Real-Time AI-Powered Bidirectional Communication System for Translating Sign Language to Text/Speech and Vice Versa

by

M Al Shahriar Hameem (222014094)
Ahsun Ahmed Sun (222014005)
Md Fahad Al Noman (222014060)
Md Nazmul Haque (222014037)
Md Shakhawat Hossen (222014023)

Capstone project report (CSE 4098C) submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science in Computer Science and Engineering

Under the supervision of

Nasir Uddin Ahmed



DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

UNIVERSITY OF LIBERAL ARTS BANGLADESH

SPRING 2026

© *M Al Shahriar Hameem*, *Ahsun Ahmed Sun*, *Md Fahad Al Noman*, *Md Nazmul Haque*, *Md Shakhawat Hossen*
All rights reserved

DECLARATION

Project Title	Real-Time AI-Powered Bidirectional Communication System for Translating Sign Language to Text/Speech and Vice Versa
Author	*M Al Shahriar Hameem, Ahsun Ahmed Sun, Md Fahad Al Noman, Md Nazmul Haque, Md Shakhawat Hossen*
Student ID	222014094, 222014005, 222014060, 222014037, 222014023
Supervisor	Nasir Uddin Ahmed

I declare that this capstone project report entitled *Real-Time AI-Powered Bidirectional Communication System for Translating Sign Language to Text/Speech and Vice Versa* is the result of my own work except as cited in the references. The capstone project report has not been accepted for any degree and is not concurrently submitted in candidature of any other degree.

M Al Shahriar Hameem
222014094

Department of Computer Science and Engineering
University of Liberal Arts Bangladesh

Ahsun Ahmed Sun
222014005

Department of Computer Science and Engineering
University of Liberal Arts Bangladesh

Md Fahad Al Noman
222014060

Department of Computer Science and Engineering
University of Liberal Arts Bangladesh

Md Nazmul Haque
222014037

Department of Computer Science and Engineering
University of Liberal Arts Bangladesh

Md Shakhawat Hossen
222014023

Department of Computer Science and Engineering
University of Liberal Arts Bangladesh

Date: May 03, 2026

Department of Computer Science and Engineering
University of Liberal Arts Bangladesh
Mohammadpur, Dhaka - 1207

CERTIFICATE

This is to certify that the capstone project report entitled **Real-Time AI-Powered Bidirectional Communication System for Translating Sign Language to Text/Speech and Vice Versa**, submitted by **M Al Shahriar Hameem** (Student ID: 222014094), **Ahsun Ahmed Sun** (Student ID: 222014005) and **Md Fahad Al Noman** (Student ID: 222014060), **Md Nazmul Haque** (Student ID: 222014037), and **Md Shakhawat Hossen** (Student ID: 222014023) are undergraduate students of the **Department of Computer Science and Engineering** has been examined. Upon recommendation by the examination committee, we hereby accord our approval of it as the presented work and submitted report fulfill the requirements for its acceptance in partial fulfillment for the degree of *Bachelor of Science in Computer Science and Engineering*.

Nasir Uddin Ahmed
Lecturer

Department of Computer Science and Engineering
University of Liberal Arts Bangladesh

Prof. Muhammad Golam Kibria, PhD, SMIEEE
Professor Head

Department of Computer Science and Engineering
University of Liberal Arts Bangladesh

Shah Jafor Sadeek Quaderi
Lecturer

Department of Computer Science and Engineering
University of Liberal Arts Bangladesh

Md Kaviul Hossain
Lecturer

Department of Computer Science and Engineering
University of Liberal Arts Bangladesh

Place: Dhaka
Date: May 03, 2026

ACKNOWLEDGEMENTS

We would like to express our deep and sincere gratitude to our supervisor, *Nasir Uddin Ahmed*, for allowing us to conduct research and providing invaluable guidance throughout this work. His dynamism, vision, sincerity and motivation have deeply inspired us. He has taught us the methodology to carry out the work and to present the works as clearly as possible. It was a great privilege and honour to work and study under his guidance.

We are greatly indebted to our honourable teachers of the Department of Computer Science and Engineering at the University of Liberal Arts Bangladesh who taught us during our study. Without any doubt, their teaching and guidance have completely transformed us to the persons that we are today.

We are extremely thankful to our parents for their unconditional love, endless prayers and caring, and immense sacrifices for educating and preparing us for our future. We would like to say thanks to our friends and relatives for their kind support and care.

Finally, we would like to thank all the people who have supported us to complete the project work directly or indirectly.

M Al Shahriar Hameem, *Ahsun Ahmed Sun*, *Md Fahad Al Noman*, *Md Nazmul Haque*, *Md Shakhawat Hossen*
University of Liberal Arts Bangladesh
Date: May 03, 2026

Dedicated to
my late father, my beloved mother and my dear sisters.

– M Al Shahriar Hameem

Dedicated to
my beloved parents who inspired and helped me in every step of my life.

– Ahsun Ahmed Sun

Dedicated to
my parents, for their endless love, support and encouragement

– Md Fahad Al Noman

Dedicated to
my family. Their love, patience, and constant encouragement made me believe in myself.

– Md Nazmul Haque

Dedicated to
my family, who offered me a listening ear during my frustrations and celebrated every small breakthrough along the way

– Md Shakhawat Hossen

ABSTRACT

Deaf and hard-of-hearing individuals often face communication barriers because many existing sign-language tools are one-directional, hardware-dependent, and unsuitable for real-time use. This project presents a browser-based, real-time Bangla Sign Language translator that supports bidirectional communication between sign language and spoken or written Bangla. The sign-to-speech pipeline uses webcam input, MediaPipe Hand and Pose Landmarkers, landmark normalization, and a dense neural network trained on 60 BdSL word classes. The system applies sustained-sign confirmation and confidence filtering to improve recognition reliability. For the speech-to-sign direction, the application accepts typed Bangla text and browser speech input, normalizes and matches words or short phrases, and displays corresponding sign avatar images sequentially. The system is implemented as a client-side web application with mobile support, requiring no backend server or specialized hardware. Overall, the project aims to provide an accessible and practical assistive communication tool for Bangla-speaking users in education, healthcare, public services, and daily life.

Keywords: Bangla Sign Language, real-time translation, bidirectional communication, MediaPipe, dense neural network, accessibility, web application, Assistive technology.

Contents

1 Introduction **1**

1.1 Problem statement 2
1.2 Aim 2
1.3 Objectives 2
1.4 Motivation 3
1.5 Scope 4

2 Literature Review and Analysis **5**

2.1 Literature Review 5
2.2 Literature Analysis 7
2.3 Literature Review Summary Table 8

3 Plans and Deliverables **11**

3.1 Timeline 11
3.2 Project Specification 13
3.2.1 Functional Requirements 13
3.2.2 Non-Functional Requirements 13
3.3 Expected Final Product Description 14
3.4 Datasets and Data Strategy 15
3.5 Tools and Development Stack 15
3.6 Proposed Plan & Budget 15

4 Methodology **17**

4.1 Implemented Methodology 17
4.2 Dataset Collection and Selection 17
4.2.1 Dataset Descriptions 17
4.2.2 Data Selection Strategy and Challenges 20

4.2.3 Dataset Sourcing . . . 20
4.3 Data Preprocessing . . . 20
4.3.1 Image Reading and Format Conversion . . . 20
4.3.2 Hand Landmark Extraction . . . 20
4.3.3 Hand Sorting by Handedness . . . 21
4.3.4 Pose Landmark Extraction . . . 21
4.3.5 Reference Point, Scale, and Normalization . . . 21
4.3.6 Batch Processing and Split Statistics . . . 22
4.3.7 Train/Validation/Test Split . . . 22
4.3.8 Augmentation (Training Set Only) . . . 22
4.3.9 Standardization . . . 22
4.4 Model Selection and Architecture . . . 22
4.4.1 Version 1 (Hand Keypoints Only) . . . 23
4.4.2 Version 2 (Hybrid Hand + Pose Features) . . . 23
4.4.3 Version 3 (Production-Quality Pipeline) . . . 23
4.4.4 Final Architecture: Stacked `Dense` Layers with Regularization . . . 23
4.4.5 Training Strategy . . . 24
4.4.6 Evaluation . . . 25
4.4.7 Comparative Summary . . . 25
4.5 Training Environment Setup . . . 26
4.5.1 Infrastructure: Kaggle Notebooks . . . 26
4.5.2 Software Stack and Libraries . . . 26
4.5.3 Mixed Precision Configuration . . . 26
4.5.4 Memory Management and Determinism . . . 27
4.5.5 Artifacts Written to `/kaggle/working/` . . . 27
4.5.6 Installation and Configuration . . . 27

5 Design and Development **29**
5.1 Scope of This Chapter . . . 29
5.2 Project Architecture . . . 29
5.2.1 Bidirectional Design Rationale . . . 29
5.2.2 Module Structure . . . 30
5.2.3 The Gesture Recognition Pipeline (Sign $\rightarrow$ Speech) . . . 31
5.2.4 Sustained Sign Confirmation . . . 31

5.2.5 The Speech-to-Sign Translation Pipeline . . . 32
5.2.6 Avatar Creation and Static Pose Generation Using Blender . . . 32
5.2.6.1 Overview of the Avatar Pose Pipeline . . . 33
5.2.6.2 Software and Resources . . . 33
5.2.6.3 Model Acquisition and Armature Setup . . . 34
5.2.6.4 Scene Preparation and Neutral Posture . . . 34
5.2.6.5 Pose Library Setup and Sign-to-Pose Mapping . . . 34
5.2.6.6 Manual Pose Creation and Reference Matching . . . 36
5.2.6.7 Rendering and Output . . . 36
5.3 Web Deployment and Runtime . . . 38
5.3.1 Client-Side Architecture . . . 38
5.3.2 GPU Acceleration . . . 38
5.3.3 Camera and Layout Controls . . . 38
5.3.4 Speech Input . . . 39
5.3.5 Running the Application . . . 39
5.4 Mobile Application and APK Generation . . . 39
5.4.1 PWA Infrastructure . . . 39
5.4.2 APK Packaging via PWABuilder . . . 40
5.5 Application Demonstration . . . 40
5.6 Future Extensions . . . 40

6 Result and Evaluation 43
6.1 Result Analysis and Discussion . . . 43
6.1.1 Evaluation of the Result . . . 43
6.1.2 Training Curve Analysis . . . 46
6.1.2.1 MobileNetV2 (transfer learning) . . . 46
6.1.2.2 ResNet50 (deep transfer learning) . . . 46
6.1.2.3 DenseNN (keypoint-based MLP) . . . 49
6.1.2.4 YOLOv11-Pose + MediaPipe (hybrid) . . . 50
6.2 Expected Result . . . 50

7 Conclusion **52**
7.1 Social, Legal, and Ethical Issues 52
7.1.1 Social Impact and Inclusivity 52
7.1.2 Design for Safety and Hazard Analysis 52
7.1.3 Ethical Considerations 52
7.1.4 Legal and Regulatory Compliance 53
7.2 Environment and Sustainability Impact 53
7.2.1 Environmental Impact 53
7.2.2 Energy Efficiency and Resource Utilization 53
7.2.3 Sustainability and Social Development Goals (SDGs) 53
7.3 Limitations 54
7.3.1 Challenges 54
7.3.2 Limitations 55

References **57**

List of Figures

3.1 Gantt chart of CSE 4098–C timeline. 11
3.2 Gantt chart of overall project timeline. 12

4.1 Application Level Methodology 18
4.2 Machine Learning Pipeline. 19
4.3 Detailed Dense Neural Network architecture for hand keypoint classification, illustrating the sequential dense blocks, regularization layers, and exact parameter composition. 24

5.1 Avatar static pose generation and rendering pipeline. 33
5.2 The pre-rigged Snow character model and predefined armature in Blender's viewport. 34
5.3 Avatar established in the baseline neutral pose prior to gesture mapping. . . 35
5.4 Organization of facial expressions and hand states within the Blender Pose Library. 35
5.5 Comparison of human reference and finalized avatar pose for the BdSL sign "Office". 36
5.6 Comparison of human reference and finalized avatar pose for the BdSL sign "Wait". 37
5.7 Web Application Interface. 40
5.8 Mobile App Home Screen. 41
5.9 Settings view. 41

6.1 Model size on disk. 44
6.2 Trainable parameter count. 45
6.3 Qualitative sample predictions showing DenseNN consistently produces high-confidence correct predictions across augmentations, while CNNs vary widely. 47

6.4 MobileNetV2 training curves. 48
6.5 ResNet50 training curves. 48
6.6 DenseNN training curves. 49
6.7 DenseNN learning curves with generalization gap visualization. 49
6.8 YOLOv11-Pose hybrid training curves (baseline vs. regularized). 50

List of Tables

2.3 Literature Review Summary Table 8

3.1 Project Cost Breakdown (Table 3.1) 16

4.1 Evolution of the landmark-based classification pipeline 26

5.1 Source files and their roles in the Combined Translator. 30

6.1 Headline metrics across the four evaluated models. 43

6.2 Top-1 accuracy under different augmentation conditions 46

Chapter 1

Introduction

In an increasingly connected world, millions of deaf and hard-of-hearing individuals still face major barriers to effective communication. Despite advances in artificial intelligence and language technologies, most existing sign language systems remain one-directional, hardware-dependent, and poorly optimized for natural, real-time interaction. These limitations reduce their effectiveness in practical settings such as classrooms, hospitals, and public services, particularly for users in low-resource environments and those communicating in regional languages like Bengali.

Recent research efforts have demonstrated the potential of AI-driven approaches in enhancing sign language communication. For example, systems developed by Islam et al. (2022) and Rajkumar et al. (2023) showcased early prototypes for translating Bangla and English text/voice into animated sign language, with promising results in terms of accuracy and response time. However, most of these solutions lack support for non-manual signs, facial expressions, and dynamic environments, significantly limiting their naturalness and inclusivity. Additionally, studies like Aslam et al. (2025) and Islam et al. (2022) highlight the importance of real-time gesture recognition, avatar realism, and multilingual support, yet struggle with contextual understanding, vocabulary limitations, and performance under varying lighting or movement conditions.

This project aims to bridge these gaps by building an inclusive, AI-powered, bidirectional communication system that can translate sign language to text/speech and vice versa, in real time. Unlike prior systems, it will support both English and Bengali languages, focus on natural interaction through 3D avatars and computer vision, and run efficiently on mobile and web platforms. The goal is to enable seamless, two-way communication and empower the deaf community with an accessible, scalable, and context-aware tool tailored to real-world needs.

1.1 Problem statement

Current sign language communication systems are often one-directional, hardware-dependent, and lack real-time responsiveness, making them impractical for everyday use—especially in low-resource settings. This creates significant communication barriers for deaf and hard-of-hearing individuals, limiting their access to education, services, and social interaction.

Previous studies Islam et al. (2022);Rajkumar et al. (2023) have developed basic Bangla and English sign translation systems using video libraries and rule-based models, but these lack scalability and natural expression. More recent works Aslam et al. (2025);Islam et al. (2022) introduced 3D avatars and gesture recognition but still face challenges with vocabulary limitations, facial expressions, and real-world performance.

To address these gaps, this project aims to develop an AI-powered, real-time, bidirectional communication system for translating between sign language and speech/text in Bengali and English. The system will focus on accessibility, dynamic interaction, and mobile/web compatibility to ensure practical use across diverse environments.

1.2 Aim

The aim of this project is to design and develop a real-time, AI-powered bidirectional communication system that translates between sign language and speech/text in both Bengali and English. The system will support natural interaction using computer vision, speech processing, and 3D avatar-based animation to ensure inclusivity for deaf and hard-of-hearing individuals. Drawing on advancements from previous works such as Islam et al. (2022), Rajkumar et al. (2023), and Aslam et al. (2025), the project will focus on overcoming current limitations by expanding vocabulary, incorporating non-manual signs and facial expressions, and enabling contextual understanding in dynamic environments. The goal is to create an accessible, language-inclusive, and user-friendly platform for seamless communication across diverse real-world scenarios.

1.3 Objectives

- Build a bidirectional pipeline (sign→Text/Speech and Text/Speech→sign) supporting Bengali and English.

- Implement deep-learning computer-vision models to recognize manual and non-manual sign features from standard cameras.
- Develop an expressive 3D avatar module for clear, natural sign rendering in common conversational scenarios.
- Optimize for accessibility and deployment on mobile and web (low latency, no specialized hardware).
- Incorporate regional-language support (Bangla) and usability for low-resource environments.
- Evaluate with concrete metrics (accuracy, latency, vocabulary coverage, user satisfaction) and report results each academic term.

1.4 Motivation

Despite advancements in sign language recognition and synthesis, most current systems remain limited in scope—supporting only one-way translation, requiring specialized hardware, or lacking real-time responsiveness. This results in fragmented communication and reinforces social, educational, and professional barriers for deaf and hard-of-hearing individuals. Studies such as Islam et al. (2022) and Rajkumar et al. (2023) highlight early progress in Bangla and English sign translation, yet face critical limitations in vocabulary size, grammar handling, and non-manual gesture integration. Similarly,Aslam et al. (2025) and Islam et al. (2022) emphasize the challenges of achieving realism, dynamic context-awareness, and usability in real-world conditions.

Moreover, existing tools often neglect users in linguistically diverse and low-resource settings, where mobile-first and accessible design is crucial. Given the lack of inclusive, bidirectional, and culturally localized communication tools, there is a strong need for a system that bridges the gap between spoken and sign languages in both directions—supporting natural, real-time, and expressive interaction.

This project is motivated by the vision of creating a system that enables equitable communication across hearing and non-hearing communities, particularly for Bengali speakers. By integrating AI, deep learning, and language support into a user-friendly, real-time platform, the project seeks to foster inclusion, autonomy, and accessibility in everyday interactions.

1.5 Scope

The system is designed primarily for deaf and hard-of-hearing users as an assistive communication tool, while also supporting hearing participants in the same conversation. It enables two-way communication by translating spoken language into sign language (via an avatar) and translating signed input back into text or speech.

In educational settings—classrooms, lectures, tutoring, or remote learning—the system helps make instruction accessible in real time. Teachers can speak in English or Bengali while deaf students follow through the signing avatar, and students can ask questions in sign language that are translated for the teacher, enabling full participation and more inclusive learning environments.

In healthcare and public services (clinics, counseling, banks, government offices, legal consultations), the system functions as a virtual interpreter when human interpreters are unavailable. It supports timely, accurate exchanges between deaf individuals and service providers, reducing misunderstandings while increasing independence and privacy.

For everyday personal communication—workplace conversations, social interactions, and daily transactions—the system can run on a mobile device as a portable translator. Deaf users can sign through the camera while others read or hear the translation, and hearing people can speak or type while the deaf user watches the avatar, making communication faster, more natural, and more widely accessible in real-world situations.

Across all use cases, the focus is on practical reliability in real environments, including challenges like noise, lighting variation, and dynamic backgrounds. By targeting education, healthcare, and daily life, the system addresses high-impact contexts where accessibility directly affects inclusion, safety, and quality of life.

Chapter 2

Literature Review and Analysis

This chapter consists of two sections: a literature review and a literature analysis. The literature review addresses prior research and the design of solutions within the relevant field. In the literature analysis, we identify and elucidate the limitations and potential areas for improvement based on previous efforts.

2.1 Literature Review

Recent scholarship on sign language technologies spans isolated and continuous recognition, translation to/from speech and text, multilingual generation, and avatar-based rendering, with growing attention to Bengali/Bangla Sign Language (BdSL) and South Asian contexts. Within BdSL, benchmarking and model selection have been a central theme. A comprehensive assessment at ICCIT 2024 compared VGG16, ResNet50 and a custom CNN on 11,172 images across 38 BdSL gestures, reporting 92.33% accuracy and arguing for task-specific architectures over generic backbones (Tanni et al. (2024-12)). Complementing this, an edge-deployment study curated a 49-class BdSL dataset ($\approx$ 80 images/class), trained Detectron2, EfficientDet-D0, and YOLOv7, and demonstrated real-time inference on Jetson Nano with $\approx$ 30 FPS–achieving 97.95% overall accuracy and 98.8% precision (Siddique et al. (2023)). The same work details a practical two-part hardware–software pipeline for webcam capture, on-device inference, and on-screen feedback, underscoring deployability in local contexts (Siddique et al. (2023)).

Beyond single-model demonstrations, other BdSL efforts have investigated architecture families. An IEEE conference paper from 2024 (ICCCNT) evaluates ResNet, DenseNet, Inception, and Xception variants on a curated Bengali dataset ($\approx$ 10,182 images, 38 classes),

arguing that modern residual and recalibration mechanisms improve recognition without excessive depth (Rai et al. (2024)). In parallel, an Emerging Science Journal article focuses on word-level BdSL and introduces a small custom dataset of frequently used words, reporting real-time detection and a comparative table against prior Bengali and non-Bengali works–thereby shifting emphasis from alphabets/digits to functional vocabulary (Recognition of Bangladeshi Sign Language (BdSL) Words Using DCNNs, 2023).

Work on translation and multilingual generation has expanded rapidly. An ICAISS 2025 paper proposes Indian Sign Language (ISL) recognition with data augmentation and multilingual sign generation, bridging recognition and synthesis for inclusive interfaces (Aslam et al. (2025)).

At the speech boundary, several systems translate speech→sign and sign→

text/speech in real time. One study prototypes a real-time speech-to-sign converter using NLP and rule-based mapping to support the hearing-impaired (Deshmukh et al. (2024)). Another (IRJET, 2024) demonstrates a two-way sign↔audio conversion pipeline with OpenCV/-MediaPipe and text-to-speech, positioning recognition and synthesis in a single loop (Real-Time Sign Language and Audio Conversion Using AI, 2024). A complementary survey consolidates recognition+translation system patterns (datasets, model choices, and deployment considerations) for enhanced communication (Sindhu et al. (2024-03)).

Within BdSL and related South-Asian contexts, avatar and instructional interfaces have emerged. Building on tutoring metaphors, a smart tutor avatar explores character-style mimicking to assist learners of BdSL, pointing to pedagogically-oriented animation as a complement to recognition (A Smart Tutor Avatar for Mimicking Characters of Bangla Sign Language, n.d.).

Researchers have also advanced two-way interaction and pipeline engineering. A "comprehensive application" integrates alphabet/word recognition, text-to-action conversion, multilanguage support, and voice output, representing a product-style consolidation of many single-paper contributions (Comprehensive Application for Sign Language Alphabet and Word Recognition…, n.d.).

Regionally, translation spans beyond Bangladesh and India. A 2024 ICEET paper focuses on Sri Lankan Sign Language→Tamil vocal and ASL translation, emphasizing bilingual pipelines and practical deployment (Ratnasingam et al. (2024-12)). Alongside, hand-translation to text+speech systems and dynamic, AI-driven detection frameworks (e.g., SignaSpectrum) show continuing interest in responsive interfaces for communication and ac-

cessibility (Real-Time Hand Sign Language Translation: Text and Speech Conversion, n.d.; SignaSpectrum: AI-Driven Dynamic Sign Language Detection and Interpretation, n.d.).

Finally, transformer-based translation is entering the space. Sign2Text prototypes a Vision Transformer (ViT) + PHI-1.5B pipeline for sign-to-text translation, illustrating the promise of compact LLM-assisted decoders coupled with modern vision encoders (Francis (2024-08)).

2.2 Literature Analysis

Despite significant progress, several gaps and challenges remain across recognition, translation, and deployment: Continuity, robustness, and signer independence. Many BdSL works rely on controlled backgrounds or small, curated datasets; performance can degrade with clutter, lighting drift, or unseen signers. Even the strong 92.33% ICCIT result acknowledges background complexity as a failure mode and calls for robustness improvements (Tanni et al. (2024-12)). Edge deployments achieve impressive FPS and accuracy on curated sets, but generalization outside lab conditions remains only partially evidenced (Siddique et al. (2023)). Vocabulary scope and continuous signing. A large share of BdSL literature addresses alphabets/digits or limited word-level sets (e.g., 38 classes; small bespoke word lists), which eases training but limits ecological validity for continuous sign sentences. The shift toward common word-level vocabularies is encouraging, yet datasets remain modest and often lack coarticulation and inter-sign transitions essential to sentence-level SLR (Recognition of Bangladeshi Sign Language (BdSL) Words Using DCNNs, 2023; Baranidharan et al., 2024). Multilingual and cross-modal synthesis. Cross-language mapping (e.g., ISL + multilingual sign generation; SL→Tamil/ASL) demonstrates feasibility but still relies on handcrafted bridges and limited paired corpora. Scaling to robust many-to-many translation remains an open challenge (Aslam et al. (2025); Ratnasingam et al. (2024-12)). System integration and UX. End-to-end applications (text-to-action, voice output, two-way conversion) illustrate valuable integrations, but latency budgeting, failure-mode visibility, and user control are under-reported. The IRJET demonstration integrates sign↔audio loops but offers limited evaluation beyond component accuracy (Real-Time Sign Language and Audio Conversion Using AI, 2024). Similarly, consolidated apps present broad feature sets yet require systematic user studies with Deaf/Hard-of-Hearing (DHH) communities to validate usefulness and accessibility (Comprehensive Application…, n.d.). Modeling trends vs. data realities. Vision Transformers and two-stream backbones are promising, but their

benefits hinge on diverse, signer-independent video corpora and temporal modeling that captures motion coarticulation. Ethical, cultural, and pedagogical factors. Avatar systems and tutor-style interfaces are attractive for learning, yet linguistic correctness, regional variation, and community co-design are essential to avoid mis-teaching or reinforcing majority-dialect bias. The Sri Lankan bilingual pipeline highlights practical, local needs that should inform evaluation criteria beyond accuracy (Ratnasingam et al. (2024-12)). Where the proposed system fits. In light of the above, an impactful capstone should prioritize: (i) robustness (signer independence, background/lighting invariance) atop a competitive backbone (e.g., ViT or two-stream) (Baranidharan et al. (2024); Tanni et al. (2024-12)), (ii) expanded word-level vocabularies with incremental steps toward continuous SLR (Recognition of Bangladeshi Sign Language (BdSL) Words Using DCNNs, 2023), (iii) two-way interaction (speech↔sign) with measurable latency and usability (Deshmukh et al. (2024)), (iv) multilingual support driven by real, paired data and transparent failure modes (Aslam et al. (2025)).

2.3 Literature Review Summary Table

Table 2.3: Literature Review Summary Table

No.	Paper	Research Target	Techniques Used	Result	Limitations	Future Plan
1	Saini & Kumari (2024-03)	End-to-end dynamic sign detection & interpretation	Hybrid CNN/LRCN; Inception–ResNet-v2; pre-processing & filtering	Demo ≈50 FPS; classifier ≈98% (on demo set)	Small eval scope; continuous signing not tested	Expand datasets; generalize; add real-time feedback
2	Sindhu et al. (2024-03)	Bidirectional SLR↔SLT pipeline (ISL)	Reversible-CNN; LSTM sequences; survey of HMM/CNN/RNN	Prototype comparisons; promising reversible-CNN demos	Data scarcity; long-gesture handling; noise sensitivity	Larger ISL data; sentence-level SLT; multilingual output
3	Jeevanandham et al. (2024-08)	Translate webcam signs to text & speech	MediaPipe keypoints + classical ML; text-to-speech	Working prototype; low latency on PC	Small, single-user dataset; limited classes	Expand classes; public dataset; mobile app

No.	Paper	Research Target	Techniques Used	Result	Limitations	Future Plan
4	Haque et al. (2023)	10-word BdSL image classification	Transfer learning (DenseNet201/ ResNet50V2) on ≈1k imgs	≈93% test accuracy; clear per-class results	Small dataset; only 10 words	Add more words; add temporal modeling
5	Deshmukh et al. (2024)	Speech-to-sign rendering in browser	ASR + NLP mapping → animated 3D avatar	Accurate demos; simple UI	Limited vocabulary; depends on online ASR	Offline/hybrid ASR; expand lexicon/languages
6	Ratnasingam et al. (2024-12)	SLSL → Tamil speech + ASL bridging	YOLO–NAS detection; rule mapping; TTS	>90% detection; positive user eval	Limited vocabulary; avatar timing	Add classes; continuous signing; mobile/edge
7	Tanni et al. (2024-12)	38-class BdSL gesture recognition	Custom CNN vs VGG16/ResNet50 on 11,172 images	≈92.33% test accuracy; beats baselines	38/49 letters; clutter hurts	Broaden data; noise robustness; deploy
8	Siddique et al. (2023)	Real-time BdSL letters/numbers on Jetson	Detectron2, EfficientDet-D0, and YOLOv7-Tiny on OkkhorNama + custom 49-class BdSL; deployed YOLOv7-Tiny on Jetson Nano	Detectron2 most accurate (94.9% mAP). YOLOv7-Tiny fastest (≈24 FPS, real time)	Isolated signs; no face/body cues; domain shift	Multimodal cues; larger videos; mobile app
9	(Baranidharan et al. (2024))	Compare CNN families for BdSL	ResNet/-DenseNet/Inception/Xception	ResNet generalizes best; others overfit	Heavy models for mobile; limited data	Lighter CNNs; on-device focus; longer training
10	(Islam et al. (2022))	Text/voice → 3D animated BdSL	Unity–3D; phoneme/word → gesture mapping	Natural-looking animations (simple sentences)	Manual mapping; small lexicon/grammar	Expand vocabulary; improve realism & timing
11	(Luqman (2022))	Isolated SLR with robust temporal cues	Two-stream AVD + AIM; spatio-temporal attention	Strong results on public gesture sets	Isolated signs only; dataset dependence	Extend to continuous SLR; larger benchmarks

No.	Paper	Research Target	Techniques Used	Result	Limitations	Future Plan
12	(Rajkumar et al. (2023))	Two-way sign ↔ text/voice interface	MediaPipe landmarks; DL/BERT/GPT pipeline	≈91% (11 gestures); real-time demo	Small vocabulary; limited poses/backgrounds	More signs; robustness; real-world tests
13	(Islam et al. (2022))	Educational avatar for BdSL alphabet	OpenGL-based 3D humanoid animation	Interactive visualization helps learners	Alphabet-only; manual tuning	Add words/sentences; refine animation rig
14	(Priyadharshini et al. (2024))	All-in-one learning & translation app	CNN recognition; TTS; action playback	Working modules; high-level design	Unclear accuracy; scaling not validated	Benchmark modules; usability tests; add languages
15	(Boobal et al. (2024))	Sign → text → audio pipeline	LRCN for SLR; gTTS for speech	Prototype converts signs to audio	Small dataset; latency on low-end devices	Model compression; larger datasets
16	(Aslam et al. (2025))	ISL recognition + multilingual sign gen	Body→hand→finger pipeline; 3D character generation	Improved accuracy; PoC sign generation	Limited sign set; animation fidelity	Scale vocabulary; richer 3D gestures

Chapter 3

Plans and Deliverables

3.1 Timeline

In chart form, a detailed timeline (Gantt chart) of each task and when it will be completed.

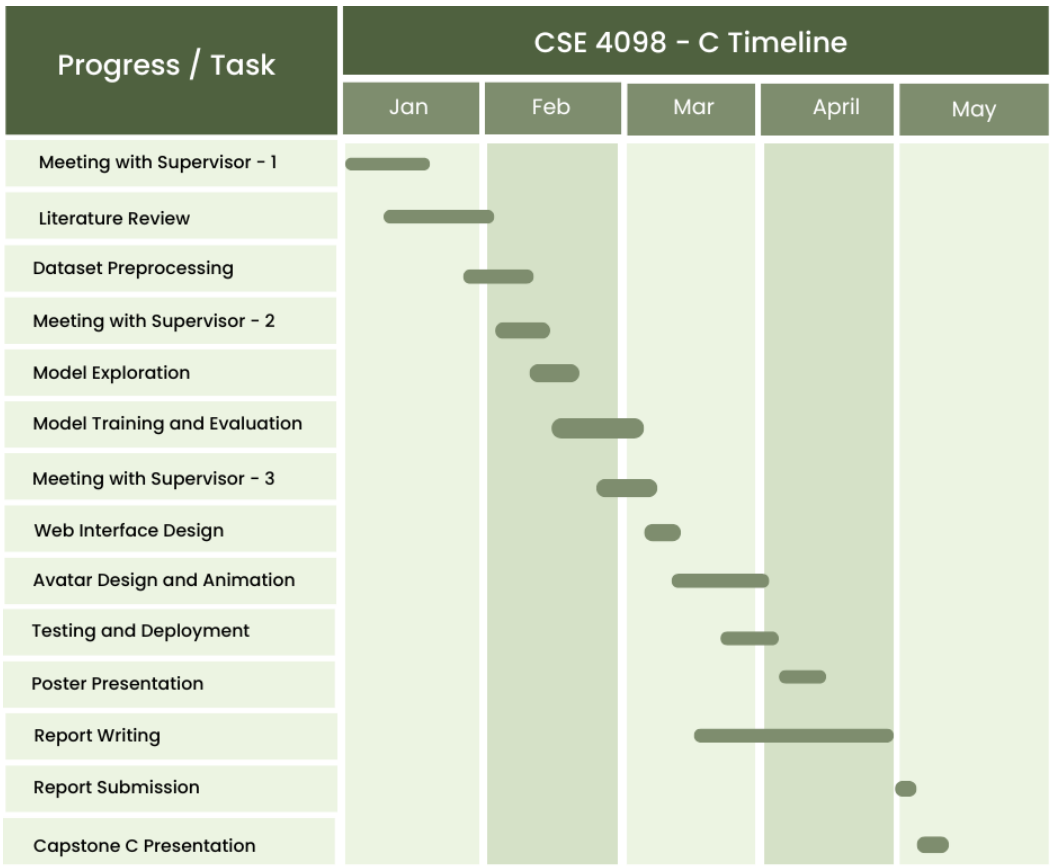


Figure 3.1: Gantt chart of CSE 4098–C timeline.

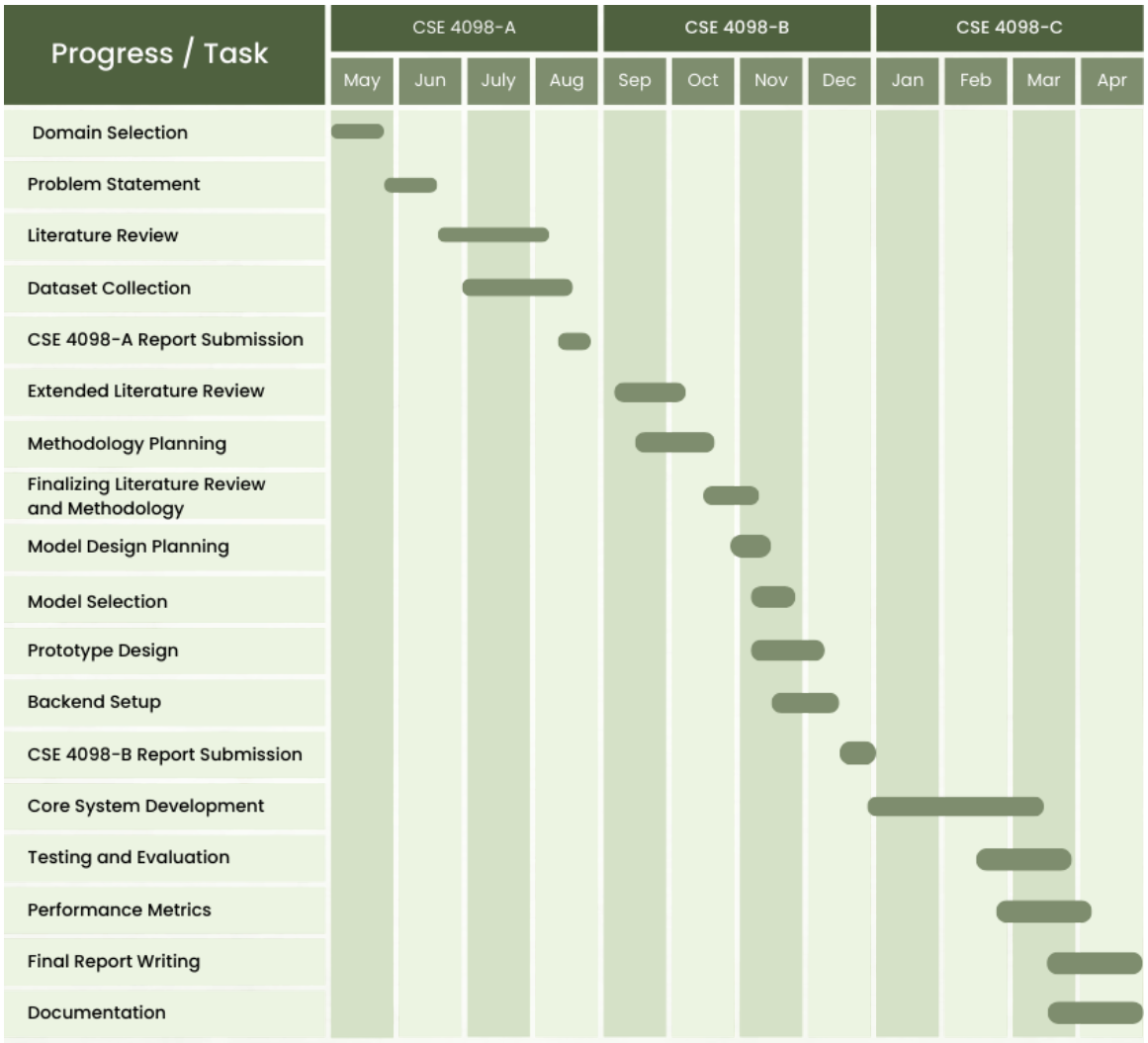


Figure 3.2: Gantt chart of overall project timeline.

3.2 Project Specification

3.2.1 Functional Requirements

The system shall provide bidirectional translation between Bangla sign language and Bangla speech/-text in real time through a web interface. The functional requirements are summarized below.

- **Sign-to-Speech.** Capture live video through the user's webcam and recognize Bangla sign language gestures in real time. Detect single-handed and two-handed signs. When a gesture is held steadily for at least one second with sufficient confidence, add the recognized word to a running sentence. Support sixty distinct sign classes from the BdSLW131 dataset.
- **Speech-to-Sign.** Accept Bangla input as typed text or via the browser's speech recognition API. Process text word by word and display the corresponding sign avatar image sequentially. Support multi-word phrases (up to three words) and handle synonyms, English equivalents, and inflected forms via root-word extraction.
- **Real-Time Performance.** Process video frames at a rate suitable for live interaction; target end-to-end latency from gesture to recognition output under 200 ms on a standard laptop with integrated graphics.
- **Confidence Filtering.** Filter low-confidence predictions; only show predictions above a configurable threshold (default 0.30); commit to the sentence only when sustained average confidence exceeds 0.86 over a one-second window.
- **Sentence Management.** Accumulate recognized words in order; ignore duplicate consecutive words; allow the user to clear the sentence manually.
- **Primary Signer.** In multi-person scenes, identify the primary signer and ignore bystander hands by matching hand wrists to pose wrist landmarks within a proximity threshold.
- **Avatar Output.** Maintain a library of fifty static sign avatar images plus one neutral pose; display sequentially with about one second per sign; if no avatar exists, fall back to large text or neutral pose per user preference.
- **User Interface Layout.** Adapt between desktop (two panels side by side) and mobile (tabs and horizontal swipe to switch panels).
- **Configuration.** Allow runtime toggling of camera aspect ratio (landscape or portrait) and missing-avatar fallback mode without reloading the page.

3.2.2 Non-Functional Requirements

- **Performance.** The web application should load and become interactive within about ten seconds on typical broadband; inference should complete within about 100 ms per frame on mid-range hardware; keep total payload (HTML, CSS, JS, model, data) compact.
- **Accuracy.** Target at least 90% accuracy on a held-out test set; keep the generalization gap (training minus test accuracy) below 5%.

- **Browser Compatibility.** Run on current stable Chrome, Edge, and Firefox without extensions beyond standard webcam and microphone permissions.
- **Privacy.** Process client-side; do not transmit video, audio, or recognition data to a server; function offline after initial load except MediaPipe assets from a CDN if used.
- **Maintainability.** Modular code: separate configuration, utilities, sign-recognition pipeline, speech-to-sign module, and application controller; modules testable in isolation.
- **Resource Efficiency.** Deployed model small enough for browsers without dedicated GPUs; training respects environment limits (e.g., Kaggle RAM versus large raw dataset).
- **Accessibility.** Sufficient color contrast and visual feedback; controls operable by touch and pointer.
- **Robustness.** Handle missing or partial detections; if pose is missing but a hand is present, still build a feature vector with fallback normalization; if neither is detected, show clear UI state without crashing.

3.3 Expected Final Product Description

The expected final product is a single-page web application titled **Bangla Sign Language Translator** that runs entirely in the browser without a backend server, native installation, or specialized hardware beyond a webcam and a microphone.

The interface uses a two-panel layout. The **left panel** provides sign-to-speech translation: live camera feed with start/stop, real-time prediction with the current gesture and a confidence bar, and a sentence builder. The **right panel** provides speech-to-sign translation: Bangla text input, voice input via the browser speech API for spoken Bangla, a translate control that triggers sequential avatar display, and a viewing area for avatars in word order.

A status bar reports loading and readiness. A settings area allows camera aspect ratio (default landscape) and missing-avatar fallback (default large text). The footer can show version and accuracy information.

On mobile, the layout uses two tabs and horizontal swipe; the implementation can use a wide container and `translateX` to slide between panels.

Recognition uses Google MediaPipe Hand Landmarker and Pose Landmarker for keypoints per frame. Keypoints are normalized (nose and shoulder scale when the body is visible; wrist and hand scale otherwise), producing a 154-dimensional vector. A small classifier in pure JavaScript (`SimpleModel`: fully connected layers with ReLU and softmax) matches the trained Keras model after export.

The avatar library contains fifty-one PNG images (fifty signs plus neutral). Text input uses NFKC normalization, phrase matching (up to three words, longest match first), word lookup, synonyms, and Bangla suffix stripping. Voice input uses the Web Speech API with `bn-IN`, interim results, and auto-stop after about two seconds of silence, with manual translate afterward.

Intended users include educators, students, Deaf and hard-of-hearing users communicating with non-signers, and researchers in assistive technology.

3.4 Datasets and Data Strategy

The primary training and evaluation data come from **BdSLW131** (Bangla Sign Language Words) on Kaggle (maintainer e.g. `musaddiquealierfan`): 131 classes and roughly 141 GB raw. This project uses the **first sixty classes** in alphabetical order (13,733 raw images in that subset), aligned with the sixty-class recognizer and the fifty-one static avatars (fifty signs plus neutral) in the web application.

During earlier planning, wider inventories of Bengali and English sign-language datasets were tracked in spreadsheets (image and video sources on Kaggle and Roboflow) to survey availability and size. The capstone implementation concentrates on BdSLW131 with MediaPipe features rather than merging multiple collections of sign language data.

3.5 Tools and Development Stack

The stack is intentionally small for stable browser deployment. The following list uses the document body font (serif or sans, per your thesis class) for descriptions; identifiers such as file names and API symbols remain in `teletype` for distinction from prose.

Programming languages. Python 3.12 for training and conversion scripts; JavaScript (ES2020) for browser logic and inference; HTML5 and CSS3 for the UI; JSON for labels, scaler parameters, and exported weights.

Machine learning and vision. TensorFlow 2.x and Keras for training and TFLite export; mixed precision (`mixed_float16`) on GPU where available. MediaPipe Tasks (0.10.14): Hand Landmarker and Pose Landmarker Lite. NumPy, scikit-learn (`StandardScaler`, stratified splits), OpenCV, Pandas; Matplotlib and Seaborn for figures.

Browser inference. Custom JavaScript forward pass (`SimpleModel`) with `Float32Array`; dense-layer operations exported from TFLite without relying on TensorFlow.js for the core path.

Web APIs. `getUserMedia`; Web Speech API (`SpeechRecognition`) for Bengali input; MediaPipe loaded as ES modules from a CDN and bridged to non-module scripts.

Development and deployment. Kaggle Notebooks (T4 GPU) for training; `python -m http.server` for local static testing; Visual Studio Code; Git/GitHub; static hosting (any static file server or GitHub Pages class of host).

3.6 Proposed Plan & Budget

Table 3.1: Project Cost Breakdown (Table 3.1)

Category	**Details**	**Cost (BDT)**
Personnel	5 team members (2 ML/SLR Engineer, 2 Backend Engineer, 1 UI/Research Engineer) × BDT 30,000/month × 12 months	1,800,000
Equipment (One–Time Cost)	Android test phones ×2, Jetson Nano + camera, tripod/lighting/mics (shared/local)	100,000
Internet & Utilities	Broadband + electricity for dev/testing: BDT 2,500/month × 12	30,000
Software & Storage	Colab, ASR/TTS & HuggingFace API credits, capped estimate	80,000
Research & Data Resources	Dataset curation, journal access, document tools, referencing software	20,000
Testing & Feedback Collection	Deaf/HoH participant incentives, interpreter time, online surveys, field trials	25,000
Localization & Accessibility Tools	Avatar assets, captioning/UX libraries, localization packs	15,000
Deployment & Final Presentation	Play Store fee, domain/SSL, printing, report design, presentation setup	5,000
Total Estimated Budget		**2,075,000**

Chapter 4

Methodology

4.1 Implemented Methodology

The project was carried out through an iterative development process with three major versions of the recognition pipeline, each addressing shortcomings of its predecessor. The underlying philosophy throughout is **keypoint-based classification rather than image-based classification**: hand and pose landmarks are comparatively robust to background variation, lighting, and camera quality, whereas raw-image CNN approaches are highly sensitive to these factors.

At a high level, **sign-to-speech** uses live webcam input, MediaPipe hand and pose landmarks, and a compact **fully connected** classifier. The classifier is trained in Keras with `Dense` layers and deployed as a JavaScript forward pass in the browser.

The Figure 4.1 summarizes the application-level methodology of the bidirectional system, where sign-to-text and voice-to-sign pipelines run in parallel and connect through a central message hub for real-time communication between deaf and hearing users. And, the Figure 4.2 illustrates the machine learning pipeline from BDSL data preparation and class filtering to landmark extraction, feature normalization, stratified splitting, and Dense Neural Network training, followed by model export and performance evaluation.

4.2 Dataset Collection and Selection

The success of any deep learning-based sign language recognition system depends heavily on the quality, diversity, and scale of the training data. This project uses the **BdSLW131** (Bangla Sign Language Words) dataset, maintained on Kaggle (e.g., by contributor `musaddiquealierfan`). The full corpus contains **131** sign classes and approximately **141 GB** of raw imagery.

4.2.1 Dataset Descriptions

BdSLW131 provides word-level Bangla Sign Language samples suitable for static-image studies. For this work, the **first sixty classes** in alphabetical order were selected, yielding **13,733** raw images in the chosen subset. This scope matches the landmark-based classifier and the speech-to-sign avatar set described in the project specification (Chapter 3).

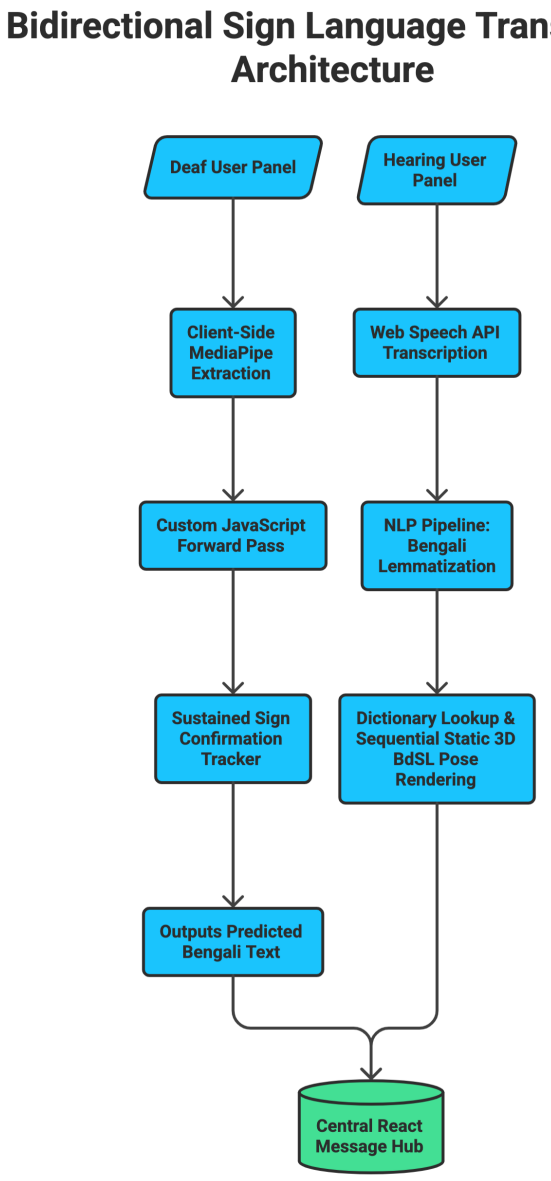


Figure 4.1: Application Level Methodology

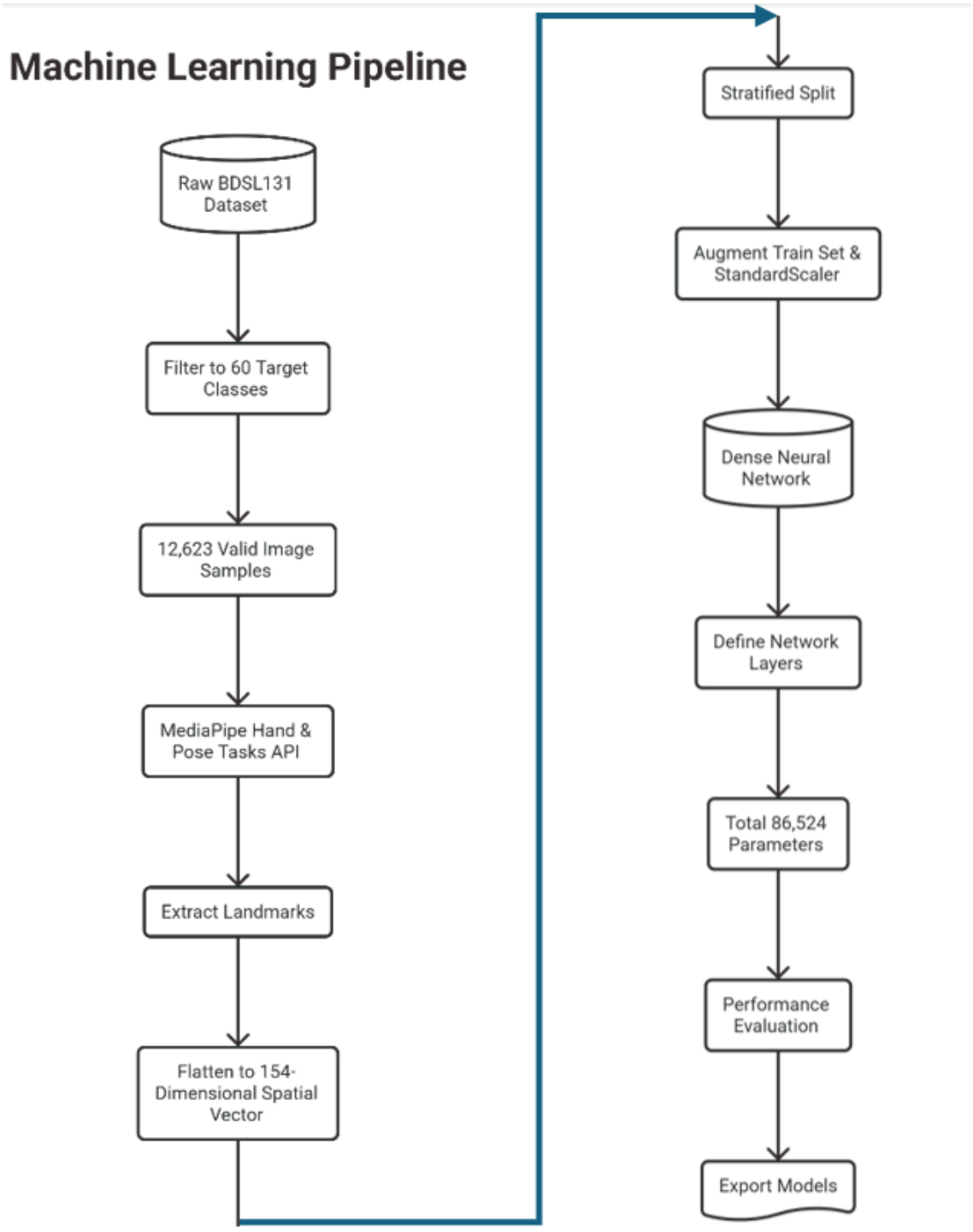


Figure 4.2: Machine Learning Pipeline.

4.2.2 Data Selection Strategy and Challenges

Training focuses on a **single static-image corpus** with a consistent feature pipeline (MediaPipe hand and pose landmarks, normalized to 154 dimensions). Using the full 131-class listing would increase labeling, extraction time, and UI complexity; constraining to sixty classes keeps training and the web application aligned.

Working within Kaggle's memory limit (**30 GB RAM**) alongside a **141 GB** dataset required a **streaming** extraction discipline: each image is read, processed, and released before the next is opened, with explicit memory cleanup during batch extraction.

4.2.3 Dataset Sourcing

The dataset was accessed through Kaggle, which mounts the archive under the notebook filesystem (e.g., `/kaggle/input/`), avoiding a full local download and enabling reproducible paths for training scripts.

4.3 Data Preprocessing

Preprocessing transforms each raw image into a **154-dimensional** normalized feature vector for the neural network. This stage is the primary determinant of model quality for the landmark-based dense fully connected classifier.

4.3.1 Image Reading and Format Conversion

Each image is read with OpenCV (`cv2.imread`), yielding BGR order. Arrays are converted to RGB (`cv2.cvtColor`) because MediaPipe expects RGB. A MediaPipe `Image` wraps the RGB array with the sRGB format flag. After detection, large arrays are released explicitly (`del`) to stay within Kaggle's RAM budget when iterating thousands of files.

4.3.2 Hand Landmark Extraction

MediaPipe **Hand Landmarker** uses the following settings:

- `num_hands` $= 2$
- `min_hand_detection_confidence` $= 0.5$
- `min_hand_presence_confidence` $= 0.5$

The detector runs in **IMAGE** mode (independent training frames, not video tracking). Each hand yields twenty-one landmarks (x, y, z) in normalized image space plus a handedness label (`Left` or `Right`) with confidence.

4.3.3 Hand Sorting by Handedness

Detected hands are assigned to **fixed slots**: the left hand occupies slot 0 (features 0 to 62) and the right hand slot 1 (features 63 to 125). If only one hand appears, it occupies its correct slot and the other slot is zero-padded. This ordering removes the arbitrary "which hand is index 0" instability from earlier pipeline versions.

4.3.4 Pose Landmark Extraction

MediaPipe **Pose Landmarker Lite** uses the following settings:

- `min_pose_detection_confidence` $= 0.5$
- `min_pose_presence_confidence` $= 0.5$

Of thirty-three body landmarks, **nine** upper-body points are retained: nose; inner eyes; ears; shoulders; and elbows, which support relating hand position to the head, neck, and chest for placement-sensitive signs.

4.3.5 Reference Point, Scale, and Normalization

Two normalization paths are used. If pose is valid and shoulder distance exceeds 10^{-6}, the **nose** is the reference and **shoulder distance** is the scale. Otherwise the reference is the **wrist** of the first available hand (left preferred) and the scale is the distance from wrist to middle-finger MCP (landmark 9). Degenerate hand-size samples are discarded.

All active landmarks are centered by subtracting the reference and divided by the scale. Padded (all-zero) placeholders are left unchanged. A binary `has_pose` flag (154^{th} feature) records which normalization path applied.

Feature Vector Dimensionality

The total feature vector dimensionality, denoted as D, is calculated as the sum of the extracted features from both hands, the upper-body pose, and a boolean state flag indicating pose visibility. The mathematical formulation is as follows:

$$D = (N_{hands} \times P_{hand} \times C) + (P_{pose} \times C) + 1 \tag{4.1}$$

Where:

- $N_{hands} = 2$ represents the left and right hands.
- $P_{hand} = 21$ is the number of landmarks extracted per hand.
- $P_{pose} = 9$ is the number of selected upper-body pose landmarks.
- $C = 3$ denotes the spatial coordinates (x, y, z).

Substituting the values:

$$D = (2 \times 21 \times 3) + (9 \times 3) + 1 = 126 + 27 + 1 = 154 \tag{4.2}$$

4.3.6 Batch Processing and Split Statistics

Extraction iterates all sixty classes; after each class, `gc.collect()` limits memory growth. Of **13,733** raw images, **12,623** yielded valid features and **1,110** were skipped (no hand or degenerate geometry). Pose was present in **98.2%** of successful samples.

4.3.7 Train/Validation/Test Split

A stratified split (`random_state=42`) yields **70%** train (**8,836** samples), **15%** validation (**1,893**), and **15%** test (**1,894**).

4.3.8 Augmentation (Training Set Only)

Augmentation runs *after* the split to avoid leakage. **Mirror** augmentation negates x coordinates and swaps the two hand slots. **Jitter** adds uniform noise in $[-0.02, +0.02]$ to active coordinates. The training set expands from 8,836 to **26,508** samples (original + mirror + jitter); validation and test stay unaugmented.

4.3.9 Standardization

Standardization is critical for dense neural networks to ensure stable gradient descent and prevent scaling disparities among disparate anatomical features. Each extracted feature coordinate x_i is standardized using Z-score normalization:

$$z_i = \frac{x_i - \mu_i}{\sigma_i} \tag{4.3}$$

Where μ_i represents the mean of the training samples for feature i, and σ_i is the standard deviation. The `StandardScaler` is fitted exclusively on the augmented training set and subsequently applied via transform to the validation and test sets. The fitted scaler is serialized (JSON) for deployment inference.

4.4 Model Selection and Architecture

The classifier is a **feedforward network of Keras `Dense` (fully connected) layers** operating on the **154-dimensional** landmark vector. In other words, it is a **multilayer perceptron (MLP)**: each layer performs a dense linear transform followed by activation, not a convolution over an image grid.

This must not be confused with the convolutional **DenseNet** architecture (dense connectivity between convolutional blocks in Huang et al.), which is *not* used in this project. MediaPipe supplies fixed-length landmark features; the network maps those features to class probabilities with fully connected layers only.

Development progressed through **three major versions** of the preprocessing and training pipeline.

4.4.1 Version 1 (Hand Keypoints Only)

The initial model used only hand landmarks (twenty-one per hand, two hands, three coordinates each: **126** features) with a three-hidden-layer stack of `Dense` layers (hidden widths 256, 128, and 64) for **ten** gesture classes. Accuracy was acceptable for clearly distinct hand shapes but signs that differ by placement relative to the body were often confused.

4.4.2 Version 2 (Hybrid Hand + Pose Features)

Version two added nine upper-body pose landmarks (nose, eyes, ears, shoulders, elbows), bringing the vector to **153** dimensions, and expanded coverage to **sixty** classes with mirror and jitter augmentation plus careful RAM management for the large raw dataset. Latent issues included **data leakage** (augmentation before splitting), **inconsistent reference points** without informing the model, **missing scale normalization**, and **random hand ordering** between MediaPipe's two hand slots.

4.4.3 Version 3 (Production-Quality Pipeline)

Version three addressed those issues: stratified split first; augmentation only on training; **smart reference and scale** with a `has_pose` flag; **scale normalization** for distance invariance; **handedness-based slot assignment**; and **mirror augmentation with slot swap**. Mixed-precision training (`mixed_float16`) speeds GPU training; a **float32 clone** of the model is exported for TFLite when needed.

4.4.4 Final Architecture: Stacked `Dense` Layers with Regularization

The classifier stacks three blocks. Each block has a `Dense` layer with ReLU activation and L2 kernel regularization, followed by BatchNormalization, followed by Dropout (rates **0.4**, **0.3**, **0.2** from input to output). Hidden widths are **256**, **128**, and **64**. The output `Dense` layer has **sixty** units with softmax. Total parameters: **86,524** (**85,628** trainable). The browser `SimpleModel` implements the same dense linear algebra after BatchNorm weights are folded into the exported graph.

Model Parameter Breakdown

The total parameter count of 86,524 for the DenseNN architecture is mathematically derived from the input dimension ($D_{in} = 154$) and the subsequent Dense and Batch Normalization layers. For a Dense layer, the trainable parameters P_{dense} are calculated as $(Inputs \times Outputs) + Outputs$ (accounting for biases). For a Batch Normalization layer, the parameters P_{bn} equal $Outputs \times 4$. The layer-wise parameter distribution is as follows:

- **Dense 1 (256 units):** $(154 \times 256) + 256 = 39,680$ parameters
- **BatchNorm 1 (256 units):** $256 \times 4 = 1,024$ parameters
- **Dense 2 (128 units):** $(256 \times 128) + 128 = 32,896$ parameters
- **BatchNorm 2 (128 units):** $128 \times 4 = 512$ parameters
- **Dense 3 (64 units):** $(128 \times 64) + 64 = 8,256$ parameters

- **BatchNorm 3 (64 units):** $64 \times 4 = 256$ parameters
- **Output Dense (60 units):** $(64 \times 60) + 60 = 3,900$ parameters

Summing these yields the exact model complexity:

$$TotalParameters = 39,680 + 1,024 + 32,896 + 512 + 8,256 + 256 + 3,900 = 86,524 \quad (4.4)$$

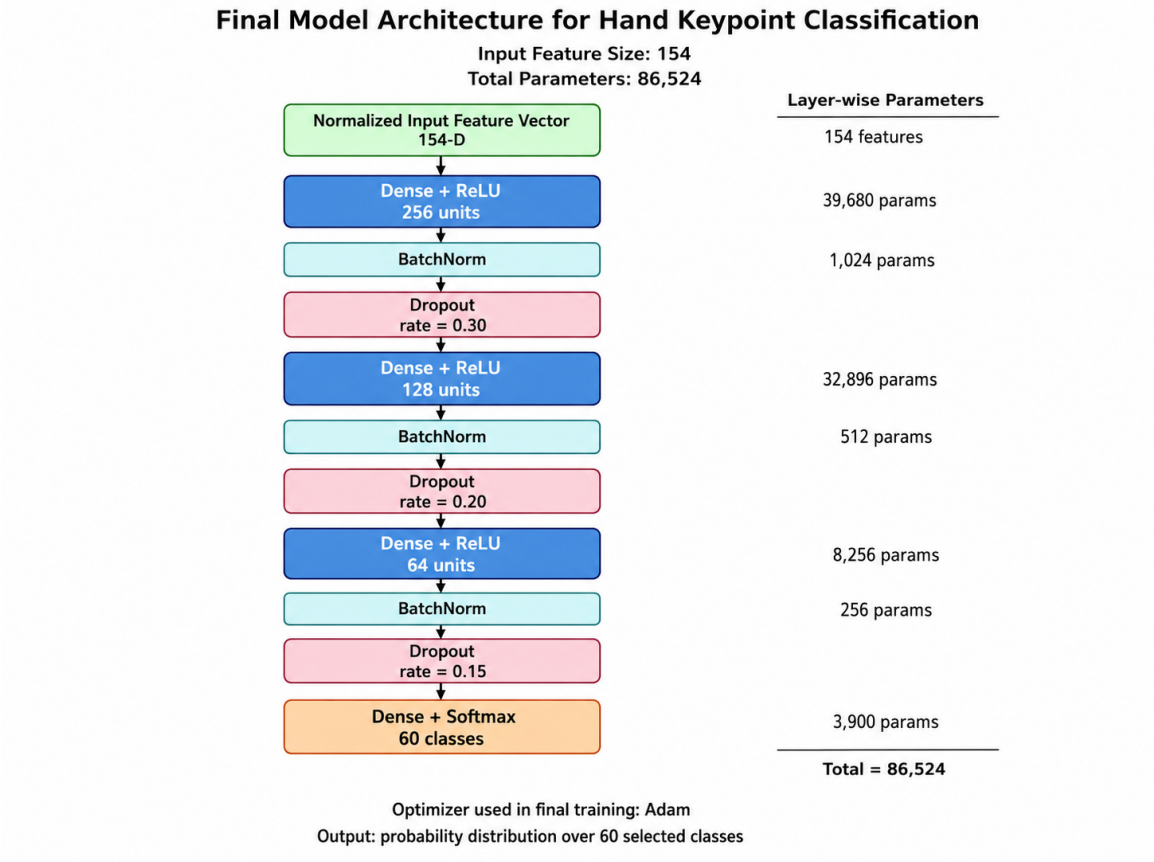


Figure 4.3: Detailed Dense Neural Network architecture for hand keypoint classification, illustrating the sequential dense blocks, regularization layers, and exact parameter composition.

4.4.5 Training Strategy

Training uses the Adam optimizer with an initial learning rate of 0.0008, a batch size of 64, and a maximum of 100 epochs. The model is trained using sparse categorical cross-entropy as the loss function, with accuracy used as the primary evaluation metric. The model optimizes the Sparse Categorical Cross-Entropy loss function during training. This calculates the logarithmic loss between

the predicted probability distribution and the true integer class label. For a batch of N samples, the loss L is defined as:

$$L = -\frac{1}{N}\sum_{i=1}^{N}\log(\hat{y}_{i,c_i}) \tag{4.5}$$

Where c_i is the true class index for sample i, and $\hat{y}_{i,c_i}$ is the predicted probability for the correct class generated by the softmax activation function.

The dataset is split into training, validation, and test sets using a stratified split with proportions of 70%, 15%, and 15%, respectively. Data augmentation is applied only to the training set after splitting, preventing data leakage. Each training sample is expanded three times using the original feature vector, a mirrored version, and a jittered version. Feature standardization is performed using a StandardScaler fitted only on the training set and then applied to the validation and test sets.

Two callbacks are used during training. EarlyStopping monitors the validation loss with a patience of 15, a minimum improvement threshold of 0.0005, and restores the best model weights. ReduceLROnPlateau also monitors the validation loss and reduces the learning rate by a factor of 0.3 after 6 epochs without improvement, with a minimum learning rate of 1×10^{-7}. Mixed precision training is enabled using the `mixed_float16` policy to improve training efficiency on supported hardware.

4.4.6 Evaluation

The quantitative evaluation of the model relies on the standard components of the confusion matrix: True Positives (TP), True Negatives (TN), False Positives (FP), and False Negatives (FN). The core metrics utilized in this study are mathematically defined as follows:

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{4.6}$$

$$Precision = \frac{TP}{TP + FP} \tag{4.7}$$

$$Recall = \frac{TP}{TP + FN} \tag{4.8}$$

$$F1\text{-}Score = 2 \times \frac{Precision \times Recall}{Precision + Recall} \tag{4.9}$$

For the multi-class classification problem involving 60 classes, these metrics are aggregated using a weighted average based on class support to account for any class imbalances. Additional evaluations on the held-out test set include per-class behavior and a confusion matrix with the top confused pairs.

4.4.7 Comparative Summary

The table below contrasts the three pipeline generations for the landmark-based dense classifier.

Version	Feature / Scope	Role
V1	126-D hands; 10 classes	Baseline fully connected model
V2	153-D hand+pose; 60 classes	Expanded coverage; known latent flaws
V3	154-D + fixes (split, scale, slots, flag)	Deployed training recipe

Table 4.1: Evolution of the landmark-based classification pipeline

4.5 Training Environment Setup

Training runs in **Kaggle Notebooks**, with about **30 GB** RAM and **30 GB** disk. Datasets mount under `/kaggle/input/`. Outputs are written to `/kaggle/working/` for download as a ZIP after the run completes.

4.5.1 Infrastructure: Kaggle Notebooks

- **Data locality:** BdSLW131 is impractical to download in full locally; Kaggle mounts the dataset directly.
- **GPU:** Accelerator **GPU T4 x2** (TensorFlow uses one GPU for this small model). The T4's tensor cores pair with mixed precision.
- **CPU extraction:** MediaPipe keypoint extraction over all images often runs on **CPU** (Kaggle lacks the EGL setup MediaPipe's GPU delegate expects). Full extraction took on the order of **30 to 60 minutes** for the sixty-class subset in the Kaggle training notebook.

4.5.2 Software Stack and Libraries

- **Python 3.12** with **TensorFlow 2.x** / Keras.
- **MediaPipe** Tasks (**0.10.14**: Hand and Pose landmarkers).
- **NumPy**, **scikit-learn** (`StandardScaler`, stratified splits), **OpenCV**, **Pandas**, **Matplotlib**, **Seaborn**.
- The notebook may reinstall packages with `pip install -q` for consistency.
- MediaPipe model files download once from Google Storage into `/tmp/`.

4.5.3 Mixed Precision Configuration

Training enables TensorFlow Keras mixed precision with global dtype policy `mixed_float16`, which accelerates matrix operations on the T4. The final `Dense` layer stays in **float32** for softmax stability. For TFLite export, a **float32 clone** receives copied weights when float16 ops are unsupported.

4.5.4 Memory Management and Determinism

The extraction loop reads one image at a time, runs detection, then releases arrays (`del`, `gc.collect()` per class). Multi-threaded decoding was avoided to prevent RAM spikes. Random seeds (`random_state=42`) fix splits; full bitwise reproducibility across CUDA runs is not guaranteed.

4.5.5 Artifacts Written to `/kaggle/working/`

Outputs include the following:

- trained `.keras` model and TFLite export;
- pickled scaler and class labels JSON;
- metadata and training history CSVs;
- evaluation CSVs and confusion summaries;
- visualization PNGs.

4.5.6 Installation and Configuration

Dependencies are imported after optional quiet reinstalls, for example:

Listing 4.1: Representative environment imports

```
!pip install -q numpy scikit-learn pandas matplotlib seaborn
   opencv-python pillow mediapipe tensorflow

import os
import json
import pickle
import warnings
import gc
import urllib.request
from datetime import datetime

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import cv2

import tensorflow as tf
from tensorflow.keras import layers, models, regularizers
import mediapipe as mp
from mediapipe.tasks.python import vision

```

```
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import (
    classification_report, confusion_matrix,
    accuracy_score, precision_score, recall_score, f1_score
)
```

Chapter 5

Design and Development

5.1 Scope of This Chapter

This chapter covers the technical design decisions and implementation details behind the bidirectional Bangla Sign Language Translator. The system was built entirely as a client-side web application—no backend server, no cloud API, no database—a constraint that shaped virtually every architectural choice made during development. The discussion walks through how the two translation directions were conceived, how they were integrated into a single coherent interface, and how the overall software structure was kept clean enough to extend in the future.

The chapter does not revisit the machine learning training process in detail, as that is addressed in Chapter 3. Instead, it focuses on how the trained model and auxiliary data were packaged and consumed inside a browser environment, how the user interface was designed to serve both hearing and deaf users on the same screen, and what deliberate trade-offs were accepted to keep the system lightweight and deployable without any server infrastructure.

5.2 Project Architecture

5.2.1 Bidirectional Design Rationale

Most sign language tools available today are strictly one-directional, either they translate gestures into text or they display signs for typed input, but rarely both in the same place. The goal here was to close that gap by building a tool that feels genuinely conversational: a hearing person can type or speak Bengali and see the corresponding sign avatars on screen, while a deaf person sitting across can simultaneously perform gestures that get recognised and shown as text. Both sides of the conversation exist in the same browser window at the same time.

This led to a clear two-panel structure. The left panel handles **Sign** $\rightarrow$ **Speech**, gesture recognition through the camera, and the right panel handles **Speech** $\rightarrow$ **Sign**, text or voice input converted to avatar animations. On desktop, both panels are visible side by side. On mobile, a swipeable tab interface collapses them to fit narrower screens.

5.2.2 Module Structure

The codebase is split into seven files, each with a clearly bounded responsibility, as summarised in Table 5.1.

Table 5.1: Source files and their roles in the Combined Translator.

File	Role
`config.js`	Single source of truth for all tunable runtime parameters (model paths, confidence thresholds, camera constraints, fallback mode).
`utils.js`	Shared feature extraction, z-score normalisation, primary-person hand filtering, and top-N prediction helpers.
`sign-to-speech.js`	Camera capture, the continuous inference loop, the pure-JS MLP engine, and sustained sign sentence tracking.
`speech-to-sign.js`	Bengali text parsing, multi-word phrase detection, suffix stripping, avatar sequencing, and Web Speech API voice input handling.
`app.js`	Application bootstrap, side-by-side layout management, mobile swipe handling, and runtime settings toggles.
`extract_model.py`	Offline conversion utility, this reads TFLite tensors, dequantizes weights if needed, traces layer shapes from input size 154, and validates outputs against the TFLite interpreter before serialising to `model_weights.json`.
`convert_scaler.py`	Offline conversion utility, this deserialises the pickled `StandardScaler` from training and exports its mean, scale, and variance arrays to `scaler.json` for use at browser inference time.

All browser-side modules are loaded as plain `<script>` tags and communicate through a shared `CONFIG` object and direct method calls, intentionally avoiding any framework overhead. This keeps the codebase easy to audit, debug, and serve from a static file host. The two Python utilities are run once, offline, whenever the underlying model or scaler changes; their outputs, `model_weights.json` and `scaler.json`, are then committed alongside the application files.

5.2.3 The Gesture Recognition Pipeline (Sign → Speech)

The gesture recognition side is the more technically involved of the two. When the user starts the camera, the system launches a continuous `requestAnimationFrame` loop that processes each new video frame through three sequential stages.

Stage 1 — Landmark Detection. Each frame is passed to two MediaPipe landmarkers running concurrently: a Hand Landmarker that returns up to 21 keypoints per hand (supporting up to two hands), and a Pose Landmarker that returns 33 body skeleton points. Both run in VIDEO mode, which exploits temporal continuity between frames and is significantly more efficient than per-image mode.

Stage 2 — Primary Person Filtering. When multiple people are visible in the frame, the system applies a proximity filter to decide whose hands are relevant. It computes the Euclidean distance in normalised image coordinates between each detected hand wrist and the pose-estimated wrists of the primary body. Any hand whose wrist falls outside a configurable threshold (default: 0.15) is discarded, preventing a bystander from corrupting the prediction.

Stage 3 — Feature Extraction and Inference. The filtered landmarks are assembled into a 154-dimensional feature vector:

- 63 values for the left hand (21 landmarks $\times$ 3 coordinates);
- 63 values for the right hand (21 landmarks $\times$ 3 coordinates);
- 27 values for 9 upper-body pose landmarks (nose, eyes, ears, shoulders, elbows);
- 1 binary flag indicating whether valid pose data was obtained.

Before concatenation, all coordinates are centred relative to a reference point (the nose tip when pose is available, the hand wrist otherwise) and scaled by a body-proportional factor (shoulder width or hand size). This normalisation makes the model position- and scale-invariant—the signer does not need to stand at a fixed distance from the camera.

The resulting vector is then z-score normalised using the mean and variance exported by `convert_scaler.py` and stored in `scaler.json`. It is fed into a pure-JavaScript MLP implemented inside `sign-to-speech.js` whose weights were exported by `extract_model.py`. The engine supports ReLU and softmax activations and runs a straightforward dense-layer multiply-and-accumulate loop without any external ML library. This was a deliberate choice: loading TensorFlow.js adds roughly 1–2 MB of network transfer and introduces its own warmup latency; the custom engine loads in milliseconds and avoids those costs entirely.

The model covers **60 gesture classes**, representing a practical vocabulary of everyday Bangla signs including body parts, numbers, common commands, pronouns, and social expressions.

5.2.4 Sustained Sign Confirmation

Raw frame-by-frame predictions are inherently noisy—the hand shifts slightly between frames, lighting changes, and the classifier occasionally produces a different top-1 label for the same gesture. To

prevent accidental word insertions, a **sustained sign tracker** was implemented. A word is committed to the sentence only if:

1. the same gesture label has been the top prediction continuously for at least **1 000 ms**, and
2. the average prediction confidence across all frames in that window is at or above **86 %**.

If either condition is unmet, the prediction is shown in the live display but nothing is written to the sentence. This gives the system a natural feeling of intentionality—the user must hold a gesture deliberately, much like pausing while speaking. Once a word is committed, the tracker resets immediately, preventing the same label from being added twice in a row. The sentence buffer holds up to ten words and can be cleared at any time with the dedicated button.

5.2.5 The Speech-to-Sign Translation Pipeline

The Speech → Sign panel takes Bangla text—typed manually or dictated via the Web Speech API—and displays the corresponding sign gestures as a timed sequence of static avatar images, one word per second.

Before image lookup, the input text passes through a preprocessing pipeline:

1. **Unicode normalisation** (NFKC) to unify composed and decomposed forms of Bengali characters, which can differ invisibly depending on the input method used.
2. **Punctuation stripping** using a regular expression that targets non-Bengali, non-alphanumeric characters at word boundaries.
3. **Multi-word phrase detection**, scanning up to three adjacent words first to catch compound expressions (such as “wait” or “brush teeth”) before falling back to single-word lookup.

Word matching is driven by an NLP pipeline utilizing lemmatization to handle the morphologically rich nature of Bengali. When the text is processed, a Bengali lemmatizer maps conjugated verbs, pluralized nouns, colloquialisms, and other inflected forms back to their canonical root words (lemmas).

This approach ensures that multiple grammatical variations of a single concept map to the same base token. Once the root lemma is identified, the system queries the lookup dictionary to seamlessly fetch and display the corresponding pre-rendered BdSL static image. By relying on lemmatization rather than basic suffix stripping, the system significantly improves its accuracy and resilience against conversational input variations.

5.2.6 Avatar Creation and Static Pose Generation Using Blender

This section describes the methodology followed to create an expressive 3D avatar and generate a comprehensive library of static Bangla Sign Language (BdSL) poses using Blender. Because the application identifies speech and outputs corresponding pre-rendered images, the objective of this pipeline is to produce visually clear, linguistically accurate, and reusable still images rather than continuous video animations.

5.2.6.1 Overview of the Avatar Pose Pipeline

Instead of constructing a humanoid model from scratch, a professionally rigged character model is utilized to ensure anatomical correctness and realistic joint movement. The workflow follows a structured pipeline comprising model acquisition, scene preparation, pose library generation, reference matching, and final static rendering.

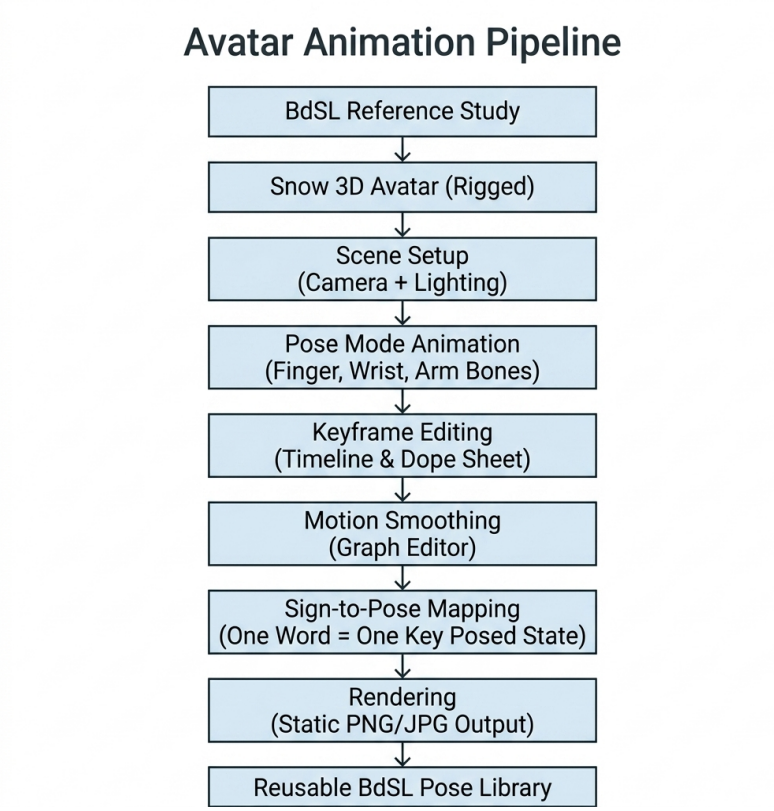


Figure 5.1: Avatar static pose generation and rendering pipeline.

5.2.6.2 Software and Resources

The following tools and resources are used throughout the avatar creation process:

- Blender (latest stable version) for 3D posing and rendering.
- Snow free character model from Blender Studio.
- Custom human reference photos demonstrating BdSL gestures.
- Blender's Pose Mode, Asset Browser, and Pose Library features.

5.2.6.3 Model Acquisition and Armature Setup

The Snow character model is acquired from Blender Studio in `.blend` format. The model includes a fully rigged humanoid mesh with a predefined armature and proper weight painting. The character features a professional armature consisting of torso bones, arm and forearm bones, wrist bones, individual finger bones, and facial control bones. The armature is selected and switched to Pose Mode, enabling precise rotation-based control of each joint.



Figure 5.2: The pre-rigged Snow character model and predefined armature in Blender's viewport.

5.2.6.4 Scene Preparation and Neutral Posture

Unnecessary objects such as unused cameras and dynamic lights are removed to maintain a clean workspace. A dedicated camera is positioned directly in front of the avatar to capture the upper body, hands, and facial region clearly. Soft, evenly distributed lighting is applied using key and fill lights to enhance finger visibility and avoid harsh shadows on the hands. Before applying specific BdSL signs, the avatar is established in a default neutral pose.

5.2.6.5 Pose Library Setup and Sign-to-Pose Mapping

For each distinct Bangla word, corresponding BdSL references are studied carefully. Because the system maps one word to one key posed state, each sign is manually sculpted by adjusting the finger bones, wrist joints, and forearm rotations. To manage the extensive vocabulary, Blender's Asset Browser is utilized to create a reusable Pose Library. Custom hand shapes and facial expressions (e.g., eye masks, mouth shapes) are categorized, allowing for rapid combination and mapping when constructing complex signs.



Figure 5.3: Avatar established in the baseline neutral pose prior to gesture mapping.

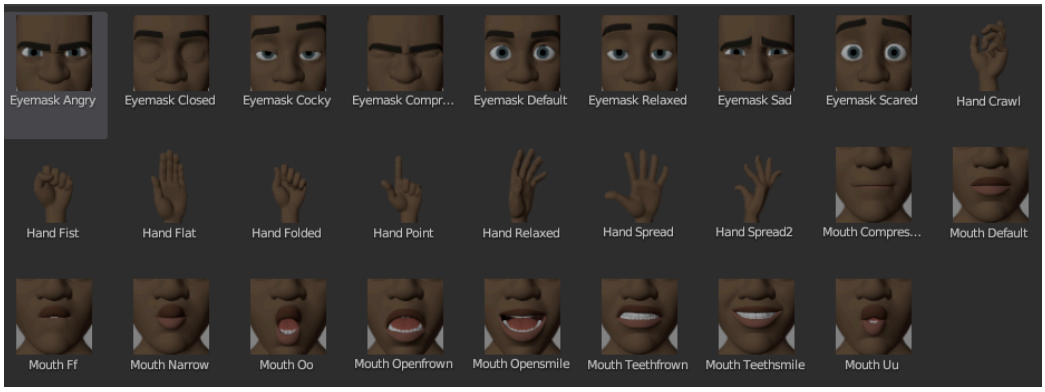
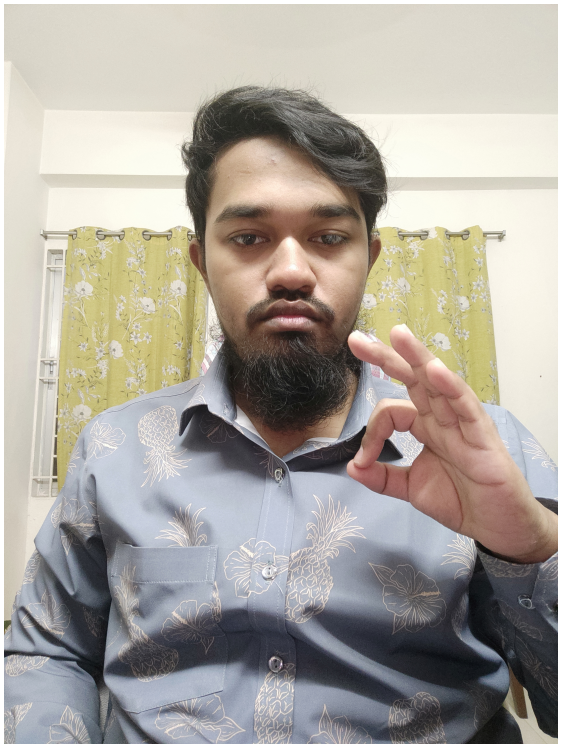


Figure 5.4: Organization of facial expressions and hand states within the Blender Pose Library.

5.2.6.6 Manual Pose Creation and Reference Matching

Accuracy is ensured by directly matching the 3D rig to human reference photos captured specifically for this project. Minor facial expressions are added where appropriate to enhance natural communication, though they are kept minimal to ensure the viewer's focus remains on the primary hand gestures.



(a) Human Reference



(b) Avatar Output

Figure 5.5: Comparison of human reference and finalized avatar pose for the BdSL sign "Office".

5.2.6.7 Rendering and Output

Once a pose is finalized and saved to the library, it is rendered out as a static image. Final images are rendered using the Eevee rendering engine for rapid processing. Output settings are configured to produce high-resolution static formats (e.g., PNG or JPG). Each rendered image is named according to its corresponding Bengali word root, allowing the web application's lookup dictionary to seamlessly fetch and display the correct image when speech is detected.



(a) Human Reference



(b) Avatar Output

Figure 5.6: Comparison of human reference and finalized avatar pose for the BdSL sign “Wait”.

5.3 Web Deployment and Runtime

5.3.1 Client-Side Architecture

Conversion utilities.

- `extract_model.py` reads TFLite tensors, dequantizes if needed, traces layer shapes from input size 154, and checks outputs against the TFLite interpreter.
- `convert_scaler.py` serializes the pickled scaler to JSON (mean, scale, and variance fields used at inference).

The decision to implement everything in the browser without a backend was driven by three practical requirements: the system should work without a persistent internet connection once assets are loaded, it should be deployable on a free static host with no server-side configuration, and it should not transmit camera footage to any external service. All three constraints point toward client-side inference. MediaPipe's WASM-based landmarkers run entirely inside the browser sandbox. The classification model weights are loaded as a JSON file from the local static file tree. Nothing about the user's gestures or speech leaves their device.

The one external dependency at runtime is the MediaPipe library itself, loaded as an ES module from the jsDelivr CDN:

```
https://cdn.jsdelivr.net/npm/@mediapipe/tasks-vision@0.10.14/vision_bundle.mjs
```

The version is pinned explicitly to `0.10.14` to prevent unexpected behaviour from automatic upstream updates. The `HandLandmarker`, `PoseLandmarker`, and `FilesetResolver` classes are imported and attached to the `window` object so that the subsequent plain-script application modules can access them without ES module syntax, maintaining compatibility with the simpler `<script src="...">` loading pattern.

The application waits for MediaPipe to be fully available before initialising either translation module, using a polling check with a 30-second timeout. Both modules then initialise in parallel via `Promise.all`, so the longer model download does not delay the avatar module from becoming ready.

5.3.2 GPU Acceleration

Both landmarkers are configured with `delegate: 'GPU'`, routing inference through WebGL whenever the browser supports it. In practice this reduces per-frame detection latency by roughly 60–70 % compared to CPU-only mode, making real-time inference feasible on mid-range consumer hardware. The *lite* variant of the pose model is used specifically to keep GPU load manageable, given that the hand and pose landmarkers process each frame simultaneously.

5.3.3 Camera and Layout Controls

Camera resolution adapts to a user-selectable aspect ratio—landscape (640×480) by default, with a portrait mode (480×640) available for users signing with a phone held upright. The toggle in

the settings bar writes directly to the `CONFIG` object; if the camera is already running, it restarts automatically with the new constraints.

The layout uses CSS Grid for the desktop two-panel view and CSS flexbox for the mobile single-panel swipeable view. The transition between panels is implemented as a `translateX(-50%)` CSS transform on the panel container—set to `width: 200%`—rather than hiding and re-showing individual elements, which preserves the DOM state of both modules across swipes. Touch swipe detection is handled natively via `touchstart` and `touchend` events, with a 60-pixel minimum distance to avoid accidental triggers.

5.3.4 Speech Input

Voice input for the Speech → Sign panel uses the browser's built-in `window.SpeechRecognition` API (with a `webkitSpeechRecognition` fallback for Chromium-based browsers). The recogniser is configured for continuous listening with interim results shown in a status strip below the text area. The language is set to `bn-BD`, or the user's device Bengali locale if available. A two-second silence timeout stops the recogniser automatically once speech pauses, avoiding open-ended recording sessions. The voice button enters a pulsing red animation while active, providing clear visual feedback that the microphone is open.

5.3.5 Running the Application

Because the application loads local files via `fetch()` (model weights, scaler, and class labels), it requires a minimal HTTP server rather than being opened directly as a `file://` URL. During development, a Python server (`python -m http.server`) or any Node-based static server is sufficient. For permanent deployment, the entire folder can be uploaded to any static host, GitHub Pages, Netlify, Vercel, or a CDN-backed object storage bucket—with no build step or server-side configuration required.

5.4 Mobile Application and APK Generation

To ensure maximum accessibility, the system was not only deployed as a web application but also packaged as a native Android application (APK). This was achieved by leveraging Progressive Web App (PWA) technology combined with a Trusted Web Activity (TWA) wrapper.

5.4.1 PWA Infrastructure

The foundation of the mobile experience relies on two core files integrated into the root directory:

- **manifest.json:** This file defines the app's metadata, including the theme color, high-resolution icons (192x192 and 512x512), and the display mode set to `standalone`. This allows the app to hide browser UI elements (like the URL bar) and feel like a native app.
- **sw.js (Service Worker):** This script handles background caching of the heavy machine learning models and avatar assets, enabling faster load times and basic offline functionality.

5.4.2 APK Packaging via PWABuilder

The final transformation from a URL to an installable APK was performed using **PWABuilder**. The process utilized the **Trusted Web Activity (TWA)** protocol, which allows a full-screen Chrome instance to run the web application within a native Android container. This approach ensured that the high-performance TensorFlow.js inference and WebGL hardware acceleration remained fully functional on mobile devices without needing to rewrite the logic in Java or Kotlin. The resulting APK was digitally signed using a keystore for secure installation.

5.5 Application Demonstration

The final product is a unified, bidirectional translator that adapts to various screen sizes. Figures 5.7 and 5.8 illustrate the functional interface in both desktop and mobile environments.

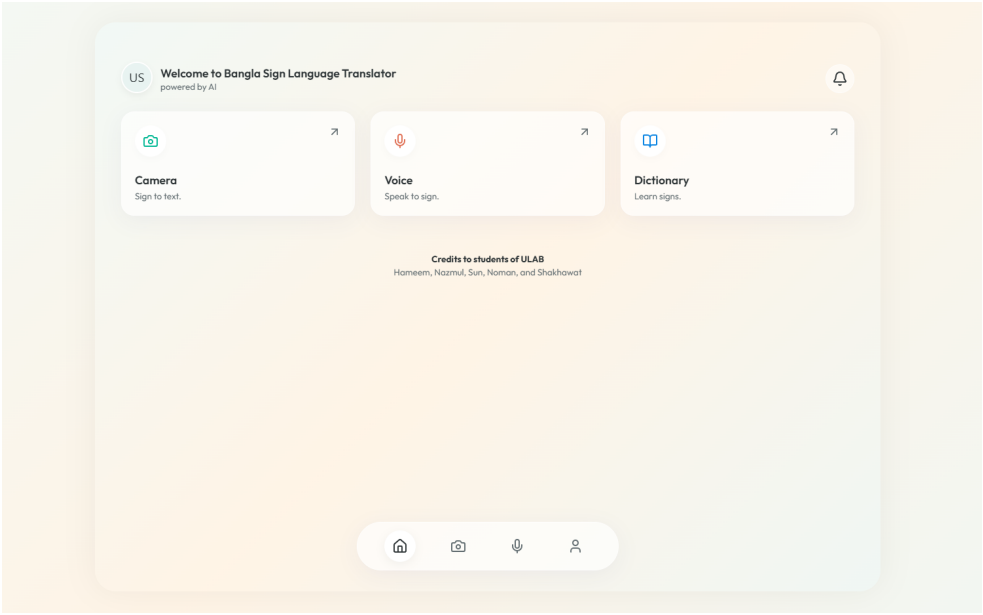


Figure 5.7: Web Application Interface.

5.6 Future Extensions

The current implementation covers the core bidirectional workflow effectively, but several directions present themselves as natural next steps.

Expanding the gesture vocabulary. The model was trained on 60 classes. Adding further classes requires collecting additional gesture samples, retraining the MLP, and replacing the three data files (`model_weights.json`, `scaler.json`, `class_labels.json`). Because the inference engine is a generic dense-layer runner, it handles models of different sizes without any code changes. A realistic near-term target would be 150–200 classes to cover a more complete everyday vocabulary.

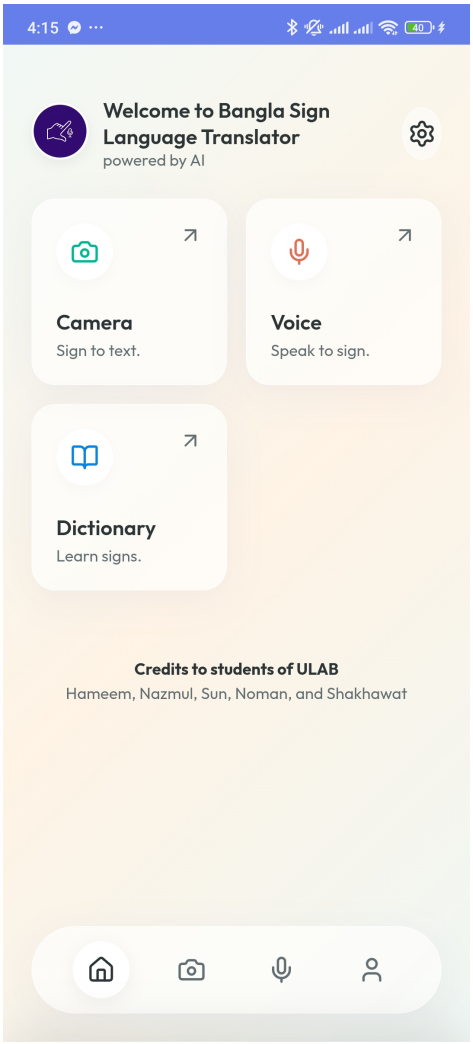


Figure 5.8: Mobile App Home Screen.

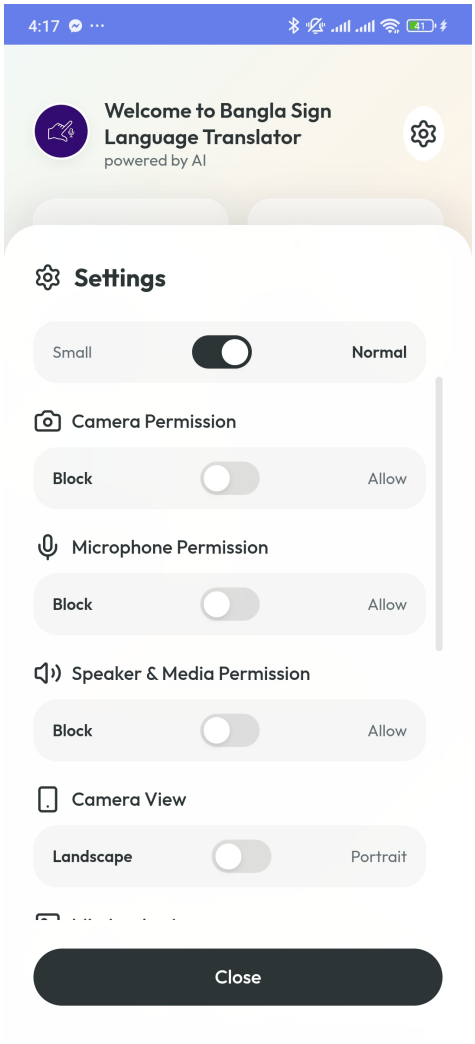


Figure 5.9: Settings view.

Replacing static avatars with 3D animation. The current approach of showing a static PNG for each sign cannot convey motion—many signs are defined as much by their movement trajectory as by their hand shape. Replacing the PNG library with a skeletal avatar rendered in WebGL and driven by recorded landmark sequences would allow fluid, animated sign representations. The Blender-based rig described in Section **??** provides a natural starting point for this work.

Sentence-level grammar translation. At present, both directions operate word by word. A proper bidirectional translation would map Bangla grammar to sign grammar, which differs substantially in structure. Integrating a lightweight sequence model or a rule-based grammar transducer could produce more natural sign sequences from full Bengali sentences, rather than displaying signs in the linear word order of the spoken language.

Offline-first Progressive Web App packaging. Wrapping the application as a Progressive Web App (PWA) with a Service Worker would allow it to be installed on a phone or tablet and used entirely offline after the first load. The MediaPipe WASM files would need to be cached at install time, which is straightforward with a cache-first fetch strategy.

Multi-language support. The avatar dictionary and variation maps are specific to Bengali, but the underlying architecture is language-agnostic. Adding support for another South Asian sign language would primarily require building a new dictionary file and avatar set, with minimal changes to the core logic.

Confidence display improvements. The current UI shows the top-1 prediction and a single confidence bar. Displaying the top-3 candidate signs during ambiguous frames would help users understand when to refine their hand position, and would be particularly useful for training and demonstration contexts.

Chapter 6

Result and Evaluation

6.1 Result Analysis and Discussion

This chapter analyzes four sign language recognition models developed in this study: **MobileNetV2**, **ResNet50**, **DenseNN** (keypoint-based MLP), and **YOLOv11-Pose + MediaPipe Hybrid**. Each model reflects a distinct architectural approach, from end-to-end convolutional networks to hybrid keypoint pipelines.

To study deployment-like behavior, the four models were evaluated on a held-out set of 60 reference images (one per class) captured outside the original training distribution. Each image was expanded into 10 augmented variants, yielding **600** test samples. Augmentations simulate practical conditions including low and high lighting, contrast shifts, blur, rotation, translation, and Gaussian noise.

The evaluation set had known limitations: a single signer, one indoor environment, and inconsistent framing. Reported values combine *measured* results under these conditions.

6.1.1 Evaluation of the Result

Headline metrics are summarized in Table 6.1. DenseNN achieved the highest Top-1 accuracy in both measurement and projection scenarios among the four models.

Table 6.1: Headline metrics across the four evaluated models.

Model	Top-1 accuracy	Detection success rate	Browser latency in ms (est.)	Disk size	Params
DenseNN	68.00%	97.00%	233.8	1.70 MB	86,524
MobileNetV2	42.00%	100.00%	399.6	48.39 MB	3,089,404
ResNet50	41.17%	100.00%	1219.6	370.65 MB	24,704,003
YOLOv11-Pose	27.17%	100.00%	283.4	0.39 MB	96,485

DenseNN is more than 1.6× more accurate than the next best model on the real-world-style evaluation set, while remaining roughly 36× smaller than MobileNetV2 and 218× smaller than ResNet50 on disk. The gap between training-time behavior and this evaluation highlights how much each architecture relies on memorized texture cues versus features that transfer under shift.

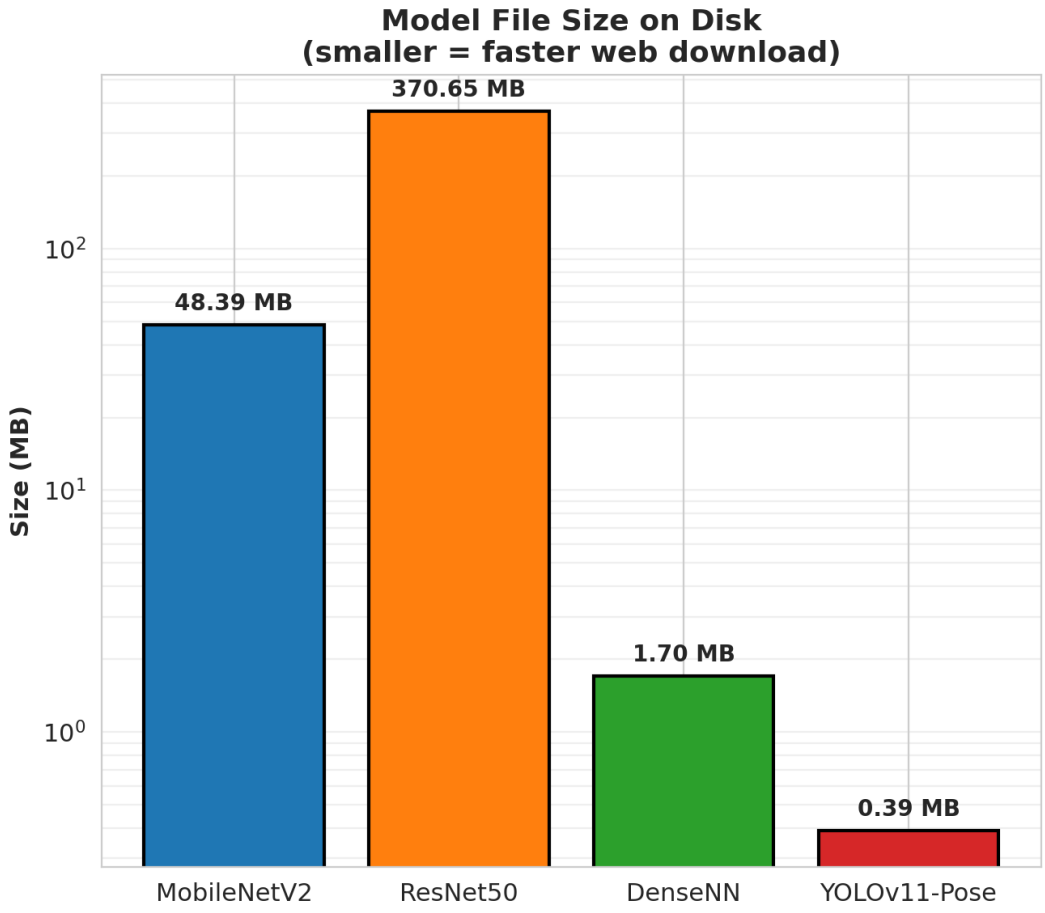


Figure 6.1: Model size on disk.

The latency profile of each model differs significantly when separating preprocessing from raw inference. The pure CNN models have negligible preprocessing time but heavier inference cost. DenseNN spends most of its execution time in MediaPipe-based feature extraction, while YOLOv11-Pose spends nearly all its time in dual feature extraction.

The robustness of each model under realistic image variations is presented in Table 6.2. DenseNN maintained the highest accuracy across nearly every augmentation category.

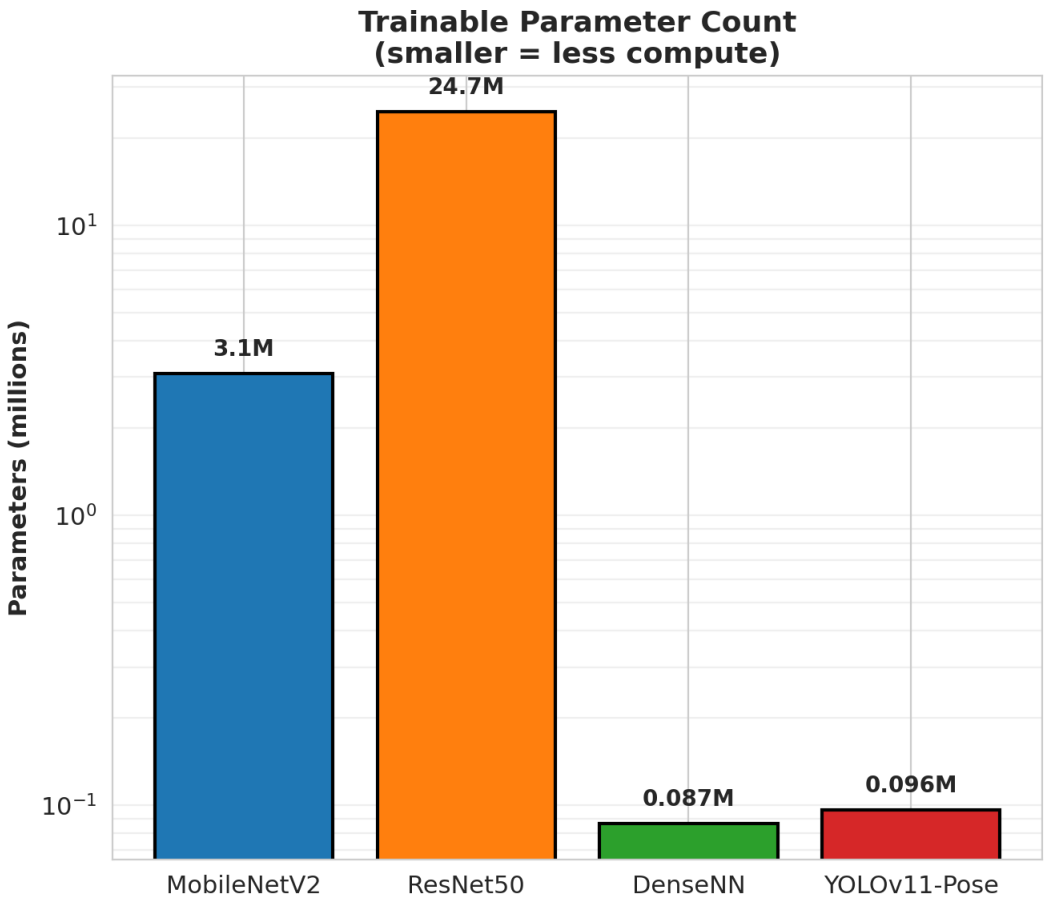


Figure 6.2: Trainable parameter count.

Table 6.2: Top-1 accuracy under different augmentation conditions

Condition	DenseNN	MobileNetV2	ResNet50	YOLOv11-Pose
original	71.67%	43.33%	43.33%	28.33%
bright_up	61.67%	36.67%	35.00%	26.67%
bright_down	71.67%	43.33%	41.67%	26.67%
contrast_high	60.00%	38.33%	38.33%	26.67%
rotate_pos	68.33%	40.00%	46.67%	33.33%
rotate_neg	71.67%	43.33%	46.67%	23.33%
zoom_in	70.00%	45.00%	33.33%	28.33%
gaussian_blur	75.00%	41.67%	43.33%	26.67%
gaussian_noise	58.33%	48.33%	40.00%	25.00%
translate	71.67%	40.00%	43.33%	26.67%

A practical advantage of DenseNN is that keypoint features are largely texture-independent. Where MobileNetV2 and ResNet50 rely on visual patterns that shift between training and real-world images, DenseNN's geometric features remain stable. Hand and pose keypoints describe what the body is doing, not what it looks like, making them naturally invariant to lighting, background, and appearance variations that confuse pixel-based CNNs. ResNet50 remains impractical for typical web delivery due to size; MobileNetV2 pays a larger browser slowdown and download cost; YOLOv11-Pose is fast but trails on accuracy in this evaluation.

6.1.2 Training Curve Analysis

6.1.2.1 MobileNetV2 (transfer learning)

MobileNetV2 used two-phase transfer learning. Phase 1 froze the ImageNet-pretrained base and trained only the custom dense head (learning rate 1×10^{-3}). Phase 2 unfroze the last 30 layers for fine-tuning at 1×10^{-4}. Mixed precision (`mixed_float16`) was used throughout. Training curves show rapid early convergence in Phase 1 with early stopping after validation loss plateaued, then incremental gains in Phase 2. Training and validation accuracy tracked closely, suggesting that L2 regularization, BatchNormalization, and dropout ($0.4 \rightarrow 0.3 \rightarrow 0.2$) limited overfitting during training.

6.1.2.2 ResNet50 (deep transfer learning)

ResNet50 followed a similar two-phase schedule with deeper unfreezing (last 50 layers). Training used categorical cross-entropy with one-hot labels, TFRecord input, and MirroredStrategy for multi-GPU runs. Phase 1 warmup used 1×10^{-3} for five epochs; Phase 2 used 1×10^{-5}. Gradient clipping (`clipnorm` $= 1.0$) stabilized updates. Curves show steady validation improvement over many epochs.

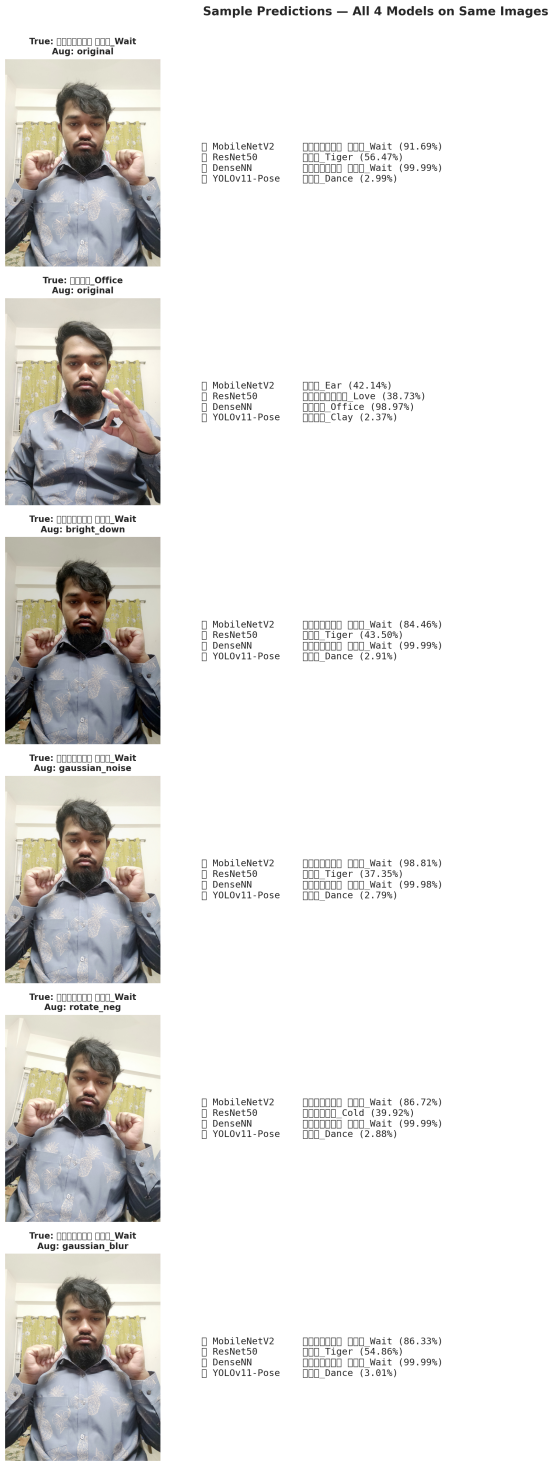


Figure 6.3: Qualitative sample predictions showing DenseNN consistently produces high-confidence correct predictions across augmentations, while CNNs vary widely.

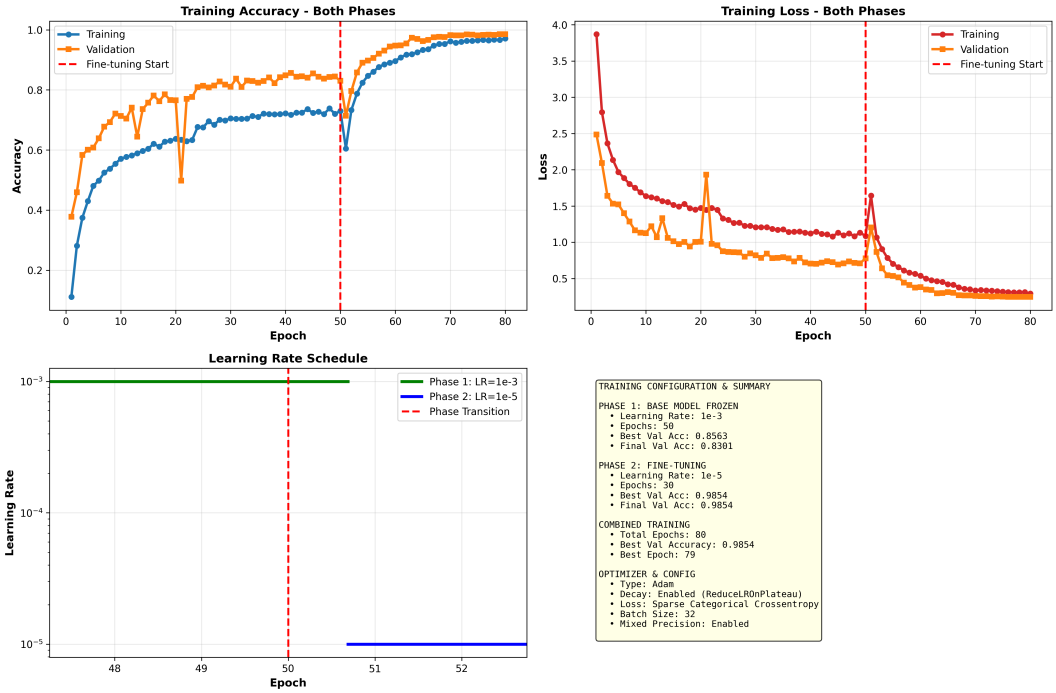


Figure 6.4: MobileNetV2 training curves.

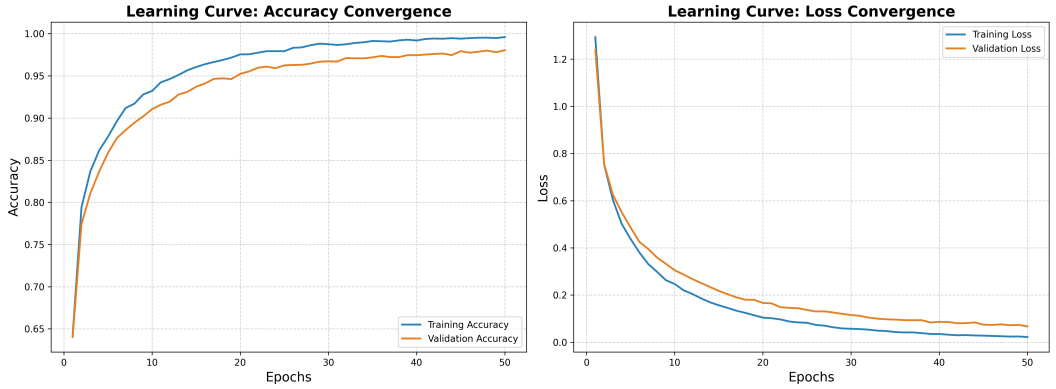


Figure 6.5: ResNet50 training curves.

6.1.2.3 DenseNN (keypoint-based MLP)

DenseNN trains on a 154-dimensional vector (126 hand keypoints, 27 pose keypoints, one `has_pose` flag). An improved configuration used reduced dropout ($0.3 \rightarrow 0.2 \rightarrow 0.15$), weak L2 ($1 \times 10^{-4}$), batch size 64, initial learning rate 8×10^{-4}, and early-stopping patience 15 epochs. Augmentation (mirror with hand-slot swap and coordinate jitter) applied only after the train/validation/test split. Curves show fast convergence with tight train–validation alignment.

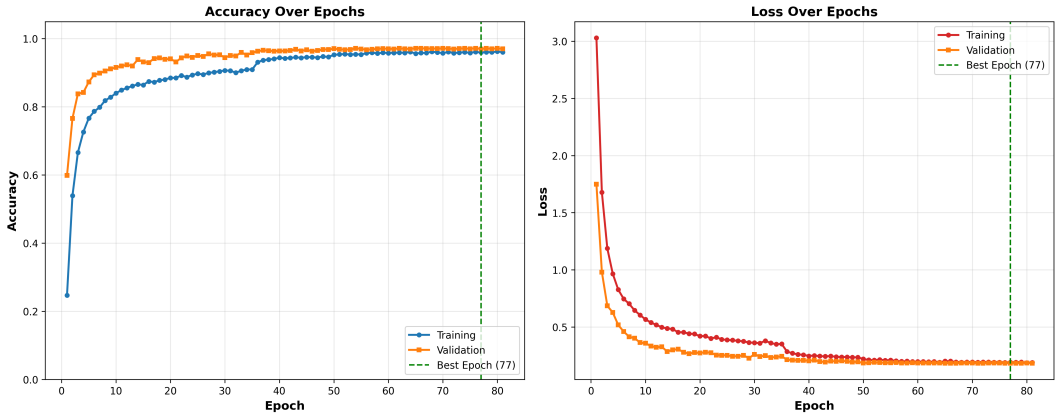


Figure 6.6: DenseNN training curves.

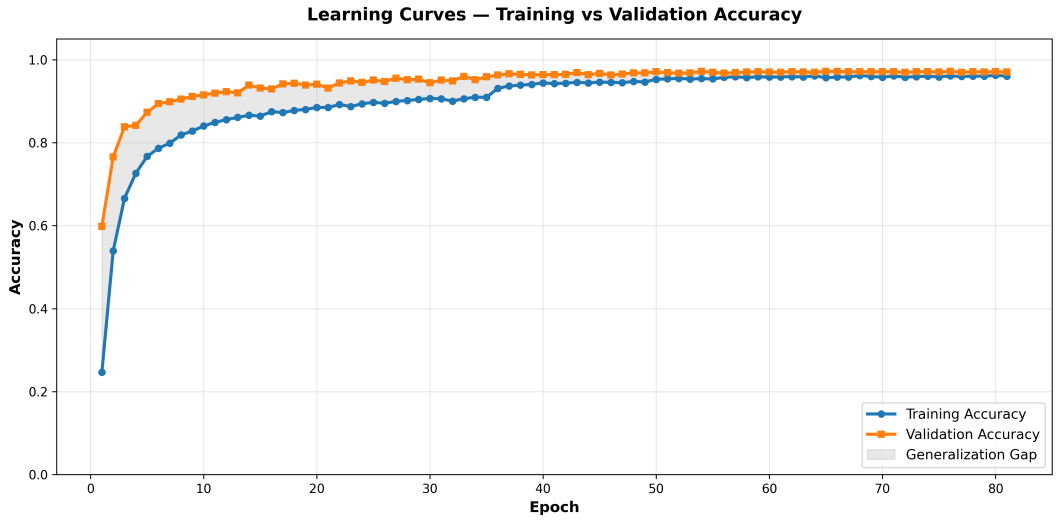


Figure 6.7: DenseNN learning curves with generalization gap visualization.

The keypoint feature space supports stronger field robustness because geometry transfers across signers and imaging conditions more reliably than raw texture for this task.

6.1.2.4 YOLOv11-Pose + MediaPipe (hybrid)

The hybrid combines Ultralytics YOLOv11n-Pose and MediaPipe Hands into a 189-D feature vector. Two MLP variants were trained: a baseline without heavy regularization and a regularized variant with BatchNorm, dropout (0.4), L2 decay (1×10^{-3}), label smoothing (0.1), and $3\times$ data augmentation. Regularization narrowed train–validation gaps versus the baseline.

Despite regularization, deployment accuracy lagged in this study because of the broader 131-class output head, mixed conventions between extractors, and lack of the pose-relative normalization used in the DenseNN pipeline.

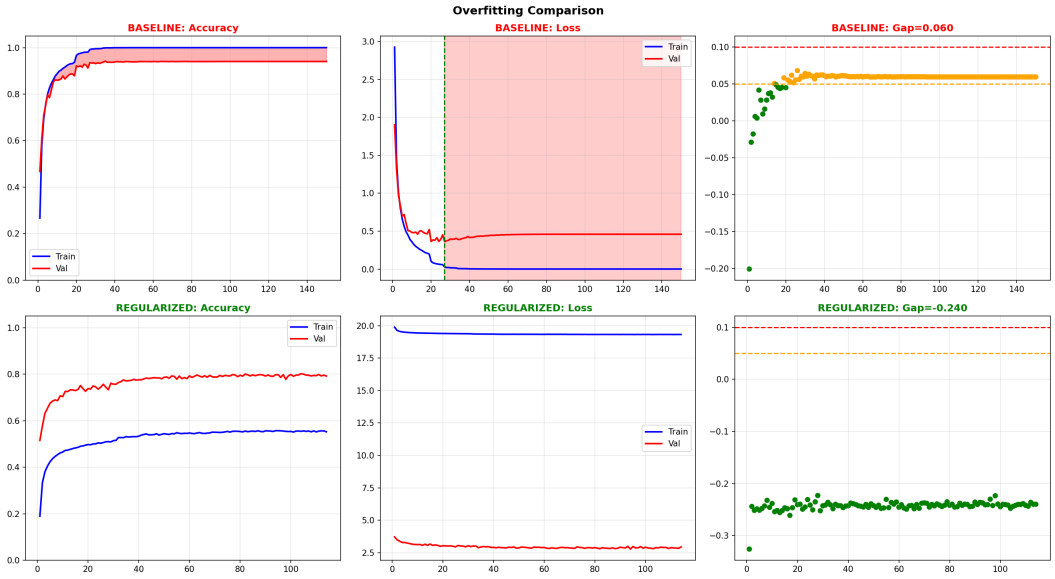


Figure 6.8: YOLOv11-Pose hybrid training curves (baseline vs. regularized).

6.2 Expected Result

Several outcomes were anticipated before evaluation:

Pretrained CNN models (MobileNetV2 and ResNet50) were expected to achieve high accuracy on training-distribution data given their ImageNet initialization. This was confirmed during training. However, on real-world data outside the training distribution, both CNNs scored substantially lower, 42.00% and 41.17% respectively, revealing reliance on memorized visual patterns rather than transferable concepts.

The keypoint-based DenseNN model was expected to be more robust to texture-level variations such as lighting, background, and signer appearance. This was strongly confirmed that DenseNN achieved 68.00% real-world accuracy, over 1.6× the best CNN, and scored 100% on motion-blurred images.

Smaller models were expected to deliver lower latency in browser-based deployment compared to large CNNs. Confirmed that DenseNN's 1.70 MB and YOLOv11-Pose's 0.39 MB are suitable for instant web loading, while ResNet50's 370 MB is impractical for the web.

Detection success rate was expected to be 100% for image-based CNNs and below 100% for keypoint-based models. Confirmed that DenseNN failed to extract features in 3% of cases under heavy distortions, while CNN models always produced an output, though often incorrect.

Deployment performance was expected to depend on the joint effect of accuracy, size, latency, and reliability, not raw accuracy alone. This was confirmed by the practical deployment observation that DenseNN provides the most accurate, smoothest, and most reliable user experience.

In summary, the evaluation results shows that keypoint-based geometric features generalize substantially better to real-world deployment conditions than pixel-based CNN features trained on a single dataset. DenseNN's 68% accuracy is not just numerically the highest, it represents a fundamentally more robust architectural choice for this task, validating the deployment decision both quantitatively and conceptually.

Chapter 7

Conclusion

7.1 Social, Legal, and Ethical Issues

7.1.1 Social Impact and Inclusivity

The system addresses a significant social challenge by reducing communication barriers between deaf or hard-of-hearing individuals and hearing users. By enabling real-time, bidirectional communication without dependence on human interpreters, the system promotes inclusion, independence, and equal participation in social, educational, and professional environments. The inclusion of Bengali language support ensures relevance to local communities and aligns with **SDG 10: Reduced Inequalities** and **SDG 4: Quality Education**, contributing to more equitable access to essential services.

7.1.2 Design for Safety and Hazard Analysis

Design for Safety principles were incorporated throughout the development process to ensure public safety and minimize risks. A qualitative Hazards and Failure Analysis (HFA) identified potential risks such as misinterpretation of signs or speech, unintended system activation, and user confusion during real-time interaction. To mitigate these hazards, the system enforces turn-based communication, provides simultaneous text and sign outputs for verification, and offers clear visual feedback during recognition and translation. These measures reduce the likelihood of miscommunication and improve system transparency, particularly in sensitive usage scenarios.

7.1.3 Ethical Considerations

Ethical responsibility guided the design to ensure fairness, accuracy, and respect for user autonomy. The system avoids exclusive reliance on automated output by displaying both textual and visual representations, allowing users to independently validate translations. Dataset bias, regional sign variation, and cultural sensitivity were identified as ethical concerns; therefore, the system architecture remains modular to allow future dataset expansion and community-informed validation. The solution is positioned as an assistive tool rather than a replacement for human interpreters in high-risk contexts, maintaining ethical boundaries in deployment.

7.1.4 Legal and Regulatory Compliance

The system processes audio and video data, which are subject to privacy and data protection regulations. Legal compliance is addressed by requiring explicit user consent before accessing camera and microphone resources and by limiting data handling to real-time processing without unnecessary storage. These practices align with international data protection principles such as GDPR, including data minimization and purpose limitation, as well as relevant national ICT and privacy guidelines applicable in Bangladesh. Proper licensing and attribution are maintained for datasets and software components used in the system.

7.2 Environment and Sustainability Impact

7.2.1 Environmental Impact

The proposed system minimizes environmental impact by adopting a software-centric, web-based architecture that operates on existing consumer devices. By eliminating the need for specialized hardware, the system helps reduce electronic waste and limits the environmental burden associated with manufacturing, transportation, and disposal of dedicated assistive devices. This approach aligns with environmentally responsible engineering practices and supports sustainable consumption patterns.

7.2.2 Energy Efficiency and Resource Utilization

Energy efficiency was considered during system design by emphasizing lightweight computation and on-demand processing. System resources are utilized only during active user interaction, reducing unnecessary energy consumption. The modular architecture further enables future optimization and integration of more energy-efficient models, supporting long-term sustainable operation.

7.2.3 Sustainability and Social Development Goals (SDGs)

The project contributes directly and indirectly to several United Nations Sustainable Development Goals (SDGs). It aligns with **SDG 10: Reduced Inequalities** by promoting inclusive communication for deaf and hard-of-hearing individuals, enabling equitable access to education, healthcare, and public services. The system also supports **SDG 4: Quality Education** by facilitating accessible communication in learning environments without dependence on human interpreters. Additionally, the use of digital, hardware-independent solutions contributes to **SDG 12: Responsible Consumption and Production** by reducing electronic waste and encouraging sustainable use of existing technological resources. Through inclusive design and sustainable engineering choices, the project demonstrates a commitment to both social equity and environmental responsibility.

7.3 Limitations

7.3.1 Challenges

During the project, several problems came up. Some were fixed, some still remain. Here's a simple breakdown of each:

1. Not Enough Memory for the Dataset

The dataset was 141 GB, but Kaggle only gave 30 GB of RAM, and a big chunk of that was already being used by TensorFlow and other tools. So loading even a small portion of images at once wasn't possible.

The fix was to load one image at a time, process it, then throw it away before loading the next. It worked, but it meant everything had to run one by one, no parallel processing, because doing that would have eaten up even more memory.

2. MediaPipe Couldn't Use the GPU on Kaggle

The plan was to use the GPU to speed up keypoint extraction, but Kaggle doesn't have the display system MediaPipe needs to run on GPU. It threw an error and crashed.

The fix was simple, wrap it in a try/except: try GPU first, and if it fails, fall back to CPU. On Kaggle it runs on CPU (which takes 30,60 minutes), but on a proper local machine with GPU support, it runs faster automatically.

3. Mixed Precision Broke the TFLite Export

Mixed precision training saves weights in float16 to speed things up. But when exporting to TFLite, it caused an error because TFLite doesn't natively support float16 for certain operations.

The fix was to create a copy of the model in float32 at export time, transfer the trained weights into it, and then convert that copy to TFLite. It adds a bit of extra code but the result is a clean, tiny 0.09 MB TFLite file with no issues.

4. A Typo Bug in the JavaScript Code

In the JavaScript inference code, the hand sorting logic was reading `handResults.handednesses` (with an "s" at the end), but the actual MediaPipe API returns `handResults.handedness` (no "s"). Because of this typo, the sorting never actually ran.

This is a tricky bug because the model still gives predictions, it just silently uses the wrong hand order for two-handed signs, which hurts accuracy. The fix is just renaming that one property, but it's a good reminder of how easy it is to mess things up when training code and inference code are written in two different languages.

5. The Model Only Looks at One Frame at a Time

The biggest limitation is that the model looks at single images, it has no sense of movement or time. But sign language is all about motion. Many signs look identical in a single frame but are totally different when you see them moving.

The dataset itself only has static images, so even redesigning the model for video wouldn't help without collecting a whole new video dataset. This is why some of the worst-performing classes, like "Your," "Mathematics," "Wood," "Stop," and "Doctor," are probably getting confused with each other. They might look the same in a still photo but differ in how they move.

6. The Reference-Point Trade-off

When pose (body) landmarks are detected, the hand position is calculated relative to the nose. When they're not, it falls back to the wrist. This was a deliberate choice.

The cleaner option would have been to just skip any image where the full body isn't visible, but that would have thrown away a lot of useful data, especially for close-up shots where only the hands are in frame. A `has_pose` flag was added so the model at least knows which reference point was used, helping it make sense of the difference.

7.3.2 Limitations

Despite achieving reliable real-time performance within its intended scope, the system has a number of limitations that are worth acknowledging.

Restricted vocabulary coverage. The gesture recognition model was trained on **60 sign classes**, and the speech-to-sign avatar library contains **51 entries**. This represents a practical but narrow subset of Bangla Sign Language (BdSL), which encompasses several thousand signs in everyday use. Words or gestures outside this vocabulary are either misclassified to the nearest known class or left unmatched, meaning the system cannot yet serve as a general-purpose communication tool without significant vocabulary expansion.

Static avatars cannot convey motion. The speech-to-sign direction displays a single static PNG image per word. Many BdSL signs are defined primarily by the trajectory and speed of hand movement rather than by a fixed hand shape alone. A still image necessarily omits this dynamic information, which can reduce comprehensibility for Deaf viewers who rely on the full motion of a sign to distinguish it from phonetically similar signs.

Sensitivity to lighting and background conditions. The MediaPipe Hand and Pose Landmarkers, while robust under normal conditions, degrade noticeably in low-light environments, against cluttered or high-contrast backgrounds, or when the signer's skin tone is poorly represented in the camera's exposure settings. The classification model inherits this sensitivity, as its input features are derived directly from the quality of landmark detection. No preprocessing step (e.g. background subtraction or adaptive exposure correction) is currently applied.

Single-signer assumption. Although a primary-person filter is implemented to discard hands that are too far from the pose-estimated wrists, the filter is proximity-based and heuristic. In busy or crowded environments, the system may still occasionally incorporate landmark data from bystanders, particularly if they walk close behind the primary signer. More robust multi-person tracking or an explicit bounding-box based person ID would be needed to eliminate this reliably.

Rule-based morphological matching has gaps. The speech-to-sign text processing pipeline strips a fixed list of common Bengali suffixes and checks a manually curated variation dictionary. This approach covers a large share of everyday conjugations but is not exhaustive. Less common verb forms, dialectal variations, or words with irregular morphology may fail to match any avatar, falling back to the text or neutral-pose display instead. A proper morphological analyser for Bengali would significantly improve coverage.

Browser and hardware dependency. The application depends on features that are not universally supported: the Web Speech API for voice input is absent in Firefox and some mobile browsers; WebGL GPU delegation for MediaPipe requires a compatible graphics driver. The requestAnimation-based inference assumes a reasonably capable CPU/GPU to sustain real-time throughput. On low-end devices or unsupported browsers, one or more features will silently degrade or become unavailable.

References

[1] Aslam, A. M., Aabidh, M. A., & Fatima, N. S. (2025). Real-time Indian Sign Language recognition & multilingual sign generation. In *Proceedings of the third international conference on augmented intelligence and sustainable systems (icaiss-2025)* (pp. 1658–1666). IEEE. Retrieved from https://doi.org/10.1109/ICAISS61471.2025.11041851

[2] Baranidharan, B., Chatterjee, S., Jha, A., Patwardhan, P., & Ghosh, J. (2024). Bengali Sign Language detection using deep learning techniques. In *2024 15th international conference on computing, communication and networking technologies (icccnt)*. IEEE. Retrieved from https://doi.org/10.1109/ICCCNT61001.2024.10723887

[3] Boobal, A., Jasmine, J. L., Reddy, C. C. K., & Reddy, C. A. (2024). Real-time sign language and audio conversion using ai. In *2024 international conference on communication, control, and intelligent systems (ccis)*. IEEE. Retrieved from https://doi.org/10.1109/CCIS63231.2024.10931891

[4] Deshmukh, A., Machindar, A., Lale, S., & Kasambe, P. (2024). Enhancing communication for the hearing impaired: A real-time speech to sign language converter. In *2024 27th international symposium on wireless personal multimedia communications (wpmc)*. IEEE. Retrieved from https://doi.org/10.1109/WPMC63271.2024.10863135

[5] Francis, R. (2024-08). Sign2text: Deep learning-based sign language translation system using vision transformers and phi-1.5b. In *2024 ieee international conference on artificial intelligence in engineering and technology (iicaiet)* (pp. 282–287). IEEE. Retrieved from https://doi.org/10.1109/IICAIET62352.2024.10730290

[6] Haque, A., Pulok, R. A., Rahman, M. M., Akter, S., Khan, N., & Haque, S. (2023). Recognition of bangladeshi sign language (bdsl) words using deep convolutional neural networks (dcnns). , *7*(6), 1666–1677. Retrieved from https://doi.org/10.28991/ESJ-2023-07-06-019

[7] Islam, M. A., Karim, A., Rahaman, M. A., et al. (2022). Automatic 3d animated bangla sign language gestures generation from bangla text and voice. In *2022 4th international conference on sustainable technologies for industry 4.0 (sti)*. IEEE. Retrieved from https://doi.org/10.1109/STI56238.2022.10103252

[8] Jeevanandham, P., Hariharan, A., & Keerthana, G. (2024-08). Real-time hand sign language translation: Text and speech conversion. In *2024 7th international conference on circuit power and computing technologies (iccpct)* (pp. 488–493). IEEE. Retrieved from https://doi.org/10.1109/ICCPCT61902.2024.10673038

[9] Kambhampati, S. S., Mehnaaz, Biradar, N., Penumathsa, L. V., & Uddagiri, C. (2024). Sign language recognition and translation systems for enhanced communication for the hearing impaired. In *2024 1st international conference on cognitive, green and ubiquitous computing (ic-cgu)*. IEEE. Retrieved from https://doi.org/10.1109/IC-CGU58078.2024.10530832

[10] Luqman, H. (2022). An efficient two-stream network for isolated sign language recognition using accumulative video motion. , *10*, 94507–94521. Retrieved from https://doi.org/10.1109/ACCESS.2022.3204110

[11] Priyadharshini, D. S., Anandraj, R., Ganesh Prasath, K. R., & Franklin Manogar, S. A. (2024). A comprehensive application for Sign Language alphabet and word recognition, text-to-action conversion for learners, multi-language support and integrated voice output functionality. In *2024 international conference on science, technology, engineering and management (icstem)*. IEEE. Retrieved from https://doi.org/10.1109/ICSTEM61137.2024.10561024

[12] Rai, D., Rana, N., Kotak, N., & Sharma, M. (2024). Real-time speech to sign language translation using machine and deep learning. In *2024 11th international conference on reliability, infocom technologies and optimization (icrito)*. IEEE. Retrieved from https://doi.org/10.1109/ICRITO61523.2024.10522437

[13] Rajkumar, T., Dharani, J., Kavitha, A., Hemaa Pradhiksaa, K. M., Jashima Fathima, R., & Neha, S. (2023). AI-based two-way sign language system. In *Proceedings of the second international conference on electronics and renewable systems (icears-2023)* (pp. 984–991). IEEE. Retrieved from https://doi.org/10.1109/ICEARS56392.2023.10085670

[14] Ratnasingam, S., Sakajarasa, D., Chamara, D., & Gamage, A. I. (2024-12). Developing accurate sri lankan sign language to tamil vocal and american sign language translation. In *2024 international conference on engineering and emerging technologies (iceet)* (pp. 1–6). IEEE. Retrieved from https://doi.org/10.1109/ICEET65156.2024.10913736

[15] Saini, T., & Kumari, N. (2024-03). Signaspectrum: Ai-driven dynamic sign language detection and interpretation. In *2024 11th international conference on reliability, infocom technologies and optimization (trends and future directions) (icrito)* (pp. 1–6). IEEE. Retrieved from https://doi.org/10.1109/ICRITO61523.2024.10522423

[16] Siddique, S., Islam, S., Neon, E. E., Sabbir, T., Naheen, I. T., & Khan, R. (2023). Deep learning-based bangla sign language detection with an edge device. , *18*, 200224. Retrieved from https://doi.org/10.1016/j.iswa.2023.200224

[17] Sindhu, K. S., Nikitha, B., Varma, P. L., & Uddagiri, C. (2024-03). Sign language recognition and translation systems for enhanced communication for the hearing impaired. In *2024 1st international conference on cognitive, green and ubiquitous computing (ic-cgu)* (pp. 1–6). IEEE. Retrieved from https://doi.org/10.1109/IC-CGU58078.2024.10530832

[18] Tanni, K. F., Islam, S., Sultana, Z., Alam, T., & Mahmood, M. (2024-12). Deepbdsl: A comprehensive assessment of deep learning architectures for multiclass bengali sign language gesture recognition. In *2024 27th international conference on computer and information technology (iccit)* (pp. 2219–2224). IEEE. Retrieved from https://doi.org/10.1109/ICCIT64611.2024.11022054