t \Leftarrow X_k$
    $X_k \Leftarrow t + exp(-2\pi i\, k/N)\, X_{k+N/2}$
    $X_{k+N/2} \Leftarrow t - exp(-2\pi i\, k/N)\, X_{k+N/2}$
  end for
end if

$N = n_1 n_2$ can be expressed as smaller DFTs of size n_1 and n_2 where the output is transposed [50]. This makes it possible to calculate the DFT depth first. In general, the Cooley-Tukey algorithm calculates the DFT of a frame by first performing n_1 DFTs of size n_2, then multiplying the result with the twiddle factor, before finally performing n_2 DFTs of size n_1.

Due to the nature of the Fourier transform on real-valued data the resulting FFT has an output which is mirrored, i.e. the data in the first half of the output is equal to the second half, albeit reversed. This property helps reduce the input data width of the input data to a neural network or an HMM.

Performing a Fourier transform on a signal produces a complex result consisting of a frequency and a phase shift. In speech recognition, the power of the frequency is the interesting part. To compute the power, the absolute value of the complex result is calculated. Performing this on the whole signal produces a power spectrum of the signal.

An FFT performed on a short time signal, such as a framed and overlapping speech sample, is called a Short-Time Fourier Transform (STFT), as opposed to an FFT which

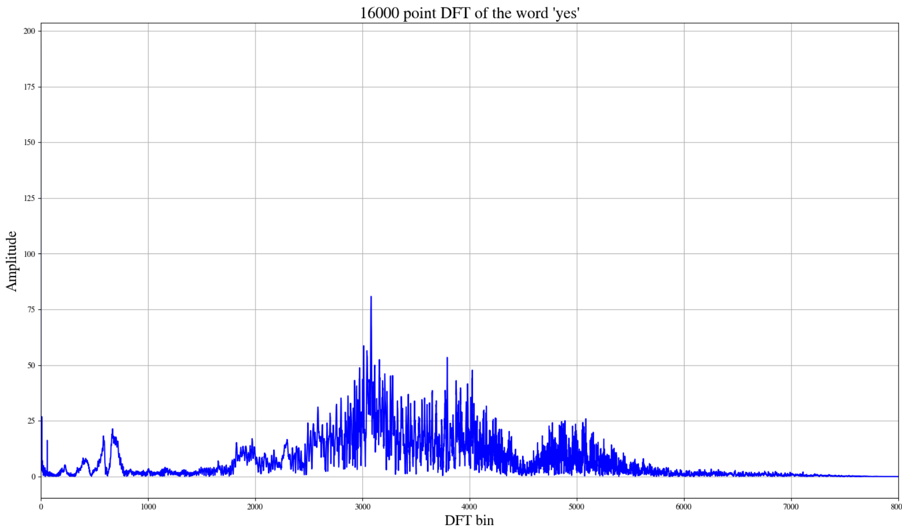


Figure 3.8: DFT of the word "yes" with an FFT size of 16000

is most often on a complete, non-overlapping signal. Performing Short-Time Fourier Transform (STFT) on a signal is inherently more resource-intensive when compared to an FFT, as the STFT of a signal requires an FFT to be calculated for each frame, but in return extracts more detailed frequency information from the input data. As the length of the DFT is decreases, so is the amount of "bins" for the FFT, i.e. the horizontal resolution of the transform is reduced. E.g. a DFT of 16000 compared to a DFT of 1024 represents a horizontal resolution decrease of 15.625.

3.4.2 Framing

Framing divides the signal into smaller pieces, usually with a duration between 10 ms to 20 ms. This interval is commonly used in speech recognition to capture rapid changes in the voice data which would otherwise be masked when analysing longer frames.

Framing a signal can also be seen as an extension of the concept of phonemes: Dividing a signal, or a word, into smaller pieces for further analysis and sequential mapping. By mapping the frames as a sequence, their relation can be extracted and used in the speech recognition process, either by a neural network or an HMM. Figure 3.9 shows the DFT of 1024 samples (64 ms of the word "yes". Note how the peak of the transform has shifted from around 3000 Hz in figure 3.8 to around bin 210, equivalent to 3300 Hz, while the upper frequencies are almost flat.

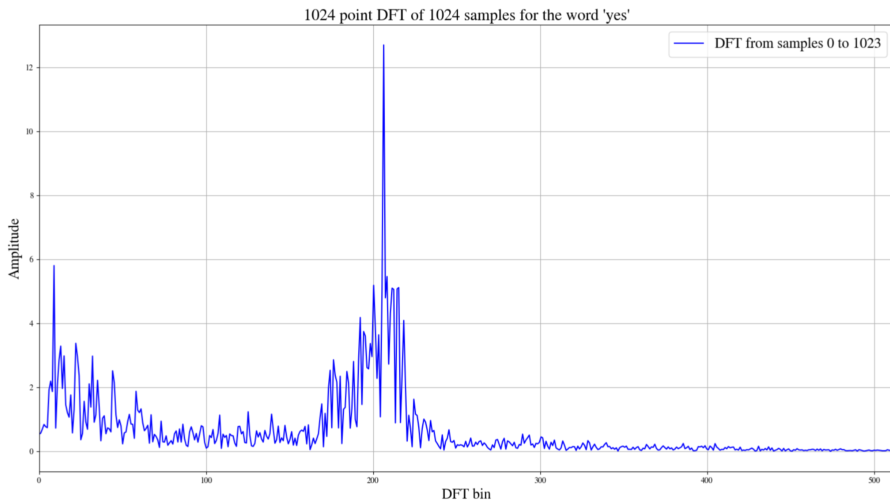


Figure 3.9: DFT with a length of 1024 for the word "yes"

Additionally, framing can be performed by overlapping frames on top of each other, commonly performed by shifting the framing window N samples for a given number of iterations. This technique is called a sliding window and results in a Short-Time Fourier Transform (STFT).

3.4.3 Window function

After framing, a windowing function is applied to the framed data to weight the samples in the middle of the window, reducing the effect of the side-lobes generated at the start and end of each FFT frame. The most common weighting function for speech recognition is the Hamming Window, given in equation (3.3).

$$w[n] = 0.54 - 0.46 \cdot \cos\frac{2\pi n}{N}, \quad 0 < n < N \tag{3.3}$$

The Hamming Window is a special case of the Hann-function which weights the data in the middle of the frame while reducing the flank data, as seen in figure 3.10. This, together with the sliding-window technique, produces a smoothly varying sequence of detailed spectral data [51].

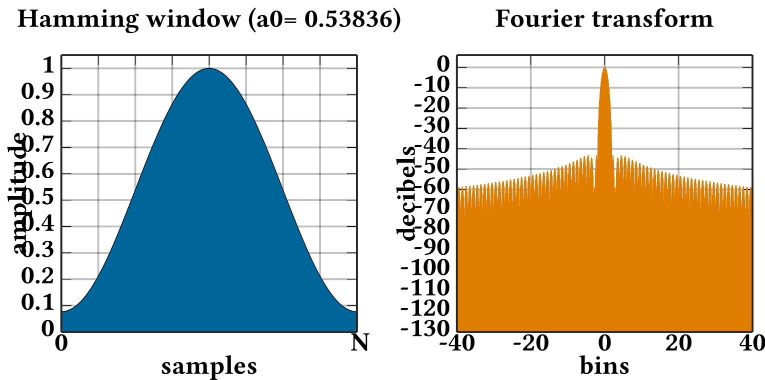


Figure 3.10: Hamming window weighting with corresponding FFT [52]

Figures 3.11 shows two overlapping DFT frames with and without Hamming-windowing. Note how the windowed DFT lowers the peaks of the information between bins 0 and 50, indicating that the input data causing the peaks is located in an overlapping area. As such, windowing produces a more realistic view of the frequency information stored in the input data.

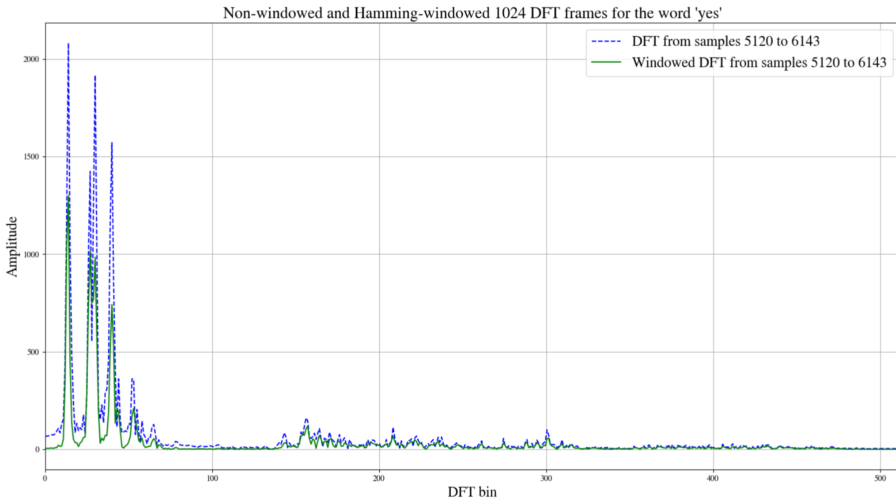


Figure 3.11: Non-windowed and Hamming-windowed DFT frames for 1024 samples of the word "yes"

3.4.4 Mel-frequency scaling

To further weight the signal in terms of perceived loudness, mel-frequency scaling is applied to the power spectrum calculated by the STFT. Mel-frequency scaling is based on the mel scale as devised by Stevens et al. in 1937 [53] and further improved by Makhoul and Cosell in 1976 [54]. The scale was created by playing individual pitches to test subjects, increasing the pitch until the test subject perceives the pitches to have the same distance between them. By setting 1000 Hz to equal 1000 mels, (3.4) can be used to convert from frequency (in hertz) to mels.

$$m(f) = 2595 \log(1 + \frac{f}{700}) \tag{3.4}$$

3.4.5 Mel-scale filterbank

A filterbank is a series of bandpass-filter and can be used to express the area of a frequency signal as a single, accumulated value. Combined with mel-scaling, this can create mel-filterbanks consisting of a varying number of mel-filters. Each mel-filter has a triangular shape when viewed on a logarithmic scale, reaching a peak at the filter's specified frequency, e.g. a mel-filter at frequency f starts at 0 at $f-1$, increases up to 1 at f and decreases to 0 again at $f+1$. The filters are equally distributed in the mel-frequency range before being transformed back to regular frequencies, with the filters having the same length as the result of the FFT. Figure 3.12 shows a 10-filter mel-filterbank from 0 Hz to 8000 Hz.

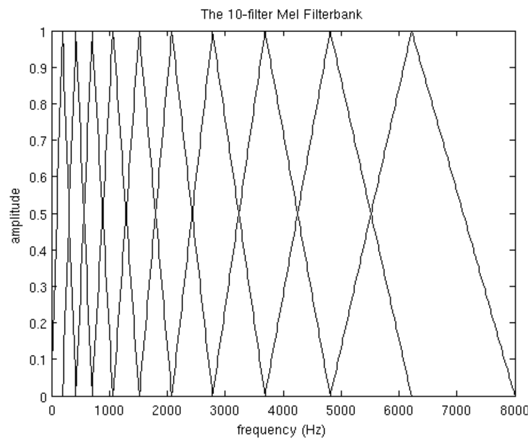


Figure 3.12: Mel-filterbank of 10 filters from 0 Hz to 8000 Hz [55]

3.4.6 Mel Frequency Cepstral Coefficients

After applying mel-frequency scaling to the power data, Mel-Frequency Cepstrum Coefficient (MFCC) can be calculated. MFCC values describe the mel-frequency cepstrum, represent the time based power spectrum of a signal. MFCC values are calculated by taking the logarithm of the mel-weighted power data, to better resemble the signals

perceived by the human ear, and then performing a Discrete Cosine Transform (DCT) to reduce the impact of low-energy components. After calculating the MFCC values, only the first half of the values are used as the upper half represent high-frequency data, which is not used for most speech recognition purposes.

Chapter 4

Implementation

This chapter covers the process of programming the pre-processing in C++, using Vivado HLS to synthesise the design, and Vivado to run the pre-processing on an FPGA. This chapter also covers the training of the three neural networks and the process of deploying them on an FPGA. The code for the pre-processing, both C++ and HLS implementations, is available on GitHub[1] along with the code for the neural network training and verification. The code for the OpenVINO inference is available on on GitHub[2] aswell.

4.1 Pre-processing

4.1.1 Testing using Xilinx FFT example code

To test the design flow of the HLS Intellectual Property (IP), Vivado IP Block Design tool and Vivado SDK, an FFT example was modified and run on an Ultra96 development kit with a Xilinx Zynq UltraScale+ MPSoC ZU3EG A484 FPGA. The example, found in Xilinx User Guide 871 - HLS Tutorials [56], described the process of synthesising C-code using Vivado HLS, importing it and connecting it with the Zynq Processor System

[1] https://github.com/andernil/lstm_speech_recognition
[2] https://github.com/andernil/OpenVINO_project/tree/master-thesis/

(PS) using the Vivado IP Block Design tool and finally setting up communication with the PS and performing tests using Vivado SDK.

The C-code was an AXI streaming interface to and from the Vivado FFT IP core. The two main parts of the code, which were synthesised, were the input, which performed windowing and framing, and the output, which re-ordered the output from the FFT IP and transmitted it back to the Zynq PS. The frame shift was 512 samples, and the data had an input width of 16 bits in the ap_fixed-format with 1 bit for integers and the remaining 15 for fractional values.

Initially, the project files were imported into Vivado HLS and simulated using C-simulation. This simulation passed, but synthesis failed due to errors with the memory interfaces. The example code had not been updated to fit with newer updates, resulting in the AXI streaming interface having double declarations for the data_pack pragma. This was fixed by removing the data_pack pragma, as it was already included in the AXI pragma. Fixing this made the two functions, real2xfft and xfft2real, synthesise without error and successfully pass software/hardware co-simulation. Afterwards, the code was edited to be more general purpose: The framing was removed, instead using all the input data as opposed to storing some of the input data for further use in the framing process, shifting the framing process from the FPGA to the CPU. After synthesising the input and output functions and exporting them as IP blocks, the IP cores and the Zynq PS were connected using the Vivado IP Block Design tool according to the tutorial.

Continuing the tutorial, a "Hello World"-example was instantiated on the PS and uploaded to the Ultrascale+ FPGA to test whether the communication was functioning. When attempting to connect to the JTAG connected to the Ultra96, Vivado SDK was unable to detect it. To fix this, specifically on Linux, the user running Vivado SDK had to be added to the user group "dialout". After doing this, the Ultra96 was successfully programmed with the example.

After checking that communication with the target device functioned, the tutorial was continued and edited to fit with the modified HLS code. The test data was changed to a sample from the Google Speech dataset, and the input length was edited to 1024 samples per transfer. The execution time was measured using the XTime_GetTime-

function.

4.1.2 Prototyping in C++

As the chosen HLS tool, Vivado HLS supported C and C++ as input languages; the pre-processing was initially programmed in C++ using as few external libraries as possible to avoid dependency issues when porting the code to Vivado HLS. The data type used for the program was double-precision floating point to ensure accurate reference and training data. Using information from guides for pre-processing of audio signals online [55][57], the pre-processing program was divided into five sections:

Reading the input data: The input files were stored in the .wav-format. To read these files, the header of the file had to be decoded to determine the size of the file, which had a fixed format consisting of a mixture of little- and big-endian format values. The header information required for extracting the audio data was the number of channels, sample rate, byte rate, data size and the data field, which were all little-endian. The samples from the Kaggle Speech Challenge had a sample rate of 16 kHz, a byte rate of 16 bit, 1 channel and were 1 s. Based on these specifications, a function was written which filled a pre-allocated 16000-field short int array and the input file-name and returned the audio data and the peak value of the audio data.

Windowing and scaling: The samples had varying amplitude depending on the recording levels used. To remove this variation, the input data was normalised to ±1 before being used further in the pre-processing. The windowing shifted the FFT along the sample data, performed in a for-loop by incrementing the start position of the input data until the end of the data was reached. The number of frames of FFT-data was be calculated based on the length of the input data, the FFT length and the frame step length, i.e. how many samples the window shall shift for each iteration, given in (4.1).

$$N_{frames} = \frac{N_{samples} - len_{DFT}}{N_{shift} + 1} \tag{4.1}$$

In addition to the sliding DFT window, a windowing function was applied for each frame. The selected windowing function was the Hamming window, which was calculated in advance prior to the FFT-framing and then applied to each frame. The framed data was input into the FFT function.

Computing the FFT: The FFT method implemented in the prototype was the Cooley-Tukey algorithm. This was chosen for its simplicity and speed, and was implemented as a recursive function in two parts: FFT and Separate. Initially, the input data was separated into two halves: Even number elements to the lower half and odd number elements to the upper half of the input data array. This was performed recursively until the input array area only contained one sample. Afterwards, the DFT was calculated using the FFT function. After the DFT was computed, the amplitude of each element was calculated. Only the first half of the output data from the FFT was used due to the mirrored nature of the FFT. The amplitude was found by calculating the absolute value of the FFT based on the real and imaginary parts of the FFT.

Generating the mel-filterbanks: To transform the power data into mel-filterbank energies, a mel-filterbank was required based on a user-definable amount of mel-filters and specified upper and lower frequency bounds. The filters in the filterbank were calculated by initially calculating the upper and lower mel-frequency bounds based on the specified frequencies. Next, the difference between each filter frequency was calculated and converted back to regular frequencies. Then, filters were created with a length equal to the output of the FFT. The resulting filter arrays were then filled with values rising to and sinking from 1 at the peak position, generating triangular filters according to the ones described in chapter 3.4.5.

Calculating the mel-scaled filterbanks for each frame: After generating the mel-filterbanks, each filter was multiplied with each frame of the FFT and the total power in the frame was summed, generating mel-filterbank energies for each FFT frame. After the mel-filterbanks were calculated, the logarithm of each sum was calculated along with the average logarithm of all the filterbank-energies. To improve

the Signal-to-Noise ratio, the average logarithm was subtracted from the filterbank-energies, balancing the signal across the mean. This also helped with the case of silence or constant sound, as the balancing would result in 0 as the filterbank-energy for every filterbank. Figure 4.1 shows the resulting mean-normalized log-mel-filterbanks for the word "yes".

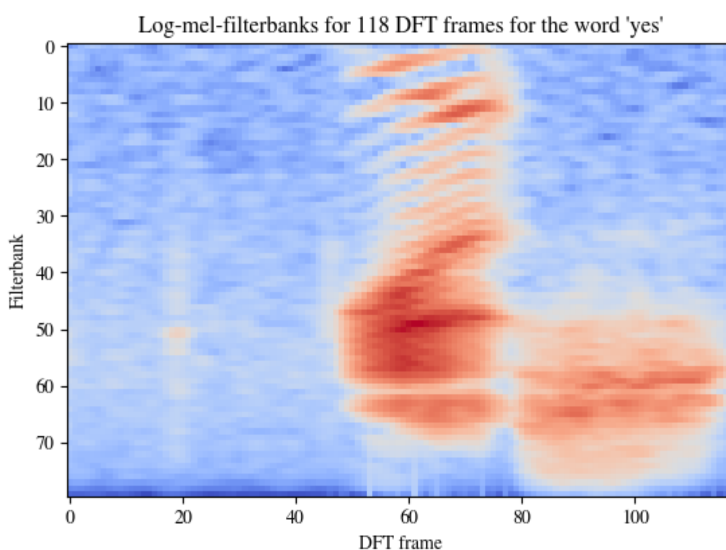


Figure 4.1: Hot/cold Colormap-representation of log-mel-filterbank energies for the word "yes". Calculated using an FFT windows size of 1024, a shift length of 128 and 80 mel-filterbanks from 300 Hz to 8000 Hz

The final code was tested using a sample from the Google Speech dataset of the word "yes" and compared with a Python-implementation[3]. The Python code had to be modified slightly, removing pre-emphasising and matching parameters such as upper and lower frequency bounds.

[3]https://github.com/jameslyons/python_speech_features

4.1.3 Porting the code to Vivado HLS

To run the pre-processing code on an FPGA, the code had to be synthesised into HDL using Vivado HLS. The code was added to a Vivado HLS project, and the initial build attempt resulted in several errors. The original pre-processing code used the standard C math-library, while the HLS code required the use of Vivado HLS' math-library. Replacing the library made the code build properly, but upon running C-simulation, several errors arose: Dynamic arrays and recursive functions were not allowed by the synthesis tool. Dynamic arrays were fixed by defining constant variables and reducing memory allocation functions, but as the FFT function, the Cooley-Tukey algorithm, was recursive, the function had to either be unrolled or replaced. Unrolling proved unsuccessful, as initial investigation showed that the function would have to be unrolled for the length of the FFT, i.e. for a 1024 length FFT the function would have to be unrolled 1024 times. Instead, the function was replaced by the FFT IP-core from Xilinx. Based on documentation from Xilinx [58][59], the FFT core was set up using an FFT-length of 1024 with 32-bit fixed point values using the Radix-2 FFT algorithm based on the Cooley-Tukey algorithm, with the output in natural order, i.e. FIFO, selected using the struct shown in figure 4.2. Several FFT configurations are available, but Radix-2 was chosen due to having a relatively small logic footprint and reasonable speed.

```
#define FFT_INPUT_WIDTH 32
#define FFT_OUTPUT_WIDTH FFT_INPUT_WIDTH

struct FFT_params : hls::ip_fft::params_t {
    static const unsigned ordering_opt = hls::ip_fft::natural_order;
    static const unsigned input_width = FFT_INPUT_WIDTH;
    static const unsigned output_width = FFT_OUTPUT_WIDTH;
    static const unsigned config_width = 24;
    static const unsigned arch_opt = 2;
    static const unsigned max_nfft = 10;
};
```

Figure 4.2: Settings struct used for the Xilinx FFT IP-core in Vivado HLS

To test the pre-processing program, a test bench was created to verify the functionality of the pre-processing code. The test-bench loaded .wav-data from a text file along with a text file containing the results of the FFT from the original C++ pre-processing

program. The results, referred to as the "golden reference", was compared with the output of the HLS code. The accuracy for the results was calculated along with the total accuracy for the whole dataset.

Initially, the results generated by the IP-core FFT differed substantially from the golden reference. The cause of this was scaling issues in the FFT core. The input and output data to the FFT was 32 bit fixed point with 1 bit for whole numbers and 31 bits for the decimal values, as the FFT IP core required data to be within ±1. This caused incorrect values to be output when the results of the FFT were equal to or greater than 2. To fix this, the input data was scaled down by right-shifting the value until no overflows occurred, which in the case of the test data was a 9 right-shifts or a division by 512. Performing right-shifts before the FFT and left-shifts afterwards, the output data matched the golden reference.

After C-simulation achieved sufficient accuracy, synthesis was performed with a target clock speed of 100 MHz. Initially, the top function was the window FFT-function, but this synthesis failed due to interface errors. This was fixed by placing the pragmas required by the HLS tool inside the window FFT-function instead of in the FFT-function. This allowed the code to synthesise correctly, but upon running the software/hardware co-simulation, the simulation got stuck, most likely indicating an error in the design. Instead, the FFT-function was selected as the top function, and the pragmas moved there. This code synthesised and simulated properly, and was exported as an IP core for further use in the Vivado IP Block Design tool.

4.1.4 Integrating the HLS IP with the Zynq Processing Unit

To easier facilitate communication with the Zynq PS, the interfaces were changed to AXI-streaming interfaces to communicate with the PS, similar to the example from the tutorial used in chapter 4.1.1. To set up communication with the PS, the setup and steps from this tutorial were used for the pre-processing HLS IP, though the width of the AXI interface was changed to 64 bit to accommodate for the data size of 32 bit. The completed block design, shown in figure 4.3, was generated by the IP Block Design Tool.

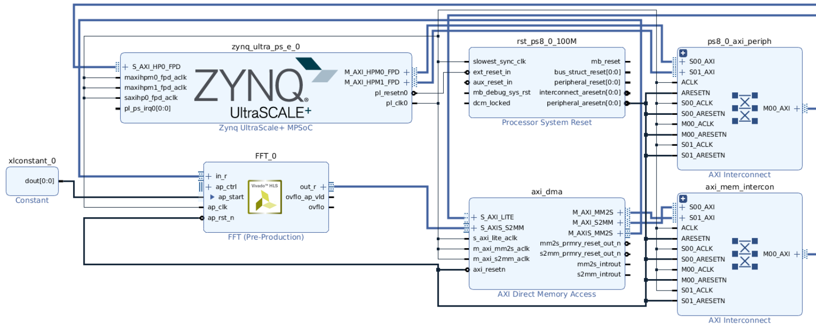


Figure 4.3: Block diagram from Vivado IP Block Design tool for the communication between the FFT HLS IP block and the Zynq PS

After passing design verification, an HDL interface was created and the code was synthesised and exported to Vivado SDK. Following the tutorial for the FFT example, the "Hello World"-program was programmed onto the FPGA and run successfully. Afterwards, the same C-code from the tutorial was used for interfacing between the CPU and FPGA.

4.2 Neural network training

4.2.1 Deciding the network topology

Designing a neural network from scratch is complex and time-consuming, so to save time it was decided that an existing neural network was to be used. The three networks from de Andrade et al. [17], as mentioned in chapter 2.3.2, were selected due to being available on GitHub and having achieved good classification accuracies. In addition to this, the three networks would serve as an indication on the compatibility of the FPGA acceleration SDKs in terms of layers and topologies. As the models were designed in

Keras, it was used as the framework for the training and verification process.

4.2.2 Preparing the input data

As the data from the Google Speech Commands dataset were stored as .wav, they had to be converted into log-mel-filterbank energeis before being input into the neural networks. Pre-processing parameters were selected according to de Andrade et al. [17]: The input .wav-data was framed into frames of 1024 samples with a frame shift of 128 bits, with Hamming-windowing applied afterwards. The mel-filterbank had 80 filters from 300 Hz to 8000 Hz. In addition, the input recording was normalised to ±1 before framing to reduce loudness variations between samples, and afterwards right shifted 9 times, or divided by 512, to match the scaling of the HLS-implementation. The result of the pre-processing, a 118x80 input data matrix, was stored to a text file with a filename corresponding to the input sample and stored in a subdirectory, "mels", within the source folders.

After the pre-processing program had been compiled with the specified parameters, two Python-scripts were written to automate the data generation process. The first, *generate_mels.py*, performed pre-processing for all .wav-files in a given folder. All samples from the dataset were stored within subfolders corresponding to their category, i.e. all samples for the word "no" were stored to the folder "no/mels/". This made it easy to select each word for a sample, as Python has a function for cycling through all elements in a folder. In addition to this script, another script, *generate_all_mels.py*, was written which called *generate_mels.py* for each subfolder, or word, in the dataset and defined in a target-directory array. To generate the pre-processing data for the words used in the Kaggle challenge, *generate_all_mels.py* was run for the words in question.

The Kaggle challenge required ten words to be interpreted by the neural network in addition to an unknown word and silence. For these networks, the unknown word was "Marvin". For the silence samples, text files with zeroes were generated. If no sound was present during recording, the pre-processing would get 0 from the FFT which would, which would be set to a low-bound value of 1e-10 to avoid errors from calculating the logarithm of 0.

4.2.3 Loading data into the network

To load the pre-processed data into the network during training, the data was loaded from the log-mel-filterbank text files, corresponding to each recording in the dataset, into the training program written in Python. The training program read through subfolders in the dataset corresponding to the words in the target list, storing all recording names in an input file register. The target list was split into three parts with a split ratio of 60/30/10 for training, validation and verification, respectively. The verification filenames and label values were saved to a text file for use in the verification process.

Loading the input data from the text files was performed by using the numpy.loadtxt function from the Numpy-library. This function loaded an input text file using a specified delimiter, in this case, a comma, into the numpy array. The array was then reshaped into a matrix with dimensions of 118x80. Initially, all training data used was loaded into memory before training, but as the training dataset increased during development, the memory size increased beyond the computer's capabilities. To fix this, a data generator was programmed, loading data from a list of files generated from the label folders. The input data was loaded into a 1D-array which was reshaped to match the input dimensions of the network prior to being loaded into the neural network. The data generator was stored as a class in a separate Python file, *data_classes.py.*

4.2.4 Training the network

The three network models were stored in a separate Python file, *NetworkModels.py*, defined as classes, while the training program was stored as *train.py*. The models were trained according to de Andrade et al. [17]: The optimiser used was "Adam", the number of epochs was 40, and the batch size was 64. The network was trained using Keras' "model.fit_generator" with three callback functions called after the end of each epoch: Early stopping, checkpointer and learning rate adjustment. The early stopper was set to stop after the learning rate had stopped improving for 5 consecutive epochs, while the learning rate adjuster was set to adjust the learning rate from 0.001 down to a minimum of 4e-5 using the function shown in equation (4.2). The LSTM and att-LSTM layers were trained using GPU-deployed LSTM layers, called CuDNNLSTM-layers.

For non-GPU devices, these layers were replaced with regular LSTM layers with *tanh* activation functions and *sigmoid* recurrent activation functions.

$$lrate = 0.001 \cdot 0.4^{\left\lfloor \frac{1+epoch}{10} \right\rfloor} \tag{4.2}$$

After training, the network which achieved the highest validation score during training was saved in the as a .hdf5-file.

4.2.5 Verifying network functionality

After the networks had been trained, they were verified using a separate verification program in Python, *predict.py*. The program loaded the saved neural network along with the list of verification files. The program ran for a user-definable number of iterations and loaded a verification file for each iteration. The number of iterations was divided by 12 to ensure that an equal amount of verification files for each word were loaded and tested. The networks were tested using the Keras-function "model.predict", returning an array of classification values for each label, representing the possibility of each label being the input word. The label with the highest value, and the classification value, were returned and compared with the reference label in a confusion matrix, and the classification values were plotted in a graph. In addition to the classification values, the execution time was recorded and saved in a text file for each classificaion using "time.perf_counter()" for further analysis. Additionally, the classification values were filtered using thresholding to generate precision and recall curves for all words and for the whole model.

To prepare the LSTM networks for deployment on non-GPU targets, the CPU-compatible networks were created based on the weights of the trained networks. The weights extracted from the .hdf5 network file and applied to a network with corresponding, CPU-compatible architecture using a Python-script, *create_cpu_network.py*. The CPU-compatible network models were stored in a separate model file, *Network-Models_for_CPU*.

4.3 FPGA acceleration

4.3.1 Intel OpenVINO

To accelerate the neural networks on an FPGA, the Intel OpenVINO SDK was used. OpenVINO version 2019 R1.01 was installed on a server at NTNU running Ubuntu 16.04 with an Intel Arria 10 GX development kit, following the "Install Intel® Distribution of OpenVINO™ toolkit for Linux with FPGA Support" guide [60]. During installation of the FPGA support-files, the *setup_env.sh*-script had to be edited to target the Arria 10 development kit. This was done by changing "USE_HDDLF" from 1 to 0.

To make the network compatible with the Inference Engine for deployment on the FPGA, it was converted to an Intermediate Representation using the Model Optimizer. As the optimiser was incompatible with the .hdf5-format of Keras, it was converted to TensorFlow's .pb-format by "freezing" the model. This was done by writing a Python-script which loaded a Keras-model, froze it using graph-utilities from the TensorFlow library and saved it as a TensorFlow-model.

After converting the file to the .pb-format, the network was optimised using the TensorFlow-version of the Model Optimizer, specifying the input model, the input shape and the precision. Only the CNN was converted to an intermediate representation. As LSTM networks were not supported for TensorFlow nor Keras, attempts were made to convert the model into a framework compatible with OpenVINO and LSTM layers. Using Microsoft's MMdnn-toolkit, the CPU-compatible LSTM and att-LSTM networks were converted into MXnet from TensorFlow. This, however, proved unsuccessful as the conversion had created layers unsupported by OpenVINO. Another solution was attempted: By switching the Keras-backend from TensorFlow to MXNet, MXNet-models could be created and exported after training. The LSTM networks were re-trained and saved as MXNet-models, but were still incompatible with OpenVINO.

To run the neural network on the FPGA, a C++-program was written to interface with the Interface Engine, load the input data and verify the accuracy of the network. The program was based on the program written for the semester project, removing the voice recording functionality, instead changing the data loading to load verification

data from text file. The program follows the program flow illustrated in figure 2.2. Initially, the program reads several input arguments: The path to the neural network IR files, the number of iterations to run, the path to the verification register text file, what device to deploy to and whether to display detailed debug information. To decide the input data, the program read the verification register text file and added them to an input data file array. The number of loops was divided by 12 to get an equal amount of files. To avoid running the network without input data, the input data was "padded" with samples from the "unknown" label. This padding was performed after an equal amount of input data has been loaded for each label in case of uneven label distribution. The input and output precisions of the network were set to single-precision floating point.

```
Performance stats for: dense_2/Relu
Layer status: Executed
Exec type: jit_avx2_FP32
layer type: ReLU
Realtime run: 1us
----
Performance stats for: dense_3/MatMul
Layer status: Executed
Exec type: gemm_blas_FP32
layer type: FullyConnected
Realtime run: 1us
----
Performance stats for: dense_3/Softmax
Layer status: Executed
Exec type: ref_any_FP32
layer type: SoftMax
Realtime run: 2us
```

Figure 4.4: Layer execution times as reported by the Inference Engine

After deciding what input text files to use for verification, the samples were loaded iteratively for each inference loop. The corresponding label was checked against the reference label, and the output was stored in a text file and printed to the console. In addition to the labels, the execution time for each layer in the network was reported by the Inference Engine and, if running with debug mode, printed to the console, producing the output shown in figure 4.4. The total runtime for each inference and prediction was stored, and the average, median and minimum execution times were

calculated along with the corresponding throughputs, as shown in figure 4.5.

```
Sample: unknown/MFCC/e95c70e2_nohash_0.dat
Loading array
Read 9441 input values
Filling input buffer
Results:
Neural Network output
Predicted word: unknown
Actual word: unknown
Execution time from IE:     1.424ms
-------------------------------------
Inference Engine measurements
Time        Avg: 1.424ms.          Median: 1.425ms.          Fastest: 1.388ms.
Throughput  Avg: 702.2 samples/s.  Median: 701.8 samples/s.  Fastest: 720.5 samples/s.
```

Figure 4.5: Execution times and throughputs reported at the end of Inference Engine-based prediction using OpenVINO

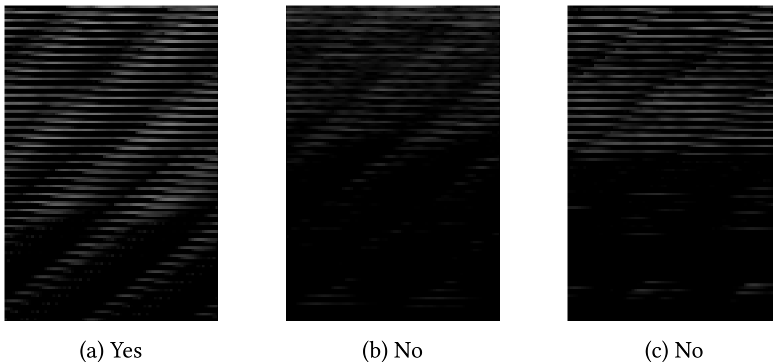
To run the network on an FPGA, the FPGA was programmed with a pre-generated bitstream supplied by Intel along with the OpenVINO SDK. The bitstreams used were "2019R1_A10DK_FP16_ResNet_SqueezeNet_VGG" and "2019R1_A10DK_FP11_ResNet_VGG", due to VGG having similar network topology to the inferred networks. The model was converted into single-precision and half-precision floating point to examine the effects of reducing the model precision.

4.3.2 Xilinx DNNDK

In addition to OpenVINO, FPGA acceleration using Xilinx DNNDK was attempted. Xilinx DNNDK SDK version 2.08 was installed on an AVNet Ultra96 and a laptop; the laptop serving as a development platform while the Ultra96 served as the inference target. Initially, example code from Xilinx was run to demonstrate the capabilities of the system. One example, showcasing pose detection CNN acceleration, achieved a throughput of around 20 frames per second, while another example showcasing image classification using ResNet, achieved a throughput of around 25 images per second. As the development process using DNNDK was similar to OpenVINO, performing quantisation on the network then creating an intermediate representation before deployment through an API in C++, the network models were initially prepared for quantisation. DNNDK version 2.08 supported only Caffe-models, requiring the Keras-trained models to be converted, which was performed using MMdnn. As MMdnn

had no support for LSTM layers, only the CNN model was converted into Caffe. In addition, it was discovered through communications with Xilinx that LSTM networks were not supported in the current version of DNNDK.

After conversion, quantisation was attempted using the DNNDK-tool, DECENT. Initial optimisation efforts proved fruitless, as the quantisation required input data during the quantisation process to validate the optimisations. DNNDK required the input data to be specified using any data-layer in Caffe, such as image data or HDF5. After creating a text file according to the specifications of the image data layer, with the log-mel-filterbank text files as the "image" data input. This did not work, as the data layer required input files in image-data formats such as .png or .jpeg. To provide image files for the data input, a Python-script was written to convert the log-mel-filterbank energies into images. Using the Pillow library, the 118x80 log-mel-filterbank energies was converted into a 118x80 pixel greyscale image in the .png-format. Figure 4.6a shows a generated image for the word "yes", while figures 4.6b and 4.6c show images for the word "no".



(a) Yes (b) No (c) No

Figure 4.6: DECENT input images generated from log-mel-filterbank energies using the Pillow library for Python

After picture generation, the input data list was updated with the generated images. Running the quantisation again resulted in a reported model loss of 0. In addition, an optimised network was generated. Following quantisation, kernel compilation using

the DNNC tool was attempted. Using the newly quantised network, DNNC was run targeting a 64-bit ARM processor, as found on the Ultrascale+ on the Ultra96. Running DNNC resulted in an error and a subsequent program exit, reporting an assertion failure for convolutional layer 2, as shown in figure 4.7.

```
[DNNC][DEBUG] Generate dpu instruction for node [conv2d].
dnnc: /tmp/DNNC_V010_Package/dnnc/submodules/asicv2com/src/SlNode/SlNodeConv.cpp:77: void SlNodeConv::ge
nerate_convinit_op(const YAggregationType&, const YAggregationType&, uint32_t, uint32_t): Assertion `con
v_param->get_shift_cut() >= 0' failed.
Aborted (core dumped)
```

Figure 4.7: Error message generated by DNNC upon kernel compilation

Researching online and reaching out to Xilinx resulted in little progress in regards to debugging the network, so a decision was made to wait for the next update to the DNNDK SDK, scheduled for late April, but finally released in May. Further attempts at accelerating the networks using DNNDK were not attempted as focus was shifted to OpenVINO.

Chapter 5

Results

This chapter presents the results of the implementation. First, the results for the pre-processing are shown for the CPU implementation, followed by the results of the code deployed on the FPGA. Following the pre-processing are the results of the neural networks in terms of training and verification on CPU. Lastly, the chapter presents for the FPGA acceleration of the neural networks, including the successful OpenVINO acceleration and the unsuccessful DNNDK acceleration. The code pre-processing source code and neural network related code is available on GitHub.[1]

5.1 Pre-processing

5.1.1 On Processing Unit

Running on an Avnet Ultra96, the initial C++-code, prior to being ported to Vivado HLS, had a total execution time of 718.6 ms with the time spent for each section of the code shown in table 5.1.

[1] https://github.com/andernil/lstm_speech_recognition

Table 5.1: Execution times for each part of the C++ pre-processing code running on an Ultra96

Code section	**Time spent [µs]**
Read .wav-file	1125
Calculate window FFT	633104
Generate filterbanks	1067
Calculate log-mel filterbank energies	83314

5.1.2 On FPGA

5.1.2.1 Xilinx FFT example code

Software/hardware co-simulation using Vivado HLS reported an estimated execution latency of 1032 clock cycles and an execution interval of 512 clock cycles for the xfft2real-function, and an execution latency of 1025 clock cycles and an execution interval of 1025 clock cycles for the real2xfft-function. No estimate was provided for the FFT-core as it was not part of the synthesised code. The 16-bit FFT example code was deployed on the Ultrascale+ MPSoC ZU3EG A484 FPGA on an Ultra96. The input data was 118 frames of 1024 samples of 16-bit floating point data. Including I/O-operations to and from the PS to the FPGA, the synthesised program used 758566 clock cycles, or 3792.73 µs on the pre-processing, not counting post-processing such as scaling. Figure 5.1 shows the calculated power data from the 16-bit HLS implementation and the C++ source. In addition to the 16-bit FFT example, a modified 32-bit FFT example was simulated and synthesised, though it was not able to run on the FPGA. Still, it achieved an estimated clock cycle count equal to the 16-bit implementation.

Using this power data, the corresponding log-mel-filterbank energies were calculated, shown in figure 5.2.

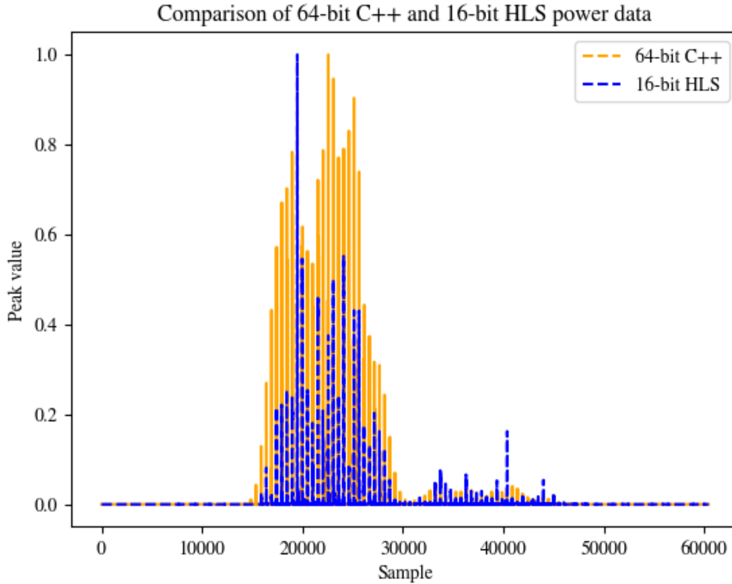


Figure 5.1: Comparison of power data calculated using 64-bit C++ and 16-bit HLS

5.1.2.2 32-bit FFT custom code

Software/hardware co-simulation using Vivado HLS reported an estimated execution latency of 7331 clock cycles and an execution interval of 7332 clock cycles for the FFT-function, copying and data to and from the FFT IP-core in addition to performing the FFT with the FFT-function as the top function. For 118 frames, this results in an estimated execution latency of 865176 clock cycles. In addition, selecting the window-FFT-function as the top function, which included scaling, framing and windowing, resulted in an estimated execution latency and interval of 2529653 clock cycles, or 25.30 ms at a clock speed of 100 MHz. This estimate was for the whole FFT-preprocessing, i.e. for all 118 frames with an FFT-length of 1024. Both FFT and window-FFT as top functions yielded an average accuracy of 99.86% on the output data when compared to the C++-implementation. The resources used for the synthesised

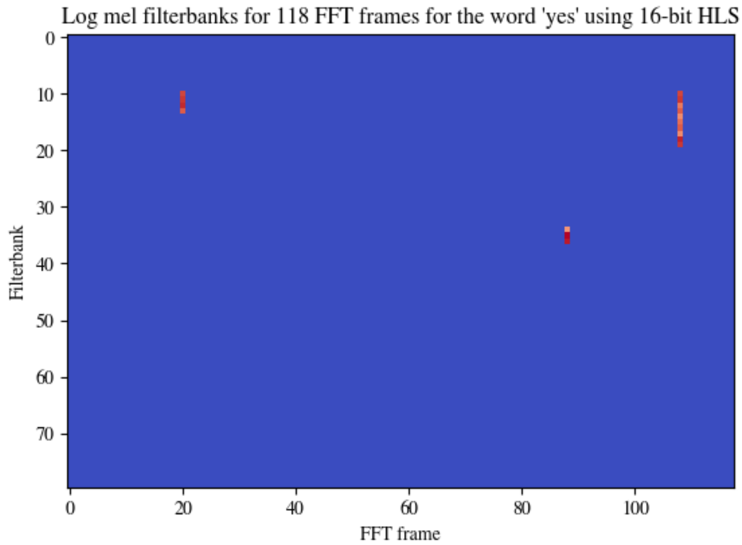


Figure 5.2: Hot/cold Colormap-representation of log-mel-filterbank energies for the word "yes". Calculated using 16-bit HLS FFT with a window size of 1024, shift length of 128 and 80 mel-filterbanks from 300 Hz to 8000 Hz

solution is shown in table 5.2.

The synthesised IP blocks for both implementations were not able to run on the FPGA, waiting indefinitely for the AXI DMA service to become available after the initial data transfer. Due to no debug information being available, no further progress was made.

5.2 Neural networks

The networks were trained on a desktop computer with an Intel i5 3570k CPU, an Nvidia GTX 1070 GPU and 8 GiB of DDR3 RAM. Verification was performed on the

Table 5.2: Resource usage estimates for synthesised single-precision mel-log-filterbank energy pre-processing on Ultrascale+ ZU3EG A484 FPGA

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	62	219	5658	-
FIFO	8	-	196	272	-
Instance	13	44	20426	19930	-
Memory	16	-	213	852	-
Multiplexer	-	-	-	589	-
Register	-	-	1831	-	-
Total	37	106	22885	27301	-
Available	432	360	141120	70560	-
Utilisation (%)	8	29	16	38	-

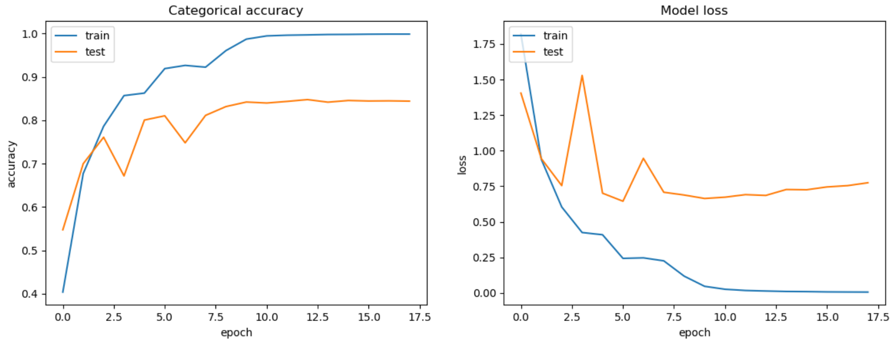
same computer used for training unless otherwise stated.

5.2.1 Training results

The networks were trained with 1740 recordings for each category, limited by the number of recordings for the unknown word, "Marvin". The early-stopper was set with a tolerance of 5.

5.2.1.1 CNN

The CNN was trained for 17 epochs, lasting a total of 6 hours, before being stopped by the early-stopper, achieving a maximum validation accuracy of 0.847. Figure 5.3 shows the accuracy and loss through the training epochs.



(a) CNN training and validation accuracies during training

(b) CNN training and validation loss during training

Figure 5.3: Training and validation loss and accuracy for CNN

5.2.1.2 LSTM

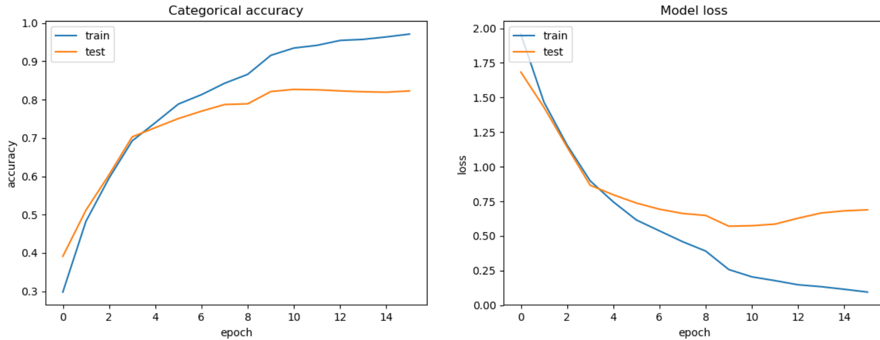
The LSTM network trained for 3 hours across 15 epochs, achieving a maximum validation accuracy of 0.823. Training and validation accuracy and loss are shown in figure 5.4

5.2.1.3 Attentive LSTM

The att-LSTM network trained for 4 hours across 26 epochs, achieving a validation accuracy of 0.86276. Training and validation accuracy and loss are shown in figure 5.5

5.2.2 Verification results

The three networks were verified using a subset of the dataset, consisting of 2088 samples in total, split from the rest during training. Classification was performed on the same computer as training. From these, 600 were used for verification; 50 for each class. Initially, no threshold value was used for the predictions; it was later increased to 1 and decreased by 0.05 for 20 steps to produce precision and recall curves for further



(a) LSTM training and validation accuracies during training

(b) LSTM training and validation loss during training

Figure 5.4: Training and validation loss and accuracy for LSTM network

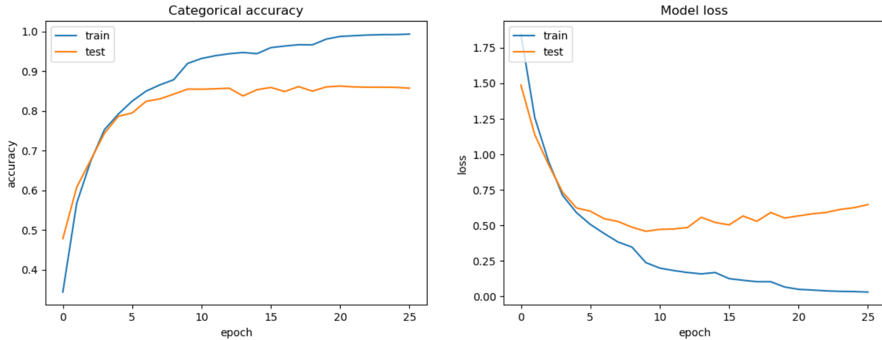
analysis. Precision and recall were calculated for every word and for the complete model by averaging across the per-word precision and recall.

To calculate precision and recall, three classification classes were defined for a given word: True positive, the corresponding label and input word matching the given word; false positive, different input word but the predicted label matches the given word; and false negative, the incorrect label for the given word or sub-threshold classification value. The false negative class was undefined as the model always classified a word, either "silence" for no word or "unknown" for unknown words.

Using the verification program from earlier, *predict.py*, with the same words used for verification for all networks, the three networks were tested with varying thresholding. Initially, no thresholding was used, resulting in the following confusion matrices, along with average precision and recall curves. Precision and recall curves for every word with all three networks are included in appendix B.

5.2.2.1 CNN

Figure 5.6 shows the confusion matrix for the CNN with no threshold on the prediction values. Average prediction time was 5 ms, while the median was 2.3 ms. The



(a) att-LSTM training and validation accuracies during training

(b) att-LSTM training and validation loss during training

Figure 5.5: Training and validation loss and accuracy for att-LSTM network

classification accuracy accuracy was 84.3%.

Figure 5.7 shows the precision and recall for the network with thresholding on the prediction value. The prediction threshold was reduced from 1.00 to 0.00 in decrements of 0.05 for 20 steps, stabilising at a precision of 0.84 and a recall of 0.84.

5.2.2.2 LSTM

Figure 5.8 shows the confusion matrix for the LSTM network with no threshold on the prediction values. Average prediction time was 26 ms, while the median was 24 ms using GPU-accelerated LSTM-layers. On the non-GPU model, the average prediction time was 521 ms, while the median was 514 ms. Both models achieved a classification accuracy of 83.1%.

Figure 5.9 shows the precision and recall for the network with thresholding on the prediction value. The prediction threshold was reduced from 1.00 to 0.00 in decrements of 0.05 for 20 steps, stabilising at a precision of 0.834 and a recall of 0.837.

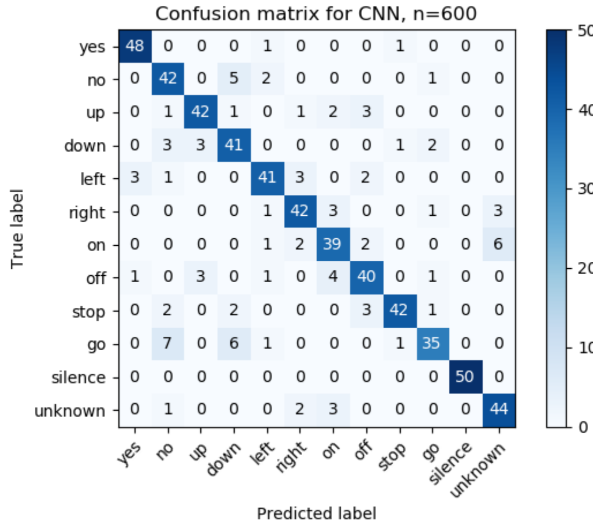


Figure 5.6: Confusion matrix for CNN for 12 words with no thresholding

5.2.2.3 Attentive LSTM

Figure 5.10 shows the confusion matrix for the att-LSTM network with no threshold on the prediction values. Average prediction time was 27 ms, while the median was 24 ms. On the non-GPU model, the average prediction time was 540 ms, while the median was 516 ms. Both networks achieved a classification accuracy of 86.5%.

Figure 5.11 shows the precision and recall for the network with thresholding on the prediction value. The prediction threshold was reduced from 1.00 to 0.00 in decrements of 0.05 for 20 steps, stabilising at a precision of 0.86 and a recall of 0.86.

5.2.2.4 CNN on HP Z800 server

As only the CNN was optimised into an intermediate representation using OpenVINO, it was the only network run on the FPGA. To provide grounds for comparison, the Keras CNN implementation was run on the HP Z800 workstation to reduce the runtime impact of having different CPUss when comparing the OpenVINO results with the

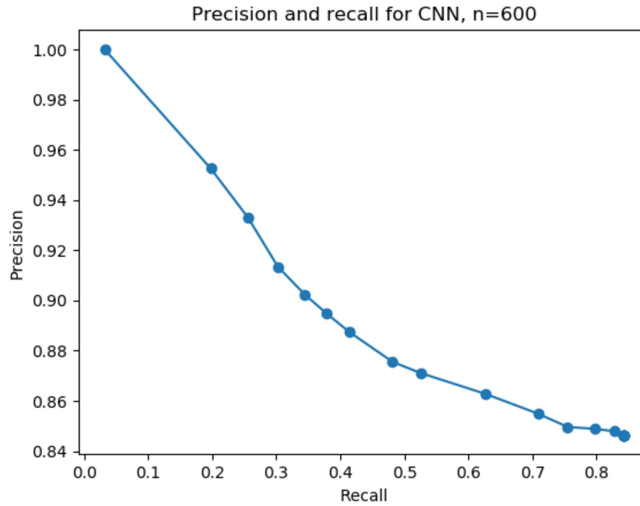


Figure 5.7: Precision and recall curve for CNN with thresholding from 1.00 to 0.00 with 0.05 decrements per step

Keras results. Running on the Z800 server, the network had an average runtime of 22 ms and a median time of 2.5 ms.

5.3 Neural network acceleration using OpenVINO

Initially, the network was optimised using the Model Optimiser with a precision of 32-bit floating point. After getting results, the network was optimised again with a precision of 16-bit floating point for further optimised FPGA deployment.

5.3.1 On CPU

The optimised model was run on the workstation CPU, achieving an average runtime of 7.4 ms and a median of 7.4 ms. The confusion matrix for the CPU inferred network is shown in figure 5.12, while figure 5.13 shows the execution times for the network.

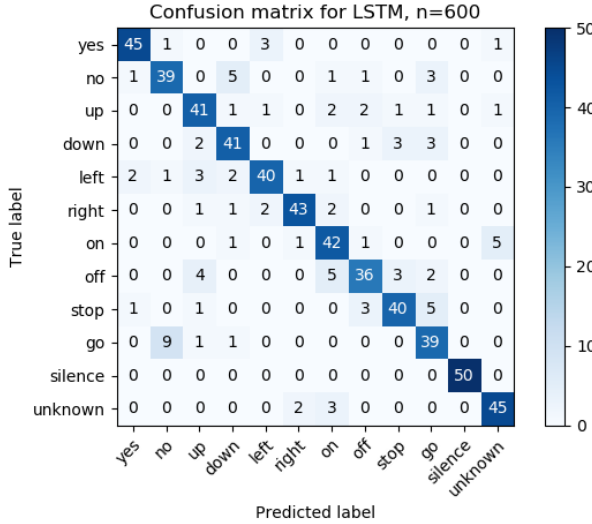


Figure 5.8: Confusion matrix for KWS LSTM for 12 words with no thresholding

The network achieved a classification accuracy of 84.6%.

5.3.2 On FPGA

On FPGA, the model was run using four combinations of model and bitstream precisions: single-precision and half-precision floating point models and 16-bit and 11-bit FPGA bitstreams. The network was deployed in "Hetero" mode, accelerating all convolutional and pooling layers on the FPGA while the dense-layers were run on the CPU. Table 5.3 shows the minimum, median and average execution times for the different configurations. Figure 5.14 shows the execution times for target- and bitstream combinations, while table A.1 shows the execution times for each layer for the CPU and FPGA implementations. The confusion matrices for the FPGA inferences are included in appendix C.

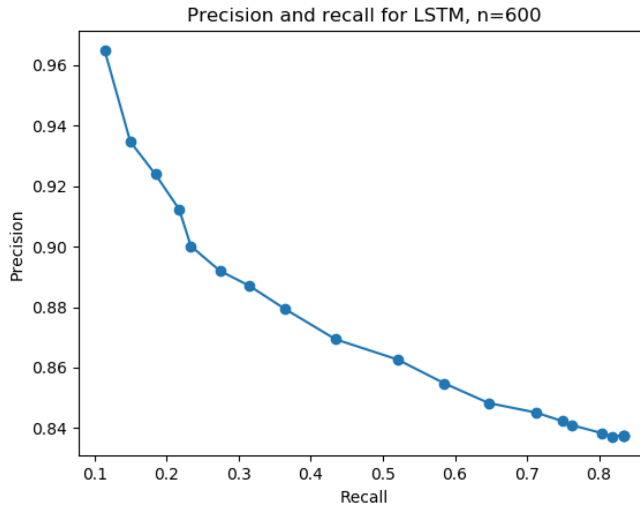


Figure 5.9: Precision and recall curve for KWS LSTM network with thresholding from 1.00 to 0.00 with 0.05 decrements per step

Table 5.3: Execution time minimums, medians and averages for OpenVINO accelerated CNNs on different inference targets and precisions

Device	Precision	Bitstream	Minimum [ms]	Median [ms]	Average [ms]
CPU	FP32	-	7.00	7.41	7.45
FPGA	FP32	FP16	5.50	6.17	6.46
FPGA	FP32	FP11	3.16	3.70	3.97
FPGA	FP16	FP16	3.59	4.08	4.44
FPGA	FP16	FP11	3.13	3.69	3.96

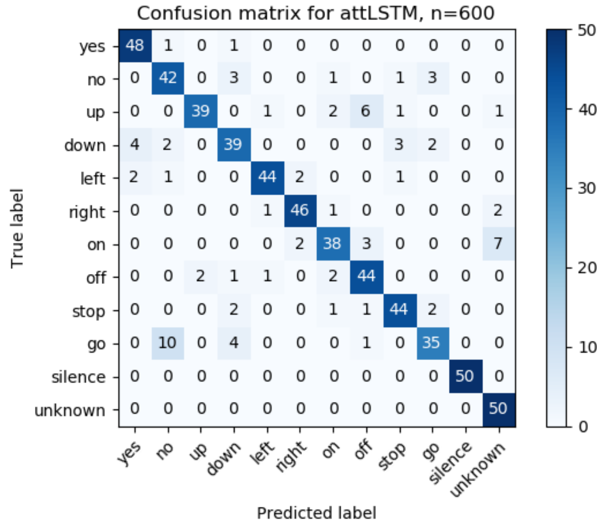


Figure 5.10: Confusion matrix for KWS att-LSTM network for 12 words with no thresholding

Table 5.4: Classification accuracies for OpenVINO accelerated CNNs on different inference targets and precisions

Device	Precision	Bitstream	Accuracy [%]
CPU	FP32	-	84.8
FPGA	FP32	FP16	84.8
FPGA	FP32	FP11	84.5
FPGA	FP16	FP16	84.8
FPGA	FP16	FP11	84.6

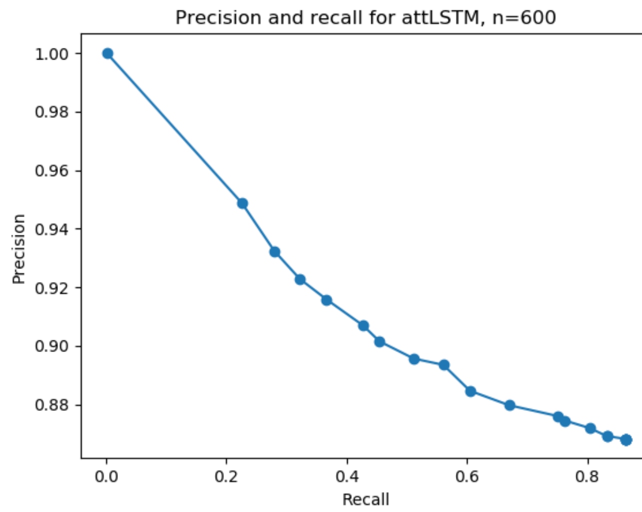


Figure 5.11: Precision and recall curve for KWS att-LSTM network with thresholding from 1.00 to 0.00 with 0.05 decrements per step

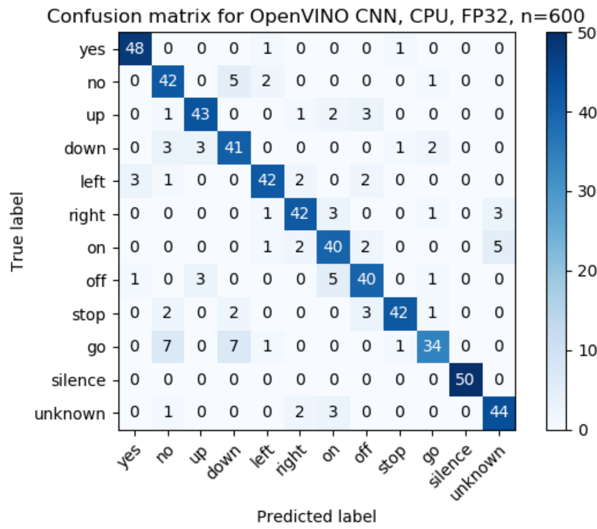


Figure 5.12: Confusion matrix for single-precision CPU-inferred KWS CNN

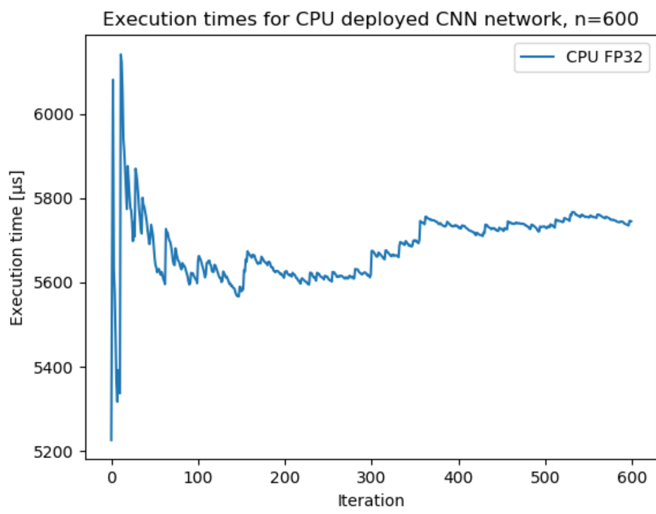


Figure 5.13: Runtimes for single-precision CPU-inferred KWS CNN

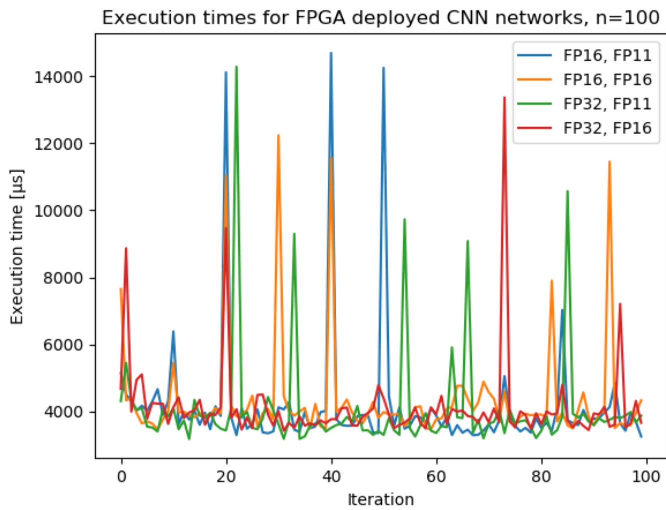


Figure 5.14: Execution times for different precision CPU/FPGA-deployed KWS CNNs using OpenVINO

Chapter 6

Discussion

This chapter investigates and discusses the results from the previous chapter. Along with pre-processing, neural networks and FPGA acceleration results, the chapter also discusses the programs and tools used.

6.1 Pre-processing

6.1.1 Comparing C++ and HLS

Figures 4.1 and 5.2 reveal that the power data generated by the two implementations, illustrated in figure 5.1, differs too much to produce accurate results. Figure 5.1 does show some similarities in the normalised power data calculated by the two implementations; however, the result differs in several areas, most notably peak values. The cause of this is the resolution of the data types. Figure 5.1 shows normalised values close to zero, while in actuality the C++-values are greater than 0, leading to more detailed power data and consequentially more detailed log-mel-filterbank energies. This result is somewhat predictable, as half-precision floating point is inherently less accurate than double-precision floating point. This inaccuracy, as a result of lower resolution, is confirmed when running the ported C++-code in Vivado HLS with an FFT

resolution of 16-bit, achieving an average accuracy of 21%. Figure 5.1 shows that the half-precision FPGA implementation is able to calculate a rough but somewhat correct FFT, indicating that it can be used for less precise tasks. The 32-bit FFT implementation was not able to run on the FPGA, but the simulated results indicate that single-precision floating point is sufficient for the task of speech pre-processing, achieving an average calculation accuracy of 99.86%.

Analysing the execution times in table 5.1, it is clear that the FFT is the most time-consuming part of the computation, comprising 88% of the total execution time. The 16-bit FFT implementation is 166 times faster by comparison, though highly inaccurate. No results were achieved for the 32-bit FPGA-implementation; however, an execution time can be estimated based on the synthesis results of the attempted 32-bit port of the C++ code, the 32-bit modification of 16-bit Vivado example code, and the execution time results of the 16-bit implementation. Synthesising the ported C++-code with the FFT-function as the top function, with a resolution of 16 bits, results in an estimated clock cycle interval equal to the 32-bit implementation. Based on this, it can be assumed that the execution latency and interval of the FFT IP-core, using the Radix-2 architecture, is similar for 16-bit and 32-bit precisions. Changing the architecture of the FFT IP-core from Radix-2 to streaming, which is the architecture used for the 16-bit example, results in an execution interval and latency of 3196 clock cycles, half of the cycles spent on the Radix-2-architecture. Adjusting the FFT-length to 512 for the same code results in an execution interval and latency of 1654 clock cycles.

Based on the assumptions and results above, the execution time of the Vivado HLS-port of the C++-code can be roughly estimated to be four times that of the 16-bit example, resulting in an estimated execution time of 16 ms. This is faster than the estimate provided during synthesis, which was 25.30 ms. Realistically, the execution time estimated during synthesis would be lower than the implemented speed due to the overhead associated with transferring data to and from the FPGA from the CPU. This would be negligible compared to the time spent performing the FFT as the PS is located on the same chip as the FPGA. For comparison's sake, the estimated time spent performing the custom pre-processing can be rounded up to 26 ms. This would result

in a speedup by a factor of 24 when compared to the PS implementation. Taking into account the whole pre-processing, the estimated execution time would be 111.5 ms, resulting in a total pre-processing speedup by a factor of 6.4.

Examining table 5.2, the synthesised code utilises 29% and 39% of DSP and Lookup Table (LUT) resources on the FPGA. Though no statistics are available for the resource utilisation of the DPU core used by DNNDK, it most likely uses a substantial amount of DSP and LUT resources as they are a source of significant speedup when compared to flip-flops. As such, running DNNDK and pre-processing on an Ultra96 is unlikely, most likely requiring a larger DNNDK-compatible development kit such as the ZCU102 which has a larger and more powerful FPGA.

The neural networks used, and the ones from McMahan et al. [15], did not use MFCC values for input. This was, according to McMahan et al. [15], due to the "high noise intolerance of MFCC features" [15]. de Andrade et al. [17] did not specify any reason for not using MFCC values as the input for their networks, but it was most likely for the same reasons. This affects pre-processing runtime, as MFCC values requires performing several DCTs. The FFT IP core from Xilinx has a runtime re-configurable transform length, and could be reduced to the required DCT length. These transforms would then most likely be computed using the FFT module instantiated on the FPGA, which would increase pre-processing time.

6.1.2 Using Vivado HLS for FPGA acceleration

Using Vivado HLS for FPGA acceleration of C/C++-code is a seemingly simple task at first glance, with examples and several compatible libraries to help port code. When problems occur, however, debugging information is scarce, error messages are vague, and pragma directives can be confusing; though Xilinx does have several user guides to help explain how the program works. Due to this, the initial learning phase is difficult unless supported by an Field Applications Engineer (FAE). The integration process, using Vivado and Vivado SDK to run the program on FPGA and PS, provides less debug information than Vivado HLS, making debugging difficult, and due to the complex nature of the interfacing, a difficult task. The IP Block Design tool helps

alleviate some of the complexity through connection automation, but this also makes it more difficult to debug.

6.2 Neural networks

6.2.1 Accuracy evaluation

Comparing the three networks, it's initially clear that the att-LSTM network has the highest accuracy, achieving an overall classification accuracy of 86.5%, though not as high as the 96.9% achieved by de Andrade et al. [17]. The achieved score was not significantly higher than the 84.3% achieved by the CNN, and even more surprisingly, the LSTM network performed worse overall than the CNN, with a classification accuracy of 83.1%. Figures 5.7, 5.9 and 5.11 show that the att-LSTM curve has the highest and most linear degradation of classification precision as the threshold is decreased and recall increases. The LSTM network declines the quickest, losing 6% of precision between thresholds of 1.00 and 0.75, while the CNN is similar to the att-LSTM-network, though decreasing quicker.

The precision and recall curves in appendix B also show that the networks are good at classifying different words. This indicates that for a network requiring high precision and recall, an ensemble of neural networks can be used; however, this is more computationally intensive.

The confusion matrices for the networks, figures 5.6, 5.8 and 5.10, show that the most challenging words to classify are "go" and "on", with "go" being the most difficult, being classified as "no" for at least 7 classifications. This is not surprising, as both words have similar duration and sound. Interestingly, "no" is not misclassified as "go" a comparable number of times, instead being classified as "down". Figures B.2 and B.10 reveal that the precision sharply declines as the threshold is decreased for "no", indicating that the classification confidence is low for this word. This can indicate that more training is required to increase the accuracy or that the training data is too similar, not capturing the differences separating the two words. This can possibly be solved by applying high-pass filtering to the audio samples during pre-processing to

decrease the impact of the low-frequency sounds of the "o" in "no" and "go".

6.2.2 Runtime comparison

Comparing the runtimes from chapters 5.2.2.1, 5.2.2.2 and 5.2.2.3 for CPU, it is clear that non-GPU accelerated deployment of LSTM networks comes at a significant decrease in throughput. The median classification time of the CNN was roughly 226 times shorter than the LSTM network; however for GPU-accelerated LSTM layers the CNN classification time was roughly 10 times shorter. Still, this is a considerable difference for CPU-deployed networks. The difference might be a result of memory speed differences, as GDDR-ram used mainly by the GPU has higher bandwidth and wider busses than the DDR-ram used mainly by the CPU.

6.2.3 Using Keras for neural network development

Designing and developing neural networks using Keras is simple but powerful. Networks are designed by specifying layers, and connections are automatically made between the layers. Each layer features several adjustable parameters, and are all documented in Keras' user guide as well as other online resources. Due to its simplicity, Keras has been widely adopted by researchers and hobbyists, resulting in readily available information online on user forums and blogs. As Keras allows for several backends to be used, the generated networks can be deployed on different targets or in different situations. MXNet, for instance, uses fewer resources than TensorFlow, making it more ideal for more extensive networks requiring more resources, while TensorFlow can be used for smaller networks on platforms such as Google's Coral edge-AI device.

6.3 FPGA acceleration

6.3.1 OpenVINO accuracy

Figures 5.6, 5.12 and the figures in appendix C reveal that the results are practically the same, with small variations for the words "off", "on". Table 5.4 confirms this, as the maximum accuracy deviation from the CPU implementation is 0.03%.

6.3.2 OpenVINO runtimes

To serve as a basis for evaluating the speedup provided by FPGA inference, the CNN was run on the Z800 workstation with CPU as the target device. As shown in chapter 5.2.2.4, the network was able to achieve a median classification time of 2.05 ms, while the optimised network using OpenVINO achieved a median classification time of 7.45 ms. Comparing the results with table 5.3, the Keras CPU implementation was quicker than all FPGA variations. This is surprising, as one would expect the combination of the optimisations performed by the Model Optimiser in addition to the parallel possibilities of the FPGA to make the runtime faster.

The runtimes in table A.1 reveal why the execution times are greater: The dense1-layer, arguably the most computationally intensive layer in the network, is run on the CPU. In addition, most of the time spent during FPGA processing is spent on transfer to and from the FPGA. The table clearly illustrates the benefits of FPGA acceleration when comparing the convolutional layers. As all convolutional layers are run on the FPGA, the execution time is roughly improved by a factor of 6.5 for all FPGA and precision configurations. All FPGA execution times are relatively similar, further strengthening the assumption in chapter 6.1.1 that FPGA execution times are similar for half- and single-precision floating point computations.

Examining figure 5.14 reveals relatively large spikes in the execution time, sometimes increasing the execution time by a factor of 3.5. The reason for this might be low system priority; however, it seems that the spikes occur roughly at the same iteration for all curves, such as iteration 20, 76 and 85. This might indicate a buffer overload when transferring to and from the FPGA. Comparing the execution times for CPU and

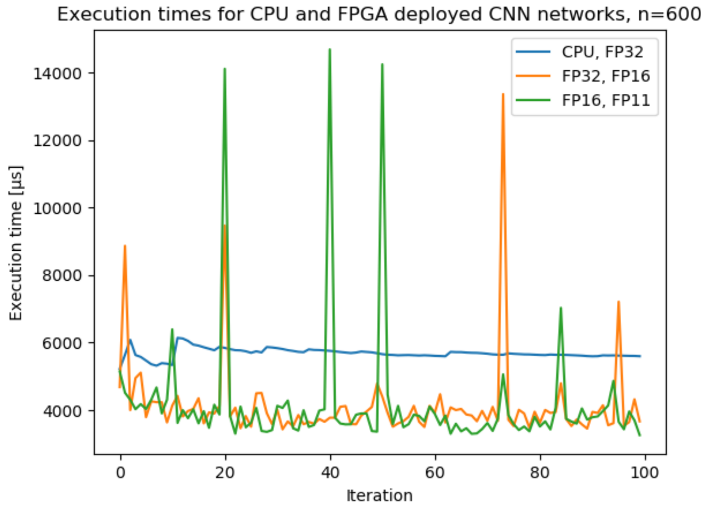


Figure 6.1: Classification times across 100 iterations with CPU and FPGA as inference targets using OpenVINO

FPGA in figure 6.1 reveals that the CPU has no indicating, further strengthening this theory, though figure 5.13 shows some initial spiking in execution time, though not as severe as the FPGA spikes.

6.3.3 Using Xilinx DNNDK for FPGA acceleration

The Xilinx DNNDK SDK is the most recent of the two SDKs, something that becomes apparent when developing with it. There is little information online, and only one user guide available in addition to some examples on Github, and this makes it difficult to program for once the target application deviates from the examples included with the SDK. In the case of speech recognition, the major flaw limiting the adoption of Xilinx DNNDK is the quantisation step, as the input data methods available were designed primarily for image recognition. Still, speech recognition might be possible at the current state of the SDK, as the HDF5-input format might be used to input arbitrary

data into the quantisation program. Recently, Xilinx added support for TensorFlow networks to the SDK, which might also make it possible to use DNNDK for non-image recognition purposes. In addition, a separate Xilinx AI SDK is included along with the newest version, which simplifies development and integration of DNNDK with Petalinux, the OS running on the Ultra96.

Most of these problems can be categorised as early-adoption problems, as more documentation and user guides will most likely appear once users start using the program. Having the SDK compatible with the Ultra96 reduces the entry cost for FPGA acceleration for consumers, increasing the chance of user adoption. In addition to entry cost, DNNDK has an advantage over OpenVINO as it does not use all the resources available on the FPGA, allowing for custom HDL code to be run alongside it.

6.3.4 Using OpenVINO for FPGA acceleration

Compared to DNNDK, OpenVINO is a more mature and well-documented FPGA acceleration toolkit. Documentation from Intel's end is good, and it has a sizeable following resulting in active forums and blogs which can serve as helpful resources when programming with the SDK. The program flow is relatively simple to follow, and the compatibility with several neural network frameworks makes it versatile. The possibility to run the same accelerated program with the Inference Engine on any target device, by changing one parameter, makes it simple to test and verify the functionality of the program. The quantisation is also optional with several options available, with 8-bit integer precision quantisation performed similarly to DNNDK through verification.

6.4 Error sources

6.4.1 Classification accuracy

As mentioned in chapter 6.2.1, the trained networks were not able to reproduce the classification accuracies achieved by de Andrade et al. There are several error sources which might contribute to this lowered accuracy, such as incorrect training data. As

mentioned in chapter 4.2.2, the input samples from the .wav-file were normalised to ± 1 and then divided by 512, further narrowing the dynamic range, which might cause inaccurate FFT results. This seemingly small error is propagated through the rest of the pre-processing, possibly creating a larger error as a result.

Differences in pre-processing are, arguably, the cause of the different classification accuracies. de Andrade et al. [17] used the Python-library Kapre [61] to perform the pre-processing, which might perform additional processing on the input signal such as pre-emphasising or filtering.

During training, an early stopper was used to avoid unnecessary training epochs due to stagnated or lowered classification accuracy. The tolerance used during the training of the networks in this report might have been too strict, stopping the training process when the classification accuracy reached a local minimum instead of the global minimum.

6.4.2 Classification times

As shown in figure 5.14, the classification times achieved using OpenVINO varied significantly at times, spiking at nearly 15 ms. These deviations are most likely a result of process priorities. As the operating system used did not feature any real-time capabilities, OpenVINO had lower priority than the operating system, resulting in it being halted when OS tasks such as garbage removal were due to be performed. On an edge-AI device, the OS would most likely be an Real-Time Operating System (RTOS), ensuring more predictable classification times.

In addition to priority, the hardware on the system provided limitations on the classification times. OpenVINO has recommended system requirements of a 6th or 8th generation Intel CPU, and the Intel Arria 10 GX development kit requires a PCI Express Gen 3.0 x8 port. The Z800 workstation has neither of these: The CPU was a 2nd generation Intel CPU, and the motherboard used in the system had a PCI Express Gen 1.0 x8 port. Despite these limitations, OpenVINO ran successfully on the system, though most likely at limited capacity. The 6th generation and onward support the AVX2 instruction set extensions, which OpenVINO uses when targeting the CPU for

inference. In regards to the PCI Express port, the generation difference represents a speed difference of a factor of 4, with Gen 1.0 at 2 Gbit s^{-1} and Gen 3.0 at 8 Gbit s^{-1}. Examining table A.1 strengthens the theory that transfer speed is a limiting factor, as roughly 2/3 of the processing time on FPGA is spent on transfer to and from the FPGA.

6.5 Further work

6.5.1 FPGA acceleration of pre-processing

As no actual results were achieved for the FPGA acceleration of the single-precision pre-processing on the Ultra96, a next step would be to continue working on this example. This would, initially, involve fixing the communication between the FPGA and PS, as this seemed to be the problem during the initial implementation.

6.5.2 FPGA acceleration of the whole CNN using OpenVINO

Only part of the KWS CNN was accelerated on the FPGA using OpenVINO, most likely due to the reshape layer in the network. Replacing this layer, while maintaining the same functionality, could result in all of the network being accelerated on the FPGA. This would accelerate the Dense-layers, possibly resulting in a classification time lower than the Keras implementation. This would also give a more realistic result in terms of implementation, as an embedded system using an FPGA for neural network acceleration would likely not have an Intel CPU.

6.5.3 FPGA acceleration using DNNDK

In addition to HLS, no results were achieved for acceleration using DNNDK. The starting point would be to use the quantiser with a .hdf5-layer in the converted Caffe-model to load data during quantisation. An alternative would be to explore the possibilities when using a TensorFlow-model as support for TensorFlow was added in release 3.0 of the SDK.

If image data was the only method of loading data during the quantisation process, different image formats and representations could be used to see if one is maintains the data resolution as required. One of the limitations with image data is the reduced precision, as most image formats store data as 3-channel RGB data, with 8 bits of resolution for each colour. As the log-mel-filterbank energies calculated during pre-processing are negative and positive, 8-bit unsigned data used by some image formats cannot correctly represent the pre-processed data.

If acceleration was successful, further work could be done to make the pre-processing work on the FPGA to run alongside the DPU on the Ultra96.

6.5.4 FPGA acceleration of LSTM networks

As only the CNN was accelerated using OpenVINO, and LSTM networks were not supported by DNNDK, the LSTM network could be re-written using MXNet utilising only supported MXNet-layers. Though the network would not be a replica, the main functionality might be kept and reasonable classification results could be achieved.

Chapter 7

Conclusion

As the results and discussion show, FPGA acceleration of C/C++-code for speech recognition pre-processing code is possible, albeit somewhat difficult when the goal is to create code communicating with the FPGA and the PS. The half-precision FFT-code was able to run properly but was too inaccurate to provide useable data. The single-precision FFT-code was promising, with an average computation accuracy of 99.86 when compared with the double-precision C++-implementation, with an estimated runtime of 26 ms. Given more time and help from an FAE at Xilinx, the code could be integrated successfully on the Ultra96, though this illustrates a problem with using Vivado HLS and the associated tools to perform intra-MPSoC communication between the FPGA and PS: It is difficult when not in direct contact with support, as the freely available resources are not sufficient to quickly develop the desired application. Still, if the code is successfully deployed on the Ultra96, it will most likely not be able to run alongside the DNNDK DPU as the DPU most likely uses most if the resources on the device, leaving little to no resources for the pre-processing to use.

The three neural networks were trained successfully using the pre-processed data; however they were not able to achieve the same accuracies as achieved by de Andrade et al. [17]. This might have been a result of different input data, and also illustrates that neural network accuracy is severely dependent on input data. Still, the networks

achieved reasonably good results, though somewhat surprising. The regular LSTM network performed worse than the CNN, while also having longer classification times. The classification times also illustrate a problem with LSTM layers: Though capable of achieving results better than CNNs, they require substantially more computational power. This was proved when running the LSTM layers on the CPU, resulting in a median classification time 21 times higher than the GPU-accelerated version, and 226 times higher than the CPU deployed CNN. While LSTM networks are promising for speech recognition purposes, CNNs are not far behind in terms of classification accuracy while still ahead in terms of classification speed.

While the three networks were trained and deployed on a CPU and GPU successfully, only the CNNs was able to be accelerated on an FPGA. The results achieved using FPGA acceleration were worse than the ones achieved using a relatively old CPU, but the results were most likely limited by outdated hardware. Comparing layer-by-layer, the FPGA showed promising results, and if the network can be adjusted to run primarily on the FPGA, even better results can be achieved.

Acceleration of LSTM networks was limited by the layers supported by OpenVINO, while acceleration using DNNDK was limited by the quantisation requirements. OpenVINO reveals itself as a more mature platform with better documentation and framework support, but DNNDK shows promising results despite its limitations. As more and more users adopt DNNDK for FPGA acceleration of neural networks, and Xilinx creates more user guides and examples, DNNDK will become a powerful contender to OpenVINO.

The goal of this thesis was to examine the FPGA acceleration possibilities for three speech recognition neural networks, and the associated pre-processing code, using acceleration toolkits such as Intel OpenVINO, Xilinx DNNDK and Vivado HLS. This is possible, achieving reasonable results without requiring any HDL to be programmed, but somewhat difficult when relying mostly on freely available resources with little support from the software manufacturers. Still, the results are promising, especially considering the focus the developers have on improving the tools and programs in question, in addition to the sinking cost of entry, with the Ultra96 leading the charge at $250.

References

[1] A. Samuel, "Some studies in machine learning using the game of checkers," *IBM Journal of Research and Development*, vol. 3, 1959.

[2] A. M. TURING, "I.—COMPUTING MACHINERY AND INTELLIGENCE," *Mind*, vol. LIX, no. 236, pp. 433–460, Oct. 1950. eprint: `http://oup.prod.sis.lan/mind/article-pdf/LIX/236/433/9866119/433.pdf`. [Online]. Available: `https://doi.org/10.1093/mind/LIX.236.433`.

[3] A. L. Samuel, "Some studies in machine learning using the game of checkers," *IBM Journal*, vol. 3, no. 3, Jul. 1959.

[4] S. J. Lighthill and N. S. Sutherland, *Artificial Intelligence: a paper symposium.* Science Research Council, 1973.

[5] R. Collobert, S. Bengio, and J. Mariéthoz, "Torch: A modular machine learning software library," IDIAP, 1920 Martigny, Switzerland, Research report 02-46, 2002.

[6] Google Brain Team, *Tensorflow*, 2017. [Online]. Available: `https://www.tensorflow.org/`.

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

[8] K. H. Davis, R. Biddulph, and S. Balashek, "Automatic recognition of spoken digits," *The Journal of the Acoustical Society of America*, vol. 24, no. 6, pp. 637–642, 1952.

[9] S. E. Levinson, L. R. Rabiner, and M. M. Sondhi, "An introduction to the application of the theory of probabilistic functions of a markov process to automatic speech recognition," *The Bell System Technical Journal*, vol. 62, no. 4, pp. 1035–1074, Apr. 1983.

[10] S. K. Das and M. A. Picheny, "Issues in practical large vocabulary isolated word recognition: The ibm tangora system," in *Automatic Speech and Speaker Recognition: Advanced Topics*, C.-H. Lee, F. K. Soong, and K. K. Paliwal, Eds. Boston, MA: Springer US, 1996, pp. 457–479. [Online]. Available: `https://doi.org/10.1007/978-1-4613-1367-0_19`.

[11] J. G. Wilpon and D. B. Roe, "At&t telephone network applications of speech recognition," in *Proceedings from the COST232 Workshop*, Nov. 1992.

[12] A. L. Gorin, G. Riccardi, and J. H. Wright, "How may i help you?" *Speech communication*, vol. 23, no. 1-2, pp. 113–127, 1997.

[13] P. Warden, "Speech commands: A dataset for limited-vocabulary speech recognition," *arXiv preprint arXiv:1804.03209*, 2018.

[14] G. Brain, *Tensorflow speech recognition challenge*, 2018. [Online]. Available: `https://www.kaggle.com/c/tensorflow-speech-recognition-challenge/`.

[15] B. McMahan and D. Rao, "Listening to the world improves speech command recognition," *CoRR*, vol. abs/1710.08377, 2017. arXiv: `1710.08377`. [Online]. Available: `http://arxiv.org/abs/1710.08377`.

[16] F. Yu and V. Koltun, "Multi-scale context aggregation by dilated convolutions," *arXiv preprint arXiv:1511.07122*, 2015.

[17] D. C. de Andrade, S. Leo, M. L. D. S. Viana, and C. Bernkopf, "A neural attention model for speech command recognition," *arXiv preprint arXiv:1808.08929*, 2018.

[18] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," *arXiv preprint arXiv:1409.0473*, 2014.

[19] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in neural information processing systems*, 2017, pp. 5998–6008.

[20] A. Nilsen, K. Svarstad, and F. Bochud, "Fpga acceleration of neural network voice recognition," Semester project, Norwegian University of Science and Technology, 2018. [Online]. Available: `https://github.com/andernil/OpenVINO_project/blob/master/Hey_Spark_OpenVINO/Project_Thesis_Voice_Recognition_Cisco_AN.pdf`.

[21] D. Lin, S. Talathi, and S. Annapureddy, "Fixed point quantization of deep convolutional networks," in *International Conference on Machine Learning*, 2016, pp. 2849–2858.

[22] P. Colangelo, N. Nasiri, A. Mishra, E. Nurvitadhi, M. Margala, and K. Nealis, "Exploration of Low Numeric Precision Deep Learning Inference Using Intel FPGAs," *arXiv preprint arXiv:1806.11547*, 2018.

[23] Khronos Group, *Opencl*, 2008. [Online]. Available: `https://www.khronos.org/opencl/`.

[24] Community, *OpenCL Caffe*, 2018. [Online]. Available: `https://github.com/BVLC/caffe/tree/opencl`.

[25] Intel, *Computer Vision Hardware*, Accessed 2018-12-09. [Online]. Available: https://software.intel.com/en-us/openvino-toolkit/hardware.

[26] S. Zeng, K. Guo, S. Fang, J. Kang, D. Xie, Y. Shan, Y. Wang, and H. Yang, "An efficient reconfigurable framework for general purpose cnn-rnn models on fpgas," in *2018 IEEE 23rd International Conference on Digital Signal Processing (DSP)*, Nov. 2018, pp. 1–5.

[27] S. Herculano-Houzel, "The human brain in numbers: A linearly scaled-up primate brain," *Frontiers in Human Neuroscience*, vol. 3, p. 31, 2009. [Online]. Available: https://www.frontiersin.org/article/10.3389/neuro.09.031.2009.

[28] F. Rosenblatt, "The perceptron: A probabilistic model for information storage and organization in the brain," *Psychological Review*, vol. 65, no. 6, 1958.

[29] S. Grossberg, "Contour Enhancement, Short Term Memory, and Constancies in Reverberating Neural Networks," *Studies in Applied Mathematics*, vol. 52, no. 3, pp. 213–257, [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/sapm1973523213.

[30] A. Perez-Uribe, "Artificial Neural Networks: Algorithms and Hardware Implementation," in *Bioinspired Computing Machines: Towards Novel Computational Architectures*, D. Mange and M. Tomassini, Eds. PPUR Press, 1998, ch. 11, pp. 289–316.

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

[32] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," *CoRR*, vol. abs/1502.03167, 2015. arXiv: 1502.03167. [Online]. Available: http://arxiv.org/abs/1502.03167.

[33] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[34] C. Olah, *Understanding lstm networks*, 2015. [Online]. Available: http://colah.github.io/posts/2015-08-Understanding-LSTMs/.

[35] M. Schuster and K. K. Paliwal, "Bidirectional recurrent neural networks," *IEEE Transactions on Signal Processing*, vol. 45, no. 11, pp. 2673–2681, 1997.

[36] Incfk8, *Structural diagrams of unidirectional and bidirectional recurrent neural networks*, accessed 2019-06-01, 2015. [Online]. Available: https://commons.wikimedia.org/wiki/File:Structural_diagrams_of_unidirectional_and_bidirectional_recurrent_neural_networks.png.

[37] J. Cho, K. Lee, E. Shin, G. Choy, and S. Do, "Medical image deep learning with hospital PACS dataset," *CoRR*, vol. abs/1511.06348, 2015. arXiv: 1511.06348. [Online]. Available: http://arxiv.org/abs/1511.06348.

[38] A. Alwosheel, S. van Cranenburgh, and C. G. Chorus, "Is your dataset big enough? sample size requirements when using artificial neural networks for discrete choice analysis," *Journal of Choice Modelling*, vol. 28, pp. 167–182, 2018. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1755534518300058.

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

[40] H. Shin, H. R. Roth, M. Gao, L. Lu, Z. Xu, I. Nogues, J. Yao, D. Mollura, and R. M. Summers, "Deep convolutional neural networks for computer-aided detection: Cnn architectures, dataset characteristics and transfer learning," *IEEE Transactions on Medical Imaging*, vol. 35, no. 5, pp. 1285–1298, May 2016.

[41] J. Redmon, S. K. Divvala, R. B. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," *CoRR*, vol. abs/1506.02640, 2015. arXiv: `1506.02640`. [Online]. Available: `http://arxiv.org/abs/1506.02640`.

[42] H. Sak, A. W. Senior, and F. Beaufays, "Long short-term memory based recurrent neural network architectures for large vocabulary speech recognition," *CoRR*, vol. abs/1402.1128, 2014. arXiv: `1402.1128`. [Online]. Available: `http://arxiv.org/abs/1402.1128`.

[43] R. Reed, "Pruning algorithms-a survey," *IEEE Transactions on Neural Networks*, vol. 4, no. 5, pp. 740–747, Sep. 1993.

[44] H. Li, A. Kadav, I. Durdanovic, H. Samet, and H. P. Graf, "Pruning filters for efficient convnets," *CoRR*, vol. abs/1608.08710, 2016. arXiv: `1608.08710`. [Online]. Available: `http://arxiv.org/abs/1608.08710`.

[45] M. C. McFarland, A. C. Parker, and R. Camposano, "Tutorial on high-level synthesis," in *Proceedings of the 25th ACM/IEEE Design Automation Conference*, IEEE Computer Society Press, 1988, pp. 330–336.

[46] G. Martin and G. Smith, "High-level synthesis: Past, present, and future," *IEEE Design Test of Computers*, vol. 26, no. 4, pp. 18–25, Jul. 2009.

[47] D. Palaz, M. Magimai.-Doss, and R. Collobert, "Convolutional neural networks-based continuous speech recognition using raw speech signal," in *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, Apr. 2015, pp. 4295–4299.

[48] A. Graves, N. Jaitly, and A. Mohamed, "Hybrid speech recognition with deep bidirectional lstm," in *2013 IEEE Workshop on Automatic Speech Recognition and Understanding*, Dec. 2013, pp. 273–278.

[49] Terencehonles, *Hidden markov model*, accessed 2019-04-05, 2009. [Online]. Available: https://commons.wikimedia.org/wiki/File:HMMGraph.svg.

[50] C. S. Burrus, *Fast Fourier Transforms.* Burrus-Williamson, 2008, ch. 11. [Online]. Available: https://cnx.org/contents/gua6b7go@22.1:ulXtQbN7@15/Implementing-FFTs-in-Practice.

[51] J. W. Picone, "Signal modeling techniques in speech recognition," *Proceedings of the IEEE*, vol. 81, no. 9, pp. 1215–1247, Sep. 1993.

[52] O. Niemitalo, *Window function and frequency response - hamming (alpha = 0.53836, n = 0...n)*, accessed 2019-04-07, 2013. [Online]. Available: https://commons.wikimedia.org/wiki/File:Window_function_and_frequency_response_-_Hamming_(alpha_%3D_0.53836,_n_%3D_0...N).svg.

[53] S. S. Stevens, J. Volkmann, and E. B. Newman, "A scale for the measurement of the psychological magnitude pitch," *The Journal of the Acoustical Society of America*, vol. 8, no. 3, pp. 185–190, 1937.

[54] J. Makhoul and L. Cosell, "Lpcw: An lpc vocoder with linear predictive spectral warping," in *ICASSP '76. IEEE International Conference on Acoustics, Speech, and Signal Processing*, vol. 1, Apr. 1976, pp. 466–469.

[55] *Mel frequency cepstral coefficient (mfcc) tutorial*, accessed 2019-05-09. [Online]. Available: http://practicalcryptography.com/miscellaneous/machine-learning/guide-mel-frequency-cepstral-coefficients-mfccs/.

[56] *UG871 Vivado Design Suite Tutorial - High-Level Synthesis*, v2018.3, Xilinx, Dec. 2018.

[57] H. Fayek, *Speech processing for machine learning: Filter banks, mel-frequency cepstral coefficients (mfccs) and what's in-between*, accessed 2019-05-09, 2016. [Online]. Available: https://haythamfayek.com/2016/04/21/speech-processing-for-machine-learning.html.

[58] *UG902 Vivado Design Suite User Guide - High-Level Synthesis*, v2018.3, Xilinx, Dec. 2018.

[59] *PG109 LogiCORE IP Product Guide - Fast Fourier Transform v9.0*, Xilinx, Oct. 2017.

[60] *Install intel® distribution of openvino™ toolkit for linux with fpga support*, accessed 2019-05-24, 2019. [Online]. Available: https://docs.openvinotoolkit.org/2019_R1.01/_docs_install_guides_installing_openvino_linux_fpga.html.

[61] K. Choi, D. Joo, and J. Kim, "Kapre: On-gpu audio preprocessing layers for a quick implementation of deep neural network models with keras," *CoRR*, vol. abs/1706.05781, 2017. arXiv: 1706.05781. [Online]. Available: http://arxiv.org/abs/1706.05781.

Acronyms

ANN Artificial Neural Network. 11, 17, 18

API Application Programming Interface. 3, 13, 15

ASIC Application Specific Integrated Circuit. 15

att-LSTM Attentive LSTM. iii, v, xiv, xv, 9, 48, 50, 60, 62, 63, 67, 68, 76

BRNN Bidirectional Recurrent Neural Network. xiii, 24

CNN Convolutional Neural Network. iii, v, xii, xiv–xvi, 6, 8, 9, 13–15, 27, 28, 50, 52, 53, 59–61, 63, 64, 66, 67, 69–71, 76–78, 82, 83, 86, 100, 116–119

CPU Central Processing Unit. xii, xv, 1, 3, 10, 12, 13, 31, 40, 46, 49, 50, 55, 58, 63–65, 74, 77–79, 81, 82, 86, 100

DCT Discrete Cosine Transform. 38, 75

DFT Discrete Fourier Transform. xiv, 31–36, 42

DNN Deep Neural Network. 1, 18

DNNDK Deep Neural Network Development Kit. 4, 14, 15, 75, 79, 80, 82, 83, 85, 86

DPU Deep-learning Processor Unit. 15, 75, 83, 85

DSP Digital Signal Processor. 12, 75

FAE Field Applications Engineer. 75, 85

FFT Fast Fourier Transform. xiv, 12, 16, 31–33, 35, 37, 39–47, 56, 57, 73–75, 81, 85

FPGA Field-Programmable Gate Array. iii–vi, xii, xv, 1–5, 10–13, 15, 29, 31, 39, 40, 44, 46, 50, 52, 55, 56, 58, 59, 63–65, 73–75, 78–80, 82, 83, 85, 86, 100

GPU Graphical Processing Unit. 6, 10, 12, 26, 48, 49, 58, 62, 63, 77, 86

HDL Hardware Description Language. iii, v, 11, 13, 15, 16, 29, 44, 46, 80, 86

HLS High-level Synthesis. iii, xiv, 4, 5, 11, 13, 15–17, 29, 39, 44–46, 75, 82

HMM Hidden Markov Model. xiii, 1, 6, 30–32, 34

IP Intellectual Property. 39, 45

KWS Keyword Spotting. xv, xvi, 2, 3, 7, 10, 65–71, 82, 116–119

LSTM Long Short-Term Memory. iii, v, xiii–xv, 9, 14, 23, 24, 28, 48–50, 53, 60–62, 65, 66, 76, 77, 83, 86

LUT Lookup Table. 75

MAC Multiply–Accumulate Operation. 21, 22

MCU Microcontroller Unit. 1, 11, 15

MFCC Mel-Frequency Cepstrum Coefficient. 11, 31, 37, 38, 75

MPSoC Multi-Processor System-on-Chip. 15, 39, 85

OpenCL Open Computing Language. 12

PCM Pulse-Code Modulated. 30

PS Processor System. xiv, 3, 15, 39, 40, 45, 46, 56, 74, 75, 82, 85

RNN Recurrent Neural Network. xiii, 6, 15, 23, 24, 27, 28

RTOS Real-Time Operating System. 81

SDK Software Development Kit. iv, 3, 4, 14, 15, 46, 50, 52, 54, 75, 79, 80, 82

STFT Short-Time Fourier Transform. 32–34, 36

VPU Vision Processing Unit. 12

Appendix A

Inference Engine per-layer execution times

Table A.1: Execution times for each layer of the CNN on CPU and single-precision- and half-precision FPGA. Dashed entry means that the layer was not run on the inference target

	Execution times [us]				
	CPU	**FPGA SP**		**FPGA HP**	
Layer	FP32	FP16	FP11	FP16	FP11
Conv2d_1	312	-	-	-	-
Conv2d_2	1397	-	-	-	-
Conv2d_3	2449	-	-	-	-
Preprocessing	-	107	117	122	112
To DDR	-	675	565	609	634
FPGA Execute time	-	693	466	887	520
From DDR	-	538	336	304	301
FPGA Post-processing	-	0	0	0	0
copy to IE blob	-	752	224	253	435
dense1	3338	3023	1436	1615	1550
dense1_activation	10	7	6	6	6
dense2	9	7	7	6	6
dense2_activation	2	2	3	2	2
dense3	3	3	3	3	3
dense3_activation	8	7	7	7	6
max_pooling_2d_1	413	-	-	-	-
max_pooling_2d_2	85	-	-	-	-
max_pooling_2d_3	89	-	-	-	-
max_pooling2d_3_nchw8c_nchw_flatten_1	98	-	-	-	-
Flatten_1	-	-	-	-	-
out_dense_3	-	-	-	-	-
conv2d_1_activation	-	-	-	-	-
conv2d_2_activation	-	-	-	-	-
conv2d_3_activation	-	-	-	-	-
Total	8207	5814	3170	3814	3575

Appendix B

Precision and recall curves for the 12 classification words

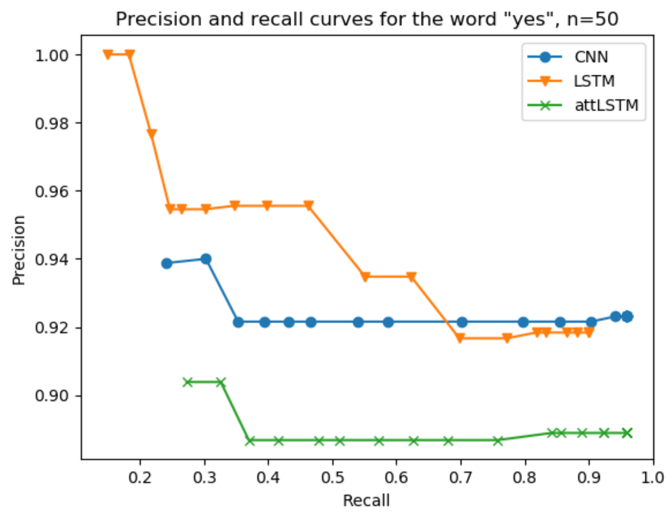


Figure B.1: Precision and recall curve for the word "yes" for all networks

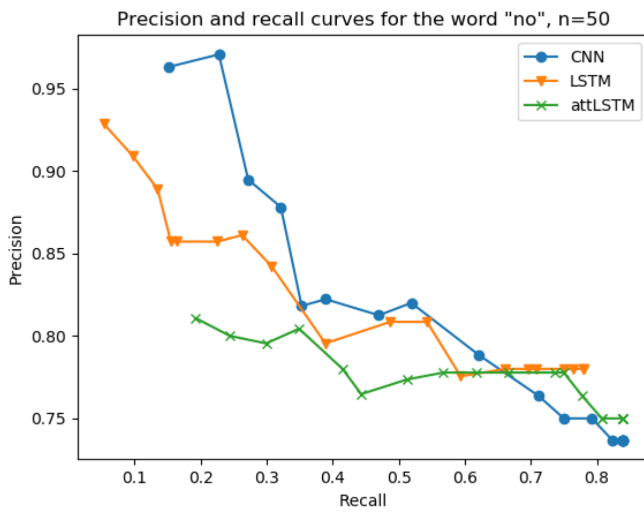


Figure B.2: Precision and recall curve for the word "no" for all networks

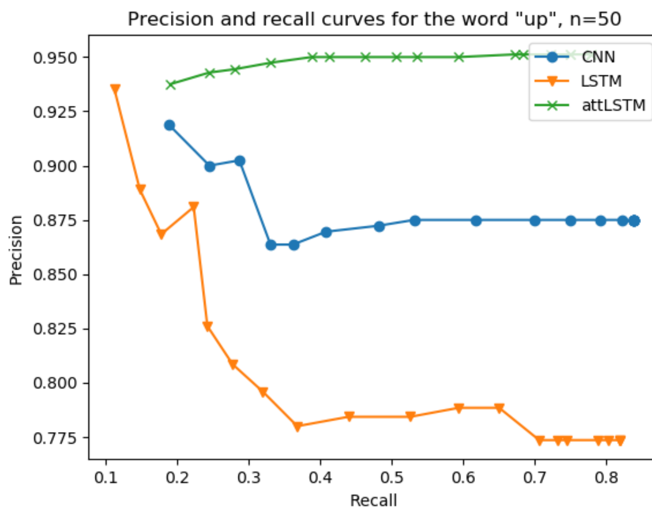


Figure B.3: Precision and recall curve for the word "up" for all networks

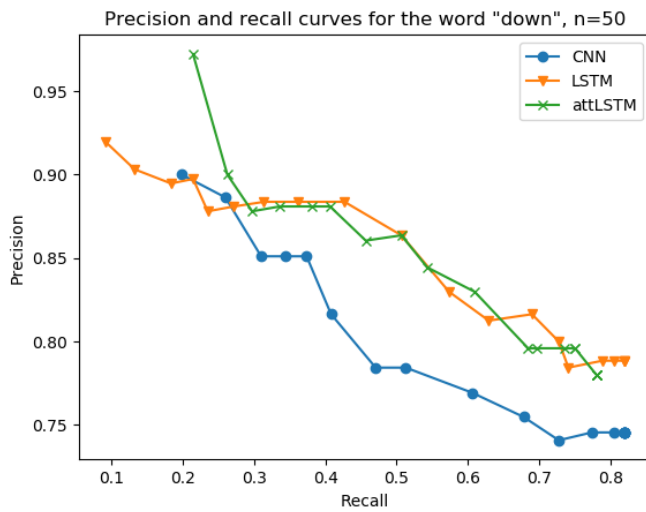


Figure B.4: Precision and recall curve for the word "down" for all networks

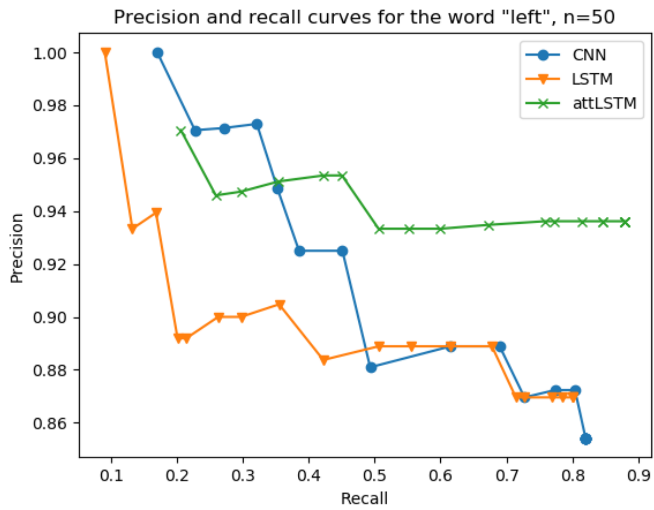


Figure B.5: Precision and recall curve for the word "left" for all networks

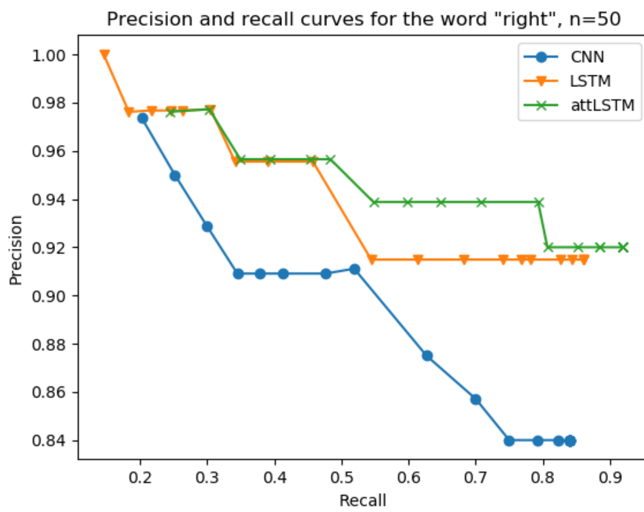


Figure B.6: Precision and recall curve for the word "right" for all networks

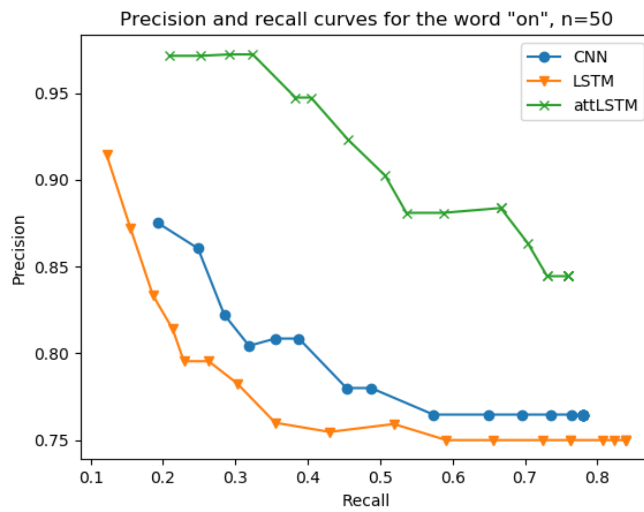


Figure B.7: Precision and recall curve for the word "on" for all networks

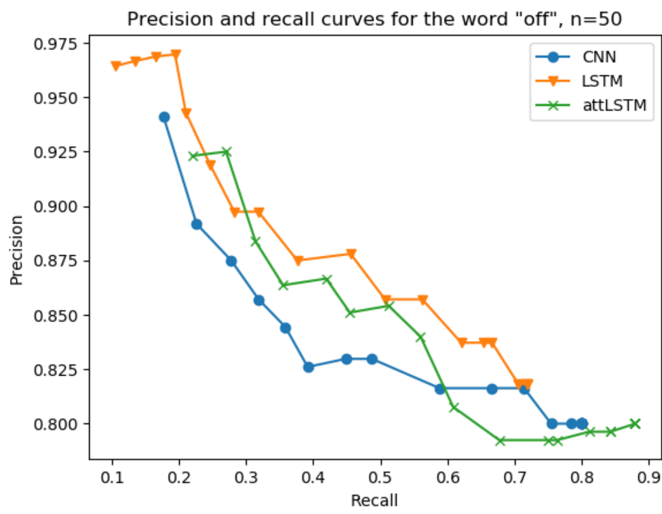


Figure B.8: Precision and recall curve for the word "off" for all networks

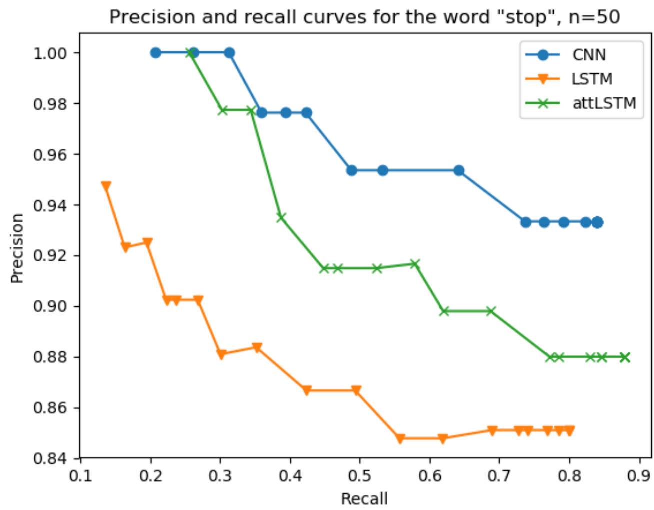


Figure B.9: Precision and recall curve for the word "stop" for all networks

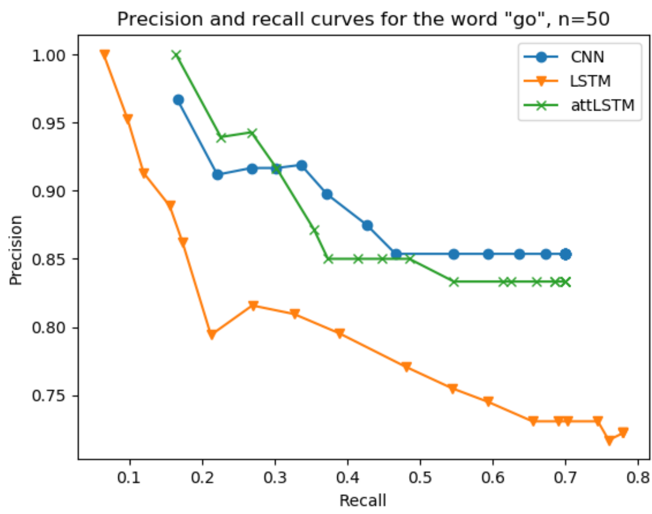


Figure B.10: Precision and recall curve for the word "go" for all networks

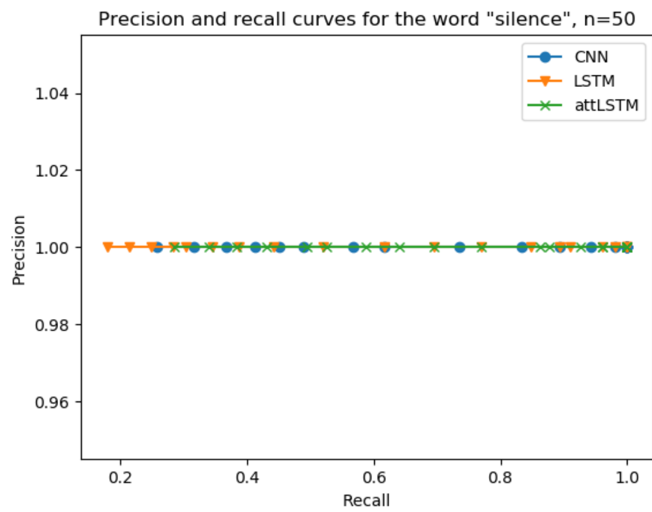


Figure B.11: Precision and recall curve for the "silence" for all networks

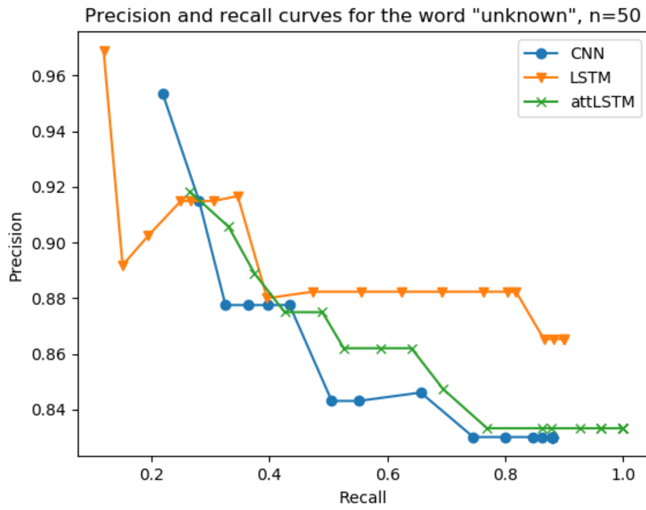


Figure B.12: Precision and recall curve for the unknown word ("marvin)" for all networks

Appendix C

Confusion matrices for FPGA-inferred CNN networks using OpenVINO

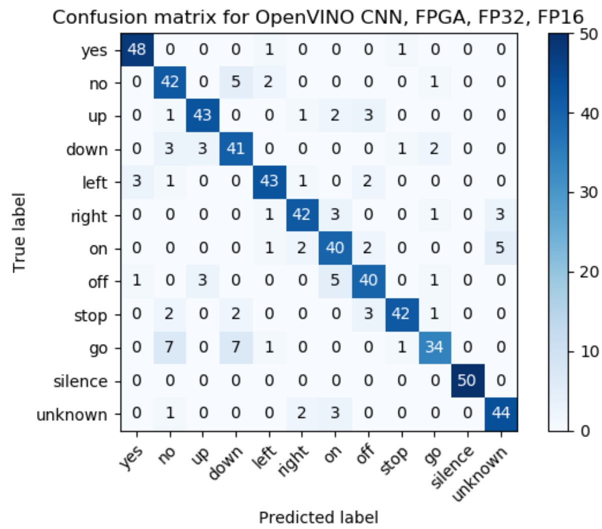


Figure C.1: Confusion matrix for single-precision KWS CNN inferred on half-precision bitstream

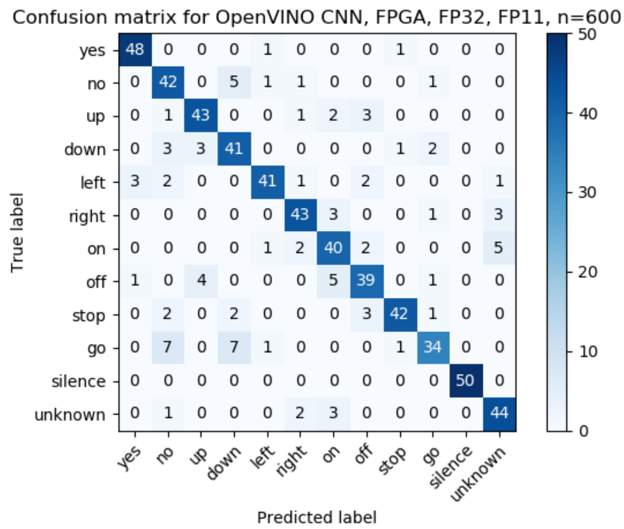


Figure C.2: Confusion matrix for single-precision KWS CNN inferred on 11-bit precision bitstream

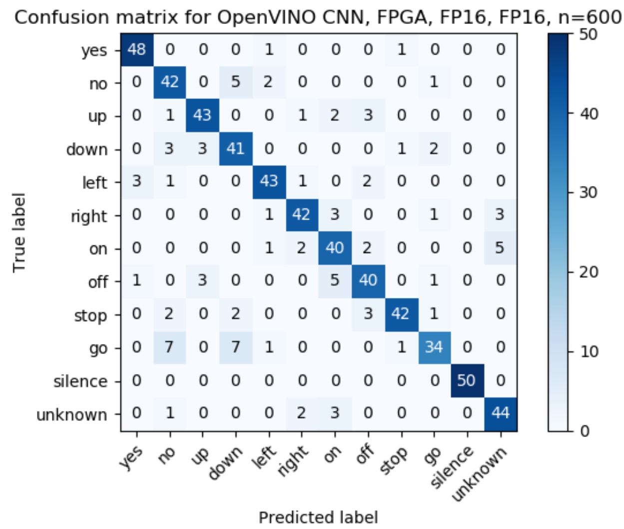


Figure C.3: Confusion matrix for half-precision KWS CNN inferred on half-precision bitstream

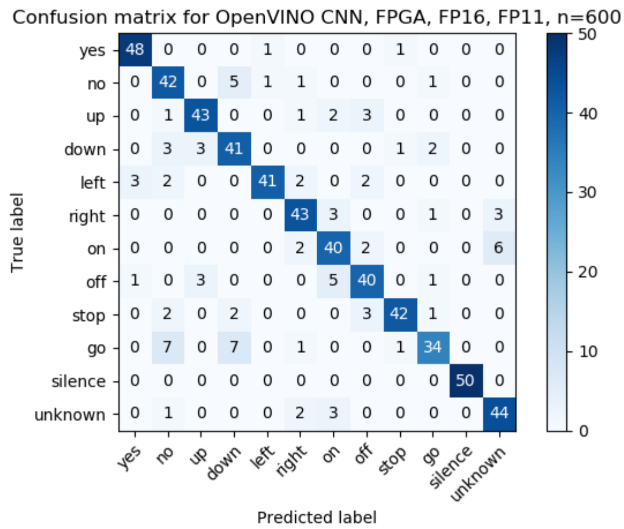
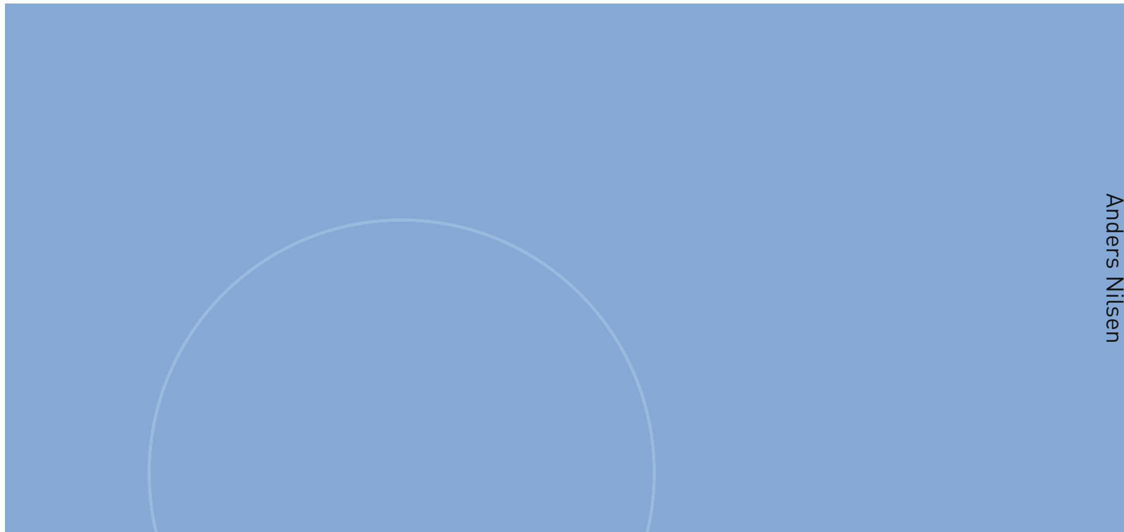
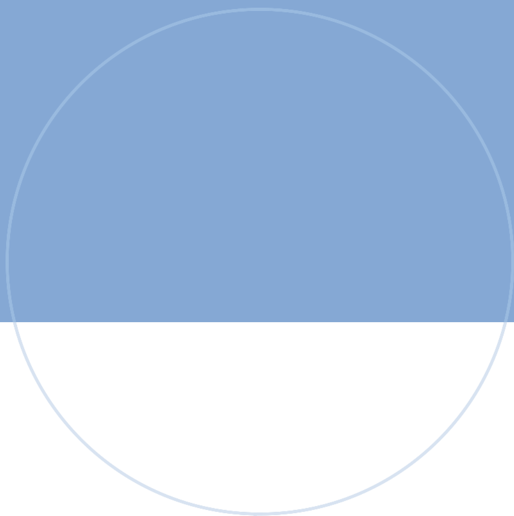


Figure C.4: Confusion matrix for half-precision KWS CNN inferred on 11-bit precision bitstream





Anders Nilsen

Accelerating keyword spotting neural networks on FPGAs using Intel OpenVINO and Xilinx DNNDK

NTNU
Kunnskap for en bedre verden