

VOICE COMMAND RECOGNITION WITH DEEP NEURAL NETWORK ON EDGE DEVICES

by

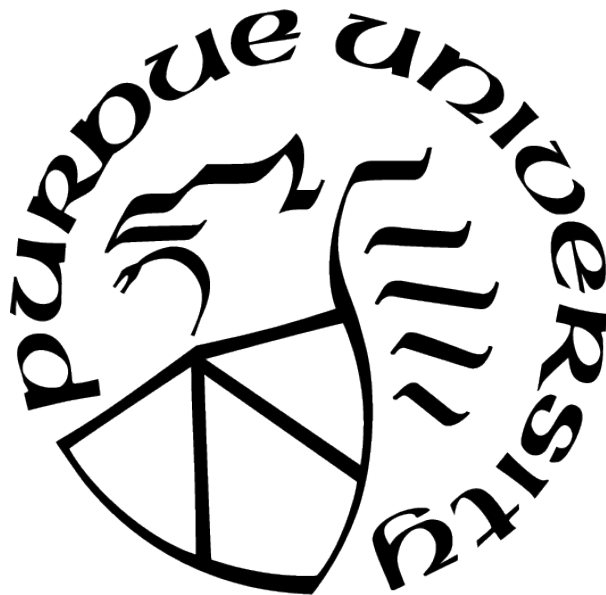
Md Naim Miah

A Thesis

Submitted to the Faculty of Purdue University

In Partial Fulfillment of the Requirements for the degree of

Master of Science in Engineering



Department of Electrical and Computer Engineering

Fort Wayne, Indiana

August 2021

**THE PURDUE UNIVERSITY GRADUATE SCHOOL
STATEMENT OF COMMITTEE APPROVAL**

Dr. Gouping Wang, Chair

Department of Electrical and Computer Engineering

Dr. Chao Chen

Department of Electrical and Computer Engineering

Dr. Bin Chen

Department of Electrical and Computer Engineering

Approved by:

Dr. Hosni Abu-Mulaweh

To my parents, beloved wife, brother and sisters.

ACKNOWLEDGMENTS

I would like to express my gratitude to Dr. Wang, my thesis advisor, for his ongoing support and encouragement for me to pursue this work. His guidance and expertise throughout the entire process allowed me to adjust and improve myself. I would also like to thank the Office of Graduate Studies for their support of this project. This assistance was immensely helpful in finishing this project more smoothly and successfully.

TABLE OF CONTENTS

| | |
|--|----|
| LIST OF TABLES | 8 |
| LIST OF FIGURES | 9 |
| ABBREVIATIONS | 11 |
| ABSTRACT | 12 |
| 1 INTRODUCTION | 14 |
| 2 NEURAL NETWORKS | 17 |
| 2.1 Convolutional Neural Network | 17 |
| 2.2 Spatial Convolution Layer | 17 |
| 2.3 Max Pooling | 18 |
| 2.4 Activation Function | 18 |
| 2.4.1 Sigmoid | 19 |
| 2.4.2 ReLU | 20 |
| 2.4.3 Softmax | 20 |
| 3 AUDIO PROCESSING | 21 |
| 4 DNN ARCHITECTURE | 25 |
| 4.1 DSCNN layer | 25 |
| 4.2 Model Architecture | 28 |
| 5 TRAINING MODEL | 30 |
| 5.1 Organizing Files | 30 |
| 5.2 Generating Datasets | 32 |
| 5.3 Creating Model | 33 |
| 5.4 Training Model | 34 |
| 6 HARDWARE PLATFORMS | 37 |

| | | |
|-------|--|----|
| 6.1 | 32-bit ARM Microcontroller | 37 |
| 6.1.1 | STM32F769NI Discovery Board | 37 |
| | Microphone MP34DT01TR | 39 |
| | Digital Filter for Sigma Delta Modulator | 40 |
| | Direct Memory Access (DMA) | 42 |
| 6.2 | Robotic Vehicle with Jetson Nano | 43 |
| 6.2.1 | Jetson Nano Developer Kit | 43 |
| 6.2.2 | Microphone Array | 44 |
| 6.2.3 | Interface Board | 45 |
| 7 | KEYWORD SPOTTING ON MICROCONTROLLER | 48 |
| 7.1 | Tools and Development Environment | 48 |
| 7.1.1 | STM32CubeMX code generator | 48 |
| 7.1.2 | X-CUBE-AI Expansion Package | 49 |
| 7.1.3 | Keil μ Vision5 IDE | 50 |
| 7.2 | Implementation in C | 51 |
| 7.2.1 | Initialization Code Generation | 51 |
| 7.2.2 | Data Acquisition | 54 |
| 7.2.3 | Feature Extraction | 55 |
| 7.2.4 | AI Implementation | 57 |
| 8 | KEYWORD SPOTTING ON JETBOT | 61 |
| 8.1 | Environment Configuration | 61 |
| 8.1.1 | Signal Acquisition from Microphone Array | 61 |
| 8.1.2 | Motor Driver | 62 |
| 8.2 | Implementation on Jetbot | 65 |
| 9 | EXPERIMENTAL RESULTS | 67 |
| 9.1 | MFCC Outputs | 67 |
| 9.2 | Training Outputs | 68 |
| 9.3 | Prediction Outputs and Execution Time | 69 |

| | | |
|-------|--|----|
| 9.3.1 | Results from Bare-Metal Microcontroller implementation | 69 |
| 9.3.2 | Results from Jetbot implementation | 71 |
| 10 | CONCLUSION | 74 |
| | REFERENCES | 76 |

LIST OF TABLES

| | | |
|-----|--|----|
| 4.1 | Proposed DNN architecture | 29 |
| 6.1 | Important features of STM32F769 Discovery board | 39 |
| 6.2 | Mapping Request for DMA2 controller at channel 8[22] | 42 |
| 8.1 | PCA9685 Motor Driver Interface | 64 |
| 9.1 | Summary of bare-metal implementation | 71 |

LIST OF FIGURES

| | | |
|-----|--|----|
| 2.1 | Basic CNN architecture[14] | 17 |
| 2.2 | Applying filter in convolution layer | 18 |
| 2.3 | Applying Max-pooling layer | 19 |
| 2.4 | Sigmoid activation function | 19 |
| 2.5 | ReLu activation function | 20 |
| 3.1 | Mel-spaced filter-bank of 16 filter | 23 |
| 3.2 | Mel Frequency Cepstral Coefficients for keyword “go” | 24 |
| 4.1 | Filter shape in standard convolution and depthwise separable convolution | 26 |
| 4.2 | Depthwise Separable Convolution with batchnorm and ReLU | 28 |
| 5.1 | Code for organizing dataset files | 31 |
| 5.2 | Code to calculate speech features, MFCC | 33 |
| 5.3 | Iterator to create dataset | 34 |
| 5.4 | Depthwise Separable CNN block | 35 |
| 5.5 | Callbacks to fit datasets into the DNN | 36 |
| 6.1 | Memory and Operations requirements for different networks [5] | 38 |
| 6.2 | STM32F769NI Discovery development board | 38 |
| 6.3 | MP34DT01TR digital MEMS microphone | 39 |
| 6.4 | JetBot AI Robot Car | 44 |
| 6.5 | Jetson Nano Developer Kit | 45 |
| 6.6 | ReSpeaker Microphone Array v2.0 | 46 |
| 6.7 | Interface Board | 47 |
| 6.8 | Schematic diagram of Jetbot | 47 |
| 7.1 | Graphical User Interface of STM32CubeMX Software | 49 |
| 7.2 | X-CUBE-AI expansion package | 50 |
| 7.3 | Keil μ Vision5 Integrated Development Environment | 51 |
| 7.4 | DFSDM Filter-0 configuration for Channel 1 | 52 |
| 7.5 | Analyzing imported trained neural network | 53 |
| 7.6 | C code to enable <i>printf</i> to transmit over USART1 | 54 |

| | | |
|------|--|----|
| 7.7 | Code to collect sample data for 1 second | 54 |
| 7.8 | Real Fast Fourier Transform with CMSIS-DSP library | 56 |
| 7.9 | Energy spectrum calculation | 56 |
| 7.10 | Calculate type-II DCT and apply lifter | 57 |
| 7.11 | Header files for AI implementation | 57 |
| 7.12 | Implementing Neural Network using X-CUBE-AI | 59 |
| 7.13 | Bare-metal implementation for KWS | 60 |
| 8.1 | Class diagram to read data from microphone array | 62 |
| 8.2 | Python script to write microphone array | 63 |
| 8.3 | Class diagram for motor driver | 64 |
| 8.4 | Jetbot keyword recognition and drive flowchart | 65 |
| 9.1 | Single feature from feature-bank of keyword “go” | 67 |
| 9.2 | Training and validation loss | 68 |
| 9.3 | Training and validation accuracy | 69 |
| 9.4 | Confusion Matrix on test data-set | 70 |
| 9.5 | Serial output on prediction | 70 |
| 9.6 | Prediction time in microcontroller | 71 |
| 9.7 | Running robot from terminal | 72 |
| 9.8 | Prediction time in Jetbot | 73 |

ABBREVIATIONS

| | |
|-------|--|
| ANN | Artificial Neural Network |
| IoT | Internet of Things |
| DNN | Deep Neural Network |
| CNN | Convolutional Neural Network |
| KWS | Keyword Spotting |
| DSP | Digital Signal Processing |
| AI | Artificial Intelligence |
| DFT | Discrete Fourier Transform |
| IC | Integrated Circuit |
| PDM | Pulse-Density Modulation |
| ADC | Analogue to Digital Converter |
| DAC | Digital to Anaglogue Converter |
| DFSDM | Digital Filter for Sigma Delta Modulator |
| RAM | Random Access Memory |
| GFLOP | Giga Floating Point Operation |
| MEMS | Microelectromechanical system |
| USB | Universal Serial Bus |
| GPIO | General Purpose Input Output |
| SPI | Serial Peripheral Interface |
| I2C | Inter IC Communication |
| GUI | Graphical User Interface |
| ONNX | Open Neural Network Exchange |
| MACC | Multiply-Accumulate |
| FFT | Fast Fourier Transform |
| MFCC | Mel-Frequency Cepstral Coefficient |
| DSCNN | Depthwise Separable Convolutional Neural Network |

ABSTRACT

Interconnected devices are becoming attractive solutions to integrate physical parameters and making them more accessible for further analysis. Edge devices, located at the end of the physical world, measure and transfer data to the remote server using either wired or wireless communication. The exploding number of sensors, being used in the Internet of Things (IoT), medical fields, or industry, are demanding huge bandwidth and computational capabilities in the cloud, to be processed by Artificial Neural Networks (ANNs) – especially, processing audio, video and images from hundreds of edge devices. Additionally, continuous transmission of information to the remote server not only hampers privacy but also increases latency and takes more power. Deep Neural Network (DNN) is proving to be very effective for cognitive tasks, such as speech recognition, object detection, etc., and attracting researchers to apply it in edge devices. Microcontrollers and single-board computers are the most commonly used types of edge devices. These have gone through significant advancements over the years and capable of performing more sophisticated computations, making it a reasonable choice to implement DNN. In this thesis, a DNN model is trained and implemented for Keyword Spotting (KWS) on two types of edge devices: a bare-metal embedded device (microcontroller) and a robot car. The unnecessary components and noise of audio samples are removed, and speech features are extracted using Mel-Frequency Cepstral Coefficient (MFCC). In the bare-metal microcontroller platform, these features are efficiently extracted using Digital Signal Processing (DSP) library, which makes the calculation much faster. A Depthwise Separable Convolutional Neural Network (DSCNN) based model is proposed and trained with an accuracy of about 91% with only 721 thousand trainable parameters. After implementing the DNN on the microcontroller, the converted model takes only 11.52 Kbyte (2.16%) RAM and 169.63 Kbyte (8.48%) Flash of the test device. It needs to perform 287,673 Multiply-and-Accumulate (MACC) operations and takes about 7ms to execute the model. This trained model is also implemented on the robot car, Jetbot, and designed a voice-controlled robotic vehicle. This robot accepts few selected voice commands such as “go”, “stop”, etc. and executes accordingly with reasonable accuracy. The Jetbot takes about 15ms to execute the KWS. Thus, this study demonstrates the implementation of

Neural Network based KWS on two different types of edge devices: a bare-metal embedded device without any Operating System (OS) and a robot car running on embedded Linux OS. It also shows the feasibility of bare-metal offline KWS implementation for autonomous systems, particularly autonomous vehicles.

1. INTRODUCTION

Internet of Things (IoT) or network interconnected devices are growing faster with the advancements in wireless networking technologies. The number is expected to cross about 125 billion by 2030 [1]. It connects a massive number of sensors and devices in cloud data centers. However, it is gradually becoming a challenging task for the clouds to handle this big data. It is required to deliver a huge computation power, which is unquestionably a serious challenge. Additionally, the increasing demand for data traffic is also touching the global maximum limits. Especially, applications that need continuous monitoring such as keyword spotting (KWS) from speech data. This data might include personal information and sending it to the cloud raise a serious concern with privacy. Edge computing has proven to be an effective way out of this problem. Instead of sending the data to the cloud, it performs computation by itself and thus overcomes the issues with bandwidth cost, privacy, and scalability [2], [3]. However, edge devices often suffer from limited computation and storage capability. It also needs to provide high accuracy outputs in real-time.

KWS on edge devices has already proven to be very useful to interact with electronic devices, for example “Google Home” and “Amazon Echo.” Only after detecting a keyword, such as “Okay Google” or “Alexa,” do these devices typically go online or record speech data and send it to the cloud. KWS is also very popular to interact with automated vehicles. Because of the unpredictable nature of cellular networks, it is not possible to maintain the connection between the vehicle and the cloud servers all the time. As the KWS does not need any internet connectivity, it can interact with the vehicle without any problem. These devices also need to be robust and noise resistant to implement in the real world. Deep Neural Networks (DNNs) have shown very high accuracy in complex applications like these. As the KWS mainly deals with time-series data, recurrent neural networks (RNN) provides a very good response for this type of application. But, this type of network needs high computation power and storage. The RNN neuron requires eight times more weight and complexity than that of a standard Convolutional Neural Network (CNN) [4]. As it emerges from a speaker’s mouth, nose, and cheeks, the speech signal is a one-dimensional function where air pressure varies with time. But feature extraction enables it to act as a single-

channel image. So, it can achieve very high efficiency for CNNs as well. Research is going on to implement CNN for KWS application and the efficiency is improving over time. These open up the possibility to successfully implement KWS on edge devices.

The DNNs have successfully been implemented on edge devices for KWS and the accuracy is going higher. In [5], [6], the authors provided a comprehensive study for different neural networks, such as CNN, LSTM, RNN, etc. It provides a comparison of different networks considering accuracy, computational complexity, and memory footprint. They also implemented the model on embedded hardware, 32-bit ARM microcontroller and achieved the best performance for a CNN-based network, Depthwise Separable Convolutional Neural Network (DSCNN). This network is a modified version of the MobileNet. In [7] the authors demonstrated MobileNet to be very efficient in classifying 2-D spatial data, such as image classification. It demonstrated a huge reduction in the computational requirement, which makes it a good choice for resource constraint devices. In [8], the authors proposed RNN based neural network, EdgeRNN, and implemented it in an edge device, Raspberry Pi. They used 1D CNN to process the frequency domain spatial information from the speech signal. Then RNN was used to process the spatial feature data. In order to reduce the power consumption and audio processing, the authors proposed the Sinc-Convolution approach to extract the features from raw audio samples in [9]. This layer is followed by a group of DSCNN to finally classify the keyword. Alongside KWS, sound recognition can also be very helpful to monitor the behavior or health of different animals or insects. In [10], the authors used ARM Cortex-M4 based microcontroller to monitor the health of bee families by analyzing sound with an ANN.

In this research, two types of edge devices- Linux based single-board computer, Jetson Nano, and a bare-metal microcontroller board are considered to implement DNN for KWS. A DNN model is proposed to reduce the computational complexity and memory footprint. This DNN is mainly based on DS-CNN and adopted from MobileNet [7]. The input shape of MobileNets is not suitable for speech features of reduced asymmetric shape. The proposed model achieved an accuracy of about 91% after training on Google Speech Command dataset version 2[11]. The feature extraction from an audio signal is also a calculation-intensive process. In order to reduce the computational cost, only 16 filters are utilized to calculate the

energy spectrum. The feature extraction process includes complex mathematical operations, such as FFT calculation, Discrete Cosine Transform, matrix multiplication, and so on. It is significantly challenging to incorporate these modified calculations in low-level C language. Using a DSP library, feature extraction is performed in a microcontroller that results in a very fast calculation. Dynamic memory allocation is used to reduce the stress on the limited memory of the microcontroller. As the DNN is written in Python 3 programming language, it is first converted into standard C equivalent code. Finally, it is implemented in the microcontroller to design a continuous KWS device. This model is also implemented in an embedded Linux-based robot car, Jetbot. It runs on Jetson Nano Development board, supported by the Linux OS, and has a GPU with a computational capacity of 0.5 TFLOPs. In this car, the DNN is implemented using Python 3 language. It reads the voice signal with a microphone array and predicts with the DNN model. The car is driven by two DC motors, which are controlled by an I2C based motor driver. This KWS system finally controls the movement of the robot with voice commands.

The second chapter of the thesis discusses some of the important neural networks and functions. The feature extraction process from the audio signal is discussed in Chapter 3. A DNN model is proposed in Chapter 4. It also discusses the basic building blocks of the DNN model. The whole process of training the model is discussed in Chapter 5. This process includes file organization, dataset preparation, model creation, and training the model with the datasets. Chapter 6 discusses the hardware used in this project, the bare-metal 32-bit ARM microcontroller, and Jetbot. Chapter 7 and Chapter 8 are the most important chapters. These two chapters discuss the implementation of the trained DNN model on microcontroller and Jetbot car respectively. There are also codes and diagrams required to clearly describe the process. Chapter 9 discusses the experimental results and outputs. Finally, the last chapter concludes the whole research.

2. NEURAL NETWORKS

Neural Networks have been proven an effective and efficient technique for automatic KWS. This chapter includes some of the important neural networks, layers and functions.

2.1 Convolutional Neural Network

Convolutional Neural Networks (CNNs) usually consists of a stack of convolutional layers with normalization and pooling layers. In order to establish cross connection to the output features of this stack, fully connected layers are also added at the end [12]. For small scale 2D spatial data classification problem, this network is very popular and high accuracy. Very deep CNN shows better performance for KWS applications in noisy and dynamic environment. In [13], the authors demonstrated a deep CNN that only uses the static features of speech signal. Thus, the model was more resistant to noisy environment. Figure 2.1 shows a basic CNN architecture, where the convolutional layers are followed by activation and pooling layers.

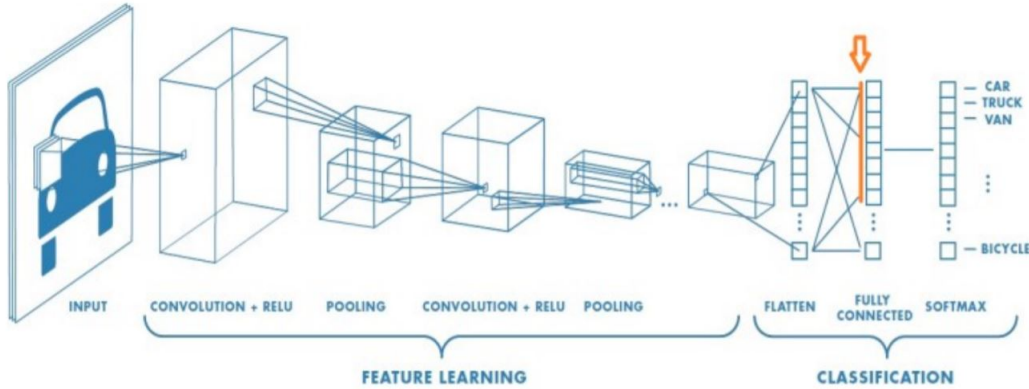


Figure 2.1. Basic CNN architecture[14]

2.2 Spatial Convolution Layer

A convolution layer serves as the fundamental element of CNN model. It extracts the features from a 2D spatial data by multiplying with a kernel. Figure 2.2 shows the convolution of a 5×3 feature matrix with a 3×2 kernel. This convolution results to an output

feature of dimension 3×2 . The output pixel value of a convolution layer can be found by Equation 2.1.

$$V = \left| \frac{\sum_{i=1}^n \left(\sum_{j=1}^m k_{ij} f_{ij} \right)}{K} \right| \quad (2.1)$$

where, f_{ij} represents the values of the feature map and k_{ij} represents the kernel values at position (i,j) . The kernel size is represented by m and n . K is the sum of the kernel coefficients, where it must not be zero.

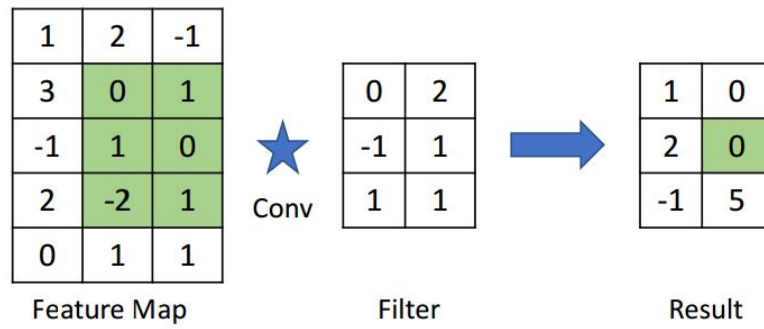


Figure 2.2. Applying filter in convolution layer

2.3 Max Pooling

A max pooling layer takes the largest data point within the selection area. It is very helpful to reduce noise. For large images, it can also be used to reduce the shape. Figure 2.3 shows the implementation of a max pooling layer in a 5×4 feature map. It uses a 2×1 kernel that results an output feature of size 5×2 .

2.4 Activation Function

The activation function add a non-linearity property to the Neural Network model. The relationship between the hidden and output layers are linear. However, natural problems are mostly non-linear, especially the classification problems. Some of the most common activation functions are Rectified Linear Unit (ReLU), Sigmoid, Softmax etc.

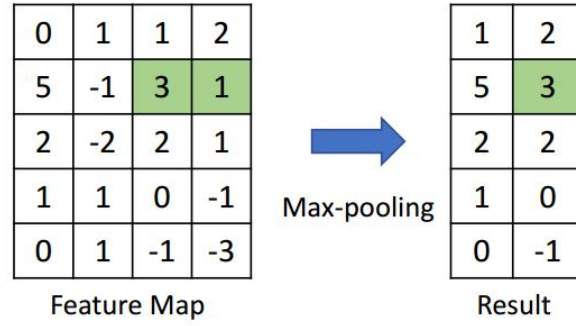


Figure 2.3. Applying Max-pooling layer

2.4.1 Sigmoid

Sigmoid activation function can be defined by Equation 2.2. It maps any real valued input in a range of 0 to 1. In Figure 2.4, for an input value ranging from -2 to 2 , the output is almost linear. However, it begins to saturate if the input goes beyond this boundary.

$$f(x) = \frac{1}{1 + e^{-x}} \quad (2.2)$$

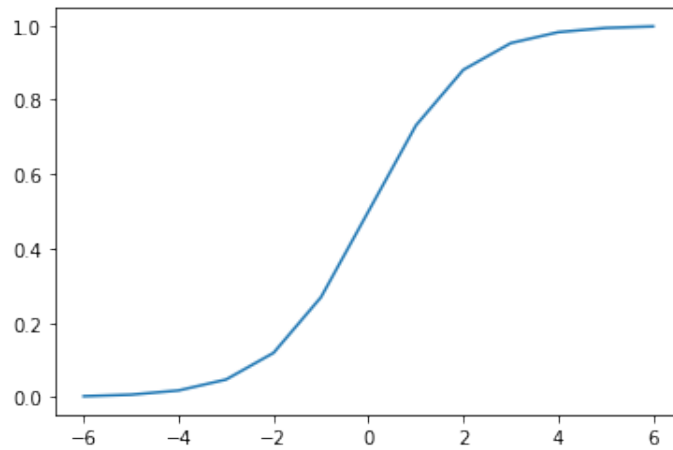


Figure 2.4. Sigmoid activation function

2.4.2 ReLU

In ReLU activation function, the output becomes zero for the negative values and multiplied by 1 for positive values. It is widely used function in deep learning and shows better performance than the sigmoid activation function. Figure 2.5 shows the input-output relationship of ReLU activation function. This function is defined by Equation 2.3.

$$f(x) = \begin{cases} 0 & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases} \quad (2.3)$$

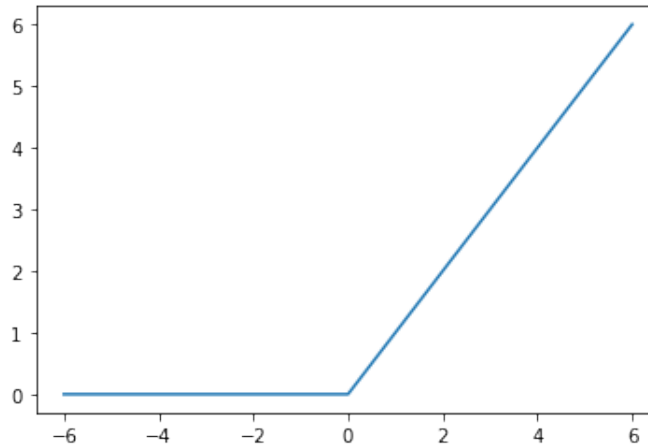


Figure 2.5. ReLU activation function

2.4.3 Softmax

For classification problems, softmax is a very important activation function. It converts a vector of K real elements into output values that sums to 1. Equation 2.4 shows the softmax function σ accepts a vector \vec{z} and converts the elements to values that sum together to 1.

$$\sigma(\vec{z})_i = \frac{e^{z_i}}{\sum_{j=1}^K e^{z_j}} \quad (2.4)$$

3. AUDIO PROCESSING

In any speech or voice recognition application, the first or basic step is to extract features from the audio samples. It extracts the necessary linguistic information while leaving the others that carry information such as emotion, background noise, etc. The key to understanding the voice signal is the process how human generates sounds to communicate with each other. The acoustic signal generated by using the vocal tract including the tongue, teeth, etc. determines the generation of a phoneme. This sound is inherently enclosed in a sort of time power spectrum envelope. In order to recognize voice data, such as speech command, the speech features need to be understood first.

The major challenge to deal with an audio signal is its dynamic behavior; it always changes with time. However, this change is not very rapid. On a short time scale, about several milliseconds, an audio signal does not change very much. It opens an opportunity to analyze the signal just like an ordinary static signal.

Mel Frequency Cepstral Coefficient (MFCC) is a very widely used feature extraction method for speech recognition applications [15]. It provides a 2D feature matrix, similar to a grayscale image, which makes it a suitable choice to implement in a Neural Network based keyword spotting system.

MFCC is one type of cepstral representation of voice signals. The main distinction between the original cepstrum and the Mel-Frequency Cepstrum (MFC) is the frequency band distribution. It was inspired by the human auditory system. Humans are good at recognizing small variations in pitch when the frequencies are low, nearly 1kHz. However, it is difficult for a human to notice changes for higher frequencies. Considering this fact makes the features more like a human ear. Equation 3.1 converts the linear frequency to Mel-scale and vice versa for Equation 3.2.

$$M(f) = 1125 \ln \left(1 + \frac{f}{700} \right) \quad (3.1)$$

$$M^{-1}(m) = 700 \left(e^{\frac{m}{1125}} - 1 \right) \quad (3.2)$$

This MFCC feature extraction method involves several steps. For simplicity, these steps are discussed assuming an input audio signal of 1 second with a sampling rate of 16kHz.

Step 1: This step begins with framing the input signal. Taking a 32ms long frame for that 16kHz signal gives $16000 \times 0.032 = 512$ samples. It is also important to take a small overlapping between two frames. For instance, consider an overlapping or stride of 256 points. So, the first frame will begin from sample 0. But the second frame will begin from 257th sample and it keeps going like that until it reaches the end. If the last frame runs out of sample, it is padded with zeros. The number of output frames, T , can be found by Equation 3.3.

$$T = \frac{L - l}{s} + 1 \quad (3.3)$$

Here, L is the length of the input audio, l is the length of each frame and s is the stride. By using the values from this example, the total number of frame, T , becomes 62.

Step 2: A Discrete Fourier Transform is taken for each of the small frames. Equation 3.4 is used to calculate the Fourier Transform. It converts the time domain signal in frequency domain.

$$S_i(k) = \sum_{n=1}^N s_i(n) e^{-j2\pi kn/N} \quad 1 \leq k \leq K \quad (3.4)$$

where, $s_i(n)$ represents the time domain small signal frame, $S_i(k)$ is the signal in frequency domain, and i is the frame number.

Now, if $P_i(k)$ is the power spectrum of any frame i , then it is obtained by the following equation:

$$P_i(k) = \frac{1}{N} |S_i(k)|^2 \quad (3.5)$$

It is also known as periodogram estimation of the power spectrum. It is done by performing a 512 point real FFT and keeping first 257 coefficients.

Step 3: A Mel-spaced filter-bank is calculated in this step. It is a set of triangular filters and applied to the periodogram estimate as obtained from step 2. Each of these filters is a

vector of 257 elements. Each vector has certain elements with non-zero values only within the frequency band of interest. Except that, mostly it is populated with zeros. For instance, taking a filter-bank containing 16 filters provides a filter-bank as shown in Figure 3.1. It filters the signal from 0Hz to 8kHz with higher resolution for lower frequencies.

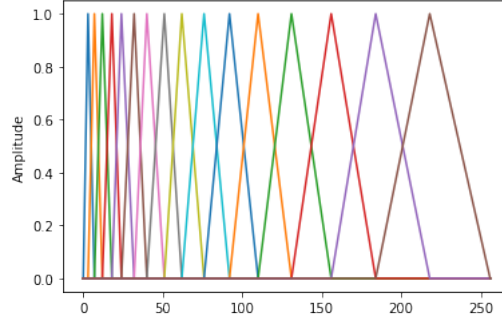
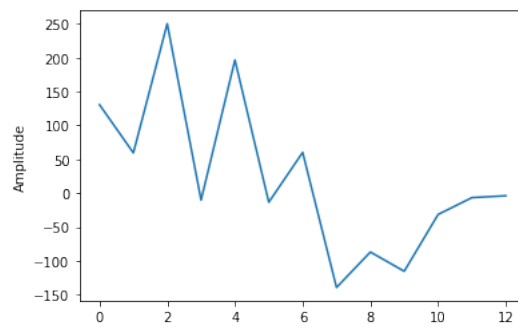


Figure 3.1. Mel-spaced filter-bank of 16 filter

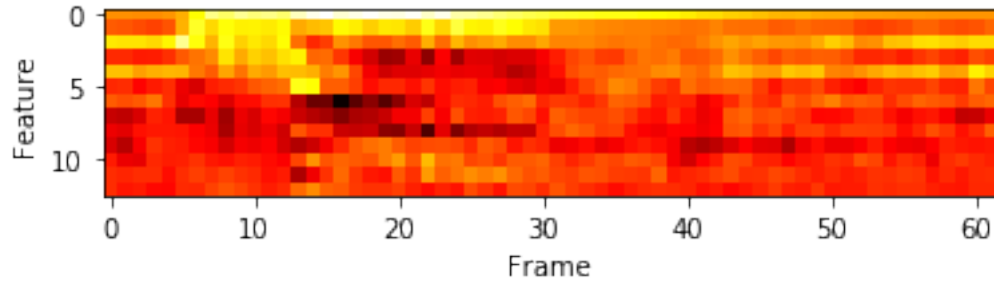
Step 4: Now it needs to implement this filter-bank to calculate energies in each band. The band energy is computed by multiplying the periodogram estimate with the filter-bank. It indicates the amount of energy in a specific filter. In order to scale down the numbers, logarithm is taken for each of the 16 energies. It provides a total of 16 log filter-bank energies.

Step 5: A Discrete Cosine Transform (DCT) is taken of these 16 log filter-bank energies. It provides 16 cepstral coefficients. But the higher order term is not usually significant as those are mostly zero. Only first 13 is kept while discarding the rests.

This illustration results a Mel Frequency Cepstral Coefficients of size 62×13 . Figure 3.2 shows the MFCC spectrogram for an audio sample of keyword “go”. Figure 3.2a shows the features from a single frame of the MFCC. An spectrogram of all features from all frames is shown in Figure 3.2b.



(a) Features from one frame



(b) MFCC Spectrogram

Figure 3.2. Mel Frequency Cepstral Coefficients for keyword “go”

4. DNN ARCHITECTURE

The core layers of the proposed DNN, which is based on DSCNNs, will be discussed first in this chapter. The model structure, as well as important parameters, will then be discussed.

4.1 DSCNN layer

This model is entirely based on the DSCNN, which is a comparatively newer version of separable convolution. Laurent Sifre developed the depthwise separable convolution and described detailed experimental results in his Ph.D. thesis, section 6.2 [16]. This structure factorizes the 2D spatial convolution into depthwise convolution and pointwise convolution. In this application, the input has only 1 channel and the depthwise convolution applies a single filter on it. Then a 1×1 2D convolution is used to combine the outputs of the depthwise convolution. The basic difference between the standard convolution and depthwise convolution is that it introduces two separate layer filtering and combining. Figure 4.1 shows the difference between the filter shape of standard 2D convolution and depthwise separable convolution. It reduces the model size, as well as the computational complexity of the network.

For single channel spatial data, a standard 2D convolution layer takes an input of $D_x \times D_y \times 1$ feature map F and gives $D_m \times D_n \times N$ feature map G , where D_x and D_y are width and height of the input feature map, D_m and D_n are the width and height of the output feature map, and N is the output depth.

A single channel 2D convolution kernel K have the size of $D_k \times D_k \times 1 \times N$ where D_k is the spatial kernel size assumed to be squared, and N is the number of output channels.

The output feature map can be computed by Equation 4.1.

$$G_{k,l,n} = \sum_{i,j,1} K_{i,j,1,m} \cdot F_{k+i-1,l+j-1,1} \quad (4.1)$$

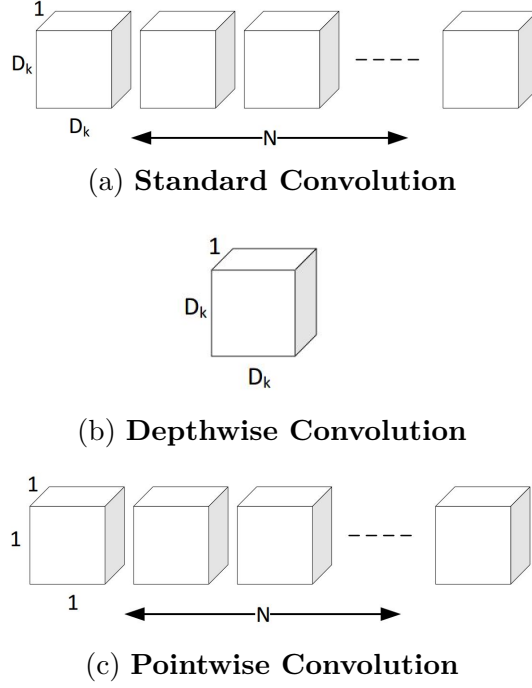


Figure 4.1. Filter shape in standard convolution and depthwise separable convolution

The computational cost for these convolutions would be given by Equation 4.2. Here the cost depends on multiplication of the input and output feature map size, and number of output channels.

$$D_x \cdot D_y \cdot N \cdot D_k \cdot D_k \quad (4.2)$$

The DSCNN also finds out the relationship between these terms in a different way. It breaks the interaction between the output channels and kernel size by using depthwise separable convolutions. In the previously mentioned standard 2D convolution, the filtering and combination take place in a single step at a cost of computational complexity. However, the depthwise separable convolution splits the filtering and combination steps and reduces the computational cost.

The depthwise separable convolutions consist of two layers: depthwise and pointwise convolutions. The depthwise convolutions used to apply only one filter. Pointwise convolution is just like a standard 2D convolution, except the kernel size is kept 1×1 . It efficiently finds

the linear combination of the output of the previously mentioned depthwise layer. After each of this block, batch normalization and rectified linear unit layers are used.

The output feature map of depthwise convolution with one filter can be calculated from Equation 4.3

$$\hat{G}_{k,l,1} = \sum_{i,j} \hat{K}_{i,j,1} \cdot F_{k+i-1,l+j-1,1} \quad (4.3)$$

where, \hat{K} is the kernel size of the depthwise convolution $D_k \times D_k \times 1$ that is applied to produce a feature map \hat{G} .

The computational cost of the depthwise convolution is:

$$D_k \cdot D_k \cdot D_x \cdot D_y \quad (4.4)$$

This convolution only filters the inputs along the channel. In order to create a new feature map, all of the outputs need to be combined. A pointwise convolution does that task. It computes a linear combination from the outputs of the depthwise convolution and creates new features.

This combination of both depthwise and pointwise convolution is known as depthwise separable convolution. The computational cost of this block can be found by Equation 4.5. It is the sum of costs from both depthwise and pointwise convolution.

$$D_k \cdot D_k \cdot D_x \cdot D_y + N \cdot D_x \cdot D_y \quad (4.5)$$

By taking the ratio of computation cost for depthwise separable convolution and standard 2D convolution, we get the reduction fraction of the computation as given by Equation 4.6.

$$\frac{D_k \cdot D_k \cdot D_x \cdot D_y + N \cdot D_x \cdot D_y}{D_x \cdot D_y \cdot N \cdot D_k \cdot D_k} = \frac{1}{N} + \frac{1}{D_k^2} \quad (4.6)$$

In this proposed DNN, 3×3 kernel is used for depthwise separable convolution. It results to 8 to 9 times reduction for the computation than standard convolution.

4.2 Model Architecture

This proposed DNN was adopted from MobileNet structure [7]. The original model was developed for 3 channel image classification of size 224×224 . However, the feature matrix returns a single channel spatial output of shape 62×13 . In order to implement the MobileNet structure for this KWS application, the filter size was reduced and few layers were discarded to avoid unnecessary computation. The proposed DNN architecture is given in Table 4.1.

A batch normalization and rectified linear unit layer are added after each of these layers, except the last three layers as shown in 4.2. This DNN model consists of 27 layers taking depthwise and pointwise convolution as separate layers. Finally, this network yields 731,111 parameters among which 721,127 are trainable.

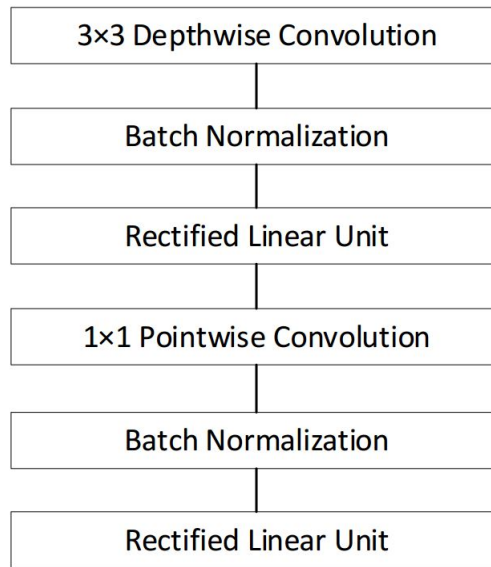


Figure 4.2. Depthwise Separable Convolution with batchnorm and ReLU

Table 4.1. Proposed DNN architecture

| Layer | Filter Shape |
|----------|------------------------------------|
| Conv2D | $3 \times 3 \times 1 \times 32$ |
| ConvDW | $3 \times 3 \times 32$ |
| Conv2D | $1 \times 1 \times 32 \times 32$ |
| ConvDW | $3 \times 3 \times 64$ |
| Conv2D | $1 \times 1 \times 32 \times 64$ |
| ConvDW | $3 \times 3 \times 64$ |
| Conv2D | $1 \times 1 \times 64 \times 64$ |
| ConvDW | $3 \times 3 \times 128$ |
| Conv2D | $1 \times 1 \times 64 \times 128$ |
| ConvDW | $3 \times 3 \times 128$ |
| Conv2D | $1 \times 1 \times 128 \times 128$ |
| ConvDW | $3 \times 3 \times 128$ |
| Conv2D | $1 \times 1 \times 128 \times 128$ |
| ConvDW | $3 \times 3 \times 128$ |
| Conv2D | $1 \times 1 \times 128 \times 128$ |
| ConvDW | $3 \times 3 \times 256$ |
| Conv2D | $1 \times 1 \times 128 \times 256$ |
| ConvDW | $3 \times 3 \times 256$ |
| Conv2D | $1 \times 1 \times 256 \times 256$ |
| ConvDW | $3 \times 3 \times 256$ |
| Conv2D | $1 \times 1 \times 256 \times 256$ |
| ConvDW | $3 \times 3 \times 256$ |
| Conv2D | $1 \times 1 \times 256 \times 256$ |
| ConvDW | $3 \times 3 \times 512$ |
| Conv2D | $1 \times 1 \times 256 \times 512$ |
| ConvDW | $3 \times 3 \times 512$ |
| Conv2D | $1 \times 1 \times 512 \times 512$ |
| Avg Pool | — |
| FC | 3597 |
| Softmax | Classifier(7) |

5. TRAINING MODEL

In this chapter, the training process of the DNN model will be discussed using a deep learning framework, TensorFlow version 2.0.0. Google Speech Command dataset version 2 [11] was used that contains spoken words for 35 different keywords. A total of 105,829 keywords were recorded from 2,618 speakers. Each sample was encoded as 16-bit single-channel PCM values, at a rate of 16kHz samples per second. The files were stored in WAVE file format with a length of one second or less. When the files are compressed, it takes about 2.7GB and the uncompressed version takes around 3.8GB on disk. Spotting keywords is different from the speech recognition of full sentences, which usually requires a very large dataset. In [17], the author describes the process of capturing the speech commands. Studio-recorded samples would be unrealistic because of the absence of background noise. The author captured all of the samples uttered from a mobile phone or laptop microphones. Some of the most commonly used keywords in robotics application was captured in this dataset; “Yes”, “No”, “Up”, “Down”, “Left”, “Right”, “On”, “Off”, “Stop”, and “Go”. The previous version of this dataset[18] had a total of 64,727 utterances from 1,881 speakers. After conducting training on both datasets, the model was evaluated on a test dataset that showed a significant improvement in accuracy from about 83% to 88%. All of which makes this dataset a very good choice for this application.

The following four basic steps were performed in order to train this model:

1. Organizing Files
2. Generating Datasets
3. Creating Model
4. Training Model

5.1 Organizing Files

All of the files in the dataset[11] were separated in 35 different folders, six of which were the the target commands. So, in order to make them useful, all files from the six directory-

“go”, “stop”, “left”, “right”, “up”, and “down” needs to be listed without any change. Rest of the files need to be listed under a name “unknown” command.

This dataset also comes with a list of validation and test files written in a *.TXT file format. Each file is identified with file name and directory. Training files were separated by discarding validation and test files from all available files. Those files were separated by seven keywords as mentioned above for each dataset. Finally, a data-frame was created for each of the training, validation and testing dataset containing information about individual file-label and file name with directory. The Python code for organizing the dataset is provided in Figure 5.1.

```
# Record all available files , along with extension
all_files = []
for root, dirs, files in os.walk(train_dir):
    all_files += [root + '/' + f for f in files if f.endswith('.wav')]

# Read file information from TXT file
test_files = pd.read_csv(train_dir + '/testing_list.txt',
                        sep=" ", header=None)[0].tolist()
val_files = pd.read_csv(train_dir + '/validation_list.txt',
                        sep=" ", header=None)[0].tolist()

# Add .wav extension
test_files = [os.path.join(train_dir, f)
              for f in test_files if f.endswith('.wav')]
val_files = [os.path.join(train_dir, f)
             for f in val_files if f.endswith('.wav')]

# Find the training files
train_files = list(set(all_files) - set(val_files) - set(test_files))

# Create labels for all files
test_labels = [_getFileCategory(file) for file in test_files]
val_labels = [_getFileCategory(file) for file in val_files]
train_labels = [_getFileCategory(file) for file in train_files]

# Test, validation and training file information in pandas DF
train_info = pd.DataFrame(list(zip(train_files, train_labels)), columns =
['files', 'labels'])
test_info = pd.DataFrame(list(zip(test_files, test_labels)), columns = [
'files', 'labels'])
val_info = pd.DataFrame(list(zip(val_files, val_labels)), columns = [
'files', 'labels'])
```

Figure 5.1. Code for organizing dataset files

5.2 Generating Datasets

The raw audio files are not suitable to directly feed into the Neural Network. It needs to be pre-processed before training. The pre-processing involves randomizing the input files and feature extraction.

A function was created in Python language to extract the features from an audio sample. The process begins with framing the audio samples. The frame length was chosen 512 with a stride of 256 points. A power spectrum was calculated for each frame with 512 point real FFT. A mel-spaced filter-bank was created consisting 16 filters. This filter-bank captured frequency of ranging 0Hz to 8000Hz. As the speech signal has more low frequency components, more filters were implemented for lower frequency than the higher counterpart. The energy of each filter band was calculated by multiplying the power spectrum with filter-bank. As the shape of power spectrum and filter-bank are (62, 257) and (16, 257), the multiplication yielded an energy-bank of shape (62, 16). Finally, a type-II DCT was computed from this energy-bank to get the Mel-Frequency Cepstral Coefficients of the audio signal. After taking 13 coefficients from each row, the *calcMFCC()* function returned a feature matrix of shape (62, 13). Figure 5.2 shows the implementation code in Python to extract MFCC from any audio file of length 16000 samples.

The labels and file names were chosen randomly among the available choices while creating datasets for training, validation and testing. The files were checked to have a same length of 16000. This length of files allows to have a feature bank of (62, 13). Smaller files were padded with 0s and larger files were cropped up to first 16000 elements. These features were scaled from -127 to +128. This allowed the data to fit into 8-bit integer datatype while applied in the microcontroller. The features were then reshaped so that it can be feed into the neural network model. The output labels were also hard coded and categorized for 7 classes. In the end, the training, validation and testing dataset contained 42421, 4990 and 5502 samples respectively.

Finally, the reshaped data were passed through an iterator as shown in Figure 5.3. It allows to load the data from a NumPy array and convert it in a stack of tensors. The datasets

```

# Function to calculate MFCC
def calcMFCC(samples, rate=16000):
    # create frames
    frames = sigproc.framesig(samples, frame_len=512, frame_step=256)

    # Find Power Spectrum
    ffts = np.fft.rfft(frames, 512)
    power_fft = np.abs(ffts)
    power_fft = 1/512*np.square(power_fft)

    # Create filter-bank
    fb = get_filterbanks(16, 512, rate, 0, rate/2)    #16*257

    # Calculate Energy bank
    energy = np.dot(power_fft, fb.T)    # 62X257 dot 257X16 >> 62X16
    energy = np.where(energy == 0, np.finfo(float).eps, energy) # if feat is
    zero, take a small number

    # Take log of Energy
    energy_log = np.log(energy)

    # Calculate type-II DCT
    flipped = np.flip(energy_log, 1)
    reordered = np.append(energy_log, flipped, axis=1)
    features = np.fft.fft(reordered, 2*config.nfilt)
    features = features.real[:, :config.nfeat]
    # lift up the high frequency dct components
    lift = 1 + 4 * (np.arange(config.nfeat)) # used previously

    return lift*features

```

Figure 5.2. Code to calculate speech features, MFCC

were created in batches that contained 32 tensors in each batch. The training buffer were randomly shuffled between 16 tensors.

5.3 Creating Model

This Depthwise Separable Convolutional Neural Network was created in Keras library backed by TensorFlow. The architecture was adopted from MobileNet[7]. This model begins with a 2D convolution layer with 32 filter elements. The stride was chosen as (2, 2). A batch normalization layer, followed by activation layer with *relu* activation function was also added. The batch normalization layer helps to keep the mean output close the 0 and the standard deviation to 1.

```

BATCH_SIZE = 32
SHUFFLE_BUFFER_SIZE = 16

train_dataset = tf.data.Dataset.from_tensor_slices((train_data))
train_dataset = train_dataset.shuffle(SHUFFLE_BUFFER_SIZE).batch(
BATCH_SIZE)

test_dataset = tf.data.Dataset.from_tensor_slices((test_data))
test_dataset = test_dataset.batch(BATCH_SIZE)

validation_dataset = tf.data.Dataset.from_tensor_slices((validation_data))
validation_dataset = validation_dataset.batch(BATCH_SIZE)

```

Figure 5.3. Iterator to create dataset

After this 2D convolution layer, a depthwise separable CNN block was added as shown in Figure 5.4. It adds a depthwise convolutional layer with a kernel size of (3, 3) and stride of (1, 1). This layer applies a single convolutional filter along the input channel, instead of spatial axis. This type of convolutional layer does not mix information with other channels. This block also added another 2D convolutional layer. However, the kernel size was kept as (1, 1) that is called pointwise convolution.

After that 12 more DSCNN block were added with filter size ranging from 64 to 512. A global average pooling layer was added followed by a dense layer with *softmax* activation function. Thus, it provides a classification for 7 keywords for this KWS application.

5.4 Training Model

In order to fit the model with the training and validation datasets, three callback functions were called as shown in Figure 5.5. The learning rate was calculated by using Equation 5.1. The *earlystopper* stops the training when the model stopped improving. It saves time along with over-fitting the model. The *checkpointer* saves the best model with lowest validation loss.

$$L_r = L_i \times D^{\frac{1+Epoch}{E_{drop}}} \quad (5.1)$$

```

# Function to add Depthwise Separable CNN block
def get_dw_sep_block(tensor, filters, strides, alpha=1.0, name=''):
    # Depthwise
    x = DepthwiseConv2D(kernel_size=(3, 3),
                        strides=strides,
                        use_bias=False,
                        padding='same',
                        name='{}_dw'.format(name))(tensor)
    x = BatchNormalization(name='{}_bn1'.format(name))(x)
    x = Activation('relu', name='{}_act1'.format(name))(x)

    # Pointwise
    x = Conv2D(filters,
               kernel_size=(1, 1),
               strides=(1, 1),
               use_bias=False,
               padding='same',
               name='{}_pw'.format(name))(x)
    x = BatchNormalization(name='{}_bn2'.format(name))(x)
    x = Activation('relu', name='{}_act2'.format(name))(x)
    return x

```

Figure 5.4. Depthwise Separable CNN block

where, L_r is the learning rate, L_i is the initial learning rate, D is a constant less than 1, and E_{drop} is the epoch drop.

Here, the categorical crossentropy loss function was used. It provides better result for classification problems with two or more classes. For optimization, *adam* optimizer[19] was used which is computationally efficient, has little memory requirement. It is based on adaptive estimation of first and second-order moments.

Finally, the model was trained by using *fit()* method. It receives the training dataset containing input features and output labels. It also handles the callback functions and other important arguments such as validation data, batch size, and so on. Finally it returns history object containing training summary such as loss and accuracy for training and validation for every epoch.

```

def step_decay(epoch):
    initial_lrate = 0.001
    drop = 0.4
    epochs_drop = 15.0
    lrate = initial_lrate * math.pow(drop,
        math.floor((1+epoch)/epochs_drop))

    if (lrate < 4e-5):
        lrate = 4e-5

    print('Changing learning rate to {}'.format(lrate))
    return lrate

lrate = LearningRateScheduler(step_decay)
earlystopper = EarlyStopping(patience=5, verbose=1, restore_best_weights=True)
checkpointer = ModelCheckpoint('DNN_trained.h5', verbose=1, save_best_only=
    True)

```

Figure 5.5. Callbacks to fit datasets into the DNN

6. HARDWARE PLATFORMS

In this project, the KWS is performed on following devices:

1. 32-bit ARM Microcontroller and
2. Robotic vehicle powered by Jetson Nano

A brief description about these hardware platforms in the next section.

6.1 32-bit ARM Microcontroller

Microcontrollers have evolved just like other computing devices. Modern microcontrollers can provide enough computational power to implement Neural Networks on them. In [20], the author implemented a very simple fully connected multi-layer Neural Network on an 8-bit cost-effective microcontroller in 2008. So, the idea of implementing DNN is not new. However, the computational complexity of DNN is increasing rapidly. In order to meet this increasing demand, microcontrollers need to be more powerful in terms of computational complexity and storage. In [5], the authors demonstrated memory and number of operations needed for different types of Neural Network as shown in Figure 6.1. In case of ideal models, it is expected to have small memory footprint and lower number of computations needed to obtain high accuracy. In this project, STM32F769NI microcontroller was used and the trained model was successfully deployed into it for KWS.

6.1.1 STM32F769NI Discovery Board

The STM32F769NI Discovery development board was developed by STMicroelectronics. It is based on 32-bit ARM Cortex M-7 core and share applications with the STM32F7 series microcontrollers as stated in [21]. The Cortex M series processors help to create cost-sensitive and power-constrained solutions with microcontroller. The Cortex-M7 based processor is the highest-performance member of the family as it was designed for mixed-signal devices and provides very high energy efficiency. The DSP capability makes it suitable for speech processing that needs to perform a tedious mathematical operations, like DFT or

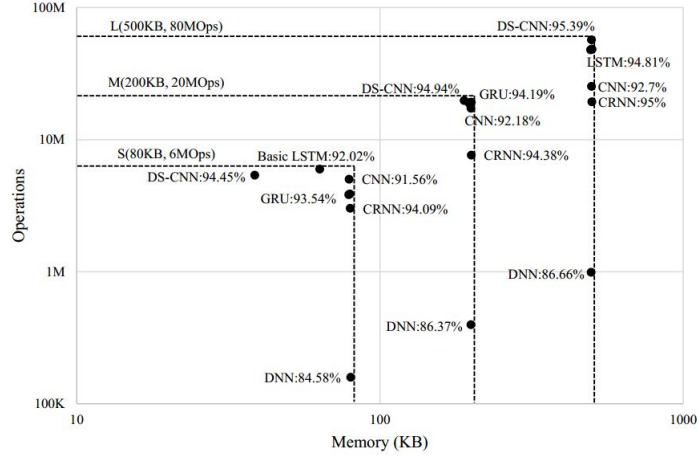


Figure 6.1. Memory and Operations requirements for different networks [5]

DCT. With the built-in floating-point unit, this processor can reduce power consumption and extend battery life by ten fold acceleration of single-precision, floating-point operations. Figure 6.2 shows the image of the development board used in this research.



Figure 6.2. STM32F769NI Discovery development board

STM32F769 Discovery board comes with four on-board microphones, which makes this device an excellent choice for voice recognition[21]. Neural Network requires relatively higher amount of flash memory and RAM, to store the model parameters or weights, compared to other embedded applications. This board provides 2 Mbytes of flash memory and 532 Kbytes of RAM, which can provide enough storage or memory requirement to classify several voice commands. Additionally, it includes 128 Mbit SDRAM, making this board more attractive

for AI applications. It has on board programmer/debugger: ST-Link, which not only connects this device with laptop/computer, but also enables the designer to program/debug. It provides huge ease of designing and testing embedded applications. It has 4 inch capacitive touch LCD display with MIPI DSI connector, which can be used to design user friendly GUI interface. All of the important features of this board is listed in Table 6.1.

Table 6.1. Important features of STM32F769 Discovery board

| S/N | Features |
|-----|---|
| 1 | Two Mbytes of Flash memory and 512+16+4 Kbytes of RAM |
| 2 | Four ST MEMS microphones on DFSDM inputs |
| 3 | Two audio line jacks, one for input and one for output |
| 4 | On-board ST-LINK/V2-1 supporting USB reenumeration capability |
| 5 | USB ST-LINK functions: virtual COM port, mass storage, debug port |
| 6 | Four inch capacitive touch LCD display with MIPI® DSI connector |

Microphone MP34DT01TR

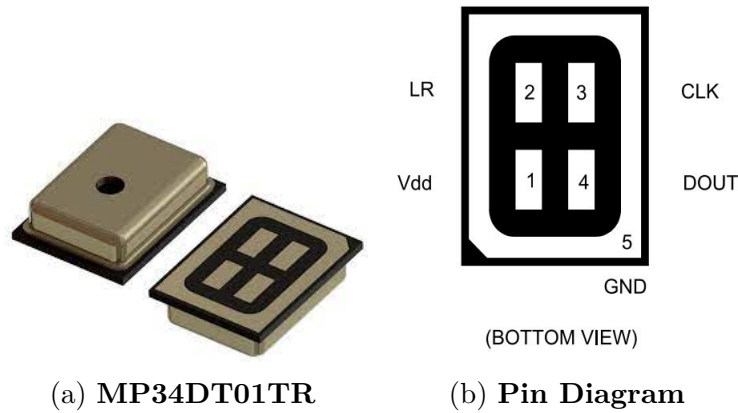


Figure 6.3. MP34DT01TR digital MEMS microphone

This board comes with four MP34DT01TR digital MEMS microphones. It is built with capacitive sensing element, consumes very low power, ultra compact, omnidirectional and easy to interface with other ICs. Figure 6.3a shows the image of the microphone, which has a dimension of 4×3×1 mm. The sensing element is made using a specialized silicon micromachining process dedicated for audio sensors. It uses a CMOS process for interfacing

that provides the output as Pulse Density Modulated signal. It has an acoustic overload point of 120 dBSPL, 63 dB SNR and -26 dBFS sensitivity.

Digital Filter for Sigma Delta Modulator

The STM32F769 microcontroller provides high performance Digital filter for sigma delta modulators (DFSDM) peripheral as an external $\Sigma\Delta$ modulator[22]. It comes with external analogue front end and digital filter, resulting greater advantage over generic ADC and widely used in audio recording with MEMS microphones. It has 4 digital filters and 8 external channels that offers to process the stream of digital signal up to 24-bit ADC resolution. This DFSDM module can be connected using several standard protocols: Manchester coded 1-wire interface and SPI interface with adjustable parameters.

The resolution and speed of conversion can be adjusted by changing few signal processing parameters: length of integrator, type of filter, order of filter, length of filter etc. It can run with two conversion mode: continuous and single conversion. It also can store data in a system RAM buffer using DMA that significantly reduces the software overhead.

DFSDM peripheral implements $Sinc^x$ filter that ends up with decreased output data rate and increased output resolution. This filter can be configured by changing the following parameters:

- Filter type and order:

- *FastSinc*
- $Sinc^1$
- $Sinc^2$
- $Sinc^3$
- $Sinc^4$
- $Sinc^5$

- Decimation Ratio or Oversampling Ratio:

- $F_{OSR} = 1-1024$, for *FastSinc* and $Sinc^x$ filter, $x = F_{ORD} = 1...3$

- $F_{OSR} = 1-215$, for $Sinc^x$ filter, $x = F_{ORD} = 4$
- $F_{OSR} = 1-73$, for $Sinc^x$ filter, $x = F_{ORD} = 5$

The transfer function of *FastSinc* and $Sinc^x$ type filters are given by equation 6.1 and 6.2 respectively:

$$H(z) = \left(\frac{1 - z^{-F_{OSR}}}{1 - z^{-1}} \right)^2 \times (1 + z^{-2 \cdot F_{OSR}}) \quad (6.1)$$

$$H(z) = \left(\frac{1 - z^{-F_{OSR}}}{1 - z^{-1}} \right)^x \quad (6.2)$$

Finally, the DFSDM module produces high resolution output signal in PCM format and the output data rate is dependent on the input serial data stream rate, and the parameters of integrator and filter. When the Fast mode is disabled, the maximum output data rate (in sample/sec) can be obtained from equation 6.3 and 6.4 for *FastSinc* and $Sinc^x$ filters respectively.

$$f_{out} = \frac{f_{clk}}{f_{OSR} \cdot (I_{OSR} + 3) + 3} \quad (6.3)$$

$$f_{out} = \frac{f_{clk}}{f_{OSR} \cdot (F_{ORD} + I_{OSR} - 1) + (F_{ORD} + 1)} \quad (6.4)$$

where, f_{clk} is the input clock rate, F_{OSR} is the filter oversampling ratio, F_{ORD} is the order of the filter, and I_{OSR} is the oversampling ratio of the integrator.

However, if the fast mode is enabled, the output data rate can be obtained by

$$f_{out} = \frac{f_{clk}}{f_{OSR} \cdot I_{OSR}} \quad (6.5)$$

This module generates final data at a maximum resolution of 24-bit. However, the processing path can be up to 32-bit long. In order to read the data, bits are right shifted at least 8-bit. This operation can be done inside of the code using bit manipulation, which would be resource intensive. An efficient alternative is to change DTRBS[4:0] bits in DFSDM_CHy-CFGR2 register, which allows right shift of incoming data up 0-31 bits. The sign bit is also maintained to have a signed integer type of sample data.

Direct Memory Access (DMA)

Direct Memory Access (DMA) is a very efficient feature of computer systems which can transfer data either from peripherals to memory or memory to memory with very high speed. It does not need any CPU assistance for data transfer, and thus CPU resources can be used for other tasks in this time.

By using a complex bus matrix architecture, the STM32F769 DMA controller provides bandwidth optimization for higher data rate. Two DMA controllers (DMA1 and DMA2) support up to 16 streams of data in total (8 for each controller), which are dedicated to manage memory for one or more peripherals. Table 6.2 shows the mapping request at channel 8 for DFSDM peripheral in DMA2 controller.

Table 6.2. Mapping Request for DMA2 controller at channel 8[22]

| Stream | Peripheral |
|----------|-------------|
| Stream 0 | DFSDM1_FLT0 |
| Stream 1 | DFSDM1_FLT1 |
| Stream 2 | DFSDM1_FLT2 |
| Stream 3 | DFSDM1_FLT3 |
| Stream 4 | DFSDM1_FLT0 |
| Stream 5 | DFSDM1_FLT1 |
| Stream 6 | DFSDM1_FLT2 |
| Stream 7 | DFSDM1_FLT3 |

DMA supports two modes of output data buffering: circular mode and double-buffer mode. In the circular mode, the output data, from peripheral or memory, starts to load at the beginning of the buffer memory, once the data transfer is completed. Two interrupts are available during this process: after finishing half, as well as full, of the buffer. This mode can be activated by the CIRC bit in the DMA_SxCR register. It needs only one buffer pointer to transfer the data. However, the double-buffer stream needs two buffer to load the data stream. After finishing data transfer to one pointer, the controller alters the memory pointer and starts from the beginning. This buffering mode can be activated by setting the DBM bit in the DMA_SxCR register.

6.2 Robotic Vehicle with Jetson Nano

Voice command recognition was also performed on a robot car, commercially known as JetBot AI Robot Car. Figure 6.4 shows the picture of the robotic vehicle. This robot is based on a single board computer device, Jetson Nano[™], developed by NVIDIA[®]. This vehicle comes with a strong and robust aluminium alloy structure. It also has a HD camera module, which can be lifted upward and downward direction along. The camera is attached with two servo motor that provides a freedom of movement along X and Y direction. There are two RGB LED strips at left and right side of the robot that can provide a colorful lighting to make it more attractive. It also provides a many example code and directions to setup the hardware correctly and configure the software packages as per user requirements[23]. The following sections give a brief description about the internal modules of this robot vehicle.

6.2.1 Jetson Nano Developer Kit

This powerful single board computer is developed by NVIDIA[®][24]. It allows to run multiple Deep Neural Network in parallel for applications like KWS, object detection, image classification and other computation intensive AI models. One of the most important feature of Jetson Nano is the 128-core NVIDIA Maxwell[™] GPU. It allows the tensors to perform computations in parallel. It has Quad-Core Arm Cortex-A57 MPCore processor, which is built with Armv8-A architecture enabled with DSP functionality. This device has a computational capability of 472 GFLOPs that is sufficient enough to run small DNN models. It has a 4 GB, 64-bit LPDDR4 RAM with a bus speed of 25.6 GB/s. It also supports external memory up to 128 GB in micro SD card. It consumes 5 Watts of power in regular mode and 10 Watts in boost mode. Figure 6.5 shows the image of this development board. This device runs on embedded Linux platform, just like another popular platform, Raspberry Pi. Both [23] and [24] provides image of the embedded Linux operating system. However, image provided by [23] includes all the necessary packages to run with the robot car and thus provides an ideal development environment.

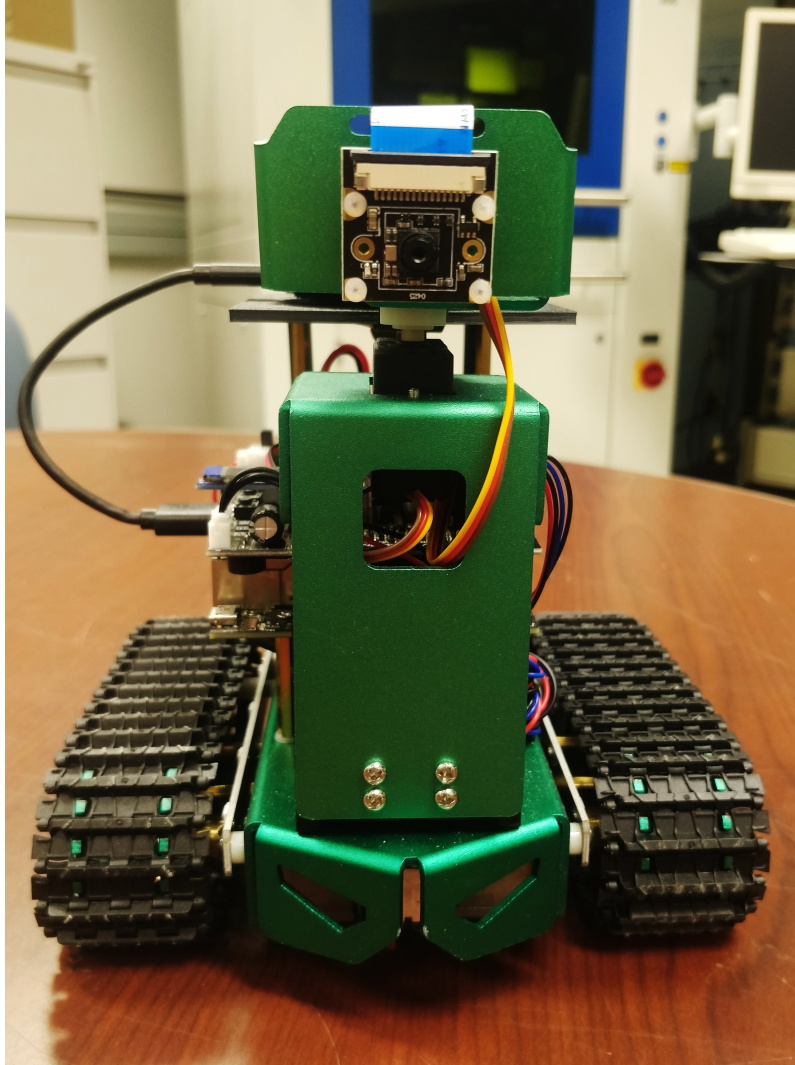


Figure 6.4. JetBot AI Robot Car

6.2.2 Microphone Array

The Jetson Nano or JetBot does not have any microphone to receive the audio signals. A microphone array, as illustrated in Figure 6.6, is used to receive the signal. The USB Audio Class 1.0 (UAC 1.0) interface is directly supported by this ReSpeaker Mic Array v2.0 module[25]. It is also supported by all major operating systems, such as Linux, Windows or MacOS.

This board can be used to detect voices as far as 5 meters, even in a noisy environment. The DSP processor, XMOS XVF-3000, enables to implement 4-mic mono echo cancellation

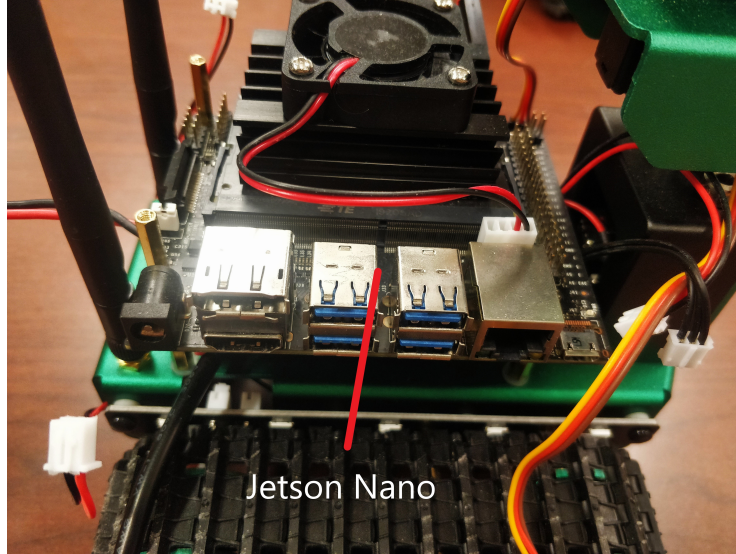


Figure 6.5. Jetson Nano Developer Kit

algorithm to extract voice data in a challenging acoustic environment. Echo cancellation, adaptive beam-forming, de-reverberation and noise suppression comes up together to efficiently pick up the true voice signal for applications like speech recognition. It also comes up with high performance MEMS digital microphones, MSM261D4030H1CPM. It has an omnidirectional sensitivity of -26 dBFS, acoustic overload point of 120 dB SPL, and SNR of 63 dB. It can also be powered directly from 5V DC supply available in USB port. All of these features makes this device an attractive solution for applications such as- smart speaker, intelligent voice assistant system, voice interacting robot and so on.

6.2.3 Interface Board

Jetson Nano comes with a 40 pin header, of which 28 pins can be used as GPIO purpose. These GPIO pins can also be used to connect with other device using different communication protocols such as- I2C, SPI or UART. However, this small computer does not provide any motor driver itself. JetBot provides an interface board consisting all the necessary hardware components that would solve the motor driver issue. This board is directly connected with the GPIO header of Jetson Nano development board. It connects a 16-channel PWM generator IC, PCA9685, via I2C bus. Each channel is capable of generating 12-bit resolution

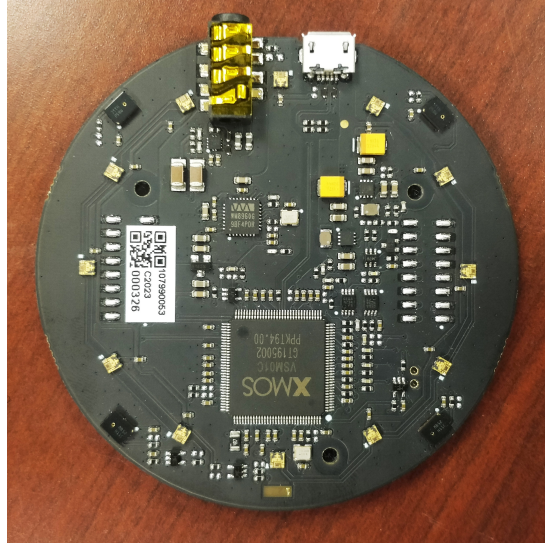


Figure 6.6. ReSpeaker Microphone Array v2.0

fix frequency PWM signal with a control frequency of 24Hz to 1526Hz with an adjustable duty cycle from 0 percent to 100 percent. Each pin can deliver up to 10mA source current if connected at 5 V power supply. However, it increases up to 25mA if connected in a totem pole or sink configuration. The right motor is controlled by channel 8 and 9 of the driver. Similarly, the left motor is controlled by channel 10 and 11 in both clockwise and counterclockwise direction.

This interface board also allows the Jetson Nano to control two servo motor using the UART protocol. As there is no feedback system in this servo motor, it only receives data and the transmit pin of the serial port is used, leaving the receive pin disconnected. This board also provides an on-board OLED display, SSD1306. It has a screen resolution of 128*32. This module is interfaced using I2C bus and the peripheral address of the OLED driver is 0x3C that stores all the register information about the driver screen.

The schematic diagram of the Jetbot robot car is shown in Figure 6.8. In this diagram, the Jetson Nano board is connected with the microphone array via a USB port. The I2C1 bus is in charge of communicating with the Motor Driver and the OLED display. Channels 8–11 of the PCA9685 driver are linked to the left and right motors, which move the car along a 2D plane.

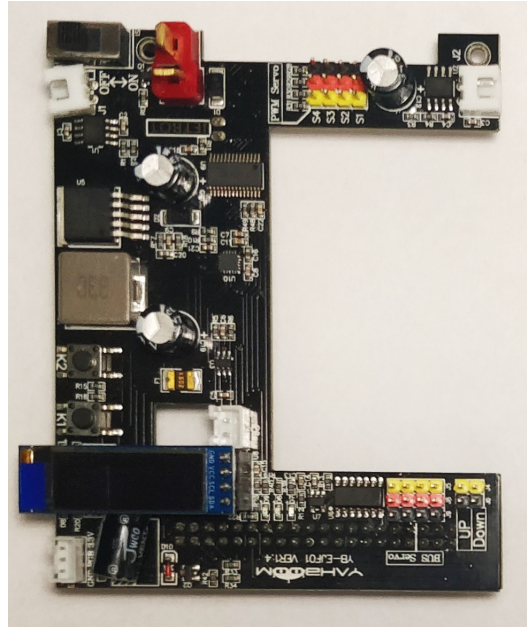


Figure 6.7. Interface Board

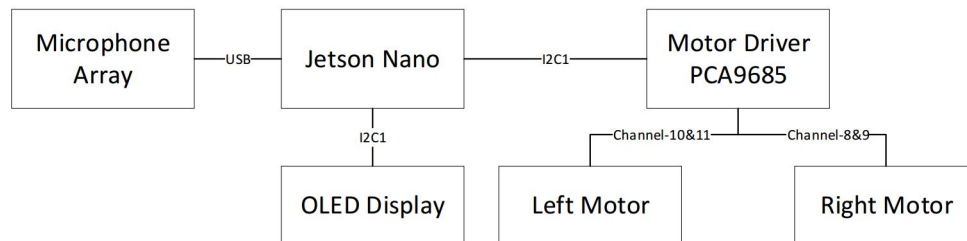


Figure 6.8. Schematic diagram of Jetbot

7. KEYWORD SPOTTING ON MICROCONTROLLER

This chapter describes the implementation of keyword spotting on STM32F769 microcontroller. A brief description of the tools used and the operating environment was also discussed. It also includes important parts of program, in C language, from the real implementation.

7.1 Tools and Development Environment

STMicroelectronics offers an easy-to-use tool, STM32CubeMX, for generating initial coding for a variety of 32-bit microcontrollers and development boards. It also includes X-Cube-AI, a software package for translating trained DNN models into C language equivalents. A brief descriptions are provided about these tools and development environment in the next sections.

7.1.1 STM32CubeMX code generator

STM32CubeMX is a graphical tool for configuring STM32 microcontrollers and microprocessors in a very simple manner. It can be used to generate initialization C code very easily by following few steps.

The first step is to select the microcontroller, microprocessor or development board that satisfies the user requirements. The second step is to configure the GPIOs and clock configurations for the whole system. It also allows to configure peripherals, such as DFSDM or DMA, interactively. This step not only helps to resolve pin conflict, but also provides a clock-tree setting graphical interface, power consumption calculator, and middleware stacks. Popular middleware, such as FreeRTOS, can be enabled through this step. Finally the C code can be generated compatible with multiple popular IDE, such as Keil μ Vision, TrueStudio, IAR Embedded Workbench, and so on. If there is any modification needed in the middle of development, it allows to keep the user coding while generating modified initialization code. Figure 7.1 shows the GUI of STM32CubeMX Software for showing the procedure to configure DFSDM peripheral.

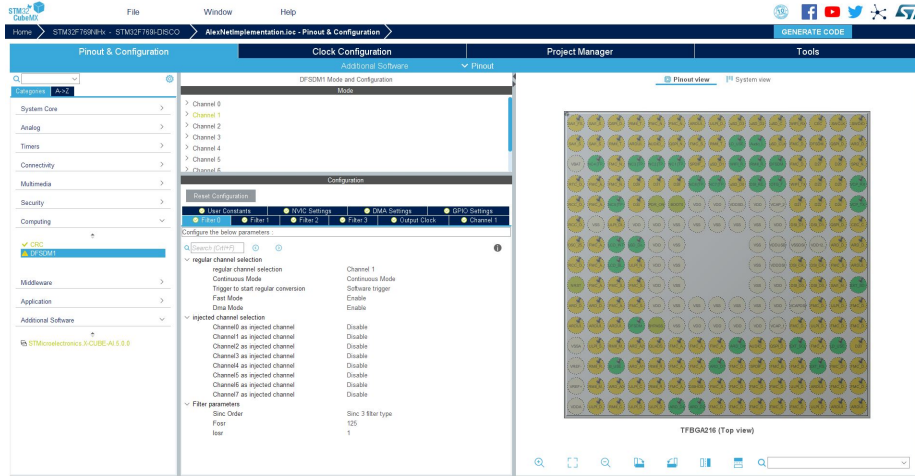


Figure 7.1. Graphical User Interface of STM32CubeMX Software

7.1.2 X-CUBE-AI Expansion Package

STM32Cube.AI ecosystem supports an expansion package X-CUBE-AI that can automatically convert pre-trained DNN models into C code. It can be used to validate the model either on desktop computer or on a microcontroller. Moreover, it can provide the performance measurement without handmade ad hoc C code. It has native support for two most popular deep learning frameworks such as Keras and TensorFlowTM Lite. It also supports other frameworks that gives ONNX standard output format for DNN models such as PyTorchTM, Microsoft[®] Cognitive Toolkit, MATLAB[®], and so on. It also supports 8-bit quantized networks from Keras and TensorFlowTM Lite. To implement larger networks, it allows the developer to retrieve weights from an external Flash memory and activation buffers from an external RAM.

Figure 7.2 shows the implementation of a saved network in X-CUBE-AI expansion package. The DNN was saved in Keras with an extension of (*.h5). It also provides a flexibility to compress the network by a factor of 4 or 8. After analyzing the network, this package also returns network complexity (in MACC), space required in Flash memory and RAM. Finally, it can generate C code in an application template that makes it easier to implement the model in a microcontroller.

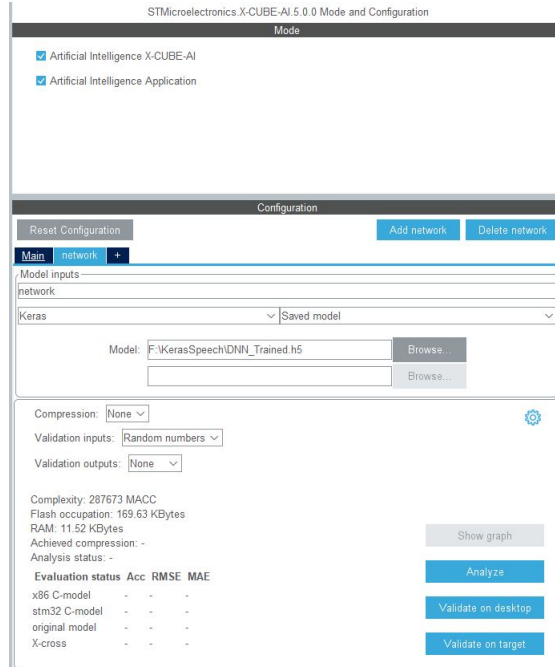


Figure 7.2. X-CUBE-AI expansion package

7.1.3 Keil μ Vision5 IDE

After generating the initialization code in STM32CubeMX, it needs a development environment to program and debug in the hardware component. Keil[®]Microcontroller Development Kit (MDK) supports a wide variety of 32-bit ARM[®] microcontrollers to create, build, and debug embedded software. The μ Vision IDE provides a powerful development environment that combines project management, program debugging, source code editing, build facilities, and run-time environment as shown in Figure 7.3. Its user friendly GUI accelerates the development process.

The debugger allows the developer to test, verify, and optimize the code. With this debugger, one can perform tasks such as simple and complex breakpoints, watch windows, and execution control. It not only aids in bug fixing, but it also boosts productivity.

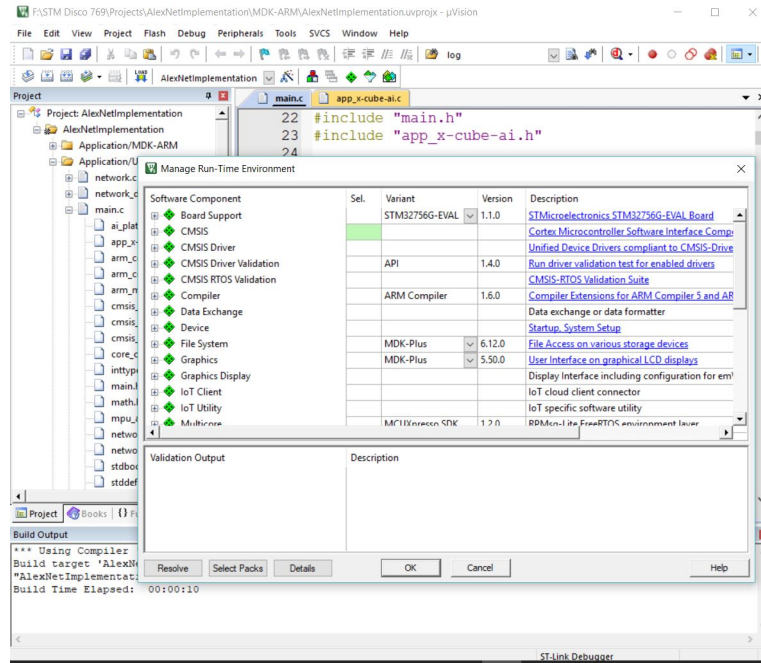


Figure 7.3. Keil uVision5 Integrated Development Environment

7.2 Implementation in C

This section describes the creation of an embedded application for keyword recognition that uses a previously trained DNN model. It can be separated into three major steps as follows:

1. Initialization Code Generation
2. Data Acquisition
3. Feature Extraction
4. AI application

7.2.1 Initialization Code Generation

Before developing application software, the microcontroller needs to be initialized. As discussed in Section 7.1, STM32CubeMX software was used to initialize the microcontroller. A new project was created using STM32F769I-DISCO development board.

First, the system clock was selected as 216MHz from the Clock Configuration tab. Secondly, the DFSDM peripheral was configured which can be found under Computation category under Pinout & Configuration tab. Channel 1, connected to one of the MEMS microphones, was used in PDM/SPI input from channel 1 and internal clock mode. In the output clock configuration parameters, the Divider was set to 54, which allows the DFSDM filter to operate from a 2MHz clock source (the input clock to DFSDM peripheral was 128MHz). After that, the Filter-0 was configured as shown in Figure 7.4 to enable the DMA and FAST continuous conversion mode. Equation 6.5 was used to calculate the filter parameter $F_{OSR}=125$ and $I_{OSR}=1$, which resulted in an output sample rate of 16kHz. Finally, in the DMA Settings, DFSDM_FLT0 request was added to DMA2 stream in Circular mode. This DMA configuration allows to transfer data from peripheral to memory. The interrupt priority was set to low for the stream, which introduces callbacks after half and full conversion.

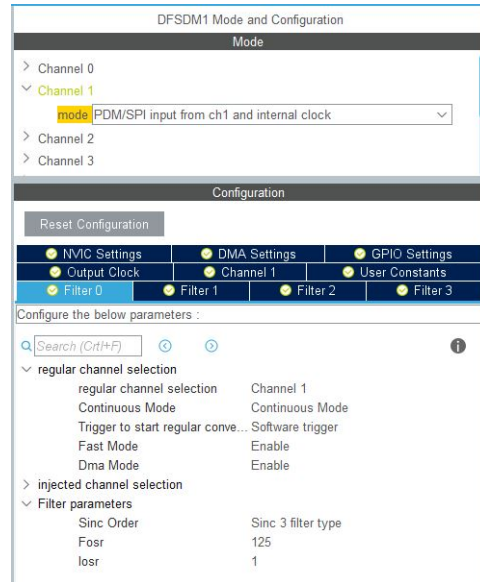
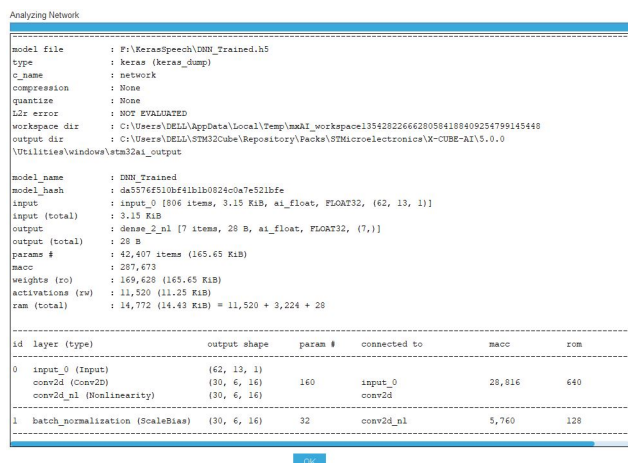


Figure 7.4. DFSDM Filter-0 configuration for Channel 1

The trained DNN model was converted using X-CUBE-AI package, which can be added to the environment from the Additional Software. In this project, X-CUBE-AI version 5.0.0 was used. To make the development easier, an Application Template was also included from this package. In the Mode and Configuration window, a network was added for conversion.

The previously trained DNN model in Keras was added from the local storage as the model input parameters. This network can be analyzed by clicking the Analyze button. It shows the computational complexity and amount of storage required in the flash as well as in the RAM as shown in Figure 7.5. It also gives more detail information of each network layer, such as- input shape, required RAM, MACC, and so on. All of these parameters were verified with the original trained network, as discussed previously.



The screenshot shows a window titled "Analyzing Network" with a text area containing the following information:

```

model file      : F:\KerasSpeech\DNN_Trained.h5
type           : Keras (keras_dump)
c_name        : network
compression    : None
quantize       : None
Ldr error      : NOT EVALUATED
workspace_dir  : C:\Users\DELL\AppData\Local\Temp\msAI_workspace13542822666220584189409254799145448
output_dir     : C:\Users\DELL\STM32Cube\Repository\ Packs\STMMicroelectronics\K-CUBE-AI\5.0.0
               \Utilities\windows\stm32ai_output

model_name     : DNN_Trained
model_hash     : da574f5f10b4f41b0824c0a7e521b4fe
input          : input_0 [806 items, 3.15 KiB, ai_float, FLOAT32, (62, 13, 1)]
input (total)  : 3.15 KiB
output         : dense_2_nl [7 items, 28 B, ai_float, FLOAT32, (7,)]
output (total) : 28 B
param #        : 42,407 items (165.65 KiB)
macco          : 287,473
weights (ro)   : 169,628 (165.65 KiB)
activations (rw) : 11,520 (11.25 KiB)
ram (total)    : 14,772 (14.43 KiB) = 11,520 + 3,224 + 28
  
```

Below the text area is a table summarizing the layers:

| id | layer (type) | output shape | param # | connected to | macco | rom |
|----|----------------------------------|--------------|---------|--------------|--------|-----|
| 0 | input_0 (Input) | (62, 13, 1) | | | | |
| | conv2d (Conv2D) | (30, 6, 16) | 160 | input_0 | 28,816 | 640 |
| | conv2d_pl (Nonlinearity) | (30, 6, 16) | | conv2d | | |
| 1 | batch_normalization (Scale/Bias) | (30, 6, 16) | 32 | conv2d_pl | 5,760 | 128 |

Figure 7.5. Analyzing imported trained neural network

One last important configuration is the USART1 communication interface, which is connected with the ST-Link programmer and very helpful to send and read message from a computer. It can be found under connectivity category. It was set to operate in Asynchronous mode. The baud rate was selected as 115200 bits/second, word length as 8 bits, parity as None, and stop bit as 1.

Finally, in the Project Manager tab, MDK-ARM v5.27 was selected as IDE, and both minimum size for heap and stack was selected as 0x20000. The code was generated by clicking on the Generate button and opened in µVision IDE for the next phase of development. In the IDE, another small initialization code, as shown in Figure 7.6, was included in the *main.c* file to use *printf()* function for transmitting characters over USART1. Thus, it allows the connected computer to receive a message.

```

#ifdef __GNUC__
#define PUTCHAR_PROTOTYPE int __io_putchar(int ch)
#else
#define PUTCHAR_PROTOTYPE int fputc(int ch, FILE *f)
#endif
PUTCHAR_PROTOTYPE{
    HAL_UART_Transmit(&huart1, (uint8_t *)&ch, 1, 0xFFFF);
    return ch;
}

```

Figure 7.6. C code to enable *printf* to transmit over USART1

7.2.2 Data Acquisition

A 32-bit integer type buffer pointer of size 768 was configured as DMA buffer to store 24-bit integer type PCM audio samples from DFSDM peripheral. Two callback functions- *HAL_DFSDM_FilterRegConvHalfCpltCallback()* and *HAL_DFSDM_FilterRegConvCpltCallback()* were included to update two global flags *conv_half* and *conv_full* respectively that indicates the DMA transfer buffer state.

```

void collect_samples(q31_t *all_samples){
    uint16_t index = 0;
    while(index < SAMPLE1SEC){ // Loop through asking samples
        if (conv_half){ // If conv_half flag is set
            // Loop through first half of DMA buffer
            for (size_t i = 0; i < AUDIO_REC >> 1; i++, index++)
                // Copy first half, discard 8 MSBs of DMA buffer
                all_samples[index] = adc_values[i] >> 8;

            conv_half = false; // Reset half conversion flag
        }

        if (conv_full){ // If conv_full flag is set
            // Loop through second half of DMA buffer
            for (size_t i = AUDIO_REC >> 1; i < AUDIO_REC; i++, index++)
                // Copy second half, discard 8 MSBs of DMA buffer
                all_samples[index] = adc_values[i] >> 8;

            conv_full = false; // Reset full conversion flag
        }
    } //End while
} //End collect_samples

```

Figure 7.7. Code to collect sample data for 1 second

A function, *collect_samples()*, was used to gather and store the speech samples in a pointer variable called *all_samples* as shown in Figure 7.7. When half of the DMA buffer, *adc_values*, is populated with DFSDM data, it sets the global flag *conv_half* in the callback function. Similarly for full completion of DMA buffer, *conv_full* is set. During the half conversion, first half of the DMA buffer is copied into *all_samples* pointer. For the full conversion, second half of the buffer is copied. The sample data was right shifted 8-bit for both cases, because the DFSDM output was 24-bit long. At the end of this function, *all_samples* was populated with 16128 integer type audio samples.

7.2.3 Feature Extraction

The feature extraction is associated with several steps. As this process involves some advanced computation, CMSIS-DSP library was utilized for faster operation. It comes with the firmware of STM32F769 microcontroller and version 1.15.0 was used in this project. The details information is available in [26] about the process of including this library in Keil μ Vision IDE. It provides very efficient function for complex mathematics, matrix operation, filtering, and so on. The steps for feature extraction are as follows:

Step 1: All available samples were framed with an step of 256. The number of frames can be found by Equation 3.3.

Here, T is the total number of frames, L is the length of available samples, l is the length of the window and s is the stride. Taking $L = 16128$, $l = 512$, and $s = 256$, the total number of frames yielded $T = 62$. Calculations are performed with each of these frames as stated in next steps.

Step 2: In this step, a 512-point Real Fast Fourier Transform was performed on each frame and calculated power spectrum by taking square of each FFT sample. CMSIS-DSP provides *arm_rfft_q31()* function that was used to calculate RFFT of individual frame. After the transform, absolute value was taken for 257 samples. The power was also calculated as shown in Figure 7.8. As these calculations are memory intensive, memory was allocated dynamically in the program. For the ease of reading, dynamic memory allocation was avoided in the programming.

```

// Calculate Real Fast Fourier Transform
arm_rfft_init_q31(&RFFT_Instance, 512, 0, 1); // RFFT instance for 512-
point
arm_rfft_q31(&RFFT_Instance, frame, ffts); // Calculate RFFT of the frame
arm_abs_q31(ffts, ffts, 257); // Take the absolute values

// Calculate Power Spectrum from the RFFT output
for (size_t i = 0; i < 257; i++)
    power_fft[i] = (float32_t) pow(ffts[i], 2);

```

Figure 7.8. Real Fast Fourier Transform with CMSIS-DSP library

Step 3: A mel-spaced filter-bank was applied to calculate energy in each filter. A filter-bank of size (257, 16) was computed in Python Mel-scale equation and then parameters were saved into a variable that was hard coded into flash memory. The energy is a dot product of previously obtained power spectrum signal of size (1, 257) and the hard coded filter-bank. This dot product was calculated usgin *arm_mat_mult_f32()* function that gave an energy spectrum of size (1, 16) as shown in Figure 7.9.

```

// Initialize Matrices: Power, Filter-bank & Filtered
arm_mat_init_f32(&POWER, 1, 257, power_fft);
arm_mat_init_f32(&FILTER, 257, 16, filt_bank);
arm_mat_init_f32(&ENERGY, 1, 16, energy);

// Calculate Energy: Dot product of Power spectrum and Filter (1X257) .
(257X16) = (1X16)
arm_mat_mult_f32(&POWER, &FILTER, &ENERGY);

```

Figure 7.9. Energy spectrum calculation

Step 4: This step is relatively straight forward. A natural logarithm was calculated for each element using *log()* function of C.

Step 5: A type-II Discrete Cosine Transform (DCT) was calculated from the "log filter-bank energies." CMSIS-DSP does not provide any function to perform type-II DCT. For this calculation, the data was re-ordered first, extended double in length and mirrored the second half, while keeping the first half as original. Then a 32-point FFT was performed in order to calculate the DCT as shown in Figure 7.10. Here, the *dct_energy* variable stored the calculated type-II DCT values.

```

// Re-organize the energy data
for (count = 0; count < 13; count++){
    energy[13 + count] = energy[13 - count - 1]; // second half (mirrored)

// Calculate FFT (32 point) for the re-organized data
arm_rfft_32_fast_init_f32(&RFFT_DCT);
arm_rfft_fast_f32(&RFFT_DCT, energy, dct_energy, 0);

// Keep only real values (13 features) and Apply linear lifter for high
frequency components
for (count = 0, i = 0; count < 2 * 13; count += 2, i++){
    features[i] = dct_energy[count]; // Take real values only
    features[i] *= 1 + (i << 2);    // Dynamic multiplier
}

```

Figure 7.10. Calculate type-II DCT and apply lifter

Step 6: Only the real values were kept from *dct_energy* variable. However, higher frequency elements of this DCT output is smaller, but important. So, a dynamic multiplier was used to lift up the higher frequency elements as shown in Figure 7.10. Thus it gave the feature for single frame.

Step 7: Finally, these features were copied in another variable *feat_bank* that represents the feature bank. It gives a feature matrix of size (62, 13), which is ready to be implemented on the trained DNN model.

7.2.4 AI Implementation

The converted model parameter and weights were stored in *network.c* and *network_data.c* files respectively. To implement the DNN in microcontroller, four header files were included as shown in Figure 7.11.

```

#include "ai_datatypes_defines.h"
#include "ai_platform.h"
#include "network.h"
#include "network_data.h"

```

Figure 7.11. Header files for AI implementation

After including the header files, it involves following steps to perform prediction of the Neural Network:

1. Allocate memory to hold intermediate values for neural network.
2. Create global pointer that holds the model parameters.
3. Create wrapper that holds the data and the model information.
4. Set the working memory and get weights or biases from model.
5. Set pointers wrapper to the data buffers.
6. Create instance of neural network.
7. Initialize neural network model.
8. Fill up input buffer.
9. Run the model and make prediction.
10. Process the output data.

At the end of the implementation of the Neural Network, the output was converted into floating point data type and stored in *predictions* variable as shown in Figure 7.12. This *predictions* variable was a floating point array of seven elements that indicated keywords Go, Stop, Left, Right, Up, Down, and Unknown with an index from 0 to 6 respectively. If any value of the *predictions* array crossed a threshold limit (0.8), corresponding keyword was sent to the USART1 terminal. The USART1 was connected to a computer and thus the speech command was detected. Figure 7.13 shows the flowchart to implement DNN on bare-metal microcontroller platform for KWS.

```

// Memory for activation , input and output buffer
static ai_u8 activations[AI_NETWORK_DATA_ACTIVATIONS_SIZE];
static ai_i8 in_data[AI_NETWORK_IN_1_SIZE_BYTES];
static ai_i8 out_data[AI_NETWORK_OUT_1_SIZE_BYTES];

// Global pointer to hold the model
static ai_handle network = AI_HANDLE_NULL;

// Input and output wrapper
static ai_buffer ai_input[AI_NETWORK_IN_NUM] = AI_NETWORK_IN ;
static ai_buffer ai_output[AI_NETWORK_OUT_NUM] = AI_NETWORK_OUT ;

// Working memory for weights/biases
const ai_network_params params = {
    AI_NETWORK_DATA_WEIGHTS(ai_network_data_weights_get()),
    AI_NETWORK_DATA_ACTIVATIONS(activations)
};

// Wrapper to the data buffers
ai_input[0].n_batches = 1;
ai_input[0].data = AI_HANDLE_PTR(in_data);
ai_output[0].n_batches = 1;
ai_output[0].data = AI_HANDLE_PTR(out_data);

// Instance of the NN
ai_network_create(&network, AI_NETWORK_DATA_CONFIG);

// Initialize the NN – Ready to be used
ai_network_init(network, &params);

// Populate the input buffer
for (ai_size i=0; i < AI_NETWORK_IN_1_SIZE; i++ )
    ((ai_float *)in_data)[i] = feat_bank[i];

/* Perform the inference */
aiRun(in_data, out_data);

// Post-Process – process the output buffer
for (size_t i = 0; i < 7; i++)
    predictions[i] = ((float32_t *)out_data)[i];

```

Figure 7.12. Implementing Neural Network using X-CUBE-AI

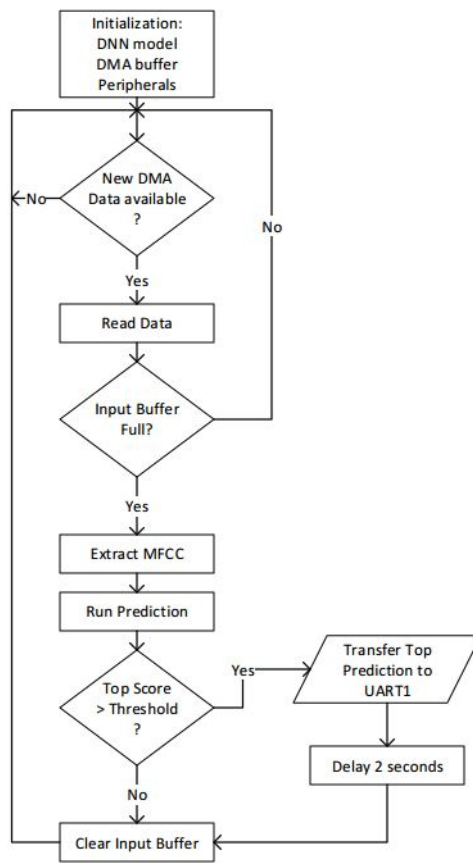


Figure 7.13. Bare-metal implementation for KWS

8. KEYWORD SPOTTING ON JETBOT

This section discusses about the implementation of trained DNN model in a robot vehicle, Jetbot, running in embedded Linux platform. It needs to configure the software environment first and then run the model for inference. The microphone array and motor controller also need to be interfaced before running the prediction. Following sections provides brief description which leads to a real time voice command driven robot vehicle.

8.1 Environment Configuration

The configuration begins with the most important python library for this project, TensorFlow. The DNN model was trained in TensorFlow version 2.0.0. However, Jetson Nano does not support generic version, instead it has its own GPU version of TensorFlow. First, the JetPack version needs to be checked, V42 in this case, and installed from [27].

8.1.1 Signal Acquisition from Microphone Array

The microphone array, as shown in Figure 6.6 provides six channel of audio data. The first channel contains processed data for speech recognition applications. Next four channel provides raw data from four microphones. The sixth channel is the merged data from of all microphones. As this trained DNN needs only one channel of data, only channel-0 was used and discarded rest of channels. An object oriented programming approach was adopted here. Figure 8.1 shows the class diagram to read the microphone array.

The main instance *ReadMic* is related to *MicConfig* and *Stream* classes. It inherits some important attributes to read available information from the microphone array. The *raw_data* attribute collects data from all six channels and the *single_channel* attribute returns only channel-0 data. It also have an attribute *stream* which is a child of *Stream* class. It initiates a connection between Jetson Nano and the microphone array. As the *MicConfig* class is associated with the *ReadMic* class, they shares some attributes such as rate, channels, width, etc. These attributes are needed to open up the stream. The *ReadMic* class also has two methods to handle the signal acquisition process. The *callback()* method

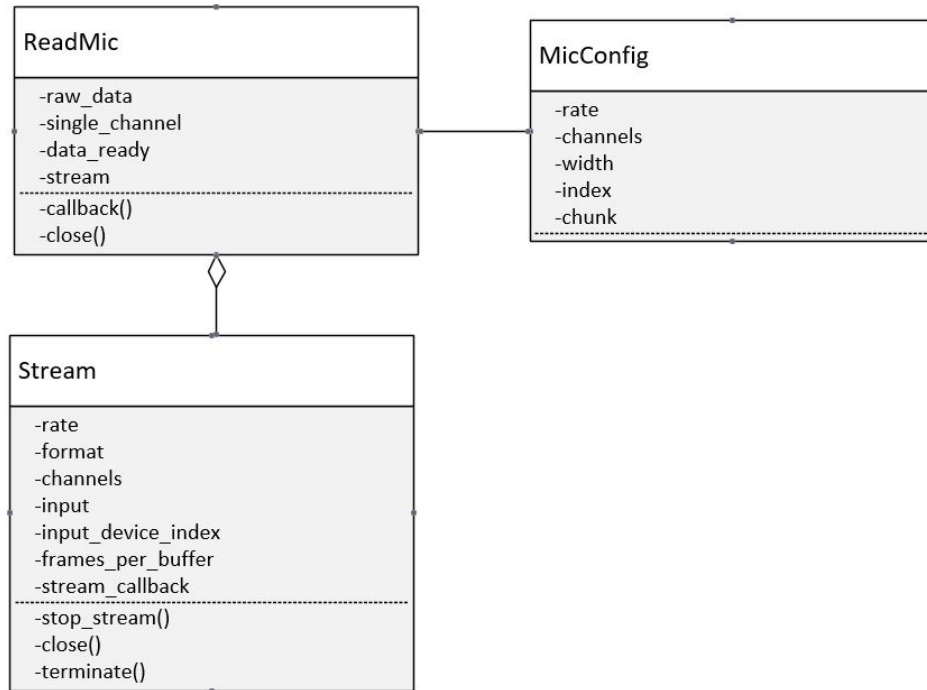


Figure 8.1. Class diagram to read data from microphone array

is called by *stream* attribute after it finishes signal acquisition from all six channels. In this callback function, signal processing is performed by using NumPy[28] and channel-0 data is stored in the *single_channel* attribute. It also sets a flag, attribute *data_ready*, after a successful operation.

The *Stream* class has three methods. As the name suggests, *stop_stream()* method stops the stream temporarily and *close()* method closes the stream. However, method *terminate()* disconnects the connection with the microphone array so that it can be used by other program. If the *close()* method from *ReadMic* class is called, it invokes all of these three methods from *Stream* class and thus, ends the communication with the microphone array. Figure 8.2 shows part of the microphone reading script written in Python 3.

8.1.2 Motor Driver

There are three DC motors in the Jetbot robot car. These motors are driven by a motor driver IC, PCA9685, as discussed in section 6.2.3. It is a 16-channel PWM generator and

```

class ReadMic:
def __init__(self):
    cfg = MicConfig() # Microphone configuration
    self.raw_data = np.array([]) # From 6 channel
    self.single_channel = np.array([]) # Channel-0
    self.data_ready = False # Flag for finished conversion
    self.p = pyaudio.PyAudio()
    self.stream = self.p.open( # Stream Class
        rate=cfg.rate,
        format=self.p.get_format_from_width(cfg.width),
        channels=cfg.channels,
        input=True,
        input_device_index=cfg.index,
        frames_per_buffer=cfg.chunk,
        stream_callback=self.callback)

# Callback method to extract single channel data
def callback(self, in_data, frame_count, time_info, status):
    all_data = np.frombuffer(in_data, dtype='int16')
    self.single_channel = all_data[:6] # Read Channel-0
    self.voice_std = np.std(self.single_channel)
    self.data_ready = True # new data available
    return (all_data, pyaudio.paContinue)

```

Figure 8.2. Python script to write microphone array

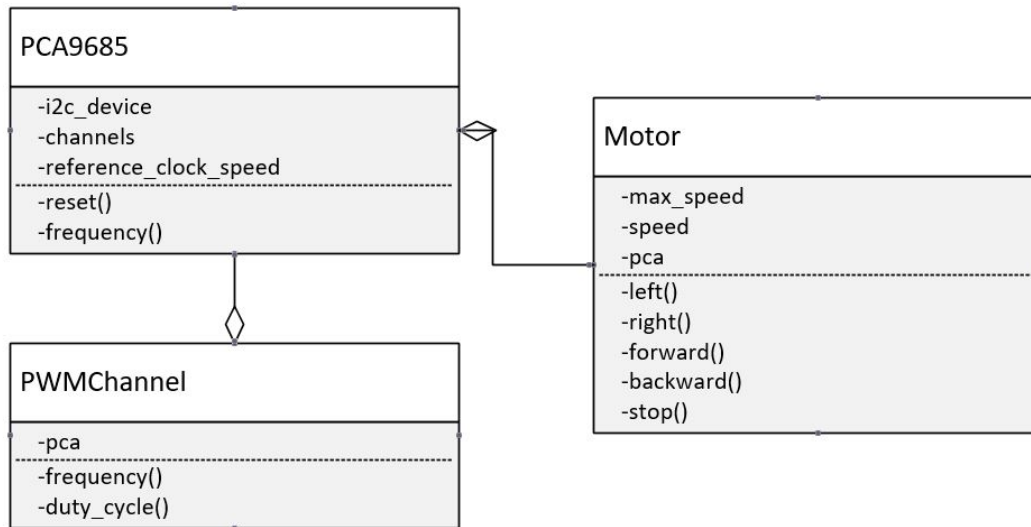
the connection summary is provided in Table 8.1. Each DC motor needs two PWM channel to control rotation in both clockwise and counter-clockwise direction. Motor-1 is connected to pin-15 of the driver IC and thus powered from the PWM channel-8 as instructed by the data-sheet [29]. Similarly another terminal of Motor-1, IN1B, is connected to pin 16 of the driver IC and powered by channel-9. It enables the car to rotate the right caterpillar tread. The car also has another DC motor, Motor-2, powered from channel-10 and channel-11 of the driver IC that can rotate the left caterpillar tread. The car can be moved along a surface by controlling the rotation of these two tread. There is another motor, Motor-3, to lift the camera module in upward or downward direction. It is very helpful to implement this car in image recognition applications.

Adafruit Industries provides a driver library to interface PCA9685 with Jetson Nano. Object oriented programming approach is also adopted for the car movement as shown in 8.3. In this diagram, the *Motor* instance is responsible for the car movement. The attribute *speed* can set the car speed in any direction. It has five methods to control the movements-

Table 8.1. PCA9685 Motor Driver Interface

| Function | Name | Channel | Pin | Remarks |
|----------|------|---------|-----|---------------|
| Motor 1 | IN1A | 8 | 15 | Right Motor |
| - | IN1B | 9 | 16 | - |
| Motor 2 | IN2A | 10 | 17 | Left Motor |
| - | IN2B | 11 | 18 | - |
| Motor 3 | IN3A | 12 | 19 | Up-Down Motor |
| - | IN3B | 13 | 20 | - |

left(), *right()*, *forward()*, *backward()*, and *stop()*. Each of these methods set the duty cycle of the responsible channel. For example, to move the car in forward direction, channel-8 and channel-11 was set to a duty cycle according to the asking velocity. The *PCA9685* class handles connection through through the I2C bus. It is aggregated with the class *PWMChannel* and can control the duty-cycle of any channel by calling the *duty_cycle()* method. It can also set the frequency of the PWM waves up to 1526Hz. When the car needs to be stopped, *stop()* method is called that sets the duty cycle of all channels to zero. It is also called at the beginning of any movement to reset the previous state.

**Figure 8.3.** Class diagram for motor driver

8.2 Implementation on Jetbot

The trained model was implemented in Jetbot using Keras in TensorFlow library. The instances developed in the previous sections for microphone array and motor drive were also used here. Figure 8.4 shows the flowchart that implement the trained DNN model to drive the robot car.

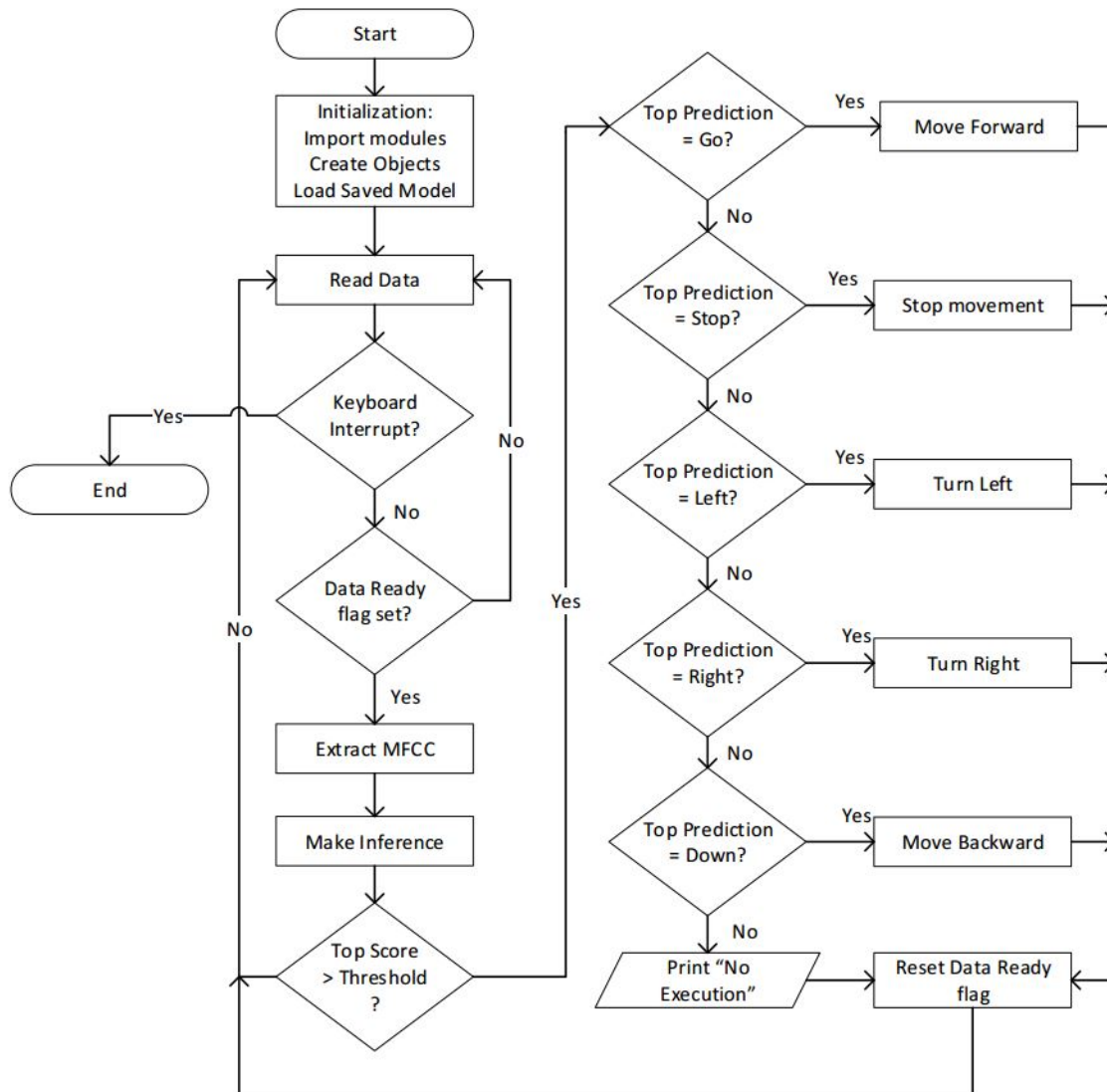


Figure 8.4. Jetbot keyword recognition and drive flowchart

The procedure begins with importing the required modules and instances. From the instance *MicRead*, a class was created, *speech*. When the samples were ready, a flag attribute

was set and *single_channel* attribute stored the audio samples from channel-0. Another class, *car*, was created from the *Motor* instance. It handles the movement of the vehicle by calling the methods discussed in the previous section. An object, *model*, was also created to load the trained neural network.

After reading the samples from the *speech* class, MFCC was extracted. As the model accepts an input with a shape of (1, 62, 13, 1), the feature bank was reshaped and feed into the network for prediction. The network made inference with the provided inputs and provided an array of prediction in a range of 0 to 1. The array elements were sorted and if the highest score crossed a threshold limit, the prediction was accepted to move the car. Otherwise, it went back to the infinite loop that is only ended with an external keyboard interrupt. For a higher score, the command was checked for keywords- *go*, *stop*, *left*, *right*, and *down*. If the keyword is *unknown*, a message was printed, "No execution." At the end of executing the command, the *data_ready* flag was reset and returned to the infinite loop again.

9. EXPERIMENTAL RESULTS

Following the extraction of features from audio files, the DNN was trained. During the training process, training accuracy and loss were recorded. The results of the training process and their implementation on edge devices are discussed in this chapter.

9.1 MFCC Outputs

The speech features were extracted using MFCC. The data rate of the input audio files was 16000 samples per second and all of the data were not of the same duration. As stated before, audio inputs were resized by a length of 16000 during training. The samples were divided into multiple frames of length 32ms (512 samples) with a stride of 16ms (256 samples). It provided a feature-bank of 62 feature sets, each containing 13 features. Figure 9.1 shows the plot of one feature from the feature-bank of a speech command “go” using both Python 3 and C languages. It shows that the extracted features using both languages are almost identical. Although there is a small difference between these outputs, it is acceptable to implement them in this KWS application.

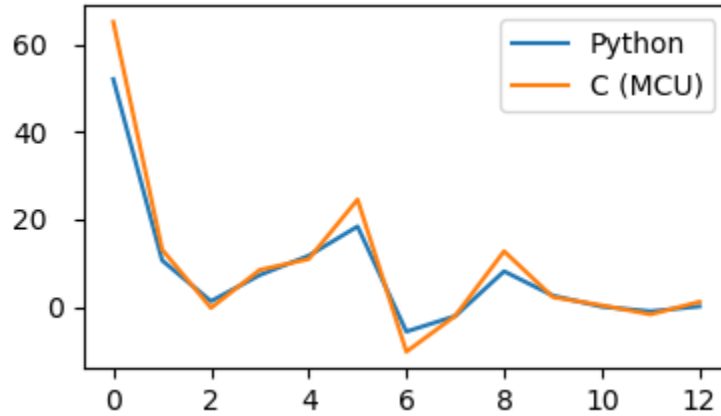


Figure 9.1. Single feature from feature-bank of keyword “go”

9.2 Training Outputs

One of the very important parameters in training the DNN model is the loss function. In an ideal case, the training loss is expected to reduce after finishing each epoch. Moreover, the validation loss is expected to be lower than the validation loss. When the validation loss becomes higher than the training loss, the model is called over-fitted. However, slightly over-fitting does not affect the overall performance that much. Figure 9.2 shows the training and validation loss of the DNN model as indicated by the blue and orange lines respectively. The training loss started with 1.72 and the validation loss started with 0.81. After finishing the epochs, the loss lines intersected with each other with a value of about 0.55. At the end of 13 epochs, the model had a validation loss of 0.3 and a training loss of 0.17. It indicates the model is slightly over-fitted. However, it did not affect the performance.

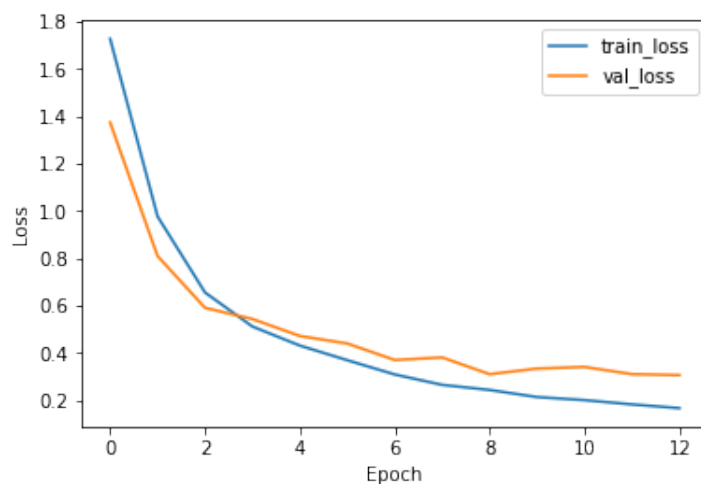


Figure 9.2. Training and validation loss

Figure 9.3 shows the training and validation accuracy over the epochs. The training accuracy was very low at the beginning of training, about 30 percent. It increased with the epochs and crossed the validation accuracy after the fourth epoch at about 80 percent. The final validation accuracy, 90 percent, was slightly lower than the training accuracy, 93 percent. It was caused due to over-fitting as discussed previously.

Finally, a confusion matrix was generated as shown in Figure 9.4 by performing prediction on the test data-set. The horizontal axis indicates the predicted outputs and the vertical

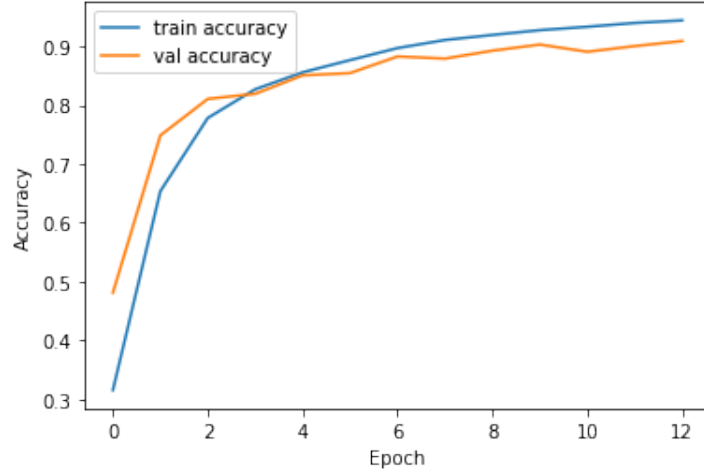


Figure 9.3. Training and validation accuracy

axis indicates the actual outputs. It also shows a heat-map for all labels, where the darker means less accurate. The keyword *stop* and *left* were predicted with higher accuracy and the *unknown* keyword was less accurate.

$$Accuracy = \frac{FP + FN}{TP + TN} \quad (9.1)$$

The test accuracy was calculated by Equation 9.1 as the ratio of false prediction (False Positive, FP and False Negative, FN) to true prediction (True Positive, TP and True Negative, TN). By calculating this equation, the test accuracy was obtained 91%.

9.3 Prediction Outputs and Execution Time

The trained DNN models were applied on the STM32769 microcontroller and Jetbot robot car. The outputs were monitored in computer and the execution time was also recorded.

9.3.1 Results from Bare-Metal Microcontroller implementation

After converting the DNN model for C equivalent, it occupied only 11.52 Kbyte (2.16%) RAM and 169.63 Kbyte (8.48%) Flash of the test device. It also requires a memory com-

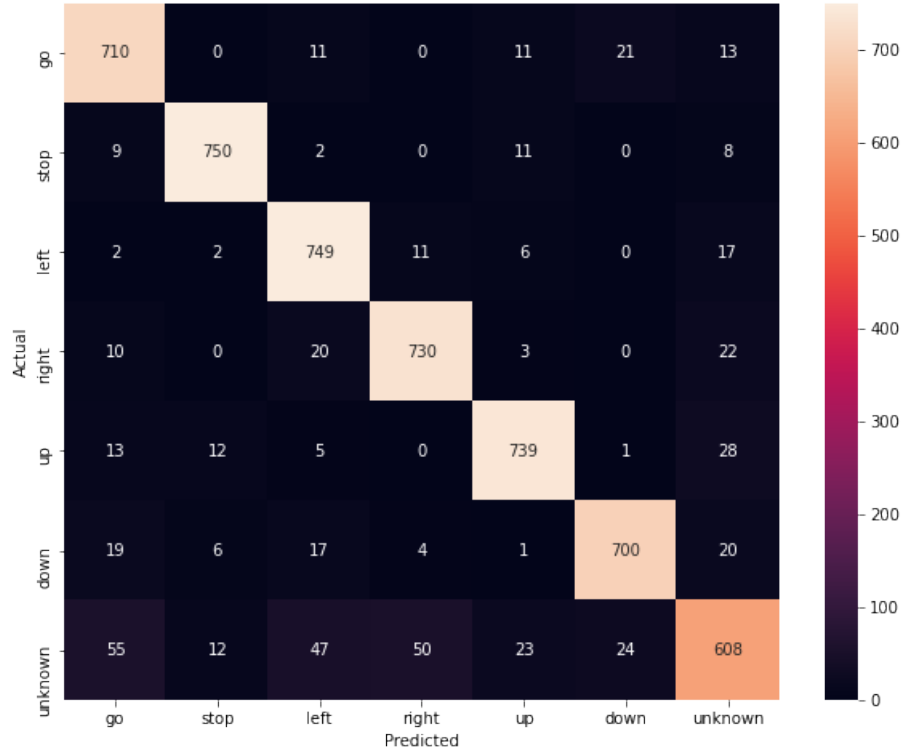


Figure 9.4. Confusion Matrix on test data-set

plexity of 287,673 MACC. The microcontroller was programmed to provide a serial output for the predicted keywords along with the prediction score. The baud rate was set to 115200 bits per second. The predictions were received by connecting a computer with ST-LINK debugger USB port of STM32F769 microcontroller board. A serial data receiver software, Arduino Serial monitor was used to capture those strings as shown in Figure 9.5. This test was performed on real user and all of the trained keywords were predicted correctly.

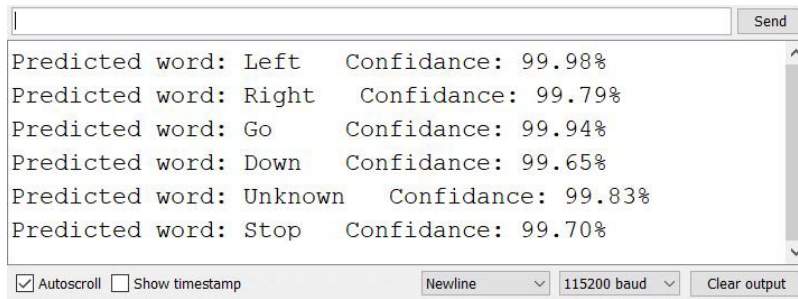


Figure 9.5. Serial output on prediction

The microcontroller took about 7ms to finish each prediction as shown in Figure 9.6. In order to measure the prediction time, the User1_LED pin of the board was used. At the beginning of the execution, this pin was turned high, which in changed to low at the end of the execution. An oscilloscope was connected with this pin and checked for the signal in single pulse mode and triggered on rising edge. It not only allowed to measure the signal, but also enabled to have an idea about the execution time by observing the blinking LED for continuous operation. It also helped to find bugs in the program, such as memory leakage. A summary of the bare-metal implementation is provided in Table 9.1.

Table 9.1. Summary of bare-metal implementation

| Name | Value |
|----------------|-----------------------|
| Flash Usage | 169.63 kBytes (8.48%) |
| RAM Usage | 11.52 kBytes (2.16%) |
| Complexity | 287673 MACC |
| Execution Time | 6.941 ms |

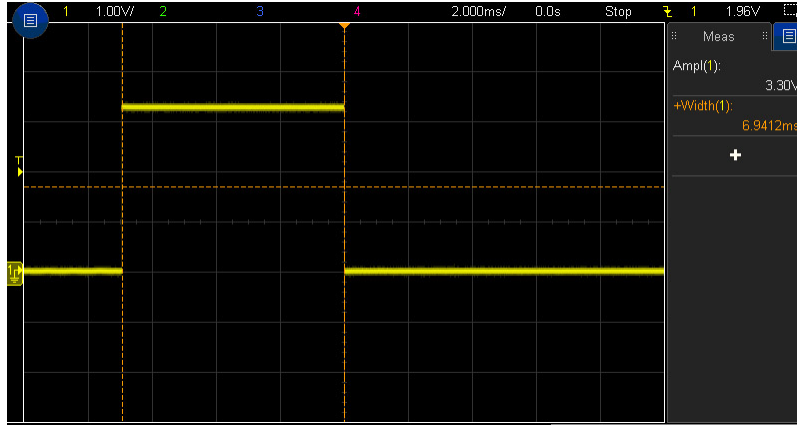


Figure 9.6. Prediction time in microcontroller

9.3.2 Results from Jetbot implementation

The trained DNN was implemented on Jetbot and the prediction was used to drive the car. This device was accessed remotely from a linux terminal. After running the implementation script, top prediction and score was shown in the terminal as shown in Figure 9.7. In

this figure, the first command was *unknown*, having a confidence of 75%. As this command is not involved with any movement, the output printed “Not executed”. However, when it found expected commands, such as- *stop* or *go*, associated function was called and the motor was moved along that direction. At the end of the prediction, the infinite loop was closed using keyboard interruption. This exit triggered several process to terminate the connection with microphone array and motor driver.

```
stop
Command: unknown Score: 0.75035185
Not executed
Command: stop Score: 0.9014772
stop
Command: stop Score: 0.73475146
stop
Command: go Score: 0.9537486
move forward
Command: left Score: 0.99927765
turn left
Command: right Score: 0.9988153
turn right
Command: down Score: 0.9999975
move backward
Command: stop Score: 0.96988577
stop
^CExit prediction :)
Exiting...
Cleaning up pins
jetbot@jetbot-desktop:~/Voice_control$
```

Figure 9.7. Running robot from terminal

The execution of the DNN model takes about 15ms as shown in Figure 9.8. This execution time was measured using internal timer of the Jetson Nano. This time varied from about 13ms to 20ms over the execution period.

```
Execution time: 0.015846967697143555
Execution time: 0.014245271682739258
Execution time: 0.01661372184753418
Execution time: 0.016358137130737305
Execution time: 0.017489194869995117
Execution time: 0.01668858528137207
Execution time: 0.014424562454223633
```

Figure 9.8. Prediction time in Jetbot

10. CONCLUSION

A Deep Neural Network-based model is proposed, trained, and implemented in two different types of edge devices in this research. The first type of edge device includes a 32-bit ARM Cortex-M7 bare-metal microcontroller. This device is designed to send the DNN model's output, keyword, to a UART terminal. The second device is Jetbot, a self-driving robot car that responds to voice commands. In this device, the same DNN model is used, and the car is moved based on the command it recognizes.

The MFCC method is used to extract speech features, which results in 2D spatial data. These features are extracted using only 16 filters. For faster calculation speed, the model is implemented in C language on the bare-metal device, using a DSP library. It is implemented in Jetbot in Python 3 using the popular NumPy library. The output of MFCC is nearly identical in these two different implementations.

The model has been trained using the Google Speech Commands dataset version 2. It achieves a final accuracy of around 91%. The accuracy would improve even more as the filter dimension was increased while extracting the features. It would, however, raise the cost of computation. Given the network's reduced computational complexity, this efficiency is adequate for KWS applications.

The trained model must be converted into C equivalent code before it can be implemented in a microcontroller. For this conversion, XCUBE-AI proves to be a very useful tool. It also includes a template, which makes implementation much easier. A driver IC is interfaced with the Jetson Nano in order to use the KWS to navigate the robot car. The speed of the corresponding motor is controlled after the keyword is detected. In the microcontroller, the average DNN execution time is around 7ms. The Jetbot, on the other hand, has an execution time of about 15ms. Even though Jetbot has much more processing power, the longer processing time could be due to software overhead.

A noise from the caterpillar treads appears while implementing the model in Jetbot. It interferes with the speech frequency bands. As a result, the model performs better when the car's treads are stationary than when it is moving. A more noise-resistant model may be a

good solution for improving performance. There is also room to improve the DNN model's accuracy while keeping the computation cost and memory footprint to a minimum.

REFERENCES

- [1] A. Alnoman, S. K. Sharma, W. Ejaz, and A. Anpalagan, “Emerging edge computing technologies for distributed iot systems,” *IEEE Network*, vol. 33, no. 6, pp. 140–147, 2019.
- [2] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, “Edge computing: Vision and challenges,” *IEEE internet of things journal*, vol. 3, no. 5, pp. 637–646, 2016.
- [3] B. A. Mudassar, J. H. Ko, and S. Mukhopadhyay, “Edge-cloud collaborative processing for intelligent internet of things: A case study on smart surveillance,” in *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, IEEE, 2018, pp. 1–6.
- [4] M. H. Samavatian, A. Bacha, L. Zhou, and R. Teodorescu, “Rnnfast: An accelerator for recurrent neural networks using domain-wall memory,” *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, vol. 16, no. 4, pp. 1–27, 2020.
- [5] Y. Zhang, N. Suda, L. Lai, and V. Chandra, *Hello edge: Keyword spotting on micro-controllers*, 2018. arXiv: [1711.07128](https://arxiv.org/abs/1711.07128) [[cs.SD](#)].
- [6] L. Lai and N. Suda, “Enabling deep learning at the lot edge,” in *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2018, pp. 1–6. DOI: [10.1145/3240765.3243473](https://doi.org/10.1145/3240765.3243473).
- [7] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, *Mobilenets: Efficient convolutional neural networks for mobile vision applications*, 2017. arXiv: [1704.04861](https://arxiv.org/abs/1704.04861) [[cs.CV](#)].
- [8] S. Yang, Z. Gong, K. Ye, Y. Wei, Z. Huang, and Z. Huang, “Edgernn: A compact speech recognition network with spatio-temporal features for edge computing,” *IEEE Access*, vol. 8, pp. 81 468–81 478, 2020.
- [9] S. Mittermaier, L. Kürzinger, B. Waschneck, and G. Rigoll, “Small-footprint keyword spotting on raw audio data with sinc-convolutions,” in *ICASSP 2020-2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, IEEE, 2020, pp. 7454–7458.
- [10] V. K. Abdrakhmanov, R. B. Salikhov, and K. V. Vazhdacv, “Development of a sound recognition system using stm32 microcontrollers for monitoring the state of biological objects,” in *2018 XIV International Scientific-Technical Conference on Actual Problems of Electronics Instrument Engineering (APEIE)*, 2018, pp. 170–173. DOI: [10.1109/APEIE.2018.8545278](https://doi.org/10.1109/APEIE.2018.8545278).

- [11] Google, *Speech commands dataset version 2*, 2018. [Online]. Available: http://download.tensorflow.org/data/speech_commands_v0.02.tar.gz.
- [12] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, “Going deeper with convolutions,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 1–9.
- [13] Y. Qian, M. Bi, T. Tan, and K. Yu, “Very deep convolutional neural networks for noise robust speech recognition,” *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, vol. 24, no. 12, pp. 2263–2276, 2016.
- [14] A. Mahajan and S. Chaudhary, “Categorical image classification based on representational deep network (resnet),” in *2019 3rd International conference on Electronics, Communication and Aerospace Technology (ICECA)*, IEEE, 2019, pp. 327–330.
- [15] S. Davis and P. Mermelstein, “Comparison of parametric representations for monosyllabic word recognition in continuously spoken sentences,” *IEEE transactions on acoustics, speech, and signal processing*, vol. 28, no. 4, pp. 357–366, 1980.
- [16] L. Sifre, “Rigid-motion scattering for image classification,” PhD thesis, Aug. 2014. [Online]. Available: https://www.di.ens.fr/data/publications/papers/phd_sifre.pdf.
- [17] P. Warden, *Speech commands: A dataset for limited-vocabulary speech recognition*, 2018. arXiv: 1804.03209 [cs.CL].
- [18] Google, *Speech commands dataset version 1*, 2017. [Online]. Available: http://download.tensorflow.org/data/speech_commands_v0.01.tar.gz.
- [19] D. P. Kingma and J. Ba, *Adam: A method for stochastic optimization*, 2017. arXiv: 1412.6980 [cs.LG].
- [20] N. J. Cotton, B. M. Wilamowski, and G. Dundar, “A neural network implementation on an inexpensive eight bit microcontroller,” in *2008 International Conference on Intelligent Engineering Systems*, 2008, pp. 109–114. DOI: 10.1109/INES.2008.4481278.
- [21] STMicroelectronics, *Stm32f769ni discovery*, 2021. [Online]. Available: <https://www.st.com/en/evaluation-tools/32f769discovery.html>.
- [22] STMicroelectronics, *Stm32f769ni reference manual*, 2021. [Online]. Available: https://www.st.com/resource/en/reference_manual/dm00224583-stm32f76xxx-and-stm32f77xxx-advanced-arm-based-32-bit-mcus-stmicroelectronics.pdf.
- [23] L. Shenzhen Yahboom Technology Co., *Jetbot ai robot car*, 2021. [Online]. Available: <http://www.yahboom.net/study/JETBOT>.

- [24] N. Corporation, *Jetson nano developert kit*, 2021. [Online]. Available: <https://developer.nvidia.com/embedded/jetson-nano-developer-kit>.
- [25] L. Seeed Technology Co., *Respeaker mic array v2.0*, 2021. [Online]. Available: <https://www.seeedstudio.com/ReSpeaker-Mic-Array-v2-0.html>.
- [26] A. Ltd., *Cmsis-dsp library for cortex-m microcontroller*, 2021. [Online]. Available: <https://www.keil.com/pack/doc/CMSIS/DSP/html/index.html>.
- [27] N. Corporation, *Tensorflow for jetson nano*, 2021. [Online]. Available: https://developer.download.nvidia.com/compute/redist/jp/v42/tensorflow-gpu/tensorflow_gpu-2.0.0+nv19.11-cp36-cp36m-linux_aarch64.whl.
- [28] NumPy, *Numpy library for python*, 2021. [Online]. Available: <https://numpy.org/>.
- [29] A. Industries, *Pca9685 data-sheet*, 2021. [Online]. Available: <https://cdn-shop.adafruit.com/datasheets/PCA9685.pdf>.