

AMERICAN UNIVERSITY OF BEIRUT

A Quantitative Evaluation of Tiny Machine
Learning Models for Limited-Vocabulary Speech
Processing Applications

by

YASMINE ALI ABU ADLA

A thesis

submitted in partial fulfillment of the requirements
for the degree of Master of Engineering
to the Department of Electrical and Computer Engineering
of Maroun Semaan Faculty of Engineering and Architecture
at the American University of Beirut

Beirut, Lebanon
April 2023

AMERICAN UNIVERSITY OF BEIRUT

A Quantitative Evaluation of Tiny Machine Learning Models for Limited-Vocabulary Speech Processing Applications

by

YASMINE ALI ABU ADLA

Approved by:

Dr. Mazen Saghir, Associate Professor
Electrical and Computer Engineering

Advisor



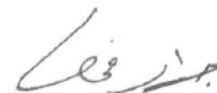
Dr. Mariette Awad, Associate Professor
Electrical and Computer Engineering

Co-Advisor

Mariette Awad

Dr. Jihad Fahs, Assistant Professor
Electrical and Computer Engineering

Member of Committee



Dr. Wassim El Hajj, Professor
Computer Science

Member of Committee



Date of thesis defense: April 26, 2023

AMERICAN UNIVERSITY OF BEIRUT

THESIS RELEASE FORM

Student Name: Abu Adla Yasmine Ali
Last First Middle

I authorize the American University of Beirut, to: (a) reproduce hard or electronic copies of my thesis; (b) include such copies in the archives and digital repositories of the University; and (c) make freely available such copies to third parties for research or educational purposes

- As of the date of submission of my thesis**
 After 1 year from the date of submission of my thesis .
 After 2 years from the date of submission of my thesis .
 After 3 years from the date of submission of my thesis .



Signature

May 8, 2023

Date

ACKNOWLEDGEMENTS

I would like to express my deepest appreciation to my supervisor, Dr. Mazen Saghir and Co-Advisor Dr. Mariette Awad, for their unwavering support, guidance, and encouragement throughout my research journey. Their expertise, insight, and constructive feedback have been invaluable in shaping my ideas and refining my research methodology.

I am also grateful to the members of my thesis committee, Dr. Jihad Fahs and Dr. Wassim El Hajj, for their valuable contributions, insightful feedback, and constructive criticism during the development of this thesis. Their collective expertise has played an instrumental role in shaping the direction and scope of my research.

Furthermore, I would like to extend my sincere appreciation to my colleagues, friends, and family members who have provided me with unwavering support and encouragement during the challenging times. Their unwavering support, words of wisdom, and kindness have been a constant source of inspiration and motivation throughout my academic journey.

Finally, I would like to express my gratitude to the American University of Beirut, which has provided me with the resources, infrastructure, and intellectual environment necessary for the successful completion of my research. I would also like to extend my sincere thanks to the US-Middle East Partnership Initiative (MEPI) for providing me with a full scholarship to participate in The Tomorrow's Leaders Graduate Program. This program, funded by the U.S. Department of State's MEPI, not only supported my academic pursuits but also provided me with invaluable leadership development training. Without this scholarship, pursuing graduate studies in a field that I am passionate about would have been nearly impossible. I am grateful for the opportunity to have been a part of this program and for the unwavering support of the MEPI team.

Thank you all for your invaluable support, encouragement, and guidance throughout this journey.

ABSTRACT

OF THE THESIS OF

Yasmine Ali Abu Adla for Master of Engineering
Major: Electrical and Computer Engineering

Title: A Quantitative Evaluation of Tiny Machine Learning Models for Limited-Vocabulary Speech Processing Applications

Tiny Machine Learning (TinyML) is a rapidly growing field that aims to bring machine learning to resource-constrained embedded systems such as microcontrollers. These devices have limited processing power, memory, and energy, which makes it challenging to deploy traditional machine learning models designed to run on powerful servers with large amounts of memory and processing capabilities. To address this challenge, TinyML models are highly optimized and compressed, using techniques such as quantization, pruning, and weight sharing to reduce their memory footprint and increase their computational efficiency. This allows intelligent applications to run on devices that were previously incapable of running complex algorithms.

This study investigates the impact of model compression techniques on the performance of four deep learning models - Convolutional Neural Networks, Long Short-Term Memory, Gated Recurrent Units, and Bidirectional Long Short-Term Memory- for a limited-vocabulary speech processing task in Arabic, specifically focusing on the Levantine dialect. We evaluate the effectiveness of these techniques in reducing the memory footprint of the models, improving their accuracy and performance, and minimizing inference time and energy consumption. To evaluate the real-world performance of our optimized models, we deploy them on two distinct edge devices that represent different resource-constrained environments. One device has limited processing power and memory, while the other has relatively more computational resources but still constrained by limited memory. By analyzing the performance of our models on these two devices, we gain insights into the effectiveness of different compression techniques for TinyML models and their suitability for deployment on edge devices.

Our experiments demonstrate the efficacy of model compression techniques in significantly reducing the memory footprint of deep learning models by up to 89%

while maintaining an accuracy of over 97%. Moreover, the optimized models result in a significant reduction in inference time and energy consumption by 99%, making them highly suitable for deployment on resource-constrained edge devices. Our optimized models achieve real-time performance for limited-vocabulary speech recognition tasks, with an average inference time of less than 500ms on both edge devices.

Overall, this study highlights the potential of model compression techniques for developing efficient TinyML models that can be deployed on resource-constrained edge devices. The significant reduction in memory footprint, inference time, and energy consumption of our optimized models showcases their practicality and effectiveness for real-world applications. The efficient and accurate speech processing techniques developed in this study have the potential to significantly improve Arabic speech applications, such as improving the accuracy of Arabic speech recognition and enabling efficient speech-to-text translation. The deployment of these techniques on resource-constrained edge devices can facilitate the development of Arabic speech applications in various domains, such as healthcare, education, and business. Furthermore, the optimized models developed in this study can enhance communication and accessibility for Arabic speakers with speech impairments or disabilities. The potential of TinyML-based Arabic speech processing applications to improve communication and accessibility for Arabic speakers is vast, and this study provides valuable insights into the development of efficient and practical models for this purpose.

Keywords— TinyML, neural networks, Arabic speech recognition, model compression, quantization, pruning, weight clustering, edge devices.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS	1
ABSTRACT	2
ABBREVIATIONS	8
1 Introduction	10
2 Related Work	12
2.1 Digital Assistants	12
2.1.1 Amazon Alexa	12
2.1.2 Google Assistant	14
2.1.3 Apple Siri	15
2.1.4 Microsoft Cortana	15
2.2 Machine Learning Models for Speech Processing Applications	16
2.2.1 Hidden Markov Models	17
2.2.2 Convolutional Neural Networks	19
2.2.3 Recurrent Neural Networks	20
2.2.4 Long Short-Term Memory Units	23
2.2.5 Deep Learning Models for Natural Language Processing Applications	24
2.2.6 Machine Learning Frameworks and Libraries	26
2.2.7 Deep Learning Frameworks and Libraries	26
2.3 Speech Datasets	27
2.3.1 Arabic Datasets	27
2.3.2 English Datasets	29
2.3.3 Large Data Sets	29
2.4 Embedded Systems	31
2.4.1 Single Board Computers and Microcontrollers	31
2.4.2 Digital Microphones	34
3 Overview of TinyML	36
3.1 TinyML Frameworks	36
3.1.1 TensorFlow Lite	37
3.1.2 Edge Impulse	37
3.1.3 Embedded Learning Library	37
3.1.4 ARM-NN	38
3.1.5 STM32 Cube AI Library	38
3.2 TinyML for Speech Recognition – Related Work	38

3.3	Techniques for Model Compression	41
3.3.1	Quantization	42
3.3.2	Pruning	42
3.3.3	Weight Clustering	44
3.3.4	Hybrid Approaches	44
4	A TinyML Model for Spoken Arabic Digit Recognition - Methodology	45
4.1	Levantine Arabic Audio Dataset	45
4.2	Data Pre-processing and Cleaning	46
4.2.1	Data Augmentation	46
4.2.2	Mel-Frequency Cepstral Coefficient Feature Extraction	48
4.2.3	Normalization	49
4.3	Automatic Speech Recognition Models	49
5	Results and Discussion	55
5.1	Impact of TFLite Compression on Model Characteristics	55
5.1.1	Impact on Parameter and Model Size	56
5.1.2	Impact on Model Performance	57
5.1.3	Impact on Inference Time	62
5.1.4	Impact on Energy Consumption	66
5.2	Deployment on Edge Devices	68
5.2.1	Raspberry Pi Model 3	68
5.2.2	Arduino Nano 33 BLE Sense	69
6	Conclusions and Future Work	71
	APPENDIX	73
	Bibliography	80

ILLUSTRATIONS

2.1	Block Diagram of Amazon Alexa Architecture. Adopted from [9]	14
2.2	Three State Hidden Markov Model	18
2.3	Recurrent Neural Network Architecture	21
2.4	Difference in Feed-forward Neural Network Architecture and Recurrent Neural Network	21
2.5	Architecture of LSTM Adapted from [32]	23
2.6	I2S MEMS Microphone Block Diagram. Adapted from [70]	34
2.7	Frequency Response of the Digital MEMS Microphone. Adapted from [71]	35
2.8	Timing Diagram showing an I2S Transaction. Adapted from [72]	35
3.1	Constant Sparsity Pruning	43
3.2	Constant Sparsity vs. Polynomial Decay Pruning	43
4.1	Typical TinyML Workflow	45
4.2	Demographic Distributions	47
4.3	MFCC Extraction	48
4.4	Model Accuracy on Testing Set	49
4.5	CNN Model	51
4.6	LSTM Model	52
4.7	GRU-LSTM Model	53
4.8	Bi-LSTM Model	54
5.1	Impact of Pruning on Models' Accuracy	59
5.2	Impact of Weight Clustering on Models' Accuracy	60
1	Impact of Pruning on Models' Precision	74
2	Impact of Pruning on Models' Recall	75
3	Impact of Pruning on Models' F1 Score	76
4	Impact of Weight Clustering on Models' Precision	77
5	Impact of Weight Clustering on Models' Recall	78
6	Impact of Weight Clustering on Models' F1 Score	79

TABLES

2.1	Transition Matrix of a Three State Hidden Markov Model	17
2.2	Comparison between the Technical Specifications of Different SBCs	33
5.1	Impact of Compression Optimizations on Model Size in MB	57
5.2	Impact of Compression Optimizations on Model Accuracy	61
5.3	Impact of Compression Optimizations on Model Precision	61
5.4	Impact of Compression Optimizations on Model Recall	61
5.5	Impact of Compression Optimizations on Model F1-Score	61
5.6	Impact of Compression Optimizations on inference time in CNN Model . .	63
5.7	Impact of Compression Optimizations on Inference Time in LSTM Model .	63
5.8	Impact of Compression Optimizations on Inference Time in GRU Model . .	64
5.9	Impact of Compression Optimizations on Inference Time in Bi-LSTM Model	64
5.10	Impact of Compression Optimizations on Energy Consumption in Joules . .	67
5.11	Comparison of Deployment Techniques on Raspberry Pi Model 3	69

ABBREVIATIONS

ARM	Advanced Reduced Instruction
ADC	Analog-to-Digital Converter
API	Application Programming Interface
ARM	Advanced Reduced Instruction Set Computer Machine
AI	Artificial Intelligence
AVS	Amazon Voice Service
AWS	Amazon Web Services
ASR	Automatic Speech Recognition
Bi-LSTM	Bi-directional Long Short-Term Memory
BPTT	Backpropagation Through Time
BERT	Bidirectional Encoder Representations from Transformers
CNTK	Cognitive Toolkit
CPU	Central Processing Unit
CV	Computer Vision
CRA	Convolutional-recurrent-attention
DCs	Data Centers
DM	Data Mining
DL	Deep Learning
DNN	Deep Neural Networks
DTW	Dynamic Time Wrapping
EI	Edge Impulse
ELL	Embedded Learning Library
EON	Enterprise Objects Framework
FB	Filter Banks
FC	Fully Connected
FFT	Fast Fourier Transform
FLOPs	Floating-point Operations
GPIO	General-Purpose Input/Output
GPUs	Graphics Processing Units
GRU	Gated Recurrent Unit
HMM	Hidden Markov Model
HTTP	Hypertext Transfer Protocol
iOS	iPhone Operating System
IDE	Integrated Development Environment
ICs	Integrated Circuits
I2C	Inter-Integrated Circuit
IoT	Internet of Things
JSON	JavaScript Object Notation

IPA	Intelligent Personal Assistant
LSTM	Long Short-Term Memory
MCUs	Microcontrollers
MF	Mel Frequency
MFFCs	Mel Frequency Cepstral Coefficients
MEMS	Micro-electromechanical systems
MSA	Modern Standard Arabic
MSB	Most Significant Bit
ML	Machine Learning
NN	Neural Network
NLP	Natural Language Processing
NLU	Natural Language Understanding
ONNX	Open Neural Network Exchange
OS	Operating System
PCM	Pulse-Code Modulation
PDM	Pulse Density Modulated
PoST	Part of speech tagging
RAM	Random-access memory
RISC	Reduced Instruction Set Computer
RNNs	Recurrent Neural Networks
SBCs	Single Board Computers
SCK	Bit Clock
SD	Serial Data Line
SPI	Serial Peripheral Interface
SNR	Signal-to-Noise Ratio
SVM	Support Vector Machines
SoC	System on Chip
TF	TensorFlow
TFLite	TensorFlow Lite
TFLM	Tensorflow Lite for Microcontrollers
TinyML	Tiny Machine Learning
TPD	Thermal Design Power
UART	Universal Asynchronous Receiver/Transmitter
WS	Word Select

CHAPTER 1

INTRODUCTION

Voice assistants like Apple Siri, Google Assistant, and Amazon Alexa are widely used in smart phones and smart speakers to search for information, play media, and control home appliances [1]. The technology is also being introduced to the transportation, healthcare, and manufacturing industries to ease consumer access to services and improve worker efficiency and safety [2].

Voice assistants rely on sophisticated Machine Learning (ML), Deep Learning (DL), and Natural Language Processing (NLP) models to process and understand speech. The models are based on deep, complex, neural networks that require large data storage and intensive computations for both training and inference. Because smart phones and smart speakers have limited resources to store and run these models, speech queries are mainly processed in cloud data centers. To operate reliably, these devices therefore need access to high-bandwidth Internet.

As more speech-enabled devices come online, it will become harder to provide the necessary communication, storage, and compute bandwidth to handle the increased speech traffic in cloud data centers. To address this problem, researchers in industry and academia are developing techniques to compress machine learning models so they can run on resource-limited devices deployed closer to their data sources at the edge of the cloud. For example, Google, which developed the popular *TensorFlow* machine learning framework, also developed *TensorFlow Lite* (TFLite) to compress large machine learning models so they can be deployed on smart phones and edge devices. More recently, Google introduced *Tensorflow Lite for Microcontrollers* (TFLM) to enable tiny machine learning (TinyML) models to run on inexpensive and resource-constrained microcontrollers.

TFLM uses a number of techniques to compress TensorFlow Lite models. These include weight quantization, pruning, and clustering. However, little is understood about how best to apply these optimizations and their actual impact on the size, accuracy, energy consumption, and inference time of compressed machine learning models.

Throughout this research, we aim to answer two research questions:

1. How can time series algorithms for limited-vocabulary speech processing applications be deployed for inference on microcontrollers?
2. What is the impact of available compression algorithms on the performance of machine learning models in terms of performance, energy consumption, inference time, and memory footprint?

In this study, we explored the impact of different compression techniques on four different Deep Learning model for spoken Arabic digit recognition. This application was used as a case study in our research work to enable Arabic speakers with physical disabilities to operate an elevator using simple speech commands. Because it is impractical and expensive to provide high-bandwidth Internet access to an elevator cabin, a compressed model could enable an offline, TinyML speech interface to run on an inexpensive microcontroller inside the elevator.

This study is structured into six chapters. Chapter 2 discusses the relevant work on Digital Assistants, Machine Learning Models for Speech Processing Applications, Available Speech Datasets, and Embedded Systems. Chapter 3 provides an overview of TinyML and the different techniques used for model compression. Chapter 4 presents our methodology, while Chapter 5 discusses our results, including the impact of different compression techniques on the size, accuracy, energy consumption, and inference time of the DL models. In Chapter 6, we present our conclusions and describe future work.

CHAPTER 2

RELATED WORK

In this chapter we provide an overview of a number of topics that are relevant to this work. Section 2.1 provides an overview of digital assistants and the technologies behind them. Section 2.2 presents different ML algorithms for speech recognition applications and available ML and DL frameworks and libraries. Section 2.3 lists the available speech datasets in different languages in various speech processing applications and Data Centers. Section 2.4 provides an introduction to embedded systems by comparing the technical specifications of the most popular Single Board Computers (SBCs) and discusses digital Micro-electromechanical systems (MEMS) microphones and the related digital communication protocol.

2.1 Digital Assistants

During the past several years, there has been a significant increase in the deployment of voice-controlled Personal Digital Assistants. Over the past two decades, technological advancements, such as Automatic Speech Recognition (ASR), Natural Language Processing, and Text-to-Speech Synthesis, have been integrated into commercial Digital Assistants such as Apple’s Siri, Google Assistant, and Microsoft Cortana. In fact, these recent technologies have managed to showcase the concept of Artificial Intelligence (AI) as an end-user product. Personal Digital Assistants, nowadays, are able to easily anticipate and provide the users’ needs and requests. With the rise of technological advancements, Digital Assistants are able to interact with the user and take on mundane tasks, such as setting alarms, controlling household appliances, and setting up schedules and calls, to ease the user’s life.

The following subsections give an overview on some of the most common Personal Digital Assistants: how they work and a brief technological overview.

2.1.1 *Amazon Alexa*

Amazon Alexa is a voice-controlled Intelligent Personal Assistant (IPA) that has been deployed by the Amazon company for its line of Echo devices. Alexa has been designed as a software, that operates on Amazon Echo devices and performs voice-based tasks and functions while communicating through local Wi-Fi Internet connection with Amazon’s AWS (Amazon Web Services) cloud servers or any other networked devices as seen in Figure 2.1. The information architecture of the command process is as follows: the user starts by saying a hot key word then gives a command [3]. In order to deal with the

request, Alexa sends a JavaScript Object Notation (JSON) request to an AWS Lambda Function in the cloud. The Lambda function, which is serverless computing function that contacts servers on the cloud, is found at the back-end code that deals with the intent and provides Alexa with the appropriate user response. When the command is sent to the cloud, the first Alexa system to receive is the audio signal is the ASR that converts the audio to a text string. Then the text is sent to the Natural Language Understanding (NLU) system to interpret the recognition result and generate an intent. And finally, a TT (component is used to convert the NLU output as synthesized speech [4]. Alexa generates a “Card” of information, which is available to the users on the Alexa app, to keep track of the request and answer given to the user. Furthermore, Alexa utilizes data retention to get smarter and better each day; the more a customer interacts with Alexa, the more data it would have to process, store, and continuously train the Machine Learning algorithm and adopt better to the consumer’s speech patterns, vocabulary, and pronunciation [4].

In addition to Alexa being able to obtain data from Amazon servers, it is also able to control home appliances such as the lighting and in-house temperature. The user is able to control home devices using Alexa by registering through the Alexa Skill app, where the user will have to scan for the appliance and connect it to the internet [5]. First, the user utters the voice command to the Amazon Echo device, which in return receives the command and transfers it to the Amazon Voice Service (AVS). The audio is processed by the AVS and converted into text and forwarded to Alexa Skills. Alexa Skill is a type of General-Purpose Input/Output (GIPO) control that is able to control various appliance. The Skill sends the JSON data to Ngrok cloud Service, which in return transfers the data to the server located inside of the device. Once the device receives the JSON data, it executes the control command [5]. Alexa has multiple microphones that are able to detect speech patterns from any direction by employing noise cancellation and far-field voice recognition. The user is able to activate Alexa by triggering its speech recognition software, which is based on a Convolutional-recurrent-attention (CRA) model, through certain “wake-up words” such as “Alexa” or other predefined wake-up words. The user is able to select the desired wake-up word by either asking Alexa to change the wake word, where the user is asked to utter the word and wait for Alexa to approve, or through the Alexa app, where the user just selects the wake word of choice [6]. However, despite current voice-recognition advancements, the current Alexa is still not able to distinguish between the voices of several users [3].

Recently, Amazon has empowered Alexa with a full set of Arabic language skills to communicate in Arabic with users with different dialects for region [7]. Alexa is now able to stream Arabic music, recite the Quran, and integrate the Hijri calendar. The new and improved Alexa is able to understand commands in the Modern Standard Arabic (MSA) and the Gulf dialects [8]. Alexa is also able to output speech in both dialects but uses the Gulf dialect for less formal speech such as setting alarms and streaming music. The major difference when dealing with the Arabic language is that Amazon has decided to use only two diacritics, the shaddah and maddah, in the words to enhance the performance of the model when passing entity words from the ASR to the NLU and all the way to the text-to-speech engine. The model was trained at the beginning using a Bidirectional Encoder Representations from Transformers (BERT)-based language model, which is pre-trained on the Arabic, French and English languages using unlabeled data. The aim was to introduce sentences that had masked out words for the model to try and predict. Then the trained model was used to perform NLU tasks and fine-tuned on an annotated and labeled corpus comprising of French and English data. This was done to teach the model

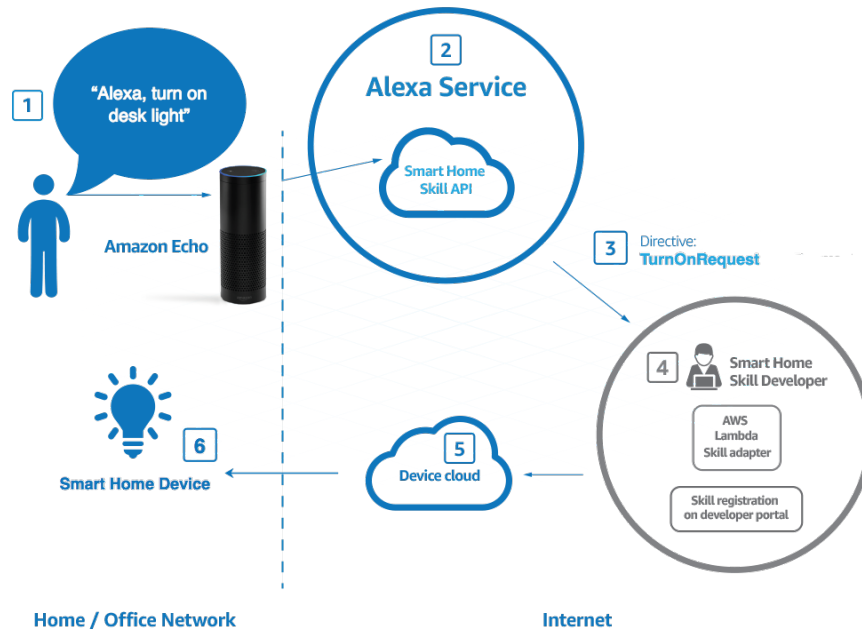


Figure 2.1: Block Diagram of Amazon Alexa Architecture. Adopted from [9]

the general principals of NLU which can be transferred to an optimized model on the Arabic labeled data. Finally, the model is fine-tuned on a balanced dataset comprising of the three languages to ensure that model optimization on the Arabic language did not compromise the performance of the model on the remaining two languages.

2.1.2 Google Assistant

Google Assistant is an Artificial Intelligence based virtual assistant powered by Google, that is available on mobile devices and smart household appliances. Google Assistant relies on Deep Neural Networks (DNN) to provide users with services such as voice commands and voice searching to complete tasks or control a certain device [10]. In order to communicate with Google Assistant, the user starts a conversation by saying the hot word “Hey Google!” and then proceeds to request a certain task or intent. The entire action is then run on the cloud regardless of the type of device used. In fact, the enabled device sends the user’s command to Google Assistant, which sends it to the fulfilment service via Hypertext Transfer Protocol (HTTP) POST requests [11]. To illustrate, a fulfillment is a service, or application, or logic that takes the intent of a user and figures out an adequate action. The fulfilment finds a relevant response and then sends it back to Google Assistant, which in return returns it to the user [11].

Despite many encountered problems and inaccuracies, Google Assistant can be used offline only if the needed data is downloaded beforehand. To illustrate, if the user wants Google to play music offline, he will have to download a certain playlist for Google to stream; additionally, if the user wants Google’s help in navigation, he/she would have to download the map of the area beforehand. Google Assistant is also able to perform simple commands such as turning on/off the flashlight, turning on Airplane mode, and adjusting the speaker volume [12].

Recently, Google has announced that Google Assistant on mobile devices now supports Arabic for Egyptian and Saudi Arabian users[12].

2.1.3 *Apple Siri*

Siri is a voice-controlled virtual assistant for Apple’s iPhone Operating System (iOS) users that utilizes sequential natural language inference and contextual awareness to respond to user requests. When the user says the hot word “Hey Siri!”, the device is activated and the user’s request is converted and sent as a voice file (at a rate of 16,000 samples per second) to Apple’s data center, where it is converted to text via the Nuance speech-to-text engine. Everything sent to Apple Siri is processed on the cloud so that much of the offload is processed on powerful computers rather than the mobile device itself and so that the data continuously improves the service. Then, Siri utilizes Natural Language Processing to understand the context of the intent by analyzing the subject keyword and linking it to connected objects and verbs. This is done by a DNN that sifts through thousands of phrases to determine what the input phrase means. Finally, once Siri understands the given task, it determines what task needs to be done and then returns to the user the needed action [13].

Recently, Apple’s new iOS 15 release has introduced new on-device speech processing to process audio requests entirely on the iPhone rather than it being on the server side. Apple has included in its products the A12 Bionic chip for offline support and request processing, such as setting timers and alarms, making phone calls, sending messages, as well as controlling audio playback. The A12 is a 64-bit Advanced Reduced Instruction Set Computer (RISC) Machines (ARM)-based System on Chip (SoC) [14].

Despite many encountered problems, Apple has trained Siri to understand Arabic commands in the Modern Standard Arabic dialect. However, many users have portrayed their frustration when trying to communicate with the Arabic version of Siri.

2.1.4 *Microsoft Cortana*

Microsoft Cortana is yet another cloud-based voice-activated IPA that has been developed by Microsoft and launched into windows 10 devices for Android users. Cortana is able to set reminders, send emails, surf the web, and even tell jokes. After the user activates the device by saying “Hey Cortana!” and inputs a certain task, the voice signal is processed via signal processing algorithms that include analog to digital conversion, filtering, and gain control. Then, ASR is performed using the Dynamic Time Wrapping (DTW) algorithm, which performs speech recognition based on template matching. The DWT algorithm decodes the feature vector into a sequence of words and then matches the audio to popular English words that exist in Cortana’s program dictionary. After detecting and understanding the speech, Cortana uses “command mode” semantic property to determine how to answer the user and output a list of instructions that need to be executed. After performing the task, the user is prompted with a response [15].

All in all, most of the utilized IPAs are cloud-based algorithms that require constant Wi-Fi access and connection. In spite of the numerous advantages of having cloud-based IPAs, this remains impractical in cases and locations where constant Internet connection is unavailable. Furthermore, most of these voice-controlled digital assistants only except commands/request in certain languages such as English, German, Italian, French, Por-

tuguese, and Spanish and in limited dialects. IPAs that do indeed support the Arabic language is only offered in the MSA dialect or in the Egyptian and Gulf dialects; none of these virtual assistants support the Levantine dialect let alone the Lebanese one. Therefore, there has been a growing and immense need to develop an IPA that can be accessed and used offline. Additionally, it is of great importance for any voice-based product to understand and communicate with users via the Arabic language and in different dialects as well. In response to this need, we propose an approach utilizing machine learning techniques for keyword spotting specifically tailored for the Levantine dialect, aiming to design an offline IPA that is effective and accurate for Arabic users

2.2 Machine Learning Models for Speech Processing Applications

Over the past several years, Artificial Intelligence, namely Machine Learning, has grown immensely in the field of data analytics and computation. AI includes any method that enables computers to mimic human behavior such as Machine Learning, Natural Language Processing, Computer Vision (CV), robotics etc. As for ML, it aims to build intelligent systems that are able to learn and improve from experience without being explicitly programmed. In order to develop real-world applications, such as smart technologies, automation, and exploratory data processing, ML algorithms are needed [16]. ML algorithms can be divided into four categories: supervised learning, unsupervised learning, semi-supervised learning, and reinforcement learning. The branch of supervised learning involves the training of a model with labeled data, while unsupervised learning allows the model to discover patterns and clusters within the data itself. As for reinforcement learning, it involves the training of a model based on a system of reward and punishment. In this paper, the main focus is on supervised learning models, specifically Neural Networks.

Supervised learning algorithms necessitate the presence of labels for the input data; these labels are used to train, evaluate, and optimize the ML model to make predictions. The most popular supervised ML algorithms include Linear and Logistic Regression, Support Vector Machines (SVM), and Neural Networks (NN). Neural Networks is a growing domain in the ML field that is inspired by the human biological neural networks. NNs are typically depicted as a set of connected units organized in layers known as artificial neurons. The main advantage of utilizing supervised learning algorithms is the high achievable accuracy and the predefined number of categories. As for the main disadvantage, it includes model overfitting, where the model performs well on the training data but models the noise as well causing it to perform poorly on the new testing data, and the how computationally expensive it is [17].

On the other hand, unsupervised learning algorithms do not require the presence of labels to train the ML models but rather discover knowledge from the data. One of the most popular unsupervised learning algorithms, is the Hidden Markov Model (HMM), which handles sequence data samples. HMM can be trained using sequence-based normal samples to generate normal base lines. The samples that have low likelihood are categorized as having abnormal behavior. Unsupervised learning algorithms are faster to train and computationally less expensive when compared to supervised algorithms. However, the main challenge of unsupervised models is how to identify the features for modeling [17].

The following sections give an overview of the most popular ML algorithms that are most commonly used in NLP applications, such as Hidden Markov Models, Recurrent

Table 2.1: Transition Matrix of a Three State Hidden Markov Model

State	Next State			
	Verb (V)	Noun (N)	Adj (A)	
Previous State	Verb	$P(V V) = 0.03$	$P(N V) = 0.95$	$P(A V) = 0.02$
	Noun	$P(V N) = 0.48$	$P(N N) = 0.02$	$P(A N) = 0.5$
	Adj	$P(V A) = 0.03$	$P(N A) = 0.93$	$P(A A) = 0.04$

Neural Networks, and Long Short-Term Memory Units.

2.2.1 Hidden Markov Models

Hidden Markov Models are one of the most common unsupervised learning models that consists of a finite set of states. The transition probability matrix or distribution is the set of probabilities that determine the transition between the states. In fact, transitioning between states depends on a certain input. Additionally, the probability of transitioning depends on the probability of transitioning from the previous state to the current one. The transition between states continues until an output state is reached or an observation is made. However, given that the states are hidden to the external observer, this type of model is called the Hidden Markov Model [17], [18].

The HMM probability model has three main elements: a set of experiments having well-defined results, a sample space (Ω), and the event, which is a subset of the sample space. Furthermore, the HMM depend on conditional probability, which means that the probability of a certain event X to happen, depend on the occurrence of a previous event Y. The formula of conditional probability is shown in Equation 2.1 below

$$P(X/Y) = \frac{P(X \cap Y)}{P(Y)} \quad (2.1)$$

where

- $P(X/Y)$ is the conditional probability
- $P(X \cap Y)$ is the probability intersection between event X and Y
- $P(Y)$ is the probability of event Y occurring

An example of a three state (Noun, Verb, Adj) HMM is shown in Figure 2.2 below. Values of the transition probabilities are shown as well, where, for example, the transition probability from adj to verb is 0.03 and that from noun to verb is 0.48. Table 2.1 shows the transition matrix that shows the transition probabilities from one state to another.

2.2.1.1 Hidden Markov Models for Natural Language Processing Applications

Hidden Markov Models are used in the literature for various NLP applications in the Arabic Language such as morphological analysis, part of speech tagging, and text classification. The next subsections discuss these techniques respectively.

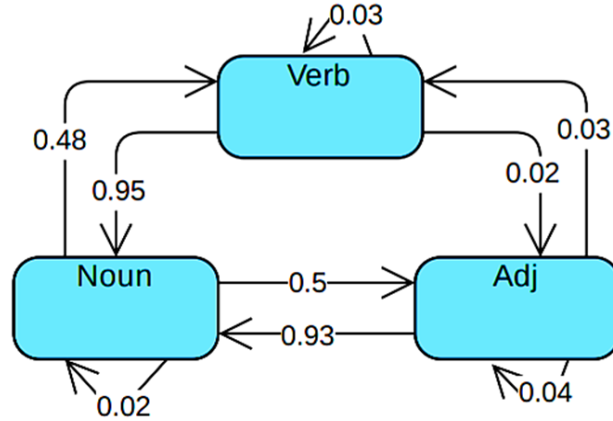


Figure 2.2: Three State Hidden Markov Model

Morphological Analysis HMM are used in morphological analysis, where the morpheme, or the smallest unit of meaning, of a given word is found. When it comes to Arabic words, their origin comes from three (tri-root) or four-letter (quad-root) words. By adding weights to these basic blocks, other words are formed with various meanings; by detecting the weight used, the root can be then determined [18].

Research done by Boudlal et al. [19] utilized HMM in morphological analysis for Arabic words. The first step was to divide the sentence into words and then proceed to find the root of each word. Whenever a word is identified, it was further divided into its prefix, suffix, a root. The root is then determined based on the context; this was done by representing all the possible roots as hidden states in the HMM and the best one was selected based on the observation. In addition, the location of the word in the sentence was taken into consideration. Researchers were able to attain a correct root for more than 98% of the training set of an already written corpus consisting of 500,000 Arabic words.

Another study was able to determine the stem of the word by eliminating the prefixes and suffixes in any word using HMM. In this case, the word is decomposed into the letters it is formed of and the states of the model are the prefixes and the suffixes while the transition is done by moving from one letter to another. The highest likelihood of the word will be given the best path. After training on a set comprising of 15 million words, a precision of 95% was achieved [20].

Part of Speech Tagging HMM tagging, otherwise known as sequence labeling, is a process that maps a tag sequence to an input one. To exemplify, if there is a tagging process having the following input: X_1, X_2, \dots, X_m then the output will be a sequence having the following elements: Y_1, Y_2, \dots, Y_m . Part of speech tagging (PoST) is when the input is a sentence, and the output is a tag for each word in that sentence. For example, if we have a sentence consisting of four words, the output will consist of four tags stating the part of speech for each word [18].

A study done on PoST took into consideration the structure of a sentence since it is a crucial part in forming Arabic sentences. In this research, other than depending on the sentence structure, HMM as well as morphological analyzers were applied. The first

step was to use the morphological analyzer to reduce the size of the tags lexicon, which can be done by deriving the root of the Arabic word. The HMM was used to model the sentence structure to consider the sequence of logical linguistics. In the HMM, the states were the tags and the transitions were determined by the syntax of the sentence. The proposed model was trained and tested on a book dated back to the Third Hijri century; a recognition rate of 96% was achieved when evaluating the model [21].

Researchers Hadni et al. [22] integrated HMM and the Rule Based method to create a PoST tagging hybrid model. Three tags were introduced in this study: Noun, Verb, and Particles. The hybrid model was trained and evaluated on the Holy Quran and Kalimat Corpuses where an accuracy of 98% and 97.6% was obtained respectively.

Another research study used the HMM and Rule Based method but instead of using three tags, they introduced four: Noun, Verb, Particle, and Quranic. When training and testing their hybrid model, the Holy Quran was used and an accuracy of 97.6% was achieved [23].

Text Classification Text classification is an automated algorithm that is used to sift and sort documents based on different categories. Text classification has recently gained great interest given that it can be employed in various applications such as automatic indexing, monitoring news, and routing emails. This type of process can be achieved using HMM.

A proposed study applied HMM to extract features in text classification such as the prefixes, suffixes, and the stem of a word. After that, the extracted weights were unified by grouping different states that have a common meaning. In the Markov model, each state is a letter, and the word is formed by many transitions. The parameters of the HMM were learned according to a set of steps [24].

Automatic Speech Recognition Automatic Speech Recognition is an interdisciplinary field of computational linguistics that develops methodologies that allow intelligent machines perform to identify audio recordings spoken aloud and convert them into readable text [25].

Research done by Dua et al. [26] proposed a system for ASR of isolated words in the Punjabi language. The Hidden Markov Model Toolkit, which is based on the HMM, was exploited and tested on a dataset comprising of 115 different words from different speakers, each repeated three times. The proposed system achieved an accuracy of 95.63%.

Another research trained and tested the HMM on the news broadcast speech corpus of the MSA and Egyptian colloquial Arabic. Their dialectal speech recognition system reached an accuracy of 99.34% [27].

Although HMM are worth exploring, in this study we focus on newer Neural Networks, such as Recurrent Neural Networks (RNNs) and Long Short-Term Memory Units, that have been introduced as an improvement to traditional Markov Models.

2.2.2 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) are an essential tool for image recognition and computer vision tasks, as they are designed to extract features from images at different levels of abstraction. They have been widely used in various applications, including object detection, face recognition, and image segmentation. However, CNNs are not limited to visual data, and they have also been applied successfully in speech recognition.

The architecture of a typical CNN for speech recognition includes convolutional layers, pooling layers, and fully connected layers. The convolutional layers apply filters to the input audio signal to extract relevant acoustic features, such as pitch, timbre, and spectral information. The filters are learned during the training process and can vary in size, shape, and number. The pooling layers then reduce the dimensionality of the feature maps while retaining the most important information. The most common type of pooling used in speech recognition is temporal pooling, which aggregates the features over time to reduce the size of the feature maps. Finally, the fully connected layers generate a classification output by taking the high-level features learned by the previous layers and mapping them to the output classes. The fully connected layers combine the extracted features from the previous layers to generate a final output, which in the case of speech recognition is typically a sequence of phonemes or words [28].

During the training process, the CNN learns the optimal values of the convolutional filters and the weights of the fully connected layers by minimizing a loss function. The loss function measures the difference between the predicted output and the actual output. The optimization process involves adjusting the values of the filters and weights to minimize the loss function, using techniques such as backpropagation.

One of the main advantages of using CNNs in speech recognition is that they can learn to recognize different speaking styles and accents. Researchers have developed CNN architectures that are robust to speaker variability, such as models that use frequency-domain features that are less sensitive to changes in speaking style and accent than time-domain features. Another challenge in speech recognition is the presence of background noise. To address this challenge, researchers have developed CNN models that are trained on noisy speech data, allowing them to learn to extract relevant features even in the presence of background noise. Additionally, some models use advanced techniques such as time-frequency masking, which allows them to separate speech from noise.

CNNs have shown promising results in the field of speech recognition, achieving state-of-the-art performance on benchmark datasets. However, there are still challenges to be addressed, such as improving the robustness of the models to different speaking styles and environmental factors. Nevertheless, the use of CNNs in speech recognition is an active area of research, and it is expected that they will continue to play a significant role in advancing the field.

2.2.3 Recurrent Neural Networks

Recurrent Neural Networks are supervised learning algorithms that are a subset of feed-forward NNs that have the ability to process information across time steps. RNN models can model temporal dependencies, meaning that RNNs are best suited when the input and/or output is a sequence of dependent points (sequential data points) [29]. Basically, RNNs are a set of nodes that are linked together to communicate and output results for certain problems. RNNs have a large number of hidden layers that exchange data every time step to achieve better performance. RNNs are used in various applications that introduce sequential data such as Automatic Speech Recognition, language modelling and translation, and even recognition of human activity.

RNNs provide feedback in NN so that the ML model can utilize the outputs of previous time steps while processing the current input time-step. RNNs aim to introduce memory cells, similar to the functionality of long-term and short-term memory cells in humans, to the NN model. These cells are known as recurrent layers. This memorization enables

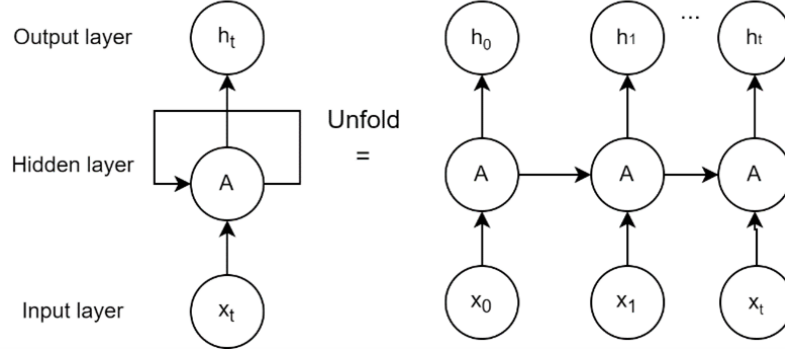


Figure 2.3: Recurrent Neural Network Architecture

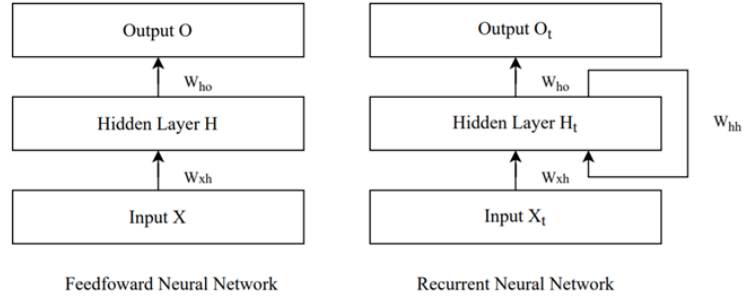


Figure 2.4: Difference in Feed-forward Neural Network Architecture and Recurrent Neural Network

the model to learn from past mistakes and make better predictions. Figure 2.3 shows an example of an RNN consisting of an input layer, two hidden layers (H), a recurrent network, and an output layer. The input layer takes in the output of a sensor, such as audio signals or speech, handwriting, or genomes, and converts it into a feature vector. Then comes two Fully Connected (FC) hidden layers followed by a recurrent network that provides the feedback. Finally, an output layer is added [30].

RNNs extend the functionality of Feed-forward Neural Networks by considering the previous inputs $X_{0:t-1}$ as well as the current input X_t as seen in Figure 2.4. This process is described in Equation 2.2 where $H_t \in \mathbb{R}^{n \times h}$ is the hidden state and the input at time t , $X_t \in \mathbb{R}^{n \times d}$ given that n is the number of samples, d is the number of inputs, and h is the number of hidden layers H . Additionally, $W_{xh} \in \mathbb{R}^{d \times h}$ represents the weight matrix, $W_{hh} \in \mathbb{R}^{h \times h}$ is the hidden-state-to-hidden-state matrix, and $b_h \in \mathbb{R}^{1 \times h}$ is the bias parameter. All these parameters are passed to the activation function ϕ , such as the logistic sigmoid function, to compute the gradients of back-propagation [31]. The final output variable is shown in Equation 2.3.

$$H_t = \phi_h(X_t W_{xh} + H_{t-1} W_{hh} + b_h) \quad (2.2)$$

$$O_t = \phi_o(H_t W_{ho} + b_o) \quad (2.3)$$

2.2.3.1 Backpropagation Through Time

Usually, Neural Network algorithms are trained using Backpropagation however, in RNNs a technique known as Backpropagation Through Time (BPTT), which is an adaptation of the regular backpropagation, is used.

The first step is to forward pass the input X_t through the network's hidden layers H_t and the output O_t one step at a time. In order to minimize the error, the cost function $\mathcal{L}(Y, O)$ is defined in Equation 2.4 to show the difference between the output value of the NN and the targeted value Y_t . The cost function or the loss function sums up all the loss terms ℓ_t computed in every iteration.

$$\mathcal{L}(O, Y) = \sum_{t=1}^T \ell_t(O_t, Y_t) \quad (2.4)$$

Using the chain rule, the partial derivative of the loss function is computed with respect to each of the three matrices W_{ho} , W_{hh} , and W_{xh} as shown in Equations 2.5, 2.6, and 2.7 respectively.

$$\frac{\partial \mathcal{L}}{\partial W_{ho}} = \sum_{t=1}^T \frac{\partial \ell_t}{\partial O_t} \cdot \frac{\partial O_t}{\partial \varphi_o} \cdot \frac{\partial \varphi_o}{\partial W_{ho}} = \sum_{t=1}^T \frac{\partial \ell_t}{\partial O_t} \cdot \frac{\partial O_t}{\partial \varphi_o} \cdot H_t \quad (2.5)$$

$$\frac{\partial \mathcal{L}}{\partial W_{hh}} = \sum_{t=1}^T \frac{\partial \ell_t}{\partial O_t} \cdot \frac{\partial O_t}{\partial \vartheta_o} \cdot \frac{\partial \vartheta_o}{\partial H_t} \cdot \frac{\partial H_t}{\partial \vartheta_h} \cdot \frac{\partial \vartheta_h}{\partial W_{hh}} = \sum_{t=1}^T \frac{\partial \ell_t}{\partial O_t} \cdot \frac{\partial O_t}{\partial \vartheta_o} \cdot W_{ho} \cdot \frac{\partial H_t}{\partial \vartheta_h} \cdot \frac{\partial \vartheta_h}{\partial W_{hh}} \quad (2.6)$$

$$\frac{\partial \mathcal{L}}{\partial W_{xh}} = \sum_{t=1}^T \frac{\partial \ell_t}{\partial O_t} \cdot \frac{\partial O_t}{\partial \vartheta_o} \cdot \frac{\partial \vartheta_o}{\partial H_t} \cdot \frac{\partial H_t}{\partial \vartheta_h} \cdot \frac{\partial \vartheta_h}{\partial W_{xh}} = \sum_{t=1}^T \frac{\partial \ell_t}{\partial O_t} \cdot \frac{\partial O_t}{\partial \vartheta_o} \cdot W_{ho} \cdot \frac{\partial H_t}{\partial \vartheta_h} \cdot \frac{\partial \vartheta_h}{\partial W_{xh}} \quad (2.7)$$

Since every H_t depends on the previous time step, the last part of the previous equations can be substituted as shown in Equations 2.8, 2.9, 2.10, and 2.11 [31].

$$\frac{\partial \mathcal{L}}{\partial W_{hh}} = \sum_{t=1}^T \frac{\partial \ell_t}{\partial O_t} \cdot \frac{\partial O_t}{\partial \vartheta_o} \cdot W_{ho} \cdot \sum_{k=1}^t \frac{\partial H_t}{\partial H_k} \cdot \frac{\partial H_k}{\partial W_{hh}} \quad (2.8)$$

$$\frac{\partial \mathcal{L}}{\partial W_{hh}} = \sum_{t=1}^T \frac{\partial \ell_t}{\partial O_t} \cdot \frac{\partial O_t}{\partial \vartheta_o} \cdot W_{ho} \cdot \sum_{k=1}^t (W_{hh}^T)^{t-k} \cdot H_k \quad (2.9)$$

$$\frac{\partial \mathcal{L}}{\partial W_{xh}} = \sum_{t=1}^T \frac{\partial \ell_t}{\partial O_t} \cdot \frac{\partial O_t}{\partial \vartheta_o} \cdot W_{ho} \cdot \sum_{k=1}^t \frac{\partial H_t}{\partial H_k} \cdot \frac{\partial H_k}{\partial W_{xh}} \quad (2.10)$$

$$\frac{\partial \mathcal{L}}{\partial W_{xh}} = \sum_{t=1}^T \frac{\partial \ell_t}{\partial O_t} \cdot \frac{\partial O_t}{\partial \vartheta_o} \cdot W_{ho} \cdot \sum_{k=1}^t (W_{hh}^T)^{t-k} \cdot X_k \quad (2.11)$$

As seen above, the powers of W_k^{hh} need to be stored as we pass by each loss term in the overall loss function. However, as the loss function becomes larger, this method becomes numerically unstable since Eigen values smaller or greater than one vanish or diverge, respectively. The solution of this problem would be to use the Truncated BPTT,

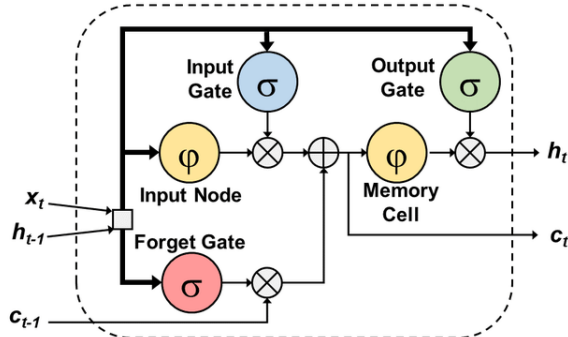


Figure 2.5: Architecture of LSTM Adapted from [32]

which truncates or cuts off the sum at a size that is computationally convenient. This means that anything preceding the cut-off time step does not get considered. Since the BPTT unfolds the RNN to add more layers at each iteration, then the truncation basically limits the number of hidden layers [31].

2.2.4 Long Short-Term Memory Units

Vanishing or exploding gradients are one of the major problems of RNNs and NNs. As seen in Equations 2.8, and 2.10, $\frac{\partial H_t}{\partial H_k}$ introduces over a long sequence a new matrix multiplication. If small values (<1) exist in the matrix multiplication, this will cause the gradient to decrease with every layer until it finally vanishes. Thus, the contribution of the previous steps towards the current time step will stop. On the other hand, if large values (>1) exist in the matrix multiplication, it will cause an exploding gradient, which will alter the weights significantly. Therefore, Long Short-Term Memory Units were introduced to handle the dilemma of the vanishing gradient [31].

LSTMs are an extension of RNNs where they allow them to learn more time steps than they previously could. LSTMs are able to achieve that by storing more information outside the architecture of the traditional NN model in structures known as gated cells. LSTMs have three main structures (Figure 2.5): an input gate I_t , an output gate O_t to read the given input, and a forget gate F_t to reset the contents of the gated cells. The computations done at each gate is shown in Equation 2.12, 2.13, and 2.14. The equations below use the weight matrices W_{xi} , W_{xf} , and $W_{xo} \in \mathbb{R}^{d \times h}$ and W_{hi} , W_{hf} , and $W_{ho} \in \mathbb{R}^{h \times h}$ have their corresponding bias terms b_i , b_f , and $b_o \in \mathbb{R}^{1 \times h}$. Furthermore, all the gates get passed through the sigmoid activation function σ to transform the output to values either having the value of 0 or 1 [31].

$$I_t = \sigma(X_t W_{xi} + H_{t-1} W_{hi} + b_i) \quad (2.12)$$

$$O_t = \sigma(X_t W_{xo} + H_{t-1} W_{ho} + b_o) \quad (2.13)$$

$$F_t = \sigma(X_t W_{xf} + H_{t-1} W_{hf} + b_f) \quad (2.14)$$

The candidate memory cell $\tilde{C}_t \in \mathbb{R}^{n \times h}$ is similar to the previous equations, but instead uses the tanh activation function to give an output between -1 and 1. The equation of the candidate memory cell is shown in Equation 2.15, where $W_{xc} \in \mathbb{R}^{d \times h}$ and $W_{hc} \in \mathbb{R}^{h \times h}$ are the weight matrices and the bias term $b_c \in \mathbb{R}_{1 \times h}$.

$$\tilde{C}_t = \tanh(X_t W_{xc} + H_{t-1} W_{hc} + b_c) \quad (2.15)$$

Another structural element of LSTMs is the old memory content $C_{t-1} \in \mathbb{R}^{n \times h}$, which controls how much old memory gets preserved in order to get the new memory content C_t (Equation 2.16).

$$C_t = F_t * C_{t-1} + I_t * \tilde{C}_t \quad (2.16)$$

Finally, Equation 2.17 shows the computation of the hidden layers $H_t \in \mathbb{R}^{n \times h}$ [31].

$$H_t = O_t * \tanh(C_t) \quad (2.17)$$

2.2.5 Deep Learning Models for Natural Language Processing Applications

RNNs and LSTMs have been widely used in the literature to deal with various NLP applications such as text generation, automatic word recognition, text classification, and even sentiment analysis. The following parts highlight the role of LSTMs and RNNs in NLP applications.

2.2.5.1 Automatic Speech Recognition

Automatic Speech Recognition is the process of using computers or machines to convert speech signals into their textual transcription. Given that CNNs, RNNs, and LSTMs are best suited for time series data, researchers started applying these models in word recognition applications.

Hunshamar [33] designed and evaluated two different RNN based wake-up word detection algorithms. The dataset used contained speech utterances from different speakers in the English language. The data was pre-processed at the beginning and then the Mel Frequency Cepstral Coefficients (MFCCs) were computed as features to be introduced to the ML model. The first algorithm utilized a sliding window to search for the wake-up words in a continuous stream of data. The other system, on the other hand, makes use of the memory capabilities of the RNN to search for the wake-up words without the need of any overlap. A shorter window frame is used, and in order to eliminate the need of overlapping computations, the final state of a frame is transferred to the first input of the proceeding frame. The sliding-window approach showed promising results as it achieved an accuracy of 97.41%, a precision of 90.51%, and a recall of 88.19%. As for the second approach, an accuracy of 97.45% was achieved as well as a precision and recall of 92.05% and 86.61% respectively.

Another study done in 2019 applied NN and LSTMs to create an Arabic speech recognition application. The Mel Frequency (MF) and Filter Banks (FB) coefficients were extracted as the desired features and then encoded to a certain vector size to train the recurrent LSTM model. The model was trained and evaluated on two datasets: spoken digit recognition and spoken TV commands, and an accuracy of 95% was obtained [34].

In 2018, a few researchers presented a DL approach for speech recognition in the Arabic language. 1040 samples of Arabic language, 840 for training and 200 for testing,

was utilized to extract the MFCC and introduce to the LSTM model. The system gave promising results, where it gave an accuracy of 94% [35].

2.2.5.2 Language Modeling

Language modeling is one of the essential elements in the field of NLP. Text generation or natural language generation, which is a type of language modeling, is the process of producing meaningful texts in applications such as automatic report generation and automatic documentation systems, etc. The aim of text generation is to empower computers to understand text vocabulary. Recently, one of the promising research domains is the deployment of RNNs on text modules to validate the learning process. In the case of text generation, the RNN and LSTM models are trained on datasets to learn text structure and meaning and then generate, as an output, a comprehensible new sequence of text. Thus, RNNs were able to prove the success of the learning process in text generation at the semantic level.

A study proposed a novel technique that utilizes LSTMs to build a generative model on Arabic texts, where it is able to predict complex and sensible sequences of long-range structure. The model was trained on the Arab World Books [36] and the Hindawi datasets [37]. In addition to the typical LSTM architecture, two gates were added to account for some features that are specific to Arabic features. To illustrate, the model was fed the schemes meaning as well as the letters non-adjacent principal. The schemes meaning allows us to understand the meaning of the word without having seen it before. As for the principle of letters non-adjacency, it states which letters cannot be adjacent to one another. Then, the model was applied on three datasets: Arabic, English, and Chinese. The standard (non-gated) LSTM model performed better on the Arabic and English datasets than on the Chinese one, where the loss function value achieved was 1.43, 1.2, and 2.13 respectively. Nevertheless, the gated model was able to improve the results obtained on the Arabic dataset since it was able to decrease the loss to reach 0.73 after 500 epochs [38].

A recent research study proposed the use of a deep and stacked LSTM model for text generation. The model was trained on the “Alice in Wonderland” text comprising of 1,63,780 characters using batch normalization to avoid the phenomena of overfitting. Then the hyper-parameters were tuned to get the optimal values. The final trained model was able to predict the next word in a sentence while maintaining its memory and history with a testing accuracy of 71.22% [39].

2.2.5.3 Character Recognition

Word or character recognition is usually used in automatic text recognition to help understand texts. Many challenges are encountered in this field given the variability in hand writings and ambiguity of some characters. Over the past few years, there has been great interest in applying RNNs and LSTMs to this field to enhance results and improve outcomes.

A study used an Arabic dataset consisting of 20,000 different handwritten words to design an automatic word recognition system [40]. The researchers applied a fully connected RNN with hidden layers containing 50 cell memory blocks each. The input layer consisted of seven nodes while that of the output layer contained 42. Online gradient descent as well as BPTT were used to train the model. When comparing the proposed model with

the HMM, it was evident that the accuracy of the RNN (88%) outperformed the Markov algorithm (71%).

2.2.6 *Machine Learning Frameworks and Libraries*

In this section, we explore the advantages and disadvantages of Machine Learning frameworks and libraries such as Scikit-Learn.

2.2.6.1 Scikit-Learn

Scikit-Learn is one of the most popular open-source Python tools that includes libraries for Data Mining (DM) and ML algorithms. Scikit-Learn has packages such as NumPy, SciPy, and Pandas that are useful when dealing with data pre-processing, feature selection and/or reduction, regression, classification, clustering, and model selection. Furthermore, it includes the Matplotlib package, which is necessary for plotting and generating graphs and charts. The following are some advantages and disadvantages of Scikit-Learn [41].

Scikit-Learn is open-source and publicly available for general purpose. It contains updated and comprehensive algorithms, in addition to it being a part of different ecosystems, where it is linked to many statistic and scientific Python packages.

On the other hand, the main drawbacks of Scikit-Learn include that Scikit-Learn is Application Programming Interface (API) oriented, which is costly in terms of development and maintenance. Moreover, its libraries do not support Graphics Processing Units (GPUs), which usually supports parallelism for large-scale DM applications.

2.2.7 *Deep Learning Frameworks and Libraries*

This section gives a description on the most popular Deep Learning frameworks and libraries while exploring their advantages and disadvantages.

2.2.7.1 TensorFlow

TensorFlow (TF) is an open-source software library that uses data flow graphs for numerical computations. TF is made to deal with large-scale distributed training as well as model inferencing in research and production systems. The mathematical operations are represented as nodes in the graphs, while the graph edges are the tensors, or data arrays, communicated between them. The architecture of the distributed TF comprises of master and slave services as well as kernel implementations, such mathematical operations, array manipulation, and state management. Furthermore, TF can run on single Central Processing Unit (CPU) systems. Below we discuss a few advantages and disadvantages of TensorFlow [41].

When it comes to advantages, TF is the most popular fast evolving open-source DL tool. It contains numerical libraries for data-flow programming, which are necessary for DL development. TF can work efficiently with multi-dimensional arrays and can support GPU/CPU computing.

On the other hand, the main disadvantage of TF is that the lower-level API is difficult to use to create DL models directly with.

2.2.7.2 Keras

Keras is a Python wrapper library that is coupled to other DL tools such as TF, Theano, and Microsoft Cognitive Toolkit (CNTK). Keras can be executed on GPUs and CPUs effortlessly given the underlying frameworks. Keras is an API designed to provide the user with the simplest and easiest design and experience. In addition, Keras utilizes a modular design to build models such as NN; where the model is a sequence of fully configurable modules that can be plugged with each other with little constraints. Additionally, Keras provides easy extensibility such that new modules are easy to add. Models can be coded using the Python Language, which makes it more compact and easier to debug. The pros and cons of Keras are discussed below [41].

The main advantages of Keras include it being open-source library that is rapid evolving. Keras has various back-end tools for industrial companies. Keras allows programmers to easily design and define DL models in a simple way. Additionally, Keras has popular APIs for DL development.

Despite its simplicity, The modular (or sequential) design is not optimal for developing new novel architectures.

2.2.7.3 PyTorch

PyTorch is a Python library for DL models developed on GPU-accelerated and general-purpose computers. This library supports tensor computation with strong GPU acceleration. PyTorch allows the user to build complex architectural models easily. It uses a method known as “reverse-mode auto-differentiation” to alter the way the network behaves with little effort. The main advantages of PyTorch are given below [41].

The main benefits of PyTorch include its support for dynamic computational graphs, automatic differentiation for NumPy and SciPy libraries, and Open Neural Network Exchange (ONNX) to easily transform models.

However, PyTorch does not have a visualization tool for monitoring and developing the model graph. Moreover, since PyTorch is not an end-to-end platform for ML development, this necessitates the translation from PyTorch to other DL frameworks.

2.3 Speech Datasets

Speech databases and datasets can be used in a variety of applications ranging from Automatic Speech Recognition and synthesis to language and speaker identification.

This section gives an overview on the most popular Arabic speech recognition datasets that are publicly available, while highlighting their main characteristics; this gives us an opportunity to contrast our dataset with the ones currently available. Additionally, we discuss available datasets in other languages as well as large datasets used in Data Centers.

2.3.1 *Arabic Datasets*

1. **Modern Standard Arabic:** this open-source free Arabic Speech corpus was recorded in a professional studio by many speakers in different dialects (Gulf, Levant, Egypt, Iraq and North-west Africa). It contains more than 3.7 hours’ worth of speech along with their phonetic and orthographic transcriptions. 13 females and 10

males mostly between the ages of 18 and 35 participated in this study. This dataset was collected and used as part of an application to create a speech synthesizer [42].

2. **Arabic Speech Command Database:** the collected database comprises of six Arabic control words as well as the Arabic digits (0-9) that were recorded from 100 volunteers using a mobile phone application with little to no background noise. The participants were each asked to read the six specified words and digits. The database comprised of 1600 recordings of 16 balanced classes sampled at 48,000 Hz. The dataset was used to compare between three different ML models to build an ASR system in the Libyan dialect [43].
3. **Multi Dialect Arabic Speech Parallel Corpora:** this dataset was designed to include four main dialects: MSA, Gulf, Egypt, and Levantine. The database consisted of recordings, which were about 32 speech hours and a total of 67,132 speech audio files. 52 participants (94% males and 6% females), between the ages of 16 and 30, were asked to speak sentences that were specific to the linguistic domain of travel and tourism. The recordings were taken in a quiet room using the Blue Yeti microphone and sampled at 48,000 Hz. This dataset can be used to translate sentences into different dialects as well as research the different characteristics of each dialect [44].
4. **King Abdulaziz City for Science & Technology Arabic Phonetics Database:** this database was released as a part of an experiment that aimed to study the airflow, air pressure, nasality, perception, as well as facial images and images of the glottis, etc. to help in in speech therapy, perception, synthesis, modeling, and recognition applications. The data collected from seven native Arabic speakers (average age of 32 years) were articulatory, acoustic, and perceptual information while pronouncing the 28 Arabic consonants. The database contains more 46,000 files recorded by CSL (Computerized Speech Laboratory, Model: 4300B) [45].
5. **Single Speaker Arabic Corpus:** this corpus is that of a single speaker pronouncing the Arabic corpus along with the recorded electroglottographic signals. The dataset comprised of seven hours of audio recordings. However, the researchers did not provide more metadata or how the audio signals were acquired [44].
6. **Algerian Arabic Speech Database:** the presented database comprises of different Arabic phonemes utterances pronounced by 300 Algerian speakers, in a quiet environment, from different Algerian regions. The speakers were of ages 18 and above, from different socioeconomic backgrounds, and were almost divided equally between female and male. The 1080 wave files were sampled at a rate of 16 KHz. This database was published in the intention of utilizing it for various NLP applications such as ASR and acoustic phonetic analysis [46].
7. **Basic Arabic Vocal Emotions Dataset:** this dataset contains audio recording of seven Arabic words spelled in different levels of emotions. 61 participants took part in this study, where 45 were males and 16 were females and at total of 1935 recordings were acquired. This database was used in applications regarding emotion recognition in the Arabic speeches [47].
8. **Arabic voice pathology database:** is a database utilized to assess and diagnose various voice disorders in the Arab region. The speakers were asked to utter three

different types of text: vowels, isolated words, and running speech while being recorded in a voice treated room using a Computerized Speech Lab model 4500. The database consisted of 366 samples (51 normal and 49 pathological) sampled at 48 KHz. Around 60% of the subjects are males and 40% are females and are within 10 and 60 years of age [48].

2.3.2 *English Datasets*

1. **Speech Commands:** this is a dataset for limited vocabulary speech recognition in the English language for 35 words. The audio recordings were collected using a web-based application, where a total of 105,829 utterances were acquired from 2,618 speakers. Background noise was added to the signals to mimic real case scenarios. The database aimed to build and compare between different ML models for speech recognition systems [49].
2. **AudioMNIST Dataset:** the open-source dataset consists of 30,000 audio recordings of 60 speakers with 50 repetition of the English digits (0-9). The audio files were recorded in quiet offices using RØDE NT-USB microphones. The participants, who were 12 females and 48 males, were within the age group of 22 and 61 years of age. This dataset was used in several ASR applications and classification tasks [50].

2.3.3 *Large Data Sets*

This section discusses speech processing models that are used in Data Centers (DCs). The computational/storage needs for training/inferencing and continuously training of these types of models is addressed; in addition, the datasets that are most used in such systems are discussed.

Recently, the rapid development of Internet of Things (IoT) devices and the shift to cloud-based systems from consumer-side computing has aided the growth of modern and large-scale DCs (i.e., customer service). Given the significant technological improvements in the hardware components and the growth of the Big Data field, DCs have started to include personalized user experiences and minimal downtime. These systems are deployed on cloud-based platforms given the complex plant operations needed for these systems.

The field of ML has been easily integrated in the DCs environment given the need of complex computations and the continuous need to monitor data. When it comes to training ML models for DCs, real-world historical datasets are needed; additionally, big data or large amounts of data are needed in this case. To illustrate, hundreds of hours of calls received to DCs, as well as the status of calls, call times, transit times, service time, and finish time are usually needed to build complex ML models [51]. Thus, only certain libraries and frameworks can be used to process this large amount of data and store it. To exemplify, Apache Hadoop, an open-source MapReduce model, has been recently developed as a solution for big data processing and storage needs. It offers many key features such as fault tolerance, automatic parallelization, scalability and data locality-based optimizations. As for the storage, traditional database systems cannot be used; instead, NoSQL database management systems offer a solution for large data by presenting a distributed soliton, consistency, availability, and partition-tolerance characteristics [52]. Furthermore, in order to properly train ML algorithms for these systems, continuous time-evolving features should be extracted [53]. Examples of popular datasets that contain hours' worth of audio recordings are listed below.

2.3.3.1 English Datasets

1. **Earnings21:** is a transcribed speech dataset licensed by the CC-BY-SA comprised of 39 hours of public companies' earning calls. The dataset creates training data by using forced alignment of existing audio against transcripts. This dataset is intended for named entity recognition applications for industry-specific terminologies [54].
2. **Librispeech:** is a publicly available dataset licensed by the CC-BY. It is made up of a 1,000 hours' worth of audiobooks recordings of the LibriVox project. The audiobooks are read by one single speaker in a noise-free environment [54].
3. **Gigaspeech:** this dataset is similar to the Earnings21 dataset where it utilizes forced alignment of audio extracted from YouTube videos to generate data. The dataset consists of 10,000 hours of English audio speech that is not publicly available due to copyright reasons [54].
4. **AMI Meeting Corpus:** is an open-source multi-modal dataset comprising of 100 meeting hour recordings. The meetings recorded are those of a design team trying to kick start a design project; some of the meetings are naturally occurring and others are scripted. The audio data was recorded using different devices such as close-talking and far-field microphones, and video cameras. Additionally, the videos were annotated and transcribed using orthographic transcription [55].

2.3.3.2 Multilingual Datasets

1. **Multilingual Spoken Words Corpus:** is a large open-source free dataset comprising of more than 340,000 words in 50 languages for one-second spoken examples. The dataset was generated using forced alignment on crowd-sourced phrases. This dataset is meant to be used in applications such as voice-enabled consumer devices and automation of call center conversations [56].
2. **Common Voice:** is a crowd-source dataset used to train speech enabled-applications. It contains more than 11,192 validated hours in 76 languages including Arabic, in addition to the metadata of the speakers. The acquired audio data was collected at a sampling rate of 48 kHz and re-sampled at 16 kHz [57].
3. **Multilingual LibriSpeech:** this dataset is a large multilingual dataset collected from LibriVox audio books on eight different languages. The dataset contains more than 50K hours of audio recordings in total [58].

After training these ML models to help automate customer care management, feedback learning or continuous learning is needed to keep improving the algorithm. This adds more complexity to the inference model given that it continuously needs to take in new data, train and learn from it to improve [59].

To sum it up, when it comes to ML models that deal with customer care management in large DCs big data is needed to train the algorithms using time-evolving features. Additionally, complex computations and large storage is a necessity. When deployed, these models need feedback and continuous learning to improve. However, in this study, given that an ASR system will be deployed on an edge device, the components are different.

In fact, even though ML models require large data to be trained, in this study big data is not needed and traditional database systems were used. Moreover, in this system time stamped data is used unlike the time-evolving features needed in the other systems. On the other hand, both systems can benefit from continuous training and feedback loops to enhance the system's performance.

2.4 Embedded Systems

The first part of this section provides an overview on embedded systems by comparing the technical specifications of three different Single Board Components. The second section discusses digital microphones and their digital audio standard.

2.4.1 *Single Board Computers and Microcontrollers*

Over the past years, the interest in Single Board Components has increased in the field of engineering due to its rather important and enhanced technical specifications. SBCs are hardware platforms, similar to microcontrollers, that function similar to general-purpose computers and whose main components are integrated on single SoC. SBCs include high memory capacity chips (might reach Gigas) both Random-access memory (RAM) and Flash technology as well as high-capacity microprocessors (32 bits and 64 bits) on a single chip. Additionally, SBCs support a large number of peripheral controllers such as graphical processors, interfacing protocols, audio, and various sensors [60]. The main advantage of SBCs is its low cost and low power consumption, which allows these components to be deployed in cases where the use of a standard PC is not suitable, yet the processing power needed is not met by a regular microcontroller [61].

The following three subsections discuss the technical specifications and the performance of three main SBCs: the Raspberry Pi, the ESP32, and the STM32 board.

2.4.1.1 Raspberry Pi

Raspberry Pi is a compact yet powerful and budget-friendly computer board that was first introduced in 2012 by the United Kingdom Raspberry Pi Foundation. This platform has a size of 85.6 x 53.98 x 17 (mm) and weighs approximately 45 g. The Raspberry Pi costs around 25-35\$ and is able to perform various types of computing while providing the option to interface with other devices via General Purpose Input/Output. This SBC supports numerous programming languages including Python, C, C++, Perl, and Ruby [62].

The Raspberry Pi uses its own Operating System (OS) called the Raspbian, which is the derivative of Linux, as a free open-source platform. Recently, non-Linux based OS have emerged in the market. However, the most preferred Raspberry Pi OS (formerly known as Raspbian) are the Linux distribution (distro) such as Debian, Puppy Linux, Arch Linux, and Fedora Remix since they are easily available, free of cost and, most importantly their capability to operate on the Pi's ARM processor. The main programming language of Raspberry Pi is Python, which is already loaded with its OS.

2.4.1.2 Arduino Nano 33 BLE Sense

The Arduino Nano 33 BLE is a small, low-power development board based on the Nordic nRF52840 Bluetooth 5.0 SoC. It measures just 45 x 18 mm and typically costs between

\$20-\$30. The Nano 33 BLE includes Bluetooth 5.0 connectivity, which enables it to communicate wirelessly with other Bluetooth-enabled devices such as smartphones and tablets. It also includes a range of sensors, including a microphone, an accelerometer, a gyroscope, and a magnetometer [63].

In addition to its connectivity and sensor capabilities, the Nano 33 BLE includes a range of I/O pins that enable it to interface with other electronic components and devices. These include digital and analog inputs and outputs, as well as interfaces for Inter-Integrated Circuit (I2C), Serial Peripheral Interface (SPI), and Universal Asynchronous Receiver/Transmitter (UART) communication protocols.

As for the operating system, the Arduino Nano 33 BLE is compatible with a wide range of operating systems, including Windows, macOS, and Linux. The board is programmed using the Arduino IDE, which is an open-source integrated development environment that is available for free on all major operating systems. The IDE provides an intuitive interface for writing, compiling, and uploading code to the board.

2.4.1.3 ESP32

The ESP32 is a powerful low power and low cost SoC series of microcontrollers that is designed for IoT applications and embedded systems. This microcontroller has integrated WiFi and Bluetooth as well as several peripheral connections. The board has a package size of 6 mm×6 mm and costs around 30\$ [64].

The ESP32 microcontroller supports the Mongoose OS, Zephyr, and other open-source OS for IoT and embedded applications. The most commonly programming environments for the ESP32 include Arduino Integrated Development Environment (IDE), PlatformIO IDE, LUA, MicroPython, and JavaScript.

2.4.1.4 STM32

The STM32 is a 32-bit microcontroller introduced by STMicroelectronics based on the ARM Cortex-M3 core, for embedded applications that require low cost and low power consumption. The STM32 microcontrollers are mostly used in power electronic systems such as system control, motor drives, programming controllers, medical devices, etc. [65].

The STM32 microcontroller has several OS that meet the real-time control requirements such as the μ Clinux, μ C/OS-II, eCos, and FreeRTOS. As for the programming environment, the STM32 has its own open-source IDE known as the TM32CubeIDE, which allows compilation, debugging, and feature reporting [66].

2.4.1.5 Comparison

Table 2.2 gives a direct comparison between the technical specifications of the Raspberry Pi, Arduino Nano, ESP32, and STM32 SBCs.

There have been many efforts to introduce real time speech recognition systems on mobile or edge devices in resource constrained environments [67]–[69]. Still, most of these systems are developed for recognizing English words; very few research has been done on the possibility of developing real time ASR systems for isolated Arabic words on low power microcontrollers. Thus, this research proposal aims to develop and deploy a ML system for the limited-vocabulary speech processing applications in the Arabic language on SBCs such as the Raspberry Pi, ESP32, or STM32 microcontrollers.

Table 2.2: Comparison between the Technical Specifications of Different SBCs

Single Board Component	Processor	RAM	Bluetooth	Wi-Fi	Storage	Power Supply
Raspberry Pi 4 Model B	Broadcom BCM2711, Quad core Cortex A72 (ARM v8) 64-bit SoC @1.5GHz	1 GB Low-Power Double Data Rate (LPDDR4) SDRAM	Bluetooth 5.0	2.4 GHz and 5.0 GHz IEEE 802.11ac wireless	MicroSD Card Slot	5.1V - 3A
Arduino Nano 33 BLE Sense	nRF52840 (Cortex-M4F) operates at 64 MHz	256 KB RAM	Bluetooth Low Energy	Requires an external WiFi module	1 MB Flash memory	4.5V to 5.5V - 50 mA
ESP32-S3 Series	Xtensa® 32-bit LX7 CPU operates at up to 240 MHz	512 KB SRAM 384 KB ROM on the chip	Bluetooth 5 Bluetooth mesh	IEEE 802.11 b/g/n-compliant	4 MB of internal flash memory	2.2V to 3.6V - 600 mA
STM32	ARM Cortex-M3 operates at 120 MHz /150 DMIPS	192 KB SRAM 64 KB core-coupled memory	Requires the addition of a Wi-Fi/Bluetooth module		2 MB of internal flash memory	2V to 3.6V -12.7 mA

2.4.2 Digital Microphones

Microphones are transducers that are able to convert sound, which is an acoustic pressure wave, to electrical signals. Over the past several years, the integration of sensors with components in the audio signal chain as well as the development of MEMS technology has made microphones more available and more compact having either digital or analog outputs. The main feature of digital microphones is integrating the analog-to-digital conversion function into the microphones, which enables an all-digital audio capture path starting from the microphone to the processor.

2.4.2.1 MEMS Microphones

The main component of a MEMS microphone is its transducer element. Essentially, the transducer is a variable capacitor of high output impedance (in gigaohms). The output of the capacitive transducer is sent to a pre-amplifier in order to decrease the output to a certain more usable range. In digital MEMS microphones, the amplifier is integrated with an Analog-to-Digital Converter (ADC) to convert the signal into a digital output in two different formats: Pulse Density Modulated (PDM) or I2S. I2S is a digital standard for transferring audio data whether it be mono or stereo [70]; a typical I2S-output digital microphone is shown in Figure 2.6. The frequency response of the MEMS microphone is shown in Figure 2.7.

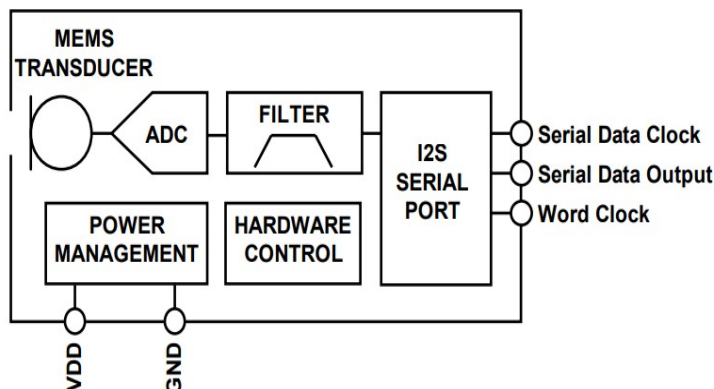


Figure 2.6: I2S MEMS Microphone Block Diagram. Adapted from [70]

2.4.2.2 The I2S Digital Audio Standard

I2S is a serial bus interface that was introduced to allow the communication between digital audio data between integrated circuits (ICs). I2S is a digital interface commonly used in audio converters and processors; however, only recently has this interface been integrated into audio devices such as digital microphones.

An I2S microphone outputs digital data at a decimated base-band audio sample rate. Given that in an I2S microphone, the decimation occurs in the microphone itself, the need for an ADC is eliminated. The I2S protocol is responsible for sending pulse-code modulation (PCM) audio data from a controller to a specific target. It has three main lines: the bit clock (SCK), the word select (WS), and the serial data line (SD).

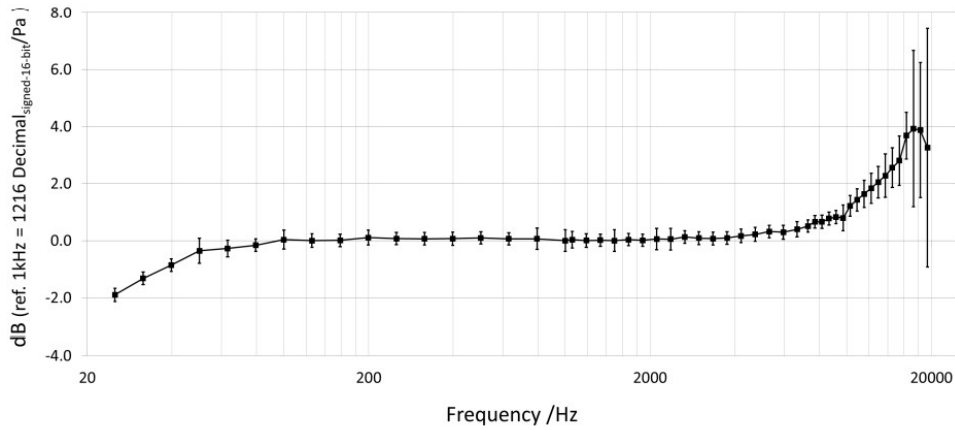


Figure 2.7: Frequency Response of the Digital MEMS Microphone. Adapted from [71]

As seen in Figure 2.8, the SCK signal runs continuously, while digital data is being sent on the SD line (Most Significant Bit (MSB) first). Usually, data is clocked out on the falling edge and clocked it in on the rising edge. On the other hand, a logic low on the WS line indicates that the word being transmitted is streamed for the left audio channel, while a logic high indicates right-channel audio.

Additionally, an I2S microphone can be easily connected to a microcontroller or edge device for various audio processing applications. I2S microphones can be connected to a common line and use two clock signals, as well as a word clock and a bit clock [70].

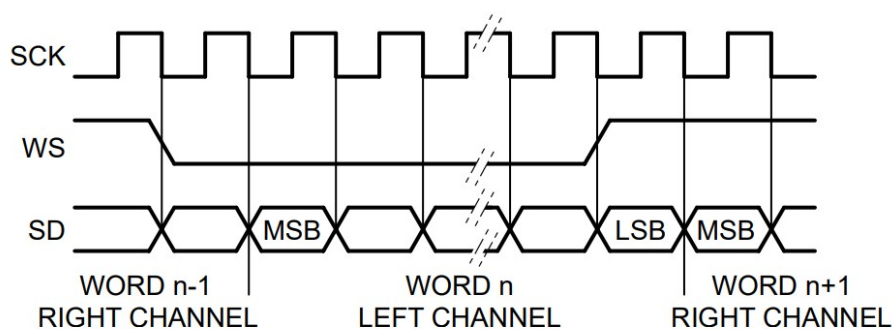


Figure 2.8: Timing Diagram showing an I2S Transaction. Adapted from [72]

CHAPTER 3

OVERVIEW OF TINYML

Tiny Machine Learning, or TinyML, is a rapidly evolving field at the intersection of machine learning and embedded systems. It enables the processing of deep learning algorithms of high accuracy on microcontrollers (MCUs) by leveraging power-constrained embedded platforms in applications that require low inference time and have limited communication bandwidth. TinyML enables compressed ML models to be deployed on resource-constrained and communication-limited microcontrollers without compromising accuracy [73], [74]. This offers several advantages over deploying ML models in the cloud, including:

1. **Lower Inference Time:** Deploying and running models locally eliminates the overhead of transferring data to, and results from, the cloud. It also eliminates the added inference time of queueing and scheduling inference tasks in the cloud. In interactive or safety-critical applications, low-inference time is a key advantage.
2. **Less Data Exchange:** By running inference locally, only results with high information content need be shared with higher-level functions in the cloud. This greatly reduces the volume of data that needs to be exchanged and saves communication bandwidth.
3. **Lower Cost and Energy Consumption:** Because they can run on inexpensive and low-power microcontrollers, TinyML models are more cost- and energy-efficient than their cloud counterparts. This enables them to run for months, and even years, using small batteries or energy harvesting circuits.
4. **Higher Privacy and Security:** For data-sensitive applications, processing data locally enhances its privacy. It also makes it easier to employ adequate measures to secure the data including running the models offline.

3.1 TinyML Frameworks

Several Integrated Development Environment (IDE) are available for generating TinyML models. These include Edge Impulse, Microsoft’s Embedded Learning Library (ELL), ST Microelectronics’ STM32Cube.AI, and ARM’s ARM-NN SDK [75]–[77]. These convert models developed using common ML frameworks like Keras, Open Neural Network Exchange (ONNX), or TensorFlow into quantized and machine optimized models expressed in a high-level language. However, because they target specific microcontrollers, hardware

dependencies limit the scope of optimizations that can be applied to the models. The following subsections describe the most popular TinyML platforms that are currently in use.

3.1.1 *TensorFlow Lite*

Google’s *Tensorflow Lite for Microcontrollers* is another platform for generating TinyML models. It consists of a *converter* that transforms Tensorflow Lite models into serialized FlatBuffer models. The FlatBuffers serialization library uses schemas to transform hierarchical data structures into flat binary buffers implemented as C++ arrays. The buffers are smaller than the original data structures, and they can be accessed without the need to parse or unpack data. This makes them especially well suited for use in bare metal embedded systems. TFLM also includes an *interpreter* that uses code generated from the FlatBuffers schema to access data efficiently. It also abstracts the underlying hardware, making it easy to port a TinyML model to different microcontrollers and apply hardware-independent optimizations to a model [78]. The work we present in this proposal is based on the TFLM platform.

3.1.2 *Edge Impulse*

The Edge Impulse (EI) platform provides a comprehensive set of tools for building, testing, and deploying machine learning models on edge devices. The platform’s web-based interface allows users to easily upload and label their own data, train machine learning models, and deploy them to edge devices. Once the models are trained, EI provides optimized libraries for running them on a wide range of edge devices, including microcontrollers, embedded processors, and smartphones. These libraries are designed to be memory and processing efficient, enabling real-time inference on resource-constrained devices [79].

One of the most powerful features of Edge Impulse is its ability to generate optimized C code directly from machine learning models trained on the platform. This process is made possible through the use of the Enterprise Objects Framework (EON) Compiler, which takes a trained model and generates C code that implements the model’s inference algorithm.

To generate C code from a trained model in Edge Impulse, the user must first export the model from the platform. This involves selecting the desired model and clicking the “Export” button in the Edge Impulse web interface. EI supports various machine learning frameworks, such as TensorFlow, Keras, and PyTorch, which can be exported in a standardized format such as ONNX or TensorFlow Lite. Once the model is exported, it can be imported into the EON Compiler, which will generate C code that can be deployed directly to edge devices. The generated code is optimized for memory and processing efficiency, and is designed to run on a specific hardware platform. This enables developers to build intelligent applications that can run directly on edge devices, without the need for additional libraries or runtime environments.

3.1.3 *Embedded Learning Library*

Embedded Learning Library (ELL) is an open-source framework that was launched by Microsoft to help deploy ML algorithms on numerous MCUs that support ARM Cortex-A and Cortex-M architectures, such as the Arduino and Raspberry Pi. ELL is basically

a cross-compiler that takes in a model generated in a certain format (i.e., OpenNeural Network Exchange or TensorFlow) as an input and outputs a compressed model in the form of an executable code for edge devices [78].

3.1.4 *ARM-NN*

ARM-NN is another open-source Linux framework that was developed by Arm to help inference ML algorithms on edge devices. This framework uses the Common Microcontroller Software Interface Standard NN library to deploy deep architectures on MCUS; this library has efficient neural networks kernels to help integrate ML on Cortex-M processor cores. Given that this library is tailored to embedded devices it utilizes fixed-point arithmetic to perform parameter quantization. The model’s parameters are reduced to 8 or 16 bits and then the compressed model is deployed on MCUs [78].

3.1.5 *STM32 Cube AI Library*

STMicroelectronics has introduced specific toolkits and libraries for its own devices; the STM32Cube.AI is an example on one of these toolkits. The STM32Cube.AI allows the deployment of pre-trained NN, provided by either TensorFlow or Keras, on STM32 ARM CortexM microcontrollers. It generates from the NN models a C code compatible with STM32 devices or a code in the standard ONNX format. The STM32Cube.AI is able to deploy large NNs by storing the models’ weights and activation buffers in the RAM or external flash memory of the MCU [80].

3.2 TinyML for Speech Recognition – Related Work

Speech recognition is a rapidly evolving field that has seen significant advancements in recent years, largely due to the development of powerful machine learning algorithms and the availability of large amounts of data. One of the most exciting areas of progress in speech recognition is the ability to deploy these algorithms on edge devices, such as smartphones, smart speakers, and wearables. This allows for faster and more efficient speech recognition, with the added benefit of increased privacy and security. In this section, we will explore the various applications of speech recognition on edge devices and discuss the results achieved by various studies.

Tsai et al. [81] proposed a new keyword spotting technique utilizing convolutional neural networks. The technique is based on densely connected convolutional networks and involves replacing normal convolution with group convolution and depthwise separable convolution to reduce model size. Additionally, squeeze-and-excitation networks are added to enhance the weight of important features, leading to increased accuracy. Two models were built to investigate the effect of different convolutions on DenseNet: SpDenseNet and SpDenseNet-L. The Google speech commands dataset was used to validate the network, which achieved better accuracy than other networks, with fewer parameters and floating-point operations (FLOPs). For instance, SpDenseNet achieved an accuracy of 96.3% with 122.63 K trainable parameters and 142.7 M FLOPs, using only about 52% of the number of parameters and 12% of the FLOPs compared to benchmark works. Furthermore, the authors varied the depth and width of the network to create a compact variant, which also outperformed other compact variants. SpDenseNet-L-narrow, for instance, achieved an accuracy of 93.6% with 9.27 K trainable parameters and 3.47 M FLOPs, using only

about 47% of the number of parameters and 48% of the FLOPS compared to benchmark works. These results suggest that the proposed technique can improve the accuracy of keyword spotting while reducing the number of parameters and FLOPs.

A study proposed a small-footprint Keyword Spotting (KWS) system running on an STM32F7 microcontroller with a Cortex-M7 core @216MHz and 512KB static RAM. The baseline CNN model achieved a validation accuracy of 90% on the Speech Command Data Set v0.01 and generated classification results every 37ms, including real-time audio feature extraction. The study evaluated the performance of different pruning and quantization methods on the microcontroller, identifying challenges for accelerating unstructured pruned models and suggesting that structured pruning is more suitable than unstructured pruning. The study also demonstrated that quantization and SIMD instruction can improve the system’s performance. The authors presented the time and power consumption for four different configurations, showing that quantization from float32 to int16 reduced elapsed time from 30.8ms to 21.4ms, and memory footprint was halved. Finally, the authors found that skipping zero weights under normal loop order was counterproductive, while skipping under weight-prioritized loop order condition was still beneficial when the pruning percentage exceeded 80% [82].

Another study showcases an effective method of implementing Deep Neural Network models on edge devices for keyword spotting tasks. The authors tested their approach on a bare-metal embedded device (microcontroller) and a single board computer (Jetson Nano), and demonstrated its effectiveness. The authors removed unnecessary audio components and noise from the samples and used Mel-Frequency Cepstral Coefficient to extract speech features. They proposed a Depthwise Separable Convolutional Neural Network model with about 721 thousand trainable parameters. After training, the converted model used only 11.52 Kbyte of RAM and 169.63 Kbyte of flash memory for the bare-metal implementation. The authors achieved a prediction accuracy of 91% for the keyword spotting task. The bare-metal implementation of the model executed a complete prediction in 7ms, while the JetBot took about 15ms for a single prediction, indicating the execution speed can depend on the hardware used [83].

Holzke et al. [84], adapted a Depthwise Separable Convolutional Neural Network and a Convolutional Neural Network for keyword spotting and used augmented training data, including real industrial noise, to improve their robustness. They also applied post-training quantization and tested the performance of both networks on multiple embedded systems, including a Google Edge TPU. Through a systematic analysis of accuracies, memory footprint, and inference times using different combinations of data augmentations, hardware platforms, and quantizations, they found that augmented training data improved the inference accuracy in noisy environments by up to 20%. The authors demonstrated that using an integer quantized network with a memory footprint of 0.57 MByte, achieving inference speeds of less than 5 ms on an embedded CPU and less than 1 ms on the Edge TPU.

A recent research paper proposed a TinyML application that classifies words (yes and no) found in audio recordings. Pre-processing of the raw audio signals was done prior to any step; first, the Fast Fourier Transform (FFT) of the audio signal was computed using the Hann window, after which the Mel-frequency scaling was applied to average the adjacent frequencies into a down sampled array. The spectrogram of the sound signal was then computed as an input to train the ML model. The researchers then designed and implemented a custom I2S module to perform the Digital Signal Processing stage on the edge device; this was done by adding a windowing module to the unit in order to

reduce operation time and energy consumption. After changing design parameters such as coefficient length and quantization degree, the researchers were able to minimize the hardware size of the designed I2S unit. The proposed system was executed on a system based on the ARM Cortex-M4 32-bit microcontroller. As for the ML model, a three layer fully connected neural network was trained on an open-source dataset comprising of a 100,00 one second wave files [85].

Another study evaluated the accuracy, performance, and computational efficiency of compressed ML model on two edge devices for ASR. On one hand, by applying quantization and PyTorch mobile optimizations in the case of Raspberry Pi CPU inferencing, an improvement of inference time by 10% was seen, as well as a reduction in the memory footprint by 50% at the cost of an increase in the Word Error Rate by 0.5% compared to the original model. On the other hand, after running the inference in the Jetson Nano GPU, the inference time was improved by a factor of three to five when compared to Raspberry Pi. Furthermore, the system had a load time of one to two seconds, a memory footprint of 300 MB, and a real time factor (a metric that measures the speed of an ASR system) less than one. The implemented system proved to be reliable, secure, and always available in ASR applications [86].

A study done by Hardy & Badets, developed a low-power RNN based classifier for an always-on Wake-Up Sensor application. The system consists of a microphone, a Low Noise Amplifier, an Automatic Gain Control that automatically adjusts the gain of the unit, and filters. The output of the microphone is converted into a spectrogram and then analyzed to detect if the subject has started speaking. The NN was trained on a speech dataset comprising of 1,536 speech recordings from the Speech Commands dataset [49] and noise segments from the MUSAN (Music, Speech, and Noise) corpus [87]. In order to deploy the algorithm on an edge device, the RNN model used, which was composed of 16 recurrent units, was quantized using tanh for bounding the weight excursion and the incremental quantization. The finalized model achieved 20 False Triggers/hour in noise at 3% No Trigger Rate in pooled conditions; moreover, its memory footprint was 0.52 kB and had an estimated power consumption of 45 nW [88].

A Sparse Embodiment Neural-Statistical Architecture was applied in a study to perform isolated speech recognition using Sparse Pulse Automata via Reproducing Kernel method. The proposed method starts by developing an ML model of a dynamical system using pulse trains, which is a technique inspired by neuromorphic engineering. Then, a rule-based solution such as lookup tables are computed to aid in the rapid deployment of the ML algorithm on edge computing platforms. The researchers used a unified theoretical framework of the Reproducing Kernel Hilbert Space (RKHS) to obtain interpretable and nonlinear solutions when compared to traditional NN. The proposed architecture was trained and tested on the TI-46-digit corpus, which contained 4000 recordings (2,700 training and 1,300 testing) from an equal number of female and male participants. An accuracy of 93.54% was achieved using the 5-Network Kernel Adaptive Autoregressive Moving Average chain model with 12 input spike channels and a size of 1880 center x 5 network x 10 words [89].

Furthermore, a paper published by Yang et al. [90] proposed a compact speech recognition model using spatio-temporal features for edge deployment. The researchers developed a ML algorithm composed of 2 main models: a 1-Dimensional Convolutional Neural Network to process the spatial information of the frequency content of the acoustic features and an RNN model to process the temporal information of the frequency content of the same features. The attributes utilized in the study are the Mel spectrogram features and

other related features extracted from the Google’s Speech Commands Datasets V1 [49]. The utilized dataset, which consisted of 65,000 one second audio recordings of 11 keyword, was split in a in a ratio of 80:10:10 for training, validation, and testing. The model achieved an accuracy of 96.82% and when successfully deployed on the Raspberry Pi 3 module, a total inference time of 0.59 seconds was observed, as well as energy consumption and a power peak of 1.234 J and 2.7 W respectively.

Based on the conducted literature, the field of speech recognition on edge devices has several gaps that need to be addressed through comprehensive research. Firstly, existing studies only focus on specific techniques and models without offering a comprehensive comparison of different approaches and their respective advantages and disadvantages. Additionally, no previous research has explored the use of the Arabic language, indicating a need for further investigation in this area. Moreover, a lack of investigation into different compression techniques for various deep learning models and testing on several Single Board Computers exists, warranting further research.

In light of these gaps, our study aims to provide a comprehensive comparison of various approaches and models used in speech recognition on edge devices while focusing on the Arabic language, which has not been extensively studied in previous research. Furthermore, we will explore different compression techniques for various deep learning models and test them on several Single Board Computers. Through our research, we will provide valuable insights into the impact of accuracy, size, inference time, and energy consumption of the models, which could aid in optimizing speech recognition on edge devices.

3.3 Techniques for Model Compression

One of the most challenging parts of deploying ML algorithms, such as Neural Networks, are the software and hardware design requirements. When running complex ML algorithms on embedded devices, it is necessary to take into consideration the small memory size and short battery life of MCUs. In fact, the ML model, with all its parameters, weights, and connections, must be small enough to fit on MCUs that have constrained on-chip (SRAM) memory (192-512 KB) and flash memory (256 KB-2 MB). Moreover, when running TinyML on microprocessors, it is important to account for the computational cost and power cycle of the ML model given the limited battery life of the edge device.

To overcome the main challenges and limitations that of running TinyML on embedded platforms, several approaches can be taken such as model reduction and lightweight frameworks. The following subsections tackle the main methods that can be utilized to address TinyML complications.

TensorFlow Lite uses a number of techniques to compress machine learning models. For speech recognition applications, the optimizations are applied to *sequence models* such as recurrent neural networks or long short-term memory networks. In this section we introduce four common techniques for compressing ML models: quantization, pruning, weight clustering, and hybrid. In later sections we evaluate the impact of these techniques on the size, accuracy, and inference time of an LSTM model for recognizing spoken Arabic digits.

3.3.1 Quantization

Quantization is a technique that significantly reduces the memory footprint of a deep neural network without compromising the model’s accuracy [91]. It is based on reducing the number of bits used to represent a model’s parameters. Quantization usually replaces 32-bit floating-point values produced during training with 16-bit floating-point or 8-bit fixed-point values that are used during inference. In addition to reducing the amount of memory needed to store a model, quantized models can usually run faster because they use simpler integer arithmetic units to operate on data.

There are two main approaches to quantization. In *quantization-aware training* [92], expected quantization errors due to inference are used during training to learn quantized model parameter values. This technique is time consuming, but it reduces the size of a model and enhances its accuracy at the same time [91]. In *post-training quantization*, model parameter values are quantized after a model has been trained. This approach generates quantized models more quickly but generally achieves lower model accuracy. However, in some cases, higher quantization errors can help a model generalize better, which increases model accuracy over new data [91]. In this study, we explore three forms of post-training quantization:

3.3.1.1 Float16 Quantization

A model’s 32-bit floating-point weights and activations are converted to 16-bit, half-precision, IEEE floating-point values [93].

3.3.1.2 Dynamic Range Quantization

A model’s weights are converted from 32-bit floating-point to 8-bit fixed-point, but activations are stored as 32-bit floating point values. During inference, activations are quantized dynamically to 8-bit fixed-point values based on their dynamic ranges. This helps reduce model execution inference time [94].

3.3.1.3 Integer Quantization

A model’s weights and activations are converted to the nearest 8-bit fixed-point values. To guide quantization, representative data sets are used to drive the TensorFlow Lite converter and provide information about the dynamic ranges of different model parameters [94].

3.3.2 Pruning

Model pruning is an effective technique for compressing over-parameterized networks and reducing their storage and computational expense. It enables models to be deployed on resource-constrained devices without incurring a significant loss in accuracy. Pruning removes a model’s dispensable or redundant weights, filters, and other structures based on their magnitudes [95]. In magnitude-based pruning, model weights with the smallest magnitudes are incrementally set to zero until a certain level of sparsity is achieved. Magnitude-based pruning is applied after training. In this proposal we explore two forms of magnitude-based pruning:



Figure 3.1: Constant Sparsity Pruning

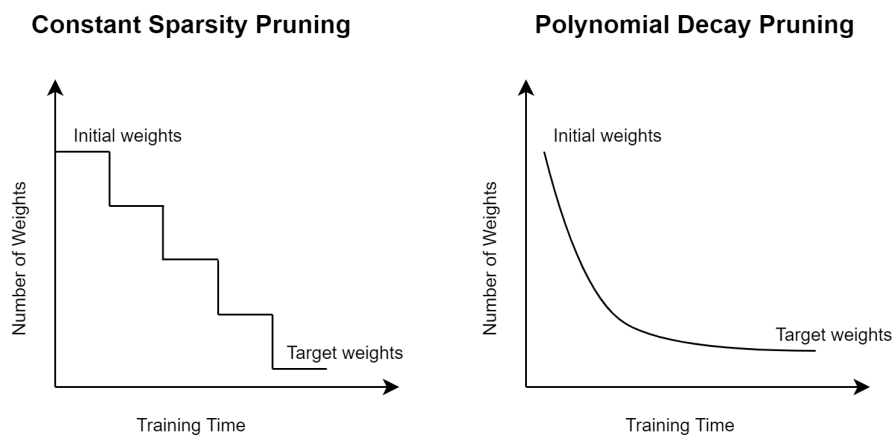


Figure 3.2: Constant Sparsity vs. Polynomial Decay Pruning

3.3.2.1 Constant Sparsity Pruning

The magnitudes of a model’s weights are set to zero to achieve a fixed sparsity level that is chosen before the model is trained [96]. Figure 3.1 shows how constant sparsity pruning is applied. First, a ML model must be built and trained. This results in an initial set of model weights. The model is then pruned and retrained repeatedly until the pre-set sparsity level is achieved. On every pruning and retraining iteration, a new subset of weights is set to zero. This effectively reduces the number of model weights incrementally. In this study we varied the sparsity level from 10% to 90% to demonstrate the impact of this technique on model size and performance. We also chose a learning rate of 0.001 to control the pruning schedule.

3.3.2.2 Polynomial Decay Pruning

The weights of the model are pruned gradually during conversion to enable the model under study to adapt to its pruned weights [96]. The model developer must set the desired percentage of sparsity at the start and end steps of the pruning process. The Tensorflow Lite converter then applies the specified pruning schedule to the model.

For this study, we set the start sparsity level to 0% and varied the end sparsity level from 10% to 90% incrementally. We also set the pruning schedule to start and end at 1000 and 2000 steps, respectively. A pruning step corresponds to processing a single batch. As with constant sparsity pruning, we chose these sparsity levels to study the effect of polynomial decay pruning on the size and performance of a model. Figure 3.2 provides a conceptual comparison of how constant sparsity pruning and polynomial decay pruning differ in their approach to model weight reduction.

3.3.3 *Weight Clustering*

Weight clustering is an important method in model compression, where similar weights of a model are clustered into a predefined number of groups/clusters and assigned the same value [97]. In this study, we use K-Means algorithm to cluster the model’s weights while varying the number of predefined clusters from 4 to 128 by power of 2 increments.

3.3.4 *Hybrid Approaches*

In an attempt to improve the performance of compressed ML models or further reduce the model’s memory footprint, many researchers combined compression algorithms. These of methods are known as hybrid or collaborative optimization methods [98]. We investigated two hybrid methods in this proposal: pruning and weight clustering followed by post-training quantization.

CHAPTER 4

A TINYML MODEL FOR SPOKEN ARABIC DIGIT RECOGNITION - METHODOLOGY

Developing a TinyML model begins by developing a regular ML model and optimizing it to meet desired size and performance objectives. Figure 4.1 summarizes the steps in a typical TinyML workflow. These includes collecting and pre-processing training data; selecting, training, and evaluating a suitable model; and compressing and deploying the model, after which inferences can be done on the edge.

In this chapter we detail the steps we followed to develop a TinyML model for recognizing spoken Arabic digits. We first describe the dataset used and the steps we followed to pre-process and clean the data. We then explain how we chose our ML model and discuss the experimental methods we followed to conduct our study.

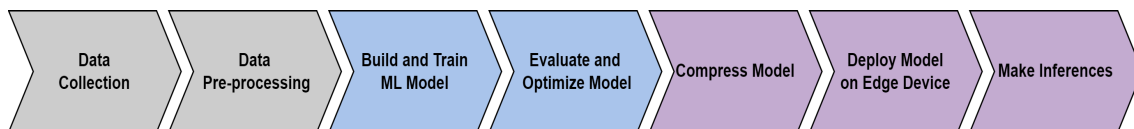


Figure 4.1: Typical TinyML Workflow

4.1 Levantine Arabic Audio Dataset

We collected our own dataset using a web interface [99], where participants were asked to record themselves saying different floor levels in their own personalized way in the Levantine dialect of the Arabic language. Our dataset includes 15 classes representing different floor levels, including B1 (basement level 1), B2 (basement level 2), GF (ground floor), and floors 1 through 12. Each participant was asked to record themselves saying each digit three separate times, with the recording set for a duration of 2 seconds. This approach was taken to create a dataset that accurately reflects the diversity of speech patterns and accents within the Levantine dialect, which is not well-represented in existing datasets. Each participant was encouraged to express themselves in their own unique way, which enabled us to gather a range of different speaking styles and variations (e.g.

some might said ”ع الطابق الأول” while others said ”ع الأول”). We opted for a web-based approach to data collection to make it easy and convenient for participants to contribute their recordings using their own devices without requiring any specialized equipment. By leveraging this approach, we were able to create a unique dataset that can help us gain a better understanding of how different speaking styles and accents impact speech recognition algorithms in the Levantine dialect.

The dataset included recordings from 159 participants who spoke the Levantine dialect of Arabic. The participants were 61 were males and 98 were females, mostly with an average age between 18 and 36. The participants were located across Mount Lebanon, Beirut, Akkar, Beqaa, South, and North Governorate. The participants were recruited from six different locations, with the largest number of participants coming from Beirut Governorate with 70 participants. Although the majority of participants were recruited from Beirut Governorate, it is important to note that the dataset is not biased towards their dialect. This is due to the fact that the participants living in Beirut have diverse accents and predominantly convey the typical Lebanese dialect.

To provide a visual representation of the demographics of the participants, Figures 4.2a, 4.2b, and 4.2c show the gender, age, and location distribution of the participants in the dataset, respectively.

By including this demographic information and providing a visual representation of the data, we can better understand any potential biases or limitations of the dataset, and can make informed decisions about how to analyze and interpret the results.

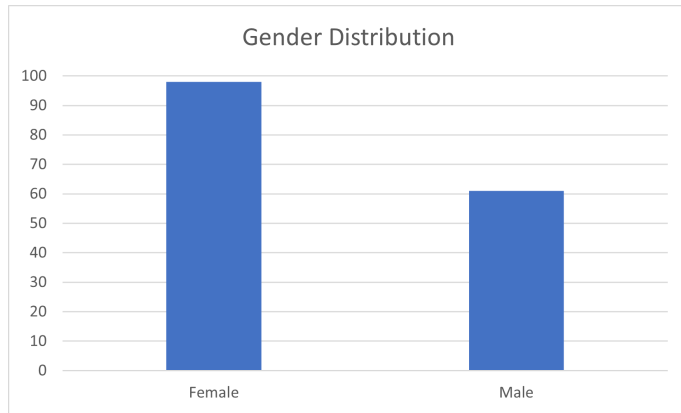
4.2 Data Pre-processing and Cleaning

Before using the dataset to develop and train a suitable ML model, we augmented, pre-processed, and cleaned the data. First, we performed data augmentation techniques such as pitch shifting and time stretching to increase the variability of the dataset. We then extracted Mel-frequency cepstral coefficients (MFCCs) from the audio samples to represent the spectral content of the speech signal. Finally, we normalized the MFCCs to have zero mean and unit variance across the dataset.

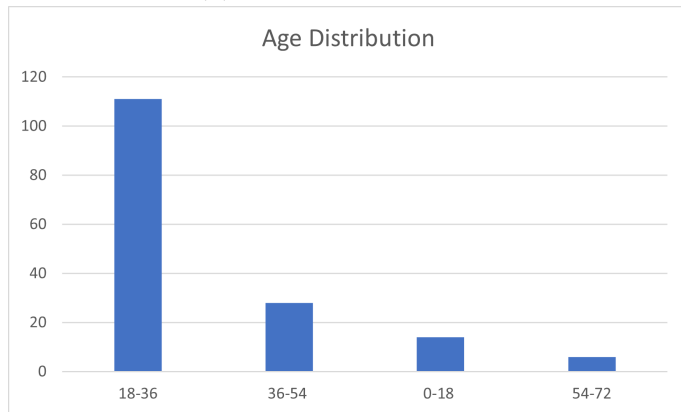
4.2.1 Data Augmentation

Audio data augmentation is a crucial technique to increase the size of the training dataset and improve the generalization performance of machine learning models. In this study, we employed several commonly used data augmentation methods to enhance the diversity and variability of our audio dataset.

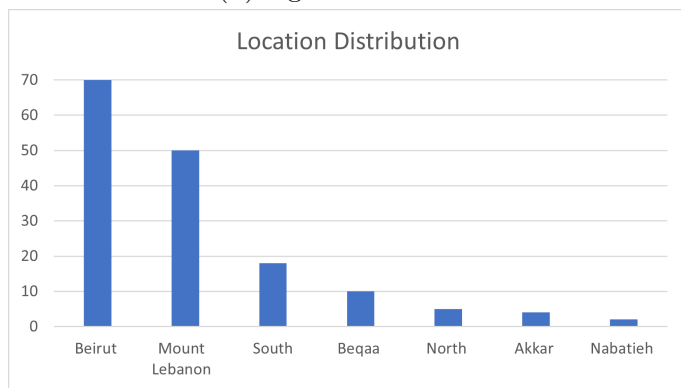
”First, we added white noise with a standard deviation (sigma) of 0.005 to each audio clip to simulate real-world background noise, and then calculated the signal-to-noise ratio (SNR) of the resulting audio signal. The SNR was found to be 22.97 dB, indicating that the signal was significantly stronger than the background noise. Next, we generated a random pitch shift factor between -3 and 3 semitones to change the pitch of the audio and create variations in tone, yet mimic the natural variation in human speech. We also applied time shift by 0.3, which randomly shifts the audio forwards or backwards by up to 0.3 seconds to create variations in timing. Additionally, we time stretched audio with stretch factor of 0.8 and 1.1, which changes the tempo of the audio while preserving the



(a) Gender Distribution



(b) Age Distribution



(c) Location Distribution

Figure 4.2: Demographic Distributions

pitch. To simulate different environments, we added elevator noise to some audio clips and chatter to others.

By applying these techniques, we increased the size and variability of our dataset from 1,941 to reach 17,469 recordings, which will improve the performance and robustness of the ML model.

4.2.2 Mel-Frequency Cepstral Coefficient Feature Extraction

A common approach to pre-processing speech samples is to extract their Mel-Frequency Cepstral Coefficients. The Mel frequency cepstrum is a method for capturing a speech sample's short-term power spectrum. MFCCs provide information about the spectral characteristics of the speech sample, enabling them to be used as input features to an ML model [100].

Figure 4.3 shows how a speech sample's MFCCs are computed. First, the speech sample's frequency spectrum is calculated using the Fast Fourier Transform. Next, relevant frequency bands are extracted by applying a Mel filter bank. This is a group of overlapping triangular filters with unit gains centered around the frequencies along the Mel scale. The Mel-scale is a non-linear frequency scale that mimics the non-linear perception of sound in the human ear, which is better at detecting low-frequency sounds. Equation 4.1 shows how sound frequencies are mapped to the Mel scale:

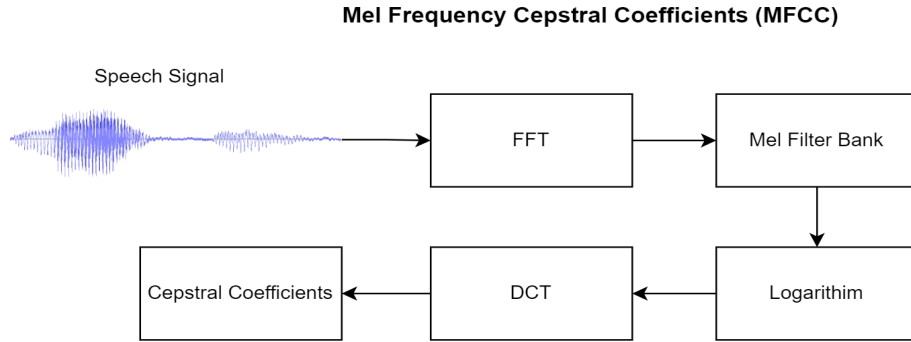


Figure 4.3: MFCC Extraction

$$\text{Mel}(\text{frequency}) = 1127 \times \ln\left(1 + \frac{\text{frequency}}{700}\right) \quad (4.1)$$

After applying the Mel filter bank, the logarithm of the power spectrum magnitude for each filter is calculated to also mimic the non-linear perception of loudness in human hearing. The power spectrum magnitude is obtained by taking the magnitude squared of the Fourier transform of a windowed segment of the audio signal.

Next, the discrete cosine transform (DCT) is applied to the logarithms of the Mel power spectrum magnitudes to decorrelate the energies of overlapping Mel filter banks. Finally, the results of the DCT are used to calculate the speech sample's MFCCs. However, only the first 12 cepstral values out of 26 (MFCCs 1 to 12) are used as speech features because they contain the most discriminative spectral information. Using a smaller set of MFCCs also reduces processing times [101].

The MFCC values for a given speech sample are expressed as a matrix with 12 columns corresponding to the calculated cepstral values and a number of rows corresponding to the duration of a sample. Because there are variations in the time it takes to utter different digits, the number of rows for MFCC matrices can vary. To ensure that the ML model will process all speech samples uniformly, we used the maximum number of rows to express all MFCC matrices, and we set to zero any unused rows associated with a speech sample. We also verified that setting unused rows in an MFCC matrix to zero did not change or corrupt the corresponding speech samples when transformed back to the time domain.

4.2.3 Normalization

After extracting the MFCC coefficients, we normalized them using Z-score normalization [102] so that all attributes can contribute to the learning process equally. Equation 4.2 shows how Z-score normalization is calculated:

$$v'_i = \frac{v_i - \mu}{\sigma} \quad (4.2)$$

Here v'_i is the normalized value and μ and σ are the mean and standard deviation of the i th column (cepstral value) of an MFCC matrix, respectively.

4.3 Automatic Speech Recognition Models

In this study, we evaluated four different deep learning architectures, which we developed using the Keras Tuner with Bayesian Optimization [103]. We assessed over 20 design iterations using the TensorFlow framework to build the optimal model architecture. The models we assessed were: (1) Convolutional Neural Networks, (2) Gated Recurrent Units (GRU), (3) Long Short-Term Memory, and (4) Bidirectional LSTM (Bi-LSTM). All the results reported in this study were after training our model on 70% of the data, out of which 10% was used for validation, and testing it on the remaining 30%. We also used an Adam optimizer, a categorical cross entropy loss function, and optimized for validation accuracy. During training we used a batch size of 32 on 100 epochs. To decide which algorithms are best suited for our application, we compared the accuracy of these four models on the mentioned dataset and selected the top three models. As seen in Figure 4.4, the top performing models were the GRU, LSTM, and Bi-LSTM.

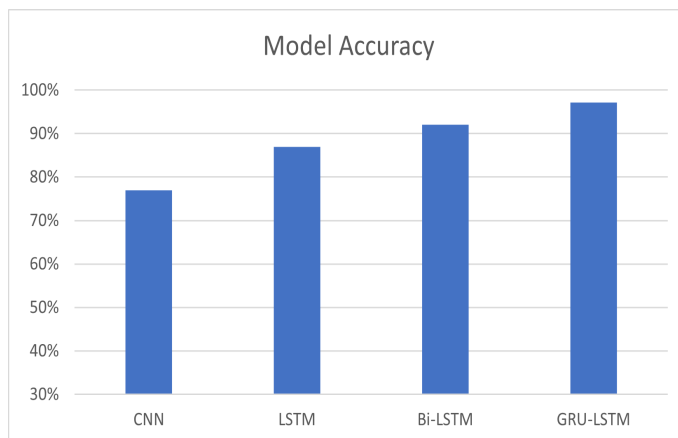


Figure 4.4: Model Accuracy on Testing Set

Figures 4.5, 4.6, 4.7, and 4.8 show the final structure of our models. Below is a detailed description of the models' architecture:

1. CNN: consists of two Conv1D layers with 8 and 16 filters, respectively, and a kernel size of 3. The ReLU activation function is used for both layers to introduce non-linearity in the model. A dropout rate of 0.25 is applied after each pooling layer to prevent overfitting. The max-pooling layer with a pool size of 2 and stride of

2 is applied after each convolutional layer, which reduces the spatial dimensions of the input by half. The output from the second pooling layer is flattened, and a softmax activation function is applied to the final dense layer with the number of neurons equal to the number of classes (15). The final dense layer with softmax activation produces a probability distribution over the classes, indicating the model's confidence in each of the possible classes.

2. LSTM: consists of an LSTM layer with 32 units, followed by a second LSTM layer also with 32 units, and a third LSTM layer with 256 units. A Flatten layer is added to convert the output tensor from the last LSTM layer to a 2D tensor. The model then has three Dense layers with 32 units each that use the relu activation function. Another Flatten layer is added, followed by a Dropout layer with a rate of 0.1 to help prevent overfitting. The final Dense layer has 15 nodes, and uses the softmax activation function to produce a probability distribution over the classes.
3. GRU-LSTM: consists of a GRU layer with 288 units, followed by an LSTM layer with 489 units. A Flatten layer is added to convert the output tensor from the last LSTM layer to a 2D tensor. The model then has a Dense layer with 96 units that uses the relu activation function. Another Flatten layer is added, followed by a Dropout layer with a rate of 0.1 to help prevent overfitting. The final Dense layer has 15 units, and uses the softmax activation function to produce a probability distribution over the classes.
4. Bi-LSTM: consists of a Bidirectional LSTM layer with 1024 output units, which allows for information to be passed both forward and backward through the sequence. This layer is followed by a Flatten layer, which converts the output tensor from the LSTM layer to a 2D tensor. Next, there are three Dense layers with 32, 512, and 512 units, respectively, which use the default linear activation function. Another Flatten layer is added to prepare the data for the final Dense layer with 15 units, which uses the softmax activation function to produce a probability distribution over the classes.

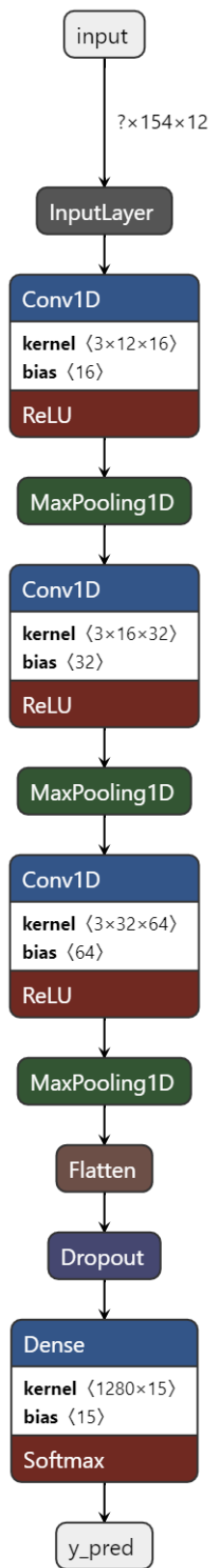


Figure 4.5: CNN Model

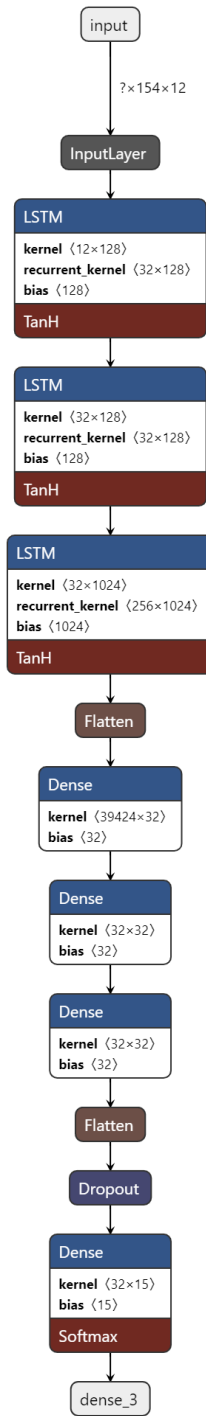


Figure 4.6: LSTM Model

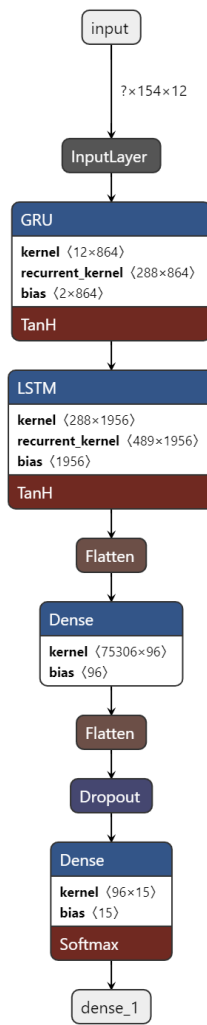


Figure 4.7: GRU-LSTM Model

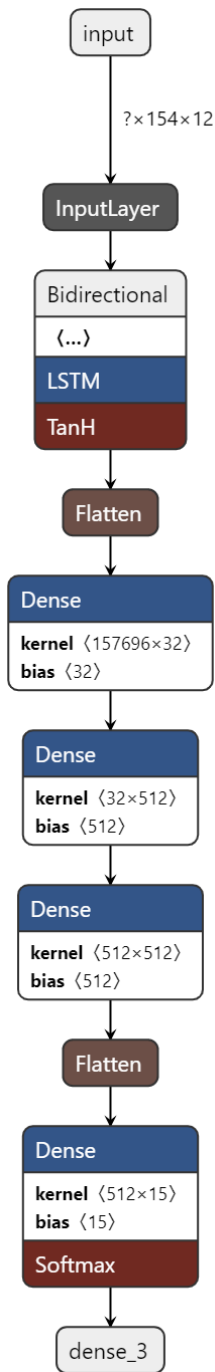


Figure 4.8: Bi-LSTM Model

CHAPTER 5

RESULTS AND DISCUSSION

In this chapter, we will explore the impact of model compression on the performance, memory footprint, inference time, and energy efficiency of four deep learning models when deployed on three different platforms: a laptop computer, a Raspberry Pi 3, and an Arduino Nano 33. Model compression has become an essential technique in recent years to reduce the size and complexity of deep learning models and improve their deployment efficiency on resource-limited platforms. By compressing the models, we aim to improve their runtime performance and reduce the memory and energy requirements without significantly sacrificing the accuracy of the models. Through our experiments, we will examine the effects of model compression on the performance of deep learning models, as well as its suitability for different platforms. The results of our analysis will provide valuable insights into the trade-offs between model accuracy, compression, and deployment efficiency on various platforms.

5.1 Impact of TFLite Compression on Model Characteristics

In this section, we thoroughly evaluated the effectiveness of TFLite compression on four deep learning models: CNN, LSTM, GRU-LSTM, and Bi-LSTM. To assess their performance, we measured the accuracy, memory footprint, energy consumption, and inference time of each model.

While it is possible to evaluate the impact of TFLite compression on these models using other platforms, we opted to conduct our evaluation on a laptop computer with specific specifications. We used a Windows 11 Dell Latitude 5410 with a quad-core, 1.6 GHz, 64-bit Intel [®] Core i5 processor, 16 GB RAM, and 1 TB solid-state drive. This decision was based on the fact that these specifications provided us with an optimal run environment, ample memory, and fast processor speed to conduct our evaluations effectively and efficiently.

By using this laptop computer, we were able to obtain accurate and reliable results that would help us better understand how TFLite compression impacted the performance of the four deep learning models.

5.1.1 *Impact on Parameter and Model Size*

In this section, we will delve into the impact of various compression optimizations on the memory footprint of deep learning models, focusing on the model size in megabytes (MB). Deep learning models are often large and computationally expensive, making them challenging to deploy on resource-constrained devices. Model compression techniques have emerged as a practical solution to address this challenge. These techniques involve reducing the size of the model and improving its efficiency without sacrificing performance. However, different compression techniques can affect the model size differently, and it is crucial to understand how each technique impacts the memory footprint of the model. We will explore some of the most commonly used compression techniques in deep learning, such as pruning, quantization, and weight clustering, and investigate how each technique affects the model size.

Table 5.1 shows the impact of several compression optimizations on the model size in megabytes for different deep learning models.

5.1.1.1 Quantization

For all models, the Integer and Dynamic Range quantization technique results in the smallest model size, with reductions ranging from 89% for the CNN model to 91% for the LSTM, GRU, and Bi-LSTM models. However, while the Float16 quantization technique also results in smaller models, it achieves less compression compared to Integer and Dynamic Range quantization techniques, with a reduction of 83% for all models.

5.1.1.2 Pruning

On the other hand, applying pruning optimizations to the TFLite models does not appear to reduce its size any further, even when the percentage pruning is varied from 10% to 90%. This is likely due to the way pruning is implemented, where pruned weights are simply assigned zero values without actually removing them from the FlatBuffer.

5.1.1.3 Weight Clustering

Similarly, applying weight clustering did not further reduce the size of the TFLite models despite varying the number of clusters from 4 to 128. This is likely because weight clustering kept the same number of model parameters but replaced all the unique parameters by a smaller set of defined values.

In conclusion, choosing the appropriate compression technique can significantly impact the size of a model. Various compression methods have been proposed, and their effectiveness varies. Integer quantization, pruning (both Constant Sparsity and Polynomial Decay), and weight clustering have proven to be the most effective techniques for reducing the size of models across various architectures. In some instances, these techniques can shrink models to less than 10% of their original size. Understanding the effects of different compression techniques on the model size is essential to optimize models for deployment on resource-constrained devices. Such optimization can enhance the performance and efficiency of these models without compromising their accuracy. Therefore, carefully selecting the compression technique based on the targeted device's resources and the desired accuracy is critical for achieving high-performance models.

Table 5.1: Impact of Compression Optimizations on Model Size in MB

Model Characteristics	TensorFlow	TensorFlow	Quantization			Pruning		Weight
	Baseline	Lite	Float16	Dynamic Range	Integer	Constant Sparsity	Polynomial Decay	Clustering
CNN	0.37	0.11	0.062	0.038	0.038	0.038	0.038	0.038
LSTM	18.5	6.1	3.1	1.5	1.5	1.5	1.5	1.5
GRU-LSTM	106	35	17.6	8.8	8.8	8.8	8.8	8.8
Bi-LSTM	87.7	29	14.6	7.3	7.3	7.3	7.3	7.3

5.1.2 Impact on Model Performance

In machine learning, model performance is a critical factor in determining the success of a project. However, in the context of deploying machine learning models on edge devices, model compression techniques are commonly used to reduce the size of models, making it easier to deploy them on devices with limited resources. In this section, we will discuss the impact of model compression on deploying machine learning models on edge devices. Specifically, we will examine the effects of different compression techniques on model performance in terms of accuracy, precision, recall, and F1 score. By exploring the trade-offs involved in selecting the appropriate compression technique for a given edge device and application, we hope to provide insights into optimizing model performance while minimizing the resources required for deployment on edge devices.

5.1.2.1 TFLite Compression

As seen in Tables 5.2, 5.3, 5.4, and 5.5, after converting all our TensorFlow models to TensorFlow Lite, we observed no change in accuracy and precision across most of the models. This is likely due to the model structure not changing fundamentally beyond the more memory-efficient implementation of the TFLite model. However, for the CNN and Bi-LSTM models, we noticed a significant improvement in precision, increasing from 92% to 97%. Similarly, the recall remained mostly unchanged except for the CNN and Bi-LSTM models, where we observed an increase from 72% to 74% and from 92% to 97%, respectively. We also observed that the F1 score remained consistent for most models, with the exception of the Bi-LSTM model, where it improved from 92% to 97%.

5.1.2.2 Quantization

For all tested models, except for integer quantization, the accuracy, precision, recall, and F1 score were the same as the TFLite model. However, more aggressive quantization techniques, such as integer quantization, resulted in a significant drop in accuracy, precision, recall, and F1 score. Specifically, for the CNN model, the accuracy dropped from 71% to 68%, precision from 97% to 82%, recall from 74% to 73%, and F1 score from 71% to 68%. Similarly, for the LSTM model, the accuracy dropped from 87% to 69%, precision from 87% to 69%, recall from 87% to 71%, and F1 score from 87% to 68%. For the GRU model, the accuracy dropped from 97% to 75%, precision from 97% to 74%, recall from 97% to 80%, and F1 score from 97% to 74%. Finally, for the Bi-LSTM model, the accuracy dropped from 97% to 82%, precision from 97% to 85%, recall from 97% to 85%, and F1 score from 97% to 83%.

5.1.2.3 Pruning

Pruning optimizations seem to have a more profound impact on the TFLite models' performance. Figures 5.1a, 5.1b, and 5.1c show the impact of varying the percentage pruning from 10% to 90% on the accuracy of the CNN, LSTM, and GRU models, respectively. For further reference, the impact of pruning on the precision, recall, and F1 score of the models can be seen in Figures 1, 2, and 3, respectively, located in appendix A.

Our results show that constant sparsity pruning led to a significant reduction in performance for the CNN model, with an average accuracy drop of 21%, precision drop of 39%, recall drop of 21%, and F1 score drop of 20%. The LSTM model showed a more moderate reduction in performance, with an average accuracy, precision, recall, and F1 score drop of 4.5%. Similarly, the GRU model exhibited a relatively minor decrease in performance, with an average accuracy, precision, recall, and F1 score drop of 3%.

In contrast, polynomial decay pruning resulted in harsher decreases in performance across all models. For the CNN model, the average accuracy drop was 44%, precision drop was 59%, recall drop was 43%, and F1 score drop was 45%. Similarly, for the GRU model, polynomial decay pruning caused an average accuracy, precision, and f1 score drop of 4%, and a recall drop of 3%. However, for the LSTM model, pruning prior to 40% actually resulted in better performance than the baseline model, which highlights the potential benefits of careful pruning selection. After 40% pruning, the model's performance started to drop, leading to an average accuracy, precision, recall, and F1 score drop of 8%.

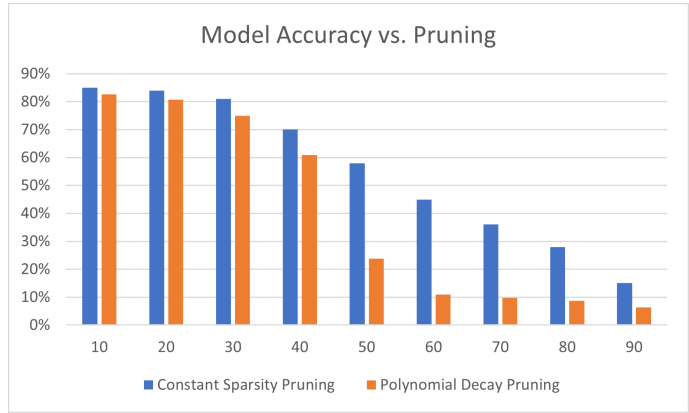
Additionally, it is important to note that the CNN model experienced a harsh dip in performance after 30% pruning for all metrics, while the LSTM and GRU models maintained a performance close to the baseline until 50% pruning, after which their performance started to drop. This highlights the importance of carefully selecting the pruning percentage for each model architecture and type of pruning to balance the trade-off between model complexity and performance.

It is also noteworthy that pruning optimizations were not applied to our Bi-LSTM model since TensorFlow Lite does not support pruning the Bi-directional layers present in our model.

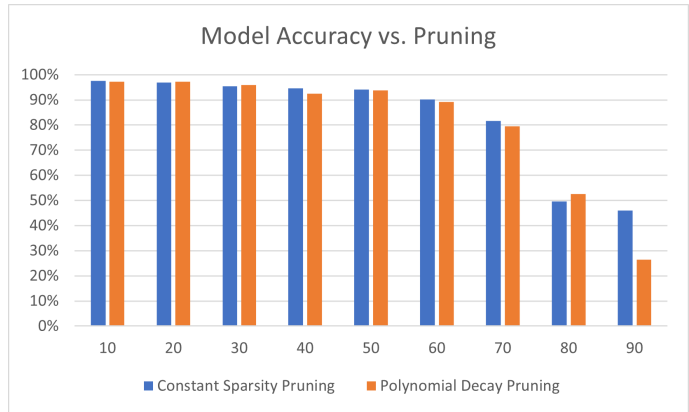
5.1.2.4 Weight Clustering

Weight clustering seemed to impact the model's accuracy the most; Figure 5.2a, 5.2b, 5.2c, and 5.2d show the impact of varying the number of clusters from 4 to 128 on the accuracy of the CNN, LSTM, GRU, and Bi-LSTM models, respectively. For additional reference, please see the impact of weight clustering on precision, recall, and F1 scores of the models in Figures 4, 5, and 6 located in Appendix B.

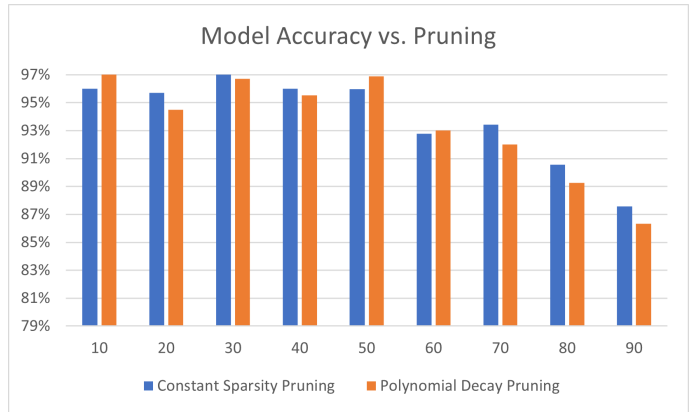
The CNN model's accuracy varied from 7% (4 clusters) to 77% (128 clusters) with an average accuracy of 46%. Notably, the model's accuracy remained close to the average when the number of clusters was set higher than 64. Similarly, the LSTM model's accuracy varied from 7% (4 clusters) to 96% (64 clusters) with an average accuracy of 73%. The model's accuracy remained close to the average when the number of clusters was set higher than 32. For the GRU model, the accuracy varied from 7% (4 clusters) to 99% (32 clusters) with an average accuracy of 82%. The model's accuracy remained close to the average when the number of clusters was set higher than 16. Finally, the Bi-LSTM model's accuracy varied from 57% (4 clusters) to 94% (32 clusters) with an average accuracy of 86%. The model's accuracy remained close to the average when the number of clusters was



(a) CNN Model



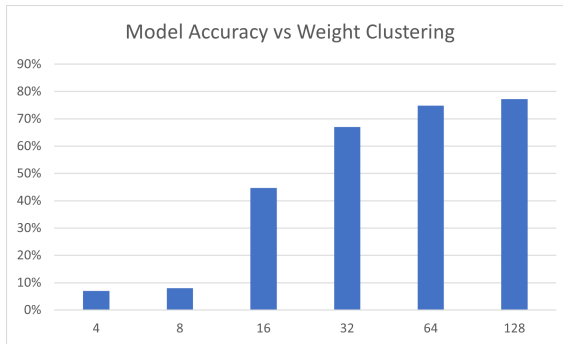
(b) LSTM Model



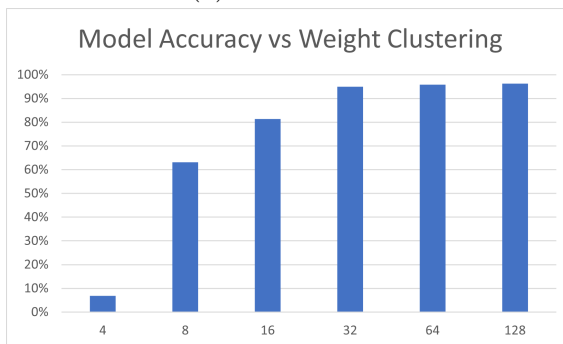
(c) GRU Model

Figure 5.1: Impact of Pruning on Models' Accuracy

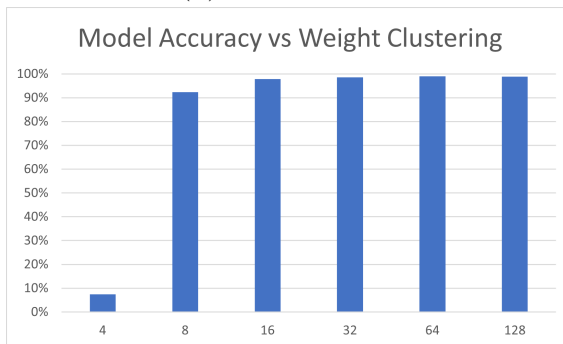
set higher than 16. For all four models, weight clustering improved the baseline accuracy when the number of clusters was higher than 32. Specifically, the CNN, LSTM, GRU, and Bi-LSTM models all exhibited improved accuracy when weight clustering was applied with a higher number of clusters. Similar patterns were observed for the precision, recall, and F1 score metrics, as their values remained very close to the corresponding accuracy values.



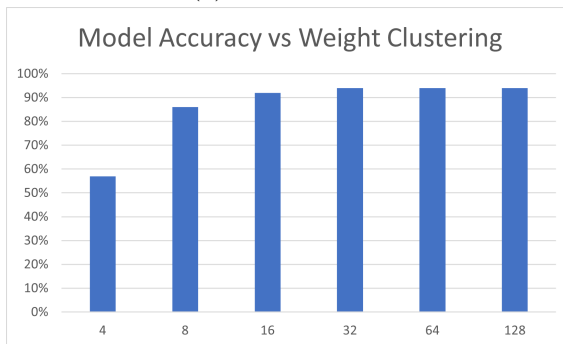
(a) CNN Model



(b) LSTM Model



(c) GRU Model



(d) Bi-LSTM Model

Figure 5.2: Impact of Weight Clustering on Models' Accuracy

Table 5.2: Impact of Compression Optimizations on Model Accuracy

Model Characteristics	TensorFlow	TensorFlow	Quantization			Pruning		Weight
	Baseline	Lite	Float16	Dynamic Range	Integer	Constant Sparsity	Polynomial Decay	Clustering
CNN	71	71	71	71	68	15 - 85 (avg=56)	6 - 83 (avg=40)	7 - 77 (avg=46)
LSTM	87	87	87	87	69	46 - 98 (avg=83)	26 - 97 (avg=80)	7 - 96 (avg=73)
GRU-LSTM	97	97	97	97	75	88 - 97 (avg=94)	86 - 97 (avg=93)	7 - 99 (avg=82)
Bi-LSTM	92	97	97	97	82	-	-	57 - 94 (avg=86)

Table 5.3: Impact of Compression Optimizations on Model Precision

Model Characteristics	TensorFlow	TensorFlow	Quantization			Pruning		Weight
	Baseline	Lite	Float16	Dynamic Range	Integer	Constant Sparsity	Polynomial Decay	Clustering
CNN	92	97	97	97	82	15 - 85 (avg=56)	7 - 83 (avg=40)	7 - 77 (avg=46)
LSTM	87	87	87	87	69	46 - 98 (avg=83)	26 - 97 (avg=80)	7 - 96 (avg=73)
GRU-LSTM	97	97	97	97	74	88 - 97 (avg=94)	86 - 97 (avg=93)	7 - 99 (avg=82)
Bi-LSTM	92	97	97	97	82	-	-	57 - 94 (avg=86)

Table 5.4: Impact of Compression Optimizations on Model Recall

Model Characteristics	TensorFlow	TensorFlow	Quantization			Pruning		Weight
	Baseline	Lite	Float16	Dynamic Range	Integer	Constant Sparsity	Polynomial Decay	Clustering
CNN	72	74	74	74	73	16 - 86 (avg=57)	4 - 83 (avg=42)	0.5 - 80 (avg=49)
LSTM	87	87	87	87	71	45 - 98 (avg=83)	25 - 97 (avg=80)	0.5 - 96 (avg=72)
GRU-LSTM	97	97	97	97	80	88 - 97 (avg=94)	87 - 97 (avg=94)	35 - 99 (avg=87)
Bi-LSTM	92	97	97	97	85	-	-	63 - 94 (avg=88)

Table 5.5: Impact of Compression Optimizations on Model F1-Score

Model Characteristics	TensorFlow	TensorFlow	Quantization			Pruning		Weight
	Baseline	Lite	Float16	Dynamic Range	Integer	Constant Sparsity	Polynomial Decay	Clustering
CNN	70	71	71	71	68	15 - 85 (avg=56)	2 - 83 (avg=39)	1 - 78 (avg=45)
LSTM	87	87	87	87	68	45 - 98 (avg=83)	24 - 97 (avg=80)	1 - 96 (avg=72)
GRU-LSTM	97	97	97	97	74	88 - 97 (avg=94)	86 - 97 (avg=93)	3 - 99 (avg=82)
Bi-LSTM	92	97	97	97	83	-	-	58 - 94 (avg=86)

In conclusion, our findings suggest that excessive pruning can alter a model’s architecture and lead to significant accuracy loss. Additionally, the study highlights the impact of the number of weight clusters on the model’s architecture and accuracy. Specifically, a low number of clusters can negatively affect the model’s accuracy. Furthermore, our study underscores the importance of weight optimization techniques, such as polynomial decay pruning and weight clustering, to achieve optimal results by eliminating redundant weights and representing weights using a smaller subset of unique values. These optimization techniques have shown to improve the generalizability and performance of machine learning models.

5.1.3 Impact on Inference Time

This section investigates the impact of model compression on the inference time of four deep learning sequence models: CNN, LSTM, GRU, and Bi-LSTM. To do this, we analyze Tables 5.6, 5.7, 5.8, and 5.9, which provide details on the effect of each compression technique on inference time, as well as the speedup or slowdown relative to the baseline and TFLite models for each of the four models.

5.1.3.1 TFLite Compression

Our experiments show that converting our baseline CNN model to a TFLite model resulted in a significant reduction in inference time from 85 msec to 0.27 msec, yielding an impressive speedup of $315\times$. Similarly, for LSTM, we observed a speedup of $1.4\times$ with a reduction in inference time from 57.5 msec to 50 msec. The reduction in inference time is likely due to the corresponding reduction in the number of operations that need to be executed and, in the case of TFLite, faster access to model parameters using efficient data structures like FlatBuffers.

However, for the GRU model, we noted a slowdown of $1.2\times$ as the baseline model required 99.3 msec while the TFLite model needed 117 msec. In the case of the bi-LSTM model, the baseline inference time was 110 msec, and the TFLite model took 130 msec, resulting in a slowdown of $1.2\times$.

5.1.3.2 Quantization

The CNN model showed significant improvements in inference time when float16, dynamic range, and integer quantization were applied, resulting in inference times of 0.27, 6, and 10 msec, respectively. These correspond to speedups of $315\times$, $14\times$, and $9\times$ faster than the baseline model, respectively. However, when compared to the TFLite model, dynamic range and integer quantization resulted in slowdowns of $52.5\times$ and $31.5\times$, respectively, highlighting the importance of carefully considering the impact of quantization on model performance and selecting the appropriate quantization approach for the specific model.

On the other hand, the LSTM model showed slower inference times when float16, dynamic range, and integer quantization were applied, resulting in inference times of 67, 68, and 810 msec, respectively. These correspond to slowdowns of $1.2\times$, $1.2\times$, and $14\times$ slower than the baseline model. When compared to the TFLite model, the slowdowns were even greater, with dynamic range and integer quantization resulting in slowdowns of $1.7\times$ and $1.6\times$ slower, respectively, while the use of integer quantization resulted in a slowdown of $20.2\times$ slower.

The GRU model showed mixed results when float16, dynamic range, and integer quantization were applied, resulting in inference times of 95, 110, and 3050 msec, respectively. These correspond to a slight speedup of $1.05\times$ and slowdowns of $1.1\times$ and $30.7\times$ slower than the baseline model. When compared to the TFLite model, float16 resulted in a small speedup of $1.2\times$, while dynamic range and integer quantization resulted in slowdowns of $1.06\times$ and $26\times$ slower, respectively.

Similarly, the Bi-LSTM model exhibited varied performance when float16, dynamic range, and integer quantization were applied, resulting in inference times of 90, 140, and 4300 msec, respectively. These correspond to a slight speedup of 1.2 and slowdowns of $1.3\times$ and $39\times$ slower than the baseline model. Comparing these results to the TFLite

Table 5.6: Impact of Compression Optimizations on inference time in CNN Model

Model	TensorFlow		Quantization			Pruning		Weight
	Baseline	TensorFlow Lite	Float16	Dynamic Range	Integer	Constant Sparsity	Polynomial Decay	
Characteristics								
Inference Time (msecs)	85	0.27	0.27	6	10	0.1 - 0.79 (avg=0.24)	3.5 - 8.5 (avg=5.9)	3.1 - 8.1 (avg=4.8)
Speedup w.r.t Baseline Model	1	315	315	14	9	108 - 850 (avg=354)	10 - 24 (avg=14)	10 - 27 (avg=18)
Speedup w.r.t TFLite Model	-	1	1	-52.5	-31.5	(-2.9) - 2.7 (avg=1.1) (-31.5) - (-13) (avg=-22)	(-30) - (-11.5) (avg=-17.8)	

Table 5.7: Impact of Compression Optimizations on Inference Time in LSTM Model

Model	TensorFlow		Quantization			Pruning		Weight
	Baseline	TensorFlow Lite	Float16	Dynamic Range	Integer	Constant Sparsity	Polynomial Decay	
Characteristics								
Inference Time (msecs)	57.5	40	68	67	810	31 - 66 (avg=46)	37 - 40 (avg=37)	57 - 69 (avg=65)
Speedup w.r.t Baseline Model	1	1.4	-1.2	-1.2	-14	(-1.1) - 2 (avg=1.3)	1.4 - 1.6 (avg=1.6)	(-1.2) - 1 (avg=0.9)
Speedup w.r.t TFLite Model	-	1	-1.7	-1.6	-20.2	(-1.6) - 1.2 (avg=-1.1)	1.08 - 1 (avg=1.08)	(-1.4) - (-1.7) (avg=-1.6)

Table 5.8: Impact of Compression Optimizations on Inference Time in GRU Model

Model Characteristics	TensorFlow		TensorFlow		Quantization		Pruning		Weight Clustering
	Baseline	Lite	Float16	Dynamic Range	Integer	Constant Sparsity	Polynomial Decay		
Inference Time (msecs)	99.3	117	95	110	3050	85 - 92 (avg=89)	113 - 133 (avg=119)	110 - 112 (avg=111)	
Speedup w.r.t Baseline Model	1	-1.2	1.05	-1.1	-30.7	1.08 - 1.17 (avg=1.12)	(-1.1) - (-1.33) (avg=-1.1)	(-1.12) - (-1.13) (avg=-1.12)	
Speedup w.r.t TFLite Model	-	1	1.2	1.06	-26	1.3 - 1.4 (avg=1.3)	1.04 - 1.1 (avg=1.02)	1.04 - 1.1 (avg=1.1)	

Table 5.9: Impact of Compression Optimizations on Inference Time in Bi-LSTM Model

Model Characteristics	TensorFlow		TensorFlow		Quantization		Weight Clustering
	Baseline	Lite	Float16	Dynamic Range	Integer		
Inference Time (msecs)	110	130	90	140	4300	129 - 185 (avg=154)	
Speedup w.r.t Baseline Model	1	-1.2	1.2	-1.3	-39	(-1.1) - (-1.7) (avg=-1.4)	
Speedup w.r.t TFLite Model	-	1	1.4	-1.08	-33	(-1.4) - 1 (avg=-1.1)	

model, float16 resulted in a modest speedup of $1.4\times$, while dynamic range and integer quantization resulted in slowdowns of $1.08\times$ and $33\times$ slower, respectively.

5.1.3.3 Pruning

The CNN model exhibited notable improvements in inference time using constant sparsity pruning, achieving inference times ranging from 0.1 to 0.79 msec and an average inference time of 0.24 msec. On the other hand, polynomial decay pruning achieved inference times ranging from 3.5 msec to 8.5 msec, with an average inference time of 5.9 msec. While the inference times for constant sparsity and polynomial decay pruning are generally faster than the baseline model, they are still slower than the speedup achieved by the TFLite model in most cases.

Similarly, the LSTM model demonstrated significant improvements in inference time using constant sparsity pruning, with inference times ranging from 31 to 66 msec and an average inference time of 46 msec. Meanwhile, polynomial decay pruning achieved inference times ranging from 37 msec to 40 msec, with an average inference time of 37 msec. When compared to the TFLite model, the performance of the LSTM model varied depending on the sparsity level, with constant sparsity pruning resulting in faster or slower inference times than the TFLite model, and polynomial decay pruning consistently resulting in faster inference times.

The GRU model also showed significant improvements in inference time with constant sparsity pruning, achieving inference times ranging from 85 to 92 msec and an average inference time of 89 msec. However, with polynomial decay pruning, inference times ranged from 113 msec to 133 msec, resulting in an average inference time of 119 msec, which was slower than the baseline model. Despite this, the performance of the GRU model was consistently better and faster than the TFLite model across all levels of sparsity.

Overall, these findings suggest that constant sparsity pruning can be an effective method for improving the inference time of neural network models without sacrificing accuracy. However, the choice of pruning method and sparsity level can have a significant impact on the performance of the model, and it is important to carefully consider these factors when implementing pruning techniques.

5.1.3.4 Weight Clustering

Weight clustering optimization proved effective in reducing inference time for both the CNN and LSTM models. The CNN model exhibited notable improvements in inference time, ranging from 3.1 to 8.1 msec and an average of 4.8 msec, while the LSTM model demonstrated significant enhancements, with inference times ranging from 57 to 69 msec and an average of 65 msec. Despite these improvements, neither model could outperform the TFLite model in any of the cases, highlighting the superior performance of the TFLite model in terms of inference speed.

On the contrary, the GRU model demonstrated slower inference times than the baseline model, with a range of 110 to 112 msec and an average of 111 msec. However, it still achieved faster inference times than the TFLite model.

For the Bi-LSTM model, weight clustering led to slower inference times, ranging from 129 to 185 msec, with an average of 154 msec. These results indicate that weight clustering did not improve the inference time speed of the Bi-LSTM model and that it was consistently slower than both the baseline and TFLite models in all cases.

5.1.4 *Impact on Energy Consumption*

In this particular section, the focus of the study is to examine and analyze the effect of various compression techniques on the energy consumption of deep learning models. To achieve this objective, the method employed involved computing the total power consumption of my laptop computer, referred to as the Thermal Design Power (TDP), multiplied by the duration it takes each model to execute a single inference. The results obtained from this process are presented in Table 5.10 and discussed in detail to provide a comprehensive understanding of the impact of compression techniques on energy consumption.

Thermal Design Power (TDP) is the amount of power that a computer’s cooling system is required to dissipate to maintain the temperature within the operating limits of the computer. TDP is measured in watts and is a specification of the maximum amount of power that a computer component can draw under normal usage conditions. In the context of computing, TDP is a useful metric for estimating the power consumption of a given component. This is because it represents the maximum amount of power that the component can consume under normal conditions, and provides a baseline for calculating the power consumption of the entire system. By multiplying the TDP of my laptop computer, which is 15 watts, with the time it takes the machine learning model to perform a single inference, one can estimate the energy consumption of the model. The inference time was measured on the same laptop computer using the same input data for each model. This is a useful metric for evaluating the efficiency of different machine learning models and compression techniques, and can inform decisions regarding the design of energy-efficient machine learning systems.

For the LSTM, GRU, and Bi-LSTM models, the energy consumption of 8-bit integer quantization can sometimes exceed that of the baseline due to the trade-off between computation time and memory access. While quantizing a model to 8-bit integers reduces the precision of weights and activations, it also reduces the memory usage and computation time required for inference. However, when the quantization is suboptimal, the computation time may increase as the CPU may perform extra calculations to convert between the 8-bit integer format and the original model format. Additionally, 8-bit integer quantization can result in more memory access, which can lead to increased power consumption if the memory access is not optimized or if the cache is not used efficiently.

Similarly, pruning (LSTM and GRU) and weight clustering (GRU and Bi-LSTM) can also lead to increased energy consumption if not implemented properly. Pruning involves removing the weights with the smallest magnitude, which can reduce the number of operations required for inference and the memory footprint of the model. However, if pruning is not optimized, it can lead to increased computation time and memory access. This is because the sparsity induced by pruning can result in irregular memory access patterns, which can cause cache misses and result in additional memory accesses. On the other hand, weight clustering involves grouping weights into clusters and representing each cluster with a single value. This can reduce the number of parameters and the memory footprint of the model. However, if the clustering is not optimized, it can result in increased computation time and memory access. This is because the quantization boundaries used for clustering may not align with the actual distribution of weights, leading to quantization errors and loss of accuracy.

In the table that compares the energy consumption of different models based on compression, the energy consumption is correlated (linear) with the inference time of each model. This is because the time required to perform a single inference directly affects the

Table 5.10: Impact of Compression Optimizations on Energy Consumption in Joules

Model Name	TensorFlow		TensorFlow Lite		Quantization		Pruning		Weight Clustering
	Baseline	Lite	Float16	Dynamic Range	Integer	Constant Sparsity	Polynomial Decay		
CNN	1.2	0.004	0.004	0.09	0.15	0.0015 - 0.011 (avg=0.003)	0.05 - 0.12 (avg=0.08)	0.04 - 0.12 (avg=0.07)	
LSTM	0.86	0.6	1	1	12.1	0.4 - 0.9 (avg=0.7)	0.5 - 0.6 (avg=0.5)	0.8 - 1 (avg=0.9)	
GRU	1.5	1.8	1.4	1.7	45.8	1.3 - 1.4 (avg=1.3)	1.7 - 2 (avg=1.8)	1.7 (avg=1.7)	
Bi-LSTM	1.65	1.95	1.3	2.1	64.5	-	-	1.9 - 2.7 (avg=2.3)	

amount of energy consumed by the model, and this relationship is reflected in the table..

By comparing the energy consumption and inference times of different models, one can identify which models are more energy-efficient than others. This information can be used to guide decisions regarding the design of machine learning systems, such as selecting which models to deploy in resource-constrained environments or identifying areas for further optimization. Additionally, by measuring the energy consumption of models at different compression levels, one can assess the trade-offs between model size, accuracy, and energy consumption, and make informed decisions regarding the best balance of these factors for a given application.

5.2 Deployment on Edge Devices

In this section, we will focus on two TFLite models: the smallest (CNN) and the most accurate (GRU-LSTM), based on all the evaluated compression techniques. It should be noted that we only considered TFLite models for deployment on the Raspberry Pi Model 3 since Edge Impulse currently only supports conversion of TFLite models to C code without further compression optimizations. Our goal is to deploy these models on the Raspberry Pi Model 3 board and the Arduino Nano 33 BLE Sense and analyze their size, accuracy, inference time, and power consumption. We will evaluate the performance of these models on both devices and compare the results to understand the trade-offs between model size, accuracy, and energy efficiency.

The Raspberry Pi Model 3 board and the Arduino Nano 33 BLE Sense are two popular microcontroller boards widely used in IoT applications. The Raspberry Pi Model 3 board is a powerful device with a quad-core ARM Cortex-A53 processor and 1 GB of RAM, while the Arduino Nano 33 BLE Sense is a low-power device with an Arm Cortex-M4F processor and 256 KB of RAM.

By deploying the selected models on both devices, we can analyze the impact of the hardware constraints on the model performance. We will measure the inference time and power consumption of the models and compare them with the results obtained from the evaluation of the models on a desktop computer. We will also evaluate the accuracy of the models on the microcontroller boards and compare them with the accuracy obtained from the evaluation on a desktop computer.

5.2.1 *Raspberry Pi Model 3*

In this section, we deployed the CNN model on the Raspberry Pi Model 3 using two different techniques. The first technique involved deploying the TensorFlow Lite model, while the second technique involved deploying the C code using Edge Impulse. We repeated the same deployment process for the GRU-LSTM model. We will compare the performance, size, accuracy, and inference time of both techniques for both models to determine which method is more suitable for deployment on the Raspberry Pi.

As shown in Table 5.11, for each deployed model, we computed its accuracy on the test set, as well as its inference time for a single audio sample. The energy consumption was computed by first measuring the power of the model using the PowerTOP function in the Raspberry Pi, which measures the power consumption of different processes running on the Raspberry Pi, and then multiplying it by the inference time. PowerTOP is a Linux tool designed to measure the power consumption of various components on a system, such

as CPU, GPU, and memory. It uses the kernel’s Performance Monitoring Counters to gather information about the system’s power usage and presents it in an easy-to-read format. PowerTOP can be used to identify which applications or processes are consuming the most power on a system, allowing users to take appropriate action to optimize power consumption. In addition, it can also suggest changes to the system’s configuration that can help reduce power consumption, such as disabling unnecessary hardware components or reducing the system’s CPU frequency. To use PowerTOP to measure the power consumption of an ML model on a Raspberry Pi, you would need to first run the model on the Pi and then run PowerTOP to monitor the system’s power usage during the model’s execution. This can help you identify which components of the Pi are consuming the most power during the model’s execution, and can help you optimize the model and/or the Pi’s configuration to reduce power consumption.

Table 5.11: Comparison of Deployment Techniques on Raspberry Pi Model 3

Model	Technique	Accuracy	Size	Inference Time	Energy
CNN	TensorFlow Lite	70%	113 KB	0.7 ms	0.00371 J
CNN	EON Compiler	64%	54 KB	67 ms	0.0141 J
GRU-LSTM	TensorFlow Lite	95%	35,240 KB	1000 ms	15 J
GRU-LSTM	EON Compiler	94.15%	660 KB	430 ms	6.45 J

For the CNN model, The TensorFlow Lite technique achieved a higher accuracy of 70%, while the EON Compiler technique achieved a lower accuracy of 64%. However, the EON compiled model has a smaller size at 54 KB, compared to the TensorFlow Lite model at 113 KB. In terms of inference time, the TensorFlow Lite model outperformed the EON compiled model, taking only 0.7 ms for inference compared to 67 ms for the EON model. Additionally, the TensorFlow Lite model was more energy-efficient, consuming only 0.00371 J compared to 0.0141 J for the EON compiled model.

For the GRU-LSTM model, the TensorFlow Lite deployment technique achieves a higher accuracy of 95%, while the EON Compiler technique achieves a slightly lower accuracy of 94.15%. However, the TensorFlow Lite model is much larger in size at 35,240 KB compared to the EON Compiler model, which is only 660 KB. The inference time for the TensorFlow Lite model is also much longer at 1000 ms compared to the EON Compiler model, which takes only 430 ms to make inference times. In terms of energy consumption, the EON Compiler model is more efficient, consuming only 6.45 J compared to 15 J for the TensorFlow Lite model.

Overall, the table suggests that the TensorFlow Lite deployment technique is generally more efficient in terms of accuracy, inference time, and energy consumption for the CNN model. However, for the GRU-LSTM model, the EON Compiler technique is more efficient in terms of size, inference time, and energy consumption, but still achieves a slightly lower accuracy compared to TensorFlow Lite.

5.2.2 *Arduino Nano 33 BLE Sense*

In this section, we focused on deploying the CNN model on the Arduino Nano 33 BLE Sense board. Due to the limited memory and computational resources of this board, we only deployed the CNN model. For this deployment, we used Edge Impulse, as it allows us to convert the model to C code, which further compresses the model and makes it more efficient for deployment on resource-constrained devices like the Arduino Nano.

The deployed model had a RAM usage of 9.9 KB and a Flash usage of 44.1 KB. The model achieved an accuracy of 64% on the test set, which is the same accuracy achieved on the Raspberry Pi. The inference time for one audio sample was 28 ms, which is faster than the inference time achieved on the Raspberry Pi.

In terms of energy consumption, the power consumption of a machine learning model is a critical aspect to consider when deploying on low-power devices. In this study, we measured the power consumption of a CNN model on an Arduino Nano 33 BLE during inference, and found it to consume an average current of 90 mA, with a voltage of 3.3V. To measure the current consumption of an Arduino Nano 33 BLE board, we used a multimeter in current measurement mode. First, we set the multimeter to measure current in the milliamperage range and connected the red probe to the positive power supply pin of the Arduino board and the black probe to the ground pin. Then, we powered on the board and let it run its program or perform the intended task while the multimeter measured the current consumption. We repeated this process for different scenarios, such as when the board was idle, when it was running a machine learning model. By comparing the current consumption measurements for each scenario, we could evaluate the power efficiency of the board and identify any areas for optimization or improvement.

Based on the given inference time of 28ms and assuming a continuous operation mode, we calculated the power consumption to be 297 mW. The energy consumption of the model can then be calculated as the product of the power and inference time, which is 8.316 mJ.

The low energy consumption of the model on the Arduino Nano 33 BLE demonstrates its suitability for deployment on low-power devices with limited resources. The compact size of the model and fast inference time further enhance its potential for real-time applications, such as in IoT devices and wearable technologies. The results of this study suggest that the CNN model is a viable solution for low-power machine learning applications, and future research can explore further optimizations to improve its energy efficiency and performance.

CHAPTER 6

CONCLUSIONS AND FUTURE WORK

In this study, we investigated the effectiveness of model compression techniques for developing efficient TinyML models that can be deployed on resource-constrained edge devices. Specifically, we focused on the impact of these techniques on the performance of four deep learning models for a limited-vocabulary speech processing task in Arabic, with a particular emphasis on the Levantine dialect. Our experiments demonstrate that model compression techniques can significantly reduce the memory footprint of deep learning models while maintaining high accuracy and resulting in a substantial reduction in inference time and energy consumption. Furthermore, we deployed our optimized models on two distinct edge devices, which represent different resource-constrained environments, and evaluated their real-world performance. Our optimized models achieved real-time performance for limited-vocabulary speech recognition tasks, with an average inference time of less than 500ms on both edge devices.

The results of this study highlight the potential of model compression techniques for developing efficient TinyML models that can be deployed on resource-constrained edge devices for speech recognition tasks in Arabic, specifically the Levantine dialect. These results can guide the development of TinyML applications in various fields, including healthcare, smart cities, and industrial automation, by enabling the deployment of machine learning models on edge devices with limited resources. The study's findings are particularly relevant for regions where Arabic is the primary language, and the Levantine dialect is the most commonly used. These regions can benefit from the deployment of efficient speech recognition models that can run on resource-constrained devices, enabling the development of innovative applications that can improve people's lives.

The findings of this study suggest several avenues for future research in the field of TinyML. Firstly, although we focused on limited-vocabulary speech processing tasks, future studies could investigate the effectiveness of model compression techniques for larger vocabulary speech recognition tasks. Furthermore, the performance of our models was evaluated for a specific speech recognition task in the Levantine dialect. Future studies could investigate the performance of these models for other Arabic dialects and languages, which would help to broaden the applicability of these models across different regions and languages.

Secondly, the study focused on four deep learning models commonly used for speech recognition tasks. Future studies could investigate the effectiveness of model compression techniques for other types of machine learning models, such as decision trees, support vector machines, and random forests. This would enable researchers to identify the most

effective compression techniques for different types of machine learning models and develop efficient TinyML models for various applications.

Thirdly, although our study evaluated the performance of optimized models on two distinct edge devices, future studies could evaluate their performance on a more extensive range of devices, including smartphones, wearables, and IoT devices. This would enable the development of efficient speech recognition applications that can run on a broad range of edge devices with varying degrees of resource constraints.

In addition to model compression techniques, another potential area for future development in the context of TinyML for limited-vocabulary speech processing in Arabic and Levantine dialect is hardware acceleration. With the increasing demand for intelligent edge devices, there is a growing need for specialized hardware that can efficiently run machine learning models. Hardware acceleration can significantly improve the performance of TinyML applications by providing specialized computational units that can perform specific tasks more efficiently than general-purpose processors. One promising approach for hardware acceleration of TinyML models is field-programmable gate arrays, which are programmable logic devices that can be customized to perform specific tasks. FPGAs offer several advantages over traditional processors, such as higher parallelism and lower power consumption. Furthermore, FPGAs can be programmed to implement custom neural network architectures, which can significantly improve the efficiency of TinyML models. Another potential area for hardware acceleration is the use of dedicated hardware accelerators for specific deep learning operations, such as convolutional or recurrent layers. By offloading these computations to specialized hardware, edge devices can achieve significant performance improvements while minimizing energy consumption.

Finally, the study evaluated the performance of optimized models for speech recognition tasks in Arabic. Future studies could investigate the performance of these models for other applications, such as image and video processing, natural language processing, and predictive maintenance. This would enable the development of a broad range of efficient TinyML applications that can run on resource-constrained edge devices, paving the way for the development of innovative and practical applications that can improve people's lives.

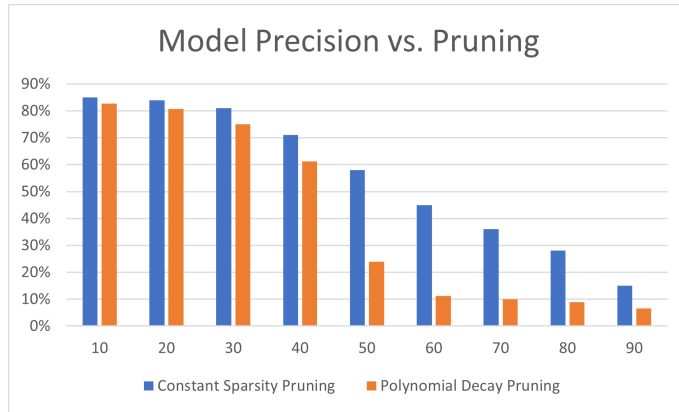
In conclusion, the findings of this study highlight the potential of model compression techniques for developing efficient TinyML models for limited-vocabulary speech recognition tasks in Arabic, specifically the Levantine dialect. The study's findings provide a foundation for future research in the field of TinyML, enabling the development of innovative applications that can run on resource-constrained edge devices and improve people's lives.

APPENDIX

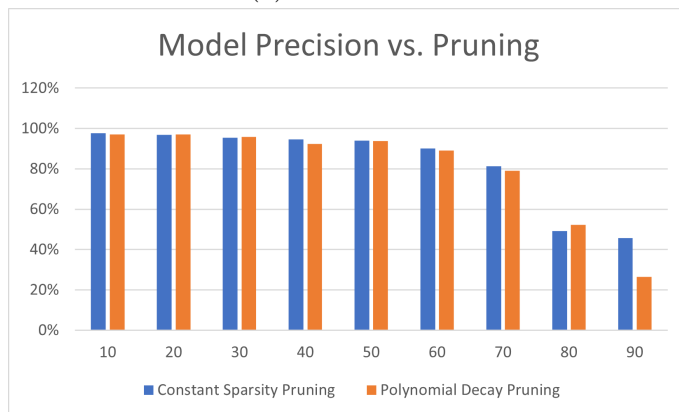
This appendix provides a comprehensive analysis of the impact of pruning and weight clustering on the performance of several machine learning models, including CNN, LSTM, and GRU-LSTM. The goal of this research was to investigate how pruning and weight clustering affect model performance and determine the optimal level of pruning and number of clusters for each model.

The appendix contains a total of 21 figures that visually illustrate the results of the analysis. The first nine figures illustrate the impact of pruning on the precision, recall, and f1 score of each model, while the remaining 12 figures demonstrate how weight clustering impacts the accuracy, completeness, and overall performance of each model. The figures are labeled and organized by model type, making it easy to compare the results across different models.

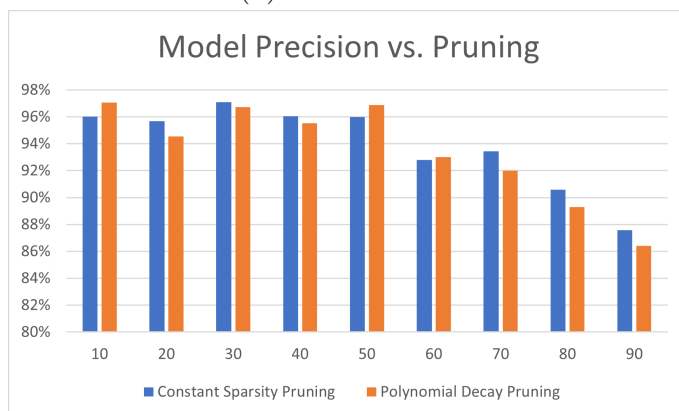
These figures provide a supplementary resource for readers who wish to delve deeper into the analysis and understand the significance of the findings. By examining the changes in recall, precision, and f1 score as pruning levels increase and the number of clusters change, readers can gain valuable insights into the performance of each model and determine the optimal pruning level and number of clusters for their specific needs.



(a) CNN Model

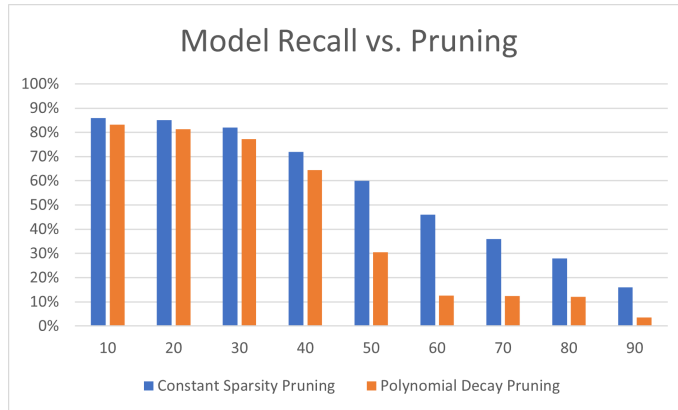


(b) LSTM Model

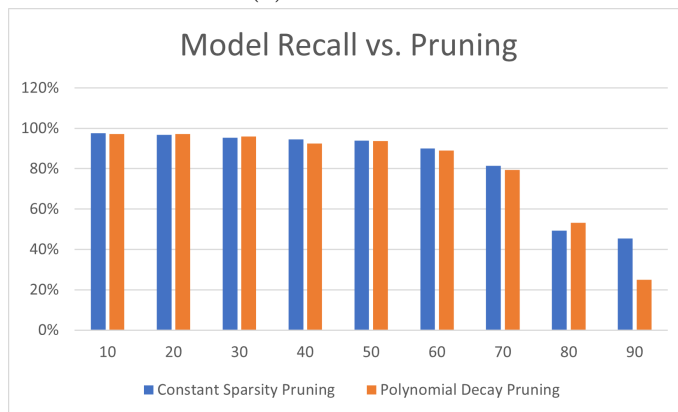


(c) GRU Model

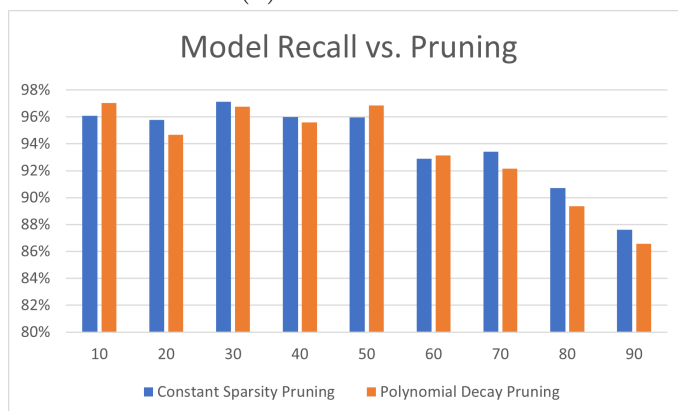
Figure 1: Impact of Pruning on Models' Precision



(a) CNN Model

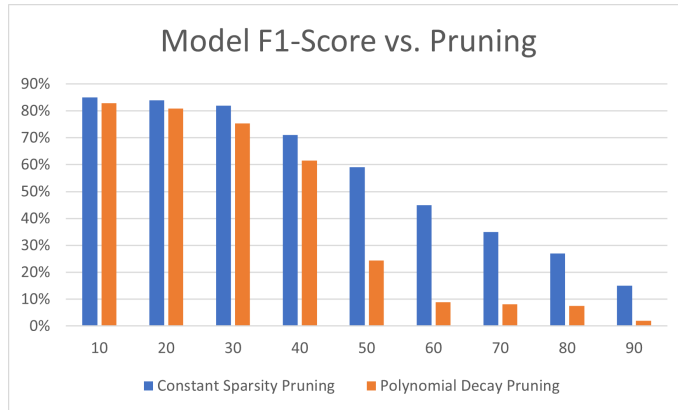


(b) LSTM Model

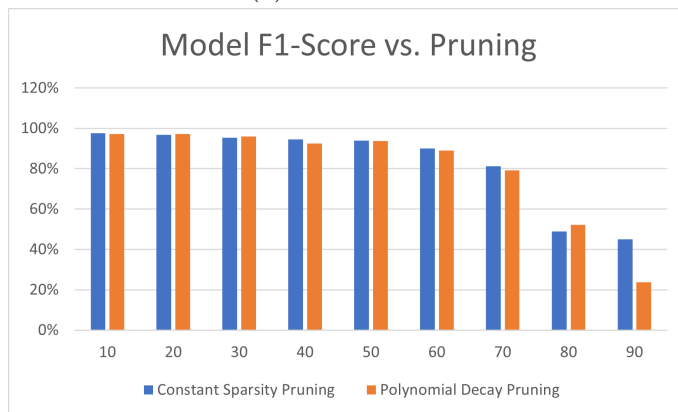


(c) GRU Model

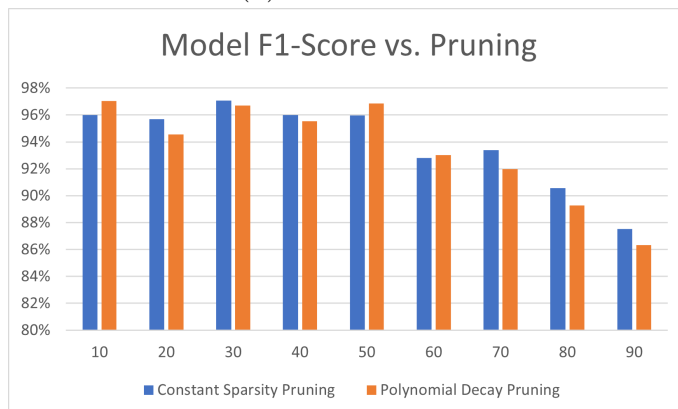
Figure 2: Impact of Pruning on Models' Recall



(a) CNN Model

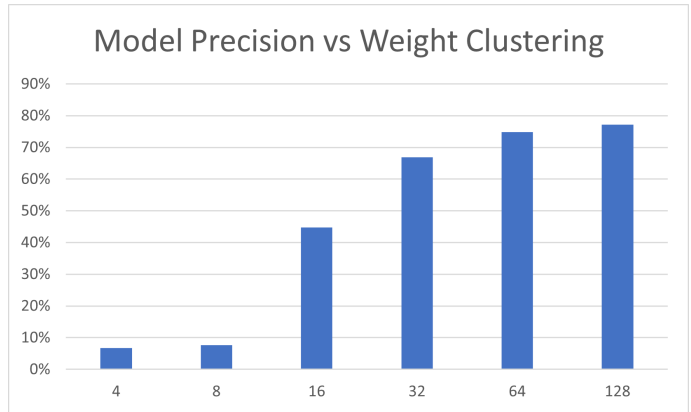


(b) LSTM Model

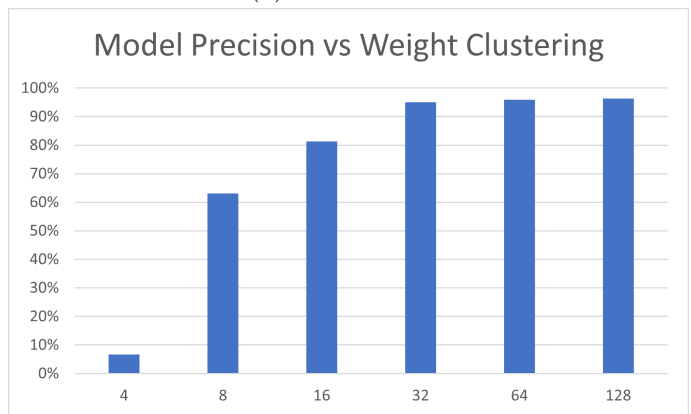


(c) GRU Model

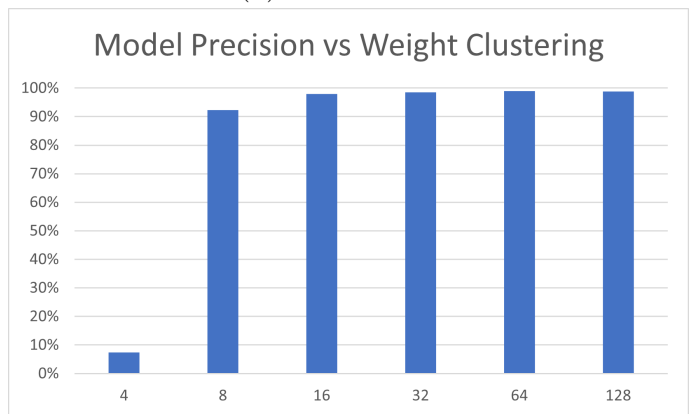
Figure 3: Impact of Pruning on Models' F1 Score



(a) CNN Model

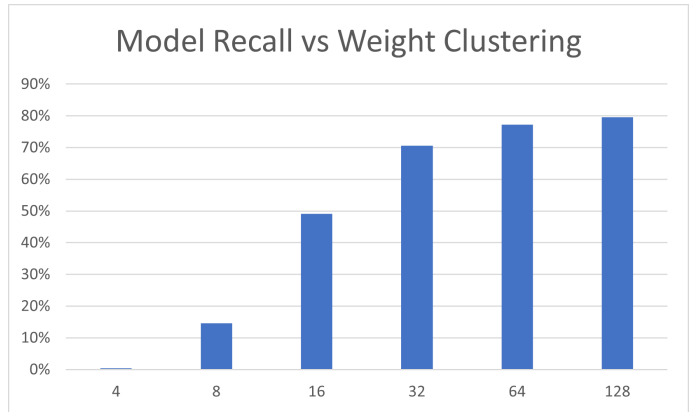


(b) LSTM Model

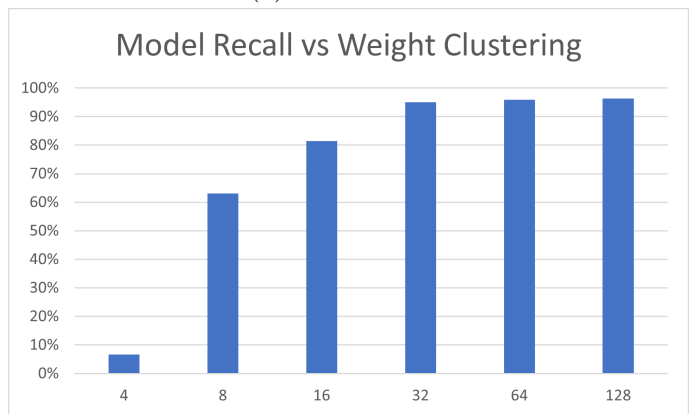


(c) GRU Model

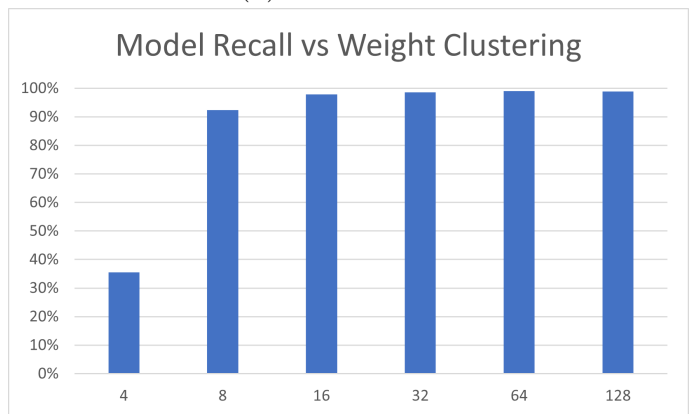
Figure 4: Impact of Weight Clustering on Models' Precision



(a) CNN Model

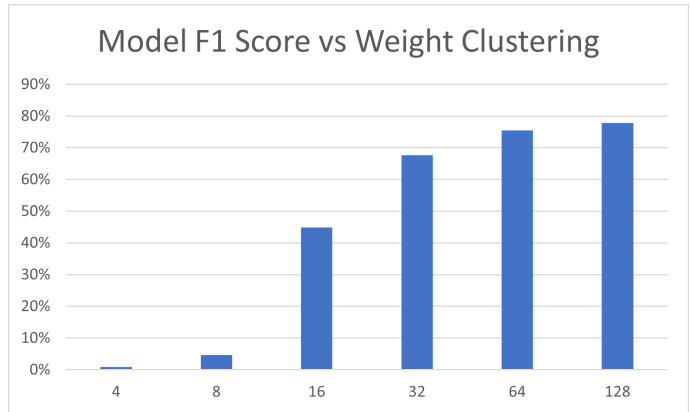


(b) LSTM Model

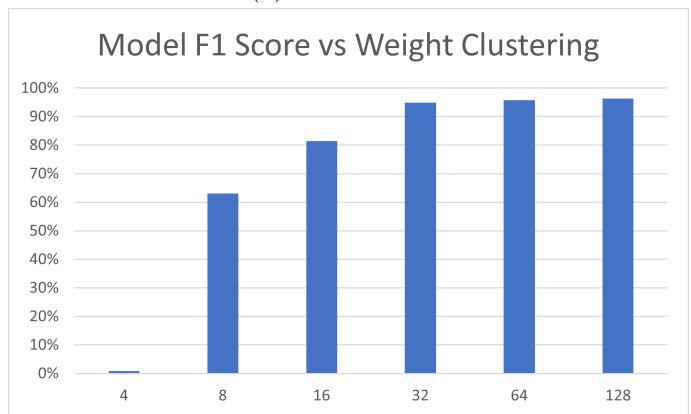


(c) GRU Model

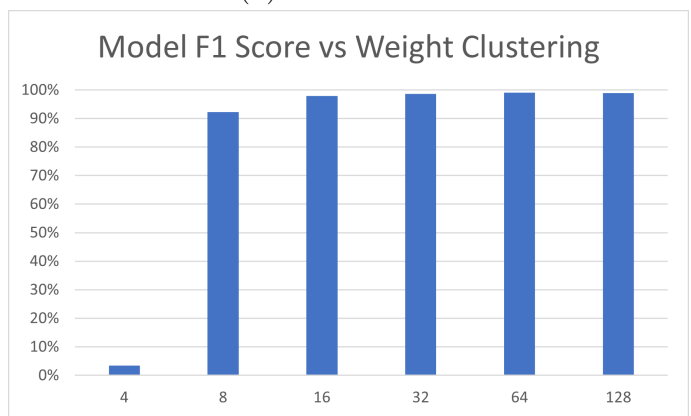
Figure 5: Impact of Weight Clustering on Models' Recall



(a) CNN Model



(b) LSTM Model



(c) GRU Model

Figure 6: Impact of Weight Clustering on Models' F1 Score

BIBLIOGRAPHY

- [1] T. Ammari, J. Kaye, J. Y. Tsai, and F. Bentley, “Music, Search, and IoT: How People (Really) Use Voice Assistants,” *ACM Transactions on Computer-Human Interaction*, vol. 26, no. 3, 2019, ISSN: 1073-0516. DOI: [10.1145/3311956](https://doi.org/10.1145/3311956). [Online]. Available: <https://doi.org/10.1145/3311956>.
- [2] J. Ghofrani and D. Reichelt, “Using Voice Assistants as HMI for Robots in Smart Production Systems,” in *CEUR Workshop Proceedings*, vol. 2339, 2019.
- [3] I. Lopatovska, K. Rink, I. Knight, *et al.*, “Talk to me: Exploring user interactions with the amazon alexa,” *Journal of Librarianship and Information Science*, vol. 51, no. 4, pp. 984–997, 2019. DOI: [10.1177/0961000618759414](https://doi.org/10.1177/0961000618759414). [Online]. Available: <https://doi.org/10.1177/0961000618759414>.
- [4] “Alexa privacy and data handling overview,” 2018.
- [5] S. Guamán, A. Calvopiña, P. Orta, F. Tapia Leon, and S. G. Yoo, “Device control system for a smart home using voice commands: A practical case,” Sep. 2018, pp. 86–89. DOI: [10.1145/3285957.3285977](https://doi.org/10.1145/3285957.3285977).
- [6] L. Hardesty, *Amazon alexa’s new wake word research at interspeech*, <https://www.amazon.science/blog/amazon-alexa-new-wake-word-research-at-interspeech>, Oct. 2020.
- [7] M. Nair, *Amazon’s alexa can now take questions and answers in arabic*, <https://gulfnews.com/business/amazons-alexa-can-now-take-questions-and-answers-in-arabic-1.84236228>, Dec. 2021.
- [8] L. Hardesty, *How alexa learned arabic*, <https://www.amazon.science/latest-news/how-alexa-learned-arabic>, Jan. 2022.
- [9] M. Palermo, *Fundamentals of smart home device control, including on/off state*, 2016.
- [10] *Learn how google improves speech models*, <https://support.google.com/assistant/answer/11140942?hl=en>.
- [11] S. D. Arya and S. Patel, “Implementation of google assistant & amazon alexa on raspberry pi,” *CoRR*, vol. abs/2006.08220, 2020. [Online]. Available: <https://arxiv.org/abs/2006.08220>.
- [12] *Google assistant, your own personal google*, <https://assistant.google.com/>.

- [13] V. Ulhas Arun Thakare Pranay Bhagwat Thakare, "Siri-the intelligent personal assistant," *International Research Journal of Innovations in Engineering and Technology (IRJIET)*, vol. 4, pp. 17–19, Jan. 2020.
- [14] *About ios 15 updates*, <https://support.apple.com/en-mide/HT212788>.
- [15] Z. Paul, "Cortana-intelligent personal digital assistant: A review," *International Journal of Advanced Research in Computer Science*, vol. 8, pp. 55–57, Aug. 2017. DOI: [10.26483/ijarcs.v8i7.4225](https://doi.org/10.26483/ijarcs.v8i7.4225).
- [16] I. H. Sarker, "Machine learning: Algorithms, real-world applications and research directions," *SN Computer Science*, vol. 2, no. 3, pp. 1–21, 2021.
- [17] S. Ahmed, "Insider's misuse detection: From hidden markov model to deep learning," PhD dissertation, 2019.
- [18] D. Suleiman, A. Awajan, and W. Al Etaiwi, "The use of hidden markov model in natural arabic language processing: A survey," *Procedia Computer Science*, vol. 113, pp. 240–247, 2017. DOI: <https://doi.org/10.1016/j.procs.2017.08.363>. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S1877050917317738>.
- [19] A. Boudlal, M. O. A. O. Bebah, A. Lakhouaja, A. Mazroui, and A. Meziane, "A markovian approach for arabic root extraction," *Int. Arab J. Inf. Technol.*, vol. 8, no. 1, pp. 91–98, 2011.
- [20] A. Alajmi, E. Saad, and M. Awadalla, "Hidden markov model based arabic morphological analyzer," *International Journal of Computer Engineering Research*, vol. 2, no. 2, pp. 28–33, 2011.
- [21] J. Dror, D. Shaharabani, R. Talmon, and S. Wintner, "Morphological analysis of the qur'an," *Literary and Linguistic Computing*, vol. 19, Nov. 2004.
- [22] M. Hadni, S. Ouatik El Alaoui, A. LACHKAR, and M. Meknassi, "Hybrid part-of-speech tagger for non-vocalized arabic text," *International Journal on Natural Language Computing*, vol. 2, pp. 1–15, Dec. 2013. DOI: [10.5121/ijnlc.2013.2601](https://doi.org/10.5121/ijnlc.2013.2601).
- [23] M. Hadni, S. A. Ouatik, A. Lachkar, and M. Meknassi, "Improving rule-based method for arabic pos tagging using hmm technique," 2013.
- [24] A. Alajmi, E. Saad, and M. Awadalla, "Dacs dewey index-based arabic document categorization system," *International Journal of Computer Applications*, vol. 975, p. 8887, 2012.
- [25] S. Shareef and Y. Irhayim, "A review: Isolated arabic words recognition using artificial intelligent techniques," in *Journal of Physics: Conference Series*, IOP Publishing, vol. 1897, 2021, p. 012026.
- [26] M. Dua, R. Aggarwal, V. Kadyan, and S. Dua, "Punjabi automatic speech recognition using htk," *International Journal of Computer Science Issues (IJCSI)*, vol. 9, no. 4, p. 359, 2012.

- [27] M. Elmahdy, R. Gruhn, W. Minker, and S. Abdennadher, “Modern standard arabic based multilingual approach for dialectal arabic speech recognition,” in *2009 Eighth International Symposium on Natural Language Processing*, IEEE, 2009, pp. 169–174.
- [28] S. Indolia, A. K. Goswami, S. Mishra, and P. Asopa, “Conceptual understanding of convolutional neural network- a deep learning approach,” *Procedia Computer Science*, vol. 132, pp. 679–688, 2018, ISSN: 1877-0509. DOI: <https://doi.org/10.1016/j.procs.2018.05.069>.
- [29] Z. C. Lipton, J. Berkowitz, and C. Elkan, “A critical review of recurrent neural networks for sequence learning,” *arXiv preprint arXiv:1506.00019*, 2015.
- [30] N. M. Rezk, M. Purnaprajna, T. Nordström, and Z. Ul-Abdin, “Recurrent neural networks: An embedded computing perspective,” *IEEE Access*, vol. 8, pp. 57 967–57 996, 2020.
- [31] R. M. Schmidt, *Recurrent neural networks (rnns): A gentle introduction and overview*, 2019. arXiv: [1912.05911](https://arxiv.org/abs/1912.05911).
- [32] J. Donahue, L. Anne Hendricks, S. Guadarrama, *et al.*, “Long-term recurrent convolutional networks for visual recognition and description,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 2625–2634.
- [33] B. Hunshamar, “Wake word detection using recurrent neural networks,” M.S. thesis, NTNU, 2018.
- [34] N. Zerari, S. Abdelhamid, H. Bouzgou, and C. Raymond, “Bidirectional deep architecture for arabic speech recognition,” *Open Computer Science*, vol. 9, no. 1, pp. 92–102, 2019.
- [35] A. S. M. B. Wazir and J. H. Chuah, “Spoken arabic digits recognition using deep learning,” in *2019 IEEE International Conference on Automatic Control and Intelligent Systems (I2CACIS)*, IEEE, 2019, pp. 339–344.
- [36] *Arab world books*. [Online]. Available: <https://www.arabworldbooks.com/en>.
- [37] *Hindawi*. [Online]. Available: <https://www.hindawi.org/>.
- [38] A. Souri, Z. E. Maazouzi, M. A. Achhab, and B. E. E. Mohajir, “Arabic text generation using recurrent neural networks,” in *International Conference on Big Data, Cloud and Applications*, Springer, 2018, pp. 523–533.
- [39] I. Dhall, S. Vashisth, and S. Saraswat, “Text generation using long short-term memory networks,” in Apr. 2020, pp. 649–657. DOI: [10.1007/978-981-15-2329-8_66](https://doi.org/10.1007/978-981-15-2329-8_66).
- [40] Y. Perwej, “Recurrent Neural Network Method in Arabic Words Recognition System,” *International Journal of Computer Science and Telecommunications (IJCST), Sysbase Solution (Ltd), UK, London ,ISSN 2047-3338*, 2012. [Online]. Available: <https://hal.archives-ouvertes.fr/hal-03362907>.

- [41] G. Nguyen, S. Dlugolinsky, M. Bobák, *et al.*, “Machine learning and deep learning frameworks and libraries for large-scale data mining: A survey,” vol. 52, no. 1, 2019, ISSN: 0269-2821. DOI: [10.1007/s10462-018-09679-z](https://doi.org/10.1007/s10462-018-09679-z). [Online]. Available: <https://doi.org/10.1007/s10462-018-09679-z>.
- [42] N. Halabi, “Modern standard arabic phonetics for speech synthesis,” Ph.D. dissertation, University of Southampton, Jul. 2016. [Online]. Available: <https://eprints.soton.ac.uk/409695/>.
- [43] L. Benamer and O. Alkishriwo, “Database for arabic speech commands recognition,” Dec. 2020.
- [44] K. Almeman, M. Lee, and A. A. Almiman, “Multi dialect arabic speech parallel corpora,” in *2013 1st International Conference on Communications, Signal Processing, and their Applications (ICCSIPA)*, IEEE, 2013, pp. 1–6.
- [45] M. Alghamdi, “Kacst arabic phonetics database,” Jan. 2000.
- [46] G. Droua-Hamdani, S. A. Selouani, and M. Boudraa, “Algerian arabic speech database (algasd): Corpus design and automatic speech recognition application,” *ARABIAN JOURNAL FOR SCIENCE AND ENGINEERING*, vol. 35, pp. 157–166, Dec. 2010.
- [47] *Arabic vocal emotions dataset (baved)*. [Online]. Available: <https://github.com/40uf411/Basic-Arabic-Vocal-Emotions-Dataset>.
- [48] T. Mesallam, M. Farahat, K. Malki, *et al.*, “Development of the arabic voice pathology database and its evaluation by using speech features and machine learning algorithms,” *Journal of Healthcare Engineering*, vol. 2017, pp. 1–13, Oct. 2017. DOI: [10.1155/2017/8783751](https://doi.org/10.1155/2017/8783751).
- [49] P. Warden, “Speech commands: A dataset for limited-vocabulary speech recognition,” *arXiv preprint arXiv:1804.03209*, 2018.
- [50] S. Becker, M. Ackermann, S. Lapuschkin, K.-R. Müller, and W. Samek, “Interpreting and explaining deep neural networks for classification of audio signals,” *arXiv preprint arXiv:1807.03418*, 2018.
- [51] C. Hou, B. Cao, and J. Fan, “A data-driven method to predict service level for call centers,” *IET Communications*, vol. 16, Jun. 2022. DOI: [10.1049/cmu2.12192](https://doi.org/10.1049/cmu2.12192).
- [52] B. Ay and G. Aydin, “Call center performance evaluation using big data analytics,” May 2016, pp. 1–6. DOI: [10.1109/ISNCC.2016.7746116](https://doi.org/10.1109/ISNCC.2016.7746116).
- [53] M. Ali and Y. Lee, “Crm sales prediction using continuous time-evolving classification,” in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 32, 2018.
- [54] D. Galvez, G. Damos, J. Ciro, *et al.*, *The people’s speech: A large-scale diverse english speech recognition dataset for commercial usage*, 2021. DOI: [10.48550/ARXIV.2111.09344](https://doi.org/10.48550/ARXIV.2111.09344). [Online]. Available: <https://arxiv.org/abs/2111.09344>.

- [55] J. Carletta, S. Ashby, S. Bourban, *et al.*, “The ami meeting corpus: A pre-announcement,” Jul. 2005. DOI: [10.1007/11677482_3](https://doi.org/10.1007/11677482_3).
- [56] M. Mazumder, S. Chitlangia, C. Banbury, *et al.*, “Multilingual spoken words corpus,” in *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 2)*, 2021. [Online]. Available: <https://openreview.net/forum?id=c20jiJ5K2H>.
- [57] *Common voice*. [Online]. Available: <https://labs.mozilla.org/projects/common-voice/>.
- [58] V. Pratap, Q. Xu, A. Sriram, G. Synnaeve, and R. Collobert, “MLS: A large-scale multilingual dataset for speech research,” in *Interspeech 2020*, ISCA, Oct. 2020. DOI: [10.21437/interspeech.2020-2826](https://doi.org/10.21437/interspeech.2020-2826). [Online]. Available: <https://doi.org/10.21437/5C%2Finterspeech.2020-2826>.
- [59] R. Vijayaraghavan and P. Kannan, “Applications of data mining and machine learning in online customer care,” Aug. 2011, p. 779. DOI: [10.1145/2020408.2020537](https://doi.org/10.1145/2020408.2020537).
- [60] J. L. Álvarez, J. D. Mozo, and E. Durán, “Analysis of single board architectures integrating sensors technologies,” *Sensors*, vol. 21, no. 18, p. 6303, 2021.
- [61] S. J. Johnston, P. J. Basford, C. S. Perkins, *et al.*, “Commodity single board computer clusters and their applications,” *Future Generation Computer Systems*, vol. 89, pp. 201–212, 2018, ISSN: 0167-739X. DOI: <https://doi.org/10.1016/j.future.2018.06.048>. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0167739X18301833>.
- [62] H. Ghael, “A review paper on raspberry pi and its applications,” Jan. 2020. DOI: [10.35629/5252-0212225227](https://doi.org/10.35629/5252-0212225227).
- [63] *Arduino nano 33 ble sense*, 2023. [Online]. Available: <https://docs.arduino.cc/hardware/nano-33-ble-sense>.
- [64] *Esp32 series datasheet*, 2022. [Online]. Available: <https://www.espressif.com/en/support/documents/technical-documents>.
- [65] B. Deng, Z. Bo, Y. Jia, Z. Gao, and Z. Liu, “Research on stm32 development board based on arm cortex-m3,” in *2020 IEEE 2nd International Conference on Civil Aviation Safety and Information Technology (ICCASIT)*, 2020, pp. 266–272. DOI: [10.1109/ICCASIT50869.2020.9368860](https://doi.org/10.1109/ICCASIT50869.2020.9368860).
- [66] *Stm32 software development tools*. [Online]. Available: <https://www.st.com/en/development-tools/stm32-software-development-tools.html>.
- [67] J. Park, Y. Boo, I. Choi, S. Shin, and W. Sung, “Fully neural network based speech recognition on mobile and embedded devices,” *Advances in neural information processing systems*, vol. 31, 2018.
- [68] X. Lei, A. Senior, A. Gruenstein, and J. Sorensen, “Accurate and compact large vocabulary speech recognition on mobile devices,” Aug. 2013, pp. 662–665. DOI: [10.21437/Interspeech.2013-189](https://doi.org/10.21437/Interspeech.2013-189).

- [69] I. McGraw, R. Prabhavalkar, R. Alvarez, *et al.*, *Personalized speech recognition on mobile devices*, 2016. DOI: [10.48550/ARXIV.1603.03185](https://doi.org/10.48550/ARXIV.1603.03185). [Online]. Available: <https://arxiv.org/abs/1603.03185>.
- [70] J. Lewis, *Analog and digital mems microphone design considerations*, 2013. [Online]. Available: <https://www.analog.com/en/index.html>.
- [71] G. D’Emilia, A. Gaspari, E. Natale, N. Montali, A. Prato, and A. Schiavi, “Validated data mining technique: A case study on pressure-field digital mems microphone calibration,” *Journal of Physics: Conference Series*, vol. 1065, p. 072028, Aug. 2018. DOI: [10.1088/1742-6596/1065/7/072028](https://doi.org/10.1088/1742-6596/1065/7/072028).
- [72] *I2s bus specification*, 2022.
- [73] S. Soro, *TinyML for Ubiquitous Edge AI*, 2021. DOI: [10.48550/ARXIV.2102.01255](https://doi.org/10.48550/ARXIV.2102.01255). [Online]. Available: <https://arxiv.org/abs/2102.01255>.
- [74] J. R. et al., “Widening Access to Applied Machine Learning With TinyML,” *Harvard Data Science Review*, vol. 4, 2022.
- [75] Microsoft, *Embedded Learning Library*, 2020. [Online]. Available: [https://microsoft.github.io/ELL/..](https://microsoft.github.io/ELL/)
- [76] STMicroelectronics, *Artificial Intelligence Ecosystem for STM32*. [Online]. Available: <https://www.st.com/content/st%20com/en/ecosystems/artificial-intelligence-ecosystem-stm32.html>.
- [77] ARM, *ARM NN SDK*. [Online]. Available: <https://www.arm.com/products/silicon-ip-cpu/ethos/arm-nn>.
- [78] S. Soro, *Tinyml for ubiquitous edge ai*, 2021. DOI: [10.48550/ARXIV.2102.01255](https://doi.org/10.48550/ARXIV.2102.01255). [Online]. Available: <https://arxiv.org/abs/2102.01255>.
- [79] S. Hymel, C. Banbury, D. Situnayake, *et al.*, *Edge impulse: An mlops platform for tiny machine learning*, 2022. DOI: [10.48550/ARXIV.2212.03332](https://doi.org/10.48550/ARXIV.2212.03332).
- [80] A. Osman, U. Abid, L. Gemma, M. Perotto, and D. Brunelli, *Tinyml platforms benchmarking*, 2021. DOI: [10.48550/ARXIV.2112.01319](https://doi.org/10.48550/ARXIV.2112.01319). [Online]. Available: <https://arxiv.org/abs/2112.01319>.
- [81] T.-H. Tsai and X.-H. Lin, “Speech densely connected convolutional networks for small-footprint keyword spotting,” *Multimedia Tools and Applications*, 2023. DOI: [10.1007/s11042-023-14617-5](https://doi.org/10.1007/s11042-023-14617-5).
- [82] J. Wang and S. Li, *Keyword spotting system and evaluation of pruning and quantization methods on low-power edge microcontrollers*, 2022. arXiv: [2208.02765](https://arxiv.org/abs/2208.02765) [cs.SD].
- [83] M. N. Miah and G. Wang, “Keyword spotting with deep neural network on edge devices,” in *2022 IEEE 12th International Conference on Electronics Information and Emergency Communication (ICEIEC)*, 2022, pp. 98–102. DOI: [10.1109/ICEIEC54567.2022.9835061](https://doi.org/10.1109/ICEIEC54567.2022.9835061).

- [84] F. Hölzke, H. Ahmed, F. Golatowski, and D. Timmermann, “Keyword spotting for industrial control using deep learning on edge devices,” in *2021 IEEE International Conference on Pervasive Computing and Communications Workshops and other Affiliated Events (PerCom Workshops)*, 2021, pp. 167–172. DOI: [10.1109/PerComWorkshops51409.2021.9430865](https://doi.org/10.1109/PerComWorkshops51409.2021.9430865).
- [85] J. Kwon and D. Park, “Hardware/software co-design for tinyml voice-recognition application on resource frugal edge devices,” *Applied Sciences*, vol. 11, no. 22, 2021. DOI: [10.3390/app112211073](https://doi.org/10.3390/app112211073). [Online]. Available: <http://dx.doi.org/10.3390/app112211073>.
- [86] S. Gondi and V. Pratap, “Performance evaluation of offline speech recognition on edge devices,” *Electronics*, vol. 10, no. 21, 2021. DOI: [10.3390/electronics10212697](https://doi.org/10.3390/electronics10212697). [Online]. Available: <http://dx.doi.org/10.3390/electronics10212697>.
- [87] D. Snyder, G. Chen, and D. Povey, “Musan: A music, speech, and noise corpus,” *arXiv preprint arXiv:1510.08484*, 2015.
- [88] E. Hardy and F. Badets, *An ultra-low power rnn classifier for always-on voice wake-up detection robust to real-world scenarios*, 2021. DOI: [10.48550/ARXIV.2103.04792](https://doi.org/10.48550/ARXIV.2103.04792). [Online]. Available: <https://arxiv.org/abs/2103.04792>.
- [89] K. Li and J. Principe, “Biologically-inspired pulse signal processing for intelligence at the edge,” *Frontiers in Artificial Intelligence*, Aug. 2021. DOI: [10.3389/frai.2021.568384](https://doi.org/10.3389/frai.2021.568384).
- [90] S. Yang, Z. Gong, K. Ye, Y. Wei, Z. Huang, and Z. Huang, “Edgernn: A compact speech recognition network with spatio-temporal features for edge computing,” *IEEE Access*, vol. 8, pp. 81 468–81 478, 2020. DOI: [10.1109/ACCESS.2020.2990974](https://doi.org/10.1109/ACCESS.2020.2990974).
- [91] P.-E. N. et al., “Quantization and Deployment of Deep Neural Networks on Microcontrollers,” *Sensors*, vol. 21, no. 9, p. 2984, 2021. DOI: [10.3390/s21092984](https://doi.org/10.3390/s21092984). [Online]. Available: <https://arxiv.org/abs/2105.13331>.
- [92] J. B. et al, “Quantization and training of neural networks for efficient integer-arithmetic-only inference,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 2704–2713.
- [93] *TensorFlow Lite for Microcontrollers*. [Online]. Available: <https://www.tensorflow.org/lite/microcontrollers>.
- [94] G. A. et al, “A survey of quantization methods for efficient neural network inference,” *arXiv preprint arXiv:2103.13630*, 2021.
- [95] D. H. Le and B.-S. Hua, “Network Pruning That Matters: A Case Study on Retraining Variants,” in *International Conference on Learning Representations*, 2021. [Online]. Available: <https://openreview.net/forum?id=Cb54AMqHQFP>.

- [96] H. T. et al, “Sparsity in Deep Learning: Pruning and growth for efficient inference and training in neural networks.,” *J. Mach. Learn. Res.*, vol. 22, no. 241, pp. 1–124, 2021.
- [97] Y. S. et al, “A Unified Framework of DNN Weight Pruning and Weight Clustering/Quantization Using ADMM,” *CoRR*, 2018. [Online]. Available: <http://arxiv.org/abs/1811.01907>.
- [98] M. Y. et al, “Ladabert: Lightweight adaptation of bert through hybrid model compression,” *arXiv preprint arXiv:2004.04124*, 2020.
- [99] *Data collection website*, <https://mlproject.azurewebsites.net/>, 2023.
- [100] A. Sithara, A. Thomas, and D. Mathew, “Study of MFCC and IHC Feature Extraction Methods with Probabilistic Acoustic Models for Speaker Biometric Applications,” *Procedia computer science*, vol. 143, pp. 267–276, 2018.
- [101] U. E. Akpudo and J.-W. Hur, “A cost-efficient MFCC-based fault detection and isolation technology for electromagnetic pumps,” *Electronics*, vol. 10, no. 4, p. 439, 2021.
- [102] S. Patro and K. K. Sahu, “Normalization: A preprocessing stage,” *arXiv preprint arXiv:1503.06462*, 2015.
- [103] Keras, *BayesianOptimization Tuner*. [Online]. Available: https://keras.io/api/keras_tuner/tuners/bayesian/.