

8-2-2024

Development and Optimization of a 1-Dimensional Convolutional Neural Network-Based Keyword Spotting Model for FPGA Acceleration

Trysten E. Dembeck
Grand Valley State University

Follow this and additional works at: <https://scholarworks.gvsu.edu/theses>



Part of the [Artificial Intelligence and Robotics Commons](#)

ScholarWorks Citation

Dembeck, Trysten E., "Development and Optimization of a 1-Dimensional Convolutional Neural Network-Based Keyword Spotting Model for FPGA Acceleration" (2024). *Masters Theses*. 1129.
<https://scholarworks.gvsu.edu/theses/1129>

This Thesis is brought to you for free and open access by the Graduate Research and Creative Practice at ScholarWorks@GVSU. It has been accepted for inclusion in Masters Theses by an authorized administrator of ScholarWorks@GVSU. For more information, please contact scholarworks@gvsu.edu.

Development and Optimization of a 1-Dimensional Convolutional Neural Network-Based
Keyword Spotting Model for FPGA Acceleration

Trysten Edward Dembeck

A Thesis Submitted to the Graduate Faculty of

GRAND VALLEY STATE UNIVERSITY

In

Partial Fulfillment of the Requirements

For the Degree of

Master of Science

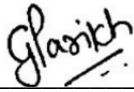
School of Engineering

August 2024

Thesis Approval Form



The signatories of the committee below indicate that they have read and approved the thesis of Trysten E. Dembeck in partial fulfillment of the requirements for the degree of Master of Science in Engineering.



07-12-2024

Dr. Chirag Parikh, Thesis committee chair

Date



7-12-24

Dr. Denton Bobeldyk, Committee member

Date



7-15-24

Dr. Samhita Rhodes, Committee member

Date

Accepted and approved on behalf of the
Padnos College of Engineering and Computing



Dean of the College

7/24/2024

Date

Accepted and approved on behalf of the
Graduate Faculty



Dean of The Graduate School

8/2/2024

Date

Dedication

To my mother, whose encouragement, support, and drive inspired me to strive for the highest levels of success in all my endeavors.

To my late father, who was almost more excited than I was for me to complete my degree program, and who took me on my very first tour of Grand Valley State University's engineering campus before I even enrolled as a first-year student. I have never known someone prouder of me than you were.

To my beloved family—your collective love, support, and encouragement have been some of my greatest strengths throughout this journey.

To my friends, who I may not have been able to see very often, but who also supported me to their fullest over the years and provided me with so much needed comfort and joy.

To my fiancé and life partner, Emilie, who served as my motivation and foundation throughout this endeavor. I am better in every way because of you, and I am so grateful for your help and support in all facets of my life while I was working on my academics and particularly this thesis. I do not know how I could have done it without you.

Acknowledgements

I would like to acknowledge the contributions of my graduate committee to my education, to my personal and professional development, and to their contributions to this thesis.

Dr. Chirag Parikh—Thank you for providing me with my foundation in FPGAs and microprocessor architectures. I would not be nearly as successful in my career without your guidance in these topics and your rigor in your courses. This foundation also served as a fundamental component of this thesis and provided me with the ability to tackle any computing issues that are presented to me. I cannot put into words the gratitude I have for your guidance through this entire process.

Dr. Denton Bobeldyk—Thank you for guiding me and teaching me throughout nearly all my computer science courses. You have been one of the greatest educators I have had the fortune to study under. Without your academic lessons and life advice, this thesis would not have been possible. You have been a massive inspiration to me, and I greatly appreciate your contributions to my academic career.

Dr. Samhita Rhodes—Thank you for being a wonderful and enthusiastic advisor throughout this process. Your courses provided me with a rigor early on in my academic career that greatly changed how I approached engineering problems, and I can with certainty attribute it to my successes.

I would also like to show my gratitude to Dr. Zachary DeBruine who originally sparked my interest in all topics related to machine learning. This thesis would not have occurred without his teachings and inspiration. Your course drastically changed the path my graduate academics took and has opened so many possibilities and opportunities for me.

Finally, I would like to thank my bosses John Videtich and Warren Guthrie for always showing an interest in my academic work, supporting me through the graduate program, and for teaching me more than I could ever have hoped to learn in class.

Abstract

Spoken Keyword Spotting (KWS) has steadily remained one of the most studied and implemented technologies in human-facing artificially intelligent systems and has enabled them to detect specific keywords in utterances. Modern machine learning models, such as the variants of deep neural networks, have significantly improved the performance and accuracy of these systems over other rudimentary techniques. However, they often demand substantial computational resources, use large parameter spaces, and introduce latencies that limit their real-time applicability and offline use. These speed and memory requirements have become a tremendous problem where faster and more efficient KWS methods dominate and better meet industry demands.

To address these challenges, this thesis presents an improved method of accomplishing the KWS task using a lightweight and efficient 1-D Convolutional Neural Network (CNN) operating on 2-D feature maps of Mel-Frequency Cepstral Coefficients (MFCCs). The model was trained using the Google Speech Commands V2 dataset, and model compression techniques such as quantization and pruning were applied to facilitate deployment onto hardware. Further minimization of inference latency was accomplished with hardware acceleration by deploying the KWS model onto a Field Programmable Gate Array (FPGA) with an open-source toolset called hls4ml. The resulting model was evaluated and compared to state-of-the-art models in literature, along with comparisons of its inference latency on different computing platforms. Finally, an application was developed to demonstrate the model running entirely on the FPGA for classifying live speech in real-time.

The developed KWS model achieved near state-of-the-art performance with far fewer parameters and a simpler architecture than comparable models in the reviewed literature. A top-one classification accuracy of 91.48% was achieved with a 30.36KB baseline model using 32-bit parameters. The baseline model was optimized and compressed to almost 50% sparsity and used 12-bit weights and activations. This compressed configuration exhibited negligible performance degradation by maintaining a top-one accuracy of 90.16% and occupying just 11.38KB of memory. These results demonstrate that 1-D CNNs are effective in accurately performing the KWS task with small parameter spaces and simple architectures. By deploying the optimized model onto FPGA hardware and running batches of samples through it, inference latencies of less than $373\mu\text{s}$ per inference, on average, were achieved indicating their usefulness in accelerating KWS models.

Table of Contents

Title Page.....	1
Approval Page.....	2
Dedication.....	3
Acknowledgements	4
Abstract	5
Table of Contents	7
List of Tables	10
List of Figures.....	11
Abbreviations.....	13
Chapter 1: Introduction	15
1.1 Spoken Keyword Spotting	15
1.2 Historical and Modern Approaches	16
1.3 Problem Statement.....	19
1.4 A Solution for Fast and Efficient KWS Models with FPGAs	19
1.5 Thesis Outline.....	21
Chapter 2: Background.....	22
2.1 Deep Spoken KWS with Neural Networks.....	22
2.2 Speech Feature Extraction.....	23

2.2.1 Human Speech Perception.....	24
2.2.2 Mel-Frequencies and MFCCs	25
2.3 1-D Convolutional Neural Networks	28
2.4 Model Compression Techniques.....	30
2.4.1 Quantization.....	30
2.4.2 Pruning.....	32
2.5 Overview of FPGA Technology	33
2.6 FPGA Deployment with hls4ml	34
Chapter 3: Review of Literature	36
3.1 Machine Learning and KWS in Literature.....	36
3.2 FPGAs as Hardware Accelerators in Literature	40
Chapter 4: Design Methodology.....	42
4.1 Dataset Selection from Google Speech Commands V2.....	42
4.2 Balancing the Training Dataset	44
4.3 Data Preprocessing and Feature Extraction	46
4.4 Model Architecture	49
4.5 Model Training	53
4.6 Model Optimization.....	54
4.6.1 Quantization-Aware Training.....	54
4.6.2 Pruning.....	55

4.7 Target FPGA Hardware.....	56
4.8 Converting KWS Model into an FPGA Block Design	57
4.9 Implementing Real-Time KWS System on the PYNQ-Z2	58
Chapter 5: Results	64
5.1 Baseline Model vs Optimized Model Performance.....	64
5.2 Performance Comparison to Related Work.....	67
5.3 FPGA Acceleration Results	68
Chapter 6: Future Work.....	72
Chapter 7: Conclusion.....	75
Appendices	77
Appendix A – Code	77
A.1 Defining Mel-Filterbank.....	77
A.2 FPGA Conversion of Quantized TensorFlow Model with hls4ml	77
A.3 Real-Time KWS Deployment Jupyter Notebook.....	78
References	80

List of Tables

Table 1: Summary of Zhang et al.'s Reviews 8-bit Quantized Neural Models [19].....	38
Table 2: Keyword Selection from Google Speech Commands V2.....	43
Table 3: Hyperparameters Considered in Cross Validation Grid Search	50
Table 4: Base Model and Optimized Model Evaluation and Parameters.....	64
Table 5: Comparison Between My Thesis' Models and Zhang et al.'s Reviewed 8-Bit Models ..	67
Table 6: Batched Model Inference Latency on Various Device Architectures for 1000 Samples.	69
Table 7: FPGA Resource Utilization of KWS Model	71

List of Figures

Figure 1: Generalized Keyword Spotting Task.....	16
Figure 2: Keyword Spotting with Neural Networks	22
Figure 3: Common Feature Extraction Pipeline	24
Figure 4: Example Mel-Filterbank.....	26
Figure 5: MFCCs from an Audio Signal	27
Figure 6: 1-D Convolutions on 1-D Data [9]	28
Figure 7: 1-D Convolution on 2-D MFCCs	29
Figure 8: IEEE 754 Single-Precision Floating Point Widths	31
Figure 9: hls4ml Design Workflow [15]	34
Figure 10: Distributions of Words in Google Speech Commands V2 and Training Datasets.....	45
Figure 11: Refined Feature Extraction and Inference Pipeline	47
Figure 12: Python Preprocessing Method	47
Figure 13: The MFCC-based 1-D CNN Model of this Thesis	51
Figure 14: TensorFlow Summary of 1-D CNN for KWS	52
Figure 15: 12-Bit Quantized Parameters.....	54
Figure 16: Converting TensorFlow Keras Layers to their QKeras Counterparts	55
Figure 17: PYNQ-Z2 Development Kit and Thesis Application. PYNQ-Z2 Screenshot in [22]..	56
Figure 18: Invoking hls4ml Package to Convert the KWS Model into an FPGA Design	57
Figure 19: Top-Level Block Diagram of Full FPGA KWS System	59
Figure 20: Expanded Neural Network IP Block.....	59
Figure 21: GPIO and Audio Codec Control Blocks.....	60
Figure 22: PYNQ-Z2 Driving Software Diagram	61

Figure 23: Example Output of Real-Time KWS System on the PYNQ-Z2.....	62
Figure 24: (Left) Confusion Matrices for Base Model (Left) and Compressed Model (Right)....	65
Figure 25: t-SNE Class Separation Visualization	66
Figure 26: Streaming KWS Application	72
Figure 27: Voice Activity Detection for KWS.....	73
Figure 28: End-to-End KWS System.....	74

Abbreviations

ASIC	Application Specific Integrated Circuit
ASR	Automatic Speech Recognition
BRAM	Block Random Access Memory
CNN	Convolutional Neural Network
CRNN	Convolutional Recurrent Neural Network
DCT	Discrete Cosine Transform
DMA	Direct Memory Access
DNN	Deep Neural Network
DS-CNN	Depth-Separable Convolutional Neural Network
DSP	Digital Signal Processing
DTW	Dynamic Time Warping
FFT	Fast Fourier Transform
FPGA	Field Programmable Gate Array
FRR	False Reject Rate
GPU	Graphics Processing Unit
GRU	Gated Recurrent Unit
GSC	Google Speech Commands
HDL	Hardware Descriptive Language
HLS	High Level Synthesis
HMM	Hidden Markov Model
IoT	Internet of Things

KWS	Keyword Spotting
LSTM	Long Short-Term Memory
LUT	Lookup Table
LVCSR	Large Vocabulary Continuous Speech Recognition
MAC	Multiply-Accumulate
MFCC	Mel Frequency Cepstral Coefficient
OOV	Out-of-Vocabulary
PL	Programmable Logic
PS	Processing System
QAT	Quantization-Aware Training
RNN	Recurrent Neural Network
RTL	Register Transfer Logic
SNR	Signal-to-Noise Ratio
SoC	System-on-Chip
STFT	Short Time Fourier Transform
TPU	Tensor Processing Unit
t-SNE	t-Stochastic Neighbor Embedding

Chapter 1: Introduction

This section first introduces and describes the spoken keyword spotting (KWS) problem and provides some of its historical solutions as well as how the field has reached its modern approaches. Subsequently, the need for highly efficient KWS systems is underscored by describing the demands of the computing systems for which they are targeted. The approaches that this thesis takes to develop a KWS system are also introduced along with the specialized deployment methods used to minimize the latency of keyword classifications.

1.1 Spoken Keyword Spotting

Spoken keyword spotting is an essential component of contemporary automatic speech recognition (ASR) systems that enables the identification of specific, single-worded keywords or commands, such as “stop” or “start,” within spoken utterances. Spoken KWS has revolutionized hands-free control over the environment and often manifests as wake-word activation applications in many consumer electronics such as smartphones, home automation devices, wearable technologies, and digital personal assistants [1].

Unlike open-vocabulary ASR systems that attempt to convert any utterance into its equivalent textual format, KWS systems use a finite vocabulary that they identify specific keywords from. KWS systems use a chosen list of words called *target words* or *in-vocabulary* words. All other words (or sounds) are referred to as *out-of-vocabulary* (OOV) and are typically classified into an additional *Unknown* category. Some KWS systems also include a *Silence* category for streams of audio where no sound is present other than background noise. Others are highly specialized and are only designed to detect a singular word while ignoring all other audio information.

The overall goal of a KWS system is to correctly identify any keywords that appear in a spoken utterance so long as the utterance contains a word that the system was designed to recognize. This generalized definition is demonstrated in Figure 1.

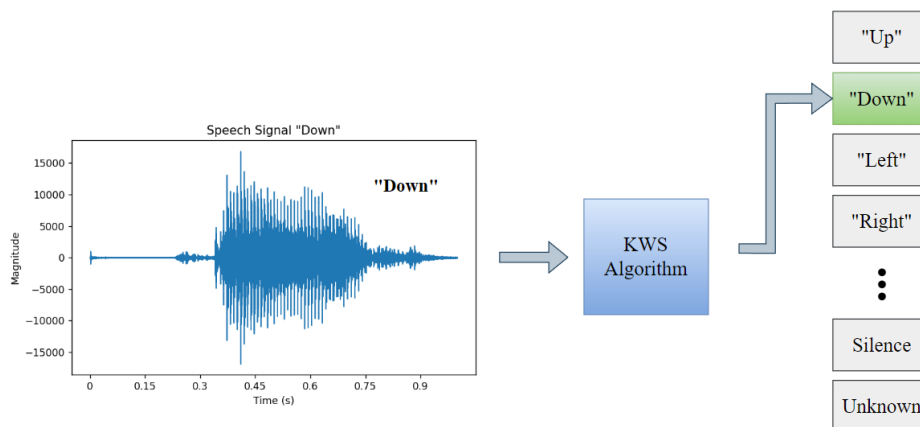


Figure 1: Generalized Keyword Spotting Task

Well-performing KWS systems maximize the number of true-positives (correctly identified keywords) and minimize the number of false-positives (keywords that were incorrectly identified despite them not appearing in the utterance) from any given audio stream. There is an emphasis on reducing false positives specifically because they can cause some action to occur unintendedly [1]. Various approaches have been applied to achieve this goal with a steady increase in overall performance as new methods are discovered and improved as described in the next section.

1.2 Historical and Modern Approaches

Historically, decision-making and pattern matching algorithms like Hidden Markov Models (HMMs) have been used to perform the KWS task by breaking the audio stream down into its phonetic features—or sounds that are perceptually distinct to humans—and generating state transitions from their specific orderings [1]. Another popular method used was Dynamic Time Warping (DTW) which utilizes a series of reference utterances that it compares to the

sample audio signal. DTW dynamically modifies the length of the reference utterances and the sample signal with respect to one another to compute similarity scores and choose the most similar reference [2]. This method effectively models the different cadences, or speeds, with which people speak—a crucial aspect of KWS systems—by lengthening and shortening the signals to determine the most comparable reference. An even more computationally complex alternative to the HMM and DTW methods is the Large Vocabulary Conversational Speech Recognition (LVCSR) approach. It uses larger and more complex methods of encoding and decoding the speech signal’s phonetic features and then performs a search across large probabilistic lattices of information to detect specific keywords [1].

Each of these methods attempts to meet the key requirement of a KWS system: generalizing to different cadences, tones, pitches, accents, and speech patterns that are unique to every individual. However, each of these methods struggle to scale to larger vocabularies and diverse voices. Observably, the HMM and DTW solutions grow in both execution time and storage size for each added keyword in the supported vocabulary. Similarly, the LVCSR method, used for transcribing multi-word utterances, is too large and complex of a solution for the smaller KWS task. Beyond these historical approaches, the application of modern deep learning methods to the KWS task has greatly improved the performance of these systems. Deep learning methods have provided models with improved abilities in generalizing to unique voices and larger vocabularies, and they have broadened the family of computing systems that they can be deployed onto.

New research focuses almost exclusively on machine learning techniques. Variants of deep neural networks have become the primary solution for their continued ability in greatly outperforming the former historical methods. Popular machine learning approaches have utilized

Recurrent Neural Networks (RNNs), Long Short-Term Memory networks (LSTMs), and Gated Recurrent Units (GRUs) for their improved abilities in learning from the strong temporal dependencies inherent to human speech [1]. Other networks primarily involve convolutional neural networks (CNNs) which are most known for their performance on image-like data. For KWS, CNNs are most applied to image-like representations of audio data in the time-frequency domain like spectrograms [2]. The exact methods of extracting more useful features from the raw speech signals vary widely among these neural network-based approaches, but some of the most popular input features for both recurrent and convolutional networks utilize this time-frequency relationship and representation of audio [1]. Combining feature extraction methods with neural networks has been the go-to technique for most modern KWS systems due to their abilities in accurately detecting keywords from larger vocabularies.

However, the increased computational complexity of deep learning-based KWS has also imposed strict resource, energy consumption, and latency restrictions for the devices on which they can be implemented. Neural network approaches often require many bytes of storage and hundreds of thousands of multiply-accumulate (MAC) operations which may need enormous amounts of computing cycles to complete [2]. These restrictions have prevented larger and more accurate KWS models from being deployed directly into embedded hardware and forces implementations to look at other methods of edge deployment such as cloud computing [1]. But, advancements in deep learning training methods have mitigated the added complexity problem by developing techniques for compressing large and complex models into a smaller footprint that are compatible with the memory and speed restrictions of embedded systems.

1.3 Problem Statement

The current state of the KWS task demonstrates that lightweight and low-latency neural network models are necessary for modern accuracy requirements and for meeting the latency and size restrictions of edge-device hardware. However, the specific neural network architectures that best balance complexity, memory footprint, and inference latency is still an evolving and debated topic in the field of KWS. The methods of deploying trained KWS models into embedded systems also vary in research, including the specific optimization and compression techniques used and the hardware architecture of the deployment systems. Solving these problems will better meet the demands of industry and modern voice-controlled Internet-of-Things (IoT) devices by improving user-experience with quick and accurate interpretations of speech.

1.4 A Solution for Fast and Efficient KWS Models with FPGAs

This thesis attempts to solve these issues by developing and training a lightweight KWS neural network and accelerating it through deployment onto a Field Programmable Gate Array (FPGA) to achieve highly accurate and ultra-low latency keyword classifications. The proposed model uses the 1-D CNN architecture and extends its research as a KWS model by utilizing the popular MFCC as its input. Similar methods have been applied in research but with bulky 2-D CNN architectures and moderate accuracies. As a result, this thesis aims to significantly reduce model size and complexity while maintaining competitive classification accuracy. In addition, the 1-D CNN of this thesis was iteratively improved through a cross-validation grid search of its hyperparameters to determine the most performant configuration of its architecture while remaining within the model size restrictions of the target deployment platform.

Even with a lightweight model, the processing limitations of the computing platforms that use KWS may still incur long latencies to complete an inference which in turn leads to poor

user experiences. Inspired by the acceleration of model training and inference of large open-vocabulary speech recognition models through parallelization with graphics processing units (GPUs), and more recently tensor processing units (TPUs), this thesis leverages FPGAs as the hardware deployment platform to determine if they are effective at overcoming latency constraints in embedded speech recognition systems [3]. While GPUs and TPUs are far too expensive in both cost and power consumption for embedded systems to utilize, FPGAs may be a more suitable alternative as they boast low costs, low power consumptions, high clock speeds, and reconfigurable computational parallelism.

Model compression techniques are a crucial component in deploying machine learning models onto the resource-limited hardware of FPGAs [4]. Therefore, this thesis utilizes both quantization and pruning techniques to enable the model’s deployment. Each of these techniques can improve resource utilization and inference latency by synthesizing a more efficient digital circuit on FPGA fabric while minimally affecting the model’s accuracy [4].

To implement neural network hardware acceleration with FPGAs, an open-source co-design tool for high-level synthesis (HLS), called hls4ml, was used to streamline the conversion of the model and its weights into an FPGA design. This approach shifted the focus of deployment from low-level hardware descriptive languages (HDL) and register transfer logic (RTL) levels to an iterative HLS design method and greatly reduced the number of latency and resource-related hyperparameters. However, hls4ml, at the time this research was conducted, supported a limited number of neural architectures. Among these are techniques like residual or skip connections that are not as easily parallelizable or synthesizable in FPGA hardware [4]. Similarly, even more advanced architectures like transformers, autoencoders, and attention-based

models (which are uncommon for the KWS task) were excluded in this research in favor of more popular, more interpretable, and better performing convolutional and recurrent style models.

To demonstrate the working solution in real-time with the developed 1-D CNN, additional software was written to deploy the KWS model onto a chosen FPGA development kit. This software enabled the classification of live audio through a user-interface on the development kit via buttons and a connected microphone. It also provided a method of graphically visualizing the output of the KWS model as it processes inputs entirely on FPGA hardware.

1.5 Thesis Outline

To describe and evaluate the solutions of this research, the rest of this thesis is structured as follows. Chapter 2 provides a background on the intricate details of the KWS task and FPGA hardware architectures to facilitate a better understanding of the problem. This includes modern preprocessing and feature extraction techniques as well as the model optimization techniques used in this thesis. Then, Chapter 3 provides a critical review of existing literature and highlights what others have done to achieve high performance KWS models. Chapter 4 proceeds to explain the design methodology of this thesis, including the design, training, and optimization of its 1-D CNN along with its conversion into an FPGA-compatible hardware design. It also describes the development of the real-time user application for classifying live speech samples. Chapter 5 subsequently evaluates the developed KWS system and discusses the results of the conducted experiments. Chapter 6 describes some areas of future work that can build onto the findings of this thesis. Finally, Chapter 7 reiterates the key findings of this thesis and addresses the broader implications of FPGA-accelerated machine learning tasks in various domains.

Chapter 2: Background

This section provides further insight into the intricacies of spoken KWS using deep learning models, FPGA technologies, and the approaches this thesis implemented to perform the KWS task. Also discussed are the preprocessing methods required for preparing audio data for use in a speech recognition model, along with the optimization techniques of quantization and pruning. The required considerations for converting a neural network into an FPGA-compatible digital circuit are also described.

2.1 Deep Spoken KWS with Neural Networks

Performing the KWS task with a deep learning technique, like neural networks, involves extending the generalized task from Figure 1 with some additional steps in the classification pipeline. Figure 2 shows the steps that are commonly found in a deep KWS system.

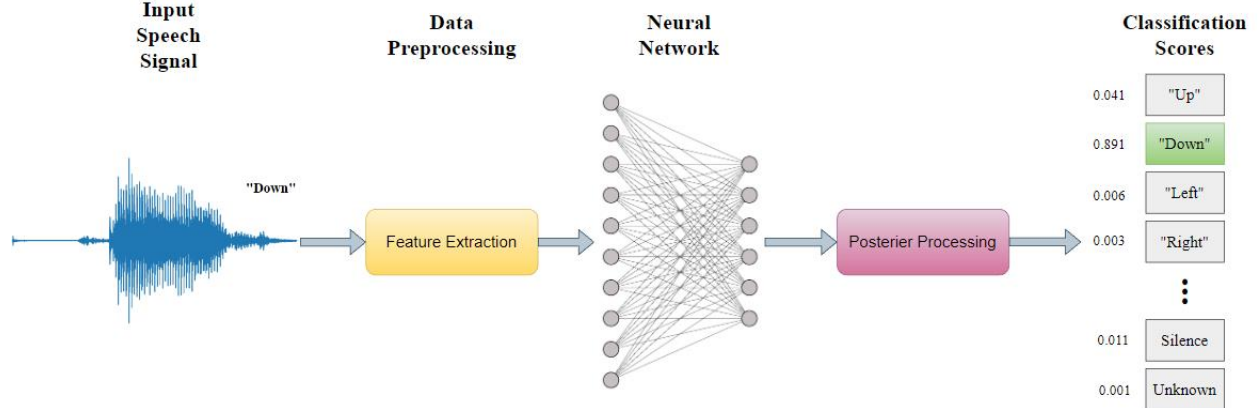


Figure 2: Keyword Spotting with Neural Networks

The feature extraction step involves specifying the data preprocessing methods that are performed on the raw audio signal before it is presented to the neural network. This step attempts to extract more useful latent features in the audio signals that the model can better learn from as opposed to the raw audio signal. The neural network itself is then handed the feature map of the

given audio signal to classify. Next, its output may have some posterior handling performed on it, such as taking the Softmax of the output to convert it to a probability distribution, before interpreting its classification scores and choosing the predicted class. Various methods of feature extraction have been combined with differing neural network architectures to determine which ones perform well for the KWS task each with differing model size and complexity considerations.

2.2 Speech Feature Extraction

Selecting the preprocessing and feature extraction methods for the audio signals going into a neural network model is a critical step in development. The output shape of the feature map determines the starting point and input shape for the model, as well as all the features that the model must learn to distinguish different inputs with. When targeting an embedded system for deployment, these preprocessing methods also need to be capable of extracting the useful latent features in audio signals, be replicable on the target deployment hardware, and be time efficient.

Common feature extraction techniques for KWS involve evaluating audio signals in the frequency domain using signal processing techniques like the Fast Fourier Transform (FFT) and Short-Time Fourier Transform (STFT) to generate spectrograms [2]. These representations provide more useful information for a model to learn from because they represent human speech through its frequency components and spectral energies which human speech and phonetics heavily rely on [1]. Generating a 2-D spectrogram representation of the audio signal also makes them easily usable with 2-D CNNs which excel at classifying image-like data. Most modern KWS models take additional steps in the frequency domain to represent the audio signals with Mel-Frequencies. The most common of these Mel-scale feature maps are the Log-Mel

spectrogram and Mel-Frequency Cepstral Coefficient (MFCC) [5]. Figure 3 shows a common feature-extraction pipeline.

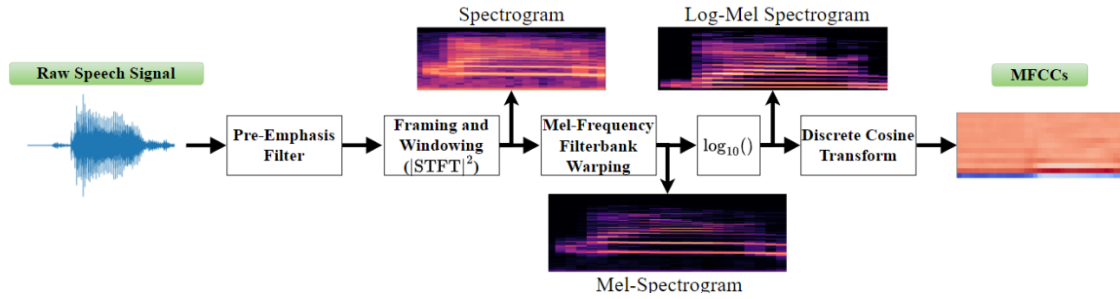


Figure 3: Common Feature Extraction Pipeline

Many applications use an initial pre-emphasis filter to remove the frequencies resulting from unwanted noise in the signal and to effectively improve the signal-to-noise ratio (SNR) [1]. Subsequently, an FFT variant, such as the STFT, is performed on the audio to translate it into the time-frequency domain while still retaining the signal’s time context. Some models have had success with using the resulting spectrogram from the STFT function, while others have been able to further improve classification accuracies by using the Log-spectrogram or the Log-Mel spectrogram [2]. However, the most performant models, in addition to the model in this thesis, utilize MFCC features that are derived from applying the Discrete Cosine Transform (DCT) on the Log-Mel spectrogram. This is due to their improved abilities in representing the specific frequency bands that humans both speak and perceive.

2.2.1 Human Speech Perception

Understanding why features like MFCCs can model human speech so well requires minor knowledge of how humans perceive sound. Human hearing involves a mixture of nonlinear processes, including power curves for loudness perception, frequency-dependent sensitivities, and the non-continuous frequency response of the ear’s basilar membrane [6]. The basilar membrane is a channel that varies in width and stiffness: it is narrow and stiff at the base and

wider and more flexible at the apex. This gradient allows it to respond to different frequencies of sound, with high frequencies causing maximum displacement near the base and low frequencies near the apex [6]. For the KWS task, the non-linear frequency response and gradient structure of the basilar membrane is of particular interest. The gradient channel is sensitive to different frequency ranges which are divided into critical bands [6]. Within each band, the ear's frequency sensitivity is relatively uniform but varies significantly between bands. These critical bands are also wider at higher frequencies and narrower at lower frequencies [6]. This means that the ear is more sensitive to changes in lower frequencies than in higher ones and is a primary reason for the use of MFCCs in audio recognition tasks because they emphasize these lower frequencies.

2.2.2 Mel-Frequencies and MFCCs

The application of the Mel scale in KWS models has significantly improved their overall performance by enhancing their ability to accurately identify and differentiate between various spoken words [7]. MFCCs have proven effective in audio classification tasks for environmental, musical, or other non-speech noises as they effectively represent the spectral information of audio signals [8]. They have also been successfully applied to speech classification tasks such as speaker recognition, emotion recognition, and language recognition [8].

MFCCs are particularly useful because Mel-Frequencies, derived from the Mel scale, are designed to mimic the human ear's nonlinear perception of sound. This scale emphasizes frequencies in a way that aligns with how humans naturally perceive speech, particularly focusing on the lower frequencies where much of the important speech information resides [8]. Additionally, MFCC features are especially useful because they retain the temporal context of speech through their extraction method which improves the ability of KWS systems to classify utterances based on how the signal changes over time [1]. The importance of a model's ability to

effectively utilize the temporal dependencies in speech is reflected in the high accuracies that RNNs, GRUs, and LSTMs have been able to achieve using the MFCC feature.

Overall, by utilizing the Mel scale, keyword spotting models can better capture the nuances of human speech, leading to more robust and precise recognition of keywords even in noisy or inconsistent acoustic environments. This alignment with human auditory perception ensures that the models are more effective in real-world applications by attenuating noise and emphasizing speech. As these are such promising features, this research focuses on an MFCC approach to training the KWS model.

To derive MFCCs, the spectral energy of a signal in the frequency domain is translated to the Mel-Frequency domain through a Mel-Filterbank. An example of a Mel-Filterbank is depicted in Figure 4.

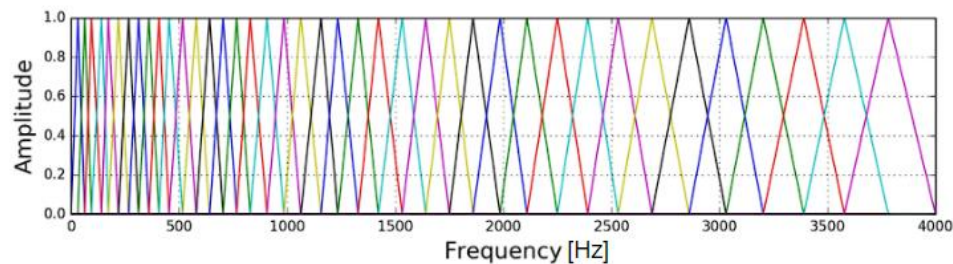


Figure 4: Example Mel-Filterbank

The Mel-Filterbank consists of a series of triangular filters that are spaced according to the non-linear Mel scale, which is more densely packed at lower frequencies and more sparsely packed at higher frequencies. In effect, this process applies a series of band-pass filters that collectively mimic a low-pass effect by focusing more on the lower frequencies and less on the higher ones. This emphasizes the lower frequency components of the spectrogram which are more critical for understanding human speech based on the ear's basilar membrane response to low frequency critical bands [7]. Multiplying a log-spectrogram by the Mel-Filterbank, as shown

in the pipeline of Figure 3, effectively reshapes the spectral information of the speech to better align with the human auditory system's non-linear sensitivity to different frequencies. By further applying the DCT to the resulting Log-Mel spectrogram, a form of dimension reduction is performed on the data while retaining its most important spectral information in the MFCCs. An example of propagating an audio signal through this pipeline to achieve the MFCCs of an utterance is demonstrated in Figure 5.

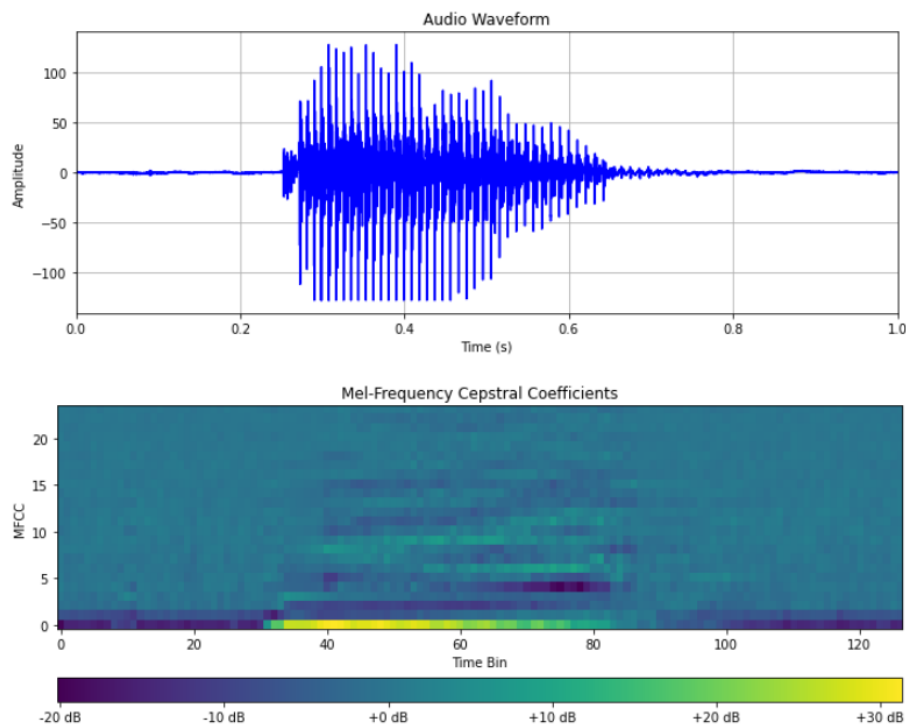


Figure 5: MFCCs from an Audio Signal

Figure 5 demonstrates a particularly useful representation of how the MFCCs are derived from a raw audio waveform. Notably, the regions of higher intensity in the MFCCs are aligned with the voice activity in the audio signal and shows how the MFCCs aggregate spectral information into short time bins. Ultimately using the 2-D MFCC feature map as the input to the 1-D CNN KWS model specifies its input shape and is the basis for the range of model sizes that can be achieved.

2.3 1-D Convolutional Neural Networks

This research focuses its network architecture on the MFCC-based 1-D CNN to further investigate their abilities in accurately capturing the strong temporal dependencies of speech.

Normally, 1-D CNNs are applied on 1-D data, as shown in Figure 6.

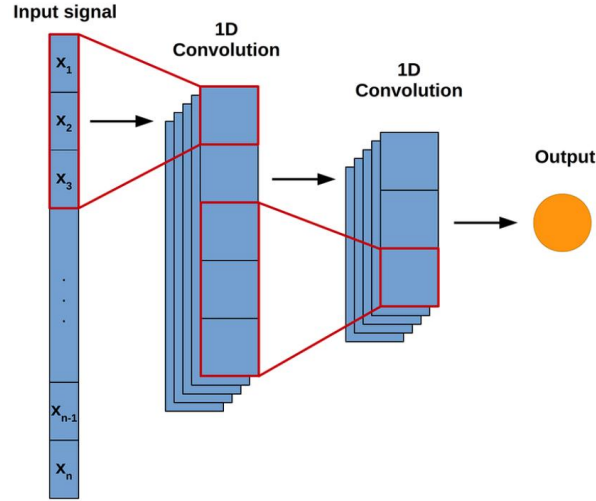


Figure 6: 1-D Convolutions on 1-D Data [9]

1-D convolution works by sliding a kernel along the input data and computing the dot product between the kernel and overlapping segments of the input at each position to produce a resulting filter [10]. Using overlapping segments gives 1-D CNNs the ability to learn short-term local temporal dependencies in its input data by aggregating speech information from distant time instances making them useful in time-series classification tasks [10]. However, 1-D convolutions can be extended to operate on 2-D data by specifying a kernel to have the same height as the input shape while having a variable width as shown in Figure 7 for the MFCC case.

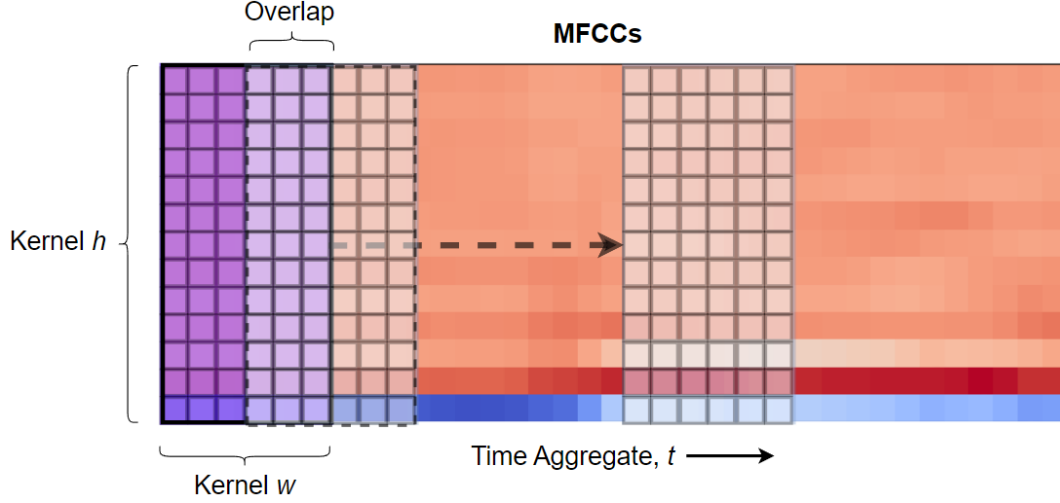


Figure 7: 1-D Convolution on 2-D MFCCs

As shown in Figure 7, the kernel uses the same height as the MFCC features but may have a variable width and step size to determine how much temporal information is aggregated by each convolution. As the width of the MFCC feature represents aggregated time frames, the sliding kernel can learn temporal dependencies by sliding along this dimension instead of focusing solely on the vertical cepstral coefficient dimension. The approach depicted in Figure 7 is the basis for MFCC-based 1-D CNN KWS models and points out some of its most pertinent hyperparameters: the kernel width, kernel height, and step size. This differs from 2-D CNNs which use a kernel that slides along images or image-like data in two dimensions. Beyond the ability of 1-D CNNs for capturing temporal dependencies, they also have other characteristics that make them more attractive than 2-D CNNs.

For example, another benefit of interest is the basic time complexity of convolution operations. For a $K \times K$ input shape being convolved by an $N \times N$ kernel, 2-D CNNs have a time complexity of $O(N^2K^2)$ while 1-D CNNs have a time complexity of $O(NK)$, making 1-D CNNs less computationally complex than 2-D CNNs and therefore providing a lower overall baseline inference latency [10]. In addition, based on a survey of 1-D CNN applications in [10], they can

provide reliable performance with very few parameters while typical 2-D CNN applications often use much larger networks and parameter spaces. 1-D CNNs are therefore especially useful for meeting the memory and latency requirements of embedded system deployment. Despite having lower computational complexity than other KWS models, compression techniques are often still required to enable the model's deployment into a resource-constrained environment.

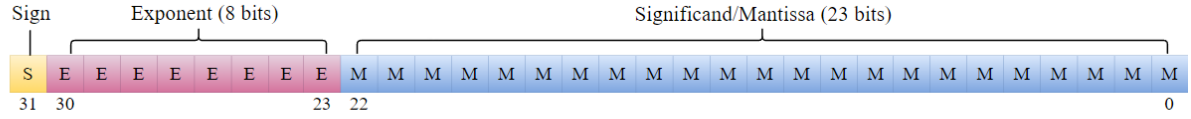
2.4 Model Compression Techniques

Machine learning models that are being deployed into embedded systems can use or require model optimization techniques so that they fit within the device's memory footprint and can run inference with sufficient latency. For the implementation of the models in this thesis, both quantization and pruning techniques were applied to the proposed KWS model to minimize its memory footprint and latency.

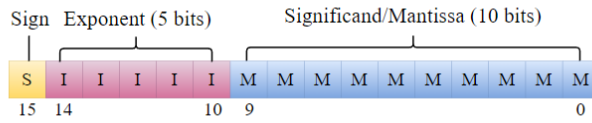
2.4.1 Quantization

During model training, the weights and activations of the model are often 32-bit signed floating-point values by default. Reducing the number of bits used to store model weights and perform their mathematical operations, known as quantization, has demonstrated significant reductions in the required memory needed for storage in embedded hardware [11]. However, quantization does reduce the range of values that model weights and activations can represent and needs to be applied carefully to take advantage of its benefits. Figure 8 shows some of the IEEE 754 specification's standard value floating-point bit-widths that are available on nearly all computing systems.

[A] IEEE 754 Single-Precision 32-bit Floating Point Value



[B] IEEE 754 Single-Precision 16-bit Floating Point Value



[C] IEEE 754 Single-Precision 8-bit Floating Point Value

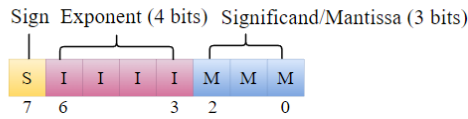


Figure 8: IEEE 754 Single-Precision Floating Point Widths

With regards to model training, quantization can be performed either during training, called quantization-aware training (QAT), or after training [11]. With QAT, the model is trained using the specified smaller bit width whereas post-training quantization truncates the trained 32-bit parameters to the specified bit width. QAT has the added benefit of being able to regain any lost accuracy through fine-tuning the model and minimizing the loss function at the newly specified bit-widths and with quantization noise present. This makes QAT the preferred method among most embedded neural network applications [12]. The KWS task has also been performed with integers instead of floating-point numbers as in [13]. This includes sizes all the way down to 8-bits which was shown to provide additional memory and latency improvements with similar performances [13]. However, [13] saw increasingly worse performances with 8-bit integer parameters for their KWS models compared to their floating-point counterparts. This was likely due to floating point numbers having the ability to represent a greater range of magnitudes, either very large or very small, depending on where the decimal is placed, whereas integers have a rigid range of magnitudes. Throughout this thesis, the increased magnitude range of floating-point numbers was suspected to allow the neural network to have more freedom in learning the

optimal numerical magnitudes necessary for learning based on the model's overall numerical regularization. In addition, the bit-widths of the weights in each layer of a model, as well as the activations of each layer, can vary within the model from layer to layer and is another hyperparameter that can be carefully tuned through cross-validation. Quantization is often paired with another compression technique called pruning to further reduce the parameter space of the model and achieve some additional benefits.

2.4.2 Pruning

Pruning is a model compression technique that aims to induce sparsity in a model's weights, neurons, filters, or layers by forcing them to zero or removing them from the architecture completely [12]. Sparsity is typically induced in a model by removing model parameters that do not have significant effects on the output or by being lower in magnitude compared to surrounding weights in a layer [14]. Pruning can be done with negligible degradation in model accuracy even when pruning 70-80% of the weights in a model making it a critical step in compressing a model for deployment in a memory-limited embedded system [14]. This also indicates that many models may use more layers and parameters than necessary to achieve their classification tasks. Similarly to QAT, pruning can be accomplished with additional training epochs to evaluate which weights do not greatly affect the output and provide fine-tuning for regaining lost accuracy [14]. This method of pruning looks at all the model's parameters and removes the weights that are significantly smaller than all the other parameters below a certain threshold. Doing this over various epochs allows the training routine to see if its current sparsity configuration greatly degrades accuracy and to try setting other weights to zero if so. Each of these methods were applied to the KWS model of this thesis. Beyond applying model

compression techniques to complete model training and development, this research also takes additional steps to deploy it onto FPGA hardware.

2.5 Overview of FPGA Technology

FPGAs represent a versatile class of digital hardware architecture that distinguishes itself by its reconfigurability and adaptability. Unlike traditional Application-Specific Integrated Circuits (ASICs), FPGAs enable developers to dynamically configure the interconnection of logic gates and other components, allowing for the implementation of custom digital circuits [15]. FPGAs consist of a grid of programmable logic blocks and programmable interconnects, which can be configured and reconfigured to perform a wide range of tasks. Such tasks can be performed with true parallelism due to the ability to have isolated regions comprising of digital circuits performing completely different functions simultaneously [15]. FPGAs are programmed using hardware description languages like Verilog or VHDL, or with high-level synthesis paradigms that use C/C++ to define the desired functionality of the digital circuit. Recent developments in FPGA design have also used their configurable nature to implement parallel compute units and accelerate large datacenters, networking stations, and search engines [15]. Their parallel computing capabilities make them especially attractive for accelerating machine learning inference.

Within an FPGA, the logic blocks themselves contain additional resources that give FPGAs their functionality. For machine learning acceleration, three resources stand out: Block RAMs (BRAMs), Digital Signal Processing (DSP) blocks, Flip-Flops (FFs), and Look-Up Tables (LUTs) [4]. The configuration of these elements is what implements specific logic and mathematical operations as a digital circuit. In addition, their configuration is made by direct electrical interconnections which significantly improves signal propagation and computation

speeds over sequential processors [4]. However, when deploying machine learning models such as the KWS model of this thesis, most model architectures are too complex and have too many parameters to store in FPGA memory units. Therefore, careful consideration of these resources, along with compression techniques, are required to ensure that the chosen architecture can be implemented.

2.6 FPGA Deployment with hls4ml

The proposed KWS model was converted into an FPGA design using the open source hls4ml package. The hls4ml package provides high-level methods for converting TensorFlow Keras models into ultra-low latency digital circuits for the target FPGA as shown in their proposed design flow in Figure 9 [16].

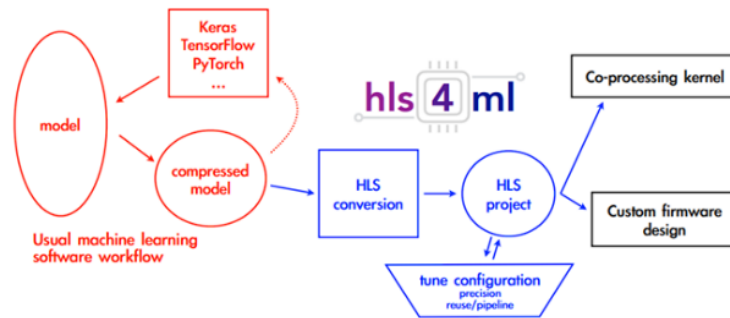


Figure 9: hls4ml Design Workflow [15]

The hls4ml framework provides numerous ways to customize the digital circuit that is generated from a neural network. These methods directly affect the latency of model inference and the utilization of important FPGA resources such as BRAM, DSPs, FFs, and LUTs [16]. Modifying these usage statistics is accomplished with hls4ml’s *precision* and *reuse* factors. Choosing the precision of each layer in the model selects the bit-width to use for all weights, activations, and multiply-accumulates in the FPGA implementation. The reuse factor determines the parallelization of the model in the FPGA fabric by altering how many times a single

multiplier (DSP block) is used in computing a layer's values. However, there is a careful balance between throughput and resource usage, particularly when modifying the reuse factor [16]. Beyond the analysis of the background information necessary to understand the KWS task, a literature review was performed to gain an understanding of what other researchers have attempted.

Chapter 3: Review of Literature

The literature review provides a critical analysis of the methods that other researchers have used to perform the KWS task with deep neural network models and provides examples of FPGAs being successful hardware accelerators. Many model architectures with a wide range of feature extraction techniques have been applied to the KWS task with vastly different resulting accuracies, model sizes, and latencies. These range from basic deep neural networks taking the raw audio signals as inputs, to RNNs and CNNs taking preprocessed feature map representations of audio data as inputs.

Due to FPGA deployment limitations restricting the implementation of modern architectures, an emphasis was placed on CNN and RNN literature. Non-machine learning methods were also omitted from the literature review due to their inability to achieve accuracies close to neural network solutions especially for systems that classify increasing numbers of keywords. Further, through the literature review, it was apparent that CNN and RNN based architectures and their variants were the most used and performant architectures in recent KWS works highlighting the importance of the temporal context of speech. Similar research was done for other applications of FPGAs being used for machine learning based hardware acceleration.

3.1 Machine Learning and KWS in Literature

Neural network techniques have been increasingly applied to the KWS task. Newer works evaluate neural network solutions and exhibit a focus on developing them for embedded system deployment. For instance, Sainath and Parada [17] compared the performance of a 2-D CNN architecture over fully connected deep neural networks. They centered their study around a 2-D CNN with which they explored different layer configurations by limiting either the total

number of multiplications in the model or the number of parameters in the model to better meet the multiplication and model size restrictions of power-constrained environments. Throughout their experiment, they developed CNN models with parameter counts between 47k and several million. They achieved relative improvements in False-Reject Rates (FRR), or the rate at which true keywords are incorrectly rejected, of 27% when shifting convolutions in frequency and 41% when pooling in time. These two models provide good improvements but are much larger than most other “small footprint” KWS models, and likely use more parameters than necessary to achieve similar accuracy if the KWS model is classifying single-word utterances. However, the experiment in [17] used longer utterances for classifying two-word pairs which likely required the substantial number of parameters to result in their well-performing FRRs. Overall, [17] provides evidence for using CNNs for the KWS task and discounts strictly fully connected neural networks for their inability to pool audio information in either frequency or time. This research also alludes to 1-D CNNs having reliable performance by sliding the kernel and pooling along the aggregated time dimension of MFCC features.

In addition, Sørensen, Epp, and May [18] provided further evidence of CNNs exhibiting high KWS classification accuracies. They extended their research to utilize the depth-separable CNN (DS-CNN) targeted for deployment on an ARM Cortex M4 microprocessor. The DS-CNN has been shown to be a more lightweight and efficient alternative to the standard CNN in small-footprint KWS due to them applying a kernel to each input channel separately and then combining the results through a single pointwise convolution instead of using kernels that operate across all channels simultaneously [18]. Through a hyper-parameter grid search, they achieved a peak accuracy of 84.7% with a 221KB model that performed inferences on the target microprocessor in 490.8ms. The accuracies listed by the work in [18] are averaged over varying

SNRs between 0-20dB suggesting that higher accuracies could be reported when a more stable acoustic environment is modeled. Still, other models have exhibited higher accuracies with smaller models that this thesis also tries to surpass with its 1-D CNN architecture. In addition, the nearly half-second inference latency is extensive and was theorized to be surpassed by a large factor with FGPA acceleration. The experiments in [18] also include an evaluation of the effects quantization had on model performance and showed that quantizing the DC-CNN model had negligible effects on accuracy but improved its memory footprint and processing speed on the microprocessor. Beyond this research, [19] performed a more extensive search across many types of neural network architectures for the KWS task and showed a different outcome with a DS-CNN architecture.

Zhang et al. [19] provides an extensive exploration of various neural network architectures on the KWS task with a focus on small, quantized models for deployment into embedded systems. The results of interest from this study were derived from models of comparable size. The model of the search in [19] is summarized in Table 1 where each model's size is reported based on its 8-bit weights and activations and were also ordered by increasing size. This list notably included an additional modification to base CNNs by adding a recurrent element to it to produce the Convolutional Recurrent Neural Network (CRNN).

Table 1: Summary of Zhang et al.'s Reviews 8-bit Quantized Neural Models [19]

Model	Accuracy (%)	Size (KB)
DS-CNN	94.40	38.60
Basic LSTM	92.00	63.30
GRU	93.50	78.80
2-D CNN	91.60	79.00
LSTM	92.90	79.50
CRNN	94.00	79.70
DNN	84.60	80.00

These results showed very high accuracies with extremely small model sizes. The most notable was the DS-CNN which had the highest accuracy at 94.40% and simultaneously the smallest size at 38.60KB showing a drastic improvement over the work in [18] and advocating for the effectiveness of CNN-like architectures at the KWS task. Many of the highest-performing models in this search employed recurrent architectures like the LSTMs, GRU, and CRNN which were described as better equipped to take advantage of long-term temporal dependencies in the input features. This indication does detract from the potential performance of the 1-D CNN proposed in this thesis due to its limitation for aggregating shorter local-time spans in the audio signal. However, this thesis continues with the 1-D CNN architecture to show promising performance with even smaller parameter spaces than shown in [19].

This literature review provided further evidence of CNN-based models exhibiting high performances when applied to the KWS task. It also provided an analysis of various other neural network architectures that have been applied to KWS, comparing their effectiveness in capturing long-term temporal dependencies. Recurrent-style architectures were highlighted for their ability to maintain and utilize information over extended periods, making them particularly effective in scenarios requiring the recognition of longer temporal patterns. Overall, the review of other neural network KWS methods resulted in plentiful evidence and motivation for further investigating 1-D CNNs and model compression techniques to achieve a high performing and very lightweight model. Further research was done to determine the effectiveness of FPGAs as hardware accelerators in machine learning problems and other applications.

3.2 FPGAs as Hardware Accelerators in Literature

FPGAs are a common solution for implementing purpose-built hardware accelerators and wherever extremely fast digital circuitry is needed. The extensive review of FPGA technology in [15] demonstrates that they are very common for accelerating data centers, search engines, network switches, and deep learning. For deep learning specifically, significant research was done in [4] to describe the optimal implementation of many neural network types in FPGA hardware. However, with HLS technologies, like Vivado HLS, any algorithm that can be described by C/C++ code can also be accelerated through synthesis into an FPGA circuit [15]. This concept was the basis for the hls4ml package in converting neural network architectures into synthesizable FPGA designs.

In fact, the hls4ml package itself, described in [16], provides significant evidence for FPGAs being useful for accelerating neural networks in other applications. The hls4ml package was originally developed for particle physics acceleration by researchers at the Large Hadron Collider who needed to quickly tag specific types of particle events and only store the types in which they were interested. In [16], the researchers also demonstrated that their 2-D CNN architecture, developed for the classical MNIST handwritten digits dataset, could be accelerated to perform inference in only 5 μ s with their hls4ml package.

For the KWS task in particular, research into KWS acceleration with FPGAs is relatively scarce. One study by Bae, Kim, Lee, and Jung [20] compiles many related works for accelerating KWS models and compares the metrics most pertinent to this thesis. In [20], various model architectures were evaluated for KWS on FPGAs and achieved a range of accuracies and latencies. The literature reviewed by [20], along with their proposed model, achieved inference latencies between 10-116ms, 10-keyword accuracies of 87.9-93%, and with models between 2-

69KB. These studies used complicated network architectures and quantized them down to two or three bits to produce much smaller networks than would normally be possible. Therefore, the accuracies and model sizes they were able to achieve with the low number of bits are highly performant compared to others in KWS literature especially for their degree of quantization. The study also placed significant emphasis on the fabrication of the KWS FPGA design into an ASIC, which furthers the draw towards FPGAs providing an important step into the development of generic speech recognition circuitry.

Chapter 4: Design Methodology

The design methodology describes how the KWS machine learning model of this study was devised, how it was trained, how it was evaluated, and how it was converted into an FPGA-compatible design. The dataset, Google Speech Commands V2, and its methods of class balancing and weighting are first described. Secondly, the preprocessing strategies used to extract the feature mappings of the dataset are outlined. Next, the chosen model architecture is described in relation to its KWS classification task along with the training methodology. Finally, the model's conversion into an FPGA-compatible design using the hls4ml package is explained. Machine learning model development is an iterative process where the specific hyperparameters used at each stage are refined through cross-validation. As such, the specific hyperparameters in each stage of development for this thesis were determined through a cross-validation grid search until the best-performing combination was found. The first step in this development process was choosing the training dataset to use as the basis for the KWS system.

4.1 Dataset Selection from Google Speech Commands V2

The initial step towards developing a deep KWS model was selecting its training dataset. The model of this thesis was trained with the Google Speech Commands V2 (GSC V2) dataset described in detail in [21] along with an analysis of its improvements over its V1 predecessor. This dataset contains 105,829 one-second-long speech files represented as little-endian 16-bit PCM-encoded WAVE files that are sampled at 16kHz. The dataset is distributed over 35 unique English words and was designed to assist in the development of speaker-independent limited vocabulary speech recognition systems [21]. The data was crowdsourced from people all over the world with different accents, cadences, and all other manners of speaking in widely variable

acoustic environments. This achieved the primary goal of [21] in gathering data that is most likely to be encountered in consumer electronics and robotics applications that should be generalizable to all people speaking a particular language.

From this dataset, 10 keywords were chosen as in-vocabulary while the other 25 were placed in an unknown class. In addition to the spoken utterances, the dataset includes longer audio files of different background noises like a whirring exercise bike, a running faucet, white noise, and others [21]. A custom program was written to segment these longer noise files into one-second audio clips for training the model to distinguish common background noises as the final silence class. Table 2 summarizes the keywords and categories that the KWS model of this thesis was designed to classify.

Table 2: Keyword Selection from Google Speech Commands V2

In-Vocabulary (Keywords)	Out-of-Vocabulary (Unknown)			Silence
down	backward	sheila	seven	washing dishes
go	bed	tree	eight	cat noises
left	bird	visual	nine	exercise bike
no	cat	wow		pink noise
off	dog	zero		running faucet
on	follow	one		white noise
right	forward	two		
silence	happy	three		
stop	house	four		
up	learn	five		
yes	marvin	six		

Each of the 10 keywords, the unknown category, and silence category resulted in the model having a 12-class output. The 10 chosen keywords were selected for being the most used across selection of words in many implementations of KWS in literature and remained common between both version of Google Speech Commands [1]. The 10 keywords were also specifically chosen to align with the research in [21] which collected them because of their usefulness in voice-controlled IoT applications. The other 25 OOV words were chosen because they cover a

wide range of phonemes, or perceptually distinct sounds, in the English language and require variable durations to utter within the one-second recordings [21]. The abundance of utterances and the diversity in word choice has made Google Speech Commands one of the most popular datasets for training KWS systems, but the way that samples are collected and balanced into a particular training dataset is another important consideration.

4.2 Balancing the Training Dataset

Having as many words with as much phonetic diversity as possible in the OOV selection is generally beneficial to the overall performance of a KWS model. However, taking the chosen dataset of Table 2 without any balancing causes there to be a significant class imbalance from there being far more samples in the unknown class than in the keyword classes. The primary concern was that generating an unbalanced training dataset could cause the model to gain a false sense of accuracy. Without balancing, this presented itself as the model reporting very high accuracies by learning to classify nearly every input as unknown strictly due to them being so common in the training dataset, and therefore performing very poorly on all the other 11 classes.

There is another key consideration in training KWS models that are designed to be always listening for keywords: most words heard by a real KWS system will most likely be unknown to it as the chosen commands do not commonly appear in normal conversation. This suggests that the system would benefit from using a training dataset consisting of a larger proportion of the unknown samples. However, this diverges from the work in [21] which provided lists of samples that would equally distribute the classes such that each of the 12 classes takes up approximately 8.3% of the training dataset. While this would produce a KWS model that can distinguish between the 12 classes, this does not take into consideration the bias in words that a real system encounters which may result in poor performance for the unknown

class. Figure 10 demonstrates how this thesis attempted to resolve this issue by showing the different distributions while using a larger distribution of samples in the unknown class.

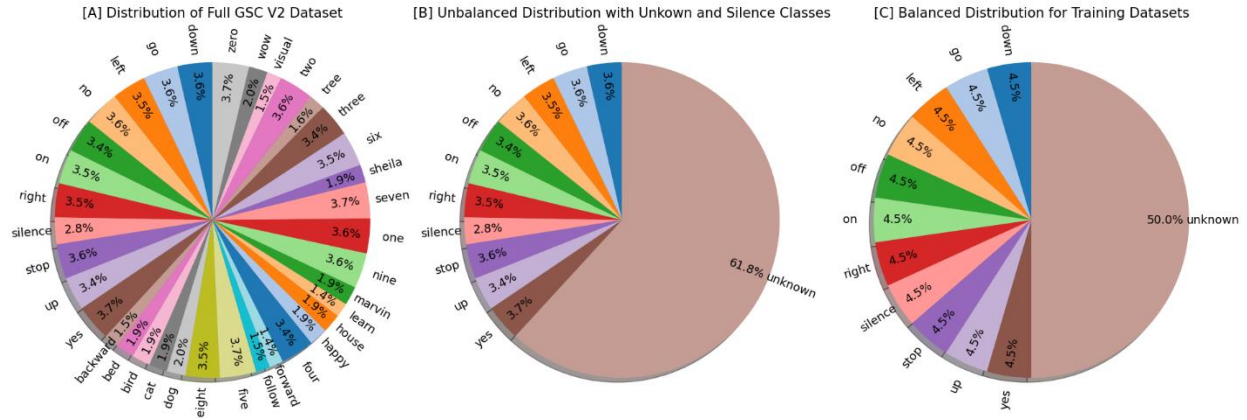


Figure 10: Distributions of Words in Google Speech Commands V2 and Training Datasets

As shown in Figure 10[A], the base dataset has roughly equal numbers of samples in each class ranging from 1.5k to 3.5k. When combining the 25 OOV words into an unknown class as in Figure 10[B] the distribution exhibits considerable imbalance. This thesis attempted to resolve this issue by ensuring that each of the 10 keywords, plus the silence class, appeared in the training dataset with equal occurrences while still using a much larger distribution for the unknown class to model a real KWS system’s deployment scenario as shown in Figure 10[C]. To further resolve this issue beyond generating a more balanced, realistic dataset, this thesis theorized that further improvements in model performance could be achieved weighting the less-frequent classes or underperforming classes in the loss function more with respect to the unknown class. Therefore, class weighting techniques were implemented by providing the underrepresented or underperforming classes more significance in reducing the loss function

based on a combination of the classes' occurrences in the dataset and empirical evidence for helping the underrepresented classes perform better.

Overall, the unknown class imposed a class imbalance issue during model training and was thus weighed the least in relation to the other classes to prevent the model from focusing on that class and having a false sense of accuracy. Using another property of the dataset, the silence class was also reduced in importance because its feature map signatures were easily distinguishable from any of the keywords, so the model was given the opportunity to focus more on the subtle distinctions between the actual words. This weighting scheme combined with class balancing provided the best performing set of trained KWS models throughout the hyperparameter grid search that were also designed to operate in a realistic deployment environment. Having the prepared dataset meant that the feature extraction methods could be refined, as well.

4.3 Data Preprocessing and Feature Extraction

With the dataset selected and prepared, its preprocessing pipeline could be developed prior to training the model. This thesis used the MFCC feature map as its input to the KWS model for their popularity and high-performance capabilities based on their ability to emphasize the intricacies of human speech. The two important caveats of defining this MFCC extraction pipeline were that it needed to be efficient and replicable on the deployment platform. Therefore, the pipeline in Figure 3 was used while discarding the intermediate spectrograms along the way, as shown in Figure 11.

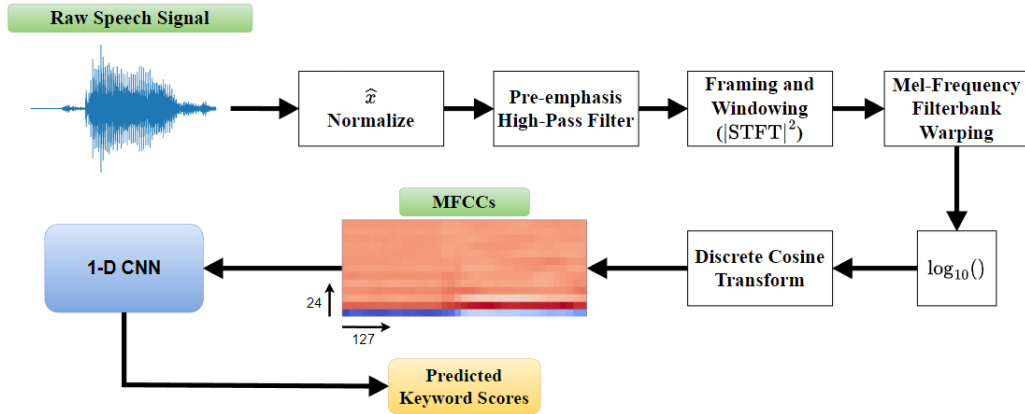


Figure 11: Refined Feature Extraction and Inference Pipeline

Notably from Figure 11, the input raw speech signals all have 16,000 datapoints as they are one-second-long audio clips sampled at 16kHz in a 16-bit PCM format. This updated data pipeline also shows the transformations that are applied to the input audio waveform to result in a predicted keyword within that utterance. As this system was primarily developed in the Python programming language, the pipeline of Figure 11 was translated into code as shown in Figure 12.

```
def extract_mfcc_features(waveform):
    # Pass audio through pre-emphasis filter
    waveform = lfilter([1, -0.97], 1, waveform)

    # Compute the STFT of the audio with the Hanning window of 16ms segments w/ 50% overlap
    _, _, stft = scipy.signal.stft(waveform, nperseg=255, noverlap=131,
                                   nfft=255, axis=1, window=get_window('hann', 255, fftbins=False),
                                   padded=False, boundary=None, scaling='psd')

    # Use the squared magnitude spectrogram
    magnitude = np.square(np.abs(stft))

    # Load and apply the Mel-Filterbank
    mel_filterbank = np.load("tf_mel_filterbank24.npy")
    mel_spectrogram = np.matmul(magnitude.T, mel_filterbank)

    # Compute the log-mel spectrogram w/ and epsilon
    log_mel_spectrogram = np.log(mel_spectrogram + 1e-6)

    # Compute MFCCs by applying the Discrete Cosine Transform
    mfccs = dct(log_mel_spectrogram, axis=1, norm='ortho')
    mfccs[0][0] *= np.sqrt(2)

    return mfccs
```

Figure 12: Python Preprocessing Method

The code performing the feature extraction during model training from Figure 12 highlights other important information that specifies the resulting shape of the MFCCs, and

therefore the input shape of the model such as the number of FFT bins and MFCCs to compute using libraries SciPy and NumPy. First, the audio waveform is normalized to be in the range of $[0, 1]$. Normalizing the input audio vector this way ensures that all audio features initially have a similar scale which prevents features with larger values from dominating the learning process and providing the model with immunity to outlier samples in the data. Normalizing the audio signal also ensured that sampling the audio with an integrated circuit or analog-to-digital converter was straightforward and independent of the data encoding scheme, whether it is 24-bit or 16-bit PCM, or another format. This was important so that the target FPGA could be selected without much concern of how the audio signals are sampled since the data will be normalized to a common scale between $[0, 1]$. Normalization can also occur after the audio signal is converted into an MFCC, but the scaling, whether 16-bit or 24-bit, adds variations in the intermediate spectrograms. Overall, normalization at the audio signal was shown to result in more efficient training, better convergence, and improved overall model performance.

Next, signal processing methods are applied to the normalized waveform. The normalized audio is passed through the pre-emphasis filter to improve its signal-to-noise ratio and amplify the high-frequency components of the signal that can be corrupted from the recording circuitry. This provides a similar effect to a high-pass filter. After pre-emphasis, the resulting signal is framed and windowed by transforming it into the spectral frequency domain using the discrete STFT. This method has the advantage of preserving the temporal context in the speech signal so that the model can learn from the strong temporal dependencies inherent to human speech based on its more useful frequency components. This research also configured the STFT to use 128 bins where each segment comprised of a 16ms window with 50% overlap per step. The Hanning windowing function was also used to reduce spectral leakage between bins, minimize edge

effects, and for being computationally efficient. In addition, the magnitude of the STFT was taken prior to squaring the result to produce a power spectrogram from the audio signal.

Furthermore, the STFT spectrogram representation was warped with the Mel-Filterbank to convert it into a Mel-Spectrogram. The Mel-Filterbank's parameters were chosen through cross-validation to use 24 Mel bins, a lower edge frequency of 20Hz, and an upper edge frequency of 7,400 Hz for a frequency resolution of 307.5Hz per bin. The programmatic way of generating this Mel-Filterbank and saving it into a file is shown in Appendix A.1. Furthermore, the base-10 logarithm of the resulting Mel-Spectrogram was taken to produce the Log-Mel spectrogram. Finally, the MFCCs were derived by applying the DCT on the Log-Mel spectrogram. This results in a 127x24 2-D feature map that the KWS model of this thesis used as its input and was the basis for the rest of the model's architecture. The 127 frames correspond to a time resolution of 7.87ms per frame and was important for accurately modeling the change in frequency over the course of each utterance.

4.4 Model Architecture

The next step of developing the KWS system involved designing and specifying the neural network model's architecture. This research utilized the 1-D CNN architecture as its basis. The shape of the MFCC features as the input to the proposed model of this study was critical for determining a model architecture that can effectively learn the relationship between the MFCCs and their associated keyword while still having a lightweight model footprint. With the input shape comfortably fixed at 127x24 and the output shape fixed at the 12 classes chosen from the Google Speech Commands V2 dataset, the remaining hyperparameters that were cross validated through a simple grid search are displayed in Table 3.

Table 3: Hyperparameters Considered in Cross Validation Grid Search

CNN Layers	Fully Connected Layers
no. convolutional layers no. filters per layer filter sizes strides padding activations pooling type/size dropout rates batch normalizations quantization bit width pruning percentage	no. fully connected layers no. neurons per layer activation function dropout rates batch normalizations quantization bit width pruning percentage

An additional caveat to consider when using Vivado HLS, the tool used to synthesize the C/C++ representation of the neural network, is that each layer must have less than 4096 parameters. This ensures that the entire layer can be completely unrolled into an equivalent FPGA circuit and have its parameters be stored in contiguous memory. As such, this study proposes a model consisting of two 1-D CNN layers and two fully connected layers as shown in Figure 13.

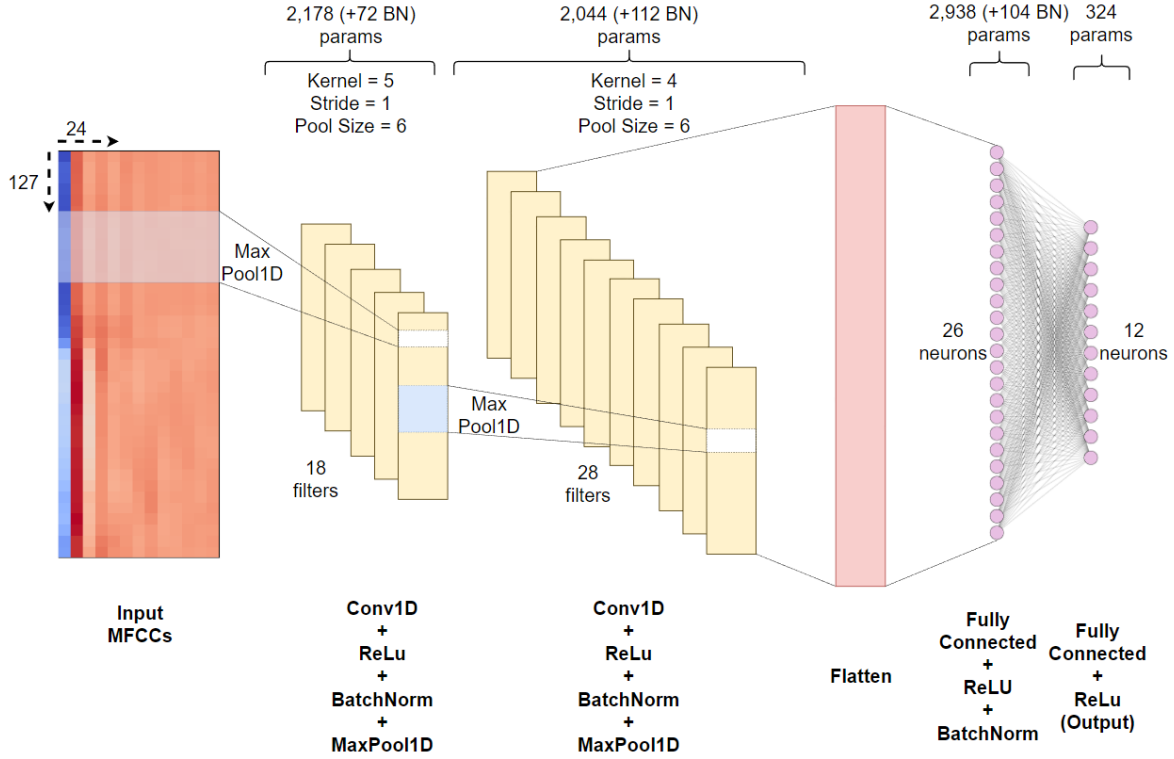


Figure 13: The MFCC-based 1-D CNN Model of this Thesis

This configuration was derived from the point in the hyperparameter grid search that best balanced classification accuracy, model size, FPGA synthesizability, and its ability to retain its accuracy after optimization with quantization and pruning. The model is first composed of the input layer which takes the 127x24 MFCCs feature map as its input. Following the input stage is a 1-D convolution (Conv1D) layer with rectified linear unit (ReLU) activations. The activations are then followed by a batch normalization layer which was implemented for numerical stability and greatly reduced training times. To complete the Conv1D block, a 1-D maximum pooling (MaxPool1D) layer was included as the dimension reduction strategy.

The first Conv1D layer had 18 filters, a convolving kernel with a width of five, and a stride of one. After the ReLu activation and BatchNorm, the MaxPool1D operation with a pooling size of six was applied to reduce the size of the feature map. The next Conv1D layer followed the same structure as the first but used 28 filters with a kernel size of four and a stride

of one. Similarly, after the ReLu activation the BatchNorm operation was applied, a MaxPool1D layer was used to further reduce the dimensions of the propagating feature map.

The flattening layer shown in Figure 13 was required to make the subsequent fully connected layers compatible with the outputs of the second Conv1D layer. Once flattened, the first fully connected layer used 26 neurons and ReLu activations. Finally, the last fully connected layer has 12 neurons, one for each of the 12 keyword classes, along with a final ReLu activation. The ReLu activation at the output was used in place of a Softmax output to minimize any unnecessary computations being done by the model, further reducing its computational latency. The TensorFlow summary of the model is shown in Figure 14.

Layer (type)	Output Shape	Param #
l1_dropout (Dropout)	(None, 127, 24)	0
l1_conv1d (Conv1D)	(None, 123, 18)	2178
l1_relu (Activation)	(None, 123, 18)	0
l1_bn (BatchNormalization)	(None, 123, 18)	72
l1_maxpool1d (MaxPooling1D)	(None, 20, 18)	0
l2_dropout (Dropout)	(None, 20, 18)	0
l2_conv1d (Conv1D)	(None, 17, 28)	2044
l2_relu (Activation)	(None, 17, 28)	0
l2_bn (BatchNormalization)	(None, 17, 28)	112
l2_maxpool1d (MaxPooling1D)	(None, 4, 28)	0
flatten (Flatten)	(None, 112)	0
l3_dropout (Dropout)	(None, 112)	0
l3_dense (Dense)	(None, 26)	2938
l3_relu (Activation)	(None, 26)	0
l3_bn (BatchNormalization)	(None, 26)	104
output_dense (Dense)	(None, 12)	324
output_relu (Activation)	(None, 12)	0
Total params: 7772 (30.36 KB)		
Trainable params: 7628 (29.80 KB)		
Non-trainable params: 144 (576.00 Byte)		

Figure 14: TensorFlow Summary of 1-D CNN for KWS

Overall, the model used 7,772 trainable parameters with each layer meeting the FPGA unwrap size restriction by having less than 4,096 parameters. Having the complete model architecture allowed for the training stage to commence with minor modifications.

4.5 Model Training

The model was trained with the TensorFlow framework and in the Python programming language. The dataset was broken into a training split of 80%, a validation split of 10%, and a test split of 10%. The test split was specifically reserved for final evaluation of the model and was never seen by the model during training or validation.

To improve model generalization to unseen data, additional dropout layers were included in between every hidden layer of the model at a 5% dropout rate to randomly force 5% of all outputs from a layer to zero. This dropout rate is lower than some other models but was strategically and empirically chosen to take advantage of the wide range of accents, cadences, pronunciations, pitches, and other speaker variabilities already present in the dataset. Additionally, since the model had so few parameters, higher dropout rates quickly degraded the model's training performance. This was exacerbated by the even smaller models that were evaluated in the grid search. To further reduce overfitting during the training process, L1 kernel regularization was implemented in every hidden layer model at a factor of 0.001 to encourage the model to learn simpler patterns in the data.

In addition, a custom learning rate scheduler was applied to half the learning rate every 10 epochs with a starting learning rate of 0.015. The training routine was set to run for 100 epochs using a batch size of 256. Finally, applying the custom class weights and shuffling the training dataset, its training commenced with TensorFlow's fit method. The model was trained with the Adam optimizer and used the sparse categorical cross-entropy loss function. Once the

base model was finished training, the model optimization and compression techniques of quantization and pruning were applied.

4.6 Model Optimization

Both pruning and quantization were applied to this thesis's 1-D CNN model to enable it to be deployed onto FPGA hardware. This further reduced the size of the model but was done carefully to select the lowest bit width and highest pruning ratio that did not degrade the model's accuracy by much. These were two additional hyperparameters that were evaluated in the cross-validation grid search. In all cases, quantization was performed on the model prior to pruning it.

4.6.1 Quantization-Aware Training

Deploying the model through hls4ml allows for custom, non-standard bit widths to be used in a way that the synthesized digital circuit can utilize it without needing any additional padding or processing [4]. It also allows for every layer in the model to be synthesized into a digital circuit with different bit widths [4]. For simplicity and ease of tracing, this thesis quantized each hidden layer and its activation to use 12 bits consisting of five integral bits and six fractional bits with 1-bit reserved for the sign bit. This custom quantized bit width is shown in reference to the base model's 32-bit width in Figure 15.

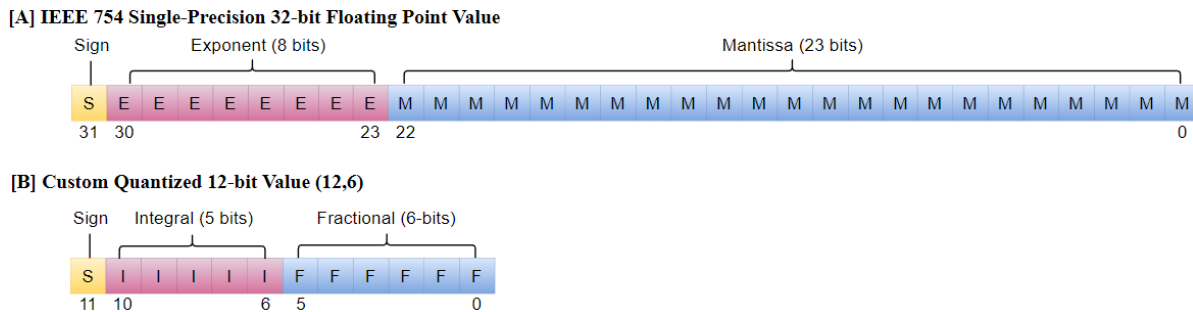


Figure 15: 12-Bit Quantized Parameters

This new bit width is 2.667 times smaller than the baseline model's 32-bit width which in turn reduces the model's size by 2.667. To implement this, the model was converted to its quantized version TensorFlow through the QKeras (quantized-Keras) module. Thus, Conv1D layers were replaced with QConv1D layers and Dense layers were replaced with QDense layers, along with their activations, as shown in Figure 16.

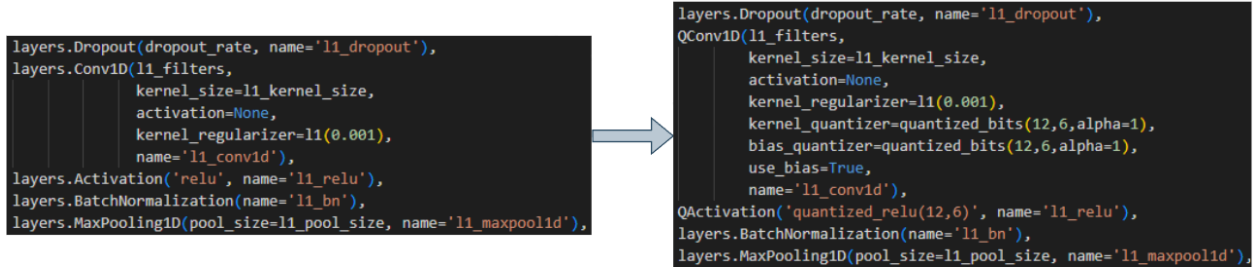


Figure 16: Converting TensorFlow Keras Layers to their QKeras Counterparts

All other training hyperparameters were left unmodified from the baseline model's training routine. This began the model in a random initialization state and trained it for the same number of epochs as the baseline but with the new 12-bit parameters. Performing QAT this way allowed the model to learn from the training dataset in the lower 12-bit width and improve its accuracy through minimizing the loss function with quantization noise present. Once the quantized model was trained, it was then pruned at various pruning ratios to determine the highest percentage of weights it could remove without greatly reducing model performance.

4.6.2 Pruning

This thesis pruned the quantized and fine-tuned KWS model's low-magnitude weights to a target sparsity of 50%. An additional 50 epochs were used after QAT to iteratively prune unimportant weights from the model. This was the final step in model optimization as it required a fully trained model to find weights that did not significantly affect the output. Lastly, the optimized and compressed model was converted into a compatible FPGA digital circuit using

hls4ml to produce an FPGA block design and bitfile that could be programmed onto the target FPGA development platform.

4.7 Target FPGA Hardware

Before converting the KWS neural network into an FPGA design with hls4ml, the target FPGA had to be selected. This was a preliminary requirement because the FPGA's resources needed to be known to appropriately utilize hls4ml's customization hyperparameters. As such, the PYNQ-Z2 was chosen as the deployment platform which is shown in Figure 17 along with its intended configuration for the real-time KWS task. The PYNQ-Z2 is an FPGA System-on-Chip (SoC) development platform consisting of a dual ARM Cortex-A9 microprocessor and a directly attached FPGA fabric. Its ARM core Processing System (PS) and its FPGA Programmable Logic (PL) communicate with one another to control on-board peripherals and implement custom FPGA functions.

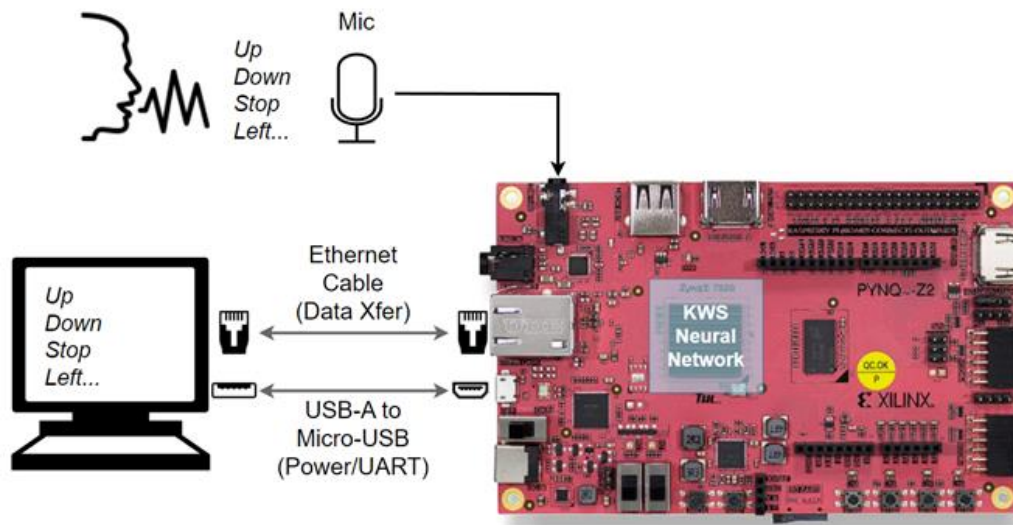


Figure 17: PYNQ-Z2 Development Kit and Thesis Application. PYNQ-Z2 Screenshot in [22].

The PS runs on the PYNQ v3.0.1 image which is an Ubuntu-like operating system where the primary programming environment is through Jupyter notebooks using the Python

programming language which allowed for communication between the ARM processor and directly attached FPGA fabric. Preprocessing and posterior handling for the model was performed on the PS side while the model and keyword inference occurred completely on the PL side. This resulted in a system that could classify utterances in real-time with extremely low latencies through a user-interface on the PYNQ-Z2 development board and could provide visual confirmations of the classified keyword and inference time through a graphical Jupyter notebook.

4.8 Converting KWS Model into an FPGA Block Design

Using the hls4ml package required a Linux environment to allow it to interface with Vivado and Vivado HLS for configuring, synthesizing, and implementing the model to ultimately generate its bitfile that could be programmed onto the FPGA. To do so, a Python program was written to configure the necessary hyperparameters to balance very low-latency inferences with FPGA resource utilization and to invoke the hls4ml package to generate the design as shown in Figure 18.

```
# Set the precision and reuse factor for the full model
hls_config['Model']['Precision'] = 'ap_fixed<12,6>'
hls_config['Model']['ReuseFactor'] = 1
hls_config['Model']['Strategy'] = 'Resource'

for Layer in hls_config['LayerName'].keys():
    hls_config['LayerName'][Layer]['Strategy'] = 'Resource' # 'Resource' or 'Latency'
    hls_config['LayerName'][Layer]['ReuseFactor'] = 36

cfg = hls4ml.converters.create_config(backend='VivadoAccelerator')
cfg['IOType'] = 'io_stream' # Must set this to 'io_stream' if using CNNs
cfg['HLSConfig'] = hls_config
cfg['KerasModel'] = model
cfg['ProjectName'] = '30kb_kws_model'
cfg['OutputDir'] = '30kb_kws_model_hls4ml/30kb_kws_model'
cfg['Part'] = 'xc7z020c1g400-1' # 'xc7z020c1g400-1' for the Zynq-7020 cfg['Board'] = 'pynq-z2'

hls_model = hls4ml.converters.keras_to_hls(cfg)

# Compile, synthesize, implement, and generate the bitstream
hls_model.compile()
hls_model.build(csim=False, export=True, bitfile=True)
```

Figure 18: Invoking hls4ml Package to Convert the KWS Model into an FPGA Design

This study focused on minimizing inference latency and maximizing throughput, so the reuse factor was set to 36 for each layer so that the model's unrolled to the greatest extent while not using every available DSP slice on the FPGA. The overall model reuse factor was set to one so that it would be completely unrolled at the model-wide level, although it required more resources. The precision was set to use the same bit-width used during model quantization with 12-bits to ensure the model outputs would exactly match the TensorFlow-trained model's outputs for the same input sequence. Additional code necessary for performing the conversion is shown in Appendix A.2. Having the FPGA block design with the KWS model as an IP block allowed the overall real-time system to be designed with a few changes.

4.9 Implementing Real-Time KWS System on the PYNQ-Z2

Implementing the real-time KWS system required taking the FPGA block design generated from Vivado HLS and hls4ml and programming it to the PYNQ-Z2. Before this was possible, the FPGA design needed to be modified in Vivado to include the ability to control the on-board peripherals like the buttons, LEDs, and most importantly the audio codec. Furthermore, additional Python code was written to run on the PS for sampling the audio signals with the audio codec, converting the audio signal into a compatible 16kHz data stream, and then transferring the extracted MFCC features to the PL for inference. First, the necessary additions were made to the base hls4ml block diagram as shown in Figure 19.

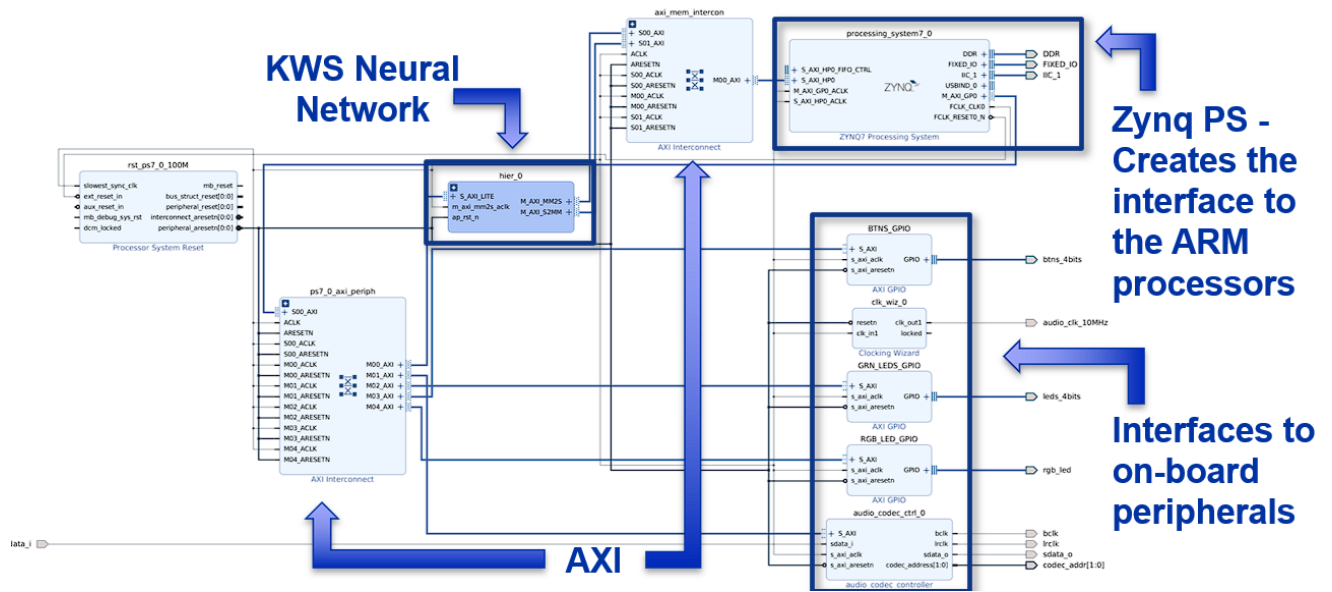


Figure 19: Top-Level Block Diagram of Full FPGA KWS System

The top-level diagram in Figure 19 shows the ZYNQ processing system which handles all the interfacing between the programmable logic in the FPGA to the ARM processor primarily based on the AXI bus protocol and Direct Memory Access (DMA) connections. The central IP block hierarchy represents the KWS 1-D CNN neural network which is shown expanded in Figure 20.

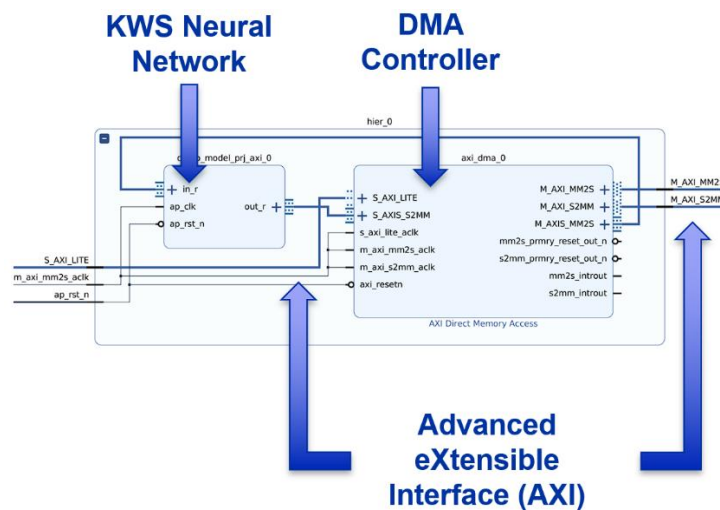


Figure 20: Expanded Neural Network IP Block

The hls4ml-generated hierarchy contains the “demo_model” IP containing the model used for the final demonstration of this thesis along with an AXI DMA IP block. The AXI DMA manages all the needed operations for rapidly transferring audio feature inputs and the predicted class outputs to and from the PS. The additional blocks that were needed included the AXI GPIO IP blocks and the audio codec controller to interface the PYNQ-Z2’s audio codec IC to the PS. These blocks are shown in Figure 21.

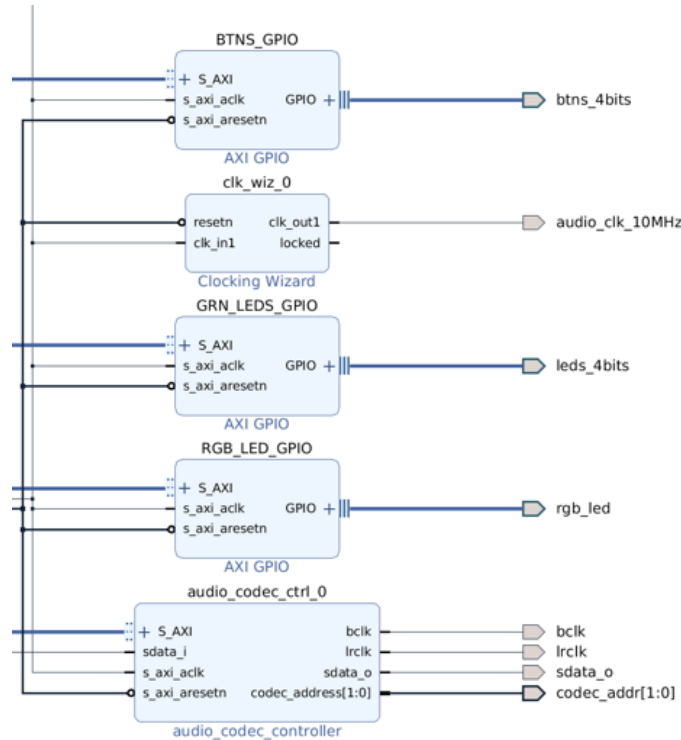


Figure 21: GPIO and Audio Codec Control Blocks

These blocks allowed for the PYNQ-Z2’s onboard peripherals to be interacted with by a user in the real-time KWS system. To interface the PS with the PL, driving Python code was written for the PYNQ-Z2’s Jupyter notebook environment to match the system-level operation diagram in Figure 22.

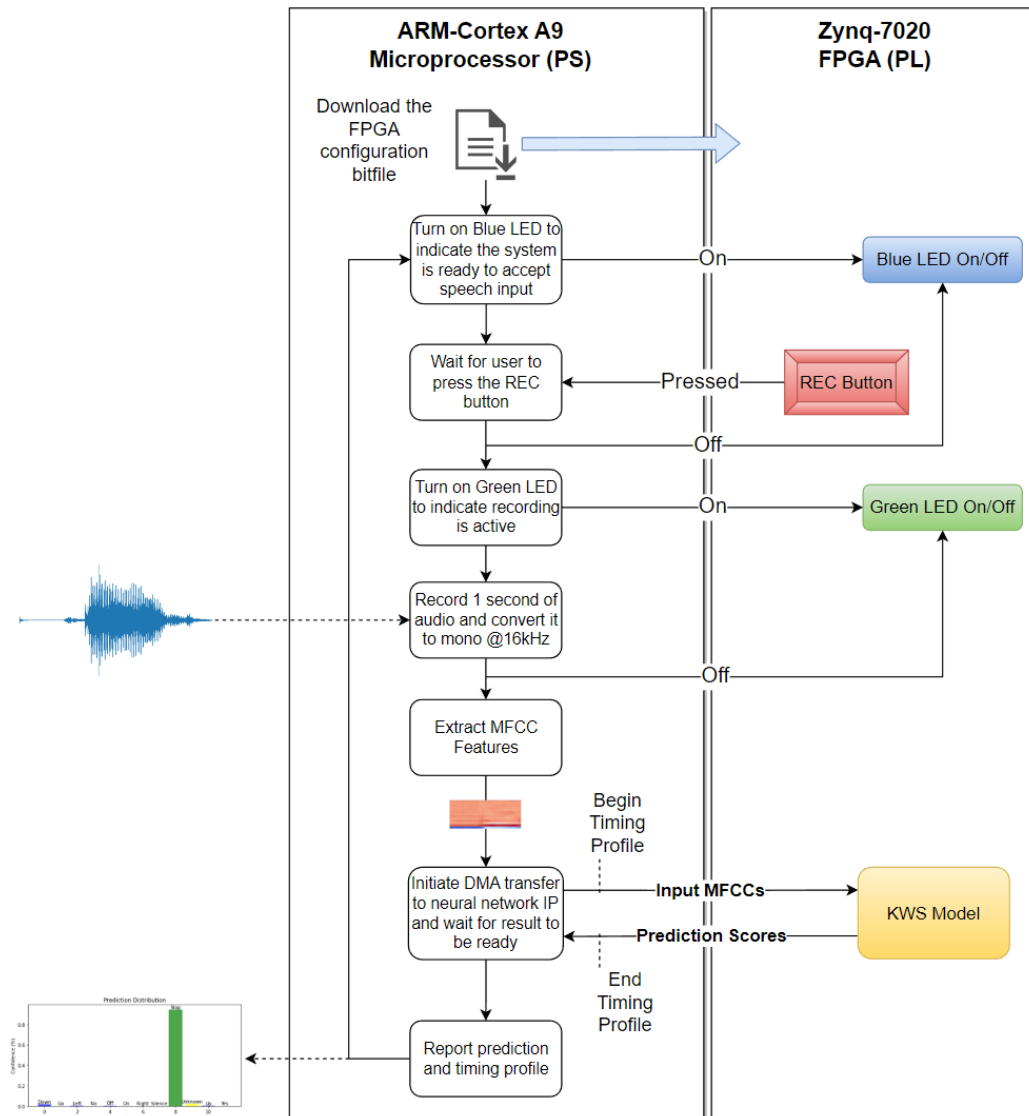


Figure 22: PYNQ-Z2 Driving Software Diagram

This software flow enabled users to observe a blue LED that indicated the system was ready to classify another keyword. Pressing the record button enables a green LED to indicate that the speaker should utter their word and then it samples the audio port for one-second and stores the resulting audio data into a buffer. The audio codec on the PYNQ-Z2 samples the audio with dual channels (interleaved) at 48kHz and as 24-bit PCM data, so it needed to be converted into mono audio at 16kHz and then normalized. This was accomplished by separating the two interleaved channels, taking the average of every sample pair, and then taking every third point

to resample the signal at 16kHz. Since it was chosen to normalize the audio signal itself, the 24-bit PCM data of the real-time audio compared to the 16-bit PCM data of the training dataset was not an issue as normalization shifts them to the same scale. Then, the same preprocess function from the training routine shown in Figure 12 was applied. To perform inference on the extracted MFCCs, a DMA transfer signal is sent to the PL which begins the parallel transfer of each point and then the PS waits until the KWS model IP block sends a ready signal back to the PS. When the data-ready signal from the PL is received the prediction scores from the FPGA DMA buffer are stored into a Python list. Finally, with posterior processing, the predicted keyword is displayed into the Jupyter notebook along with a bar graph showing the Softmax of the output and the probability distribution the model assigned to that audio input. The simplified Jupyter notebook code that achieved the functionality described in Figure 22 is shown in Appendix A.3 and uses the base overlay for all board I/O and the KWS neural network for fast inference. An example output of this program is also demonstrated in Figure 23.

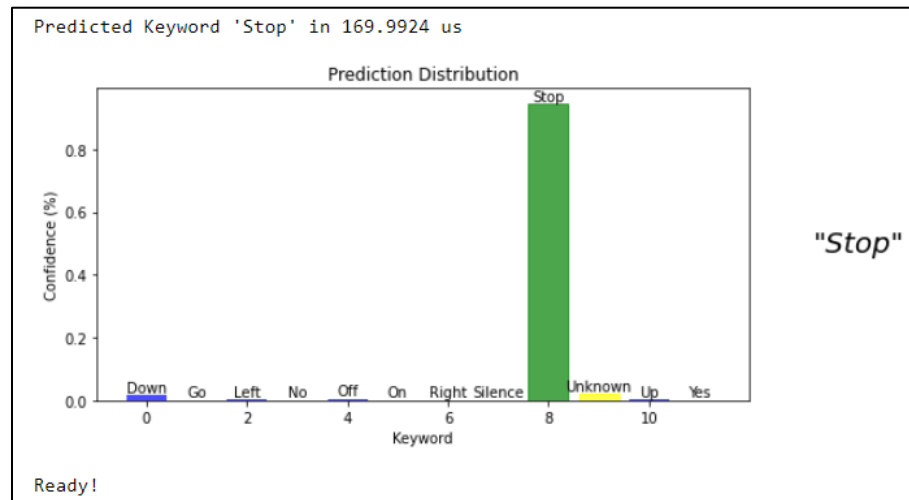


Figure 23: Example Output of Real-Time KWS System on the PYNQ-Z2

The predicted keyword is shown along with the amount of time it took the FPGA to compute and transfer its predicted classes to the PS. Since the Softmax function was applied to

the output scores of the model in posterior handling, the bar graph shows a representation of how confident it was that a particular keyword was present in the audio signal. Here, thresholding could be applied to set the required confidence needed for classifying a keyword over classifying the input as *Unknown*. In this example, the highest-scoring class was chosen as the accepted class. This functioning design completed the real-time KWS system that utilized FPGA for hardware acceleration to allow users to classify keywords from their own utterance.

Chapter 5: Results

Various types of post-training evaluation methods were applied, such as top-one accuracy, confusion matrices, and a t-distributed stochastic neighbor embedding (t-SNE) metric, to compare the performance of the baseline, unquantized and uncompressed model to the optimized model. In addition, the models of this thesis were compared to the state-of-the-art optimized models in related literature. In addition, the model's inference latency was evaluated across several different computing platforms to determine if FPGAs are successful machine learning model accelerators.

5.1 Baseline Model vs Optimized Model Performance

Both the baseline model and the optimized model were trained and evaluated with the same dataset partitions, and both performed with high accuracy. The test dataset partition was used for each all the evaluation methods and was never seen by the model during training or validation. Table 4 summarizes the results of evaluating both models with the test dataset.

Table 4: Base Model and Optimized Model Evaluation and Parameters

Model	Accuracy (%)	Sparsity (%)	Parameter Count	Size (KB)
Baseline	91.48	0.000	7772	30.36
Optimized	90.16	48.88	7772	11.38

The baseline model achieved a 91.48% top-one accuracy with 7,772 parameters taking up 30.36 Kilobytes in total, and naturally had no sparsity. The optimized model was fine-tuned to achieve a 90.16% top-one accuracy for a negligible 1.32% accuracy degradation from the base model with effectively 62.5% of the size. The target sparsity of the optimized model was 50% but achieved 48.88% sparsity within the given 50 pruning epochs. Adjusted for the 12-bit

quantized weights, the optimized model takes up only 11.38 Kilobytes for a size-reduction factor of 2.667.

Another method of evaluating the model's performance for each keyword is the confusion matrix, shown in Figure 24, which displays the distribution of true labels verses the actual predictions for each keyword.

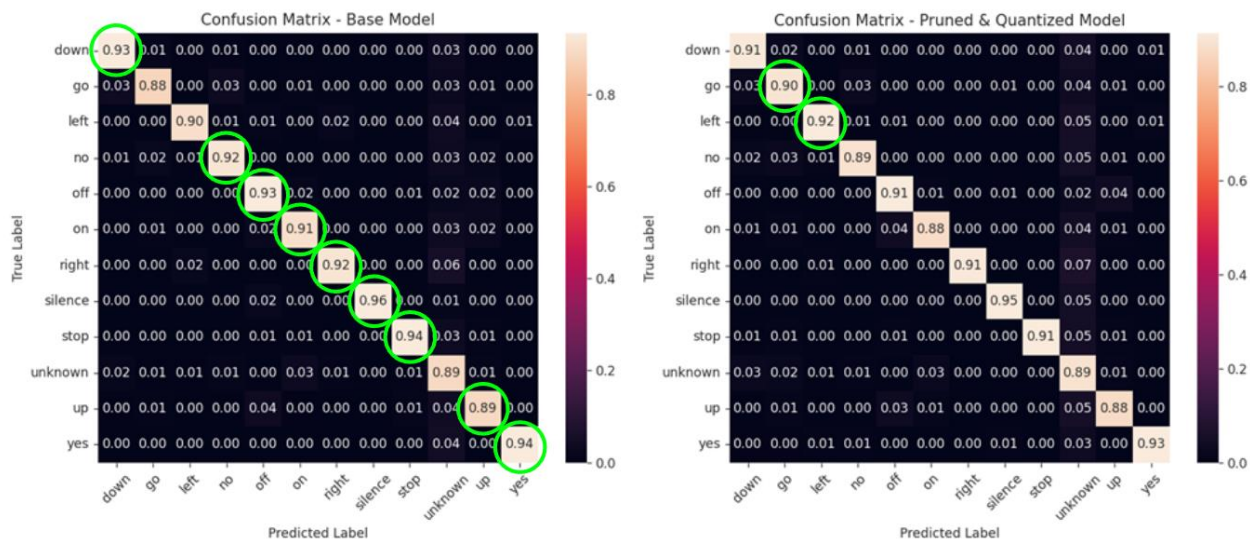


Figure 24: (Left) Confusion Matrices for Base Model (Left) and Compressed Model (Right)

The confusion matrices better show the models' abilities to classify each class correctly. Along the diagonals are the overall top-one accuracy for each class when evaluated against the test set. Some keywords perform better than others, but both the base model and the optimized model have similar confusion matrices and accuracies across keywords. Other words saw slightly improved classification accuracies in the compressed model, shown as the circled cells in the right matrix of Figure 24, with its simpler representation of the feature space and due to better numerical regularization. Overall, only minor accuracy degradations were seen across the keywords.

In addition to accuracy metrics, a t-distributed stochastic neighbor embedding metric was applied to the output layer of the optimized model. The resulting t-SNE scatterplot in Figure 25

shows how well the optimized model was able to distinguish different keywords from one another.

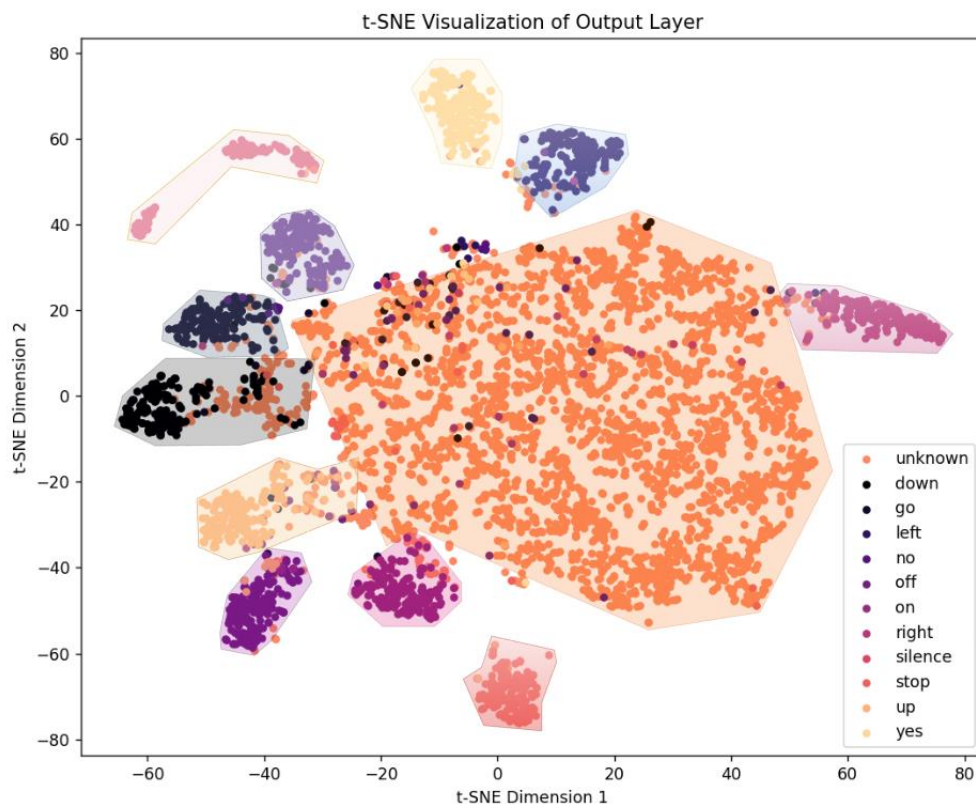


Figure 25: t-SNE Class Separation Visualization

This t-SNE plot demonstrates that the model was able to separate keywords with both high accuracy and with good confidence. Each keyword was placed tightly in its own region with good separation between other keywords indicating that the model could score a particular true keyword with a significantly higher value than the other keywords for most inferences. Notably, the large region of unknown samples was widespread across the plot which indicates the wide range of phonetic and speaker variability in the dataset. Further comparisons were done to evaluate these KWS models against state-of-the-art models in literature.

5.2 Performance Comparison to Related Work

There are many deep KWS models that vary in their neural network architectures, complexities, and performances. Zhang et al. [19] previously discussed in the literature review of section three, describes many of these different model types and provides metrics for each of them based on the goal of making highly accurate models for embedded system deployment. The models evaluated were comparable in size and performance to the model of this thesis and are summarized in Table 5 by showing the smallest models trained from each model architecture type.

Table 5: Comparison Between My Thesis' Models and Zhang et al.'s Reviewed 8-Bit Models

Model	Accuracy (%)	Size (KB)
My Thesis		
1-D CNN 12-bit quantized	90.16	11.38
1-D CNN 32-bit	91.48	30.36
Zhang et al. [19]		
DS-CNN	94.40	38.60
Basic LSTM	92.00	63.30
GRU	93.50	78.80
2-D CNN	91.60	79.00
LSTM	92.90	79.50
CRNN	94.00	79.70
DNN	84.60	80.00

Many of the models from [19] employ recurrent architectures like the Gated Recurrent Unit, Long Short-Term Memory, and the Convolutional Recurrent Neural Network. These models are better equipped to take advantage of long-term temporal dependencies in the input features compared to 1-D CNNs which are limited in their ability to extract distant temporal contexts.

However, the baseline 1-D CNN model of this thesis competitively achieved an accuracy of 91.48% which exceeded the DNN's accuracy by 6.88% with 62.05% less storage and fell just

short of the basic LSTM’s accuracy by 0.52% but with 52.03% less storage. The baseline model also had an accuracy near Zhang et al.’s most accurate model, the DS-CNN for a difference of 2.92% with this thesis’ model requiring 21.35% less storage.

Although the 1-D CNN of this thesis had an accuracy that appeared to be approaching the DS-CNN’s, the hyperparameter grid search demonstrated that this thesis’s KWS model may need to grow to a much larger size to achieve greater accuracy performance. The achieved 91.48% performance was accomplished with a meticulously crafted approach and is expected to be much more purpose-built than most other practical machine learning models and was observed to resemble the peak performance that a model of this size could achieve. Breaking the 92% accuracy mark required models that quickly grew beyond the size of the small footprint models discussed in this comparison, so the DS-CNN’s performance is highly impressive with such a small size.

Overall, these results indicate that the 1-D CNN is capable of being competitive with models that can gather longer-term dependencies but with the 1-D CNN requiring far fewer parameters, especially when optimized with quantization and pruning techniques. After completing the evaluation of the models’ accuracies, it was deployed onto FPGA hardware to assess its inference latency and resource utilization.

5.3 FPGA Acceleration Results

Both models were evaluated for their inference latency across three devices as summarized in Table 6. It is important to note that within every latency metric, the exhibited performance always depends on how advanced the device is. Thus, the exact inference latency will vary slightly between different CPUs, GPUs, and FPGAs. Similarly, the GPU required “warm-up” batches of inferences so that the proper resource communication channels could be

established in the computer. For consistency, each device type was warmed up prior to evaluating inferences, and the inference latencies were evaluated on batches of 1000 to ensure accurate measurements. Thus, running inference on singular samples unregularly was expected to exhibit longer than usual latencies.

Table 6: Batched Model Inference Latency on Various Device Architectures for 1000 Samples

Device	Baseline Model Latency (ms)	Optimized Model Latency (ms)
AMD Ryzen7 Pro 6850U CPU	4.335	4.335
NVIDIA GeForce RTX 4050 Laptop GPU	2.191	2.191
Pynq-Z2 FPGA - Base Model	0.374	0.373

The models were specifically evaluated for comparison on a high-end CPU, an NVIDIA GPU, and finally the PYNQ-Z2 FPGA. Prior to evaluating latency, each device was warmed up by classifying a batch of 1000 samples. First, the CPU was evaluated with another 1000 samples which produced an average batched inference time of 4.335ms per sample for the base model. These times were reduced when using an NVIDIA GeForce RTX 4050 Laptop GPU to get an average batched inference latency of 2.191ms per inference. An even greater reduction in time was seen when inference was run on the FPGA with the optimized model which demonstrated a speedup of 11.6 times over the CPU inference and a speedup of 5.9 times over the GPU inference with classifications completing in 373 μ s, on average. Also, running inference on single samples on the PYNQ-Z2 in real-time with no warmup demonstrated inference latencies between 1.1ms and 1.8ms which still beat the performance of the CPU and GPU even in their batched runs of classifications.

Based on these results, the compression of the model did not have obvious side effects in its latencies with respect to its baseline implementation. This can be attributed to two reasons. For one, the model only had 7,772 parameters and therefore a comparably low number of MACs

to complete, so compressing it further may not have obvious symptoms in inference latency across devices. The second and more dominating reason was that the CPU and GPU tests could not account for the custom 12-bit resolution and instead stored the 12-bit values as 32-bit numbers in memory. They also provided no optimization for zero-based multiplications. In addition, the FPGA implementations offered nearly identical latencies primarily due to the same reuse factor being used when synthesizing the design with hls4ml. Despite small differences between the latencies of the baseline and compressed models, these results demonstrate that FPGAs can accelerate the inference of KWS models beyond the ability of GPUs with application-specific hardware implementations.

The 0.373ms latency of the model in this thesis also beat out all the models described in the study of [20] which achieved latencies between 10ms and 116ms. Similarly, the compressed version of the KWS models from this thesis had a smaller size than all the reviewed models except for two which achieved 11KB and 2KB. However, despite achieving faster latencies and mostly smaller model sizes than the works in [20], all the reviewed models utilized extremely small bit widths between 1-bit and 8-bit with even more complicated network architectures. Those reviewed in [20] exhibit special cases of using as few bits as possible to achieve accurate neural networks that may have been able to achieve slightly higher accuracies had this not been a primary focus.

Finally, beyond latency, FPGA resource utilization was also benchmarked for both models to see if optimizing the KWS models could provide significant utilization reductions. These results are summarized in Table 7.

Table 7: FPGA Resource Utilization of KWS Model

FPGA Resource	PYNQ-Z2 Total	Base Model Utilization (Count)	Base Model Utilization (%)	Optimized Model Utilization (Count)	Optimized Model Utilization (%)	% Usage Reduction w/ QAT
Slice LUTs	53200	25762	48.42	25496	47.92	1.03
Slice Registers	106400	41867	39.35	38996	36.65	6.86
Slice	13300	11309	85.03	10831	81.44	4.23
BRAM Tiles	140	53	37.86	49	35.00	7.55
DSPs	220	93	42.27	92	41.82	1.08

It was observed that the compressed model did provide a reduction in resource usage across each category over the baseline as expected. Overall, these differences were slight but could be improved further by modifying the reuse and quantization parameters in the hls4ml configuration. The greatest reduction was in BRAM usage of the compressed model which corresponds to its higher usage of slice LUTs which are accessed faster by the FPGA hardware and were likely able to store the zero-based multiplication parameters. In addition, although they were implemented with the same reuse factor, the baseline model utilized one more DSP than the compressed model which was likely due to the zero-based multiplication optimization being implemented with simpler circuitry than a DSP block. This was also largely due to the model having so few parameters which left little room for FPGA synthesis optimization. Ultimately, FPGAs proved successful in accelerating the KWS model developed in this thesis.

Chapter 6: Future Work

While this research has provided significant insights into small footprint KWS models and accelerating them with FPGAs, there are several avenues for future exploration and development that can build upon the findings of this research. First, ignoring the many privacy concerns regarding speech-recognizing consumer electronics, the implementation of the real-time KWS system on the PYNQ-Z2 could be extended to operate in streaming mode with “always on” or “always listening” behavior similar to the application in Figure 26.

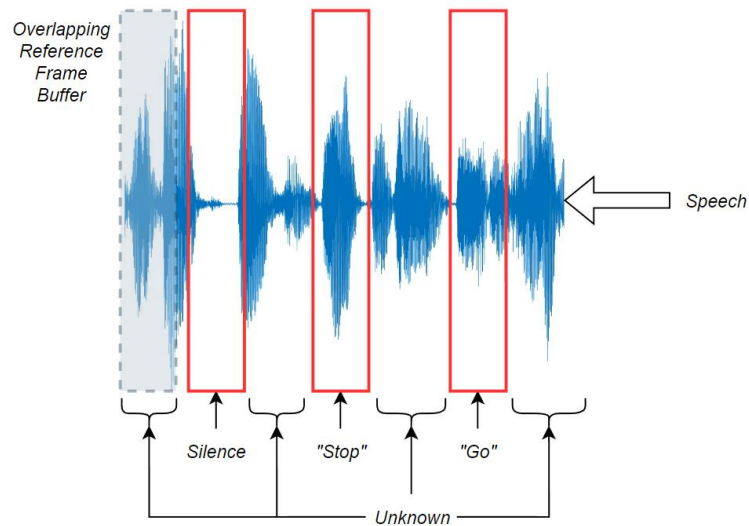


Figure 26: Streaming KWS Application

This thesis implemented the real-time system with a button that is pressed before recording and classifying a one-second audio file. This portion of the implementation can be modified to perform streaming KWS which continuously takes in new audio information and performs inference on overlapping segments of it to detect keywords as they are uttered in the environment instead of only on button presses. In this approach, topics like thresholding and refractory periods on the classification outputs are expected to be important in having a meaningful streaming KWS system. KWS models that operate on the actual hardware can be

exempt from concerns about speech recognition privacy in “always on” systems because, other than the buffering needed to perform inference, there is no true storage or transmission of the user’s speech.

Another separate extension of the work done in this thesis would be to add voice activity detection (VAD) to the real-time implementation as shown in Figure 27.

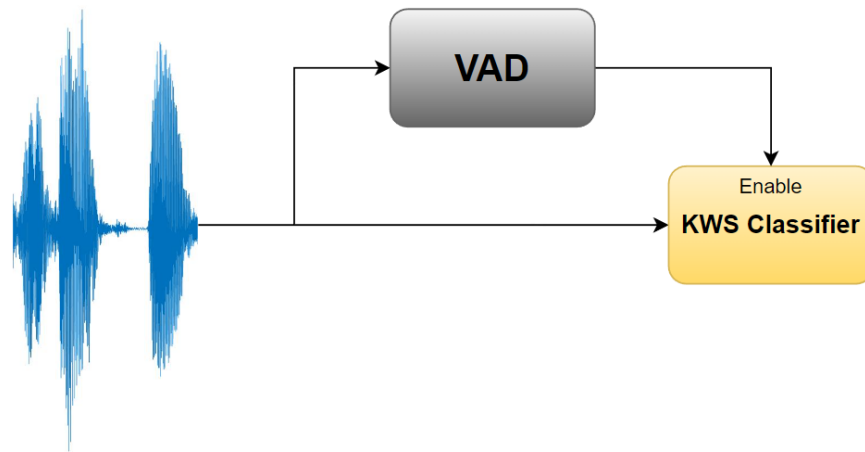


Figure 27: Voice Activity Detection for KWS

VAD allows a KWS system to only perform classifications when it is likely that voice activity is present in the audio stream. This mitigates excessive energy use and unnecessary keyword classifications when there is no speech in the environment. This type of extension would be particularly important for low-energy embedded systems and for wake word applications.

One final recommended extension of the research done in this thesis is to combine the MFCC feature extraction pipeline and the KWS neural network into a singular FPGA IP block as shown in Figure 28.

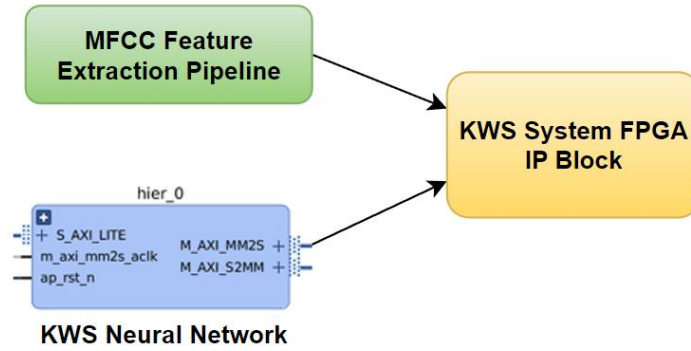


Figure 28: End-to-End KWS System

Combining both portions of the entire KWS system would provide an end-to-end KWS system and achieve even faster overall inference latencies by accelerating both KWS inference and feature extraction at the same time. There are many other avenues that could be explored to build off the work done in this thesis, too, but these examples serve as a good starting point to gain a better understanding of other issues that revolve around implementing real-time KWS systems.

Chapter 7: Conclusion

This thesis described a lightweight keyword spotting (KWS) model that achieved a baseline accuracy of 91.48% with only 7,772 weights, or 30.36 KB of storage with 32-bit floating point parameters. The model, trained using Google Speech Commands V2 dataset, used a 1-D CNN architecture to extract short-term temporal dependencies in the MFCC input features. Class balancing and weighting techniques were used to achieve the greatest training performance and accurately model a realistic KWS system. Furthermore, an optimized and compressed version of the model was developed using quantization and pruning techniques which reduced the model's bit-widths to 12-bits resulting in model that was 48.88% sparse with only a minor accuracy degradation of 1.32%. These metrics compared well with high-performance light-weight recurrent models in literature with up to a 62.5% reduction in model size of the largest 8-bit models that were evaluated.

In addition, FPGAs were utilized as the target deployment platform to evaluate their effectiveness in accelerating the inference of KWS models. The models were converted into an FPGA digital circuit with the hls4ml package, and deploying it demonstrated average inference latencies of 178 μ s. This corresponded with large speedups of up to 11.6 times and 5.86 times over CPU and GPU latencies, respectively. These metrics show that FPGAs are very effective in accelerating CNN-based neural network architectures and KWS models, and that they can achieve latencies much faster than GPUs with purpose-built digital circuits.

This thesis demonstrated that accurate and fast KWS models can be achieved with very few parameters. It utilized promising evidence in literature of neural network techniques for the KWS task and accelerating them with FPGAs and furthered research in the areas. Doing so provided a basis for utilizing FPGAs as a precursor to KWS ASICs that can be dropped onto any

embedded system to provide speech recognition capabilities. Additionally, this thesis improved the overall effectiveness of these technologies in recognizing keywords, thereby contributing to advancements in the field of speech recognition and enhancing the applicability of KWS models in various real-world scenarios.

Appendices

Appendix A – Code

A.1 Defining Mel-Filterbank

```
def save_mel_filterbank():
    mel_filterbank = tf.signal.linear_to_mel_weight_matrix(
        num_mel_bins=24,
        num_spectrogram_bins=128,
        sample_rate=16000,
        lower_edge_hertz=20,
        upper_edge_hertz=7400)

    np.save("tf_mel_filterbank24.npy", mel_filterbank)
```

A.2 FPGA Conversion of Quantized TensorFlow Model with hls4ml

```
import hls4ml

### For Synth
from pathlib import Path
import pprint
import os
import time
import tensorflow as tf
from matplotlib import pyplot as plt
import numpy as np
from qkeras.utils import _add_supported_quantized_objects

# Add quantization layers so hls4ml can use them
co = {}
_add_supported_quantized_objects(co)

# Load the TensorFlow Keras model and convert it into an HLS configuration
model = tf.keras.models.load_model('TrainedModels/30kb_kws_model.h5',
    custom_objects=co)
hls_config = hls4ml.utils.config_from_keras_model(model, granularity='name')

# Set the precision and reuse factor for the full model
hls_config['Model']['Precision'] = 'ap_fixed<12,6>'
hls_config['Model']['ReuseFactor'] = 1
hls_config['Model']['Strategy'] = 'Resource' # 'Resource' or 'Latency'
for Layer in hls_config['LayerName'].keys():
    hls_config['LayerName'][Layer]['Strategy'] = 'Resource' # 'Resource' or 'Latency'
    hls_config['LayerName'][Layer]['ReuseFactor'] = 36

cfg = hls4ml.converters.create_config(backend='VivadoAccelerator')
cfg['IOType'] = 'io_stream' # Must set this to 'io_stream' if using CNNs
```

```

cfg['HLSConfig'] = hls_config
cfg['KerasModel'] = model
cfg['ProjectName'] = '30kb_kws_model'
cfg['OutputDir'] = '30kb_kws_model_hls4ml/30kb_kws_model'
cfg['Part'] = 'xc7z020clg400-1' # 'xc7z020clg400-1' for the Zynq-7020
cfg['Board'] = 'pynq-z2'
hls_model = hls4ml.converters.keras_to_hls(cfg)

# Compile, synthesize, implement, and generate the bitstream
hls_model.compile()
hls_model.build(csim=False, export=True, bitfile=True)

```

A.3 Real-Time KWS Deployment Jupyter Notebook

```

classes = [
    'Down',
    'Go',
    'Left',
    'No',
    'Off',
    'On',
    'Right',
    'Silence',
    'Stop',
    'Unknown',
    'Up',
    'Yes'
]

# Initially download overlays for speedier downloads later
print("Downloading overlays")
base = BaseOverlay("base.bit")

# Prepare initial audio recording
pAudio = base.audio
pAudio.set_volume(45)
pAudio.select_microphone()

# Turn off all LEDs
for led in base.leds:
    led.off()

blue_led_on()

print("Ready!")
while(1):
    if(base.buttons[3].read()==1):

        blue_led_off()
        green_led_on()
        pAudio.record(1)
        pAudio.save("recorded_keyword.wav")
        green_led_off()

        pAudio.load("recorded_keyword.wav")
        pAudio.play()

```

```

# Clear Jupyter Notebook output
clear_output(wait=True)

rate, data = wavfile.read("recorded_keyword.wav")

# Convert from stereo to mono
left_chan = data[:, 0]
right_chan = data[:, 1]
data = (left_chan + right_chan) / 2.0

# Downconvert from 48kHz to 16kHz audio
data = pad_audio(data, 48000) # Pad to ensure final audio length is 16,000
data = data / 16777216.0      # Normalize data for 24-bit width
data = data[::3]              # Resample 48 kHz audio to 16 kHz by taking
every 3rd sample

mfccs = get_mfccs(data)
plot_audio_waveform(data, 16000)
plot_mfcc(mfccs)

mfccs = np.expand_dims(mfccs, axis=0)

# Download KWS overlay
kws = NeuralNetworkOverlay("kws_model.bit", (2,127,24), (2,12))

# Predict recorded audio
t_start = time.time()
y_hw = kws.predict(np.ascontiguousarray(mfccs), profile=False)
t_end = time.time()

prediction = get_prediction(y_hw[0].tolist())

# Display prediction as highest scoring keyword
print(f"Predicted Keyword '{prediction}' in {(t_end - t_start) *
1000000):.4f} us")
plot_bar(y_hw[0].tolist(), prediction, classes)

# Redownload base overlay for I/O usage
base.download()
print("Ready!")
blue_led_on()

```


References

- [1] I. López-Espejo, Z. H. Tan, J. H. L. Hansen and J. Jensen, "Deep Spoken Keyword Spotting: An Overview," in *IEEE Access*, vol. 10, pp. 4169-4199, 2022, doi:10.1109/ACCESS.2021.3139508
- [2] S. Li, G. Li, J. Han and T. Zhi, "Overview of Speech Keyword Recognition Technology," in *J. Phys.: Conf. Ser. Vol. 1827, 6th ICETIS*, Harbin, China, 2021, doi: 10.1088/1742-6596/1827/1/012013
- [3] G. S. Nikolić, B. R. Dimitrijević, T. R. Nikolić and M. K. Stojcev, "A Survey of Three Types of Processing Units: CPU, GPU and TPU," *2022 57th International Scientific Conference on Information, Communication and Energy Systems and Technologies (ICEST)*, Ohrid, North Macedonia, 2022, pp. 1-6, doi: 10.1109/ICEST55168.2022.9828625
- [4] C. Wang and Z. Luo, "A Review of the Optimal Design of Neural Networks Based on FPGA," in *Appl. Sci.* vol. 12, no. 58, Sept. 2022, doi: <https://doi.org/10.3390/app122110771>
- [5] P. Vitolo, R. Liguori, L. D. Benedetto, A. Rubino and G. D. Licciardo, "Automatic Audio Feature Extraction for Keyword Spotting," in *IEEE Signal Processing Letters*, vol. 31, pp. 161-165, 2024, doi: 10.1109/LSP.2023.3346280
- [6] M. N. Kunchur, "The human auditory system and audio," *Applied Acoustics* vol. 211, 2023, doi: <https://doi.org/10.1016/j.apacoust.2023.109507>
- [7] Gourisaria, M.K., R. Agrawal, M. Sahni and P. K. Singh, "Comparative analysis of audio classification with MFCC and STFT features using machine learning techniques." in *Discover Internet of Things*, vol. 4, no. 1, 2024, doi: <https://doi.org/10.1007/s43926-023-00049-y>
- [8] Z. K. Abdul and A. K. Al-Talabani, "Mel Frequency Cepstral Coefficient and its Applications: A Review," in *IEEE Access*, vol. 10, pp. 122136-122158, 2022, doi: 10.1109/ACCESS.2022.3223444
- [9] A. Shenfield and M. Howarth, "A Novel Deep Learning Model for the Detection and Identification of Rolling Element-Bearing Faults," in *Sensors*, vol. 20. no. 5112, doi: 10.3390/s20185112
- [10] S. Kiranyaz, O. Avci, O. Abdeljaber, T. Ince, M. Gabbouj and D. J. Inman, "1D convolutional neural networks and applications: A survey," in *Mech. Systems and Signal Process.*, vol. 151, 2021, doi: <https://doi.org/10.1016/j.ymssp.2020.107398>

- [11] T. Choudhary, V. Mishra, A. Goswami and J. Sarangapani, "A comprehensive survey on model compression and acceleration," in *Artif. Intell. Review*, vol. 53, pp. 5113-5115, Feb. 2022, doi: <https://doi.org/10.1007/s10462-020-09816-7>
- [12] Y. Guo, "A Survey on Methods and Theories of Quantized Neural Networks," 2018, arXiv:1808.04752
- [13] J. Bushur and C. Chen, "Neural Network Exploration for Keyword Spotting on Edge Devices," in *Future Internet*, vol. 15, no. 6, 2023, doi: <https://doi.org/10.3390/fi15060219>
- [14] S. K. Yeom et al., "Pruning by explaining: A novel criterion for deep neural network pruning," in *Pattern Recognition*, vol. 155, July 2021, doi: <https://doi.org/10.1016/j.patcog.2021.107899>
- [15] A. Boutros and V. Betz, "FPGA Architecture: Principles and Progression," in *IEEE Circuits and Sys. Mag.*, vol. 21, no. 2, pp. 4-29, Secondquarter 2021, doi: 10.1109/MCAS.2021.3071607
- [16] J. Duarte et al., "Fast inference of deep neural networks in FPGAs for particle physics," in *JINST*, vol. 13, July 2018, doi: 10.1088/1748-0221/13/07/P07027
- [17] T. N. Sainath and C. Parada, "Convolutional neural networks for small-footprint keyword spotting," in *Proc. Interspeech 2015*, pp. 1478-1482, doi: 10.21437/Interspeech.2015-352
- [18] P.M. Sørensen, B. Epp and T. May, "A depthwise separable convolutional neural network for keyword spotting on an embedded system," in *EURASIP J. on Audio, Speech, and Music*, vol. 10, 2022, doi: <https://doi.org/10.1186/s13636-020-00176-2>
- [19] Y. Zhang, S. Naveen, L. Lai and V. Chandra, "Hello Edge: Keyword Spotting on Microcontrollers," 2018, arXiv:1711.07128v3
- [20] S. Bae, H. Kim, S. Lee and Y. Jung, "FPGA Implementation of Keyword Spotting System Using Depthwise Separable Binarized and Ternarized Neural Networks," in *Sensors*, vol. 23, no. 12, 2023, doi: <https://doi.org/10.3390/s23125701>
- [21] P. Warden, "Speech Commands: A Dataset for Limited-Vocabulary Speech Recognition," 2018, arXiv:1804.03209
- [22] "PYNQ-Z2 Product Page." AMD.com. Accessed: Jul. 18, 2024. [Online.] Available: <https://www.amd.com/en/corporate/university-program/aup-boards/pynq-z2.html>