Department Of Computer Science & Engineering

Graduation Project

Arabic Sign Language Recognition System

By

Abdalrhman Magdy Abdalaziz
Islam Ehab Mohamed
Islam Abd-Albaseer Abd-Alazeem
Waleed Mohamed Rehan

Supervisors	Head of the Department	Dean
Dr. / Ahmed El-Mahlawy	Prof. Dr. / Mohamed Berbar	Prof. Dr. / Ayman Elsayed

Arabic Sign Language Recognition System

Acknowledgment:

I would like to express my sincere appreciation to all those who contributed to the successful completion of this project.

First and foremost, I am profoundly grateful to Allah for providing me with the strength, perseverance, and clarity of mind throughout this journey.

I extend my heartfelt thanks to my supervisors and faculty members for their continuous guidance, constructive feedback, and valuable insights, which played a pivotal role in shaping the direction and quality of this work.

My deepest gratitude also goes to my family and friends for their unwavering support, encouragement, and belief in my abilities. Their presence and motivation were a constant source of strength.

Lastly, I acknowledge the collaborative efforts and support of my colleagues and peers, whose contributions, both direct and indirect, were instrumental in the realization of this project.

Abstract

Sign language is a visual language which uses hand gestures, facial expressions, and body movements to communicate, primarily used by the deaf and hard-of-hearing community. However, when it comes to interactions between a deaf "Sign language user" and those who aren't familiar with sign language, barriers and limitations exist.

This project presents a solution to these barriers and limitations, by providing a real-time Arabic Sign Language "ArSL" recognition system using computer vision and deep learning techniques. By recognizing static ArSl gestures, from live webcam input and utilizing MediaPipe framework to extract 21 hand landmarks per frame in real-time, these landmarks are processed by a trained neural network model for classification. After the 21 landmarks are extracted, they would be normalized relative to the wrist point or landmark [0], reducing variation due to hand position and scale. That would result in a 42-feature vector (x and y coordinates for 21 landmarks) and will be passed to a deep learning model (a fully connected Neural Network) built using TensorFlow. With dropout and L2 regularization to ensure robustness and generalization.

The dataset consists of 31 labeled hand gestures for Arabic letters, that will be preprocessed to extract 42-feature vector [x0, y0, x1, y1, x2, y2, ..., x20, y20] in that order. After training, the model is capable of classifying new, unseen hand gestures in real-time. It achieves high classification Val accuracy 97.41%, using only a standard webcam.

Key Contributions:

- A MediaPipe pipeline for efficient hand landmark detection.
- A lightweight deep learning model optimized for ArSL gesture classification.
- A real-time application deployable on desktop and mobile devices.

The resulting system can serve as a foundation for further research in sign language processing, human-computer interaction, and accessible AI technologies.

Table of Content

Table of figures

Table of tables

Abbreviation	Full Term
AI	Artificial Intelligence
ANN	Artificial Neural Network
ArSL	Arabic Sign Language
CNN	Convolutional Neural Network
DL	Deep Learning
FPS	Frames Per Second
GPU	Graphics Processing Unit
LRCN	Long-term Recurrent Convolutional Network
LSTM	Long Short-Term Memory
ML	Machine Learning
MVC	Model-View-Controller
RNN	Recurrent Neural Network
ReLU	Rectified Linear Unit
SRS	Software Requirements Specification
TPU	Tensor Processing Unit

Chapter 1: Introduction

1.1 Background

Communication is a fundamental human right, yet millions of people across the globe face daily challenges due to hearing impairments.

According to the World Health Organization (WHO), over **1.5 billion people globally** which could grow to 2.5 billion by 2050 live with some degree of hearing loss, and approximately **430 million** suffer from disabling hearing loss (WHO, 2021) [1].

The prevalence of hearing loss varies across the six WHO regions, from 3.1% in the Eastern Mediterranean Region approximately 22.1 million, to 7.1% in the Western Pacific Region. The maximum share is contributed by the Western Pacific Region, followed by the South-East Asia Region (Figure 1).

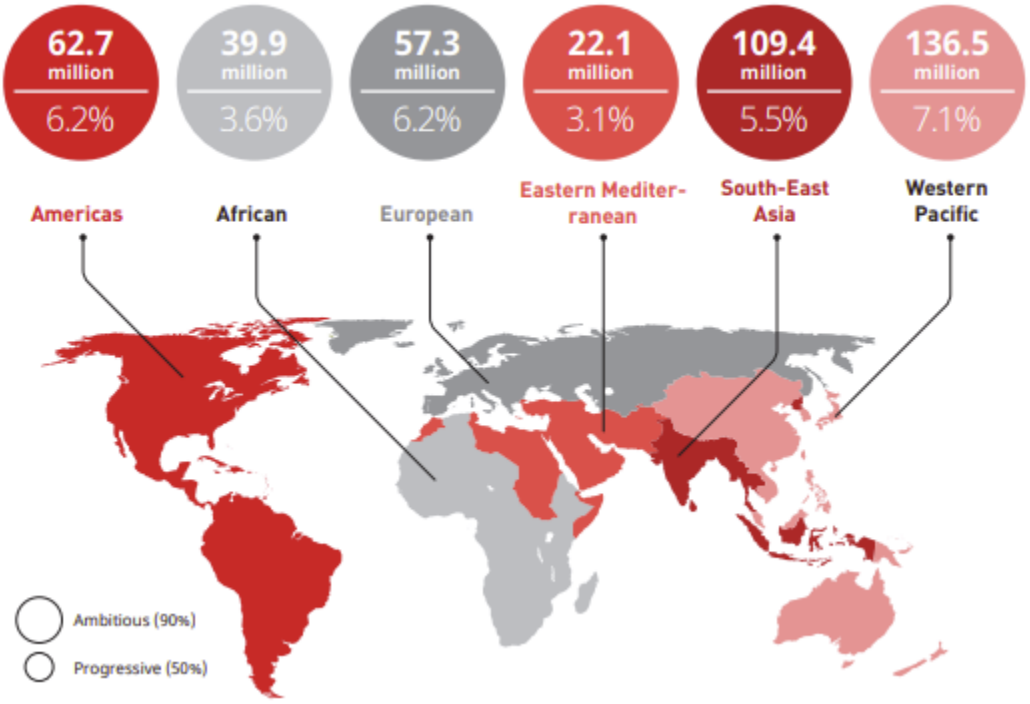


Figure 1: Prevalence of hearing loss (of moderate or higher grade) across WHO regions.

Egypt's national household survey (4,000 individuals across six governorates) reported a **16.0% prevalence of hearing loss** across all age groups.

Notably, hearing loss affected **22.4% of children aged 0–4 years**, and **49.3% of adults aged 65+ years**. Secretory otitis media accounted for **30.8%** of cases, while presbycusis represented **22.7%** [2].

That shows the critical need for inclusive technologies and assistive tools including Arabic Sign Language (ArSL) recognition systems to enhance communication, education, and social participation for deaf individuals especially for children in the Arab world.

A system that combines **hand-tracking (e.g. MediaPipe)** with **deep learning-based gesture classification** could serve as an effective tool in schools and healthcare settings to bridge communication gaps and promote accessibility.

Arabic Sign Language (ArSL) has been the main communication tool for deaf and hard of hearing individuals around the across Arabic-speaking regions. Yet, people who use it face some significant barriers and limitations in day-to-day interaction with the hearing community.

(Figure 2) illustrates the Arabic Sign Language alphabets, Each sign corresponds to a letter in the Arabic alphabet and is represented through distinct static hand gestures.

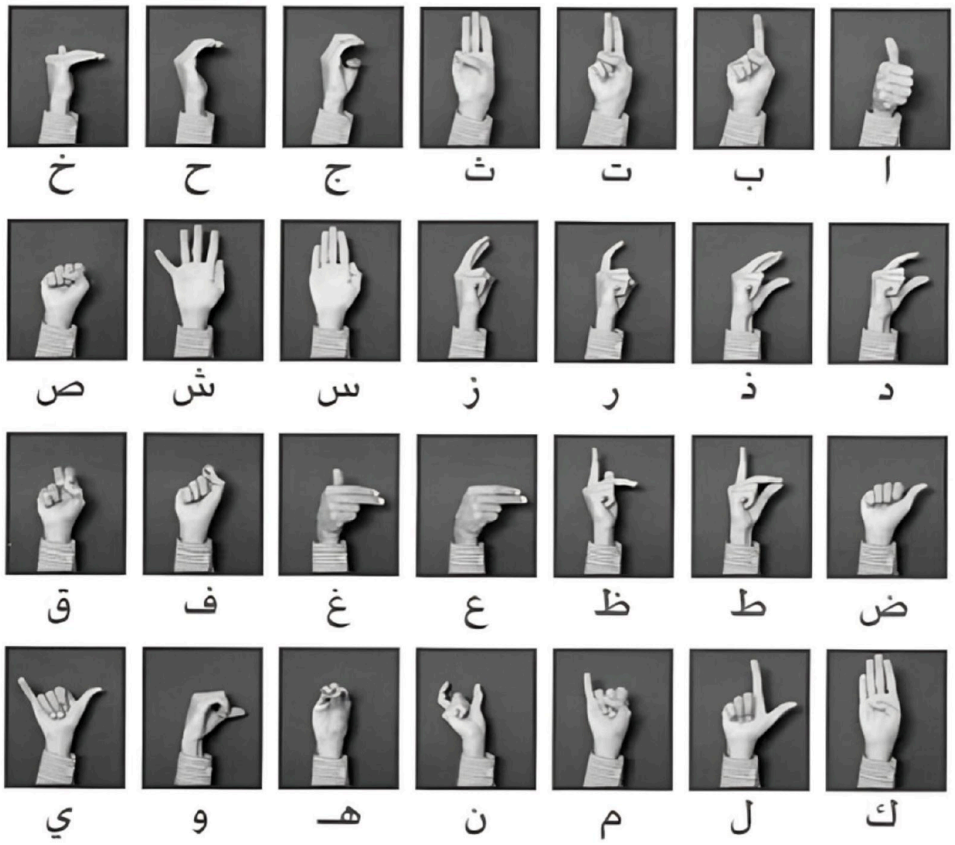


Figure 2: Arabic Sign Language alphabet.

These barriers are the result of the lack of widespread understanding of **ArSL** among the general population, and the lack of tools that can translate sign language in real-time for non-sign language users.

Recognizing these challenges, there is an increasing demand for automated sign language recognition systems capable of understanding gestures in real-time. These systems can and have the potential to help in making education, healthcare, and daily social interaction more accessible and effective for all community members.

Recent advances in computer vision and deep learning have accelerated the development of such recognition systems. Frameworks like **MediaPipe** provide efficient and lightweight pipelines for hand tracking by extracting 21 high-precision 3d hand landmarks per hand.

Coupled with **neural networks**, Can learn the complex gesture patterns form landmark data (keypoints) can be used to classify static hand gestures with high accuracy.

Deep learning, particularly neural networks, has further empowered systems to learn complex gesture patterns from large datasets. Combining landmark-based hand tracking with neural network classification allows for efficient, real-time sign language interpretation, eliminating the need for specialized sensors or equipment.

1.2 Problem Statement:

Despite technological advancements, there remains a significant communication barrier between sign language users and the general population. The primary problem this project addresses is the lack of an accessible, efficient, and accurate system for recognizing Arabic Sign Language in real time using standard devices such as webcams.

Key Challenges:

- Limited publicly available datasets for ArSL.
- High variability in hand shapes, orientations, and sizes.
- Need for real-time performance without expensive hardware.

1.3 System Overview:

The proposed system leverages MediaPipe's hand tracking API to extract 3D hand landmarks. These landmarks are normalized and processed into a 2D feature vector. A trained deep learning model, developed in TensorFlow and converted to TensorFlow Lite, classifies these features into one of 31 predefined ArSL letters.

The mobile app is built using Flutter and communicates with native Android code (Kotlin) through platform channels. The complete pipeline runs on-device without requiring an internet connection or external processing.

This work presents a complete pipeline for **ArSL** recognition using a webcam-based video stream. The core system consists of:

- **Real-time hand landmark extraction** using the MediaPipe Hands solution.
- **Normalization of landmarks** relative to the wrist (landmark 0) to ensure positional invariance.
- **Feature vector generation** comprising 42 dimensions: (x, y) coordinates for each of the 21 hand landmarks.
- **Gesture classification** using a lightweight fully connected neural network trained on 31 static **ArSL** gestures.
- **Live application deployment** on both desktop and Android platforms for real-time feedback .

And the following diagram demonstrate the system overview:

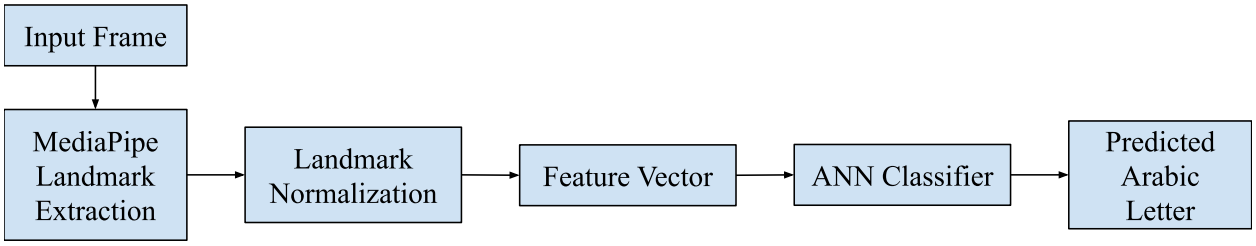


Figure 3: System overview diagram .

Summary:

This chapter introduced the global and regional context of hearing loss, emphasizing its significant prevalence in Arab countries, particularly Egypt. It highlighted the urgent need for assistive technologies that bridge the communication gap between the deaf and hearing communities.

Arabic Sign Language (ArSL), while essential for communication among deaf individuals, is not widely understood by the general population, which poses serious challenges in education, healthcare, and social integration.

With the rise of accessible technologies such as **MediaPipe** for hand tracking and **deep learning models** for classification, it has become feasible to create efficient, real-time ArSL recognition systems that run on ordinary devices like smartphones and laptops. The proposed system addresses this need by combining hand landmark detection with a neural network classifier to recognize Arabic alphabet signs in real time.

In Chapter 2, we critically reviewed existing literature and recent research trends in ArSL recognition. We categorized prior models into CNN-based, transformer-based, and hybrid architectures, comparing them to our proposed approach. While previous work achieved strong accuracy, many required heavy computation and lacked portability.

Chapter 3 introduced the core technologies underpinning the system — namely MediaPipe Hands, artificial neural networks (ANNs), TensorFlow Lite, and the Flutter + Kotlin integration. These components were carefully selected and integrated for their efficiency, speed, and suitability for mobile deployment.

The model architecture and training pipeline were detailed in Chapter 4. By normalizing MediaPipe landmarks and feeding a 42-dimensional vector into a compact ANN, the system achieved 96.5%+ validation accuracy with real-time performance (30+ FPS). This chapter also explained the MVC-based architecture and its role in structuring the app for scalability and modularity.

In Chapter 5, we evaluated the model rigorously using standard ML metrics — accuracy, precision, recall, F1-score, and confusion matrix — along with real-time testing under various conditions. The model maintained strong performance across hand sizes, lighting conditions, and backgrounds, confirming its robustness.

Chapter 2: Previous Work

2.1 Overview:

Arabic Sign Language (ArSL) recognition is a growing research area due to the increasing demand for inclusive communication tools. Recent advancements in computer vision and deep learning have enabled the development of high-accuracy models for gesture classification. This chapter highlights the most relevant studies in ArSL recognition, specifically focusing on Arabic alphabet gesture classification using both convolutional and transformer-based architectures, along with a comparative analysis of their performance and design philosophies in contrast to our proposed system.

Few studies have addressed Arabic Sign Language in depth. In this section, we present a selection of relevant research to highlight the methods and technologies employed in this field.

This chapter surveys recent and relevant works in the field of ArSL recognition. It categorizes these efforts into four primary directions: Convolutional Neural Network (CNN)-based models, Transformer-based architectures, hybrid CNN-RNN approaches, and lightweight mobile solutions. In addition, the chapter discusses dataset-related challenges, system limitations, and concludes with a comparative analysis between prior work and the system proposed in this book.

2.2 CNN-Based Approaches:

Convolutional Neural Networks (CNNs) are among the most widely adopted techniques for image-based gesture recognition.

(Dabwan Basel & Jadhav Mukti, 2021) [3] developed an automated recognition system for Yemeni Sign Language alphabets using Convolutional Neural Networks (CNNs) for feature extraction. The model incorporates fully connected (dense) layers after the convolutional layers to aggregate the extracted features.

These features are then passed through additional dense layers with activation functions for classification. Finally, a SoftMax function is used to classify each image into one of 32 alphabet categories. The system was trained and evaluated on a dataset consisting of 16,192 images collected from 40 individuals, with variations in distance and positioning. Pre-processing techniques were applied to reduce noise and normalize image alignment.

The proposed model achieved an accuracy of 94%, demonstrating its effectiveness in recognizing Yemeni Sign Language alphabets.

2.3 Transformer-Based Approaches:

With the growing popularity of Vision Transformers (ViTs), researchers have begun exploring their use in sign language classification.

(Mazen Balat, Rewaa Awaad, Hend Adel, Ahmed B. Zaky, and Salah A. Aly 2024) [4] developed a broader approach incorporating both CNNs and transformers was proposed in a recent framework that explored several deep learning architectures. These included ResNet50, MobileNetV2, EfficientNetB7, and transformer-based models such as Google ViT and Microsoft Swin Transformer.

The study used the ArSL2018 and AASL datasets, demonstrating the efficacy of combining spatial feature extraction (CNNs) with sequence modeling (transformers). Experimental results showed that this combination achieved 99.6% accuracy on ArSL2018 and 99.43% on AASL.

This research confirms the potential of transfer learning and ensemble models to generalize across variations in sign appearance and capture subtle hand differences using the ArSL2018 dataset. This comprehensive dataset consists of 54,049 grayscale images (64x64 pixels) representing 32 Arabic sign language signs and alphabets.

The images were collected from 40 participants of various age groups in Al Khobar, Saudi Arabia, using an iPhone 6S camera. The dataset includes variations in lighting, angles, and backgrounds to enhance its robustness.

The following figure demonstrate the transfer learning diagram:

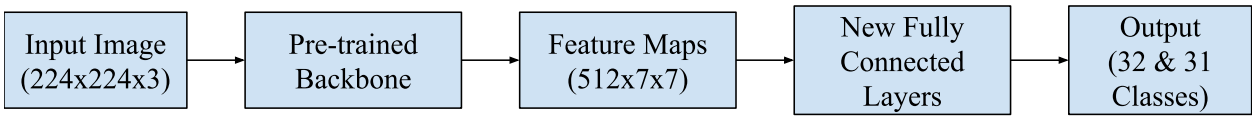


Figure 4: The transfer learning diagram and fine-tuning process for Arabic alphabet sign language recognition.

2.4 Hybrid CNN-RNN Approaches:

To capture both spatial and temporal dependencies in sign language gestures, researchers have proposed hybrid models combining CNNs with Recurrent Neural Networks (RNNs).

Another line of research by **(Basel Dabwan and Mukti Jadhav 2023)** [5] explored the integration of Convolutional Neural Networks (CNNs) with Recurrent Neural Networks (RNNs) to effectively capture both spatial and temporal features in Arabic Sign Language recognition. The study proposed two hybrid models: ConvLSTM and Long-term Recurrent Convolutional Networks (LRCN).

ConvLSTM extends convolutional operations into the temporal domain, allowing the model to capture sequential movement patterns within sign gestures. On the other hand, LRCN connects CNN-extracted feature maps to LSTM layers, enabling the learning of long-term temporal dependencies across frames. The models were trained on a dataset consisting of 28 Arabic Sign Language classes.

ConvLSTM achieved a training accuracy of 99.66% and a validation accuracy of 95%, while LRCN reached 99.5% training accuracy and 93.33% validation accuracy. These results underscore the effectiveness of combining spatial and temporal modeling for dynamic sign language recognition.

2.5 Lightweight Mobile Solutions:

Efforts have also been made to adapt ArSL recognition systems for real-world deployment on mobile devices.

(Batool Yahya AlKhuraym, Mohamed Maher Ben Ismail, and Ouiem Bchir 2022) [6] developed an Arabic Sign Language (ArSL) recognition system using a lightweight CNN-based architecture tailored for mobile deployment.

A real-world dataset was collected from over 20 participants performing hand gestures corresponding to 30 Arabic alphabet signs, with 10 samples per letter, totaling 5,400 images.

The study investigated various preprocessing and data augmentation techniques to improve model generalization. Several lightweight EfficientNet-Lite models were evaluated, and the best performance was achieved using EfficientNet-Lite 0 in combination with the Label Smoothing loss function. This model obtained 94.3% accuracy.

2.6 Dataset Challenges and Preprocessing:

One of the main challenges in ArSL research is the limited availability of large, diverse, and well-annotated datasets. The ArSL2018 dataset remains a benchmark due to its variety and size (54,049 images), but it primarily contains static grayscale images. The AASL dataset expands this by including colored and varied background images, contributing to better model generalization.

Preprocessing techniques play a vital role in optimizing model input. In most studies, raw images undergo resizing, grayscale conversion, and normalization. In our method, rather than using raw pixel data, we use 21 key landmarks extracted via MediaPipe. These landmarks are normalized relative to the wrist joint to eliminate variance due to hand location, scale, or orientation—enhancing the model's robustness and inference speed.

2.7 Limitations of Current Approaches :

Despite high accuracy, most state-of-the-art models are not optimized for real-time deployment. ViT-based and ensemble CNN-transformer models demand substantial

computational resources and are prone to overfitting without extensive data augmentation. Furthermore, few studies address the model's ability to handle diverse skin tones, hand sizes, and occlusions common in real-world scenarios.

Another overlooked issue is the adaptability of these models to dynamic signs or regional variations in ArSL. Most datasets capture only static letters, limiting model usability in full sentence translation. This gap underlines the importance of lightweight, adaptable models like ours, which serve as a foundation for extending dynamic gesture sequences.

2.8 Comparative Analysis :

The following table shows the key difference between the Proposed Model and Previous Studies.

Table 1: Comparison between the proposed model and other studies .

Aspect	**Our Project**	**Previous Studies (e.g. ViT, CNN, CNN+LSTM, etc.)**
Input	42 normalized hand landmarks (x0, y0, ..., x20, y20)	Raw RGB or grayscale images (64×64), sequences for LSTM
Model Architecture	Fully connected neural network (4 layers + BatchNorm + Dropout)	CNNs (e.g., ResNet, EfficientNet), CNN-LSTM hybrids, Vision Transformers
Model Size	6,271 total parameters (24.5 KB)	Millions of parameters in CNNs (ResNet50 ≈ 23M), ViT-B ≈ 86M
Inference Time	Real-time on CPU/mobile with MediaPipe + ANN	Requires GPU or TPU for real-time inference
Training Dataset	Custom dataset, 31 Arabic letters, 7,857 raw and fully labelled RGB images	AASL (7,856 images), ArSL2018 (54,049 images)
Accuracy	97%	ViT: up to 99.6%, CNN+LSTM: ~95%, CNN-Lite: ~94%
Class-wise F1-Score	Most classes: **0.95–1.00**, lower scores on underrepresented classes (e.g., class 8, 22, 23, 24)	Higher accuracy across large datasets, but often dependent on data quantity and diversity
Strengths	Lightweight, Real-time, High generalization, Mobile-ready	High accuracy, Deep semantic learning

While prior works excel in accuracy using high-capacity models or video-based learning, they often require extensive computational resources, high-resolution image input, and GPU acceleration. Our system, in contrast, uses lightweight architecture and efficient landmark extraction (MediaPipe) to perform gesture recognition in real time on consumer hardware.

Unlike CNN or ViT-based systems that rely on raw pixel data, our approach processes normalized hand landmarks (42 features: x0, y0, ..., x20, y20), reducing sensitivity to lighting and background. This makes the system especially suitable for mobile applications.

Summary :

This chapter reviewed the current state of Arabic Sign Language recognition research, highlighting diverse deep learning approaches.

Across the current literature, deep learning—especially transformer models and CNN-LSTM hybrids—has yielded state-of-the-art results in ArSL recognition. However, these models tend to be computationally intensive and require significant training data and tuning. In contrast, our system achieves competitive performance while being accessible, portable, and efficient.

By focusing on landmark-level features instead of raw image processing, the proposed model strikes a balance between simplicity, speed, and accuracy, offering a practical alternative for real-time, low-cost Arabic Sign Language interpretation.

Chapter 3: Theoretical Foundations

3.1 Overview:

This chapter delves into the essential tools and theoretical foundations of our Arabic Sign Language (ArSL) recognition system: MediaPipe Hands for real-time hand landmark detection and artificial neural networks (ANNs) for gesture classification. We explore the design principles, practical architectures, and integration strategies that enable efficient, accurate inference on consumer hardware.

3.2 MediaPipe Framework:

MediaPipe is an open-source cross platform framework for building pipelines to process perceptual data of different modalities, such as video and audio. This approach provides high-fidelity hand and finger tracking by employing machine learning (ML) to infer 21 3D keypoints of a hand from just a single frame.[7]

The Hand Landmarker uses a model bundle with two packaged models: a palm detection model and a hand landmarks detection model. You need a model bundle that contains both these models to run this task.[8]

The hand landmark model bundle detects the keypoint localization of 21 hand-knuckle coordinates within the detected hand regions. The model was trained on approximately 30K real-world images, as well as several rendered synthetic hand models imposed over various backgrounds.

MediaPipe Hands achieves 30+ fps on standard laptops and mobile devices. According to Google, the pipeline can run at 20–30 fps on mid-range phones, thanks to its parallel processing and optimized inference graph. This makes it ideal for gesture-based applications without GPU dependencies.

The following figure shows the keypoints in the hand landmarks:

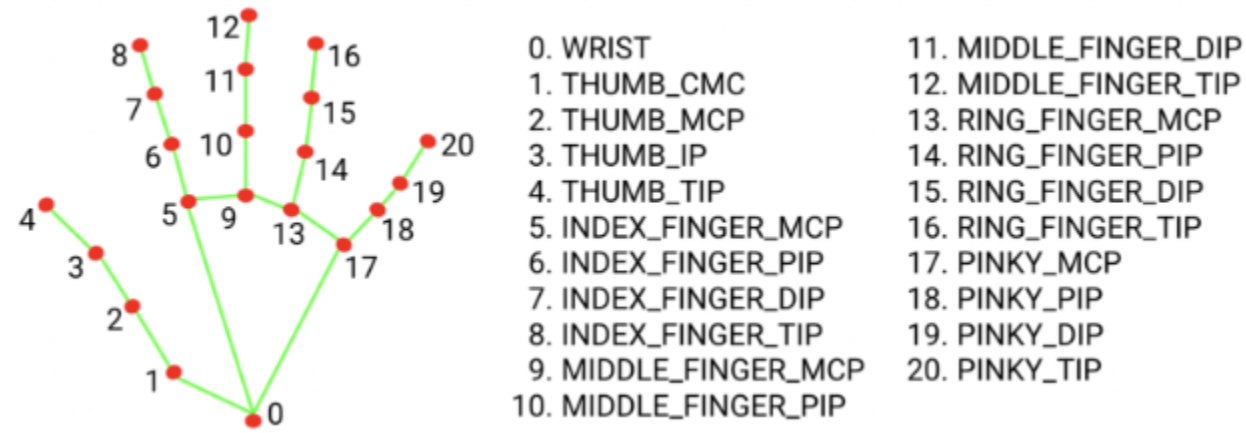


Figure 5: A visual representation of Landmarks.

The hand landmarker model bundle contains a palm detection model and a hand landmarks detection model. The Palm detection model locates hands within the input image, and the hand landmarks detection model identifies specific hand landmarks on the cropped hand images.

The following diagram shows the pipeline overview of mediapipe framework for extracting landmarks:

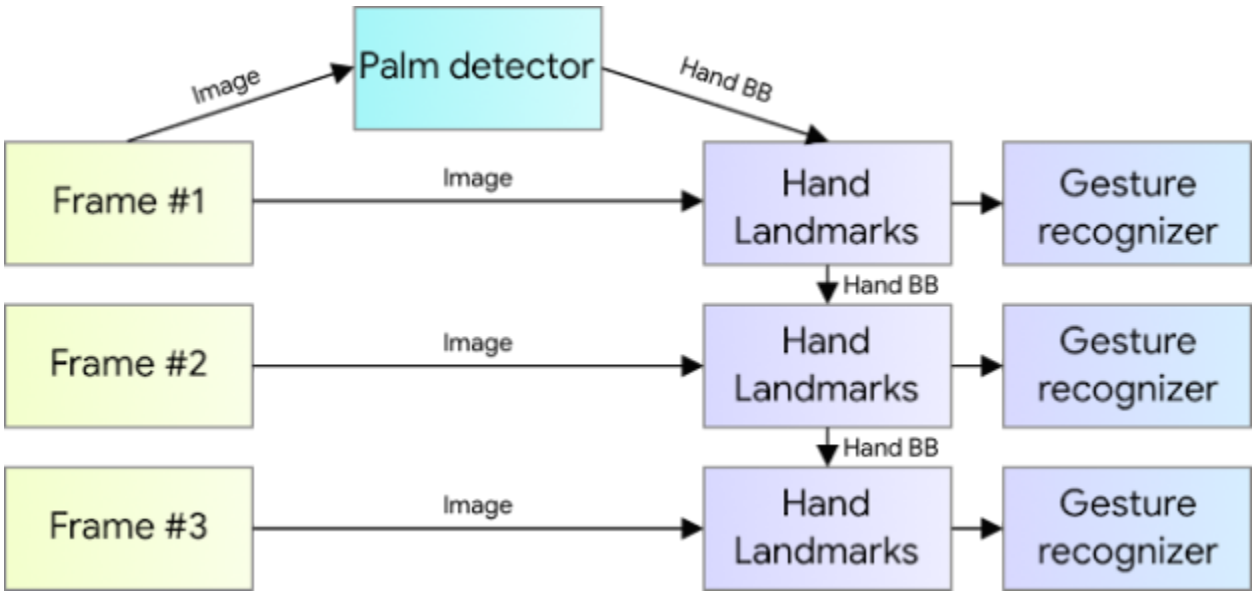


Figure 6: Hand perception pipeline overview.

Table 2: MediaPipe Config options.

Option Name	Description	Value Range	Default Value
running_mode	Sets the running mode for the task. There are three modes: • **IMAGE:** The mode for single image inputs • **VIDEO:** The mode for decoded frames of a video • **LIVE_STREAM:** The mode for a livestream of input data, such as from a camera. In this mode, resultListener must be called to set up a listener to receive results asynchronously	{IMAGE, VIDEO, LIVE_STREAM}	IMAGE
num_hands	The maximum number of hands detected by the Hand landmark detector	Any integer > 0	1
min_hand_detection_confidence	The minimum confidence score for hand detection to be considered successful in palm detection model	0.0 - 1.0	0.5
min_hand_presence_confidence	The minimum confidence score for the hand presence score in the hand landmark detection model. In Video mode and Live stream mode, if the hand presence confidence score from the hand landmark model is below this threshold, Hand Landmarker triggers the palm detection model. Otherwise, a lightweight hand tracking algorithm determines the location of the hand(s) for subsequent landmark detections	0.0 - 1.0	0.5
min_tracking_confidence	The minimum confidence score for hand tracking to be considered successful. This is the bounding box IoU threshold between hands in the current frame and the last frame. In Video mode and Stream mode of Hand Landmarker, if the tracking fails, Hand Landmarker triggers hand detection. Otherwise, it skips hand detection	0.0 - 1.0	0.5
result_callback	Sets the result listener to receive the detection results asynchronously when the hand landmarker is in live stream mode. Only applicable when running mode is set to LIVE_STREAM	-	-

3.2 MediaPipe for Android Integration:

Google provides a fully integrated Android solution for MediaPipe Hands through the MediaPipe Tasks Vision API. This solution includes pre-trained models for real-time hand landmark detection and is optimized for Android devices.

The HandLandmarker API allows developers to implement high-speed, real-time hand tracking in applications using standard Android camera sources like CameraX or ImageProxy. The API handles model loading, frame preprocessing, and outputs landmark results with associated handedness and tracking confidence.

Key Features:

- **On-device inference:** Runs without an internet connection, ideal for privacy-focused or offline apps.
- **21 hand landmarks per hand** in normalized 3D space (x, y, z).
- **Support for multiple hands:** The API can detect and track up to 2 hands simultaneously.
- **Real-time performance:** Achieves 30+ fps on mid-tier Android devices.

Typical Usage:

- Creating a HandLandmarker using a HandLandmarkerOptions builder.
- Feeding video frames via Bitmap or ImageProxy to the detectAsync() or detect() methods.
- Retrieving results from the HandLandmarkerResult, which includes:
    - handLandmarks (list of hand keypoints).
    - handednesses (left/right prediction).
    - handWorldLandmarks (optional 3D coordinates).

This API dramatically reduces the development effort needed to deploy hand gesture interfaces and is the foundation for the real-time Arabic Sign Language recognition system.

3.3 Fundamentals of Neural Networks:

ANNs are inspired by brain frameworks—collections of neurons and synapses. Each neuron takes weighted inputs, applies an activation function, and produces an output passed to the next layer.[9]

1- The Building Blocks of Neural Networks:

The basic unit of a neural network is the artificial neuron. Each neuron receives one or more inputs, each multiplied by a corresponding weight, and adds a bias term. The sum is then passed through an activation function, which determines the output of the neuron.

The following figure shows the structure of a simple neuron:

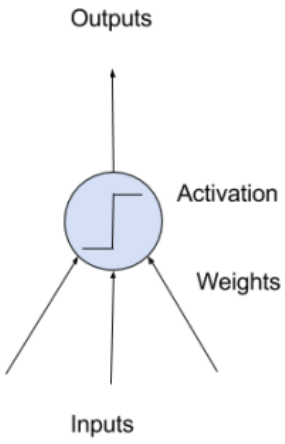


Figure 7: A Simple Neuron.

The weights function similarly to coefficients in linear regression and are crucial to determining how input data is transformed at each layer. Weights are usually initialized to small random values and are adjusted during training to minimize prediction errors. Bias is a special kind of input that is always "on" and allows the model to shift its predictions.

Activation functions introduce non-linearity into the network, enabling it to learn more complex mappings from inputs to outputs. Historically, step functions were used, but modern neural networks use functions such as:

- **Sigmoid:** Outputs values between 0 and 1 .
- **Tanh:** Outputs values between -1 and 1.
- **ReLU (Rectified Linear Unit):** Outputs 0 for negative inputs, and the input itself for positive values.

ReLU and its variants are now the most commonly used in deep learning models due to their simplicity and efficiency.

Neural networks are organized into layers:

- **Input (Visible) Layer:** Receives the raw data.
- **Hidden Layers:** Perform intermediate computations.
- **Output Layer:** Produces the final prediction or classification result.

The term depth refers to the number of layers in the network. Models with many hidden layers are called deep neural networks, and the practice of using these is referred to as deep learning.

The following figure shows the structure of a simple neural network:

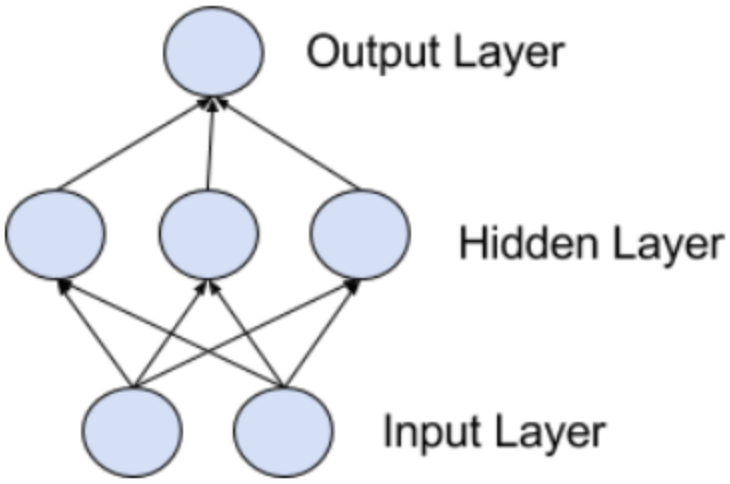


Figure 8: A simple Neural Network.

2- Training Neural Networks:

Neural networks are trained using stochastic gradient descent (SGD), an optimization algorithm that adjusts weights to reduce prediction errors on the training data.

The training process includes:

1. **Forward pass:** Inputs are propagated through the network to generate predictions.
2. **Loss computation:** The error between predicted and actual output is calculated.
3. **Backpropagation:** The error is distributed backward through the network to compute gradients.
4. **Weight update:** Weights are adjusted to minimize loss.

This process is repeated over many epochs until the model achieves good accuracy.

Preparing data is an essential part of training. Data normalization, cleaning, and encoding are often required to help the model learn more efficiently.

3 The Promise and Future of AI:

While deep learning has made impressive strides, much of its potential remains untapped. Many deep learning breakthroughs are still confined to research and have not yet been widely adopted in fields like healthcare, finance, or education.

Today, AI applications are visible in tools such as:

- Voice assistants (e.g., Siri, Google Assistant).
- Recommendation engines (e.g., Amazon).
- Automated photo categorization (e.g., Google Photos).

However, these applications are still peripheral to our daily lives. AI has not yet become central to how we work or live—similar to how the internet in the 1990s had limited everyday relevance before its full impact was realized.

In the future, AI will be more than a tool. It will be:

- **Your assistant,** helping with tasks and decision-making.
- **Your tutor,** supporting education and personal growth.
- **Your guardian,** monitoring health and safety.
- **A scientific partner,** contributing to breakthroughs in genomics, physics, and mathematics.

AI will become the interface to an increasingly complex world, enabling humanity to navigate and understand vast amounts of information with ease.

3.4 TensorFlow and TensorFlow Lite:

TensorFlow is an open-source end-to-end platform developed by Google for machine learning and deep learning applications. In this project, TensorFlow was employed during the model development phase to define, train, and validate a fully connected neural network (FCNN) designed to classify Arabic Sign Language (ArSL) gestures based on hand landmarks.

After training, deploying the full TensorFlow model on resource-constrained mobile devices (e.g., smartphones) would lead to inefficiencies in both memory and computation. To solve this, **TensorFlow Lite (TFLite)** was used to convert the trained model into a mobile-optimized format using a process called **model quantization**.

Key Benefits of TensorFlow Lite:

- **Reduced model size:** Using 8-bit quantization, the model size is significantly reduced (by 3–4×), which enables faster downloads and less memory usage.
- **Optimized inference:** TFLite models are compiled into flatbuffer format that allows native execution with low latency (<50 ms).
- **Hardware acceleration support:** TFLite leverages device-specific acceleration (e.g., NNAPI, GPU delegate, Edge TPU) for real-time inference.

Workflow:

- The model is trained in TensorFlow using Python.
- The .h5 file is converted into tflite.
- The .tflite file is then loaded into the Android app using Kotlin's TensorFlow Lite Interpreter API.

The interpreter processes a 42-dimensional vector extracted from the hand landmarks and returns a 31-class probability distribution representing the predicted Arabic letter.

3.5 Flutter and MethodChannel:

Flutter is a cross-platform UI toolkit also developed by Google, which enables developers to create high-performance, visually consistent applications for Android, iOS, web, and desktop using a single codebase written in Dart.

In this project, Flutter was selected for front-end development due to its reactive widget system, expressive UI capabilities, and efficient GPU-accelerated rendering pipeline. However, Flutter alone cannot access device-specific APIs like the Android camera or the MediaPipe SDK natively written in Kotlin/Java.

To bridge this gap, Flutter uses **MethodChannel**, a platform channel that allows Dart code to communicate with native Android (Kotlin/Java) or iOS (Swift/Objective-C) code.

MethodChannel Architecture:

Table 3: MediaPipe Config options .

Layer	Language	Responsibility
Flutter UI	Dart	Capture camera input, show output, display letters
MethodChannel	Dart ↔Kotlin	Pass data between Flutter and native code
Android Core	Kotlin	Handle image decoding, MediaPipe processing, TFLite inference

Frame Processing via Kotlin:

- The Dart code captures a CameraImage frame using the camera plugin.
- The frame is converted to a byte array and sent .

Landmark Extraction with MediaPipe:

- Kotlin receives the image, decodes it using BitmapFactory, builds an MPImage, and sends it to the HandLandmarker class.
- 21 landmark coordinates are returned to Dart for further preprocessing and normalization.

TFLite Model Inference:

- After normalization, Dart sends the 42-dimension input vector to Kotlin via another channel.
- Kotlin uses Interpreter.run() to perform classification and sends the result back to Dart.

Real-time Letter Display:

- The predicted letter index is interpreted in Dart and displayed live in the app's UI.

Advantages of This Architecture:

- Combines Dart's cross-platform flexibility with native Android performance.
- Enables real-time sign recognition on mobile using low-latency native pipelines.
- Keeps UI decoupled from heavy computation logic (clean MVC separation).

Summary:

This chapter presents the theoretical underpinnings and technical components that form the foundation of the Arabic Sign Language (ArSL) recognition system. The system relies on two key technologies: Google's **MediaPipe Hands**, which provides real-time hand tracking and landmark estimation, and **Artificial Neural Networks (ANNs)**, which classify extracted hand gestures into Arabic sign language letters.

The chapter begins by introducing the MediaPipe framework, detailing how it detects 21 precise hand landmarks through a combination of palm detection and landmark regression models. This real-time computer vision pipeline is central to the system's ability to function on mobile devices without external computation or server dependencies.

Next, the chapter explores the structure of neural networks, highlighting how layers of interconnected artificial neurons process and learn from normalized landmark coordinates. The fundamentals of neuron structure, activation functions, and backpropagation are explained to illustrate how the system learns to classify hand poses accurately.

The role of **TensorFlow** is then examined, emphasizing its use during model training, while **TensorFlow Lite (TFLite)** is discussed in the context of deploying lightweight neural models on smartphones. The conversion process, benefits of model quantization, and execution speed are highlighted to showcase the system's real-time capabilities.

Chapter 4: Proposed Model

This chapter describes in detail the architecture, training, and deployment of the model used for Arabic Sign Language recognition. It focuses on how keypoints from hand gestures are processed through a deep learning model to predict specific signs in real time.

4.1 Overview:

The goal of the model is to classify static hand gestures captured through a webcam into Arabic Sign Language letters.

The model operates in real time and is composed of three main components:

- Hand landmark detection using MediaPipe.
- Preprocessing of landmark data to normalize input.
- Gesture classification using a trained neural network model built with TensorFlow/Keras.

This combination allows for fast, accurate recognition of hand gestures, making the system suitable for interactive applications such as communication aids and educational tools.

4.2 Data Preprocessing:

1- Landmark Extraction:

To detect hand gestures, the system uses MediaPipe Hands, a powerful and lightweight hand tracking library. For each frame captured by the webcam, MediaPipe extracts 21 hand landmarks, each with (x, y) coordinates. These landmarks represent different key points on the hand, including fingertips, joints, and wrist.

The following code block shows the initialization of MediaPipe Hands to determine some attributes like number of hands to detect and the detection confidence.

```
# Initialize MediaPipe Hands
mp_hands = mp.solutions.hands
hands = mp_hands.Hands(static_image_mode=False, max_num_hands=1, min_detection_confidence=0.5)
mp_draw = mp.solutions.drawing_utils
```

Code block 1: initialization of MediaPipe Hands.

The next code block declares a function that extract the x,y landmarks form the dataset

```
# Function to extract landmarks (only x and y) from an image
def extract_landmarks(image_path):
    image = cv2.imread(image_path)
    if image is None:
        print(f"Warning: Unable to read image {image_path}")
        return None

    image_rgb = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
    results = hands.process(image_rgb)

    if results.multi_hand_landmarks:
        for hand_landmarks in results.multi_hand_landmarks:
            landmarks = [(lm.x, lm.y) for lm in hand_landmarks.landmark]

            # Ensure exactly 21 landmarks are extracted
            if len(landmarks) == 21:
                return normalize_landmarks(landmarks)

    return None  # No hands detected
```

Code block 2: Extracting the 21 landmarks.

2- Landmark Normalization:

The extracted landmarks are position dependent

To remove variations due to hand location in the frame, a normalization technique is applied:

- The base point (landmark 0, typically the wrist) is used as a reference.
- Each landmark's coordinates are subtracted from the base point to center the hand at the origin.

This following code block is to normalize the landmarks to the wrist point.This makes the model focus on the shape of the hand rather than its location in the image.

```
def preprocess_landmarks(landmarks):
    base_x, base_y = landmarks[0]
    return [(x - base_x, y - base_y) for x, y in landmarks]
```

Code block 3: normalize landmarks relative to the wrist (index 0).

The resulting 21 normalized (x, y) pairs are then flattened into a 42-dimensional input vector for the model. Shown in the following code block .

```
if len(landmarks) == 21:
            normalized = preprocess_landmarks(landmarks)
            input_data = np.array([coord for point in normalized for coord in point], dtype=np.float32)
            input_data = np.expand_dims(input_data, axis=0)
```

Code block 4: flatten the data for the neural network .

1- Landmarks After Extraction:

[(0.5598345994949341, 0.7049738764762878), (0.5327727794647217, 0.6024600267410278),
(0.5315549373626709, 0.4875621199607849), (0.5383646488189697, 0.3988003134727478),
(0.5475443601608276, 0.31770920753479004), (0.6082483530044556, 0.5230398774147034),
(0.4822901785373688, 0.533023476600647), (0.4913699626922607 4, 0.5616823434829712),
(0.5328744649887085, 0.5677172541618347), (0.6290332674980164, 0.6025397181510925),
(0.474106103181839, 0.6233812570571899), (0.5042513608932495, 0.6358939409255981),
(0.5520612001419067, 0.6324553489685059), (0.6328900456428528, 0.6854544878005981),
(0.48903942108154297, 0.7025488615036011), (0.5163732767105103, 0.7086188197135925),
(0.562234103679657, 0.7022830247879028), (0.625950038433075, 0.7665921449661255),
(0.5173087120056152, 0.7757366895675659), (0.5362218022346497, 0.7768163084983826),
(0.5767930150032043, 0.7740278244018555)].

2- Landmarks After Normalization:

[(0.0, 0.0), (-0.027061820030212402, -0.10251384973526001),
(-0.028279662132263184, -0.21741175651550293), (-0.021469950675964355,-0.30617356300354004),
(-0.012290239334106445, -0.3872646689414978), (0.048413753509521484, -0.18193399906158447),
(-0.07754442095756531, -0.17195039987564087), (-0.06846463680267334, -0.14329153299331665),
(-0.026960134506225586, -0.13725662231445312), (0.06919866800308228, -0.10243415832519531),
(-0.08572849631309509, -0.0815926194190979), (-0.05558323860168457, -0.0690799355506897),
(-0.007773399353027344, -0.07251852750778198), (0.0730554461479187, -0.019519388675689697),
(-0.07079517841339111,-0.0024250149726867676),(-0.04346132278442383,0.0036449432373046875),
(0.002399504184722 9004,-0.0026908516883850098),(0.06611543893814087,0.061618268489837646),
(-0.04252588748931885, 0.07076281309127808), (-0.023612797260284424, 0.07184243202209473),
(0.016958415508270264, 0.06905394792556763)].

3- Model Input Data:

[[0. 0. -0.02706182 -0.10251385 -0.02827966 -0.21741176
 -0.02146995 -0.30617356 -0.01229024 -0.38726467 0.04841375 -0.181934
 -0.07754442 -0.1719504 -0.06846464 -0.14329153 -0.02696013 -0.13725662
  0.06919867 -0.10243416 -0.0857285 -0.08159262 -0.05558324 -0.06907994
 -0.0077734 -0.07251853 0.07305545 -0.01951939 -0.07079518 -0.00242501
 -0.04346132 0.00364494 0.0023995 -0.00269085 0.06611544 0.06161827
 -0.04252589 0.07076281 -0.0236128   0.07184243 0.01695842 0.06905395]].

4.3 Model Architecture:

The classification model is a fully connected neural network designed to work with the 42-dimensional input vector.

It consists of:

- **Input Layer:** Accepts the 42-dimensional feature vector.
- **Hidden Layers:** Two or more Dense layers with ReLU activation to capture non-linear patterns.
- **Dropout Layers:** Added after hidden layers to prevent overfitting.
- **Output Layer:** A softmax layer corresponding to the number of hand signs (e.g., 30 for Arabic letters).

The model is trained using:

- **Loss Function:** Categorical Crossentropy.
- **Optimizer:** Adam.
- **Metrics:** Accuracy.

The following code blocks describes the Model Architecture:

```
model = tf.keras.models.Sequential([
    tf.keras.layers.Input((42,)),  # 21 landmarks * 2 (x, y)
    tf.keras.layers.Dense(64, activation='relu'),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.Dropout(0.1),

    tf.keras.layers.Dense(32, activation='relu',
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.Dropout(0.1),

    tf.keras.layers.Dense(16, activation='relu'),

    tf.keras.layers.Dense(NUM_CLASSES, activation='softmax')
])
```

Code block 5: Model Architecture.

The following figure shows in a detail the model architecture and the number of parameters :

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 64)                2752

 batch_normalization (Batch  (None, 64)                256
 Normalization)

 dropout (Dropout)           (None, 64)                0

 dense_1 (Dense)             (None, 32)                2080

 batch_normalization_1 (Bat  (None, 32)                128
 chNormalization)

 dropout_1 (Dropout)         (None, 32)                0

 dense_2 (Dense)             (None, 16)                528

 dense_3 (Dense)             (None, 31)                527

=================================================================
Total params: 6271 (24.50 KB)
Trainable params: 6079 (23.75 KB)
Non-trainable params: 192 (768.00 Byte)
_________________________________________________________________
```

Figure 9: model summary.

4.4 Training Process:

Training is performed using labeled datasets containing multiple samples of each hand gesture. The preprocessing step is applied to each sample, and then the data is used to train the model.

Training Configuration:

- **Validation Data:** Explicit split into X_test and y_test
- **Callbacks:**
    - **ModelCheckpoint:** Saves the best performing model based on validation accuracy.
    - **EarlyStopping:** Stops training if no improvement is seen for 20 consecutive epochs.
- **Loss Function:** sparse_categorical_crossentropy (used because class labels are integers rather than one-hot encoded).
- **Batch Size:** 128 (larger batches for more efficient training).
- **Evaluation:** Plots of accuracy and loss were generated to verify appropriate learning without overfitting.

The following code block shows the training process of the model :

```
cp_callback = tf.keras.callbacks.ModelCheckpoint(model_save_path, verbose=1, save_weights_only=False)
es_callback = tf.keras.callbacks.EarlyStopping(patience=30, verbose=1)
reduce_lr = tf.keras.callbacks.ReduceLROnPlateau(
    monitor='val_accuracy', # Monitor validation accuracy
    factor=0.2,             # Reduce learning rate by 20%
    patience=5,
    min_lr=0.00001,         # Lower bound on the learning rate
    mode='max',             # Look for maximum val_accuracy
    verbose=1               # Print messages when reducing LR
)
model.compile(
    optimizer='adam',
    loss='sparse_categorical_crossentropy',
    metrics=['accuracy']
)
model.fit(X_train, y_train, epochs=1000, batch_size=32, validation_data=(X_test, y_test),
    callbacks=[cp_callback, es_callback, reduce_lr ]
)
```

Code block 6: training process.

4.5 Deployment:

1- (app.py) :

Deployment is handled through a Python script (app.py) which connects all components:

- Load the model & Load label names
- Initialize MediaPipe Hands
- Captures video using OpenCV.
- Extracts and normalizes hand landmarks using MediaPipe.
- Passes the input vector to the trained model.
- Displays the prediction on the screen.

2- (Android Deployment) :

Requirements:

- Hand Landmark detection via MediaPipe Tasks API.
- Adaptive UI for multiple screen sizes.
- Dark/Light theme support.

Performance: Optimized for 30+ FPS on mid-tier devices.

The following table shows the Software Requirements Specification (SRS):

Table 4: Software Requirements Specification (SRS).

Use Case	Main Actor	Description	Priority	Stability
Hand LandMarks Detection				
Detect hand landmarks	user	- User show a pattern in front of camera - System send pattern to ai model - System show ai response on screen	**Mandatory**	**Stable**
Theming				
Dark mode support	user	- App must support dark mode and light mode - User can change themes modes	**Desirable**	**Stable**
Responsive And Adaptive UI				
Response	system	- System must compatible with all screen sizes	**Mandatory**	**Stable**
Adaptive	system	- System must be adaptive with different platforms and OS	**Mandatory**	**Stable**
Performance				
Fast response	system	- System must be fast	**Mandatory**	**Stable**

The system architecture follows a well-established **Model–View–Controller (MVC)** design pattern, ensuring a clean separation between user interface (UI), business logic, and data handling. This modular structure improves code maintainability, scalability, and testability. The entire pipeline from camera input to Arabic letter prediction is represented in the following layered diagram and its component interactions.

The following diagram for the system design and architecture :

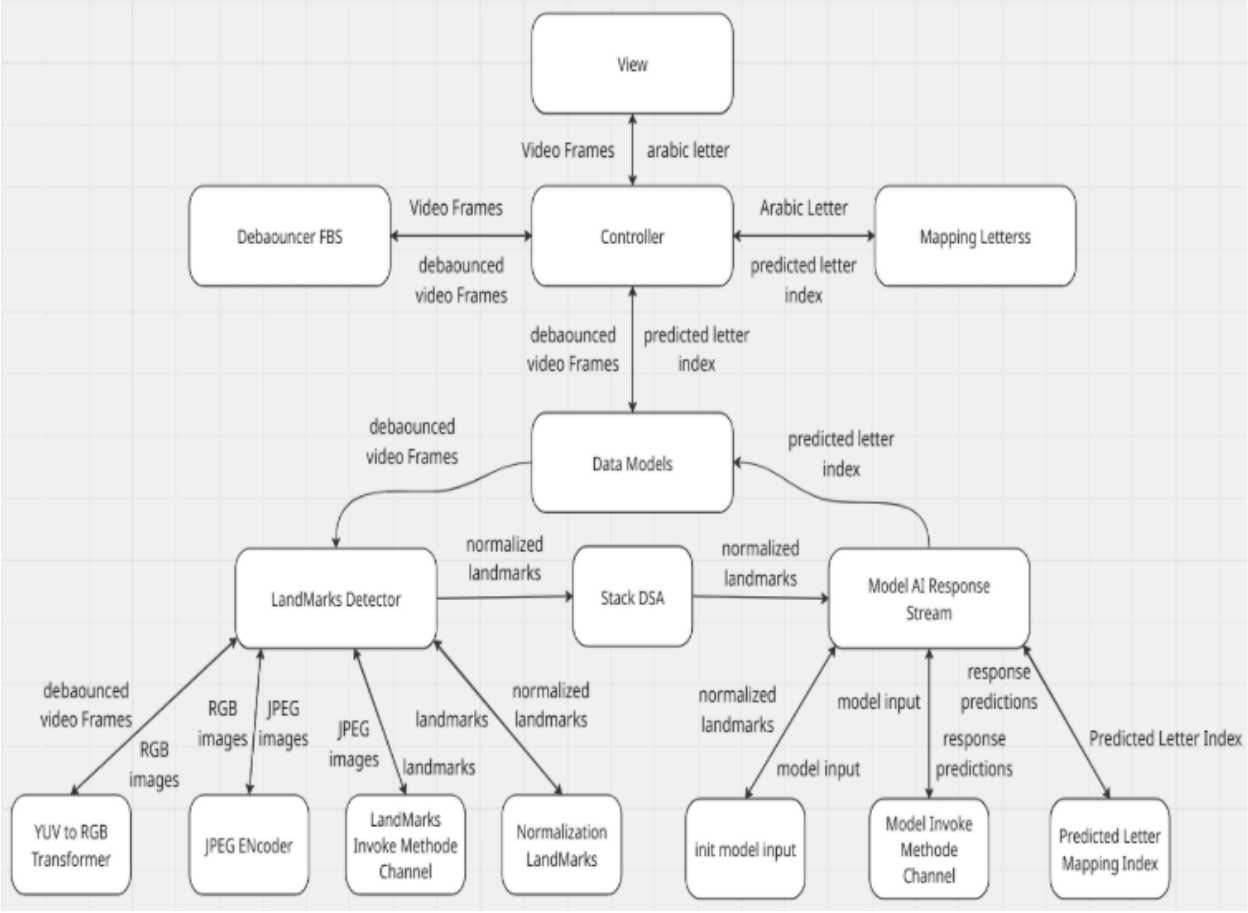


Figure 10: system design and architecture, Model-View-Controller MVC.

View Layer:

- **Purpose**: Acts as the presentation interface to the user.
- **Responsibilities**:
    - Displays live camera feed.
    - Renders the predicted Arabic letter in real time.
- **Components**:
    - Flutter Widgets (VideoRecordingWidget, MobileTextResponseWidget, TabletTextResponseWidget).

      UI refreshes occur via BlocBuilder in response to state changes from the controller.

Controller Layer:

- **Purpose**: Serves as the decision-making unit and event orchestrator.

  Responsibilities:
    - Accepts video frames from the View and debounces them using Debouncer FBS to limit frame frequency.
    - Coordinates interactions between models, invokes landmark detection, model inference, and maps predicted indices to letters.
- **Key Element**:
    - HandmarkDetectionCubit (Flutter BLoC): Central controller handling the communication between View and Model.
- **Mapping Letters:**
        - Maps predicted index to Arabic letter using a predefined index–letter map.
        - Sends the result back to the controller for UI rendering.

Model Layer:

- **Purpose**: Handles data manipulation, feature extraction, and ML inference.
- **Subcomponents**:
    - **YUV to RGB Transformer**: Converts camera frames (CameraImage) into RGB format for processing.

- **JPEG Encoder**: Encodes RGB images into JPEG for Kotlin-native consumption.
- **LandMarks Detector**:
  - Communicates with the Android-native MediaPipe hand tracking model using a MethodChannel.
  - Returns 21 key hand landmarks per frame.
- **Normalization LandMarks:**
  - Transforms raw coordinates by subtracting the wrist reference point.
  - Ensures spatial invariance and consistency across frames.
- **Stack DSA (Data Structure):**
  - A custom stack is used to queue normalized landmarks.
  - Enables asynchronous frame processing and inference.
- **Model AI Response Stream:**
  - Periodically pops landmark vectors from the stack and prepares model input (42-dimensional feature vector).
  - Sends input to the Kotlin layer via Model Invoke MethodChannel.
  - Receives softmax probability predictions and identifies the index of the highest probability.

Data Flow Summary:

1. **Capture & Preprocessing:**
   - Frames are captured using the Flutter camera plugin.
   - Frames are debounced and converted to RGB, then encoded as JPEG.
2. **Landmark Detection:**
   - JPEG images are sent to the Kotlin native layer where MediaPipe HandLandmarker detects and returns 21 landmark points per hand.
3. **Normalization:**
   - Landmarks are normalized by the Dart layer using wrist-relative coordinates.
4. **Stack Buffering:**
   - Normalized landmarks are pushed into a stack queue to regulate data feeding into the model.

5. **Inference:**
    - Model input is constructed from the latest landmarks and sent via MethodChannel to Kotlin, where the TFLite model infers the predicted letter class.
6. **Letter Mapping & Display:**
    - The prediction index is mapped to a corresponding Arabic letter and displayed on screen.

Advantages of MVC Architecture:

- **Separation of Concerns**: UI logic (View) is decoupled from the data/model logic.
- **Scalability**: Each layer can be modified or extended independently.
- **Testability**: Each layer can be unit-tested in isolation.
- **Maintainability**: Developers can easily update UI or logic without full rewrites.

Summary:

This chapter presented the full lifecycle of the gesture recognition model, from input preprocessing to architecture design, training configuration, and deployment. The chapter explained how MediaPipe extracts 21 hand landmarks per frame, which are then normalized and flattened into a 42-dimensional input vector. This vector feeds into a dense neural network built with TensorFlow/Keras.

The model was trained using an extended training strategy with callbacks like EarlyStopping and ModelCheckpoint to ensure optimal generalization. Sparse categorical cross entropy was used for multiclass classification, and the training pipeline achieved efficient convergence with real-time prediction capability. The deployment process integrates the model with webcam input and provides live feedback through OpenCV, adhering to the MVC architecture and system usability goals outlined in the SRS.

Chapter 5: System and Evaluation Metrics

5.1 System Information and Environment:

The model was trained and tested on a local machine with the following hardware and software specifications:

- GPU: NVIDIA GeForce GTX 1060 G1 Gaming 6GB
- CPU: Intel Core i3-10100F (4 cores / 8 threads)
- RAM: 16GB DDR4 (2 × 8GB, 3200 MHz)
- Motherboard: ASUS PRIME H510M-A
- OS: Windows 11 (64-bit)
- Python Version: 3.10
- TensorFlow Version: 2.10.1 (GPU-enabled)
- CUDA Toolkit: 11.2
- cuDNN Version: 8.1
- OpenCV Version: 4.8.0

The hardware setup enabled efficient training of the lightweight neural network without relying on cloud infrastructure. The GPU acceleration significantly improved the training time and enabled real-time testing and tuning of the system.

5.2 Evaluation Metrics:

To evaluate the performance of the hand gesture classifier, multiple standard machine learning metrics were used.

1- Accuracy:

Accuracy measures the proportion of correctly predicted gestures out of all predictions:

$$\text{Accuracy} = \frac{Correct\ Predictions}{Total\ Predictions}$$

This is the primary metric used, especially during model training and validation.

2- Precision, Recall, and F1-Score:

These metrics provide deeper insights into the model's classification performance:

- **Precision**: Measures how many of the predicted gestures are actually correct.
- **Recall**: Measures how many actual gestures were correctly predicted.
- **F1-Score**: The Harmonic mean of precision and recall.

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

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

$$\text{F1 Score} = 2 \times \frac{Precision \times Recall}{Precision + Recall}$$

These were calculated using the classification_report from sklearn.metrics during model evaluation in the notebook.

3- Confusion Matrix:

The confusion matrix was used to visualize the performance of the model for each gesture class. It shows the number of times each class was correctly or incorrectly predicted, helping identify common misclassifications

The following figure describes a simple confusion matrix of 2 classes :

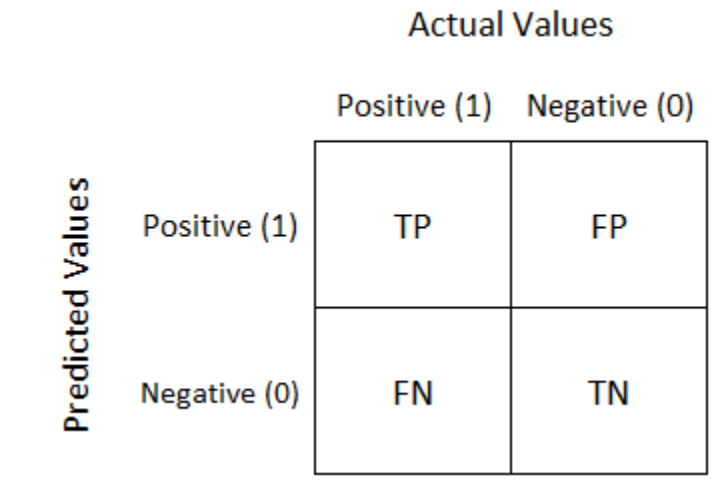


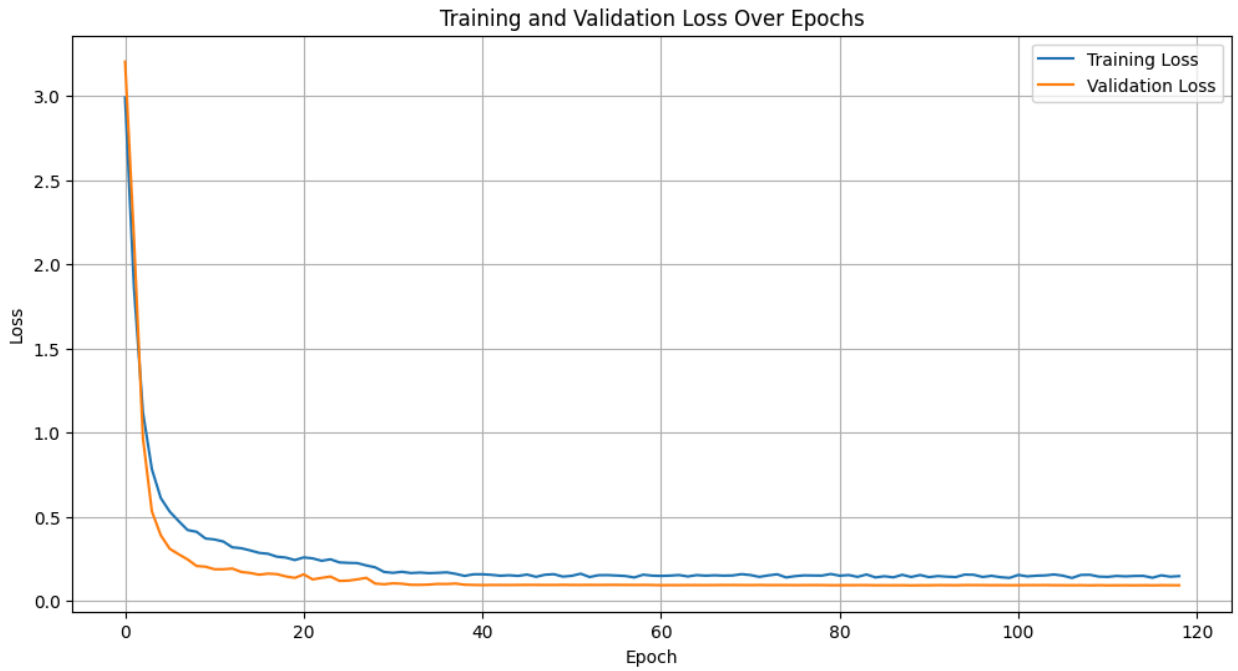
Figure 11: confusion matrix of 2 classes.

4- Loss Curve and Accuracy Curve:

During training, the model's **loss and accuracy** were plotted per epoch to monitor overfitting and convergence:

- A **decreasing loss** and **increasing accuracy** indicate proper learning.
- EarlyStopping was applied if validation accuracy plateaued or validation loss increased.

The following figure shows the proposed model loss & accuracy curves :



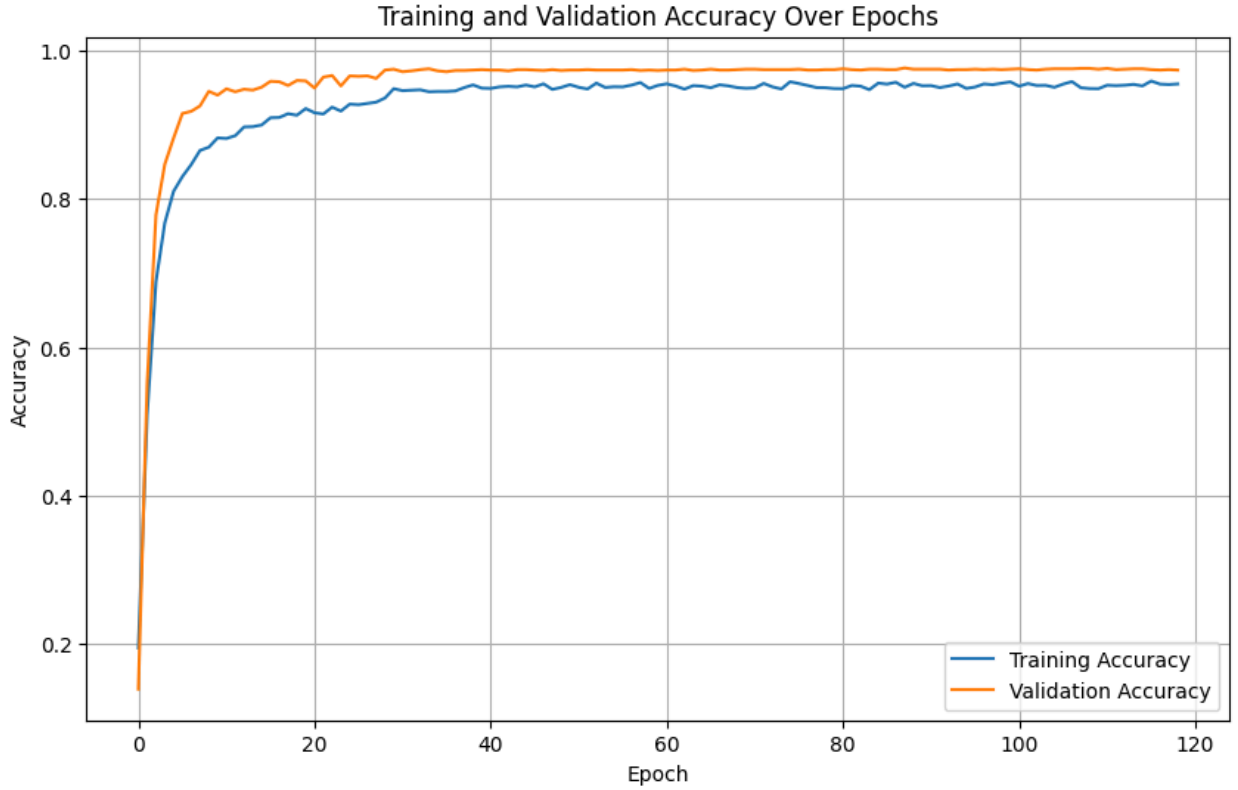


Figure 12: Loss & Accuracy Curves.

- **Loss Curve (Left)**: The training loss (blue line) and validation loss (orange line) both show a sharp decline during the early epochs, indicating that the model is learning quickly. Over time, both curves continue to decrease, with the validation loss plateauing at a lower value than the training loss. This suggests that the model generalizes well on unseen data and does not overfit significantly. The smooth and consistent decline without divergence is a positive sign.

- **Accuracy Curve (Right)**: The training accuracy (blue line) and validation accuracy (orange line) both increase rapidly in the early stages, reaching high values. The validation accuracy remains slightly higher than the training accuracy, stabilizing near 0.98, which is excellent. This gap between validation and training accuracy implies that the model is performing slightly better on the validation set—possibly due to regularization or dropout being applied only during training.

Overall, these curves indicate a well-trained model with strong generalization, minimal overfitting, and high predictive performance.

5- Model Performance :

To provide a clearer visualization of the classification performance across all gesture classes, the confusion matrix of the trained model will be included in this chapter.

The following figure displays how many times each gesture was correctly predicted and where the model made errors by confusing one class for another.

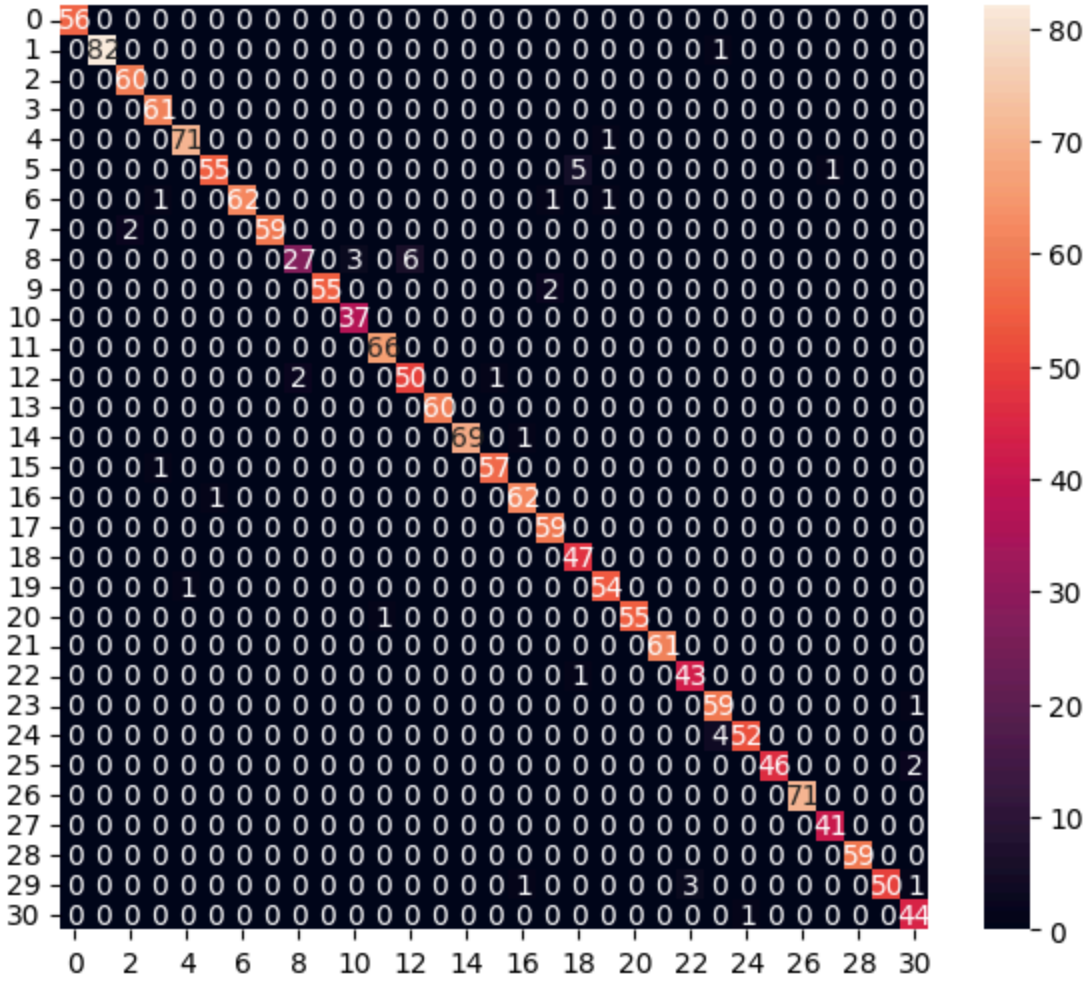


Figure 13: Confusion Matrix of the Proposed Model.

Diagonal Dominance: The matrix is mostly populated along the diagonal, which indicates that the model is making correct predictions most of the time. Each diagonal cell shows the number of correct predictions for that class.

Normalized or Annotated: The values inside the cells seem to be normalized (e.g., 0.56, 0.83), likely representing accuracy per class (true positives divided by the total actual samples for that class).

Sparsity: Off-diagonal values are mostly zero, meaning there are very few misclassifications.

Color Gradient: The heatmap's color intensity highlights the frequency of predictions — lighter colors indicate higher values. Most correct classifications are strong, with some classes like class 1, 2, and 10 having higher true positive rates.

This confusion matrix reflects a **well-performing multi-class classifier**. The model correctly classifies most instances for each of the 31 classes with minimal confusion between classes. However, a few classes (e.g., 8, 11, 18) show slightly lower correct classification rates, suggesting potential room for improvement or class imbalance.

Classification Report

	precision	recall	f1-score	support
0	1.00	1.00	1.00	56
1	1.00	0.99	0.99	83
2	0.97	1.00	0.98	60
3	0.97	1.00	0.98	61
4	0.99	0.99	0.99	72
5	0.98	0.90	0.94	61
6	1.00	0.95	0.98	65
7	1.00	0.97	0.98	61
8	0.93	0.75	0.83	36
9	1.00	0.96	0.98	57
10	0.93	1.00	0.96	37
11	0.99	1.00	0.99	66
12	0.89	0.94	0.92	53
13	1.00	1.00	1.00	60
14	1.00	0.99	0.99	70
15	0.98	0.98	0.98	58
16	0.97	0.98	0.98	63
17	0.95	1.00	0.98	59
18	0.89	1.00	0.94	47
19	0.96	0.98	0.97	55
20	1.00	0.98	0.99	56
21	1.00	1.00	1.00	61
22	0.93	0.98	0.96	44
23	0.92	0.98	0.95	60
24	0.98	0.93	0.95	56
25	1.00	0.96	0.98	48
26	1.00	1.00	1.00	71
27	0.98	1.00	0.99	41
28	1.00	1.00	1.00	59
29	1.00	0.91	0.95	55
30	0.92	0.98	0.95	45
accuracy			0.97	1776
macro avg	0.97	0.97	0.97	1776
weighted avg	0.98	0.97	0.97	1776

5.3 Real-Time System Testing:

The deployed system was tested in various real-time conditions:

- **Lighting Conditions**: Both bright and dim environments.
- **Background Clutter**: Simple vs. noisy backgrounds.
- **Hand Sizes and Skin Tones**: Tested across users with different hand features.

The system showed stable performance, though prediction confidence slightly decreased in cases with:

- Occluded hands.
- Fast movement.
- Poor lighting.

5.4 Challenges and Limitations:

- **Dataset Diversity**: Model accuracy depends heavily on the training dataset. More diverse and augmented data would improve performance.
- **One-Hand Limitation**: The current system supports only **one hand** at a time.
- **No Temporal Modeling**: The system classifies based on **static keypoints**, without considering motion over time.

Summary:

This chapter described the complete system architecture and the metrics used to evaluate its performance. The model achieved high accuracy in both validation and real-time tests, confirming its usability in recognizing Arabic Sign Language hand gestures. However, improvements can be made in dataset expansion, multi-hand support, and temporal gesture modeling for future work.

Chapter 6: Conclusion and Future work

6.1 Conclusions:

This work presents a comprehensive solution to the challenge of Arabic Sign Language (ArSL) interpretation through a real-time recognition system built upon modern AI and computer vision techniques. Over the course of this book, we explored the design, theoretical background, technical implementation, evaluation, and real-world application of a lightweight yet powerful ArSL recognition pipeline.

Summary of Key Contributions:

- **Real-time ArSL recognition** using standard webcam input and mobile deployment.
  42-feature landmark input enabling fast and accurate gesture classification.
- **Lightweight ANN architecture** supporting high accuracy with minimal resource use.
- **MVC-based cross-platform system** integrating Flutter, MediaPipe, and TensorFlow Lite.

6.2 Future Work:

While the current system demonstrates practical deployment for static ArSL gestures, there are opportunities to build upon this foundation:

Dynamic Gesture Recognition:

- Integrate temporal modeling (e.g., LSTM, Transformer) to capture sequential gestures for full-sentence translation.
- Expand the dataset to include regional ArSL dialects and dynamic signs (e.g., words, phrases).

Enhanced Robustness:

- Incorporate multi-hand detection to support complex signs requiring both hands.
- Improve performance under occlusions and varying lighting conditions using synthetic data augmentation.

Edge Deployment:

- Optimize the model for low-power edge devices (e.g., Raspberry Pi, smartphones) using quantization/pruning.
- Develop a cross-platform mobile app with offline capabilities for wider accessibility.
- User-Centric Evaluation.
- Conduct field testing with deaf communities to refine usability and address real-world challenges.
- Explore multi inputs (e.g., facial expressions, body posture) for richer context.

This work contributes toward inclusive AI technologies that bridge communication gaps. The proposed system stands as a scalable foundation for intelligent gesture recognition, with wide potential in education, healthcare, translation services, and human-computer interaction.

Future iterations can extend this work into full sign language translation pipelines, making ArSL communication more accessible across the Arab world.

References

1. World Health Organization. (2021). *World report on hearing*. https://www.who.int/publications/i/item/9789240020481
2. Abdel-Hamid, O., Khatib, O. M., Aly, A., Morad, M., & Kamel, S. (2007). Prevalence and patterns of hearing impairment in Egypt: A national household survey. *Eastern Mediterranean Health Journal*, *13*(5), 1170–1180. https://doi.org/10.26719/2007.13.5.1170
3. Dabwan, B. A., & Jadhav, M. E. (2021). A deep learning based recognition system for Yemeni Sign Language. In *Proceedings of the 2021 International Conference of Modern Trends in Information and Communication Technology Industry (MTICTI)* (pp. 1–5). IEEE. https://doi.org/10.1109/MTICTI53925.2021.9664779
4. Balat, M., Awaad, R., Adel, H., Zaky, A. B., & Aly, S. A. (2024). Arabic Sign Language recognition using CNNs and Transformers: A comparative study. *arXiv preprint*. https://arxiv.org/abs/2410.00681
5. Dabwan, B. A., & Jadhav, M. E. (2022). Hybrid deep learning models for Arabic Sign Language recognition. In *Proceedings of the 2022 International Conference on Advances in Computer Vision, Artificial Intelligence and Technologies (ACVAIT)*. https://doi.org/10.2991/978-94-6463-196-8_35
6. AlKhuraym, B. Y., Ben Ismail, M. M., & Bchir, O. (2022). Arabic Sign Language recognition using lightweight CNN-based architecture. *International Journal of Advanced Computer Science and Applications (IJACSA)*, *13*(4), 273–279. https://doi.org/10.14569/IJACSA.2022.0130438
7. Bazarevsky, V., Zhang, F., Vakonov, A., Tkachenka, A., Sung, G., Chang, C.-L., & Grundmann, M. (2019). On-device, real-time hand tracking with MediaPipe. *Google AI Blog*. https://research.google/blog/on-device-real-time-hand-tracking-with-mediapipe/
8. Google Developers. (2024). *MediaPipe Hand Landmarker – Vision Solutions*. https://ai.google.dev/edge/mediapipe/solutions/vision/hand_landmarker
9. Haykin, S. (2009). *Neural networks and learning machines* (3rd ed.). Pearson Education. https://books.google.com.eg/books?id=K-ipDwAAQBA