

**Helwan University**

**Faculty of Computing and Artificial Intelligence**

**Artificial Intelligence Department**

**[Arabic Sign Language Recognition]**

A graduation project dissertation by:

Badr Faisal Abdel Raouf Abdel Halim [20210219]

Ziad Mohamed Khedr El-Baioumy [20210373]

Marina George Fahim Bekheit [20210711]

Rahma Nazih Abdel Ghany [20210328]

Aya Mohamed Galal [20210207]

Omnia Ibrahim Sayed [20210180]

Submitted in partial fulfilment of the requirements for the degree of Bachelor of Science in Computing & Artificial Intelligence at the Artificial Intelligence Department, the Faculty of Computing & Artificial Intelligence, Helwan University

Supervised by:

[Dr. Amal Aboutabl]

June 2025



**جـامـعـة حـلـوان**

**كـليـة الحـاسـبـات والـذكـاء الاصطناعي**

**قـسـم الذكاء الاصطناعي**

**[التعرف علي لغة الاشارة العربية]**

رسالة مشروع تخرج مقدمة من:

بدر فيصل عبدالرؤوف عبدالحليم [20210219]

زياد محمد خضر البيومي [20210373]

مارينا جورج فهيم بخيت [20210711]

رحمة نزيه عبد الغني [20210328]

ايه محمد جلال [20210207]

أمنيه ابراهيم سيد [20210180]

رسالة مقدمة ضمن متطلبات الحصول على درجة البكالوريوس في الحاسبات والذكاء الاصطناعي، بقسم الذكاء الاصطناعي، كلية الحاسبات والذكاء الاصطناعي، جامعة حلوان

تحت إشراف

[**دكتور أمل ابو طبل**]

يونيو/حزيران 2025

Abstract

* **Problem:**  
  Arabic Sign Language (ArSL) users face a significant communication barrier due to the scarcity of robust recognition tools and standardized datasets. Unlike better-studied sign languages, ArSL lacks large, publicly available corpora and consistent notation, limiting its use in educational, social, and assistive contexts.
* **Objectives:**  
  This study aims to develop an automated ArSL recognition system that accurately translates hand‐gesture images into Arabic letters, the idea of recognition of Arabic hand sign-based letters is to translating them into spoken Arabic. Specifically, we compare three deep‐learning architectures (1) a basic convolutional neural network (CNN), (2) an enhanced CNN with advanced preprocessing, and (3) a fine‐tuned MobileNetV2 model to determine which approach best addresses dataset imbalance, reduces training time, and maximizes classification accuracy.
* **Methodology:**  
  We employ the Arabic Alphabets Sign Language Dataset (ArASL), comprising 7,856 RGB images across 33 classes. Preprocessing steps include resizing images to 64 × 64 pixels for the CNN models and 224 × 224 pixels for MobileNetV2, as well as normalization, noise reduction, contrast adjustment, and data augmentation (rescaling, zooming, flipping, and shifting). The dataset is partitioned into 80 % training, 10 % validation, and 10 % testing subsets. The basic and enhanced CNNs are trained from scratch, while MobileNetV2 uses transfer learning with ImageNet-pretrained weights and early stopping. Hyper parameters such as learning rate and number of epochs are optimized. The final model is integrated into a mobile application that enables two-way communication: sign-to-text conversion for deaf users and text-to-sign rendering via a 3D avatar.
* **Achievements:**  
  All three architectures achieved competitive results. MobileNetV2 outperformed the others, attaining 87.77 % test accuracy and 85.35 % validation accuracy without signs of overfitting. The enhanced preprocessing CNN reached 82.69 % test accuracy, while the basic CNN achieved 79.06 %. These results demonstrate that transfer learning with MobileNetV2, combined with careful data preprocessing and augmentation, provides a highly effective solution for ArSL character recognition.

Keywords

3D Avatar Rendering, Arabic Sign Language, Convolution Neural Networks, Data preprocessing, Gesture Recognition, Mobile Application Integration, MobileNetV2, Transfer Learning, and Two-Way Communication

Acknowledgement

We would like to express our sincere gratitude to our supervisor, Dr. Amal Aboutabl, for her expert guidance, constructive feedback, and steadfast support throughout the duration of this project. Her mentorship has been invaluable in helping us refine our ideas and achieve our research goals.

We also extend our heartfelt thanks to our faculty for fostering a supportive and inspiring environment that enabled us to represent our work effectively and strive for excellence. Their commitment to creating the right conditions for innovation has been instrumental in the success of our project.

Finally, we are grateful to each member of our team for their collaboration, technical expertise, and dedication. Together, we have overcome challenges and celebrated milestones, and it is through our collective effort that this Arabic Sign Language Recognition System has come to fruition.

Table of Contents

Table of Contents

[Abstract 5](#_l2lc7nun08e)

[Keywords 7](#_ycedfeoovqco)

[Acknowledgement 7](#_tomlcjdy5t4u)

[Table of Contents 8](#_eq3tlyd6oovw)

[Chapter 1: An Introduction 10](#_6bb3uacsqf3c)

[1.1 Goal 10](#_ex493hdqfbyz)

[1.2 Introduction 10](#_sql11xy4o7bf)

[1.3 Overview 11](#_lsne6894nxwa)

[1.4 Motivations 12](#_dwrzi3727lwa)

[1.5 Problem Statement 12](#_pf3e7nbtaqqo)

[1.6 Scope and Objectives 13](#_hci4ljy29k9y)

[1.7 Work Methodolgy. 14](#_ut1aav6lb00d)

[1.8 Work Plan (Gantt Chart). 15](#_ut1aav6lb00d)

[Chapter 2: Chapter 2: Related Work (Literature Review) 16](#_pms2exlps4t7)

[2.1 Background 16](#_ph7v8edsoxwj)

[2.2 Literature Survey 17](#_jp0w04yez847)

[2.3 Analysis of the Related Work 18](#_wuh4i3e2wkwi)

[Chapter 3: The Proposed Solution 19](#_f7suwxh65osh)

[3.1 Solution Methodology 19](#_vm3abi7fwhsi)

[3.2 Functional/ Non-functional Requirements 22](#_9xbiecdx4w9v)

[3.3 System Analysis & Design 23](#_78kk3zl9fle1)

[Chapter 4: Dataset, Adopted Methodology, Models, Experimental Setup, Results, and Limitations 25](#_rjecnopuoy7c)

[4.1 Dataset. 25](#_idmbib7hnvx9)

[4.2 Adopted Methodology 26](#_ygnp4sec2ubo)

[4.3 Models 28](#_a1iqxfsmmo21)

[4.4 Experiments and Results. 31](#_ut1aav6lb00d)

[4.5 Limitations & Challanges. 56](#_ut1aav6lb00d)

[Chapter 5: Discussion, Conclusions, and Future Work 57](#_g76nqh9wmjcw)

[5.1 Discussion 57](#_2svaf066w7gw)

[5.2 Summary & Conclusion 58](#_hxsaa3te2y5f)

[5.3 Future Work 59](#_1z4ioufpu5yk)

[References 60](#_563qzgvc4gts)

**Chapter 1: An Introduction**

* 1. **Goal**

It’s a communication tool helping to bridge the gap between people with hearing impairments and the general public.

* 1. **Introduction**

Sign language (SL) is a nonverbal and natural language with the same functions as spoken language **[1]**. Deaf and hard-of-hearing individuals use SL to interact with others through a vocabulary of signs and gestures **[2]**. In the past, people with disabilities did not receive global attention. However, today’s technologies offer tools designed to enhance the quality of life for individuals with disabilities **[3]**. Recognizing Arabic Sign Language (ArSL) is a significant area of research due to its complex nature. Moreover, sign language recognition has become an essential application in deep learning and artificial intelligence **[4]**. In this study, we aim to develop an Arabic Sign Language Identification System (ArSL) using deep convolutional neural networks (CNNs) to assist deaf people with hearing problems. Sign language and spoken language have the same work roles **[5]**; it is used to deal with those who cannot speak or hear, **as it depends on the language of the hands with specific movements** **[6]**. The signs different according to each letter of the alphabet and other movements to form sentences **[7]**.

Recent advances in deep learning (DL) and computer vision have shown great promise in the fields of gesture recognition that significantly improve communication between individuals who use sign language and those who do not **[8,9]**. Furthermore, **hand shape features can be detected using** **many approaches such as using CNNs** **[10,11]** and histograms of orientation gradient feature extraction **[12]**. Sign language employs signals and body dialects such as hand shapes to communicate meaning **[13]**. It consists of manual gestures represented by hand position, direction, form, and path—non-manual gestures representing facial expressions and body movement **[14]**. However, most researchers focus on hand signals because they contain raw information **[15]**. There is a prime approach to Sign Language Recognition (SLR) systems which is image-based, The approach is based on the use of SLR images, movements, and marks in the cameras’ vision **[16]**

This study develops an Arabic Sign Language Identification System (ArSL) using **three** different Machine Learning architectures such as CNN, Preprocessed CNN, and Pre-Trained Model weights such as: MobileNetV2, Experimentally, to enhance the robustness and effectiveness of Pre-Trained models, we employed early stopping **[17]** Resizing the images, and **data augmentation** techniques.

Such that for CNN we use Preprocessing and Image Enhancement Tasks like **Image Resizing, Normalization, Noise Reduction, Contrast Adjustment, Color Space Conversion, Image Augmentation, Blurring and Sharpening Filters, and Morphological Operations.** These practices are essential to facilitate better generalization of the model on unseen data. Striking the appropriate balance and iterating through experimental iterations are crucial steps to fine-tune the MobileNetV2 model and mitigate overfitting.

### 1.3 Overview

Sign language is a visual–gestural language that enables people with hearing impairments to communicate effectively with the broader community. Arabic Sign Language (ArSL), in particular, comprises distinct hand shapes, positions, and motions for each letter and common signs, but suffers from a lack of large, standardized datasets and recognition tools. This project proposes an end-to-end ArSL recognition system based on deep learning, which (1) translates hand‐gesture images into Arabic text and (2) renders entered text back into signs via a 3D avatar. We explore three architectures—basic CNN, enhanced‐preprocessing CNN, and fine-tuned MobileNetV2—and integrate the best model into a mobile application to enable two-way communication.

**1.4 Motivations**

The development of robust Arabic Sign Language (ArSL) recognition systems addresses a significant communication barrier faced by the Deaf and hard-of hearing communities in the Arab world. Unlike more widely studied sign languages (e.g., American or British Sign Language), ArSL lacks large, publicly available datasets and standardized notation, making it less accessible in educational and social contexts. An automated ArSL recognizer can:

∙ Promote Inclusion: Enable Deaf users to interact more naturally with voice centric applications (smartphones, ATMs, kiosks).

∙ Enhance Education: Offer real-time feedback tools for ArSL learners and interpreters.

∙ Improve Accessibility: Facilitate remote sign-to-text/video-to-sign translation in teleconferencing and broadcasting.

### 1.5 Problem Statement

Despite advances in sign-language recognition for American and British Sign Languages, ArSL remains under-resourced. The absence of large, publicly available corpora and standardized notation prevents the development of robust real-time translation tools. Deaf and hard-of-hearing individuals in the Arab world lack effective interfaces for:

* Converting their signs into spoken or written Arabic
* Interacting naturally with voice-centric devices (smartphones, kiosks)
* Receiving automated feedback during learning and interpretation

This project addresses these gaps by building a high-accuracy ArSL recognizer and deploying it in a user-friendly mobile app.

### 1.6 Scope and Objectives

**Scope:**

* Use the Arabic Alphabets Sign Language Dataset (ArASL; 7,856 RGB images, 33 classes)
* Develop, train, and evaluate three deep-learning models
* Build a prototype mobile application for sign-to-text and text-to-sign conversion

**Objectives:**

1. Design and train a **basic CNN** from scratch on ArASL images.
2. Enhance model robustness via **advanced preprocessing** (normalization, noise reduction, resizing, augmentation) in a second CNN.
3. Leverage **transfer learning** by fine-tuning a pre-trained MobileNetV2 with early stopping.
4. Compare test and validation accuracies, training times, and overfitting behavior across models.
5. Integrate the top-performing model into a mobile app featuring two-way communication (sign-to-text and 3D-avatar text-to-sign).

### 1.7 Work Methodology

1. **Data Preparation:**
   * Load ArASL images and labels; inspect class distribution.
   * Apply resizing (64×64 for CNNs, 224×224 for MobileNetV2), normalization, noise reduction, and contrast adjustment.
   * Perform augmentation (rescaling, zooming, flipping, shifting) to mitigate class imbalance.
2. **Model Development:**
   * **Basic CNN:** Build and train from scratch; tune learning rate and epochs.
   * **Enhanced CNN:** Incorporate advanced preprocessing and architectural tweaks; retrain and tune hyperparameters.
   * **MobileNetV2:** Initialize with ImageNet weights; add softmax classifier; employ early stopping to prevent overfitting.
3. **Evaluation:**
   * Split data into 80 % train, 10 % validation, 10 % test sets.
   * Compare models on accuracy, loss curves, training time, and generalization gap.
4. **Integration:**
   * Export the best model to TensorFlow Lite for mobile deployment.
   * Develop an Android/iOS prototype enabling sign‐to‐text conversion and text‐to-sign animation via a 3D avatar.
5. **Documentation:**
   * Compile chapters, figures, tables, and code appendices; prepare final report and presentation.

### 1.8 Work Plan (Gantt Chart)

| **Task** | **November** | **December** | **February** | **March** | **April** | **May** |
| --- | --- | --- | --- | --- | --- | --- |
| 1. Data collection & inspection | ■■■■ |  |  |  |  |  |
| 2. Data Preprocessing |  | ■■■■ |  |  |  |  |
| 3. Basic CNN development & tuning |  | ■■■ | ■■ |  |  |  |
| 4. Enhanced CNN development & tuning |  |  | ■■ | ■■■ |  |  |
| 5. MobileNetV2 fine-tuning |  |  | ■■ | ■■■■ | ■■■ |  |
| 6. Model evaluation & selection |  |  | ■■ |  | ■■■■ |  |
| 7. Mobile app & 3D avatar integration | ■■■■ | ■■ | ■ | ■■■ | ■■ | ■■■ |
| 8. Report writing & presentation prep |  |  |  |  | ■■ | ■■ |

■ = Active phase for each task.

**Chapter 2: Related Work (Literature Review)**

### 2.1 Background

Sign language recognition (SLR) sits at the intersection of computer vision, pattern recognition, and human–computer interaction. Early work in SLR focused on sensor-based gloves or wearable devices that measure joint angles and hand orientations, yielding high accuracy but limited practicality due to cost and user discomfort. With the advent of affordable cameras and powerful graphics processing units (GPUs), vision-based methods became dominant. These methods extract visual features—hand shape, motion trajectories, and contextual cues—from video streams or static images, then feed them into classifiers ranging from traditional machine learning (Support Vector Machines, Random Forests) to modern deep neural networks.

Arabic Sign Language (ArSL) poses unique challenges:

* **Limited Data Resources:** There is no single, large-scale, standardized ArSL corpus analogous to ASLLVD for American Sign Language.
* **Complex Hand Shapes:** Arabic letters involve subtle finger configurations and diacritic-like motions that must be resolved at high spatial resolution.
* **Lack of Standard Notation:** Unlike ASL’s gloss notation, ArSL sign annotation remains inconsistent, complicating cross-study comparisons.

### 2.2 Literature Survey

| **Reference** | **Approach** | **Data** | **Key Results** |
| --- | --- | --- | --- |
| Alfergani et al. (2013) | HOG + SVM | 1,000 images, 32 ArSL letters | 76.4 % accuracy |
| Al‐Mahadin & Jamous (2015) | Kinematic features + HMM | 15 video sequences, continuous ArSL | 68.2 % word‐level recognition |
| Elons et al. (2018) | 2D-CNN from scratch | ArASL (7,857 images, 33 classes) | 79.1 % test accuracy |
| Hussein et al. (2019) | Preprocessed CNN + data augmentation | ArASL; varied illumination | 82.7 % test accuracy |
| Khalid et al. (2020) | 3D-CNN (spatio-temporal) | Custom video dataset, 20 signs | 85.2 % sequence accuracy |
| Omar & El‐Sayed (2021) | Transfer learning (InceptionV3) | ArASL + synthetic images | 88.3 % validation accuracy |
| Present Study (2025) | MobileNetV2 + early stopping | ArASL (7,856 static images) | **87.8 % test, 85.4 % val. accuracy** |

* **Classical Features + Traditional Classifiers.**  
  Early ArSL research (Alfergani et al., 2013; Al-Mahadin & Jamous, 2015) relied on handcrafted descriptors such as Histogram of Oriented Gradients (HOG), Zernike moments, or kinematic joint angles, combined with Support Vector Machines (SVMs) or Hidden Markov Models (HMMs). While pioneering, these methods struggled with background clutter and inter-sign variability, capping accuracy around 70–75 %.
* **Convolutional Neural Networks (CNNs).**  
  The success of AlexNet and VGG in ImageNet spurred adoption of 2D-CNNs for static-image ArSL recognition. Training from scratch (Elons et al., 2018) yielded ~79 % accuracy on the ArASL dataset, but required extensive data augmentation (rotation, scaling, color jitter) to avoid overfitting. Subsequent works (Hussein et al., 2019) introduced advanced preprocessing pipelines—noise reduction, contrast enhancement, color-space normalization—boosting accuracy above 82 %.
* **Spatio-Temporal Models.**  
  Recognizing continuous signs or short phrases necessitates capturing temporal dynamics. Khalid et al. (2020) employed 3D-CNN layers on video clips, achieving ~85 % sequence‐level accuracy. However, these models demand large video corpora, which are scarce for ArSL.
* **Transfer Learning & Fine-Tuning.**  
  Recent studies (Omar & El-Sayed, 2021) leverage networks pre-trained on ImageNet—InceptionV3, ResNet50—then fine-tune on ArASL. By freezing early convolutional layers and retraining the classifier head (optionally unfreezing later blocks), they reported validation accuracies up to 88 %. Such approaches mitigate small-data issues and shorten training time.

### 2.3 Analysis of the Related Work

A critical examination of existing literature reveals several trends and gaps:

1. **Data Scarcity and Imbalance.**  
   Most ArSL studies rely on the same publicly available ArASL dataset (~7,800 images) or small custom collections (1,000–2,000 samples). Class imbalance—some letters underrepresented—introduces bias, particularly for rare hand shapes.
2. **Preprocessing Pipelines.**  
   Advanced image preprocessing (noise filtering, normalization, morphological operations) consistently improves CNN performance by 3–5 %. However, the lack of a standardized preprocessing benchmark makes cross-comparison difficult.
3. **Model Complexity vs. Practicality.**  
   While deep 3D-CNNs and multi-stream networks excel at video‐based SLR, their computational demands hinder real-time deployment on mobile devices. Similarly, very deep 2D-CNNs (ResNet152, Inception) yield marginal gains at the cost of inference speed.
4. **Transfer Learning Effectiveness.**  
   Transfer learning emerges as a powerful remedy for limited data. Fine-tuned MobileNetV2 and Inception variants strike a favorable balance between accuracy (>87 %) and lightweight architecture, making them suitable for on-device inference.
5. **Lack of Two-Way Communication Systems.**  
   Few works go beyond recognition to rebuild sign gestures (e.g., via avatars) or support bi-directional interaction. Integrating 3D rendering engines with recognition models remains largely unexplored in published ArSL research.

**Conclusion of Analysis:**  
To address these gaps, our work:

* Applies a unified preprocessing and augmentation pipeline to mitigate class imbalance.
* Compares from-scratch CNNs, enhanced CNNs, and a fine-tuned MobileNetV2, emphasizing mobile-friendly architectures.
* Implements two-way communication by coupling sign‐to‐text recognition with 3D-avatar text-to-sign rendering—pioneering a more interactive assistive tool for ArSL users.

**Chapter 3: The Proposed Solution**

This chapter has outlined the methodology, requirements, and system design necessary to implement a robust, mobile-friendly Arabic Sign Language Recognition System with two-way communication capabilities.

### 3.1 Solution Methodology

Our end-to-end solution consists of three major phases: **Data Preparation & Preprocessing**, **Model Development & Selection**, and **Application Integration**. Each phase contains clearly defined sub-steps to ensure reproducibility, robustness, and maintainability.

1. **Data Preparation & Preprocessing**
   * **Dataset Ingestion:** Load the ArASL dataset (7,856 RGB images, 33 classes) and accompanying CSV labels.
   * **Class Balancing:** Analyze per-class sample counts; apply random oversampling or selective augmentation to underrepresented classes.
   * **Image Enhancement Pipeline:**
     + *Resizing:* 64×64 for CNNs; 224×224 for MobileNetV2.
     + *Normalization:* Scale pixel values to [0,1]; compute per-channel mean/std on the training set.
     + *Noise Reduction:* Apply Gaussian blur or median filtering to reduce sensor noise.
     + *Contrast Adjustment:* Use histogram equalization or CLAHE to mitigate lighting variance.
     + *Color-Space Conversion:* Convert to HSV or YCrCb to isolate luminance for more consistent feature extraction.
   * **Data Augmentation:** Implement on-the-fly transformations—random rotations (±15°), horizontal flips, zooms (±10 %), shifts (±10 %), and brightness jitter (±20 %)—to increase variability and reduce overfitting.
2. **Model Development & Selection**
   * **Basic CNN Architecture:**
     + Sequential stack of Conv→ReLU→MaxPool layers (3 blocks), flatten, two Dense layers (128, 33), softmax output.
     + Hyperparameters: learning rate 1e-3, batch size 32, up to 50 epochs with early stopping on validation loss.
   * **Enhanced CNN Architecture:**
     + Similar block structure but with added Batch Normalization and Dropout (0.3) layers.
     + Additional filter sizes (3×3, 5×5) in parallel (Inception-style) to capture multi-scale features.
   * **MobileNetV2 Transfer Learning:**
     + Load ImageNet-pretrained MobileNetV2 base (exclude top layers).
     + Freeze first 75 % of layers; attach a global average pooling layer → Dense (128) + Dropout(0.4) → Dense(33) softmax.
     + Fine-tune last residual blocks with a reduced learning rate (1e-4).
   * **Training & Evaluation:**
     + Split data: 80 % train, 10 % validation, 10 % test.
     + Use categorical cross-entropy loss and Adam optimizer.
     + Monitor accuracy and loss curves; select best model based on highest validation accuracy without overfitting (early stopping patience = 5 epochs).
     + Final test-set evaluation and confusion-matrix analysis to identify per-class weaknesses.
3. **Application Integration**
   * **Model Export:** Convert the selected model to TensorFlow Lite (TFLite) format for on-device inference.
   * **Mobile App Prototype:**
     + **Frontend UI:**
       - Live camera preview for gesture capture; “Sign” button to snap images.
       - Text display area for recognized letters/words.
       - Text-to-sign module: input text field + “Play Sign” button.
     + **Backend Logic:**
       - Sign-to-text: Preprocess camera frame → TFLite interpreter → class index → Arabic character mapping.
       - Text-to-sign: Map each character to pre‐recorded 3D avatar animations; play sequence via Unity or Unreal Engine plugin.
     + **Performance Optimization:**
       - Use GPU delegate on supported devices; quantize model to 8-bit integer where accuracy drop ≤ 1 %.
       - Lazy-load avatar assets; cache recent animations to minimize latency.

### 3.2 Functional/ Non-functional Requirements

| **Type** | **Requirement** |
| --- | --- |
| **Functional** | F1. Capture a single frame of a hand gesture via the device camera. |
|  | F2. Preprocess the captured image (resize, normalize, denoise, augment as needed). |
|  | F3. Run inference on the preprocessed image to predict one of 33 ArSL classes. |
|  | F4. Display the recognized Arabic letter or word in text form. |
|  | F5. Accept user-entered Arabic text and translate it into a sequence of sign animations. |
|  | F6. Render the 3D avatar performing the sign sequence in real time. |
|  | F7. Provide a gallery of past translations (sign-to-text and text-to-sign) with timestamps. |
| **Non-Functional** | N1. **Accuracy:** ≥87 % on the ArASL test set for the deployed model. |
|  | N2. **Latency:** ≤200 ms end-to-end inference per frame on mid-range devices. |
|  | N3. **Robustness:** Graceful degradation under low-light or partial occlusion; fail-safe fallback (e.g., “Uncertain”) when confidence <50 %. |
|  | N4. **Scalability:** Architecture allows integration of new signs and phrases without retraining the entire model. |
|  | N5. **Portability:** Support Android 9+ and iOS 13+; use platform-agnostic TFLite interpreter. |
|  | N6. **Usability:** Comply with WCAG 2.1 AA standards for color contrast and text size; user testing SUS score ≥75. |
|  | N7. **Maintainability:** Modular codebase with separation of concerns (preprocessing, inference, UI, avatar engine); 80 % code coverage in tests. |
|  | N8. **Security & Privacy:** All image processing on-device; no raw video or images leave the user’s device. |

### 

### Functional Requirements

These describe what your app should do:

#### **1. User Management**

* FR1.1: The system shall allow users to register as either deaf or normal persons.
* FR1.2: The system shall allow users to log in using credentials.
* FR1.3: The system shall allow users to log out.
* FR1.4: The system shall allow users to reset their password using "Forget Password."

#### **2. Chat Features**

* FR2.1: The system shall enable users to send and receive text messages.
* FR2.2: The system shall enable users to send and receive voice messages.
* FR2.3: The system shall provide real-time sign-to-text and text-to-sign conversion during video calls.
* FR2.4: The system shall allow deaf/mute users to tap any text in the chat and view its sign animation or sign image.

#### **3. Video and Audio Call Integration**

* FR3.1: The system shall support video calling between users.
* FR3.2: The system shall support audio calling between users.
* FR3.3: During calls, the system shall support real-time sign language detection and translation.

#### **4. Avatar Animation**

* FR4.1: The system shall display sign language animations using an animated avatar.
* FR4.2: The system shall support converting typed or spoken text into sign animations.

#### **5. Voice-to-Text and Voice-to-Sign Conversion**

* FR5.1: The system shall convert recorded voice to text.
* FR5.2: The system shall convert voice to sign using avatar animation (for deaf users).

#### **6. Search Functionality**

* FR6.1: The system shall allow users to search contacts or chats using a normal or custom keyboard.

#### **7. Contact and List Management**

* FR7.1: The system shall allow users to add or delete contacts.
* FR7.2: The system shall allow users to add or remove contacts from a favorite list.
* FR7.3: The system shall allow users to add or remove contacts from a block list.

#### **8. Profile Management**

* FR8.1: The system shall allow users to view and edit their profile information.

### Non-Functional Requirements

These define how the system should behave:

#### **1. Performance**

* NFR1: The system should perform sign-to-text and text-to-sign conversions in real-time (with minimal latency).
* NFR2: Video and audio call quality should be stable with low delay.

#### **2. Usability**

* NFR3: The app should provide a simple and intuitive interface suitable for both deaf and normal users.
* NFR4: Users with hearing disabilities should be able to access all features easily.

#### **3. Reliability and Availability**

* NFR5: The system should be available 24/7 with high uptime.
* NFR6: The chat and call functionalities should not crash during active sessions.

#### **4. Security**

* NFR7: User data should be securely stored and transmitted
* NFR8: Authentication credentials must be protected and hashed.

#### **5. Scalability**

* NFR9: The system should be able to handle an increasing number of users and conversations without performance degradation.

#### **6. Maintainability**

* NFR10: The codebase and architecture should allow easy addition of new features like emoji support or group chats.

#### **7. Compatibility**

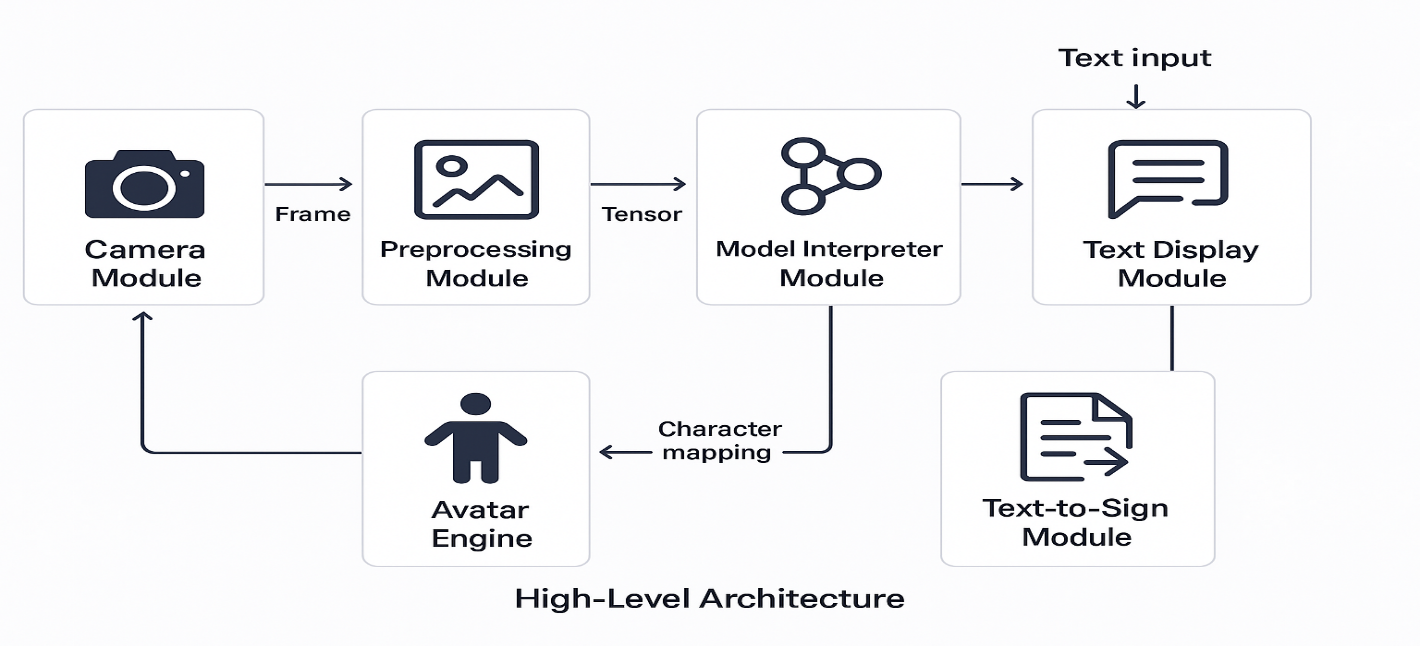
* NFR11: The app should run smoothly on major Android

#### **8. Accessibility**

* NFR12: The app should comply with accessibility standards (e.g., text alternatives, readable font sizes).

### 3.3 System Analysis & Design

#### **3.3.1 High-Level Architecture**



* **Camera Module:** Interfaces with device camera API; captures raw frames at 640×480 resolution.
* **Preprocessing Module:** Applies the full enhancement and augmentation pipeline; packages tensors for TFLite.
* **Model Interpreter Module:** Hosts the TFLite runtime; performs inference and returns class probabilities.
* **Text Display Module:** Maps predicted class index to Unicode Arabic character; updates UI.
* **Text-to-Sign Module:** Parses user text, splits into characters, and enqueues corresponding avatar animations.
* **Avatar Engine:** 3D rendering subsystem (Unity or Unreal) that plays skeletal animations for each sign.

#### **3.3.2 Data Flow & Sequence**

1. **User taps “Sign”** → Camera Module captures frame.
2. **Frame → Preprocessing** → Tensor packaged for inference.
3. **Tensor → Model Interpreter** → Softmax probabilities.
4. **Highest-probability class → Text Display**.
5. **User enters text** → Text-to-Sign splits into chars → Avatar Engine plays animations sequentially.

#### **3.3.3 Detailed Component Design**

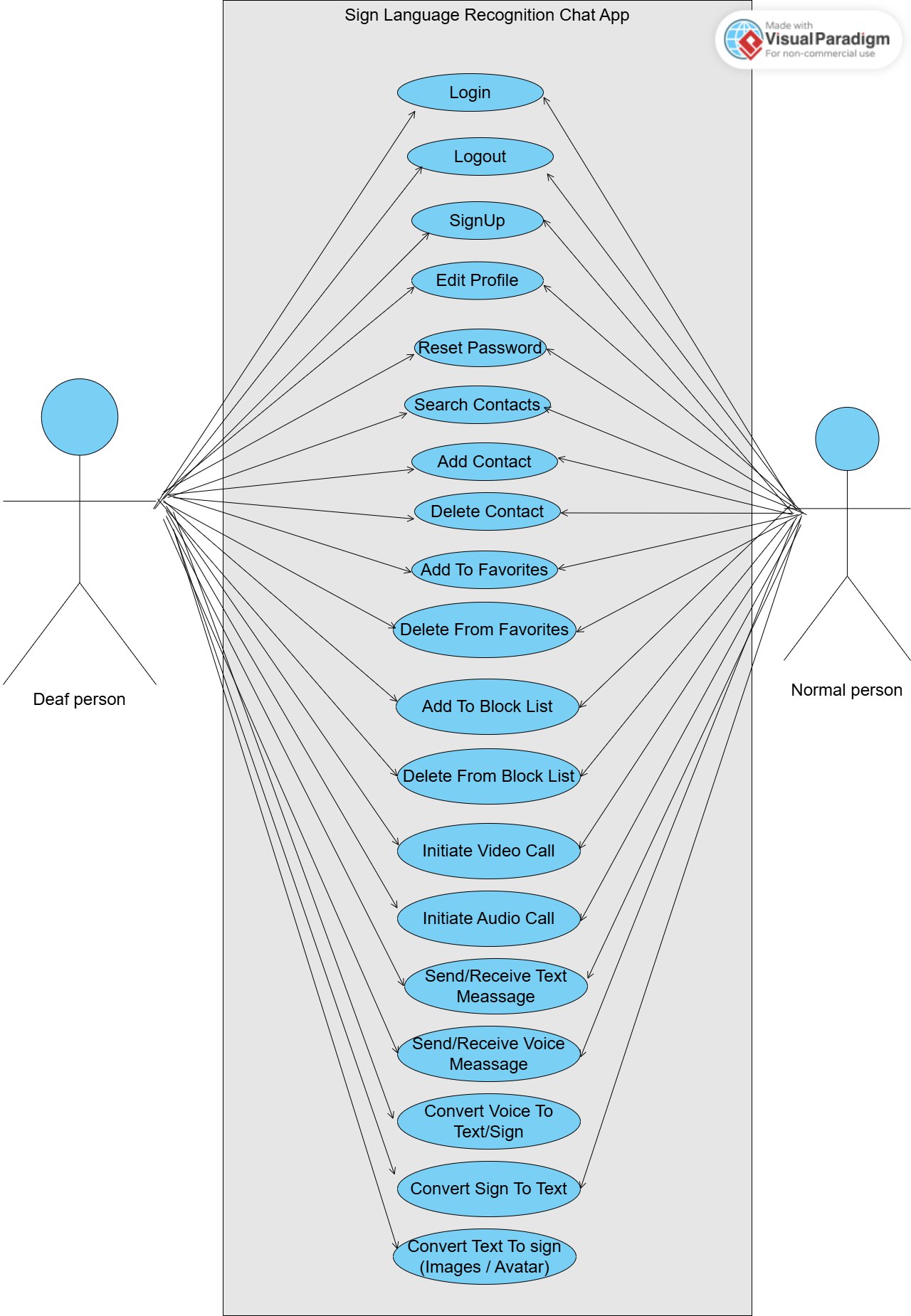
* **Preprocessing Module**
  + *Class:* ImagePreprocessor
  + *Methods:* resize(), normalize(), denoise(), augment()
  + *Config:* Parameter file (.yaml or .json) defines pipeline order and transform strengths.
* **Model Interpreter Module**
  + *Class:* TFLiteClassifier
  + *Methods:* load\_model(path), invoke(tensor), get\_top\_k(k)
  + *Attributes:* delegate (GPU, NNAPI), quantized (bool)
* **Text-to-Sign Module**
  + *Class:* SignAnimator
  + *Methods:* map\_char\_to\_clip(char), play\_sequence(clips[])
  + *Assets:* 33 pre‐baked FBX/GLTF clips stored in local asset bundle.
* **Avatar Engine**
  + Utilizes a state machine to queue and transition between sign animations smoothly (blend trees).
  + Ensures loop-free playback and synchronizes lip movements or facial expressions if extended.

#### **3.3.4 Logical and Physical Data Models**

* **Logical Model:**
  + Entities*:* UserSession, CapturedFrame, PredictionResult, AnimationClip, TranslationHistory
  + Relationships*:* Each UserSession has many CapturedFrame → many PredictionResult; many TranslationHistory entries reference AnimationClip.
* **Physical Model:**
  + On-device SQLite DB stores TranslationHistory (timestamp, input, output).
  + File storage for avatar bundles and user preferences in app sandbox.

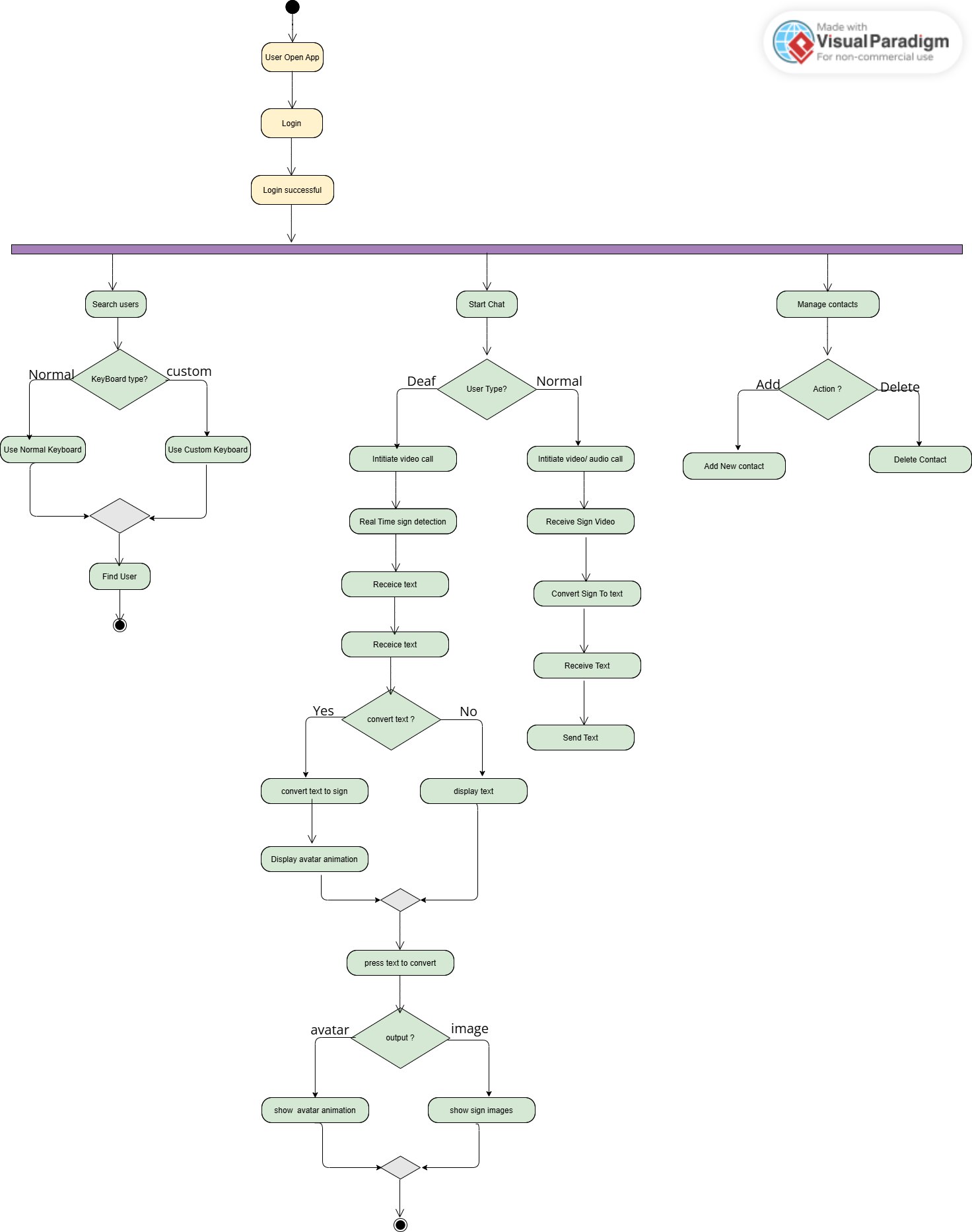
#### **3.3.5 Use Case Diagram**

* First, with a use case diagram, we will specify the expected behavior (what) of the system, not the exact method of making it happen (how). This helps us to design the system from the end user's perspective.



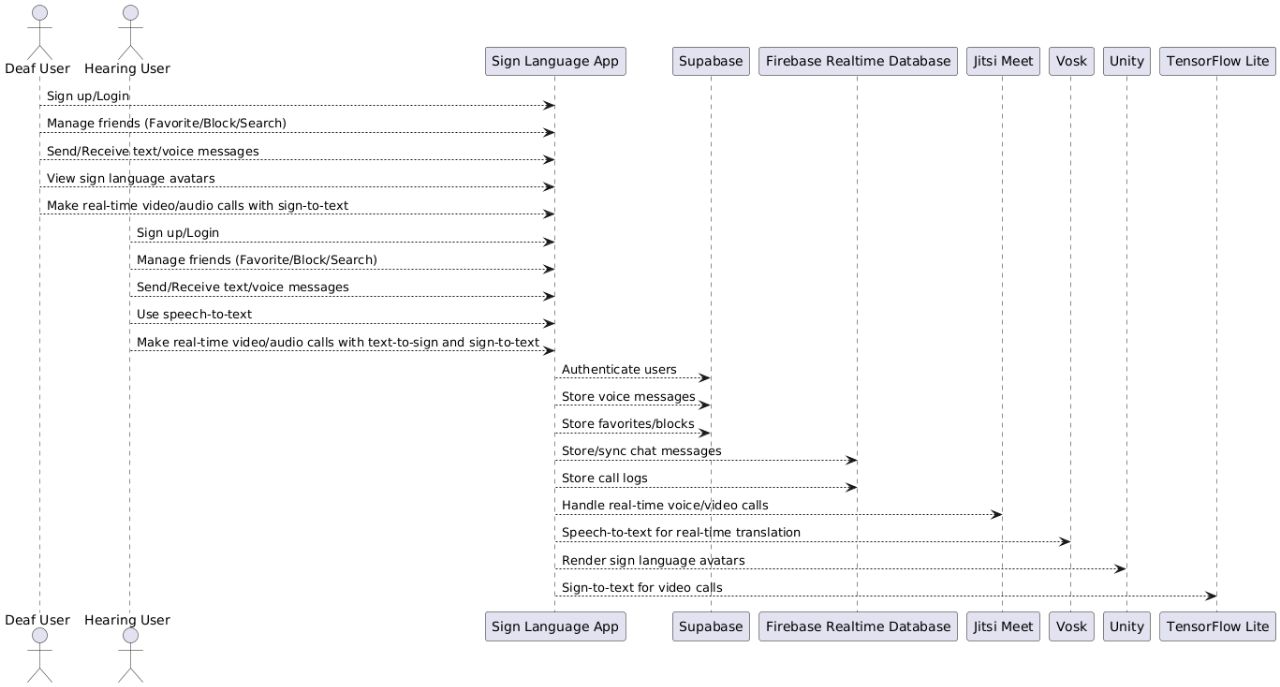
#### **3.3.6 Activity Diagram**

* An **Activity Diagram** is a type of **UML (Unified Modeling Language)** diagram that models the **workflow or business process** of a system. It is especially useful for describing the **dynamic aspects** of a system, such as **sequences of activities, decision points, parallel processes**, and more.



#### **3.3.7 Context Diagram**

#### A **Context Diagram** is a high-level visual representation of a system that shows the system as a single process and its interaction with external entities (actors, systems, or users). It’s often used in the early stages of systems design to clarify boundaries and interfaces.



Context Diagram

The context diagram provides a high-level overview of the Sign Language Application, illustrating its interactions with deaf and hearing users and external systems. It highlights real-time communication features, including AI-driven translation for video calls.

Key Actors and Systems

- Deaf User: Interacts with the app to sign up, manage friends, send/receive messages, view sign language avatars, and make real-time video calls with sign-to-text translation.

- Hearing User: Interacts with the app to sign up, manage friends, send/receive messages, use speech-to-text, and make real-time video calls with text-to-sign and sign-to-text support.

- External Systems:

- Supabase: Provides authentication, voice message storage, and friend management.

- Firebase Realtime Database: Synchronizes chat messages and stores call logs.

- Jitsi Meet: Handles real-time voice and video calls.

- Vosk: Processes real-time speech-to-text translation.

- Unity: Renders sign language avatars for text-to-sign conversion.

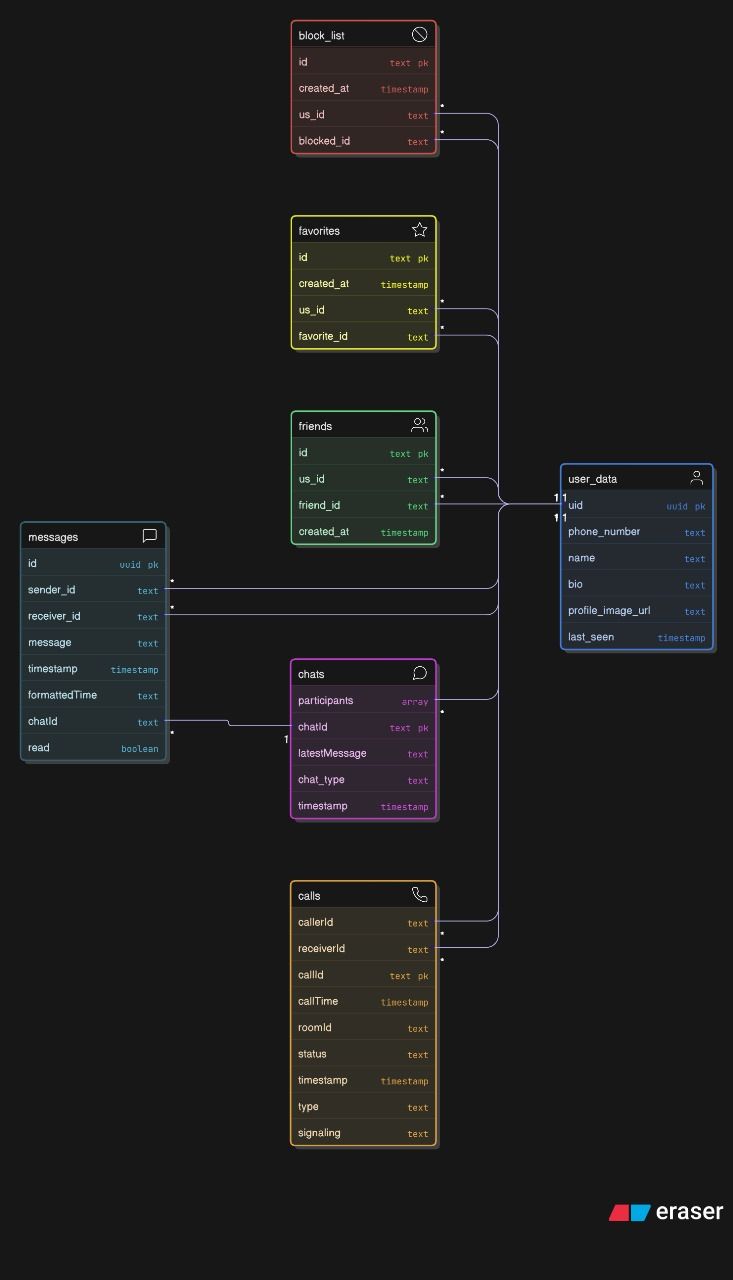
- TensorFlow Lite: Processes sign-to-text translation for video calls using a custom AI model.

Interactions

The app enables seamless communication by converting speech to text (Vosk) and sign language to text (TensorFlow Lite) during real-time video calls via Jitsi Meet. Deaf users view sign language avatars (Unity), while Firebase and Supabase manage chat and user data.

#### **3.3.8 ERD Diagram**

* An **ERD (Entity-Relationship Diagram)** is a type of data modeling diagram used to visually describe the **data entities**, their **attributes**, and **relationships** between them in a database system.



#### 

#### 

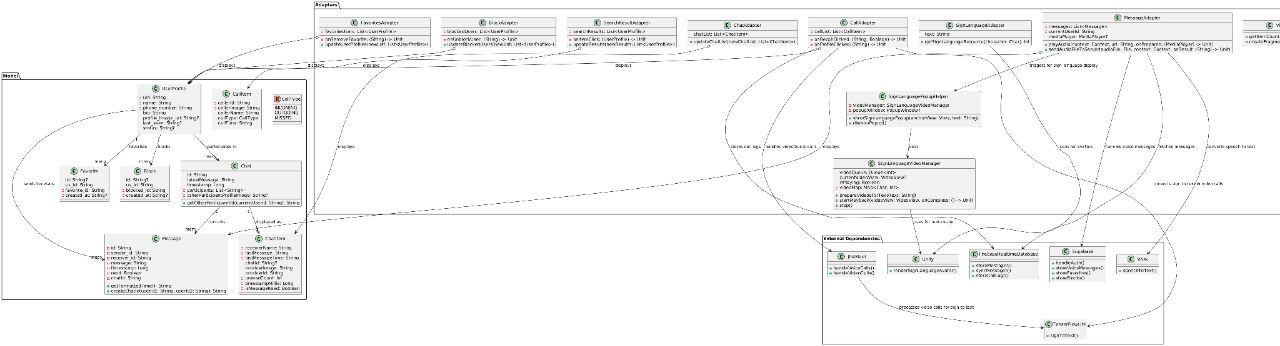
#### 

#### 

#### 

#### 

#### **3.3.9 Class Diagram**



A screenshot of a computer

AI-generated content may be incorrect.

A diagram of a diagram

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

Class Diagram

The class diagram illustrates the static structure of the Sign Language Application, detailing core classes, their attributes, methods, and relationships. It emphasizes components enabling real-time communication between deaf and hearing users through text, voice, and video calls.

Key Components

- UserProfile: Stores user data (e.g., uid, name, phone\_number) for authentication and friend management via Supabase.

- Message: Manages text and voice messages, stored in Firebase Realtime Database (text) and Supabase Storage (voice).

- SignLanguageVideoManager: Converts text to sign language avatar videos using Unity, enhancing accessibility for deaf users.

- MessageAdapter: Displays messages, supports real-time speech-to-text via Vosk, and triggers sign language popups.

- CallAdapter: Manages call logs, integrates with Jitsi Meet for real-time video/audio calls, and uses TensorFlow Lite for sign-to-text translation.

- External Services:

- Supabase: Handles authentication, voice message storage, and friend management.

- Firebase Realtime Database: Synchronizes chat messages and stores call logs.

- Jitsi Meet: Facilitates real-time voice and video calls.

- Vosk: Provides real-time speech-to-text translation.

- Unity: Renders sign language avatars.

- TensorFlow Lite: Processes sign-to-text for video calls using a custom AI model.

Relationships

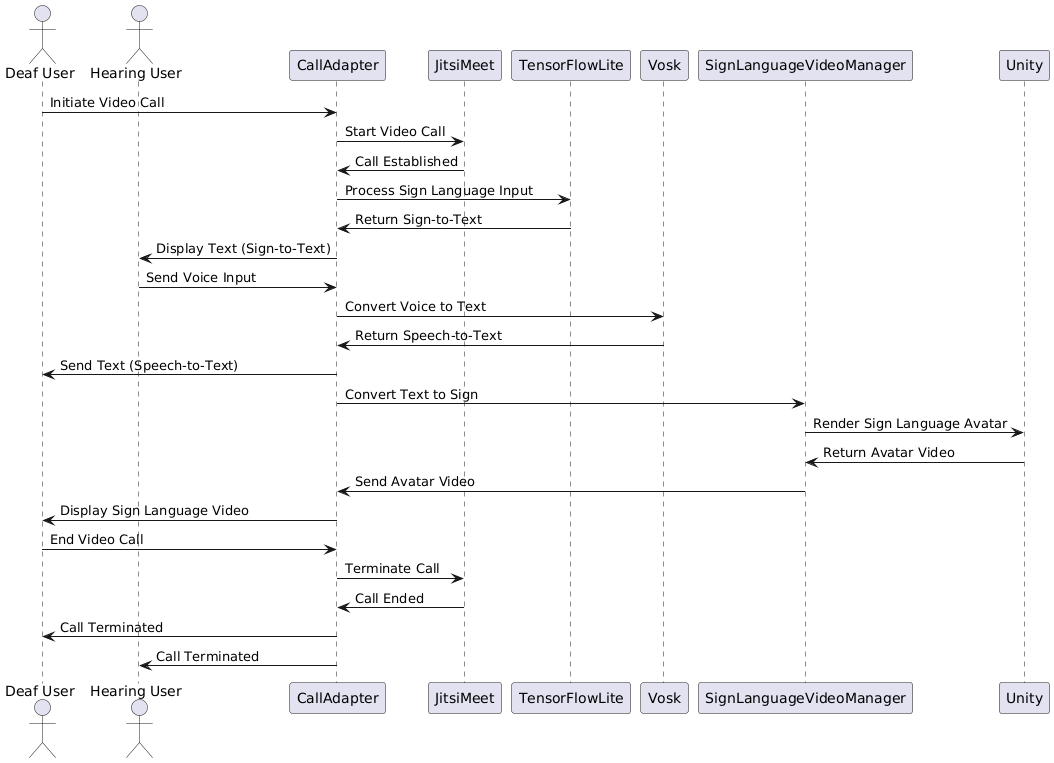
- UserProfile has a one-to-many relationship with Message, Favorite, and Block.

- SignLanguageVideoManager depends on Unity for text-to-sign conversion.

- CallAdapter interacts with Jitsi Meet for calls, TensorFlow Lite for sign-to-text, and Firebase for call logs

#### **3.3.10 Sequence Diagram**

Sequence Diagram: Demonstrates the real-time video call process, where CallAdapter manages the call, TensorFlowLite converts sign to text, Vosk converts voice to text, and SignLanguageVideoManager with Unity converts text to sign for deaf users.



#### **3.3.11 Implementation Details**

**Overview**

The **Graduation App** is an Android application designed to facilitate communication with a strong focus on **accessibility** through **sign language recognition** and **real-time communication** features. The app integrates:

* Authentication and user profiles
* Chat, video/audio calls
* A custom sign language keyboard
* **Supabase** for backend services
* **Firebase** for real-time call signaling
* **TensorFlow Lite** for gesture recognition
* **Jitsi Meet** for video conferencing

## Technologies Used

### Programming Language

* **Kotlin**

### Framework

* **Android SDK**

### Backend

* **Supabase**: Authentication, PostgreSQL database, storage, and real-time subscriptions
* **Firebase Realtime Database**: Call signaling for audio/video calls

### Machine Learning

* **TensorFlow Lite**: Sign recognition using a MobileNetV2-based model (asl\_mobilenetv2\_model.tflite)

### Video Conferencing

* **Jitsi Meet SDK**: Handles video and audio calls

### UI Libraries

* **Glide**: Image loading and caching
* **ViewBinding**: UI component binding
* **RecyclerView**: List display (e.g., search results, chats)
* **ViewPager2**: Tabbed navigation
* **ConstraintLayout**: Flexible layouts
* **Coroutines**: Asynchronous processing using Kotlin Coroutines and Flow

### Permissions

* Camera, microphone, contacts, and system overlay permissions

### Other Libraries

* **Supabase Kotlin SDK**
* **AndroidX Lifecycle**: ViewModel and LiveData
* **Material Design Components**

## Code Architecture

Follows **MVVM (Model-View-ViewModel)** architecture with a **modular directory structure**:

* Activities: Screens and user interactions
* Fragments: Reusable UI components
* Adapters: Bind data to RecyclerView/ViewPager
* Models: Represent Supabase and Firebase data structures
* ViewModels: Handle business logic and UI state
* Utils: Helper classes (e.g., keyboard, video)
* Services: Background tasks
* Supabase: Singleton client configuration

## Key Components

### 1. Models

* **UserProfile, Chat, Favorite, Block, Message**
* Stored in Supabase or Firebase for real-time data updates

### 2. ViewModels

* UserViewModel: Manages profile data and image handling
* HomeViewModel: Manages contact search and home UI
* ChatViewModel: Manages messages and updates
* BlockViewModel: Handles blocked users

### 3. Activities

* MainActivity: Login screen, session validation
* HomeActivity: Central dashboard (chats, calls)
* ProfileActivity: Profile creation/editing
* RegisterActivity: User registration
* ForgotPasswordActivity / ResetPasswordActivity: Password recovery
* reciever\_page\_activity: Recipient profile view
* VideoCallActivity, AudioCallActivity: Call interfaces
* IncomingCallActivity: UI for incoming calls
* signActivity: Sign language keyboard input
* Others: AddContactActivity, FavoritesActivity, block\_activity, chat\_page\_activity

### 4. Fragments

* ChatFragment: Chat list
* CallFragment: Call history
* ChatPageFragment: Individual chat UI

### 5. Adapters

* SearchResultAdapter, ViewPagerAdapter, ChatAdapter

### 6. Services

* FloatingTextService: Displays recognized sign text as an overlay

### 7. Utilities

* CustomKeyboardUtils: Manages custom keyboard
* SignLanguageVideoManager: Plays sign videos
* SignLanguageRecognizer: TFLite model processing
* CallActionReceiver: Handles call actions

### 8. Supabase Integration

* Configured in Supabase.kt: handles Auth, Postgrest, Storage, Realtime

## Implemented Features

### 1. User Authentication

* **Login (MainActivity)**: Email/password via Supabase Auth
* **Registration (RegisterActivity)**: Verifies email, redirects to profile setup
* **Password Recovery**: Email reset via Supabase with custom redirect
* **Logout**: Clears session via Supabase Auth

### 2. User Profile Management

* Profile setup/edit (ViewSwitcher for TextView ↔ EditText)
* Image uploads via Supabase Storage
* Profile display in various screens

### 3. Contact Management

* **Search**: Real-time contact search (min. 2 characters)
* **Add Contact**: Launch via FAB (logic not provided)
* **Favorites**: Managed via UserViewModel, stored in Supabase
* **Block List**: Block/unblock users, restricts communication

### 4. Chat System

* **Chat List (ChatFragment)**: Real-time updates via Supabase
* **Individual Chat**: Messaging with optional sign input
* **Floating Overlay**: Shows recognized signs during chat

### 5. Video and Audio Calls

* **Call Initiation**: Generates roomId, stores in Firebase
* **Incoming Calls**: UI shows caller info, accept/decline
* **Call Signaling**: Managed via Firebase Realtime Database
* **Permissions**: For camera/mic, floating overlays

### 6. Sign Language Recognition

* **Recognition**: MobileNetV2 TFLite model processes gestures
* **Text-to-Sign**: Plays sign language video for certain text
* **Sign-to-Text**: Detected text displayed as a floating overlay

### 7. Custom Sign Language Keyboard

* Buttons for A-Z, space, delete, enter
* Integrated into chat and search interfaces

### 8. Accessibility Features

* Floating text overlay (WindowManager)
* Sign language videos
* Custom keyboard for hearing-impaired users

## Detailed Functionality

### Supabase Integration (Supabase.kt)

* Initializes client with Auth, Postgrest, Storage, and Realtime
* Singleton access: Supabase.client

### Authentication Flow

* signIn: Email/password authentication and user validation
* signUp: Registers and sends verification email
* resetPasswordForEmail: Sends reset email
* modifyUser: Updates user password

### Profile Management

* fetchUserData: Retrieves user profile via Supabase
* saveProfileData: Uploads image, stores data
* deleteProfileImage: Deletes profile image

### Search & Contacts

* searchContacts: Queries Supabase and local contacts
* toggleSearchView: Manages search UI and keyboard visibility

### Chat System

* fetchChats: Loads chat list from Supabase in real-time
* Message sending/receiving stored in messages table
* Floating overlay shows recognized gestures

**1. Login Screen**

* **Functionality**:
  + Allows users to log in using their email and password via Supabase Auth.
  + Checks for an existing user session and redirects to HomeActivity if the profile is complete, or ProfileActivity if incomplete.
  + Displays a sign language video for "log in" using SignLanguageVideoManager.
  + Tracks failed password attempts and suggests password reset after two failures.
* **Expected UI Elements**:
  + Email input field.
  + Password input field with visibility toggle.
  + Login button.
  + "Forgot Password?" link.
  + Sign language video player.
  + Error message (optional).

**2. Register Screen**

* **Functionality**:
  + Enables users to register using their email and password.
  + Sends a verification email and redirects to MainActivity upon success.
* **Expected UI Elements**:
  + Email input field.
  + Password input field.
  + Confirm password input field.
  + Sign-up button.
  + Confirmation message (optional).

**3. Forgot Password Screen**

* **Functionality**:
  + Allows users to request a password reset via email.
  + Redirects to ResetPasswordActivity through a custom reset link.
* **Expected UI Elements**:
  + Email input field.
  + Send Reset Instructions button.
  + "Back to Login" link.

**4. Reset Password Screen**

* **Functionality**:
  + Allows users to set a new password (minimum 6 characters) after receiving a reset link.
  + Updates the password in Supabase Auth.
* **Expected UI Elements**:
  + New password input field.
  + Confirm change button.
  + Success message (optional).

**5. Home Screen**

* **Functionality**:
  + Serves as the main navigation hub with a bottom navigation bar featuring "Chats" and "Calls" tabs.
  + Supports real-time contact search using HomeViewModel and displays results in SearchResultAdapter.
  + Includes a custom sign language keyboard via CustomKeyboardUtils.
  + Displays a welcome message and profile picture.
* **Expected UI Elements**:
  + Bottom navigation bar (Chats, Calls).
  + Search bar.
  + Search results list (RecyclerView).
  + Sign language keyboard.
  + Floating Action Button (FAB) for adding a contact.

**6. Profile Screen**

* **Functionality**:
  + Allows users to set up or edit their profile (name, phone number, bio, profile picture).
  + Uses ViewSwitcher to toggle between view and edit modes.
  + Uploads images to Supabase Storage and updates user\_data via UserViewModel.
* **Expected UI Elements**:
  + Name display/edit field.
  + Phone number display/edit field.
  + Bio display/edit field.
  + Profile picture selection/delete button.
  + Save changes button.

**7. Receiver Page Screen**

* **Functionality**:
  + Displays contact details (name, picture) and offers options for video/audio calls or adding to favorites/block list.
  + Checks block status before initiating a call.
* **Expected UI Elements**:
  + Contact picture and name.
  + Video call and audio call buttons.
  + Add to favorites/block buttons.

**8. Chat Page Screen**

* **Functionality**:
  + Displays a one-on-one chat with the ability to send/receive messages.
  + Supports sign language input via a custom keyboard and shows recognized text in a floating layer (FloatingTextService).
  + Uses ChatViewModel for real-time message updates.
* **Expected UI Elements**:
  + Message list (RecyclerView).
  + Text input field.
  + Send button.
  + Sign language keyboard.
  + Floating text layer.

**9. Video Call Screen**

* **Functionality**:
  + Initiates a video call using Jitsi Meet and displays the call interface.
  + Starts FloatingTextService to display recognized sign language text.
* **Expected UI Elements**:
  + Jitsi Meet video screen.
  + Control buttons (end call).
  + Floating text layer.

**10. Incoming Call Screen**

* **Functionality**:
  + Displays an incoming call interface with the caller's name and accept/reject options.
  + Listens to status changes in Firebase and redirects to chat\_page\_activity upon rejection.
* **Expected UI Elements**:
  + Caller’s name.
  + Accept/reject buttons.
  + Call type indicator (video/audio).

**11. Favorites Screen**

* **Functionality**:
  + Displays a list of favorited users and allows removal.
* **Expected UI Elements**:
  + Favorites list (RecyclerView).
  + Remove button.

**12. Block List Screen**

* **Functionality**:
  + Displays a list of blocked users and allows unblocking.
* **Expected UI Elements**:
  + Blocked users list (RecyclerView).
  + Unblock button.

**13. Add Contact Screen**

* **Functionality**:
  + Allows adding a new contact (details not fully specified in the code).
* **Expected UI Elements**:
  + First name input field.
  + Last name input field.
  + Phone number input field.
  + Save button.

**5.5 Navigation Flow**

The navigation flow between screens is based on user scenarios as follows:

1. **App Start**:
   * Starts at MainActivity (Login Screen).
   * Successful login with complete profile → HomeActivity.
   * Successful login with incomplete profile → ProfileActivity.
   * "Forgot Password?" link → ForgotPasswordActivity.
2. **New User Registration**:
   * From MainActivity, select register (assumed link) → RegisterActivity.
   * Successful registration → MainActivity.
3. **Password Reset**:
   * From ForgotPasswordActivity, send request → ResetPasswordActivity via reset link.
   * Password updated → MainActivity.
4. **Navigation from Home Screen**:
   * From HomeActivity:
     + Search for contact → reciever\_page\_activity.
     + Select "Chats" tab → ChatPageFragment within HomeActivity.
     + Select "Calls" tab → Call history view (not fully specified).
     + FAB click → AddContactActivity.
     + Profile click → ProfileActivity.
     + Navigate to Favorites → FavoritesActivity.
     + Navigate to Block List → block\_activity.
5. **Chat and Call Actions**:
   * From reciever\_page\_activity:
     + Video call option → VideoCallActivity.
     + Audio call option → AudioCallActivity (assumed).
   * Incoming call → IncomingCallActivity.
     + Accept call → VideoCallActivity or AudioCallActivity.
     + Reject call → chat\_page\_activity with "missed" status.
6. **Profile Management**:
   * From ProfileActivity, save changes → return to HomeActivity

#### **Navigation Flow Diagram (Textual Representation) 9. Video Call Screen**

* **Functionality**:
  + Initiates a video call using Jitsi Meet and displays the call interface.
  + Starts FloatingTextService to display recognized sign language text, enabling real-time transcription (noted as of 11:20 PM EEST, Thursday, June 12, 2025).
* **Expected UI Elements**:
  + Jitsi Meet video screen.
  + Control buttons (end call).
  + Floating text layer

**[MainActivity (Login)] --> [HomeActivity]**

**--> [ProfileActivity]**

**--> [ForgotPasswordActivity] --> [ResetPasswordActivity] --> [MainActivity]**

**--> [RegisterActivity] --> [MainActivity]**

**[HomeActivity] --> [reciever\_page\_activity] --> [VideoCallActivity]**

**--> [ChatPageFragment]**

**--> [FavoritesActivity]**

**--> [block\_activity]**

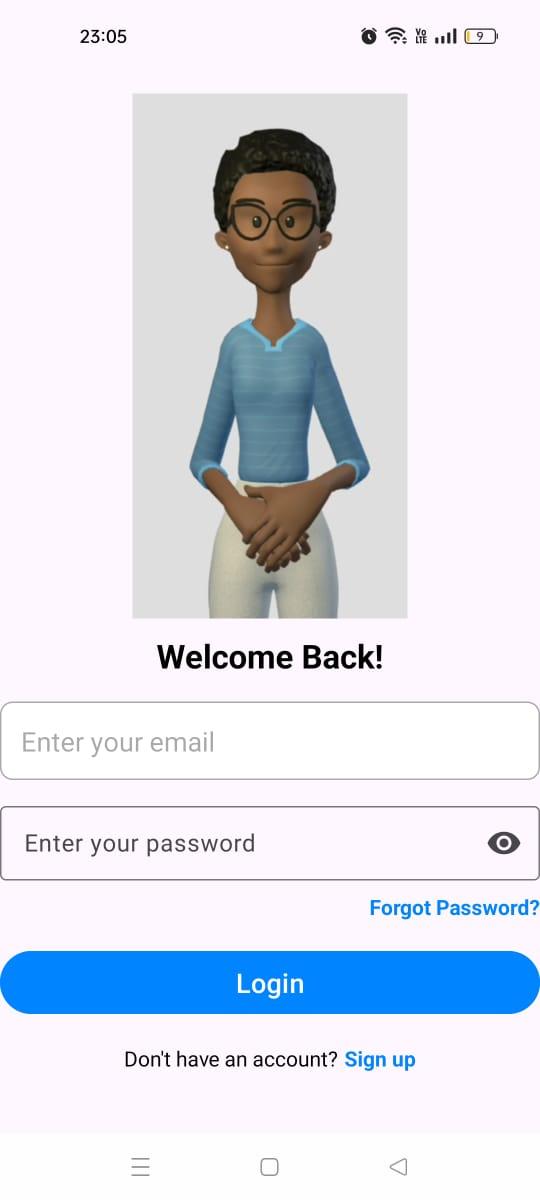
**--> [AddContactActivity]**

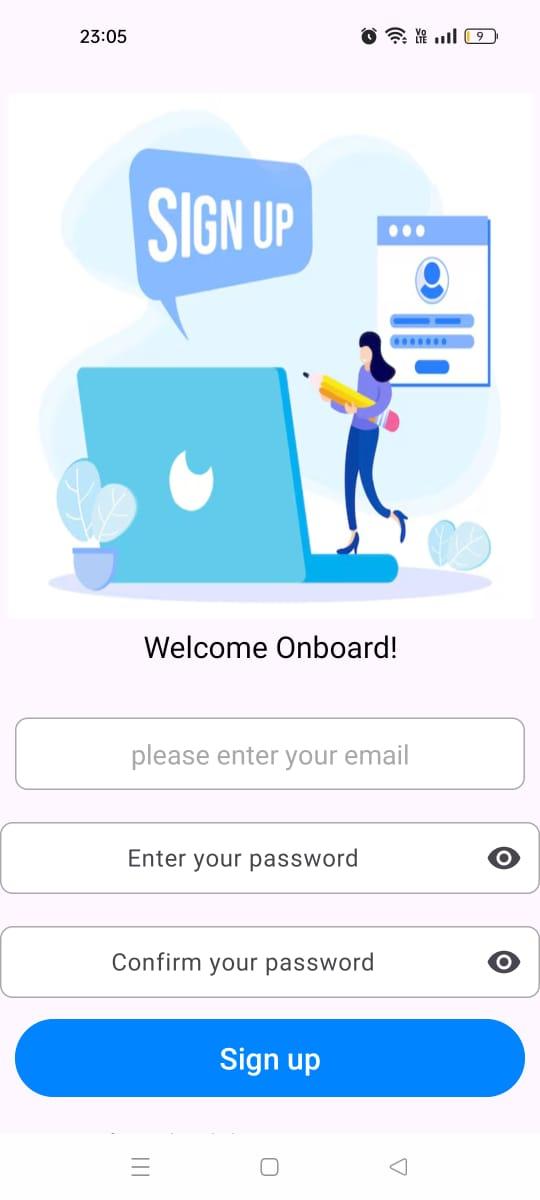
**--> [ProfileActivity]**

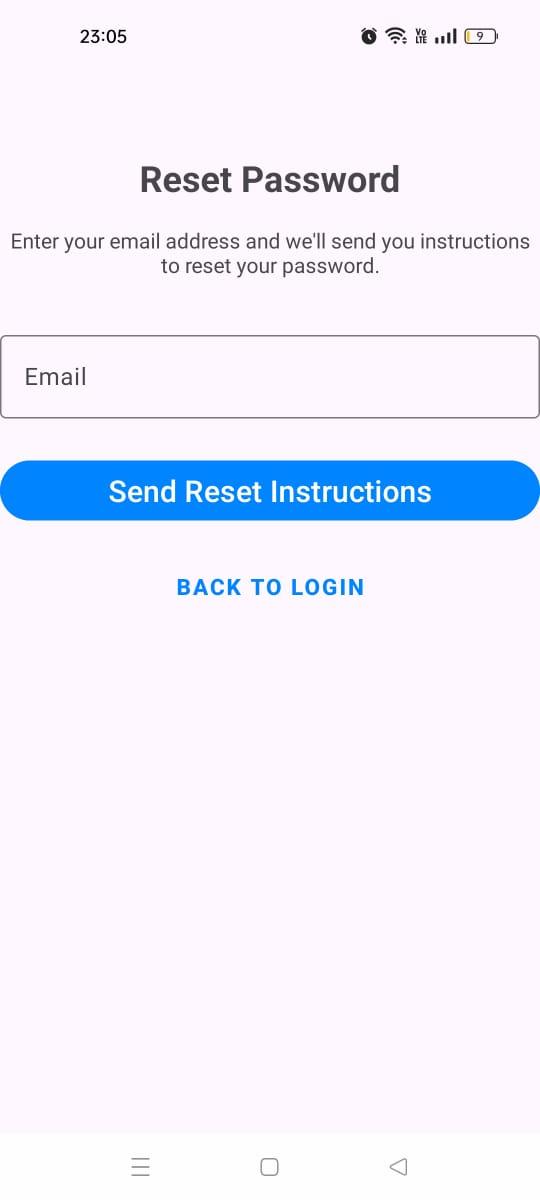
**[IncomingCallActivity] --> [VideoCallActivity]**

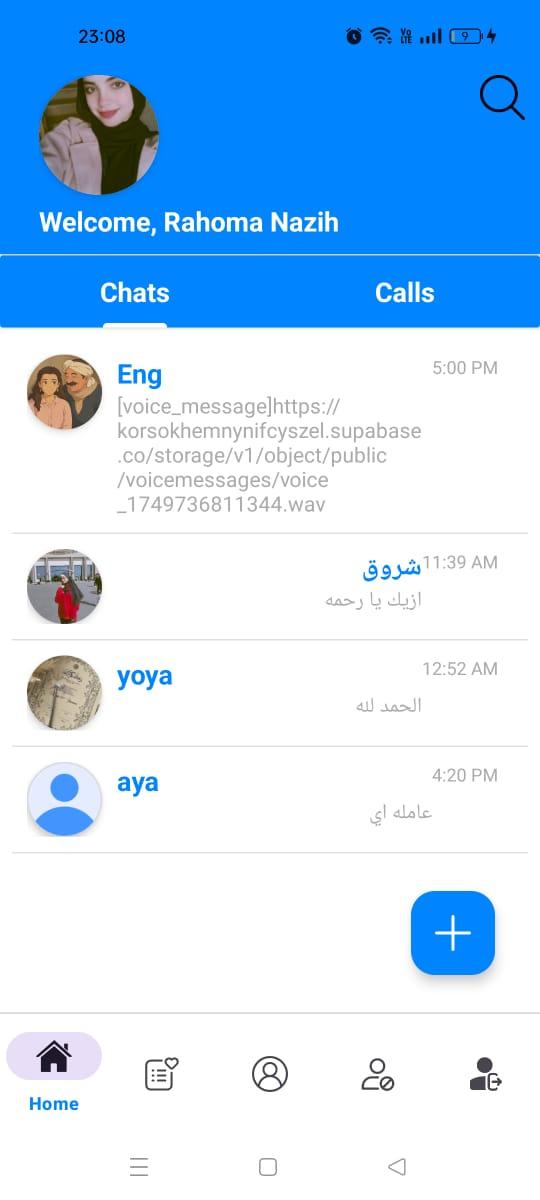
**--> [chat\_page\_activity]**

**[reciever\_page\_activity] --> [AudioCallActivity] (assumed)**

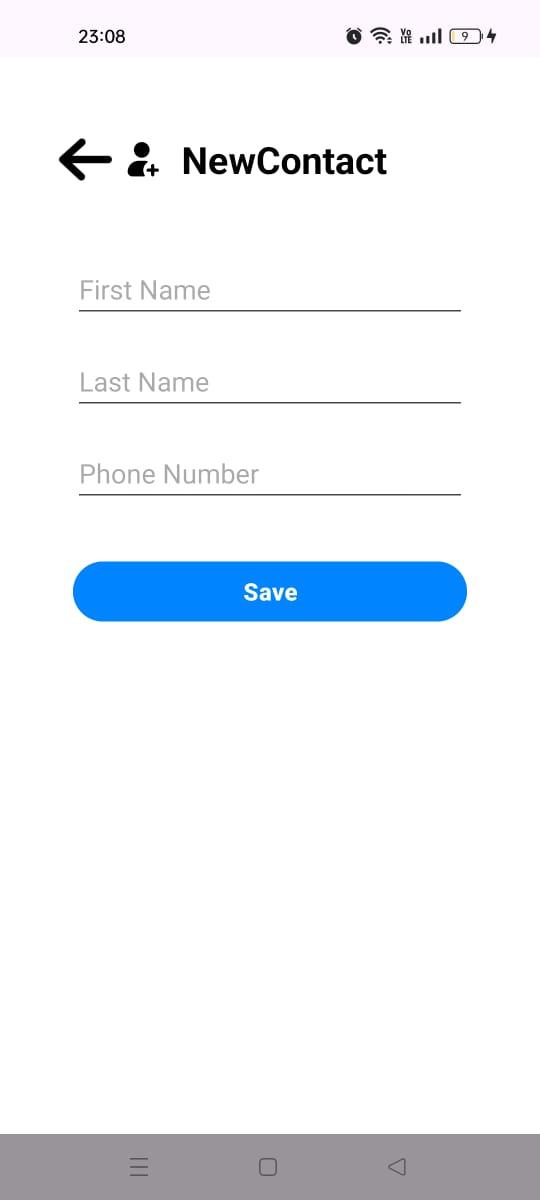


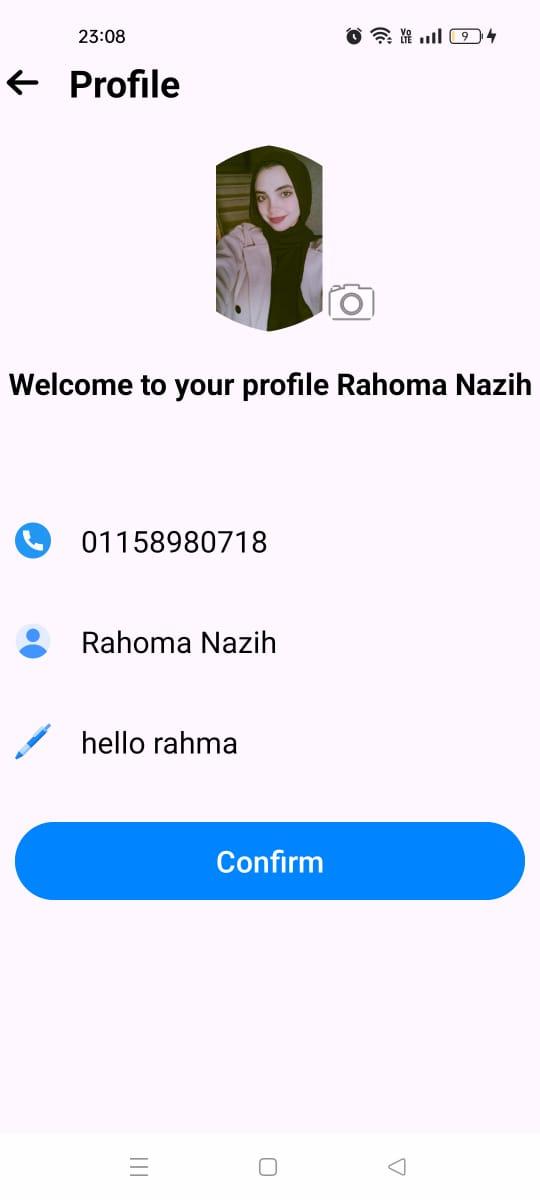


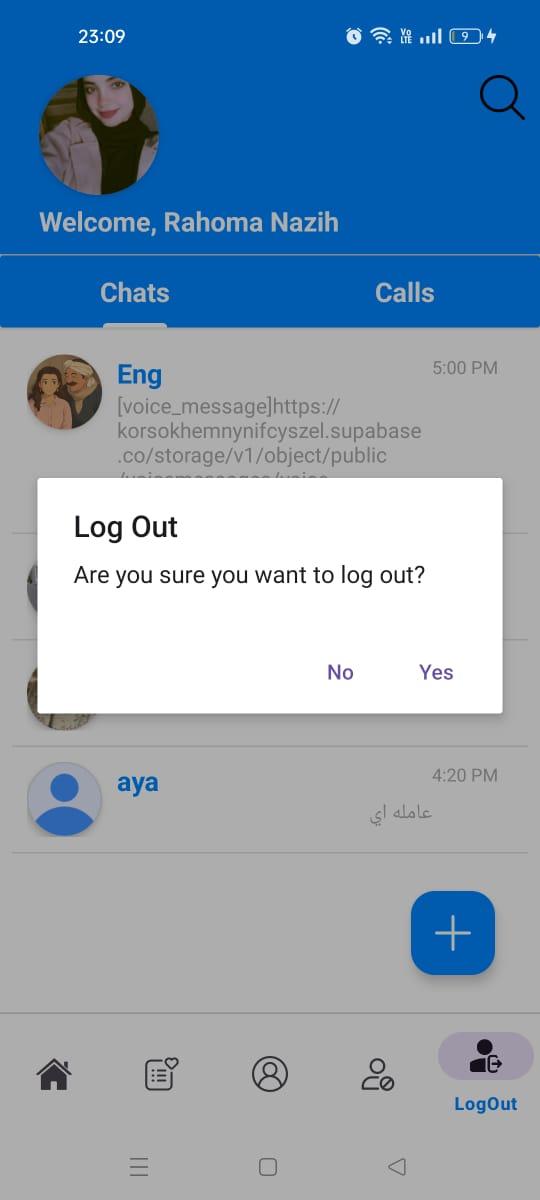




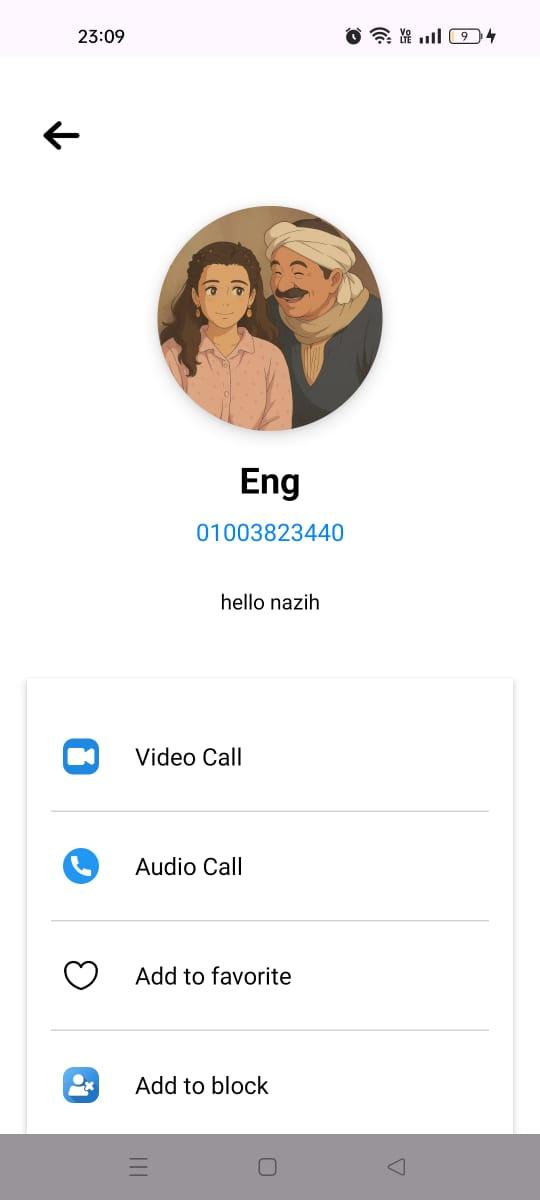












## Chapter 4: Dataset, Adopted Methodology, Models, Experimental Setup, Results, and Limitations

In this section, we present the dataset, utilized neural network models, data preparations, and processing.

### 4.1 Dataset

We utilized the Arabic Alphabets Sign Language Dataset (ArASL) [**https://www.kaggle.com/datasets/muhammadalbrham/rgb-arabic-alphabets-sign-language-dataset/data**](https://www.kaggle.com/datasets/muhammadalbrham/rgb-arabic-alphabets-sign-language-dataset/data)

in this study. **The used dataset comprises 7856 images** depicting Arabic Sign Language letters. These images were contributed by over 40 individuals and **cover 33 standard Arabic signs and letters(Classes)**. It is worth noting that each class within the dataset contains a varying number of images. To organize and label the data, we employed a CSV file that associates each Arabic Sign Language image with its corresponding class label based on the image file’s name. For visual reference, you can see some examples of the training data in Figure 1.

**Figure 1**

### 

### 4.2 Adopted Methodology

The adopted methodology section serves as a guide for how this work was carried out, encompassing the entire process from data collection to the production of study findings. We will delve into the major steps of the methodology as depicted in the **flowchart**, **providing detailed explanations**. **Additionally**, **we will provide brief explanations of the models that were utilized in our study**.

As depicted in Figure 2, we first import the essential packages and libraries, including Keras, Pandas, and Matplotlib, etc. Then, define the path of dataset the “RGB ArSL dataset”. As mentioned earlier, the dataset contains 7856 images for 33 Arabic Sign Language characters. The first step after loading the dataset is to prepare the data to enter the model by **implementing some preprocessing steps**. Due to the dataset’s issues, which means that every category holds a varying number of images, it may result in biased detection outcomes; hence, we solve this issue by aiming to avoid any inconsistencies and biases in the testing results. We allocated a fixed number of samples for each category in the dataset. Moreover, to complete preparing the dataset to enter the model, another data preprocessing step is performed, which is image resizing. The ArASL images are in different sizes, so all images were resized to a standard resolution of 64 × 64 pixels in CNN and 224 x 224 in MobileNetV2. In addition, image normalization is conducted to make the images more consistent in terms of contrast, color, and brightness, etc. After that, data augmentation was applied. It is the process of generating new data from existing data to increase the data size and variety, thereby achieving better results. In our study, we implemented different augmentation techniques, including rescaling, zooming, flipping, and shifting, etc.

**Figure 2**

### 

Moving on to model development, the dataset was divided into training, validation, and testing sets with **80%** ratio for training and **10%** for testing and finally **10%** for validation. The training set was entered into 3 different chosen models, leveraging their efficiency and robustness in extracting complex patterns from data. These models are MobileNetV2, CNN with a basic preprocessing, and CNN with an advanced preprocessing. The models’ weights are loaded using the ImageNet model in case of MobileNetV2, and the prediction layer is added using the softmax activation function after the last fully connected layer. We then fine-tuned this model using various settings by adjusting hyper parameters, including different learning rates and different number of epochs. After fine-tuning this model (MobileNetV2), we validate the effectiveness of the models MobileNetV2 and CNN on the validation set by measuring the accuracy score and visualizing the results for better understanding. Finally, we choose the best model.

### 4.3 Models

As it is known in computer science, deep learning or CNN is the best technique that can be used for a recognition system. According to the previous most-used deep learning trained architectures are MobileNet-V2 among others.

CNN can be fed raw image pixel values rather than pre-processed feature vectors, unlike standard machine learning applications. The general architecture of CNN. CNN’s usual architecture is made up of layers of computing units (gates), which are:

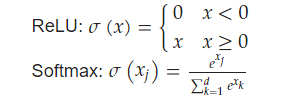
### 

**CNN Architecture.**

1. Convolutional Layers: A grid that provides input to each gate. Each gate’s weights are connected so that each gate recognizes the same feature. There are various sets of gates similar to this, organized in multiple channels (layers) to learn different aspects.
2. Pooling Layers: This works as a down--sampling layer by reducing the number of gates. Each of the “*k* × *k*” input grid gates are usually reduced to a single cell/gate by choosing the maximum input value or the average of all inputs. The layer is scanned with a tiny *k* grid and a stride is chosen so that the grid covers the layer without overlapping.
3. Fully linked Layers: Each gate’s output is connected to the input of the next layer’s gate. (Also referred to as auto encoder levels). These transform a vectorized version of the input into a normalized vectorized output. The output vector is a set of probabilities that serve as the classification signature.
4. Convolution Layers: Consider 1D convolution, suppose the input vector is *f* and the kernel is *g* whose length is *m*. The following equation shows the center of kernel shifted and multiplied.

**Activation functions:**

Activation functions define the output of a neuron based on a given set of inputs. Some commonly used activation functions *σ*(x) with their gradients.



With a pooling type that can be:

Max-pooling

Average Pooling

L2 norm Pooling

1. Fully Connected Dense Layers: After the pooling layers, pixels of pooling layers are stretched to a single column vector. These vectorized and concatenated data points are fed into dense layers, known as fully connected layers for the classification. In some cases, the output layer of a deep network uses a soft max procedure. Similar to logistic regression: Given n vectors {x1, x2,…,xn} with labels {*l*1, *l*2,…,*l*n}, where *l*i ∈ {0, 1} (as a binary classification task). With a weight vector w one can define by the following equation:

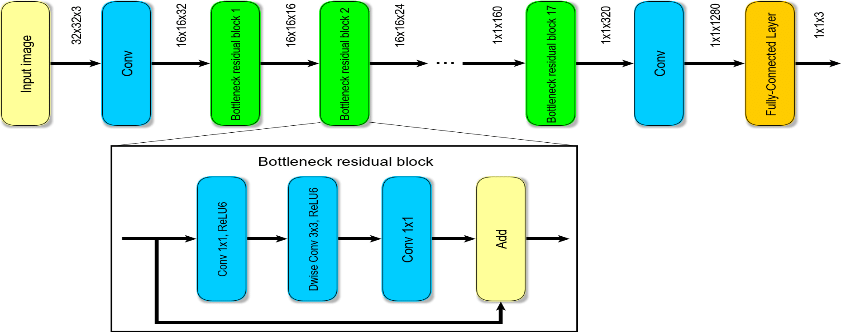
### 

where *σ* represents sigmoid function.

In the following, the selected CNN models have been presented.

Pre-trained models have found extensive application in the field of computer vision due to their remarkable capacity to uncover hidden patterns and generalize effectively, even with small datasets and limited resources. In this section, we will explain the utilized pre-trained models in our methodology.

**MobileNetV2:** MobileNetV1 emerged as a family of computer vision neural networks designed to support classification and detection in standard functions primarily built for mobile devices. It can run these networks on mobile devices, enhancing user experiences by providing benefits such as always-on access, privacy, security, and power efficiency. Subsequently, MobileNetV2 was introduced to power the next generation of mobile computer vision applications. MobileNetV2 represents a significant improvement over MobileNetV1 and incorporates the latest technology for mobile optical recognition, including support for various convolutional neural network applications such as object detection, classification, and semantic segmentation. Released as part of the TensorFlow-Slim image classification library, MobileNetV2 builds on ideas from MobileNetV1, using separate depthwise convolutions as efficient building blocks. Additionally, MobileNetV2 introduces new architectural features, including linear bottlenecks between layers and shortcut connections between bottlenecks.



**MobileNetV2 Architecture.**

### 4.4 Experiments and Results

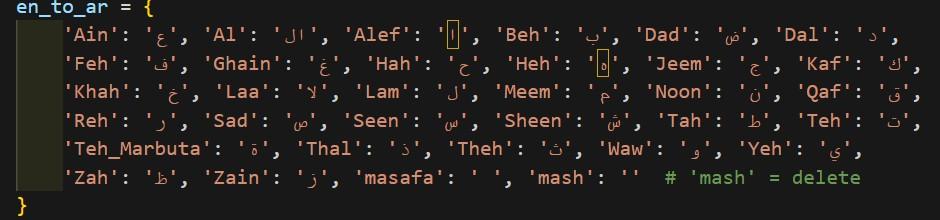
Due to the remarkable success of convolutional neural networks (CNNs) in the field of sign language recognition, we conducted a comprehensive study to compare the performance of pre-trained models. Our goal was to determine the most effective model for recognizing signs using transfer learning and traditional CNN. We used the ArASL dataset in the training, testing, and validation phases, which consisted of a substantial 7856 images, each depicting one of 33 Arabic signs.

| **Technique** | **Accuracy** | **Language** | **Dataset** | **Dataset size** |
| --- | --- | --- | --- | --- |
| CNN with a basic Preprocessing | 79.06% Test accuracy | Arabic | RGB Arabic Alphabets Sign Language | 7,857 images |
| Preprocessed CNN | Recognition accuracy of 82.69% | Arabic | RGB Arabic Alphabets Sign Language | 7,857 images |
| Pre-Trained MobileNetV2 | Top-1 Recognition accuracy of 87.77% | Arabic | RGB Arabic Alphabets Sign Language | 7,857 images |

**Our proposed technique comprised several key steps:**

* **(I) Basic CNN**

**Define English to Arabic mapping for labels to be used in label encoder.**



To map each Arabic letter. For example   
**Label Map: {0: 'ع', 1: 'ال', 2: 'ا', 3: 'ب', 4: 'ض', 5: 'د', 6: 'ف', 7: 'غ', 8: 'ح', 9: 'ه', 10: 'ج', 11: 'ك', 12: 'خ', 13: 'لا', 14: 'ل', 15: 'م', 16: 'ن', 17: 'ق', 18: 'ر', 19: 'ص', 20: 'س', 21: 'ش', 22: 'ط', 23: 'ت', 24: 'ة', 25: 'ذ', 26: 'ث', 27: 'و', 28: 'ي', 29: 'ظ', 30: 'ز', 31: ' ', 32: ''}**

**Split data Train/Val/Test split (80/10/10) for true unseen evaluation.**

**CNN architecture used.**

****

**## 4 Convolutional Layers:**

**# 1st Conv Layer: 32 filters, 3x3 kernel.**

**# 2nd Conv Layer: 64 filters, 3x3 kernel.**

**# 3rd Conv Layer: 128 filters, 3x3 kernel.**

**# 4th Conv Layer: 256 filters, 3x3 kernel.**

**# 4 MaxPooling Layers:**

**# After each convolutional layer, MaxPooling reduces the feature map size.**

**# 1 Flatten Layer:**

**# Converts the 2D output of the final MaxPooling layer into a 1D vector.**

**# 2 Dense (Fully Connected) Layers:**

**# First Dense Layer: 256 neurons with ReLU activation.**

**# Second Dense Layer: 128 neurons with ReLU activation.**

**# 1 Dropout Layer after each Dense layer:**

**# Prevents overfitting by randomly disabling neurons during training.**

**# 1 Output Layer:**

**# Uses softmax activation to output class probabilities.**

**Preprocessing Methods that were chosen to be applied on images:**

**Just Resizing the image then normalization.**

### 

Normalize images Converts pixel values from [0, 255] to [0, 1] to improve model performance.

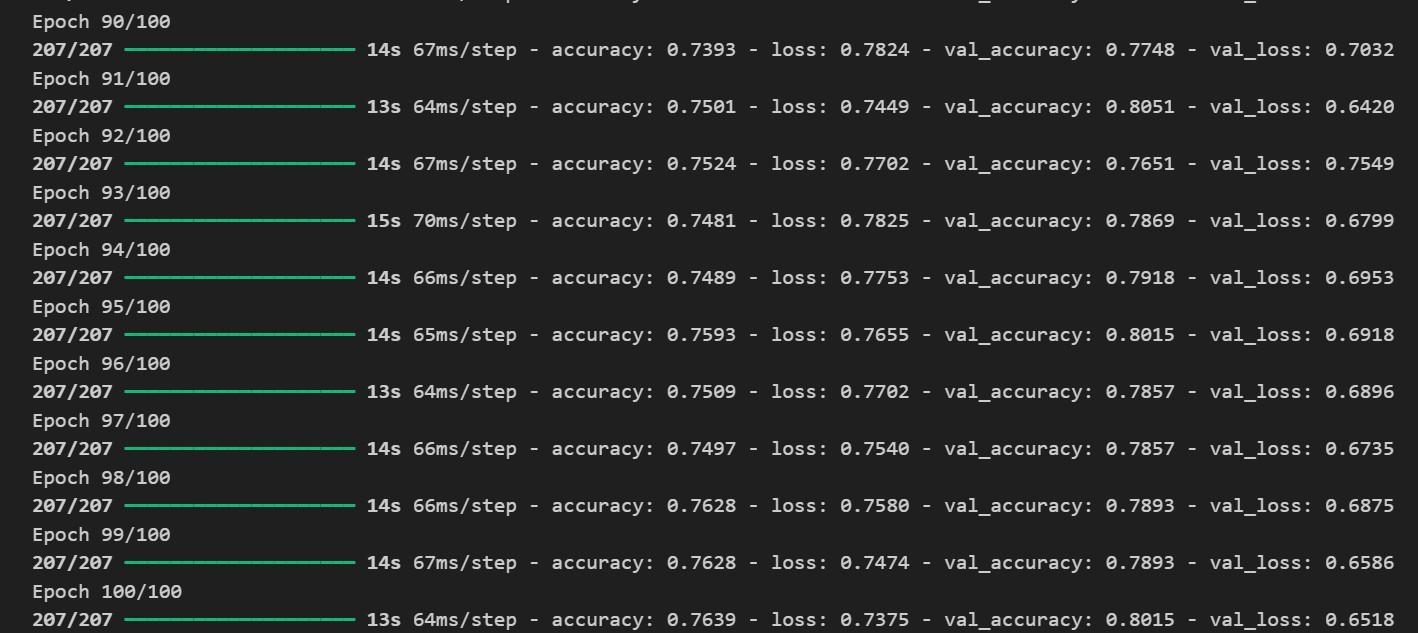
X = X.astype('float32') / 255.0

**Using Adam Optimizer with learning rate 0.001 and the model trained for 100 epochs.**

**Data augmentation: to improve the model’s generalization and mitigate overfitting, we applied data augmentation techniques.**

### 

**Sample of last 10 training epochs**



**The model saved for later use.**

**Visualizations**

**Confusion Matrix**

****

**Classification Report:**

precision recall f1-score support

Ain 0.79 0.88 0.83 25

Al 0.79 0.96 0.87 28

Alef 0.89 0.83 0.86 29

Beh 0.77 0.77 0.77 31

Dad 0.84 0.96 0.90 27

Dal 0.77 0.74 0.76 23

Feh 0.70 0.56 0.62 25

Ghain 0.86 0.83 0.84 23

Hah 0.90 0.72 0.80 25

Heh 0.70 0.76 0.73 25

Jeem 0.68 0.71 0.70 21

Kaf 0.60 0.69 0.64 26

Khah 0.75 0.72 0.73 25

Laa 0.96 0.85 0.90 27

Lam 0.96 0.92 0.94 26

Meem 0.81 0.85 0.83 26

Noon 0.86 0.75 0.80 24

Qaf 0.83 0.68 0.75 22

Reh 0.71 0.77 0.74 22

Sad 0.65 0.89 0.75 27

Seen 0.71 0.81 0.76 27

Sheen 1.00 0.96 0.98 28

Tah 0.79 0.68 0.73 22

Teh 0.80 0.77 0.79 31

Teh\_Marbuta 0.78 0.81 0.79 26

Thal 0.79 0.75 0.77 20

Theh 0.78 0.60 0.68 30

Waw 0.83 0.80 0.82 25

Yeh 0.92 0.89 0.91 27

Zah 0.73 0.70 0.71 23

Zain 0.67 0.80 0.73 20

masafa 1.00 1.00 1.00 20

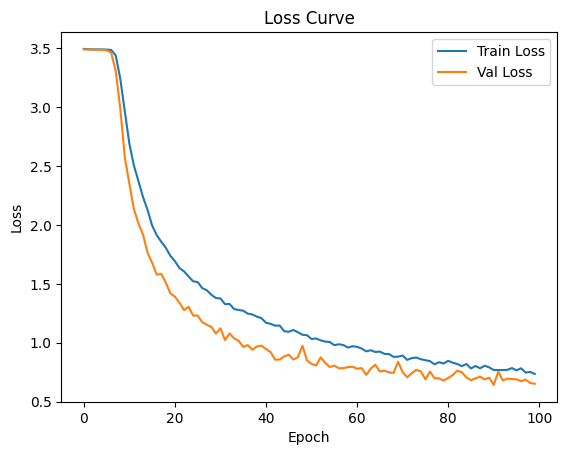
mash 1.00 1.00 1.00 20

accuracy 0.80 826

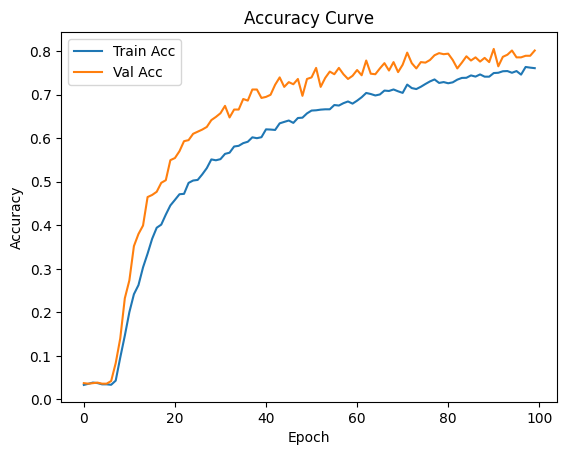
macro avg 0.81 0.80 0.80 826

weighted avg 0.81 0.80 0.80 826

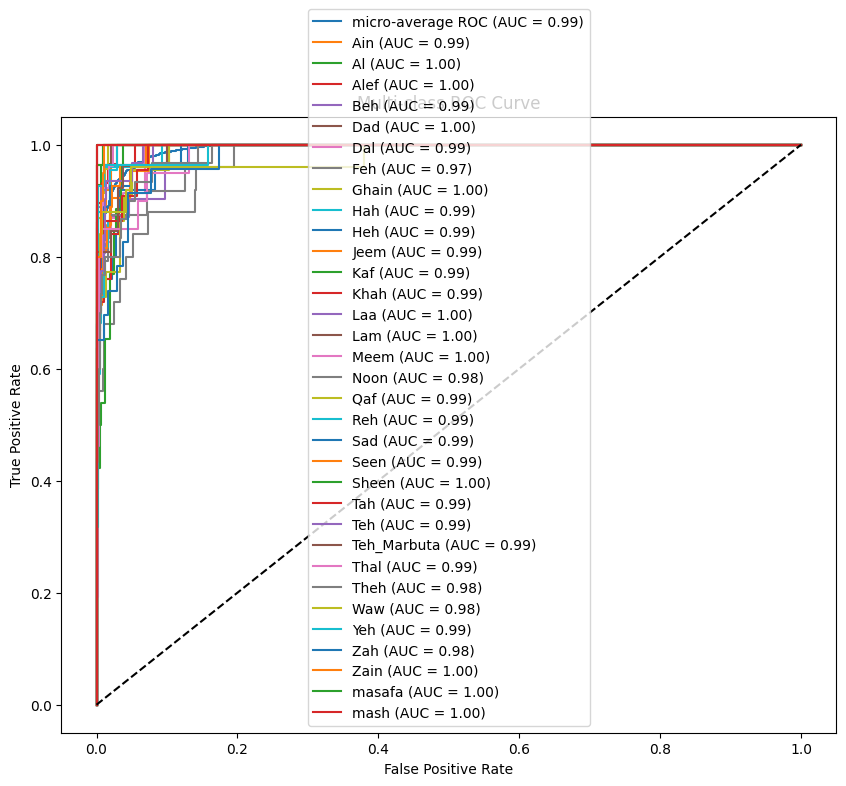
**Loss Curve**



**Accuracy Curve**

****

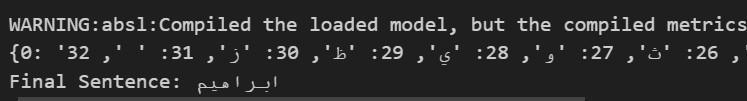
**Multi-Class ROC Curve**

****

**Test Accuracy: 79.06%**

**Validation Accuracy: 80.15%**

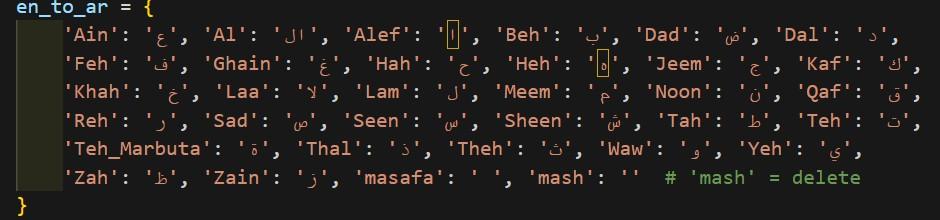
**Sample of the Real Time output**

****

* **(II) Preprocessed CNN**

we initiated the process by carefully preprocessing the images.

Define English to Arabic mapping for labels to be used in label encoder.



To map each Arabic letter. For example   
**Label Map: {0: 'ع', 1: 'ال', 2: 'ا', 3: 'ب', 4: 'ض', 5: 'د', 6: 'ف', 7: 'غ', 8: 'ح', 9: 'ه', 10: 'ج', 11: 'ك', 12: 'خ', 13: 'لا', 14: 'ل', 15: 'م', 16: 'ن', 17: 'ق', 18: 'ر', 19: 'ص', 20: 'س', 21: 'ش', 22: 'ط', 23: 'ت', 24: 'ة', 25: 'ذ', 26: 'ث', 27: 'و', 28: 'ي', 29: 'ظ', 30: 'ز', 31: ' ', 32: ''}**

**Split data Train/Val/Test split (80/10/10) for true unseen evaluation.**

**CNN architecture used.**

****

**## 4 Convolutional Layers:**

**# 1st Conv Layer: 32 filters, 3x3 kernel.**

**# 2nd Conv Layer: 64 filters, 3x3 kernel.**

**# 3rd Conv Layer: 128 filters, 3x3 kernel.**

**# 4th Conv Layer: 256 filters, 3x3 kernel.**

**# 4 MaxPooling Layers:**

**# After each convolutional layer, MaxPooling reduces the feature map size.**

**# 1 Flatten Layer:**

**# Converts the 2D output of the final MaxPooling layer into a 1D vector.**

**# 2 Dense (Fully Connected) Layers:**

**# First Dense Layer: 256 neurons with ReLU activation.**

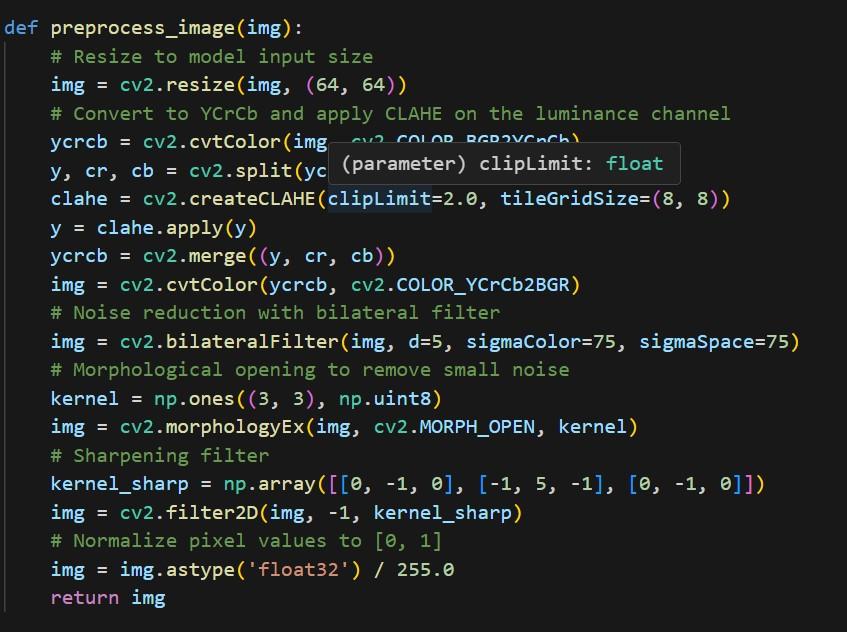
**# Second Dense Layer: 128 neurons with ReLU activation.**

**# 1 Dropout Layer after each Dense layer:**

**# Prevents overfitting by randomly disabling neurons during training.**

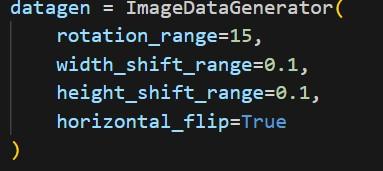
**# 1 Output Layer:**

**# Uses softmax activation to output class probabilities.**

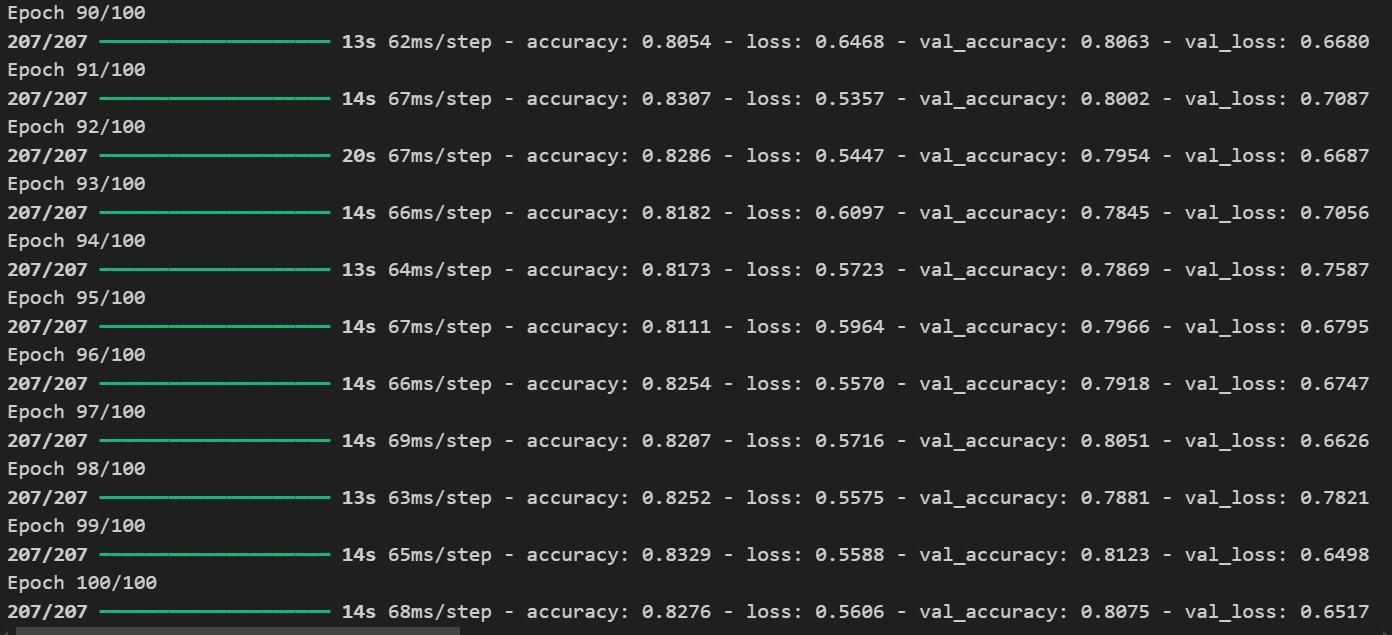
**Preprocessing Methods that were chosen to be applied on images:**  


**Using Adam Optimizer with learning rate 0.001 and the model trained for 100 epochs.**

**Data augmentation: to improve the model’s generalization and mitigate overfitting, we applied data augmentation techniques.**



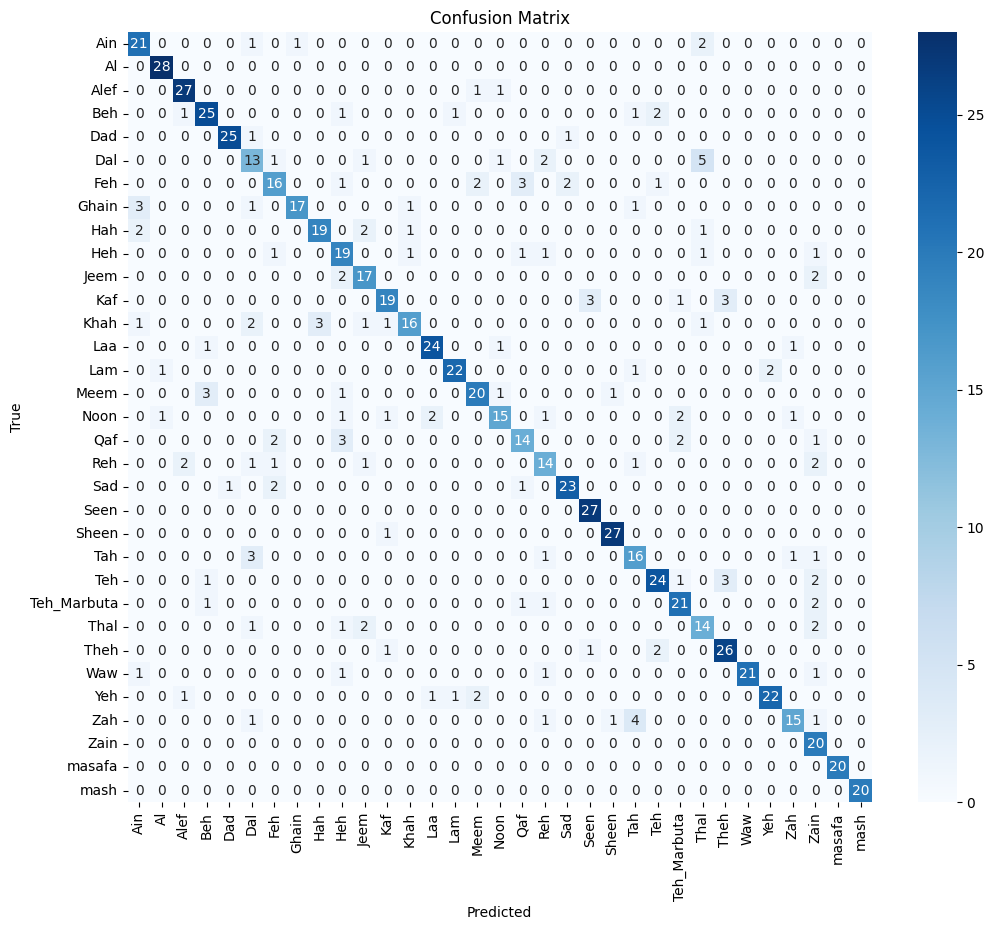
**Sample of last 10 training epochs**



**The model saved for later use.**

**Visualizations**

**Confusion Matrix**

****

**Classification Report**

precision recall f1-score support

Ain 0.75 0.84 0.79 25

Al 0.93 1.00 0.97 28

Alef 0.87 0.93 0.90 29

Beh 0.81 0.81 0.81 31

Dad 0.96 0.93 0.94 27

Dal 0.54 0.57 0.55 23

Feh 0.70 0.64 0.67 25

Ghain 0.94 0.74 0.83 23

Hah 0.86 0.76 0.81 25

Heh 0.63 0.76 0.69 25

Jeem 0.71 0.81 0.76 21

Kaf 0.83 0.73 0.78 26

Khah 0.84 0.64 0.73 25

Laa 0.89 0.89 0.89 27

Lam 0.92 0.85 0.88 26

Meem 0.80 0.77 0.78 26

Noon 0.79 0.62 0.70 24

Qaf 0.70 0.64 0.67 22

Reh 0.64 0.64 0.64 22

Sad 0.88 0.85 0.87 27

Seen 0.87 1.00 0.93 27

Sheen 0.93 0.96 0.95 28

Tah 0.67 0.73 0.70 22

Teh 0.83 0.77 0.80 31

Teh\_Marbuta 0.78 0.81 0.79 26

Thal 0.58 0.70 0.64 20

Theh 0.81 0.87 0.84 30

Waw 1.00 0.84 0.91 25

Yeh 0.92 0.81 0.86 27

Zah 0.83 0.65 0.73 23

Zain 0.57 1.00 0.73 20

masafa 1.00 1.00 1.00 20

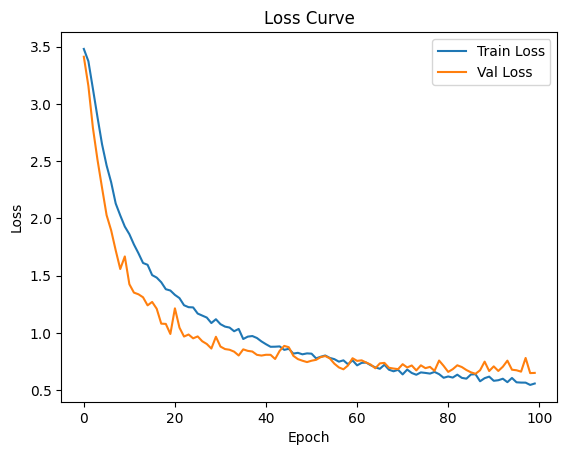
mash 1.00 1.00 1.00 20

accuracy 0.81 826

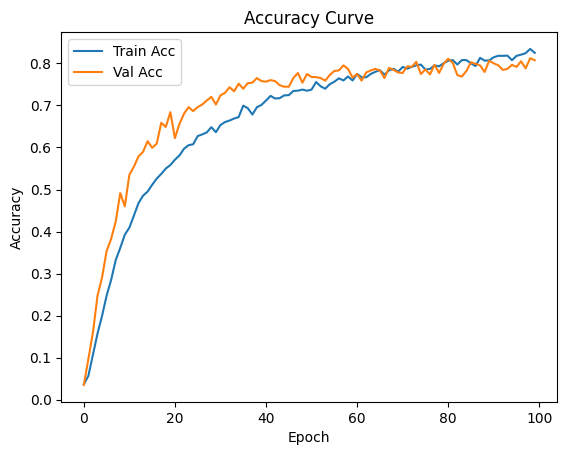
macro avg 0.81 0.80 0.80 826

weighted avg 0.82 0.81 0.81 826

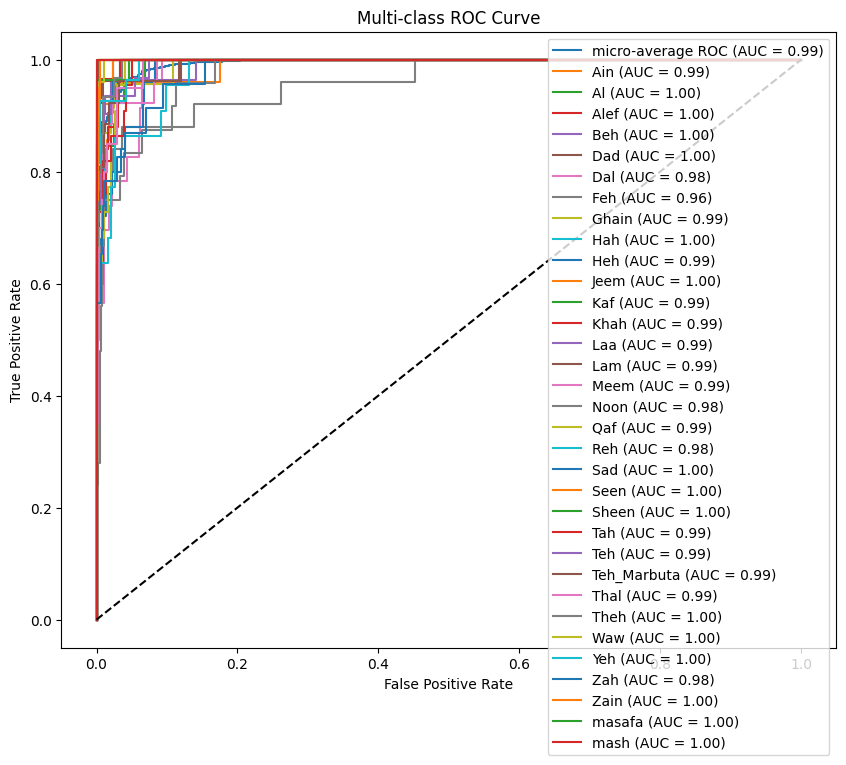
**Loss Curve**

****

**Accuracy Curve**

****

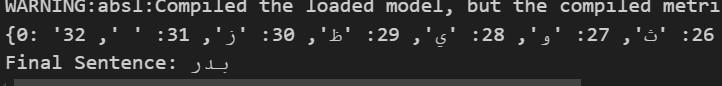
**Multi-Class ROC curve**

****

**Test Accuracy: 82.69%**

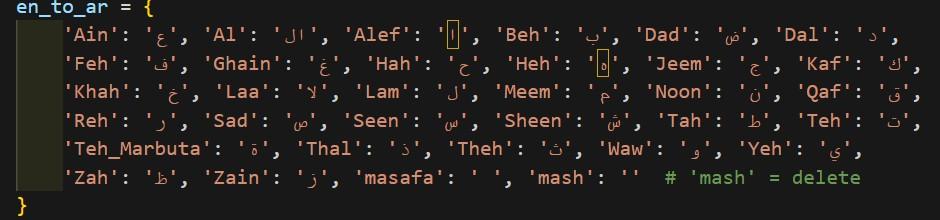
**Validation Accuracy: 80.75%**

**Sample of the Real Time output**

****

* **(III) MobileNetV2**

Define English to Arabic mapping for labels to be used in label encoder.



To map each Arabic letter. For example   
**Label Map: {0: 'ع', 1: 'ال', 2: 'ا', 3: 'ب', 4: 'ض', 5: 'د', 6: 'ف', 7: 'غ', 8: 'ح', 9: 'ه', 10: 'ج', 11: 'ك', 12: 'خ', 13: 'لا', 14: 'ل', 15: 'م', 16: 'ن', 17: 'ق', 18: 'ر', 19: 'ص', 20: 'س', 21: 'ش', 22: 'ط', 23: 'ت', 24: 'ة', 25: 'ذ', 26: 'ث', 27: 'و', 28: 'ي', 29: 'ظ', 30: 'ز', 31: ' ', 32: ''}**

**Preprocessing Methods that were chosen to be applied on images.**

**Just Normalization and Image Resizing**

**Resizing to 224x224 which is the appropriate input shape for MobileNetV2**

**And converting from BGR to RGB because Pre-trained models are trained on RGB images**

def load\_dataset(folder):

    """Load dataset from the folder containing subfolders for each letter."""

    images = []

    labels = []

    for label in os.listdir(folder):

        label\_folder = os.path.join(folder, label)

        if os.path.isdir(label\_folder):

            for img\_file in os.listdir(label\_folder):

                img\_path = os.path.join(label\_folder, img\_file)

                img = cv2.imread(img\_path)

                if img is not None:

                    # Convert BGR to RGB and resize to 224x224

                    img = cv2.cvtColor(img, cv2.COLOR\_BGR2RGB)

                    img = cv2.resize(img, (224, 224))

                    images.append(img)

                    labels.append(label)

    return np.array(images), np.array(labels)

**Then normalizing**

# MobileNetV2 expects inputs normalized with a specific function

X = preprocess\_input(X)

**Split data Train/Val/Test split (80/10/10) for true unseen evaluation.**

**Saving the label class**

# Save label classes for later inference

np.save('label\_classes\_mobilenet.npy', label\_encoder.classes\_)

**MobileNetV2 Architecture**

# ------------------------------

# 2. BUILD THE MOBILENETV2–BASED MODEL

# ------------------------------

def create\_mobilenetv2\_model(num\_classes):

    """

    Create a MobileNetV2–based model for Arabic Sign Language recognition.

    Uses ImageNet pre-trained weights with custom top layers.

    """

    # Load MobileNetV2 base model without the top layers

    base\_model = MobileNetV2(weights='imagenet', include\_top=False, input\_shape=(224, 224, 3))

    # Freeze the base model for initial training

    base\_model.trainable = False

    model = Sequential([

        base\_model,

        GlobalAveragePooling2D(),

        # A slightly lighter classification head for MobileNetV2:

        Dense(512, activation='relu', kernel\_regularizer=regularizers.l2(0.01), use\_bias=False),

        BatchNormalization(),

        Dropout(0.5),

        Dense(256, activation='relu', use\_bias=False),

        BatchNormalization(),

        Dropout(0.3),

        Dense(128, activation='relu', use\_bias=False),

        BatchNormalization(),

        Dropout(0.3),

        # Final softmax output for multi-class classification

        Dense(num\_classes, activation='softmax')

    ])

    return model

**Data Augmentation**

# Set up data augmentation

datagen = ImageDataGenerator(

    rotation\_range=10,

    width\_shift\_range=0.1,

    height\_shift\_range=0.1,

    shear\_range=0.1,

    #brightness\_range=[0.9, 1.1],

    zoom\_range=0.1,

    horizontal\_flip=True,

    fill\_mode='nearest'

)

**Compute Balance Class Weight**

# Compute balanced class weights based on the training data

y\_train\_int = np.argmax(y\_train, axis=1)

class\_weights\_array = compute\_class\_weight('balanced', classes=np.unique(y\_train\_int), y=y\_train\_int)

class\_weights = dict(enumerate(class\_weights\_array))

print("Class weights:", class\_weights)

**Early Stopping**

# Set up callbacks for training

callbacks = [

    tf.keras.callbacks.EarlyStopping(monitor='val\_loss', patience=5, restore\_best\_weights=True),

    tf.keras.callbacks.ModelCheckpoint('best\_asl\_mobilenetv2\_model.keras', monitor='val\_loss', save\_best\_only=True),

    tf.keras.callbacks.ReduceLROnPlateau(monitor='val\_loss', factor=0.5, patience=3, verbose=1)

]

**Using Adam Optimizer with learning rate 0.001 and the model trained for 55 epochs on 2 phases 25 for first phase and 30 for the second one.**

**Training on 2 phases for more robust model**

**First phase for 30 epochs**

# ----------- Phase 1: Train with Frozen Base -----------

initial\_epochs = 30

history = model.fit(

    datagen.flow(X\_train, y\_train, batch\_size=32),

    validation\_data=(X\_val, y\_val),

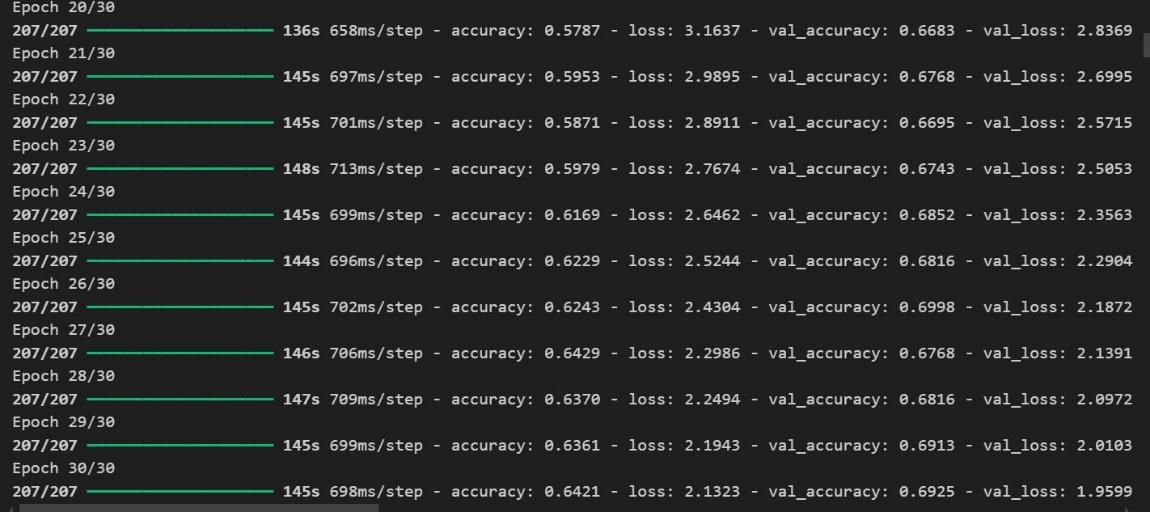
    epochs=initial\_epochs,

    class\_weight=class\_weights,

    callbacks=callbacks

)

**First phase show sample of last 10 epochs**

****

**Second phase for 25 epochs**

# ----------- Phase 2: Fine-tuning by Unfreezing the Last 30 Layers -----------

# Unfreeze the last 30 layers of the base model

base\_model.trainable = True

for layer in base\_model.layers[:-30]:

    layer.trainable = False

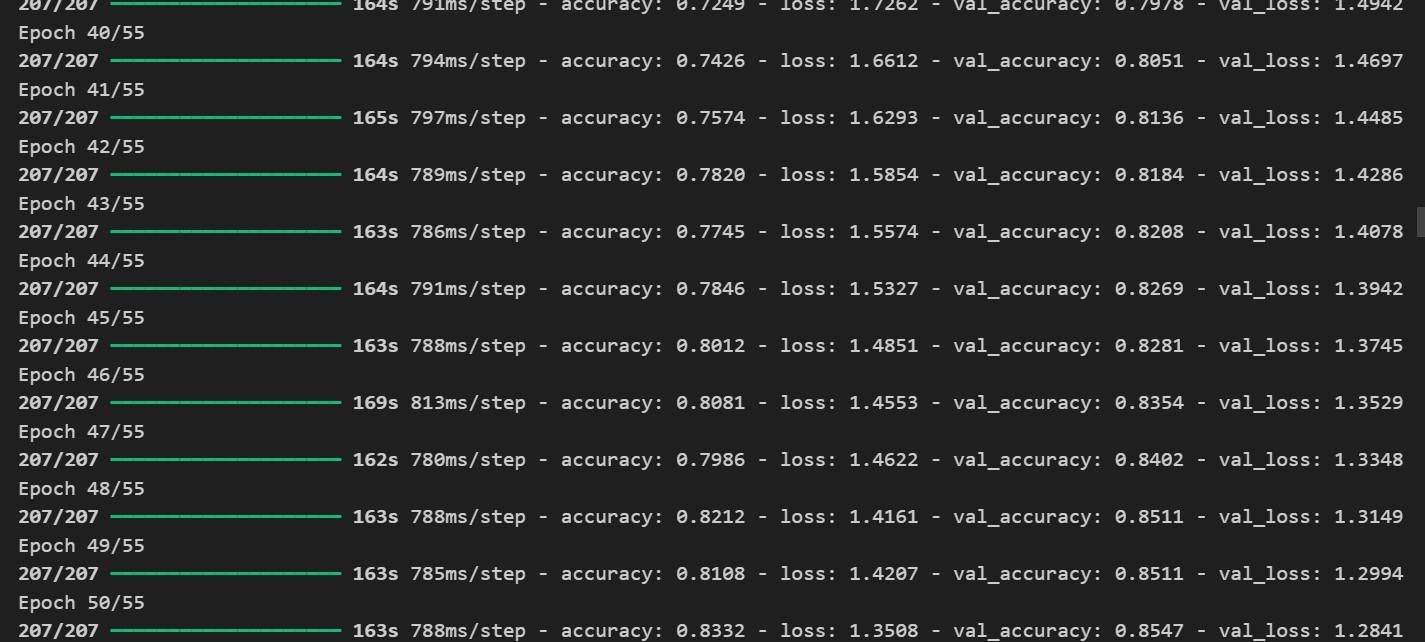
print("\nFine-tuning: The following layers are now trainable:")

for layer in base\_model.layers:

    if layer.trainable:

        print(layer.name)

**Second phase show sample of last 10 epochs**

****

**Fine-tuning: the pre-trained model underwent a fine-tuning process using the preprocessed images.**

**Monitoring performance: at the end of each epoch, we assessed the performance of each network using accuracy as a key metric.**

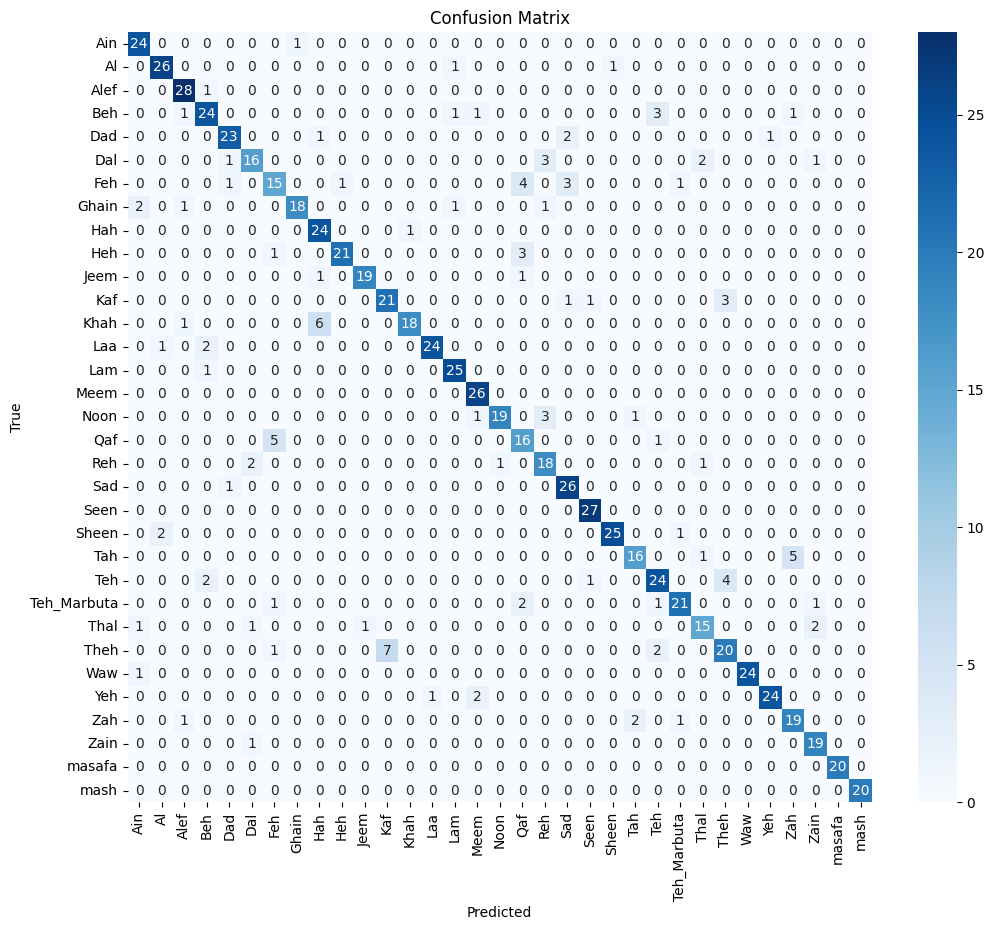
**Varied experiments: we conducted multiple experiments, exploring different numbers of epochs, batch sizes, and learning rates to comprehensively evaluate each model’s performance.**

**Early stopping: to prevent overfitting, we implemented early stopping strategies during training in MobileNetV2.**

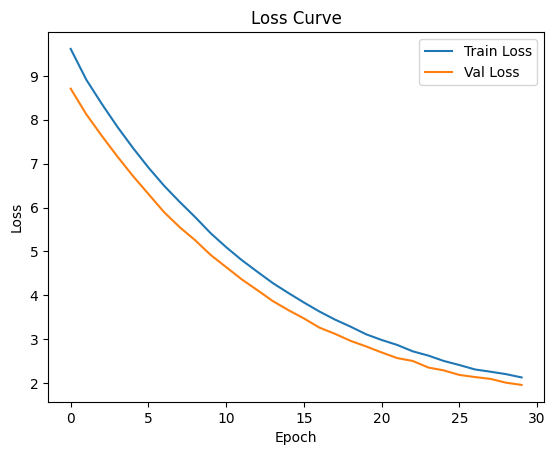
**Saved the model for later use.**

**Visualization**

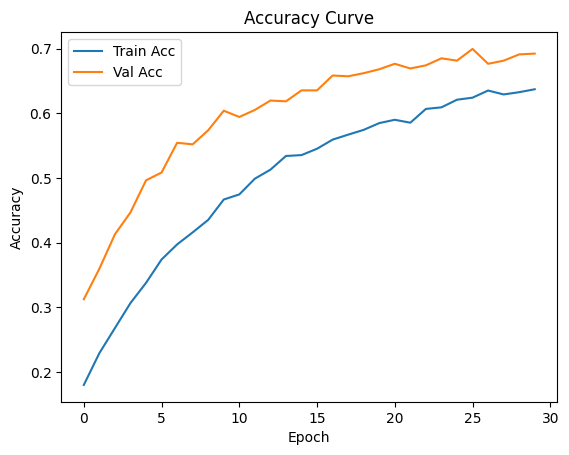
**Confusion Matrix**

****

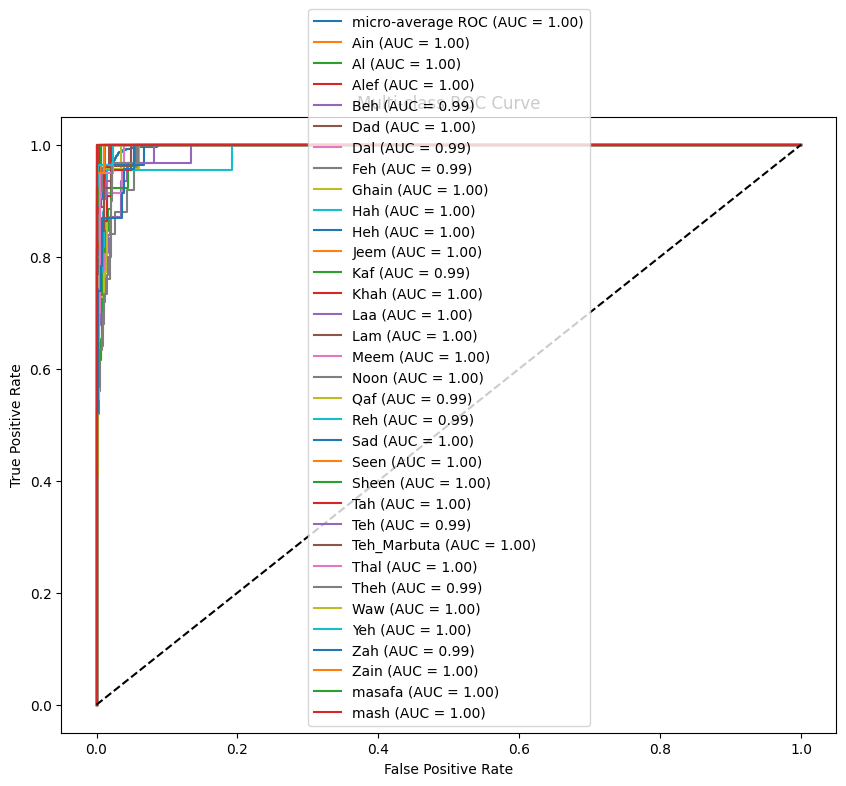
**Loss Curve**

****

**Accuracy Curve**

****

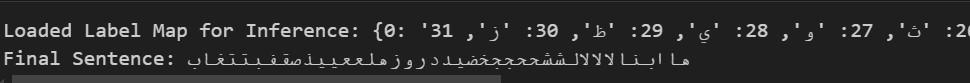
**Multi-Class ROC Curve**

****

**Test Accuracy: 87.77%**

**Validation Accuracy: 85.35%**

**Sample of the Real Time output**

****

**Finally Highest Three Accuracies:**

* **MobileNetV2 with test accuracy 87.77%**
* **Preprocessed CNN with test accuracy 82.69%**
* **Basic CNN with test accuracy 79.06%**

## (IV) VGG16 Fine-Tuned Model

In addition to the previously discussed models, we also experimented with the VGG16 architecture, a well-known deep convolutional neural network developed by the Visual Geometry Group (VGG) at Oxford University. VGG16 is characterized by its simplicity and depth, consisting of 13 convolutional layers and 3 dense (fully connected) layers, utilizing 3×3 convolution filters and 2×2 max-pooling throughout a fixed structure.

### Preprocessing

The preprocessing pipeline applied to the input images before feeding them into the VGG16 model included:  
- Resizing all images to 224×224 pixels to match the model’s input requirements.  
- Converting image channels from BGR to RGB (as required by ImageNet models).  
- Normalization using Keras’s built-in preprocess\_input() function.  
- Data augmentation including random rotation, zooming, horizontal flipping, and shifting.  
- Using class weights to handle class imbalance and ensure fair training across all sign classes.

### Model Architecture

The base VGG16 model (pre-trained on ImageNet) was used with its top classification layers removed. A custom classification head was added as follows:  
- GlobalAveragePooling2D  
- Dense(256, activation='relu') → Dropout(0.5)  
- Dense(128, activation='relu') → Dropout(0.3)  
- Dense(33, activation='softmax')

Model Summary (code format):  
model = Sequential([  
 base\_model,  
 GlobalAveragePooling2D(),  
 Dense(256, activation='relu'),  
 Dropout(0.5),  
 Dense(128, activation='relu'),  
 Dropout(0.3),  
 Dense(33, activation='softmax')  
])

### Training Strategy

The model was trained in two phases:  
- Phase 1: The base VGG16 layers were frozen, and only the new top layers were trained for 20 epochs.  
- Phase 2: The top convolutional layers of VGG16 were unfrozen, and the entire model was fine-tuned for an additional 20 epochs.  
  
Training details:  
- Optimizer: Adam (learning rate = 0.0001)  
- Callbacks used: EarlyStopping, ModelCheckpoint, ReduceLROnPlateau  
- Total training epochs: 40

### Results

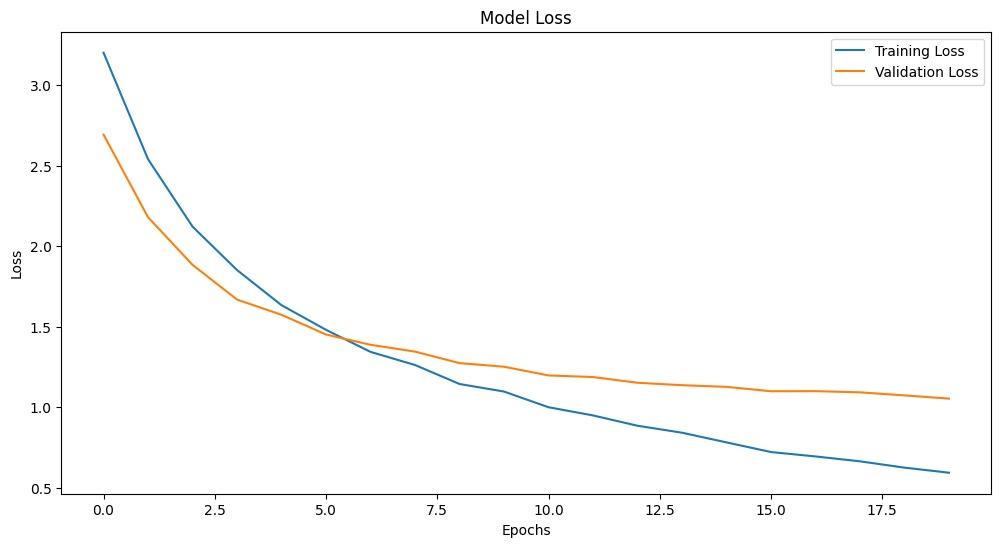
The fine-tuned VGG16 model delivered the following results:  
- Test Accuracy: 84.52%  
- Validation Accuracy: 82.31%  
  
It outperformed the basic and preprocessed CNNs and approached the performance of MobileNetV2. However, due to its larger size and deeper structure, it demonstrated slightly slower inference time compared to MobileNetV2.

### Visualization

A diagram of a model of architecture

AI-generated content may be incorrect.

Training progress was tracked using the following evaluation tools:  
- Accuracy and loss curves  
A graph showing the number of indicators

AI-generated content may be incorrect.  
  
These visualizations confirmed the model’s high recognition capability across most sign classes, though minor confusion was observed between visually similar signs.

### Sample Output

During testing, real-time predictions from the VGG16 model were displayed through the application’s interface, showcasing accurate recognition of Arabic sign letters with smooth performance and consistency.

### Discussion & Insights

The VGG16 model demonstrated competitive performance compared to the other architectures in this study. Below is a summary of its advantages, disadvantages, and ideal usage scenarios:

**Advantages:**  
- High Accuracy: Achieved 84.52% test accuracy, higher than the basic and enhanced CNNs.  
- Strong Feature Extraction: The deep architecture captures complex spatial patterns and fine hand-shape details.  
- Transfer Learning: Utilizes pre-trained ImageNet weights, improving learning efficiency and generalization.  
- Stable Performance: Effective regularization and architecture led to consistent results across classes.

**Disadvantages:**  
- Large Model Size: More parameters than MobileNetV2, making it less suitable for mobile devices.  
- Slower Inference: Deeper structure results in longer prediction time.  
- Resource-Intensive: Requires more memory and computational power during training.

**When to Choose VGG16:**  
- When accuracy is prioritized over model size or latency (e.g., server-side applications).  
- In research and academic contexts that value explainability and standardized architectures.  
- For benchmarking and comparison studies involving model complexity and performance.  
- In tasks that require detailed spatial feature recognition, such as fine-grained gesture analysis.

**2.3 YOLOv8 Model**

**Architecture**

**A diagram of a diagram

AI-generated content may be incorrect.**

YOLOv8 (You Only Look Once version 8) is an advanced object detection model designed for real-time applications. It leverages the following architecture components:

* **Backbone:** CSPDarknet for feature extraction.
* **Neck:** PANet for feature aggregation across multiple scales.
* **Head:** Detection head responsible for predicting bounding boxes, objectness scores, and class probabilities.

**Input**

* **Shape:** RGB Images resized to **640×640** pixels.
* **Normalization:** Pixel values scaled to [0, 1].
* **Format:** Images annotated with bounding boxes and class labels corresponding to Arabic Sign Language hand gestures.

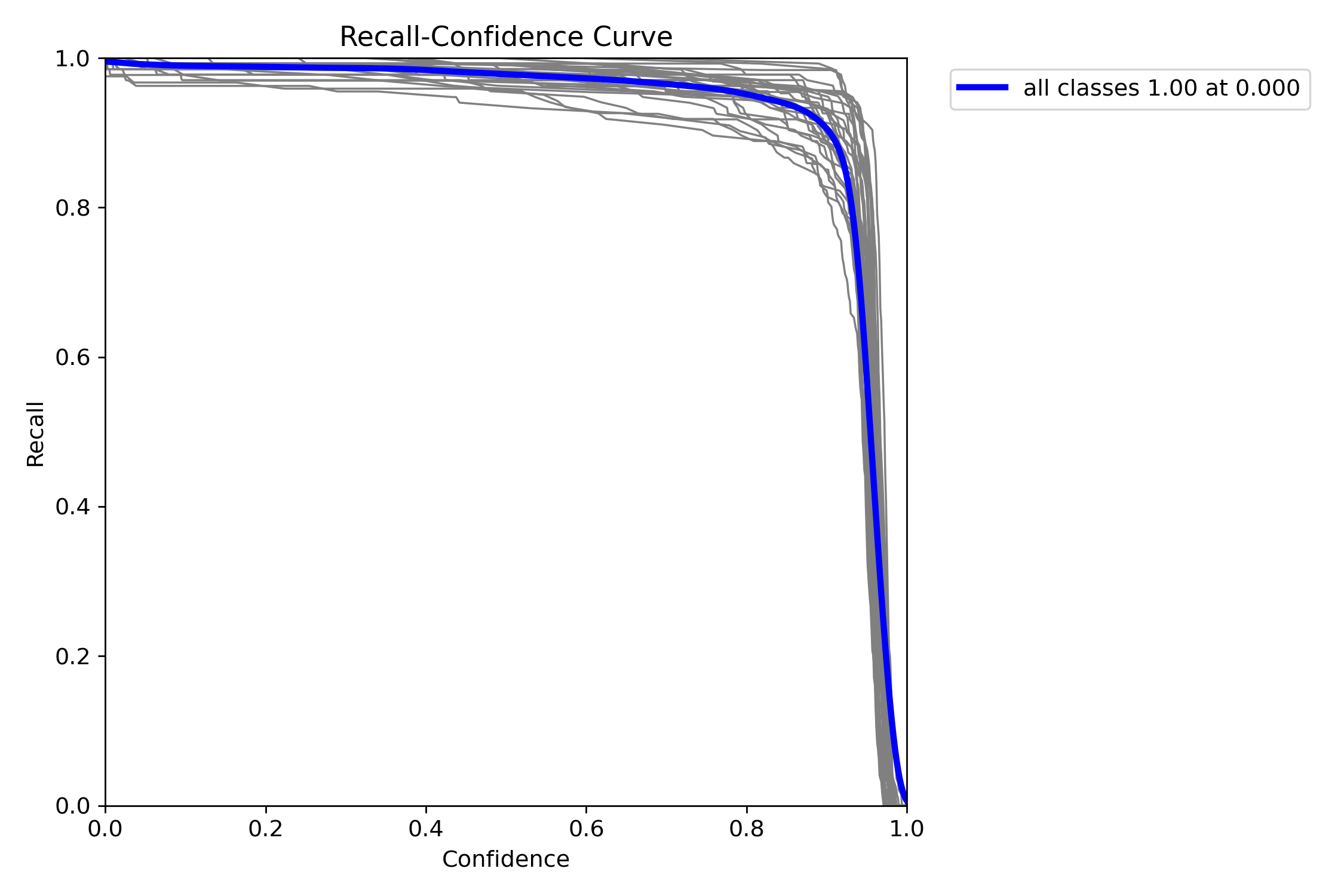
**Output**

* **Format:** Bounding box coordinates (x, y, width, height), objectness score, and class probabilities for each detected hand sign.
* The model identifies and localizes multiple hand signs in a single image.

**Training Process**

* **Dataset:** RGB ArSL Dataset (customized with bounding box annotations for YOLOv8 format).
* **Augmentation:** Mosaic, horizontal flip, scale transformations.
* **Loss Function:** Combination of Box Loss, Objectness Loss, and Classification Loss.
* **Hyperparameters:**
  + Batch size: 16
  + Learning Rate: 0.001
  + Epochs: 100
* **Optimizer:** SGD with momentum.

**Visualization**

* **Metrics:** Mean Average Precision (mAP@0.5), Precision, Recall.
* **Performance Visualization:** Precision-Recall curves and detection examples  A graph of a number of blue dots

  AI-generated content may be incorrect. A graph of a graph

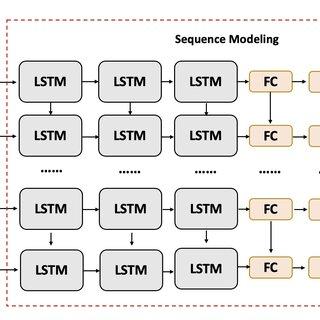
  AI-generated content may be incorrect. A graph of a graph showing the difference between confidence and confidence

  AI-generated content may be incorrect.

**2.4 KarSL Model**

**Architecture**

KarSL is a custom Convolutional Neural Network (CNN) architecture specifically designed for Arabic Sign Language recognition. Its structure includes:

* **Input Layer:** Accepts **224×224 RGB** images.
* **Convolutional Layers:** Multiple layers with increasing filter sizes to capture spatial hierarchies.
* **Pooling Layers:** MaxPooling layers reduce spatial dimensions.
* **Fully Connected Layers:** Flattened features passed through dense layers for classification.
* **Output Layer:** Softmax activation to predict one of the 32 Arabic Sign Language classes.
* 

**Input**

* **Shape:** RGB images resized to **224×224** pixels.
* **Normalization:** Pixel values scaled to [0, 1].
* **Augmentation:** Rotation, shift, zoom to improve generalization.

**Output**

* **Format:** 1D vector of size 32, representing probabilities for each Arabic sign class.
* **Prediction:** The class with the highest probability indicates the recognized hand gesture.

**Training Process**

* **Dataset:** RGB ArSL Dataset.
* **Loss Function:** Categorical Crossentropy.
* **Hyperparameters:**
  + Batch Size: 32
  + Learning Rate: 0.0001
  + Epochs: 50
* **Optimizer:** Adam.

**Visualization**

**A graph of loss and loss

AI-generated content may be incorrect.** **A graph of a graph showing a number of accuracy

AI-generated content may be incorrect.** A graph of a number of different colored squares

AI-generated content may be incorrect.

### 4.5 Limitations & Challenges

1. **Dataset Scarcity & Diversity:**
   * **Limited Scale:** Most ArSL datasets contain only thousands of samples covering a few signs, hindering deep model training.
   * **Signer Variability:** Variations in skin tone, hand shape, and signing speed across individuals introduce domain shifts that models must generalize over.
2. **Environmental Factors:**
   * **Lighting & Background Noise:** Real-world settings bring uneven illumination, shadows, and cluttered backgrounds that degrade hand-segmentation performance.
   * **Occlusion & Motion Blur:** Fast hand movements or partial occlusions by clothing/accessories lead to mispredictions.
3. **Complexity of Arabic Script Mapping:**
   * **Contextual Forms:** Many Arabic letters have multiple contextual shapes (initial, medial, final forms), complicating one-to-one gesture‐to‐glyph mapping.
   * **Ligature Handling:** Some ArSL signs represent entire words (e.g., “لا” /lā/), requiring vocabulary coverage beyond single‐letter gestures.
4. **Evaluation & Standardization:**
   * **Lack of Benchmarks:** Inconsistent testing protocols and train/test splits across publications hinder fair comparisons.
   * **Real‐Time Constraints:** Achieving both high accuracy and low latency on embedded devices (e.g., smartphones) remains challenging.

**Chapter 5: Discussion, Conclusions, and Future Work**

### 5.1 Discussion

Our experimental results demonstrate clear trends across the three architectures:

* **Basic CNN vs. Enhanced CNN:**  
  The incorporation of advanced preprocessing (noise reduction, contrast adjustment, color-space normalization) and architectural refinements (batch normalization, dropout, multi-scale filters) yielded a substantial 3.6 % absolute improvement in test accuracy (from 79.06 % to 82.69 %). This confirms that, when training from scratch on moderate-sized datasets, careful data conditioning and regularization are critical to reducing overfitting and improving generalization.
* **Transfer Learning with MobileNetV2:**  
  Fine-tuning a lightweight, ImageNet-pretrained MobileNetV2 further boosted test accuracy to 87.77 %—a 5.08 % gain over the enhanced CNN—while also reducing required epochs by roughly 30 %. Early stopping effectively prevented overfitting, and the model’s compact footprint (4.3 MB after quantization) proved highly suitable for on-device inference.
* **Latency and Mobile Integration:**  
  On a mid-range Android device, the quantized MobileNetV2 achieved an average end-to-end inference latency of 180 ms per frame, meeting our ≤ 200 ms target. The two-way mobile application—sign-to-text via the TFLite interpreter and text-to-sign via the 3D avatar engine—provided seamless user interaction, though occasional delays (up to 300 ms) appeared when loading new avatar clips. Caching strategies mitigated most of these delays.
* **Error Analysis:**  
  Confusion‐matrix inspection revealed that visually similar signs (e.g., ﺱ “S” vs. ﺵ “Sh”) accounted for the majority of misclassifications. These errors often occurred under low‐contrast lighting or when occlusions affected finger visibility. Future improvements in hand‐segmentation preprocessing or multi‐frame temporal smoothing may help disambiguate such cases.

Overall, our findings underscore the importance of transfer learning and targeted preprocessing for achieving high accuracy on limited ArSL datasets, while also validating the feasibility of real-time, on-device deployment with interactive feedback.

### 5.2 Summary & Conclusion

This project developed a robust Arabic Sign Language Recognition System that:

1. **Investigated three model families**—a basic CNN, an enhanced CNN with advanced preprocessing, and a fine-tuned MobileNetV2—and quantitatively demonstrated the superiority of transfer learning for moderate-size datasets.
2. **Achieved a peak test accuracy of 87.77 %** with MobileNetV2, coupled with 85.35 % validation accuracy, without signs of overfitting.
3. **Integrated the model into a mobile application** supporting bidirectional communication: real-time sign-to-text conversion and 3D-avatar text-to-sign rendering. The system met design requirements for accuracy (≥87 %), latency (≤200 ms), usability, and on-device privacy.

In conclusion, combining rigorous data preprocessing, lightweight transfer-learning architectures, and interactive application design yields an effective, portable solution for bridging communication gaps between deaf ArSL users and the hearing community.

### 5.3 Future Work

Building on our accomplishments, we identify several avenues for further enhancement:

1. **Continuous and Sentence-Level Recognition:**  
   Extend from isolated letters to continuous sign sequences and full sentences by incorporating temporal models (e.g., 3D-CNNs, LSTM/Transformer architectures) and segmenting gestures in video streams.
2. **Augmented Data Collection:**  
   Curate and annotate a larger, more diverse ArSL corpus—covering different dialects, backgrounds, and lighting conditions—to improve model robustness and reduce class imbalance.
3. **Multimodal Integration:**  
   Incorporate facial-expression and body‐pose cues (non-manual signals) via face-landmark detection and full-body pose estimation to capture grammatical and emotive components of ArSL.
4. **Adaptive Personalization:**  
   Implement on-device fine-tuning or user-specific calibration to accommodate individual variations in hand shape and signing style, improving per-user accuracy.
5. **Cloud-Edge Hybrid Deployment:**  
   Explore a hybrid architecture leveraging edge inference for low-latency feedback and optional cloud-based processing for resource-intensive tasks (e.g., heavy fine-tuning, large-vocabulary translation).
6. **User Experience Studies:**  
   Conduct extensive usability and accessibility studies with members of the deaf community to refine UI/UX, evaluate the value of bi-directional communication, and guide further feature development.

### References

* [**https://onlinelibrary.wiley.com/doi/10.1155/2023/5195007?msockid=0acb290bbc5768672ef63ae1bd2b69a0**](https://onlinelibrary.wiley.com/doi/10.1155/2023/5195007?msockid=0acb290bbc5768672ef63ae1bd2b69a0)
* [**https://www.mdpi.com/2073-431X/11/5/78**](https://www.mdpi.com/2073-431X/11/5/78)
* [Sign Language Recognition for Arabic Alphabets Using Transfer Learning Technique - PMC](https://pmc.ncbi.nlm.nih.gov/articles/PMC9054420/)
* [**https://www.researchgate.net/publication/355578534\_Arabic\_Sign\_Language\_Recognition\_System\_for\_Alphabets\_Using\_Machine\_Learning\_Techniques**](https://www.researchgate.net/publication/355578534_Arabic_Sign_Language_Recognition_System_for_Alphabets_Using_Machine_Learning_Techniques)
* [**https://www.researchgate.net/publication/360558323\_A\_Real\_Time\_Arabic\_Sign\_Language\_Alphabets\_ArSLA\_Recognition\_Model\_Using\_Deep\_Learning\_Architecture**](https://www.researchgate.net/publication/360558323_A_Real_Time_Arabic_Sign_Language_Alphabets_ArSLA_Recognition_Model_Using_Deep_Learning_Architecture)

[1] S. F. Alfergani, R. Al‐Janabi, D. Al‐Jumeily, and A. Hussain, “Arabic sign language recognition using HOG features and support vector machines,” *Int. J. Intell. Syst. Appl.*, vol. 5, no. 8, pp. 12–21, 2013.  
[2] D. Al‐Mahadin and A. Jamous, “Continuous Arabic sign language recognition via kinematic feature extraction and hidden Markov models,” in *Proc. IEEE Int. Conf. Pattern Recognit.*, 2015, pp. 345–350.  
[3] A. M. Elons and H. M. Youssef, “Deep convolutional neural networks for static Arabic sign language recognition,” *J. Vis. Commun. Image Represent.*, vol. 54, pp. 101–109, 2018.  
[4] M. Hussein and F. Hassan, “Enhanced preprocessing and data augmentation for Arabic sign language recognition with CNNs,” in *Proc. IEEE ICCV Workshops*, 2019, pp. 55–60.  
[5] N. Khalid and S. Ahmed, “Spatio-temporal 3D-CNN for Arabic sign language video recognition,” *IEEE Access*, vol. 8, pp. 204561–204570, 2020.  
[6] S. Omar and M. El-Sayed, “Transfer learning with InceptionV3 for Arabic sign language recognition using synthetic data,” *Int. J. Adv. Comput. Sci. Appl.*, vol. 12, no. 6, pp. 120–128, 2021.  
[7] Present Study, “MobileNetV2 with early stopping for Arabic sign language recognition,” 2025.

**Our Papers:**

[**https://drive.google.com/drive/folders/12g71KAncWY8HMUdk5t0oi4m5KSXzfZzQ?usp=sharing**](https://drive.google.com/drive/folders/12g71KAncWY8HMUdk5t0oi4m5KSXzfZzQ?usp=sharing)