IMPLEMENTATION OF DEEP NEURAL NETWORKS ON EMBEDDED SYSTEMS FOR REAL-TIME HAND GESTURE RECOGNITION USING A 24 GHz DOPPLER RADAR

M.A.M. Ijaz

219436B

MSc in Telecommunication

Department of Electronic and Telecommunication Engineering
Faculty of Engineering

University of Moratuwa
Sri Lanka

June 2025

IMPLEMENTATION OF DEEP NEURAL NETWORKS ON EMBEDDED SYSTEMS FOR REAL-TIME HAND GESTURE RECOGNITION USING A 24 GHz DOPPLER RADAR

M.A.M. Ijaz

219436B

Thesis submitted in partial fulfillment of the requirements for the degree
MSc in Telecommunication

Department of Electronic and Telecommunication Engineering
Faculty of Engineering

University of Moratuwa
Sri Lanka

June 2025

DECLARATION

I declare that this is my own work and this Thesis does not incorporate without acknowledgement any material previously submitted for a Degree or Diploma in any other University or Institute of higher learning and to the best of my knowledge and belief it does not contain any material previously published or written by another person except where the acknowledgement is made in the text. I retain the right to use this content in whole or part in future works (such as articles or books).

Signature: M.A.M Ijaz Digitally signed by M.A.M Ijaz Date: 13/06/2025

The supervisors should certify the Thesis with the following declaration.

The above candidate has carried out research for the MSc in Telecommunication Thesis under our supervision. We confirm that the declaration made above by the student is true and correct.

Name of Supervisor: Dr. Chamira Edussooriya

Signature of the Supervisor: Digitally signed by Chamira U. S. Edussooriya Date: 13/06/2025

Name of Supervisor: Eng. Kithsiri Samarasinghe

Signature of the Supervisor: A.T.L.K. Samarasinghe 2025.06.13 10:22:18 +05'30' 2025.1.0 Date: 13/06/2025

DEDICATION

To everyone who helped us make this project a success.

ACKNOWLEDGEMENT

I express my sincere gratitude to our project supervisors, Dr. Chamira Edussooriya and Eng. Kithsiri Samarasinghe from ENTC, Unversity of Moratuwa, for guiding me through the project and encouraging me to reach the milestones that i have reached along my journey to make this project a success.

Then i would like to express my gratitude to the members of the previous year final year project group Mr. Gayangana, Mr. Kavindu , Mr. Nisal and Mr. Dhanuka for their support and guidance they have provided.

I would also extend my gratitude to all the staff members of the Department of Electronic and Telecommunication Engineering who helped me by providing valuable suggestions. My family and friends for the support and motivation provided throughout the project.

Last but not least, I would like to extend my thanks to my employer, Sri Lanka Telecom PLC and its management for supporting me throughout the journey and partially financing my studies.

ABSTRACT

Real-time hand gesture recognition is a key component of next-generation human-computer interaction, with applications in smart environments, assistive technologies, and contactless control systems. This research uses deep neural networks in embedded systems for radar-based gesture recognition, where the gesture signals are captured using a 24 GHz Doppler radar sensor, and the processing is done in the embedded edge computing resources.

The project uses two types of deep learning models: a vision transformer (ViT) that captures long-range components for superior classification and a deep convolutional neural network (DCNN) that captures local spatial features. This work uses hyperparameter optimization, model pruning, and integer quantization to increase accuracy and efficiency, but with minimal loss, to achieve real-time performance on low-power hardware. Hyperparameter optimization is used to improve model architecture performance. In order to achieve better performance and lower resource consumption, pruning is employed. The integer quantization shrinks the size of the deep learning models for lower-power devices with an increased inference speed.

The proposed system is deployed on an embedded platform while ensuring low latency, reduced power consumption, and real-time inference. The deployment results validate the feasibility of radar-based gesture recognition in embedded devices. The experiments compare the optimized DCNN and ViT models, evaluating their speed, memory usage, accuracy, and energy efficiency. The results highlight the trade-offs between accuracy and efficiency, providing valuable insight into the feasibility of deep learning for real-time radar-based gesture recognition on embedded devices. This research contributes to the advancement in the deployment of deep learning models in embedded devices.

Keywords: Hand Gesture Recognition, Vision Transformers, Deep Convolutional Neural Networks, Deep Learning Model Optimization, Embedded Systems

TABLE OF CONTENTS

Declaration of the Candidate & Supervisor i

Dedication ii

Acknowledgement iii

Abstract iv

Table of Contents v

List of Figures ix

List of Tables xi

List of Abbreviations xi

List of Appendices xiii

1 INTRODUCTION 1

1.1 Introduction 1

1.2 Problem Statement 2

1.3 Proposed Solution 2

1.4 Scope and Expected Deliverables 3

1.5 Contribution of the Dissertation 3

1.6 Organization of the Dissertation 4

2 LITERATURE REVIEW 5

2.1 Radar Systems and Micro-Doppler Signatures 5

2.2 Deep Learning Algorithms for Hand Gesture Recognition 6

2.3 Deep Learning Model Optimization 7

2.4 Embedded System Implementation 8

3 METHODOLOGY 10

3.1 System Architecture 10

3.2 Doppler Transceiver 11

3.3 Signal processing 12

3.4 Data Sets 13

3.4.1 Data Set : 14 Hand Gestures 13

3.4.2 Data Set : 6 Hand Gestures 15
3.5 Deep Learning Model 16
3.5.1 Vision Transformer Architecture 16
3.5.2 Deep Convolutional Neural Networks Architecture 18
3.6 Optimization of Deep Learning Model 19
3.6.1 Hyperparameters Tuning 19
3.6.2 Model Pruning 21
3.6.3 Model Quantization 22
3.7 Model Parameters 23
3.8 Building the Model 24
3.8.1 Training Model 24
3.8.2 Converting Model to TFLite 25
3.9 Storing / Loading Models 26
3.9.1 Storing Complete Trained Model 26
3.9.2 Storing Trained Model Weights 26
3.10 Data Prepossessing 27
3.11 Embedded System 28
3.11.1 Embedded System Development Environment 29
3.11.2 Embedded System Minimum Requirements 29
3.11.3 Cost-Benefit Analysis and Selecting an Embedded System 30
3.12 Generating C Files 31
3.13 Development Environment Setup 32
3.14 The Embedded System Application 34
3.15 Debugging the Embedded System 35
3.16 Energy Measurement 36
4 RESULTS 37
4.1 Terminology and Metrics 37
4.1.1 Confusion Matrix 37
4.1.2 Accuracy 37
4.1.3 Precision 38
4.1.4 Recall 38

4.1.5 F1-Score 38
4.1.6 Specificity 38
4.2 Deep Learning Model Training Results 39
4.2.1 DCNN Model 39
4.2.2 ViT Model 40
4.3 Deep Learning Model Quantization Results 41
4.3.1 DCNN Model Size Comparison 41
4.3.2 ViT Model Size Comparison 41
4.3.3 DCNN Model with Float Weights 42
4.3.4 DCNN Model with 8 bit Integer Weights 43
4.3.5 ViT Model with Float Weights 44
4.3.6 ViT Model with 8 bit Integer Weights 45
4.4 Deep Learning Model Pruning Results 46
4.4.1 Sparsity of Model 46
4.4.2 Sparsity DCNN Model Size Comparison 46
4.4.3 DCNN Model Pruned with Float Weights 47
4.4.4 DCNN Model Pruned with 8 bit Integer Weights 48
4.5 Deep Learning Model Inference Time 49
4.5.1 Emulation Results 49
4.5.2 Implementation Results 50
4.6 Energy Measurement Results 52
4.6.1 DCNN Model Energy Consumption 52
4.6.2 ViT Model with Encode and Decoder Energy Consumption 53
4.7 Discussion 54
4.7.1 Deep Learning Model 54
4.7.2 Deep Learning Model Hyperparameter Tuning 54
4.7.3 Deep Learning Model Pruning 54
4.7.4 Deep Learning Model Quantization 54
4.7.5 Model Inference Time 55
4.7.6 Model Power Consumption 55
4.7.7 Impact of the Project 55

4.7.8 Possible Applications 55
5 CONCLUSION AND FUTURE WORKS 56
References 57
Appendix A Embedded System Comparison 60
Appendix B Energy Measurement : Selected Components 61
B.1 Power Monitor IC 61
B.2 Power Monitor Controller 61
B.3 Power Source 62
Appendix C DCNN Model 63
Appendix D ViT Model 65
Appendix E Power Measurement Controller 72

LIST OF FIGURES

Figure	Description	Page
Figure 3.1	System Architecture	10
Figure 3.2	Application Diagram	11
Figure 3.3	Data Set 1 - Image of the gestures used	14
Figure 3.4	Data Set 2 - Swipe Diagonally Doppler Signature	15
Figure 3.5	ViT with Convolutional Encoder Decode	16
Figure 3.6	ViT Attention Module	17
Figure 3.7	DCNN Architecture	18
Figure 3.8	Sparsity Pruning	21
Figure 3.9	Quantization	22
Figure 3.10	Google Colab	24
Figure 3.11	TFLite Conversion	25
Figure 3.12	Embedded Systems	28
Figure 3.13	ST Edge AI Developer Cloud	31
Figure 3.14	STM32Cube-Board	32
Figure 3.15	STM32Cube-Settings	33
Figure 3.16	Debugging the Embedded System	35
Figure 3.17	Energy Measurement Setup	36
Figure 4.1	Accuracy-DCNN	39
Figure 4.2	Accuracy-ViT	40
Figure 4.3	Confusion Matrix-DCNN-Float	42
Figure 4.4	Confusion Matrix-DCNN	43
Figure 4.5	Confusion Matrix-ViT-Float	44
Figure 4.6	Confusion Matrix-ViT	45
Figure 4.7	Sparsity vs Epochs	46
Figure 4.8	Confusion Matrix-DCNN-Pruned	47
Figure 4.9	Confusion Matrix-DCNN-Pruned-Quantized	48
Figure 4.10	Performance Summary Chart	49
Figure 4.11	DCNN Inference Time	50
Figure 4.12	ViT Inference Time	51
Figure 4.13	DCNN Model Energy Consumption	52
Figure 4.14	ViT Model with Encode and Decoder Energy Consumption	53
Figure A.1	ARM Processor Class	60
Figure B.1	Power Monitor IC - INA226	61
Figure B.2	Power Monitor Controller - Arduino Nano 33 BLE Sense	61

Figure B.3 Power Supply 62

LIST OF TABLES

Table	Description	Page
Table 3.1	Deep Learning Models Comparison	16
Table 3.2	ViT Model - Encoder Filter Specification	17
Table 3.3	DCNN Model - Filter Specification	19
Table 3.4	DCNN Hyperparameters Tuning	20
Table 3.5	ViT Hyperparameters Tuning	20
Table 3.6	Software Packages	24
Table 3.7	Data Prepossessing	27
Table 3.8	Embedded Systems Comparison	30
Table 4.1	DCNN Model Size	41
Table 4.2	ViT Model Size	41
Table 4.3	DCNN Model with Float Weights	42
Table 4.4	DCNN Model with 8 bit integer Weights	43
Table 4.5	ViT Model with Float Weights	44
Table 4.6	ViT Model with 8 bit integer Weights	45
Table 4.7	DCNN Model Size before and after Pruning	46
Table 4.8	Confusion Matrix-DCNN-Pruned	47
Table 4.9	Confusion Matrix-DCNN-Pruned-Quantized	48
Table 4.10	DCNN Model Energy Consumption	52
Table 4.11	ViT Model with Encode and Decoder Energy Consumption	53

LIST OF ABBREVIATIONS

Abbreviation	Description
AI	artificial intelligence
BLSTM	bidirectional long short-term memory
CNN	convolutional neural network
CTC	connectionist temporal classification
CW	continuous wave
DCNN	deep convolutional neural network
DL	deep learning
DT	decision tree
FFT	fast fourier transform
FMCW	frequency modulated CW
FPU	floating point unit
HAL	hardware abstraction layer
HCI	human-computer interaction
HGR	hand gesture recognition
HMM	hidden markov model
HMR	human motion recognition
IoT	internet of things
LO	local oscillator
MDS	micro Doppler signatures
MLP	multi-layer perceptron
MSENet	multiscale squeeze-and-excitation network
MViT	multiscale vision transformers
NLP	natural language processing
PTQ	post-training quantization
QAT	quantization-aware training
SFCW	single frequency CW
STFT	short time fourier transform
SVM	support vector machine
TFLite	tensorflow lite
ViT	vision transformer

LIST OF APPENDICES

Appendix	Description	Page
Appendix -A	Embedded System Comparison	60
Appendix -B	Energy Measurement : Selected Components	61
Appendix -C	DCNN Model	63
Appendix -D	ViT Model	65
Appendix -E	Power Measurement Controller	72

CHAPTER 1

INTRODUCTION

1.1 Introduction

The hand gesture recognition (HGR) is an important component in human-computer interaction (HCI) and a crucial component in the digital world, enabling the design of technology that is user-friendly, productive, secure, and pleasant to use. HGR is a major component in augmented/mixed reality systems, human-machine teaming, internet of things (IoT) driven designs, healthcare and defense applications [1] [2] [3] [4]. HGR systems can be classified into two broad categories as contact-based systems and non-contact-based systems.

In contact-based HGR systems, a device is attached to the user's body, such as a sensor or an RFID tags [4]. In the non-contact approach vision, acoustic and RF-based sensors are used to detect human gestures [5]. The noncontact approach has several advantages over that of the contact-based approach, such as being user-friendly, convenient, and hygienic, namely by reducing the risk of viral/bacterial infection in clinical applications [1], hence widely used in modern-day applications but is unable to distinguish signals from different users [6].

When considering the vision and acoustic-based approach, they have some disadvantages such as being susceptible to noisy background environment, dimmed lighting conditions, rain, smoke and external noise [1] [2]. When considering RF sensing approaches, there are no such disadvantages due to the adverse environmental conditions, and on the contrary, there are some key advantages such as protecting user privacy which is not in the case of vision and acoustic modality and hence minimize the user discomfort.

The radars operating in millimeter wavelength band such as in 24 GHz or 60 GHz has the advantage of being able to detect fine movement of fingers such as hand gestures due to the shorter wavelength of the transmitted signal over the WIFI based detection operating in 2.4 GHz or 5.8 GHz [2] [4]. Doppler radars typically work using the principle of Doppler effect, which is the phenomenon in which the reflected wave wavelength from an object changes due to the relative movement between object and observer. In addition to the Doppler shift due to the relative movement of between object and observer, we can observe a micro Doppler shift due to the micro motion in the object. These micro Doppler shifts due to the micro motions are referred to as micro Doppler effect[7]. The micro Doppler signatures (MDS) has multiple applications depending on the field of application, such as human activity detection, structural health monitoring of buildings, surveillance, and determining the nature of the target in military applications using the known signatures [7].

1.2 Problem Statement

HGR systems require a deep learning model to classify gestures into known predefined gesture classes, so that the system could be integrated as input sources for other systems and used. Deep learning models such as ViT based model are complex and require lots of computing resources to implement, hence are normally implemented on rich computing environments, and hence cost more per deployment of the HGR with deep learning models.

The deployment cost per unit must be minimized in order to improve the adaptation of HGR systems. In order for HGR system to be deployed in wide-ranging IoT systems, such as in the case of energy-limited systems, the energy consumption per classification are also to be studied.

The deep learning models used with HGR have different architectures; therefore, these models differ much in prediction accuracy, resource usage, implementation complexity, and power consumption, and hence these should also be studied to gain a better understanding of the model.

1.3 Proposed Solution

The project objective is to implement the ViT based model with a convolutional encoder decoder developed in [2] to classify human gestures from a 24 GHz Doppler radar in an embedded system. The model is trained to classify 14 hand gestures with an accuracy of 98.4% [2].

The project aims to reduce the implementation cost for the radar-based HGR system by deploying the system in an embedded environment, which costs less to implement and run, but is constrained by low memory, processing power, storage, and consumes less power. The reduction in cost will propel the wider adoption of Doppler radar-based systems for HGR.

The proposed project aimed to implement the ViT model developed in an embedded device and evaluate the results against other models, such as the simple DCNN-based model. The model will be fine-tuned to run on the selected embedded system. The target embedded system is chosen to be ARM based due to its wider adaptability. The solution aims to reduce the loss of prediction accuracy in the above process as much as possible when the models are subjected to the above tuning process.

1.4 Scope and Expected Deliverables

The main expected deliverable in the project is to implement the optimized ViT base model with encoder and decoder for hand gesture classification in a selected embedded system with prediction accuracy and inference time comparable to the original model. Achieving this requires the model to be optimized, pruned, and quantized to reduce the cost of implantation, thereby allowing it to be run on an embedded device. This requires analyzing the existing ViT model, reviewing relevant research material on model optimization, and studying similar existing implementations. A comparative study of the model and the software to test the model is also in the scope.

In terms of expected deliverables, the project expect

- Optimized the ViT based model to improve prediction speed while trying to reduce the loss of accuracy.
- Implement the developed micro-Doppler signature-based hand gesture prediction model which uses the ViT architecture coupled with encoder and decoder in an embedded system.
- Conduct a comparative study between the ViT model and the DCNN-based model to perform the same task and analyze this on different embedded systems.

1.5 Contribution of the Dissertation

The project could be broken down into a few stages after analyzing the exiting system and finalizing the desired results. The project considers an alternative strategy for each stage and decides the best based on the cost-benefit analysis. The project can be divided into few major steps as follows.

1. Analyze the existing deep learning (DL) implementation in an embedded system.
2. Optimize the ViT architecture developed coupled with the encoder decoder-based model.
3. Implement the optimized model in the selected embedded system or systems.
4. Compare and contrast the results.

The primary results of this project have demonstrated the feasibility of implementing deep neural networks in embedded systems for real-time hand gesture recognition using a 24-GHz Doppler radar. Specifically, the optimized ViT model achieved a recognition accuracy of 99.80% on a 14 hand gesture data set, running on an ARM Cortex-M7 microcontroller with an inference time of 345.690 milliseconds. DCNN based implementation achieved a recognition accuracy of 94.29% with an inference time of 213.111 milliseconds under the same conditions. This performance highlights the potential for real-time applications in human-computer interaction using radar-based gesture recognition.

1.6 Organization of the Dissertation

The dissertation was organized into 5 chapters. Chapter 1 discusses the introduction to the problem and the proposed hand gesture recognition solution. Chapter 2 reviews the available literature on radar systems, micro-Doppler signatures, deep learning algorithms used with hand gesture classification, deep learning model optimization, and existing implementation of deep learning models on embedded systems. Chapter 3 discusses the proposed solution with respect to the models used, the optimization, and the process of converting them to the executable. The chapter also discusses the power measurement setup used to measure the energy usage of each model. Chapter 4 presents and discusses the results of each model in the scenarios considered. Chapter 5 concludes the thesis with the final conclusion and future work.

CHAPTER 2

LITERATURE REVIEW

2.1 Radar Systems and Micro-Doppler Signatures

Radar technology is emerging as a promising tool for gesture detection, offering several advantages over traditional camera-based systems, such as the ability to detect through the wall, insensitive to illumination, background noise and dust, while maintaining user privacy, with a high gesture detection accuracy and a relatively low processing cost [1] [8] [9] [3] [5]. These advantages make radar an ideal tool for HCI.

Radar is an active sensing technology that works in the principal in transmitting electromagnetic waves and receiving returned waves from targets. Radars could operate as pulse or continuous wave, in continuous wave (CW) , it could be frequency modulated CW (FMCW) or single frequency CW (SFCW) [6]. Radars used the Doppler effect, which is the change in frequency of the source signal due to the relative motion of the observer and object. The radar target may have micro-motions, such as vibrations or rotations, which generate modulations on the back-scattered signal and would induce additional Doppler changes to the constant Doppler frequency shift of the bulk translational motion which is referred to as micro-Doppler effects, this micro-Doppler effect is much observable when the signal frequency is high [7]. Millimeter waves are an excellent choice to observe micro-Doppler effects [4].

These micro-Doppler effects could be used to sense human interactions such as hand gestures, and could be used to devices such consumer electronic devices, AR/VR control human robot interaction, clinical application, etc. [10] [11] [1]. Google's soli project works in millimeter wavelength at 60 GHz and is already integrated into consumer devices like the Google Pixel 4 smart phone [4] [12].

The FMCW radar operating at 24 GHz is the most common choice used for hand gesture recognition [1] [2]. There are multiple publicly available data sets for radar, i am primarily concerned with two data sets, data set one was collected using one Tx antenna and two Rx antennas [1], the data set two was collected using one Tx and four Rx antennas [2]. The project aims to work with this data set for the training and testing of the proposed system.

2.2 Deep Learning Algorithms for Hand Gesture Recognition

Deep learning is a powerful set of tools when it comes to classification problems. Once trained, deep learning models can automate the classification process, saving time and resources compared to manual classification. For the classification of micro-Doppler signatures, different deep learning techniques are used. In most recent research, activity spectrograms obtained using Doppler radar and fast fourier transform (FFT) techniques are fed to the classifier as input [2].

The support vector machine (SVM) and decision tree (DT) are used in [12] and have yielded classification accuracy's of 99.5% and 93.1%.[9] have used convolutional neural network (CNN) to classify 13 human movements and achieved an overall prediction accuracy of 95%. [13] used DCNN to classify 10 hand gestures with an average accuracy of 85.6%. State-transition-based hidden markov model (HMM) developed by [14] has achieved an overall end-to-end accuracy of 96.34% for classifying 7 hand gestures.[15]used a network composed of multiscale squeeze-and-excitation network (MSENet) , stacked bidirectional long short-term memory (BLSTM) , and connectionist temporal classification (CTC) algorithms for the purpose of continuous human motion recognition (HMR) from spectrograms. This classification architecture was able to achieve accuracies of 93.12% and 96.09% for continuous HMR with and without interference, respectively [2].

DCNN was used in [1] to classify 14 different hand gestures captured from two-antenna Doppler radar system. They prefer DCNN due to not requiring pre-defined feature estimation, and the ability of the network itself to learn the features during the training process by optimizing the network weights. They were able to achieve an average accuracy of 95.5% for hand gesture recognition of the user. They have achieved a cross-test accuracy of 51.22% when user 2 gestures were classified in the system trained on user 1. When they have mixed the training data, they obtained accuracy jumps to around 88.08%. The data set from this is publicly available and I have used it in the project.

ViT is a powerful tool for image classification adapted from the transformer model [16] initially designed for natural language processing (NLP) and has been adapted for image classification initially proposed by [17]. Since then, ViT has been extensively adapted for image classification problems, such as multiscale vision transformers (MViT) [18]. A ViT-based model coupled with an encoder-decoder was developed by [2] to classify HGR in an automated manner. The model uses a convolutional decoder with transposed convolution to up-sample the feature vectors, and a convolutional decoder is used to enhance the spatial size of feature maps so the transformer has enough pixels in each patch. The output of the encoder-decoder mechanism is then divided into 12 patches and fed into the ViT model. The patches are normalized, then subject to the multi-head attention mechanism with 4 heads, the output then is

fed to the multi-layer perceptron (MLP) network with a fully connected 1025*512 neuron with a dropout rate of 0.5, which function as the classifier. The model achieved a prediction accuracy of 98.3% for the dataset [1] compared to 95.5% for the model proposed in [1]. The project aims to implement this model [2] in the embedded system with necessary adjustment subject to prediction accuracy comparable to the original model.

2.3 Deep Learning Model Optimization

Deep Learning models are comparatively large when trained and have a large number of trained weights, making it difficult to run on a constrained environment like an embedded system, which typically has RAM and flash memory on the order of a few kilobytes and has limited processing power and without a dedicated floating point unit in some architectures; hence, the models need to be optimized for the target embedded systems if it is to run on them. There are a number of tools to optimize a deep learning model; they could be classified as pre-training and post-training optimizations. The use of a point-wise light weight convolution instead of graph convolution [19] is an example of pre-training optimization. The use of weight quantization [20] [21], pruning [22] [23] are few of the well-known methods for post-training optimizations.

Pre-training optimization involves identifying room for improvement and doing the necessary adjustment without significantly affecting the models overall accuracy. [20] discuses the use of light point-wise convolution as an alternative to bulky graph convolutions. [17] discuses about hyper-parameter tuning, which are parameters set before training and govern how learning is done. These could be like numbers of training epochs, learning rate, MLP size, number of attention heads, etc. Model simplification is also a pre-training technique, which improves the model and avoids over-fitting. These pre-training optimization could either be calculated mathematically or are done through trial and error to find the optimums.

Pruning refers to the process of removing connections in the deep learning model that have a lower impact on the prediction accuracy of the model. Model pruning not only improves the size of the model, but also slightly improves the performance of the model [22], which is a key step in fitting the model in an embedded system and then running with limited resources. Pruning could be divided into global pruning and local pruning [22]. In global pruning, connections are pruned on the basis of it globally, and in local pruning, this is done in layer levels. [22] suggest the use of a fine-grained analysis of the model layers when pruning as the model error increases beyond a pruning level. [22] proposes cluster pruning, which is pruning in-cooperating hardware-dependent parameters and follows a rule-based greedy algorithm to prune the entire network. This approach [22] is well adapted for embedded systems and has achieved an improvement in the latency and precision perspectives when pruning.

Quantization refers to the technique that reduces the precision of a model's parameters to make it run faster and use less computing power. Quantization is a powerful tool that allows the integration of modern neural networks in edge devices with strict power and computing requirements [24]. Quantization can broadly be classified as post-training quantization (PTQ) and quantization-aware training (QAT) [24]. The post-training quantization method, the mixed precision quantization proposed by [21] has achieved a reduction of 25% in memory and 44% in computational costs compared to conventional post-training quantization methods. The projects aims to subject the model to post-training quantization, and convert the weight to 8 bit integer value to allow it to be run in an embedded system without floating point unit.

2.4 Embedded System Implementation

Deploying deep learning models on embedded devices such as micro-controllers makes significant contributions for wider adaption artificial intelligence (AI) [19]. The embedded systems are typically less expensive compared to regular computers or the custom-build machines with dedicated GPUs designed to run deep learning modules. Running DL on an embedded system with constrained computing resources requires optimizations and adjustments to the existing model [20], but embedded systems have the advantage of being very low cost compared to a typical computing device and offer a better price for greater adaption.

Embedded systems can be classified on the basis of microcontroller performance as small, medium, and sophisticated. Devices like Arduino Nano 33 BLE is an example of small-class embedded systems and a Discovery kit with STM32F746NG MCU [20] is an example of a mid-range devise. Raspberry Pi 4 [19] or a Jetson Nano are in the sophisticated class of embedded devices. Certain devices such as in the case of small-class embedded systems, might not have a dedicated floating point unit (FPU) and, in contrast, devise like Raspberry Pi 4 can even run a fully functional OS, and devices like Jetson Nano have their own dedicated GPU. Depending on the class of devices, the available computing resources and the cost of the devices change.

Google's project soli radar [11] operates in V-band with a central frequency of 60 GHz and bandwidth of 6 GHz which was deployed in an embedded environment with there smart phone google's Pixel 4 and the radar sensor is used to detect human gestures and provide an interactive expedience to the user. The miniaturization of radar sensors and processing devices reduces the cost of the systems and allows them to be deployed in the consumer environment, thus allowing a wider adoption of radar-based sensors. [19] has used a Raspberry Pi 4 unit running the raspbian 10 OS to run a CNN based model to detect disease in tomato leaves, model training was done on an intel i7 PC with NVIDIA GPU with 8GB of RAM. The Raspberry Pi 4 unit used as the edge devise to detect the disease with the trained model.

A ViT-based model was used on an STM32F746 micro controller with 340 kB of memory and 1 MB of flash [20]. In the implementation stage, the researcher came across the issue with limited library support for ViT operation and has developed his own implementation for a certain function in the MCU. They have converted the tensors to int8 to reduce maximum memory usage. The maximum memory usage was limited to 256 kB, and was able to achieve 74.0% top 1 accuracy for the classification problem.

CHAPTER 3

METHODOLOGY

Concepts related to deep learning model architecture, deep learning model optimization, converting deep learning models to executables, and energy measurement of the models are discussed.

3.1 System Architecture

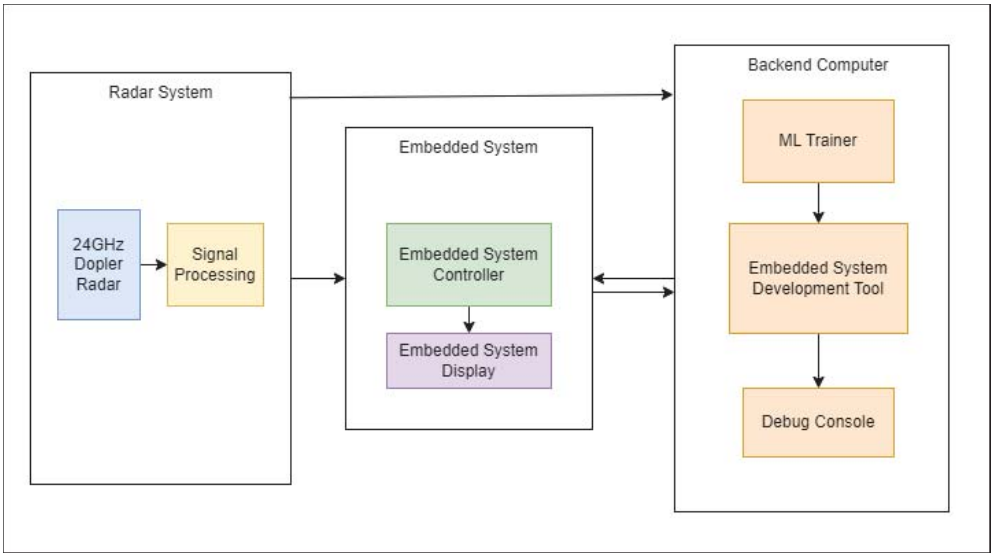

Fig. 3.1: System Architecture

The proposed system architecture is shown in Fig 3.1. The spectrum of the gestures is captured using the 24 GHz radar over 3 seconds, which is then subject to signal processing to obtain the spectrogram of the gesture. The spectrogram is then fed into the embedded system which is already trained using the samples collected to classify the hand gestures performed. The model is developed in tenser-flow in the Jupiter notebook environment in the backend computer system. The model trained can classify 14 different hand gestures and be optimized, pruned to run on embedded systems, and able to run using only integer operations which leads to faster inference. The tenser-flow model is then converted to the tflite model, which is an optimized environment for embedded systems. Then i have used the ST edge AI tool to convert the tflite model to C code and the STM cube IDE to program the embedded system. The project used two different DL classification models in the system.

The project expects the upper limit of inference to be in the 3-second interval to allow the classification result obtained before the next gesture is fed to the system, and ideally expects to be within 1 second to allow for a better user experience.

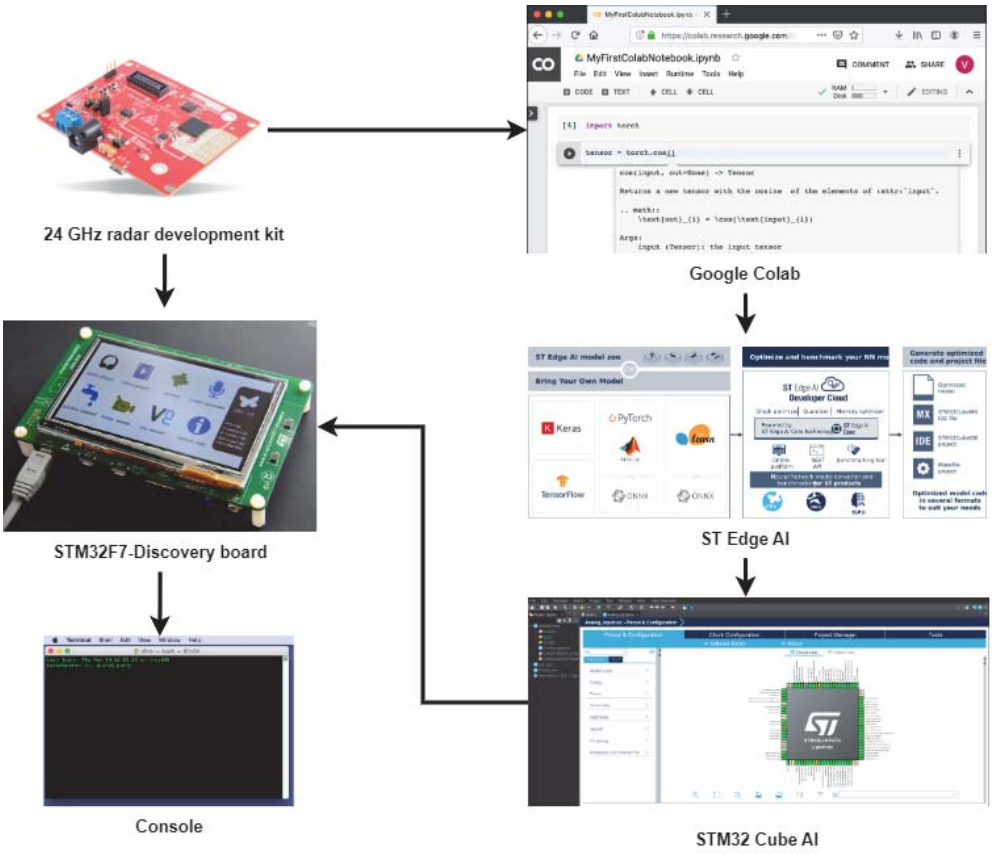

Fig. 3.2: Application Diagram

The Fig 3.2 summaries how system is trained and developed. This shows how the model is trained and transferred to the embedded system.

3.2 Doppler Transceiver

The project has used the same continuous-wave Doppler transceiver designed by [2]. It consists of a single transmitter antenna and four receiver antennas. The transmitter works on the central frequency of 24 GHz in K-band in the spectrum generated from a signal generator, the frequency is chosen for its superior performance for detection of micro gestures due to its shorter wavelength. The gestures will be captured for 3 seconds and will be sampled 8000 samples/s and each with hamming windows size of 32 ms hence giving 256 samples per gesture performed, with 50% overlap giving around 64 useful frequency points. The signal is retrieved from each of the four antennas and processed, and the processed Doppler spectrogram obtained is obtained as an image for the classification of hand gestures. The image data are then fed to the embedded systems using the UART communication protocol to predict hand gestures in real time. The spectrogram image is further stored and used to train the DL model.

Consider a typical Doppler radar that transmits a continuous wave signal sourced from its local oscillator (LO) that has an analytical representation of $e^{\jmath 2\pi f_0 t}$. When a set of L non-stationary targets moving with accelerating radial velocities $\{v_1(t), v_2(t), \ldots, v_L(t)\}$ will cause different frequency shifts to the incident signal due to the Doppler effect. These frequencies are given by $\{f_{d1}(t), f_{d2}(t), \ldots, f_{dL}(t) = \frac{2f_0}{c} v_L(t)\}$ as seen from the radar perspective. The beat signal resulting from mixing the transmitted and received signals is given by (3.1).

$$\begin{aligned} x(t) &= \underbrace{\exp\left(-\jmath 2\pi f_0 t\right)}_{\text{LO signal}} \times \underbrace{\sum_{l=1}^{L} \exp\left(\jmath 2\pi \left[f_0 + f_{d_l}(t)\right] t\right)}_{\text{Return signal}} \\ &= \sum_{l=1}^{L} \exp\left(\jmath 2\pi f_{d_l}(t) t\right). \end{aligned} \tag{3.1}$$

The signal $x(t)$ contains the Doppler signatures that correspond to the unique movements of the targets.

3.3 Signal processing

The signal was subjected to short time fourier transform (STFT) to extract Doppler signatures from each hand gesture. which can be mathematically represented as (3.2),

$$X[k, m] = \sum_{n=-\infty}^{\infty} x[n] w[n - m] e^{-jkn\frac{2\pi}{N}}. \tag{3.2}$$

Where, $x[n]$ are the radar time domain samples, $w[n - m]$ is the windowing function, n is the index of the time domain samples, k is the index of the frequency domain samples, m is the window stride index and $X[k, m]$ are the frequency domain samples. The spectrogram is obtained from $|X[k, m]|^2$ for each frame.

The hamming window is used as the windowing function and the window size is set to 256 samples. The overlap selected between subsequent window frames is 50%, that is, 128 samples. The parameters were chosen according to the time and frequency resolution requirements for the generated spectrogram images.

3.4 Data Sets

The project used two sets of data to train and test the deep learning model and its implementation in the embedded system. The project has used 80% of the labeled data for training and 20% of the labeled data for validation.

3.4.1 Data Set : 14 Hand Gestures

This data set contains around 3500 spectrogram data as images of size 180 by 60 pixels, labeled for 14 different hand gestures, collected using a two-transmission receiving antenna system by the researchers of [1], which are publicly available for use. bellow is the list of 14 different hand gestures.

1. Opening then closing of fingers together (single blinking)
2. Double blinking
3. Moving open hand towards and away from the radar(single to-and-fro)
4. Double to-and-fro
5. Rotating hand in an anti-clockwise direction (single round)
6. Double round
7. Swiping using two fingers (single swiping)
8. Double swiping
9. Moving hand towards and away with thumbs-up (single thumbs up)
10. Double thumbs-up
11. Fingers waving while the palm stays still (single waving)
12. Double waving
13. Sliding the hands from left to right (single sliding)
14. Double sliding

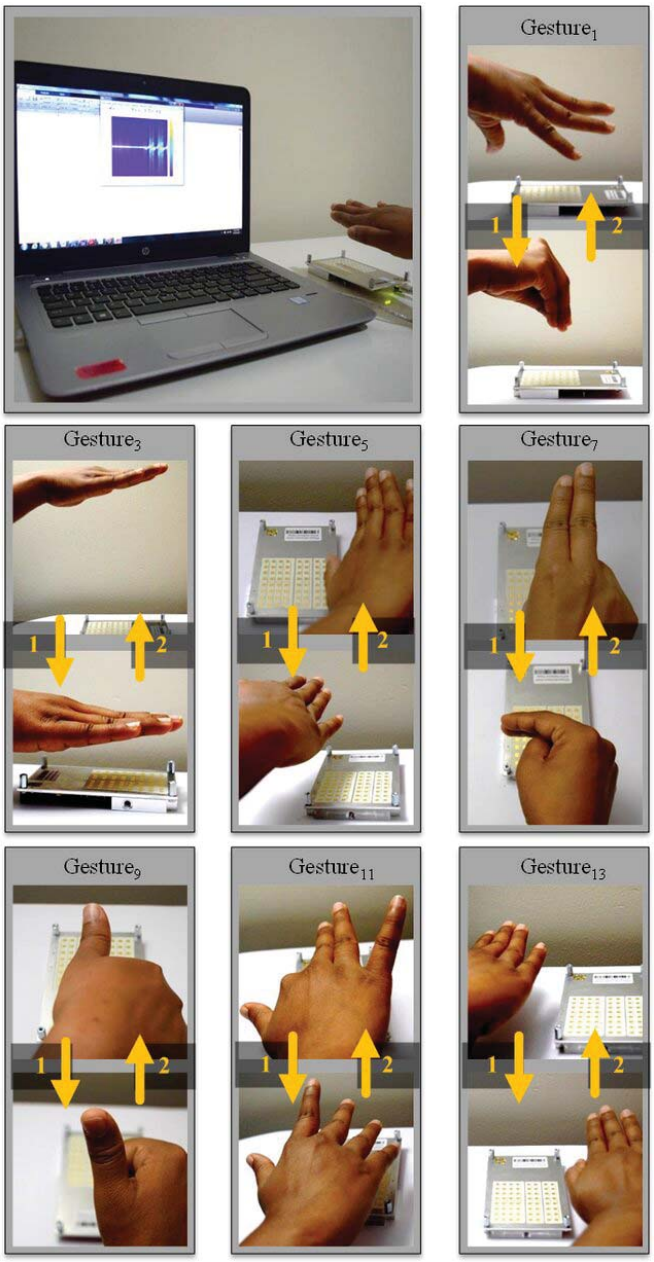

Fig. 3.3: Data Set 1 - Image of the gestures used
Source: https://www.researchgate.net/publication/330214186_Hand-Gesture_Recognition_Using_Two-Antenna_Doppler_Radar_With_Deep_Convolutional_Neural_Networks

The photos of the gestures are shown in Fig 3.3. Each of the 14 gestures has around 250 samples totaling 3,000 spectrogram samples. The data is used with 5-fold validation that is 80% for training and 20% for validation. The two of the three channels in the images represent the spectrogram from antennas 1 and 2 and the third Chanel represents the angle of arrival.

3.4.2 Data Set : 6 Hand Gestures

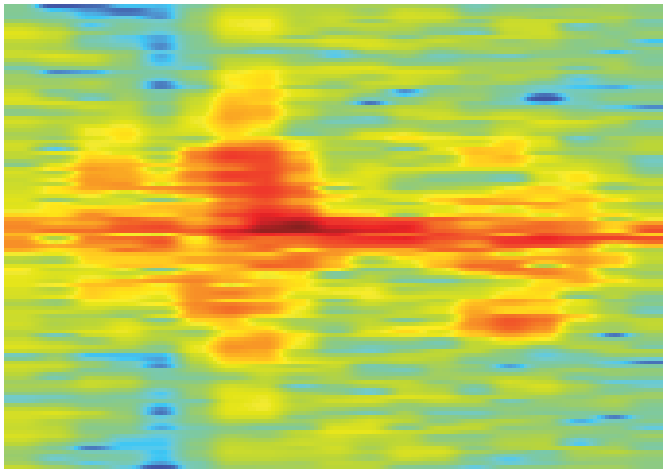

Fig. 3.4: Data Set 2 - Swipe Diagonally Doppler Signature

The Doppler signature of a gesture from the dataset is depicted in Fig 3.4. This dataset contains around 7000 spectrogram images of size 170 by 120 pixels spread over 6 different hand gesture classes and a class for no gestures as well. The data were collected by [2]. The following is a list of the six different classes of hand gestures.

1. Single Push – Person does a single pushing gesture, facing the palm towards the antenna

2. Double Push - Person does a single pushing gesture twice, facing the palm towards the antenna

3. Swipe Up-Down - Person facing towards the antenna bring his hand up and then down.

4. Swipe Diagonally - Person facing towards the antenna bring his hand forward while swiping to a right.

5. Swipe - Person facing towards the antenna lift his hand where his fingers are at the eye level. Then swipe the hand from right to left.

6. Go Away - Person facing towards the antenna bring his hand forward. Then he points his fingers away from the antenna, which were curled initially.

3.5 Deep Learning Model

The main objective of this project is to implement the ViT with the convolutional encoder decoder model developed by [2] for hand gesture clarification in an embedded system and to evaluate the results against a similar deep learning model. The reference model is chosen as a simple DCNN-based model similar to the one used in [1]. The project aims to tweak the models without affecting the overall prediction accuracy to achieve the project goal.

Each and every deep learning model has key differences between them, their own advantages, and disadvantages due to the architecture of the design. This makes them ideally suitable for different applications. The differences among the models considered could be summarized as in bellow Table 3.1.

TABLE 3.1: DEEP LEARNING MODELS COMPARISON

	ViT	DCNN
Building Component	Transformer	Convolutional layer
Data Requirement	Requires large datasets	Data-efficient
Computational Cost	Requires GPU	Relatively low
Scalability	Very High	Moderate
Execution time	Slow at inference	Comparatively fast

Note. Comparison between the ViT and DCNN Architectures

3.5.1 Vision Transformer Architecture

Transformer based classification has revolutionized the field of deep learning, particularly in natural language processing. Transformers rely on a mechanism called "attention" to weigh the importance of different parts of the input data. The transformers are adapted to vision classification problems with some changes because of their inherent advantages. Fig 3.5 shows the ViT model with the convolutional encoder detector developed by [2]. This model is to be implemented on an embedded system.

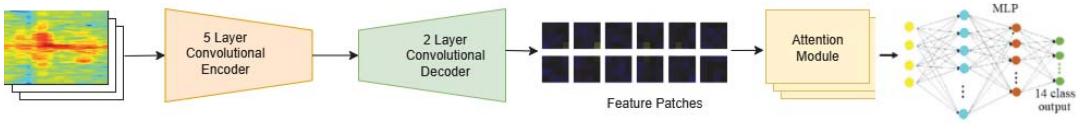

Fig. 3.5: ViT with Convolutional Encoder Decode

The ViT with convolutional encoder decoder developed by the [2] could be divided into few stages.

- Convolutional Encoder - Extract features in the image.
- Convolutional Decoder - Enhance the spatial size of feature maps.
- Vision Transformer - Understand context and relationships within the data.
- Multi-level Perception - Play the role of classifier.

Convolutional encoder used in the architecture to extract features; there are 5 layers of convolutional blocks with different number of filters in each layer.

TABLE 3.2: VIT MODEL - ENCODER FILTER SPECIFICATION

Layer	Kernal size	Number of filters	Output size
Input	-	-	$180 \times 60 \times 3$
Conv 1	7×7	4	$180 \times 60 \times 4$
Conv 2	5×5	8	$90 \times 30 \times 8$
Conv 3	3×3	16	$45 \times 15 \times 16$
Conv 4	3×3	32	$23 \times 8 \times 32$
Conv 5	3×3	64	$12 \times 4 \times 64$

The convolutional decoder is used to enhance the spatial size of features, making sure the ViT has enough pixel in each patches feed. They have used two transposed convolutional layers with 64 filters with 1×1 kernals and stride 2×2. The resultant feature map from the convolutional decoder of size $24 \times 8 \times 64$ is fed to the attention module.

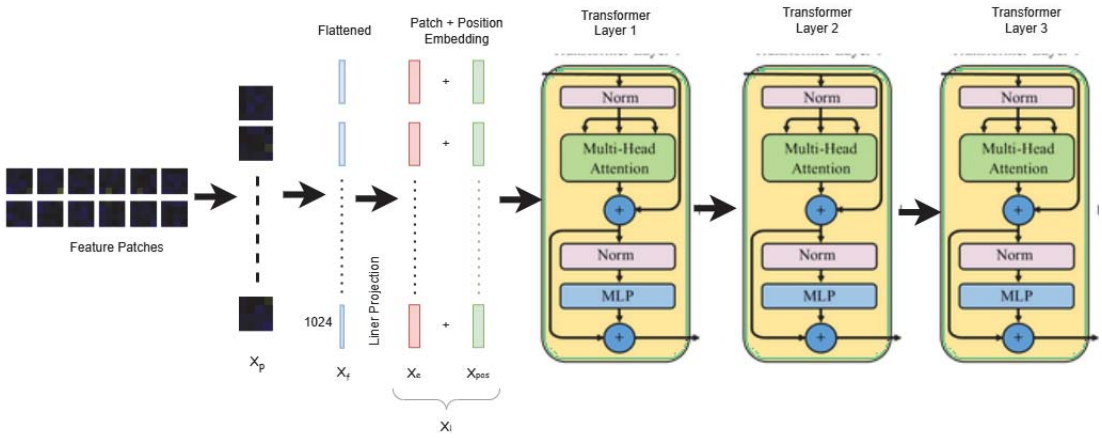

Fig. 3.6: ViT Attention Module

The output of the convolutional encoder is then fed into the ViT block, where the input is divided into 12 patches, flatten, and a positional embedding is added and normalized. The model has set the number of attention head to 4 and the dimension of Quarry, Key and Value matrices to 16. The output of the transformer layer is fed into the next transformer layer, resulting in a total of three successive transformer layers shown in Fig 3.6. Next, to provide a residual connection to the transformer, the input embeddings and output from the multi-head attention layer are added together. This tensor is then normalized and fed into the MLP block. MLP of size 1024 neurons and 512 neurons with a dropout rate of 0.5 chosen to reduce overfitting.

3.5.2 Deep Convolutional Neural Networks Architecture

Deep convolutional neural networks are a popular choice for human gesture classification, utilizing convolutional layers to effectively learn spatial and temporal features directly from raw input data, while also offering a relatively straightforward implementation. A simple DCNN model for human gesture classification based on the model developed by the researchers [1] is shown in Fig 3.7, which will be used in the project.

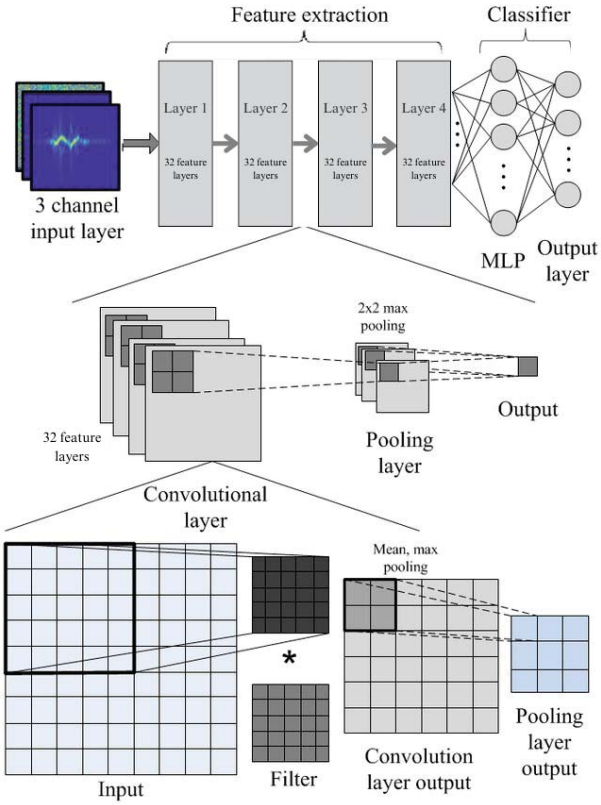

Fig. 3.7: DCNN Architecture

The DCNN architecture consists of 4 layers for feature extraction. Table 3.3 shows the filter configuration in the DCNN model used for hand gesture detection.

TABLE 3.3: DCNN MODEL - FILTER SPECIFICATION

Layer	Kernal size	Number of filters	Output size
Input	-	-	$180 \times 60 \times 3$
Conv 1	3×3	32	$178 \times 58 \times 32$
Conv 2	3×3	32	$87 \times 27 \times 32$
Conv 3	3×3	32	$41 \times 11 \times 32$
Conv 4	3×3	32	$18 \times 3 \times 32$

After each convolutional layer, a rectified linear unit (ReLU) is used as the activation function. Next, there is a 2×2 max-pooling layer. Finally, a general fully connected multi-layer perceptron (MLP) of size 128 x 128 is used for classification of gestures.

3.6 Optimization of Deep Learning Model

Deep learning model optimization is the process of tuning a model to achieve the best possible performance on a given task. This involves adjusting various aspects of the model, such as its architecture, hyperparameters, and training process, to minimize errors and maximize accuracy. The goal of the project is to maximize resource efficiency, paving the way for better performance with fewer computational resources (memory, processing power, and energy consumption). The project aims at having faster training and inference through this optimization process. This section will discuss the various optimization techniques used.

3.6.1 Hyperparameters Tuning

Hyperparameters tuning is a key step in the optimization of deep learning models, which affects the learning rate, the number of neurons in a neural network. The main goal is to shrink the models' memory foot print to allow the model to fill in an embedded system without affecting models accuracy.

In the DCNN model, I decided to change the number of layers in the model to four from three, and each layer has 32 filters each and used a dense layer of size 128, when compared to the reference DCNN model [1], the model has larger memory footprint but yields better classification accuracy 97.29% than the original models 95.50%. The model is still able to fit conformably to the desired embedded system. A comparison of the hyperparameters for the DCNN architecture is show in bellow Table 3.4 .

TABLE 3.4: DCNN HYPERPARAMETERS TUNING

	DCNN Model [1]	DCNN Model
Input	180 × 60 × 3	180 × 60 × 3
Convolution Layers	3	4
MLP Size	14	128
Batch Size	10	32
Training Epoch	10	100
Total Params	3638 (14.21 KB)	67438 (263.43 KB)
Model Accuracy	95.50%	97.29%

In the ViT model [2], the original model size was on the order of megabytes. which would make it impossible to implement on an embedded system. When the model was analyzed, it appeared that we could make some changes to the model parameter to simplify the model. The model had three successive transformer layers and a large MLP layer that followed the transformer layer.

The model was changed such that only one transformer layer is in the architecture and the MLP size was reduced considerably. We also made changes on the batch size to allow the model to learn slowly but to have improved prediction accuracy to compensate for the other changes. The Table 3.5 bellow shows the summary of the hyperparameter changes made for the ViT model [2].

TABLE 3.5: VIT HYPERPARAMETERS TUNING

	ViT Model [2]	ViT Model
Input	180 × 60 × 3	180 × 60 × 3
Transformer Layers	3	1
MLP Size	[1024, 512]	[256, 128]
Batch Size	64	32
Training Epoch	100	100
Total Params	821193 (3.13 MB)	133449 (521.29 KB)
Model Accuracy	98.3%	99.96%

The new model after the hyperparameter tuning was considerably smaller in size than that of the original model by around a factor of six but could still predict with improved accuracy.

3.6.2 Model Pruning

Pruning in deep learning is a technique that removes unnecessary parts or connections from a model to make it more efficient. This process reduces overfitting, increases generalization, and increases efficiency. The purpose of using the pruning approach was to reduce the number of parameters in the model, thus allowing the model to fit into the embedded system. Fig 3.8 shows the effect of pruning on the wights of the model.

Sparsity pruning was used to make the model lighter. The target was to remove unwanted connections as much as possible while preserving the model's functionality. pruning was done with an initial sparsity of 0.5 and a final sparsity target of 0.9, this sparsity pruning tries to set 90% of the model weight to zero, thus eliminating particular links from the model.

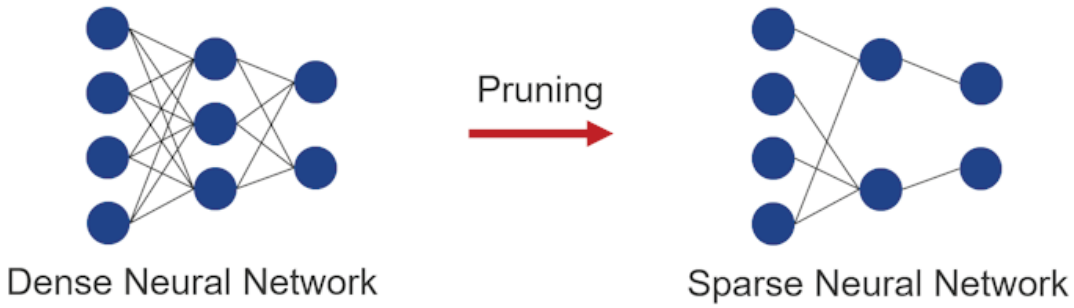

Fig. 3.8: Sparsity Pruning

The model was also subject to pruning while maintaining the mean absolute error and the mean squared error. This approach allows for the reduction of the number of weights. That is, removing as much weight in the model without altering the overall accuracy of the model.

The DCNN model as well as the ViT model with encoder and decoder were subjected to both pruning processes using the tenser-flows "tensorflow model optimization" library. Once the pruning fit process was completed with 10 epochs, the original weights were restored with the sparse weight using the "strip pruning" method of the above library.

3.6.3 Model Quantization

Quantization is a technique to reduce the computational and memory costs of running inference by representing weights and activations with low-precision data types such as an 8-bit integer (int8) instead of the usual 32-bit floating point (float32). Quantization facilitates the execution of the model in an embedded system with no floating point unit; this allows the model to be executed even in a very constrained environment. Integer calculations are faster compared to floating-point calculations. This allows for faster, near-real-time inference, which is a goal of this project. The Fig 3.9 displays how the floating point in the model is approximated to the corresponding integer wights.

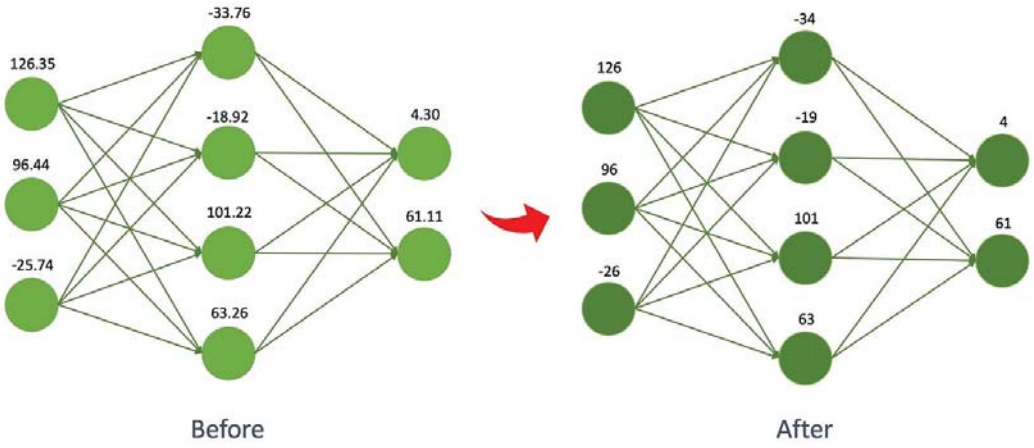

Fig. 3.9: Quantization

I have used the post-training quantization approach for the quantization of both models. The tensorflow's "tensorflow model optimization" library was used for this optimization with "sparse categorical crossentropy", which measures the difference between the model's predicted probability distribution and the true distribution of the labels and acts as a metric for evaluating performance during the quantization process as the loss function during the post-training quantization training process. This redacted the model's weights and activations from floating-point numbers to lower-precision integers (typically 8-bit).The post-training quantization training was performed for 10 epochs to ensure adequate convergence and minimal loss of accuracy due to the quantization process.

3.7 Model Parameters

After extensive analysis of the options and alternatives, these are the major decisions that were made.

- ViT model with encoder decoder specification.
  - Input Classes : 14
  - Input Size : 180 x 60 x 3
  - Input Range : [0 1]
  - Batch Size : 32
  - Encoder Layers: 5
  - Decoder Layers : 2
  - Number of Patches : 12
  - Patch Size : 8 x 12
  - Projection Dimension : 32
  - Number of Heads : 4
  - Transformer Layers : 1
  - Learning Rate : 0.001
  - Weight Decay : 0.0001
  - MLP Size : 256 x 128
  - Training Epoch : 100
- CNN model specification
  - Input Classes : 14
  - Input Size : 180 x 60 x 3
  - CNN Layers: 4 Layers with 32 Kernels with Size 3 x 3
  - Training Epoch : 100

3.8 Building the Model

In this section, how the deep learning models were developed and trained and how the models were converted to tensorflow lite (TFLite) will be discussed.

3.8.1 Training Model

The tenserflow model was developed using google's colab development environment, and the data were stored in Google drive storage. Colab provided T4 GPU processors with 12GB of RAM free to use subject to fair usage. The colab is a jupiter notebook environment as shown in 3.10.

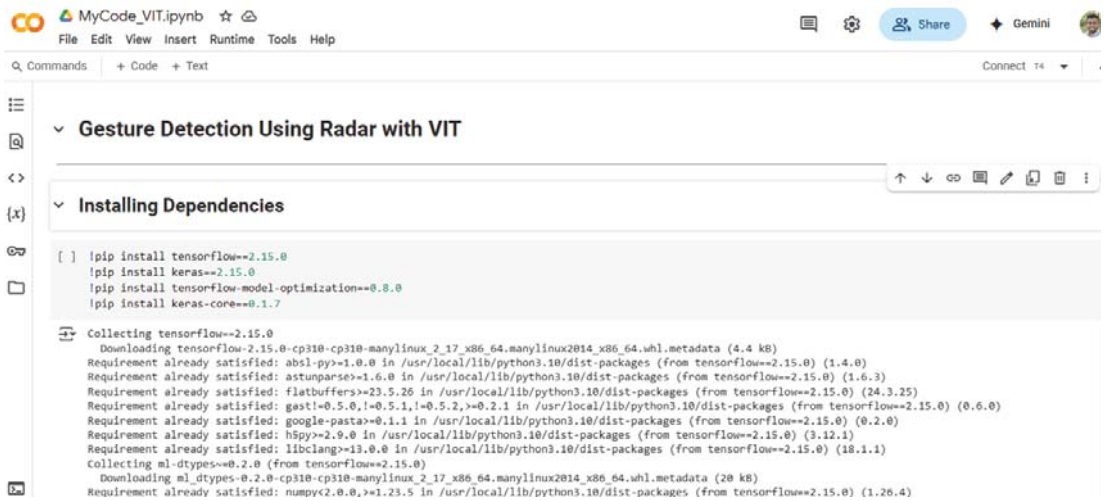

Fig. 3.10: Google Colab

The project has used the following software packages as shown in Table 3.6, The packages were chosen to avoid any conflicts during the compilation and execution.

TABLE 3.6: SOFTWARE PACKAGES

Packages	Version
Python	3.10.12
Tensorflow	2.15.0
Keras	2.15.0
tensorflow-model-optimization	0.8.0
keras-core	0.1.7

Google's colab has some restriction when using GPU for training due to its fair usage policy, in those times we had to use the CPU only mode for the training process, which is time consuming.

3.8.2 Converting Model to TFLite

TensorFlow Lite (TFLite) is a powerful open source framework, developed by google to deploy deep learning models on edge devices. TFLite models are optimized for size and speed, enabling them to run smoothly on devices with limited resources, such as an embedded system.

The models were trained, pruned, and quantized before subjecting them to tflite conversion. I used the "TFLiteConverter" function from the "Tensorflow" library for the conversion. bellow parameters are to be set for the converter to get the desired results.

- Optimization - This sets the optimization options for the converter, set to default.
- Representative data set - These set input samples that are representative of the data the model will encounter in production.
- Supported operation - This set specifies that the target device supports integer-only operations in this case.
- Inference input type - This sets the input types of the tflite model.
- Inference output type - This sets the output types of the tflite model.

The converter is fed with the above parameters and the model is converted, then the tflite model is saved to a file for further applications. Fig 3.11 illustrates this process.

Fig. 3.11: TFLite Conversion

3.9 Storing / Loading Models

The models take large amount of time to train due to the large amount of data and iteration. Its very inconvenience to rerun the training each time we need to do post-processing to the models, the storing and loading of the trained model become handy in these scenarios. There are mainly two avenues to store the models, storing the trained models as a whole and storing only the trained weights of the models.

3.9.1 Storing Complete Trained Model

Storing the trained model as a whole is very suitable for models with simple architecture. The storing and loading is straightforward, and in this approach we can store models' architecture and the associated learned wights. We haved used the keras library for this approach, and stored the developed DCNN based models for use in future processing. When it is required to subject the model to further processing or use it for prediction, We can simply load the model and do the processing; this is the main advantage in this.

3.9.2 Storing Trained Model Weights

Some deep learning models, such as our ViT-based models with encoder decoder, are complex to be stored as a whole, so we used the other approach of storing the trained weights only. I used the same keras library as in the above case. The main drawback in this approach is that we had to do additional processing to get the model loaded properly into the TensorFlow environment. This involves two stages.

1. Load the model architecture : This required loading the model with all associated functional definitions.

2. Load the stored weights : In this stage, i will load the weights onto the models reconstructed in the previous stage.

3.10 Data Prepossessing

Data preprocessing is a crucial step in deep learning. The data should be fed to the model in the proper format as required by the model. when the model is converted to tflite with integer quantization in the process of running on an embedded system, the model's input scale and zero point change. So its absolutely necessary to counter the adjustments when feeding data into those tflite models for gesture prediction.

- Zero point - Zero point is an integer value that corresponds to the floating point value 0.
- Input scale - Floating point value that determines the mapping between the original floating point input values and the quantized integer.

These attributes relevant to each model were obtained by loading the model in tflite and then invoking the get input details function. Table 3.7 shows the details obtained for the ViT model and the DCNN model.

TABLE 3.7: DATA PREPOSSESSING

	ViT Model	DCNN Model
Input shape	[1 120 170 3]	[1 60 180 3]
Input data type	int8	int8
Input scale	0.0039*	1.0
Input zero point	-128	-128

Note. * Approximate to 4th decimal place

The images are loaded and, as with the input image size, then divided by the input scale factor, then the input zero point is added. Then its rounded to the integers and flattens to an array and feed to the model as input to perform the prediction.

3.11 Embedded System

An embedded system is a specialized computing system designed to perform a specific function within a larger system. It is a combination of hardware (such as a microprocessor, memory, and input/output devices) and software tightly integrated to carry out specific tasks. These embedded systems are often limited in terms of computing, memory, and storage. Its very important to evaluate the embedded systems before choosing the ideal one for our application. Listed below are some aspects of consideration.

- Application requirements: The processing speed, memory, and input / output requirements.
- Hardware Considerations: Processing power, memory, storage, and peripherals required.
- Software Considerations: Availability of suitable compilers, debuggers, and other tools for developing and testing the software
- Cost Consideration: Balance performance and features with the overall cost of the system.

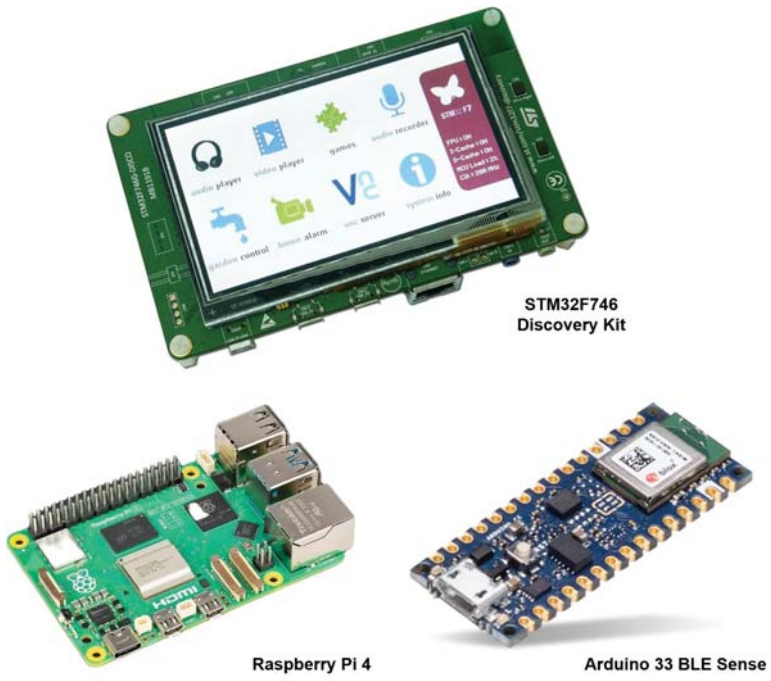

Fig. 3.12: Embedded Systems

Fig 3.12 shows the few devices that we have considered as the embedded platform.

3.11.1 Embedded System Development Environment

An embedded system development environment is a set of tools, software and hardware used to design, develop, test, and debug embedded systems. When choosing an embedded system, it is very important to analyze the set of support tools built around it. The following are the considerations.

- Integrated Development Environment : Availability of development environments like Arduino IDE, STM32CubeIDE, and etc.
- Hardware Tools: Development boards and kits for prototyping and testing.
- Libraries and Frameworks : Libraries to simplify development, such as STM32 Cube AI.

3.11.2 Embedded System Minimum Requirements

After careful analysis, a minimum set of requirements is set for the target embedded system, before finalizing the choice. following is the summary.

- Microcontroller: ARM Cortex-M4 or above
- Memory: 256KB
- Storage: 1MB of flash storage.
- I/O: Minumum 2 UART
- Power supply: 3.3V
- Availability of development board: Yes
- Application: A fairly simple IDE for development.
- Library: Availability of Library for DCNN and ViT implementation

3.11.3 Cost-Benefit Analysis and Selecting an Embedded System

A cost-benefit analysis of an embedded system involves evaluating the costs associated with designing, developing, and maintaining the system against the benefits it provides. Optimizing the cost associated with an embedded system design and lifetime is crucial for its adaptation. The cost could be either the cost of the product, the cost of development, or the running cost.

The Table 3.8 in the bottom is a quick summary of all the parameters that were considered.

TABLE 3.8: EMBEDDED SYSTEMS COMPARISON

Component	Raspberry Pi-4	STM32F746 Disco	Arduino Nano 33 BLE Sense
Processor	Cortex-A72	Cortex-M7	Cortex-M4
Clock Speed	1.5 GHz	216 MHz	64 MHz
RAM	2GB	320 KB	256 KB
Flash Memory	32GB	1 MB	1 MB
IDE/Software	Pi OS	STM32CubeIDE	Arduino IDE
Learning Curve	Moderate	Steep	Easy
DL Library's	High	High	Moderate
High-Performance	Excellent	Limited	Limited
Real-Time	Limited	Excellent	Good
Estimated Cost	100 $	50 $	40 $
Energy Consumption	Moderate	Low	Low

Note. ARM Classes : A - Application, M - Microcontroller

The project decided to go with the STM32F746 discovery board for the implementation after analyzing the collected details. This is driven primarily by the availability of a well-developed ecosystem around the STM32 chip set for the development of the deep learning model and the relatively low cost of implementation. The actual cost of implementation will be well below 15 USD if we consider developing a system around the STM32F7 chip, since the board also includes other peripherals which are not required for our application. The STM32F7 is a chip, cortex-M7 designed specifically for embedded application unlike Raspberry Pi's cortex-A72 hence consume very little power. The project hopes that the lower the total cost, the wider adoption, and the choice will be effective.

3.12 Generating C Files

The project need to convert the tflite models to executable to run them on the embedded system, this involves converting the model to a C model so that we can incorporate them into the IDE for the STM32 base microcontrollers. STM provide a set of cloud base tools which we can use to achieve this. "ST Edge AI Developer Cloud" is such an environment. I have used the tool for generating C files necessary, and it involves fairly simple steps and it also provides some optimization benchmarking support as well.

1. Upload the tflite model : Upload the tflite model.
2. Select platform : Select the platform and ST Edge AI Core Version. i used "ST Edge AI Core 2.0.0" and devise platform is set to "STM32 MCUs".
3. Quantize the model (Optional) : Quantize the model if required.
4. Quantize the model (Optional) : Optimize the model for RAM size and inference time.
5. Benchmark : Select the embedded device, run the model, and measure the inference time. The target device is set to STM32F746G-DISCO.
6. Results : View the benchmark results.
7. Generate : Download the necessary C files after setting the target board/CPU details.

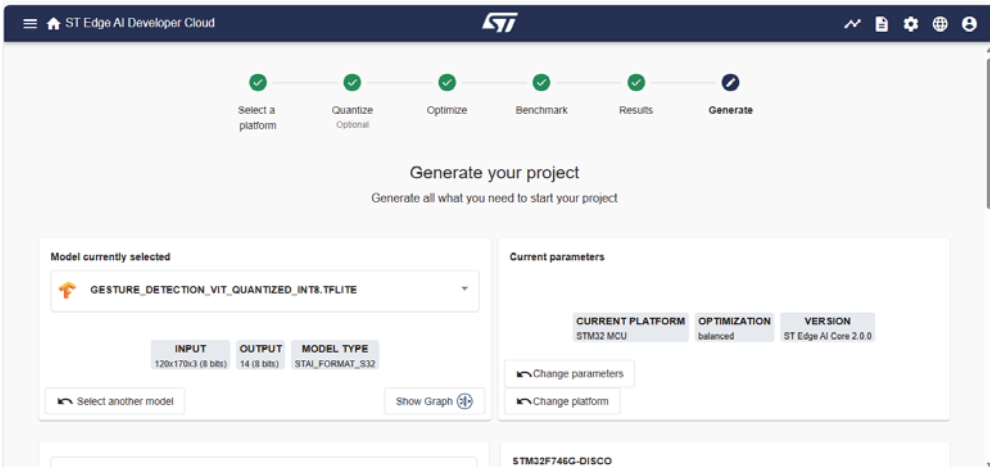

Fig. 3.13: ST Edge AI Developer Cloud

Fig 3.13 shows the interface used to generate C files from the STM Edge AI developer cloud.

3.13 Development Environment Setup

The project have used the STM32Cube development environment for the firmware development of the STM32F746NG discovery board. This is a comprehensive software ecosystem developed by STMicroelectronics to simplify and accelerate the development of applications for STM32 microcontrollers. This is eclipse and GCC toolchain, which supports debugging and flashing and allows for a seamless project configuration.

The IDE was installed with all necessary drivers and tools. i created a new project and selected the board. The Fig: 3.14 shows the project creation window.

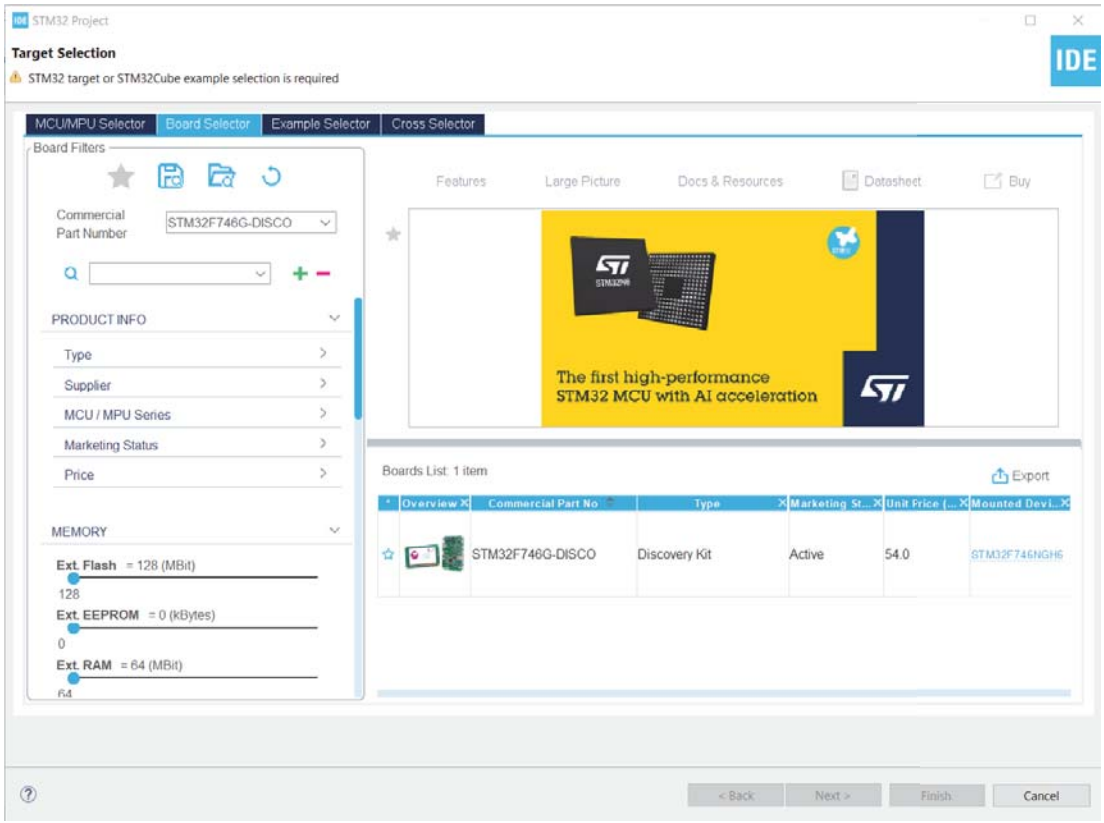

Fig. 3.14: STM32Cube-Board

I have used a few libraries for my project which are listed below.

- STM32f7xx HAL : Configuration for the STM32F746 chip.
- App x-cube-ai : Definition of AI entry functions.
- AI-Platform : Definition of APIs of the AI platform.
- Network : Contain the model and model definitions.
- STM3232746G Discovery LCD : Manged the Built-In Liquid Crystal Display

The following are the steps used to create and configure the project.

1. The software packs related to STM32 AI are installed in the project.
2. The clock speed is set to the maximum of 216 MHz.
3. Include the library necessary for basic operation of the board and the hardware abstraction layer (HAL) file is edited to enable and disable features.
4. The files generated from ST Edge AI are added and included as a library.

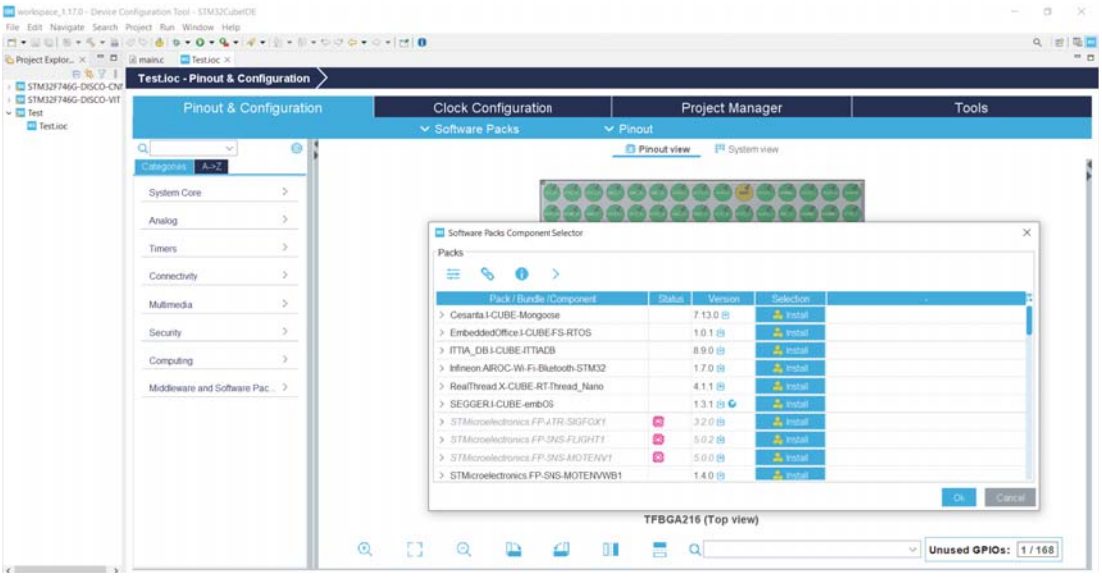

Fig. 3.15: STM32Cube-Settings

The Fig: 3.14 shows the project configuration window. I used STM32Cube IDC version 1.17.0 with the compiler "GNU Tolls for STM version 12.3".

3.14 The Embedded System Application

The project goal was to implement a simple DCNN based model and a complex ViT based model for the detection of hand gestures on radar. So obviously a set of embedded system application was required, one for each. The initial plan was to set up a standalone embedded system to run the deep learning algorithm with known image data and predict the results.

The initial work is started with the DCNN based model which is low in complexity, compared to the ViT based model. The image of the spectrum obtained from the data set and converted to the input format using the data processing discussed above and then added to the project as a header file. The project used the MX Cube AI Process library to run a performance process using the model and the data fed using the header. The process is set to run for 16 iterations and provides the model prediction as well as the average time for prediction, as well as the time for each layer in the model.

The application was optimized and prediction errors due to wrong data augmentations were identified and rectified, as well as other improvements made to the code to obtain the desired results. Once the DCNN model is confirmed to run as expected, we did the same for the ViT with the encoder decoder model.

The expectation of the project is to run the code in a real word scenario, hence the application was modified after confirming it to work as expected with static data. following are the steps.

1. The data from the radar model are obtained as an image.
2. The image is subject to preprocessing
   (a) Image is scaled to the input size
   (b) Image pixels are adjusted to the input scale (ie: Zero point and Scale)
   (c) Image pixel are cast to 8 bit integers as the model expect only integer values.
3. The data is then passed to the loaded deep learning model, and the prediction result and time are displayed using the serial terminal.
4. Prediction time measurements for each model are obtained as an average of 16 iterations of each model.

Since we were unable to use the radar set-up all the time, few adjustments were made to mimic using a GUI interface connected via the serial port to the embedded system.

3.15 Debugging the Embedded System

Debugging was a crucial step in making sure that the program runs as expected without errors. STM32Cube IDE has a rich set of debugging tools. We have used print-based debugging methods and JTAG/SWD based methods in this project. We have used Putty to visualize the output from the print base debugging.

- Print-based debug : We used print-based methods in the important flows in the application to isolate the bug, this is done by sending status text messages, via the serial port, to a connected computer. The serial data rate is set to 9600 Baud.
- JTAG based debug : We have used this when we need to explore the issue in depth. We set breakpoints, run the code line by line, and monitor the variable values in real time. STMicroelectronics ST-Link which is built into the STM32 Discovery board is used for this purpose.

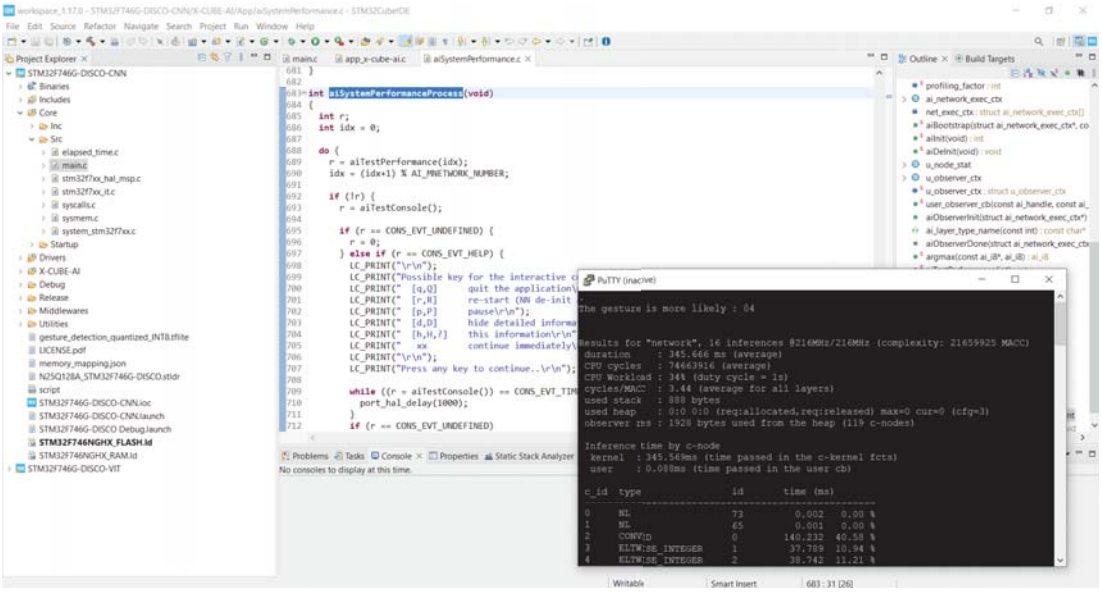

Fig. 3.16: Debugging the Embedded System

Fig 3.16 shows the setup used to debug the application program deployed on the embedded system.

3.16 Energy Measurement

Its important to have an idea of the power budget of the embedded systems to evaluate its performance and benchmark. There are few options to measure the current, voltage, and power of the board during the execution of the embedded system. "STM32 Nucleo expansion board for power consumption measurement" is the recommended option by STM which costs over 70 USD. The board offers very high accuracy of the measurements of the above parameters, but due to its cost, i chose to design a system using available components.

The architecture of the energy measurement setup is shown in Fig 3.17. The steps followed could be summarized below.

- The INA226 module used for the measuring parameters is connected to the arduino controller via the I2C interface.
- The 3.3V power out of the arduino controller is used to power the INA226 module.
- The power source positive terminal is connected to the voltage in terminal of the INA226 module, and is set to provide a constant 5V voltage and the maximum current is set to 500mA.
- The voltage out terminal of the INA226 module is connected to the voltage in terminal of the STM32F746 discovery board.
- All the ground terminals are connected together to a common rail, and the Arduino controller is connected to the laptop via the serial port.

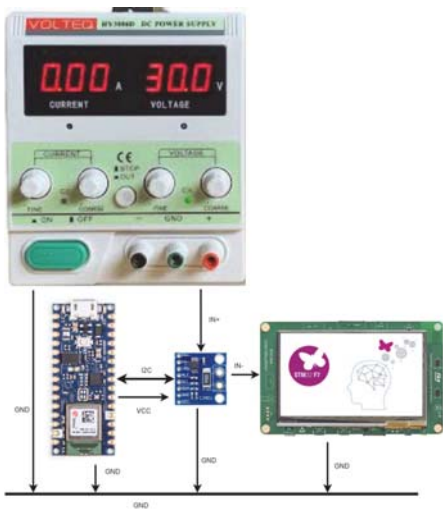

Fig. 3.17: Energy Measurement Setup

CHAPTER 4

RESULTS

Experimental results are presented in this chapter. To this end, we first present the terminology followed by the experimental results in terms of model training, prediction, and energy usage for each deep learning model concerned.

4.1 Terminology and Metrics

4.1.1 Confusion Matrix

A confusion matrix is a simple table that shows how well a classification model performs by comparing its predictions with the actual. Break down the predictions into four categories: correct predictions for both classes (true positives and true negatives) and incorrect predictions (false positives and false negatives).

- True Positive (TP): The model correctly predicted a positive outcome (the actual outcome was positive).
- True Negative (TN): The model correctly predicted a negative outcome (the actual outcome was negative).
- False Positive (FP): The model incorrectly predicted a positive outcome (the actual outcome was negative).
- False Negative (FN): The model incorrectly predicted a negative outcome (the actual outcome was positive).

4.1.2 Accuracy

This measures how often the model predictions are overall correct and gives a general idea of how well the model performs. However, accuracy can be misleading, especially with unbalanced datasets where one class dominates.

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

4.1.3 Precision

This parameter focuses on the quality of the positive predictions of the model. It tells us how many of the cases predicted as positive are actually positive. Precision is important in situations where we need to minimize false positives.

$$Precision = \frac{TP}{TP + FP}$$

4.1.4 Recall

This parameter measures how well the model identifies all the actual positive cases. Measures the proportion of true positives detected out of all the actual positive instances.

$$Recall = \frac{TP}{TP + FN}$$

4.1.5 F1-Score

This parameter combines precision and recall into a single metric to balance their trade-off. Provides a better sense of the overall performance of a model, particularly for imbalanced datasets.

$$F1 - Score = \frac{2.Precision.Recall}{Precision + Recall}$$

4.1.6 Specificity

This parameter measures the ability of a model to correctly identify negative instances. This is also known as the True Negative Rate.

$$Specificity = \frac{TN}{TN + FP}$$

4.2 Deep Learning Model Training Results

In this section, the training results of the DCNN and ViT models will be discussed. The models were trained for accuracy over 100 epochs.

4.2.1 DCNN Model

The model accuracy achieves a stable state after around 20 epochs, and then only a small improvement or almost no significant improvement is observed, as shown in the Fig 4.1.

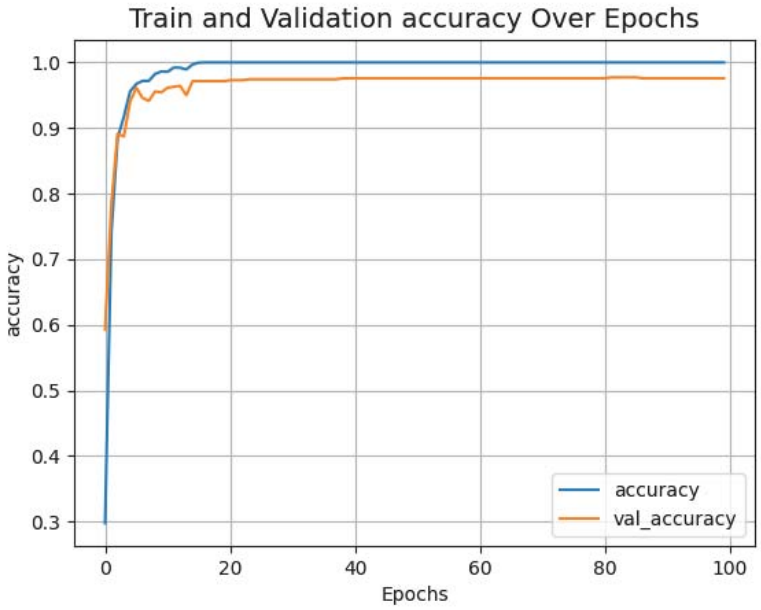

Fig. 4.1: Accuracy-DCNN

The models inference accuracy's could be summarized as follows.

- DCNN Model
  - Accuracy - 97.57 %
  - Top-5-Accuracy: 99.43 %
- DCNN Model Quantized
  - Accuracy - 94.43 %
  - Top-5-Accuracy: 99.71 %
- DCNN Model 8 Bit Integer Quantized
  - Accuracy - 94.29 %

4.2.2 ViT Model

The model accuracy achieves a stable state after around 20 epochs similar to the DCNN model, and thereafter only small improvement or almost no significant improvement is observed. One notable difference is that we can observe the convergence of training and validation accuracy's. Fig 4.2 displays changes in training and validation accuracy over each epoch.

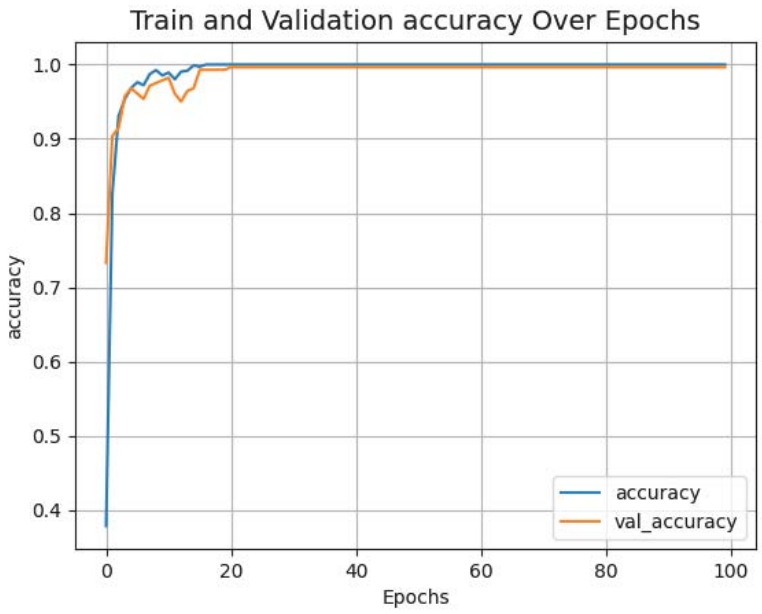

Fig. 4.2: Accuracy-ViT

The model inference accuracy could be summarized as follows.

- ViT Model
  - Accuracy - 99.96 %
  - Top-5-Accuracy: 100.00 %
- ViT Model 8 Bit Integer Quantized
  - Accuracy - 99.93 %

4.3 Deep Learning Model Quantization Results

In this section, the results of the DCNN and ViT models before and after quantization are summarized.

4.3.1 DCNN Model Size Comparison

The Table 4.1 compare the size of the DCNN model weights with its quantized variant.

TABLE 4.1: DCNN MODEL SIZE

Parameter	Float Model	8 Bit Integer Model
Total Parameters	67438	67438
Trainable Parameters	67438	67438
Non-trainable Parameters	0	0
Weights Size	263.43 KiB	66.65 KiB
Activation Size	396.96 KiB	126.56 KiB
Flash size	278.76 KiB	95.33 KiB
RAM size	400.64 KiB	103.28 KiB

4.3.2 ViT Model Size Comparison

The Table 4.2 compare the size of the ViT model weights with its quantized variant.

TABLE 4.2: VIT MODEL SIZE

Parameter	Float Model	8 Bit Integer Model
Total Parameters	133449	133449
Trainable Parameters	133201	133201
Non-trainable Parameters	248	248
Weights Size	-	132.58 KiB
Activation Size	-	92.54 KiB
Flash size	-	231.29 KiB
RAM size	-	120.49 KiB

Note. Unable to obtain deep specific parameter of the float model

4.3.3 DCNN Model with Float Weights

The confusion matrix of the DCNN model with floating point weights is shown in Fig 4.3 and the performance results are shown in Table 4.3.

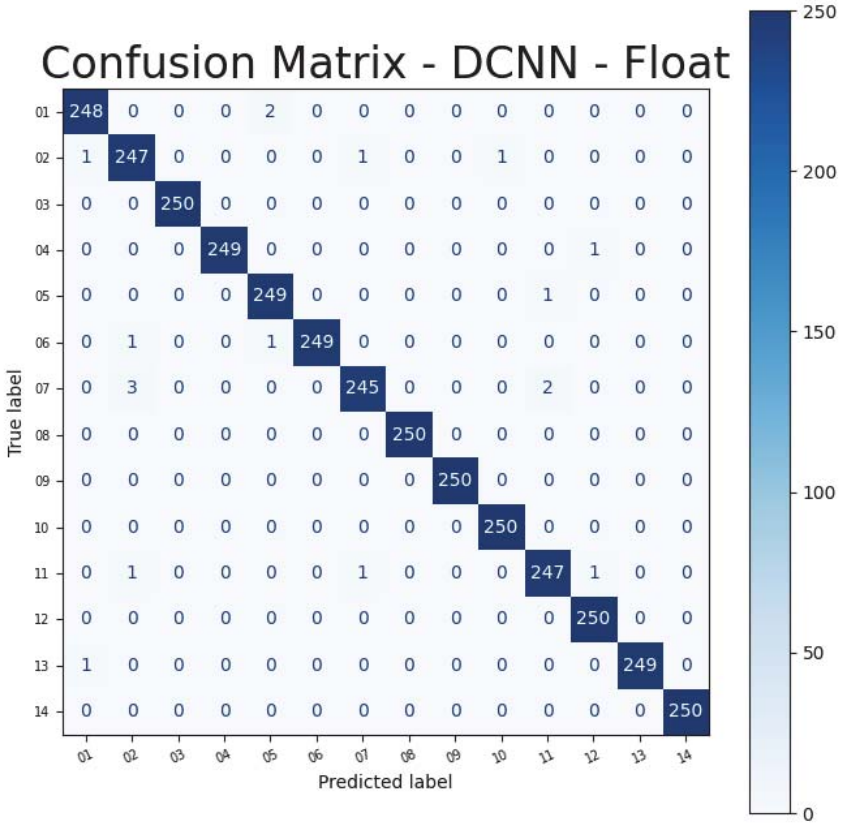

Fig. 4.3: Confusion Matrix-DCNN-Float

TABLE 4.3: DCNN MODEL WITH FLOAT WEIGHTS

Metric	Value
Accuracy	0.9949
Precision	0.9949
Recall	0.9949
F1-Score	0.9949
Specificity	0.9949

4.3.4 DCNN Model with 8 bit Integer Weights

The confusion matrix of the DCNN model with 8 bit integer weights is shown in Fig 4.4 and the performance results are shown in Table 4.4.

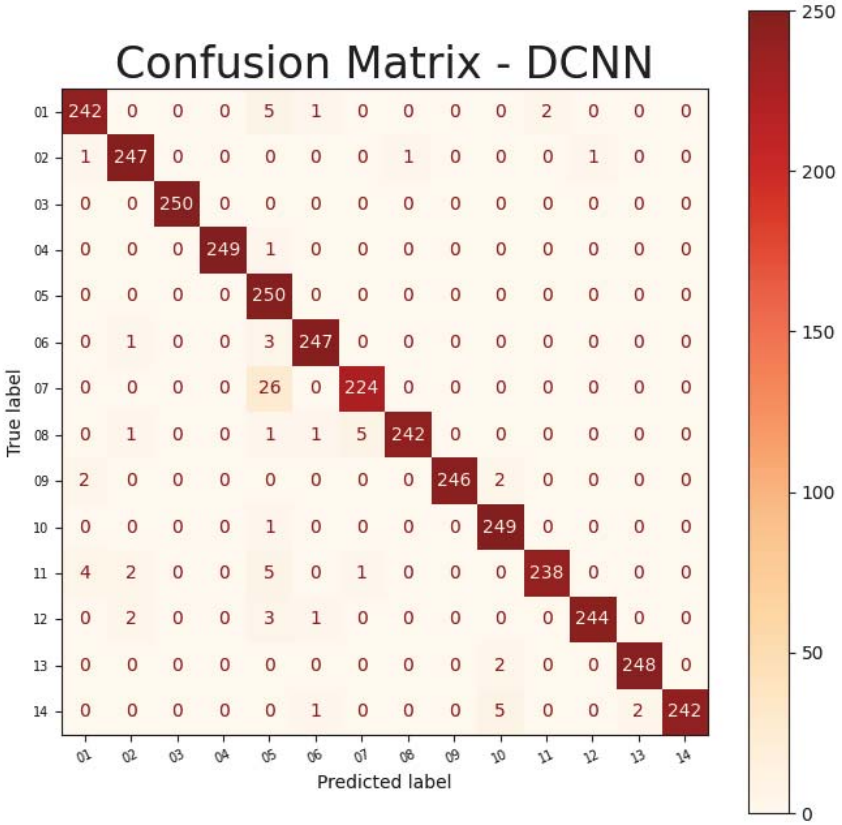

Fig. 4.4: Confusion Matrix-DCNN

TABLE 4.4: DCNN MODEL WITH 8 BIT INTEGER WEIGHTS

Metric	Value
Accuracy	0.9763
Precision	0.9782
Recall	0.9763
F1-Score	0.9766
Specificity	0.9763

4.3.5 ViT Model with Float Weights

The confusion matrix of the ViT model with floating point weights is shown in Fig 4.5 and the performance results are shown in Table 4.5.

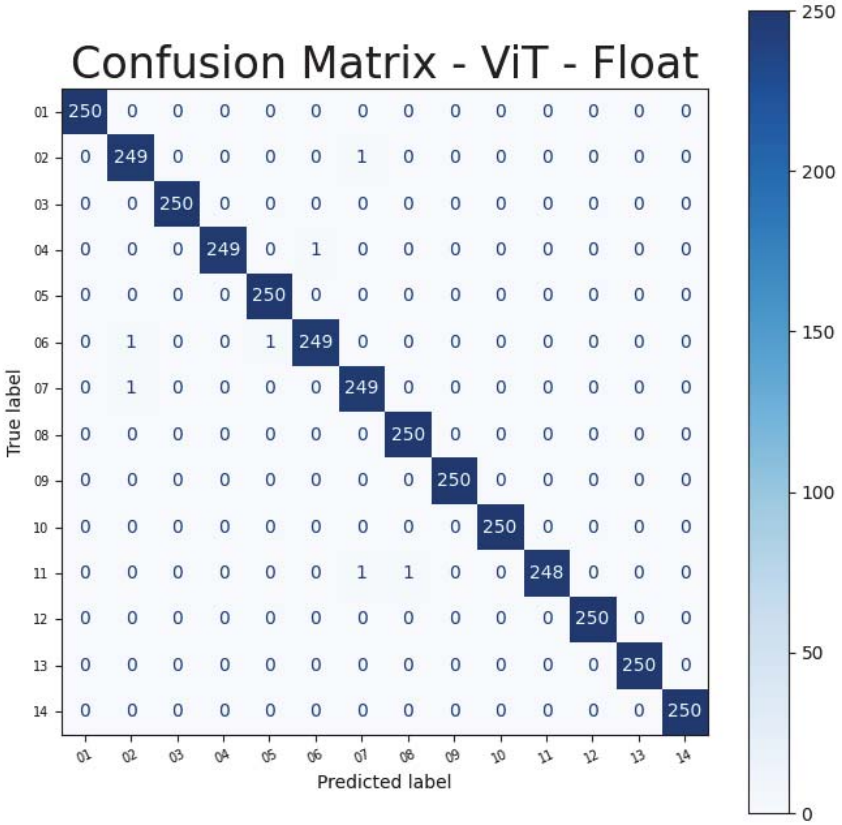

Fig. 4.5: Confusion Matrix-ViT-Float

TABLE 4.5: VIT MODEL WITH FLOAT WEIGHTS

Metric	Value
Accuracy	0.9980
Precision	0.9980
Recall	0.9980
F1-Score	0.9980
Specificity	0.9980

4.3.6 ViT Model with 8 bit Integer Weights

The confusion matrix of the ViT model with 8 bit integer weights is shown in Fig 4.6 and the performance results are shown in Table 4.6.

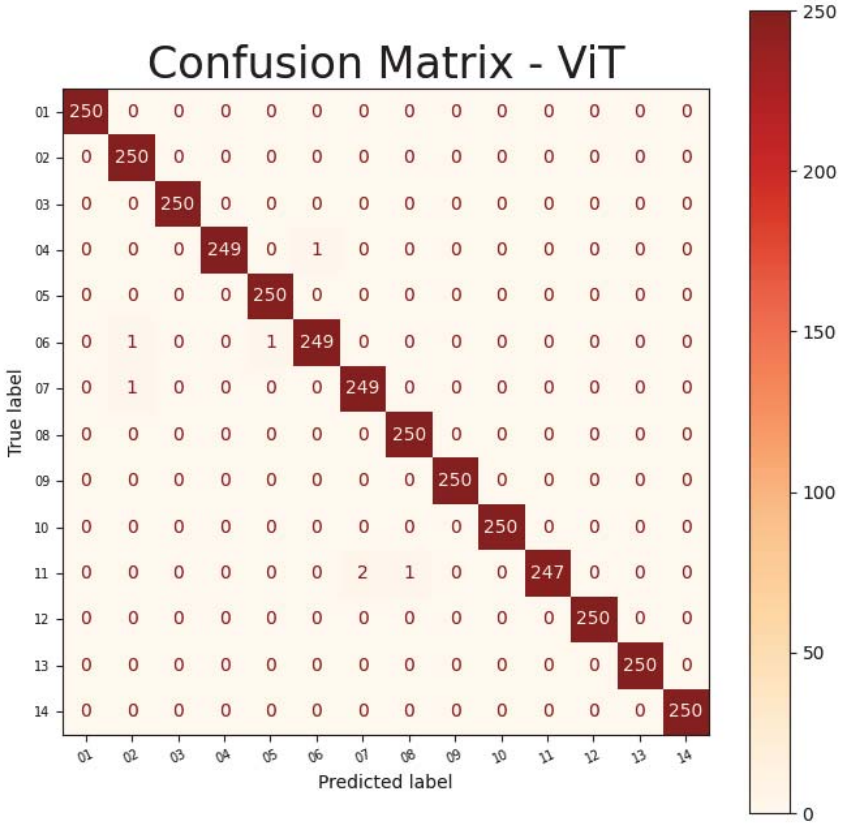

Fig. 4.6: Confusion Matrix-ViT

TABLE 4.6: VIT MODEL WITH 8 BIT INTEGER WEIGHTS

Metric	Value
Accuracy	0.9980
Precision	0.9980
Recall	0.9980
F1-Score	0.9980
Specificity	0.9980

4.4 Deep Learning Model Pruning Results

In this section, the results of DCNN models before and after pruning are summarized.

4.4.1 Sparsity of Model

The target was to achieve 80% sparsity in the weights of the DCNN model. Achieved around 78.20% sparsity over 10 epochs, which is shown in Fig 4.7.

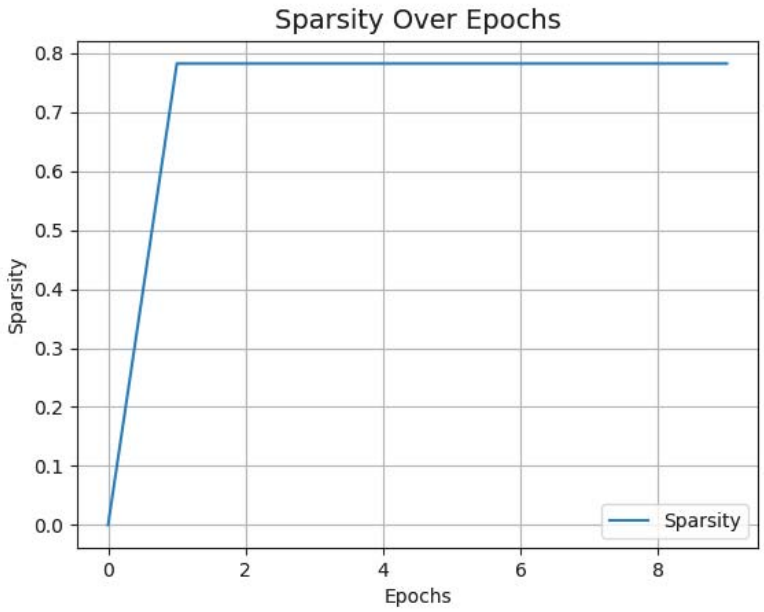

Fig. 4.7: Sparsity vs Epochs

4.4.2 Sparsity DCNN Model Size Comparison

The Table 4.7 compare the DCNN model before and after subject to pruning.

TABLE 4.7: DCNN MODEL SIZE BEFORE AND AFTER PRUNING

Parameter	Base Model	Pruned Model	Pruned & Quantized
Total Parameters	134623	67438	67438
Trainable Parameters	67438	67438	67438
Non-trainable Parameters	67185	0	0
Weights Size	263.43 KiB	263.43 KiB	66.65 KiB
Activation Size	396.96 KiB	372.61 KiB	95.36 KiB
Flash size	278.76 KiB	278.35 KiB	125.96 KiB
RAM size	400.64 KiB	376.28 KiB	98.96 KiB

4.4.3 DCNN Model Pruned with Float Weights

The confusion matrix of the DCNN model subject to pruning with floating-point weights is shown in Fig 4.8 and the performance results are shown in Table 4.8.

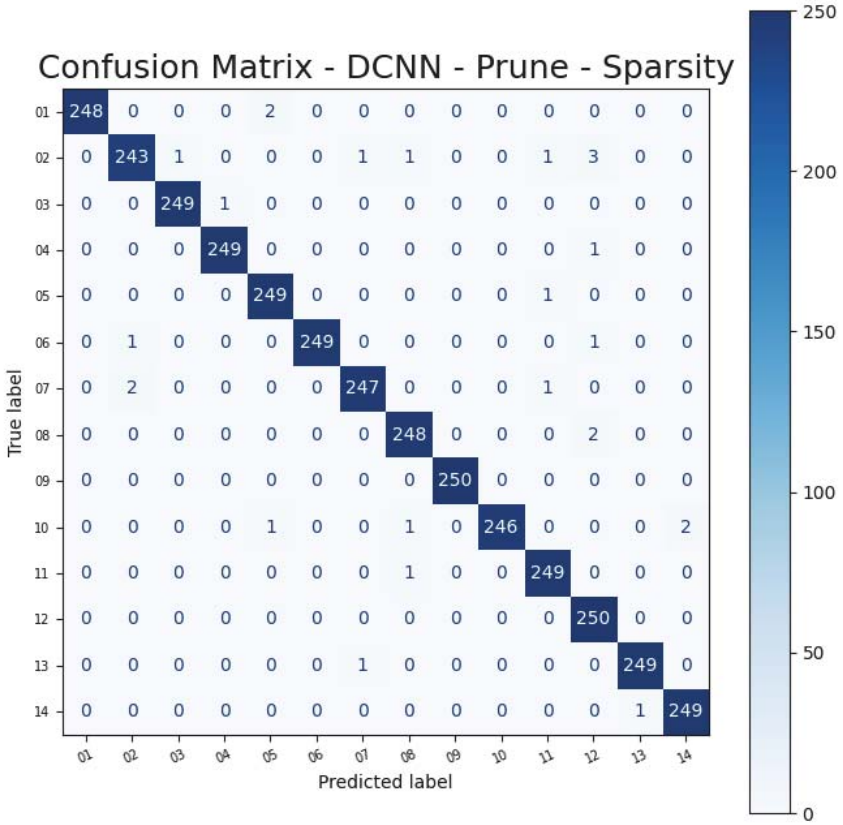

Fig. 4.8: Confusion Matrix-DCNN-Pruned

TABLE 4.8: CONFUSION MATRIX-DCNN-PRUNED

Metric	Value
Accuracy	0.9926
Precision	0.9926
Recall	0.9926
F1-Score	0.9926
Specificity	0.9926

4.4.4 DCNN Model Pruned with 8 bit Integer Weights

The confusion matrix of the DCNN model subject to pruning with 8 bit integer weights is shown in Fig 4.9 and the performance results are shown in Table 4.9.

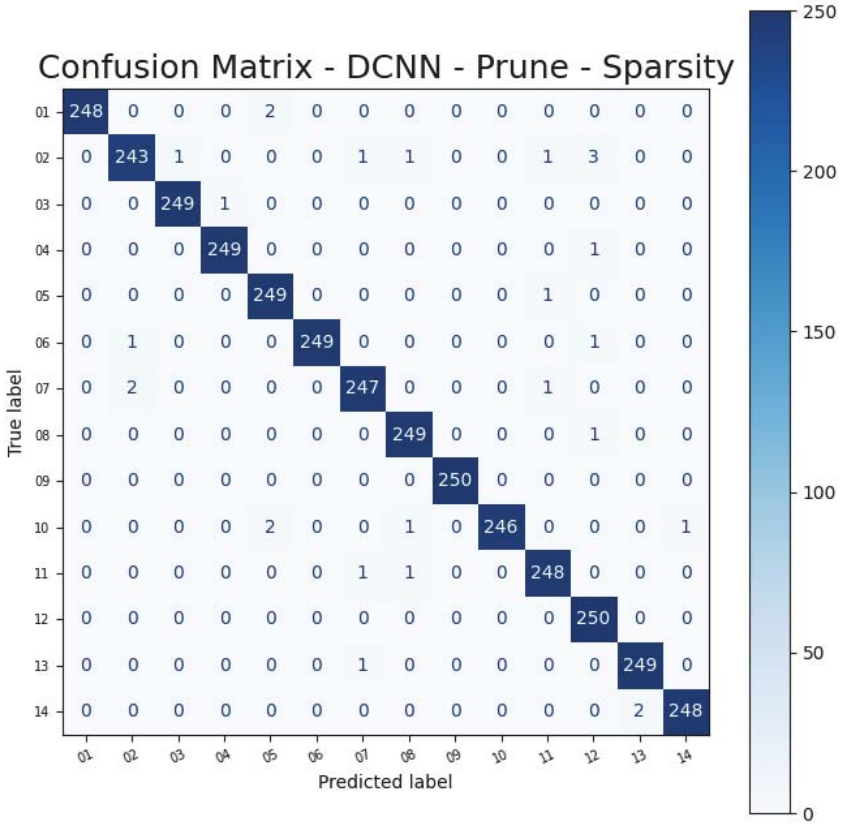

Fig. 4.9: Confusion Matrix-DCNN-Pruned-Quantized

TABLE 4.9: CONFUSION MATRIX-DCNN-PRUNED-QUANTIZED

Metric	Value
Accuracy	0.9923
Precision	0.9923
Recall	0.9923
F1-Score	0.9923
Specificity	0.9923

4.5 Deep Learning Model Inference Time

In this section, the inference time results of the DCNN and ViT models are disused.

4.5.1 Emulation Results

The project used the ST Edge AI developer cloud emulation environment. The Fig 4.10 below is a comparison between the DCNN model and its quantized model and the quantized model.

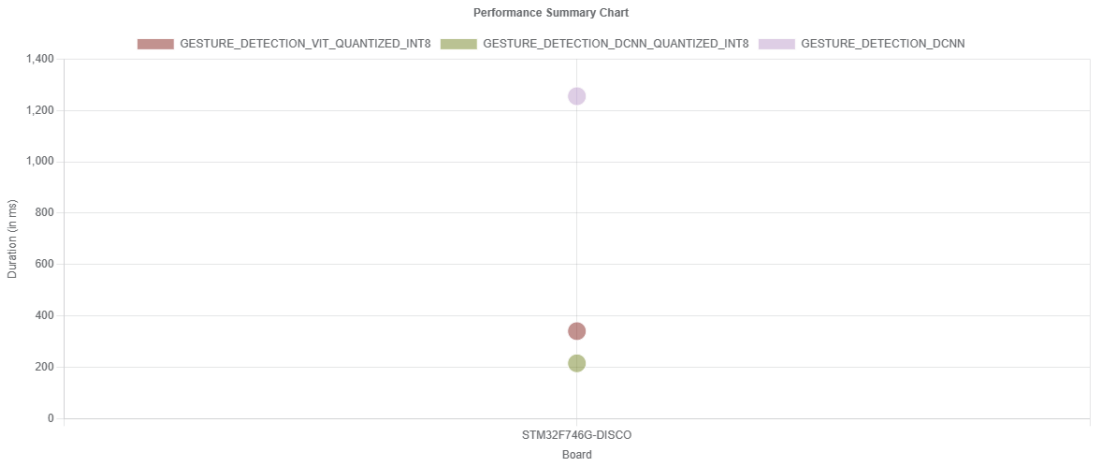

Fig. 4.10: Performance Summary Chart

In Fig 4.10 we can observe a significant deference between the inference time of the non-quantized model and the 8-bit integer quantized DCNN models. We also can observe that there is only small difference between the inference time of the DCNN model and the ViT model, The DCNN model is relatively lower in complexity than ViT, therefore a higher inference speed.

4.5.2 Implementation Results

In the implementation of embedded systems, we only used the 8 bit quantized models of each DCNN and ViT-based model. The inference time calculations for each of these models were calculated as an average of 16 inferences.

4.5.2.1 DCNN Model

The DCNN model took around 213.111 ms per inference as an average over 16 inferences. The Fig 4.11 is the sample output obtained via the serial port.

```
Results for "network", 16 inferences @216MHz/216MHz (complexity: 36099022 MACC)
 duration     : 213.111 ms (average)
 CPU cycles   : 46032152 (average)
 CPU Workload : 21% (duty cycle = 1s)
 cycles/MACC  : 1.27 (average for all layers)
 used stack   : 1088 bytes
 used heap    : 0:0 0:0 (req:allocated,req:released) max=0 cur=0 (cfg=3)
 observer res : 136 bytes used from the heap (7 c-nodes)

 Inference time by c-node
  kernel  : 213.099ms (time passed in the c-kernel fcts)
  user    : 0.005ms (time passed in the user cb)

 c_id  type                id       time (ms)
 -------------------------------------------------
 0     OPTIMIZED_CONV2D    1        114.533  53.75 %
 1     OPTIMIZED_CONV2D    3         81.423  38.21 %
 2     OPTIMIZED_CONV2D    5         15.375   7.22 %
 3     OPTIMIZED_CONV2D    7          1.510   0.71 %
 4     DENSE               9          0.229   0.11 %
 5     NL                  9          0.005   0.00 %
 6     DENSE               10         0.020   0.01 %
 -------------------------------------------------
                                    213.099 ms
```

Fig. 4.11: DCNN Inference Time

4.5.2.2 ViT Model with Encode and Decoder

The ViT model with the encoder and decoder architecture took approximately 345.690 ms per inference as an average of 16 inferences. The Fig 4.12 is the sample output obtained via the serial port.

```
Results for "network", 16 inferences @216MHz/216MHz (complexity: 21659925 MACC)
 duration     : 345.690 ms (average)
 CPU cycles   : 74669231 (average)
 CPU Workload : 34% (duty cycle = 1s)
 cycles/MACC  : 3.44 (average for all layers)
 used stack   : 888 bytes
 used heap    : 0:0 0:0 (req:allocated,req:released) max=0 cur=0 (cfg=3)
 observer res : 1928 bytes used from the heap (119 c-nodes)

 Inference time by c-node
  kernel  : 345.593ms (time passed in the c-kernel fcts)
  user    : 0.088ms (time passed in the user cb)

 c_id  type                id       time (ms)
 -------------------------------------------------
 0     NL                  73         0.002   0.00 %
 1     NL                  65         0.001   0.00 %
 2     CONV2D              0        140.250  40.58 %
 3     ELTWISE_INTEGER     1         37.792  10.94 %
 4     ELTWISE_INTEGER     2         38.743  11.21 %
 5     POOL                3          5.390   1.56 %
 6     CONV2D              4         26.203   7.58 %
 7     ELTWISE_INTEGER     5         17.367   5.03 %
 8     ELTWISE_INTEGER     6         18.388   5.32 %
 9     POOL                7          2.535   0.73 %
 10    CONV2D              8          8.610   2.49 %
 11    ELTWISE_INTEGER     9          8.356   2.42 %
 12    ELTWISE_INTEGER     10         9.052   2.62 %
 13    POOL                11         1.246   0.36 %
 14    PAD                 12         0.020   0.01 %
 15    CONV2D              12         5.494   1.59 %
 16    ELTWISE_INTEGER     13         4.146   1.20 %
 17    ELTWISE_INTEGER     14         4.570   1.32 %
 18    POOL                15         0.625   0.18 %
 19    PAD                 16         0.013   0.00 %
 20    CONV2D              16         5.979   1.73 %
 21    ELTWISE_INTEGER     17         2.226   0.64 %
 22    ELTWISE_INTEGER     18         2.421   0.70 %
 23    POOL                19         0.337   0.10 %
|
|
 117   NL                  129        0.005   0.00 %
 118   DENSE               130        0.020   0.01 %
 -------------------------------------------------
                                    345.593 ms
```

Fig. 4.12: ViT Inference Time

4.6 Energy Measurement Results

In this section, the results of the energy usage measurement of the DCNN and ViT models are disused.

4.6.1 DCNN Model Energy Consumption

The Fig 4.13 shows the power consumption of the DCNN model over a period of one second. Data was collected on the voltage, current, and power usage of the embedded system using the setup discussed above.

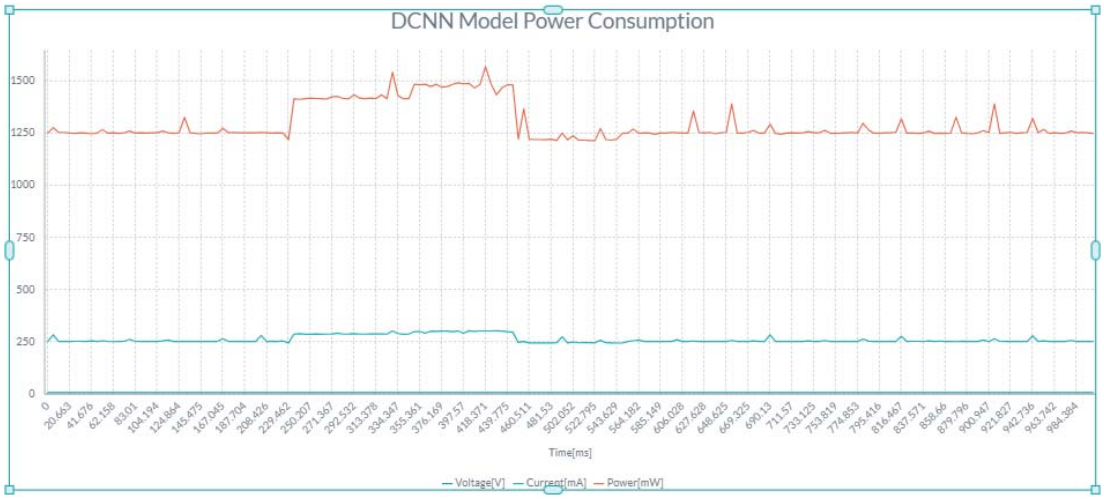

Fig. 4.13: DCNN Model Energy Consumption

When we compare the Fig 4.11 for DCNN model inference time side by side with Fig 4.13 for DCNN model power consumption we can observe the convolutions are the most energy intensive operations and the time taken corresponds to the duration of peak power usage. The following is a summary of the data collected for the DCNN model.

TABLE 4.10: DCNN MODEL ENERGY CONSUMPTION

Parameter	Min Value	Max Value
Voltage	4.93 V	5.02 V
Current	242.84 mA	301.47 mA
Power	1212.01 mW	1566.09 mW

The duration of peak power usage was 220.787 ms, and energy usage was calculated to be around 1301.6097 mJ for a duration of one second, which is for the entire board. The embedded system is calculated to use around 51.8635 mJ of energy for the execution of the model.

4.6.2 ViT Model with Encode and Decoder Energy Consumption

The bellow Fig 4.14 shows the power consumption of the ViT model over a period of one second. There is noticeable deference in current and power usage due to peak usage spreading over a longer duration than the DCNN model.

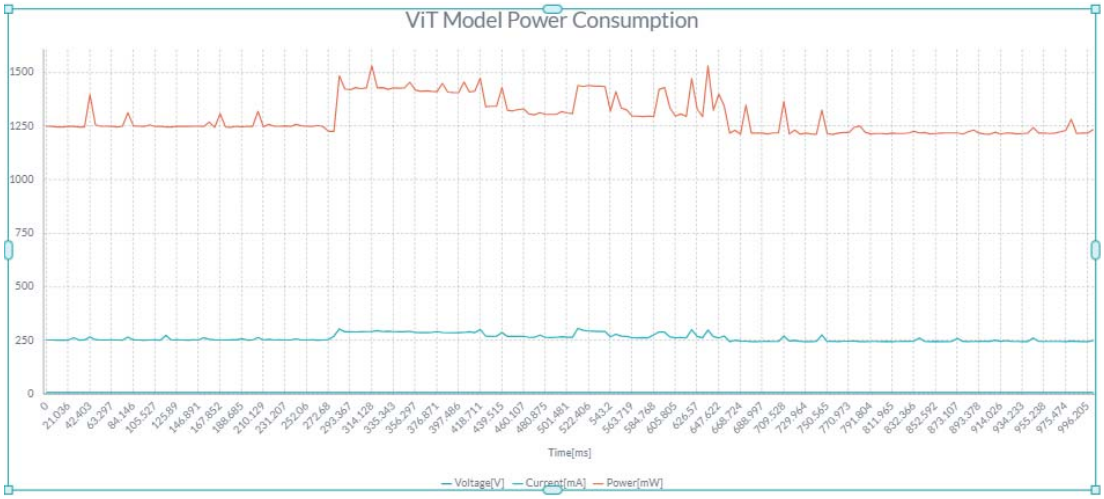

Fig. 4.14: ViT Model with Encode and Decoder Energy Consumption

When we compared Fig 4.12 for ViT inference time side by side with Fig 4.14 for the ViT model power consumption, we can observe that the power consumption spikes correspond to convolution operations and are spaced, unlike the DCNN model. Several such energy-intensive operations in the ViT model with encoder and decoder are spread over a longer time duration, and there are some low power-intensive operations in between them as well. This explains the difference in the energy consumption pattern of the DCNN model and the ViT model with encoder and decoder. The following is a summary of the data collected for the ViT model with encoder and decoder.

TABLE 4.11: VIT MODEL WITH ENCODE AND DECODER ENERGY CONSUMPTION

Parameter	Min Value	Max Value
Voltage	4.93 V	5.01 V
Current	242.72 mA	304.65 mA
Power	1209.03 mW	1529.39 mW

The peak power usage duration lasted for 317.33 ms, and energy usage was calculated to be around 1298.8153 mJ for the duration of one second, this is for the whole board. The embedded system is calculated to use around 56.3499 mJ of energy for the execution of the model.

4.7 Discussion

The results and impact of the project and also the possible applications will be briefly discussed.

4.7.1 Deep Learning Model

Deep learning model is key to detect hand gestures from Doppler radar in an automated manner. The model could be a simple deep convolutional neural network model or a complex model that was developed such as a ViT with encoder and decoder. The DCNN model is fairly simple to implement in an embedded system and has a fairly good prediction accuracy, the DCNN model which we come up with has a prediction accuracy of 97.57% and the DCNN model implemented in embedded systems has a precision around 94.29%. On the other hand, the model-based ViT, which is complex and difficult to implement, has an accuracy of 99.93% in the embedded system.

4.7.2 Deep Learning Model Hyperparameter Tuning

We had subjected the ViT model with encoder and decoder to hyperparameter tuning. The number of transformer layers was reduced to 1 from 3 and the MLP size was reduced by a factor of 4 and due to these changes the model parameters and size shrink by a factor more than 6 times, and yet the model accuracy rose to 99.96% from 98.3%.

4.7.3 Deep Learning Model Pruning

In the project we had used pruning to eliminate the connections in the model that have a lower impact on the overall accuracy of the model prediction. The initial target was to achieve the sparsity 80% of the model weights. We were able to achieve an overall sparsity of the model of 78.20% with a negligible change in the accuracy of model prediction.

4.7.4 Deep Learning Model Quantization

Model quantization was used to convert to float weight to 8 bit integer weights, which allowed for general shrinking of model size and greatly increased model inference time. The project was able to achieve a 400% reduction in the RAM size used by the DCNN model with a 600% increase in inference speed but a cost of 5% reduction in prediction accuracy. Through quantization, the project was able to comfortably fit the ViT model in the RAM.

4.7.5 Model Inference Time

The gesture samples were collected over a period of 3 seconds; therefore, for real-time gesture detection, prediction was required within this time frame. We were able to achieve an inference time of 213 ms for the DCNN model and 346 ms for the ViT model with the embedded system deployment.

4.7.6 Model Power Consumption

The model power consumption is a crucial factor when we deploy in constrained environments such as embedded systems. Using the energy measurement setup, we were able to calculate the energy usage per inference as 51.86 mJ for the DCNN model and 56.35 mJ for the ViT model with encoder and decoder.

4.7.7 Impact of the Project

Implementing deep learning models in embedded system is a great step towards a wider adoption of deep learning in everyday applications. Implementing a complex model like that of a ViT model with encoder and decoder in an embedded system like a mid-range ARM Cortex F7 is new, and this provides a great cost advantage when going for wider adoption. The project was able to design an embedded system which is cost-effective with very high accuracy to detect human gestures using a 24 GHz Doppler radar in real time. We believe that this project can be continued in the coming years as well, and this has many interesting applications.

4.7.8 Possible Applications

There are many applications for the real-time detection of hand gestures using an embedded system in wide-ranging domains. One use case is with consumer electronic devices, such as TVs, smart speakers, gaming and virtual reality applications could use the system to allow users to interact with virtual environments using natural hand movements and create immersive experiences. In healthcare, the system could help patients with limited mobility by providing hands-free control of medical devices or allowing them to communicate through sign language recognition. The system could be used to overcome the spread of pathogens, such as in a clinical environment or in public spaces, such as in times of outbreaks like COVID-19.

CHAPTER 5

CONCLUSION AND FUTURE WORKS

In this project, I was able to successfully implement and demonstrate the feasibility of real-time hand gesture recognition using Doppler radar spectral signatures on an embedded system, using a DCNN-based model and a ViT-based model with an encoder and decoder as my models, trained and validated using the publicly available 14 hand gesture spectral data. Both models achieved promising classification accuracies in the publicly available dataset, highlighting the potential of Doppler radar for real-time human-machine interaction in embedded environments.

The results of the study show that the DCNN model is potentially less complex and offers a balance between accuracy and computational efficiency for embedded deployment. The ViT-based model, though potentially more computationally intensive, offers the potential to capture more nuanced gesture features and offers a high rate of classification accuracy compared to the DCNN model. This study highlights the importance of careful model selection and optimization for embedded applications and provides a foundation for further advancements in radar-based gesture recognition.

The project was done with static data for model training and validation; therefore, future iterations may consider using real-time data captured from the radar module. There is also the possibility of integrating the system into an IoT use case, which is to provide better user value.

REFERENCES

[1] S. Skaria, A. Al-Hourani, M. Lech, and R. J. Evans, "Hand-gesture recognition using two-antenna doppler radar with deep convolutional neural networks," *IEEE Sensors Journal*, vol. 19, no. 8, pp. 3041–3048, 2019.

[2] K. Kehelella, G. Leelarathne, D. Marasinghe, N. Kariyawasam, V. Ariyarathna, A. Madanayake, R. Rodrigo, and C. U. S. Edussooriya, "Vision transformer with convolutional encoder–decoder for hand gesture recognition using 24-GHz doppler radar," *IEEE Sensors Letters*, vol. 6, no. 10, pp. 1–4, 2022.

[3] I. Nirmal, A. Khamis, M. Hassan, W. Hu, and X. Zhu, "Deep learning for radio-based human sensing: Recent advances and future directions," *IEEE Communications Surveys & Tutorials*, vol. 23, no. 2, pp. 995–1019, 2021.

[4] B. Fu, N. Damer, F. Kirchbuchner, and A. Kuijper, "Sensing technology for human activity recognition: A comprehensive survey," *IEEE Access*, vol. 8, pp. 83 791–83 820, 2020.

[5] Z. Sun, J. Liu, Q. Ke, H. Rahmani, M. Bennamoun, and G. Wang, "Human action recognition from various data modalities: A review," *CoRR*, vol. abs/2012.11866, 2020. [Online]. Available: https://arxiv.org/abs/2012.11866

[6] S. Ahmed, K. D. Kallu, S. Ahmed, and S. H. Cho, "Hand gestures recognition using radar sensors for human-computer-interaction: A review," *Remote. Sens.*, vol. 13, p. 527, 2021. [Online]. Available: https://api.semanticscholar.org/CorpusID:231991365

[7] V. Chen, F. Li, S.-S. Ho, and H. Wechsler, "Micro-doppler effect in radar: phenomenon, model, and simulation study," *IEEE Transactions on Aerospace and Electronic Systems*, vol. 42, no. 1, pp. 2–21, 2006.

[8] X. Li, Y. He, and X. Jing, "A survey of deep learning-based human activity recognition in radar," *Remote sensing*, vol. 11, no. 9, p. 1068, 2019.

[9] S.-W. Kang, M.-H. Jang, and S. Lee, "Identification of human motion using radar sensor in an indoor environment," *Sensors*, vol. 21, no. 7, 2021. [Online]. Available: https://www.mdpi.com/1424-8220/21/7/2305

[10] S. Hazra and A. Santra, "Radar gesture recognition system in presence of interference using self-attention neural network," in *2019 18th IEEE International Conference On Machine Learning And Applications (ICMLA)*, 2019, pp. 1409–1414.

[11] J. S. Suh, S. Ryu, B. Han, J. Choi, J.-H. Kim, and S. Hong, "24 GHz FMCW radar system for real-time hand gesture recognition using lstm," in *2018 Asia-Pacific Microwave Conference (APMC)*, 2018, pp. 860–862.

[12] C. Liu, Y. Li, D. Ao, and H. Tian, "Spectrum-based hand gesture recognition using millimeter-wave radar parameter measurements," *IEEE Access*, vol. 7, pp. 79 147–79 158, 2019.

[13] Y. Kim and B. Toomajian, "Hand gesture recognition using micro-doppler signatures with convolutional neural network," *IEEE Access*, vol. 4, pp. 7125–7130, 2016.

[14] Y. Sang, L. Shi, and Y. Liu, "Micro hand gesture recognition system using ultrasonic active sensing," *IEEE Access*, vol. 6, pp. 49 339–49 347, 2018.

[15] R. Zhao, X. Ma, X. Liu, and F. Li, "Continuous human motion recognition using micro-doppler signatures in the scenario with micro motion interference," *IEEE Sensors Journal*, vol. 21, no. 4, pp. 5022–5034, 2021.

[16] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," *CoRR*, vol. abs/1706.03762, 2017. [Online]. Available: http://arxiv.org/abs/1706.03762

[17] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, J. Uszkoreit, and N. Houlsby, "An image is worth 16x16 words: Transformers for image recognition at scale," *CoRR*, vol. abs/2010.11929, 2020. [Online]. Available: https://arxiv.org/abs/2010.11929

[18] H. Fan, B. Xiong, K. Mangalam, Y. Li, Z. Yan, J. Malik, and C. Feichtenhofer, "Multiscale vision transformers," 2021. [Online]. Available: https://arxiv.org/abs/2104.11227

[19] V. Gonzalez-Huitron, J. A. León-Borges, A. Rodriguez-Mata, L. E. Amabilis-Sosa, B. Ramírez-Pereda, and H. Rodriguez, "Disease detection in tomato leaves via cnn with lightweight architectures implemented in raspberry pi 4," *Computers and Electronics in Agriculture*, vol. 181, p. 105951, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0168169920331562

[20] Y. Liang, Z. Wang, X. Xu, Y. Tang, J. Zhou, and J. Lu, "Mcuformer: Deploying vision transformers on microcontrollers with limited memory," 2023. [Online]. Available: https://arxiv.org/abs/2310.16898

[21] Z. Liu, Y. Wang, K. Han, S. Ma, and W. Gao, "Post-training quantization for vision transformer," 2021. [Online]. Available: https://arxiv.org/abs/2106.14156

[22] L. Ben Letaifa and J.-L. Rouas, "Fine-grained analysis of the transformer model for efficient pruning," in *2022 21st IEEE International Conference on Machine Learning and Applications (ICMLA)*. Nassau, Bahamas: IEEE, Dec. 2022, pp. 897–902. [Online]. Available: https://hal.science/hal-04047338

[23] C. Gamanayake, L. Jayasinghe, B. K. K. Ng, and C. Yuen, "Cluster pruning: An efficient filter pruning method for edge ai vision applications," *IEEE Journal of Selected Topics in Signal Processing*, vol. 14, no. 4, pp. 802–816, 2020.

[24] M. Nagel, M. Fournarakis, R. A. Amjad, Y. Bondarenko, M. van Baalen, and T. Blankevoort, "A white paper on neural network quantization," 2021. [Online]. Available: https://arxiv.org/abs/2106.08295

APPENDIX A

EMBEDDED SYSTEM COMPARISON

I came across a few implementations of the deep learning algorithm in embedded systems during the initial review of the literature. After an initial analysis, i came across 3 classes of embedded systems.

1. Small-Scale Embedded Systems : A single 8-bit or 16-bit micro-controller, very limited resources in terms of processing power, memory, and peripherals.Eg AVR or PIC Micro controller.

2. Medium-Scale Embedded Systems : A more powerful 16-bit or 32-bit micro-controllers or microprocessors, can handle more complex tasks and may involve some level of real-time operation. Eg Arduino 33 BLE, STM32F746 ane etc.

3. Large-Scale or Sophisticated Embedded Systems : Complex embedded systems, often based on 32-bit or 64-bit microprocessors, have significant processing power, large amounts of memory, and advanced peripherals, and often run real-time operating systems. For example, Raspberry Pi 4, Banana Pi.

I have shortlisted three embedded system environments for further analysis. The project will be discussed below in detail in later parts.

- Raspberry Pi 4
- STM32F7 Discovery
- Arduino Nano 33 BLE Sense

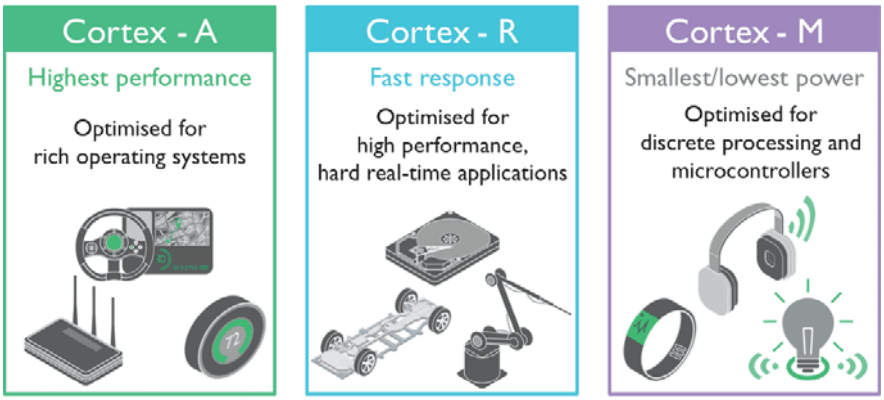

Fig. A.1: ARM Processor Class
Source: https://gsasindia.com/wp-content/uploads/2020/09/Cortex-A-Cortex-R-Cortex-M.png

APPENDIX B

ENERGY MEASUREMENT : SELECTED COMPONENTS

B.1 Power Monitor IC

I used the INA226 module to measure the voltage, current and power usage of the STM32F746G-DISCO during the execution of the model. The INA226 is a precise low-side current shunt monitor with an I2C interface, manufactured by Texas Instruments and could be used to measure current, voltage, and power in electronic circuits. This could measure voltages from 0V to 36V with a resolution of 1.25mV. This could measure current with a resolution around 1.25mA. The range and resolution of the voltage and current are adjustable depending on the use.

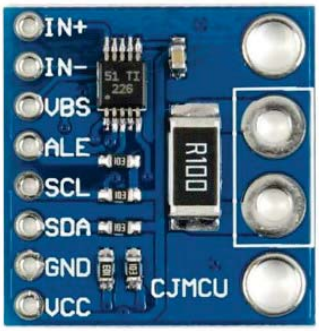

Fig. B.1: Power Monitor IC - INA226

B.2 Power Monitor Controller

I have used the Arduino Nano 33 BLE Sense as my controller to control the power monitoring IC. The device has the I2C interface required to control the power monitor module. The module has a few UART interfaces which could be used to transmit the data to a computer for further processing and visualization.

Fig. B.2: Power Monitor Controller - Arduino Nano 33 BLE Sense

B.3 Power Source

I used the power supply unit as the source. The power supply unit has short-circuit protection. The voltage level and maximum current levels could depend on the requirement.

Fig. B.3: Power Supply

APPENDIX C

DCNN MODEL

```
#Define Image Height, Width and Batch Size to load
img_height = 60
img_width = 180
batch_size = 32

#Load Images for training 80%
train_ds = tf.keras.utils.image_dataset_from_directory(
  data_dir,
  validation_split=0.2,
  subset="training",
  seed=123,
  image_size=(img_height, img_width),
  batch_size=batch_size
)

#Load Images for validation 20%
val_ds = tf.keras.utils.image_dataset_from_directory(
  data_dir,
  validation_split=0.2,
  subset="validation",
  seed=123,
  image_size=(img_height, img_width),
  batch_size=batch_size
)

#Define the model
model = keras.Sequential([
  keras.layers.Conv2D(32, 3, activation='relu'),
  keras.layers.MaxPooling2D(),
  keras.layers.Conv2D(32, 3, activation='relu'),
  keras.layers.MaxPooling2D(),
  keras.layers.Conv2D(32, 3, activation='relu'),
  keras.layers.MaxPooling2D(),
  keras.layers.Conv2D(32, 3, activation='relu'),
  keras.layers.MaxPooling2D(),
```

```
  keras.layers.Flatten(),
  keras.layers.Dense(128, activation='relu'),
  keras.layers.Dense(num_classes)
])

#Choose the tf.keras.optimizers.Adam optimizer and tf.
   ↪ keras.losses.SparseCategoricalCrossentropy loss
   ↪ function
model.compile(
  optimizer='adam',
  loss=tf.keras.losses.SparseCategoricalCrossentropy(
     ↪ from_logits=True),
  metrics = [
     keras.metrics.SparseCategoricalAccuracy(name = "
        ↪ accuracy"),
     keras.metrics.SparseTopKCategoricalAccuracy(5, name=
        ↪ "top-5-accuracy"),
  ],
)

checkpoint_filepath = "/tmp/checkpoint.weights.h5"
checkpoint_callback = keras.callbacks.ModelCheckpoint(
     checkpoint_filepath,
     monitor="val_accuracy",
     save_best_only=True,
     save_weights_only=True,
)

#Choose the optimizer and loss function
history = model.fit(
  train_ds,
  validation_data=val_ds,
  epochs=100,
  callbacks=[checkpoint_callback],
)

#View all the layers of the network using the Keras
model.summary()
```

APPENDIX D

VIT MODEL

```
#Define Image Height, Width and Batch Size to load
img_height = 120 # 60 --> 120
img_width = 170 # 180 --> 170
batch_size = 32
input_shape = (img_height, img_width, 3)

#Define model names
tflite_model_name = 'gesture_detection_VIT' # Will be
   ↪ given .tflite suffix
c_model_name = 'gesture_detection_VIT' # Will be given .h
   ↪  suffix

#Load Images for training 80%
train_ds = tf.keras.utils.image_dataset_from_directory(
  data_dir,
  validation_split=0.2,
  subset="training",
  seed=123,
  image_size=(img_height, img_width),
  batch_size=batch_size
)

#Load Images for validation 20%
val_ds = tf.keras.utils.image_dataset_from_directory(
  data_dir,
  validation_split=0.2,
  subset="validation",
  seed=123,
  image_size=(img_height, img_width),
  batch_size=batch_size
)

learning_rate = 0.001
weight_decay = 0.0001
num_epochs = 100 # 100
```

```
image_size_x = 8 # We'll resize input images to this size
   ↪  // img_height = 60 -- > 8
image_size_y = 12 # We'll resize input images to this
   ↪ size // img_width = 180 -- > 12
patch_size = 4 # Size of the patches to be extract from
   ↪ the input images
num_patches = int((image_size_x*image_size_y //
   ↪ patch_size**2)) + ((image_size_x*image_size_y %
   ↪ patch_size**2) > 0)

projection_dim = 32
num_heads = 4

transformer_units = [
   projection_dim * 2,
   projection_dim,
] # Size of the transformer layers
transformer_layers = 1
mlp_head_units = [256, 128] # Size of the dense layers of
   ↪  the final classifier [1024, 512]

data_preprocess = keras.Sequential([
      layers.Normalization(),
      layers.Resizing(image_size_x, image_size_y)
   ],
   name="data_preprocess",
)

# Unpack training images
training_dataset = train_ds.unbatch()
training_images = training_dataset.map(lambda x, y: x)
training_labels = training_dataset.map(lambda x, y: y)
training_images = np.array(list(training_images.
   ↪ as_numpy_iterator()))/255.0 # Normalize
training_labels = np.array(list(training_labels.
   ↪ as_numpy_iterator()))
```

```
# Unpack validation images
validation_dataset = train_ds.unbatch()
validation_images = validation_dataset.map(lambda x, y: x
    ↪ )
validation_labels = validation_dataset.map(lambda x, y: y
    ↪ )
validation_images = np.array(list(validation_images.
    ↪ as_numpy_iterator()))/255.0 # Normalize
validation_labels = np.array(list(validation_labels.
    ↪ as_numpy_iterator()))

# Assuming 'training_images' is your training data
normalization_layer = layers.Normalization()
normalization_layer.adapt(training_images)

# Create Patches Class
class Patches(layers.Layer):
    def __init__(self, patch_size):
        super(Patches, self).__init__()
        self.patch_size = patch_size

    def call(self, images):
        batch_size = tf.shape(images)[0]
        patches = []
        for i in range(0, images.shape[1], self.patch_size)
            ↪ :
            for j in range(0, images.shape[2], self.
                ↪ patch_size):
                patch = images[:, i:i+self.patch_size, j:j+
                    ↪ self.patch_size, :]
                patches.append(patch)
        patches = tf.stack(patches, axis=1)
        patches = tf.reshape(patches, [batch_size, -1,
            ↪ patches.shape[-1] * patches.shape[-2] *
            ↪ patches.shape[-3]])
        return patches

# Create Patches
image_size_x = image_size_x
image_size_y = 24
```

```
dim = (image_size_x, image_size_y)
resized = cv2.resize(image, dim, interpolation = cv2.
    ↪ INTER_AREA)
resized_image = tf.image.resize(
    tf.convert_to_tensor([image]), size=(image_size_x,
        ↪ image_size_y)
)
patches = Patches(patch_size)(resized_image)

# Create Patches Encoder Class
class PatchEncoder(layers.Layer):
    def __init__(self, num_patches, projection_dim):
        super(PatchEncoder, self).__init__()
        self.num_patches = num_patches
        self.projection = layers.Dense(units=projection_dim
            ↪ )
        self.position_embedding = layers.Embedding(
            input_dim=num_patches, output_dim=projection_dim
        )

    def call(self, patch):
        positions = tf.range(start=0, limit=self.
            ↪ num_patches, delta=1)
        encoded = self.projection(patch) + self.
            ↪ position_embedding(positions)
        return encoded
# Define MLP
def mlp(x, hidden_units, dropout_rate):
    for units in hidden_units:
        x = layers.Dense(units, activation=keras.
            ↪ activations.gelu)(x)
        x = layers.Dropout(dropout_rate)(x)
    return x

# Define VIT Classifier
def create_vit_classifier():
    inputs = layers.Input(shape=input_shape)
    x_1 = layers.Conv2D(4, (7, 7), activation='relu',
        ↪ padding='same')(inputs)
    x_2 = layers.BatchNormalization(axis=-1)(x_1)
```

```
x_3 = layers.MaxPool2D((2, 2), padding='same')(x_2)
x_4 = layers.Conv2D(8, (5, 5), activation='relu',
   ↪ padding='same')(x_3)
x_5 = layers.BatchNormalization(axis=-1)(x_4)
x_6 = layers.MaxPool2D((2, 2), padding='same')(x_5)
x_7 = layers.Conv2D(16, (3, 3), activation='relu',
   ↪ padding='same')(x_6)
x_8 = layers.BatchNormalization(axis=-1)(x_7)
x_9 = layers.MaxPool2D((2, 2), padding='same')(x_8)
x_10 = layers.Conv2D(32, (3, 3), activation='relu',
   ↪ padding='same')(x_9)
x_11 = layers.BatchNormalization(axis=-1)(x_10)
x_12 = layers.MaxPool2D((2, 2), padding='same')(x_11)
x_13 = layers.Conv2D(64, (3, 3), activation='relu',
   ↪ padding='same')(x_12)
x_14 = layers.BatchNormalization(axis=-1)(x_13)
x_15 = layers.MaxPool2D((2, 2), padding='same')(x_14)
x_16 = layers.Conv2DTranspose(3, (1,1), strides=(2,2))
   ↪ (x_15)
patches = Patches(patch_size)(x_16)
# Encode patches.
encoded_patches = PatchEncoder(num_patches,
   ↪ projection_dim)(patches)

# Create multiple layers of the Transformer block.
for _ in range(transformer_layers):
   # Layer normalization 1.
   x1 = layers.LayerNormalization(epsilon=1e-6)(
      ↪ encoded_patches)
   # Create a multi-head attention layer.
   attention_output = layers.MultiHeadAttention(
      num_heads=num_heads, key_dim=projection_dim,
         ↪ dropout=0.1
   )(x1, x1)
   # Skip connection 1.
   x2 = layers.Add()([attention_output,
      ↪ encoded_patches])
   # Layer normalization 2.
   x3 = layers.LayerNormalization(epsilon=1e-6)(x2)
   # MLP.
```

```
        x3 = mlp(x3, hidden_units=transformer_units,
           ↪ dropout_rate=0.1)
        # Skip connection 2.
        encoded_patches = layers.Add()([x3, x2])

    # Create a [batch_size, projection_dim] tensor.
    representation = layers.LayerNormalization(epsilon=1e
       ↪ -6)(encoded_patches)
    representation = layers.Flatten()(representation)
    representation = layers.Dropout(0.5)(representation)
    # Add MLP.
    features = mlp(representation, hidden_units=
       ↪ mlp_head_units, dropout_rate=0.5)
    # Classify outputs.
    logits = layers.Dense(num_classes)(features)
    # Create the Keras model.
    model = keras.Model(inputs=inputs, outputs=logits)
    return model

# Train Model
model = create_vit_classifier()

optimizer = keras.optimizers.AdamW(
  learning_rate = learning_rate, weight_decay =
     ↪ weight_decay
)

model.compile(
    optimizer = optimizer,
    loss = keras.losses.SparseCategoricalCrossentropy(
       ↪ from_logits = True),
    metrics = [
        keras.metrics.SparseCategoricalAccuracy(name = "
           ↪ accuracy"),
        keras.metrics.SparseTopKCategoricalAccuracy(5, name
           ↪ ="top-5-accuracy"),
    ],
)
```

```
checkpoint_filepath = "/tmp/checkpoint.weights.h5"
checkpoint_callback = keras.callbacks.ModelCheckpoint(
      checkpoint_filepath,
      monitor="val_accuracy",
      save_best_only=True,
      save_weights_only=True,
)

history = model.fit(
   x=training_images,
   y=training_labels,
   batch_size=batch_size,
   epochs=num_epochs,
   validation_split=0.1,
   callbacks=[checkpoint_callback],
)

#View all the layers of the network using the Keras
model.summary()
```

APPENDIX E

POWER MEASUREMENT CONTROLLER

```
#include <Wire.h>
#include <INA226_WE.h>
#define I2C_ADDRESS 0x40

INA226_WE ina226 = INA226_WE(I2C_ADDRESS);

unsigned long myTime;

float shuntVoltage_mV = 0.0;
float loadVoltage_V = 0.0;
float busVoltage_V = 0.0;
float current_mA = 0.0;
float power_mW = 0.0;

void setup()
{
 Serial.begin(2000000);
 while (!Serial); // wait until serial comes up on
    ↪ Arduino Leonardo or MKR WiFi 1010
 Wire.begin();
 ina226.init();

 ina226.setConversionTime(CONV_TIME_140); //choose
    ↪ conversion time and uncomment for change of
    ↪ default

 ina226.setResistorRange(0.1, 1.3); // choose resistor
    ↪ 0.1 Ohm and gain range up to 1.3A

 /* If the current values delivered by the INA226 differ
    ↪  by a constant factor
   from values obtained with calibrated equipment you
      ↪ can define a correction factor.
   Correction factor = current delivered from calibrated
      ↪  equipment / current delivered by INA226*/
```

```
  ina226.setCorrectionFactor(0.93);

  Serial.println("INA226 Current Sensor Example Sketch -
     ↪ Continuous");

  ina226.waitUntilConversionCompleted(); //if you comment
     ↪  this line the first data might be zero
}

void loop()
{
  myTime = micros();

  ina226.readAndClearFlags();
  shuntVoltage_mV = ina226.getShuntVoltage_mV();
  busVoltage_V = ina226.getBusVoltage_V();
  current_mA = ina226.getCurrent_mA();
  power_mW = ina226.getBusPower();
  loadVoltage_V = busVoltage_V + (shuntVoltage_mV / 1000)
     ↪ ;

  Serial.print("Time[us]:");
  Serial.print(myTime);
  Serial.print(",");
  Serial.print("Voltage[V]:");
  Serial.print(loadVoltage_V);
  Serial.print(",");
  Serial.print("Current[mA]:");
  Serial.print(current_mA);
  Serial.print(",");
  Serial.print("Power[mW]:");
  Serial.println(power_mW);

  if (ina226.overflow)
  {
    Serial.println("Overflow! Choose higher current range
       ↪ ");
  }
}
```