

Dissertation/Report submitted to the
UNIVERSITY OF MOHAMED BOUDIAF - M'SILA, ALGERIA



FACULTY OF MATHEMATICS AND COMPUTER SCIENCE
DEPARTMENT OF COMPUTER SCIENCE

in partial fulfillment of the requirements for the degree of
Master's Degree in Computer Science

Specialization: Computer Science

Presented by

BELOUADAH Makhoulf

Entitled

Customized Neural Network for Hand Gesture Recognition
Design and Implementation

Under the supervision of

DR. BAHACHE Mohamed

Jury Members

Dr. Lakehal Meftah ... University of M'sila - President

Dr. Mokhtari Rabah ... University of M'sila - Examiner

June, 2026

الملخص

تعرض هذه الأطروحة نظاماً للتعرف على إيماءات اليد يجمع بين استخراج معالم اليد وتصنيفها باستخدام نموذج عصبي مدمج. وتركز الدراسة على تطوير حل فعال للاستدلال في الزمن الحقيقي من خلال مقارنة مناهج متعددة تشمل النماذج المعتمدة على المعالم والنماذج المعتمدة على الصور. وتُظهر النتائج أن التمثيل القائم على معالم اليد، عند دمجه مع مصنف مناسب، يوفر توازناً جيداً بين الدقة والكفاءة وسهولة النشر في التطبيقات التفاعلية. كما توضح الأطروحة كيف يمكن دمج هذا النهج داخل بنية تطبيق حديثة تُخدم الاستخدام العملي للتعرف على الإيماءات.

الكلمات المفتاحية: التعرف على إيماءات اليد، معالم اليد، الشبكات العصبية، الاستدلال في الزمن الحقيقي، الرؤية الحاسوبية.

Abstract

This thesis presents a hand gesture recognition system that combines hand-landmark extraction with compact neural classification. The study focuses on building an efficient solution for real-time gesture understanding through a comparison of different landmark-based and image-based approaches. The findings show that a landmark-driven representation, when paired with an appropriate classifier, offers a strong balance between accuracy, efficiency, and practical deployability in interactive applications. The thesis also shows how this approach can be integrated into a modern application architecture for real-world hand gesture recognition use.

Keywords: Hand Gesture Recognition, hand landmarks, neural networks, real-time inference, computer vision.

Dedication

*In the name of **Allah**, the Most Gracious, the Most Merciful.
I dedicate this work first to **Allah**, whose guidance, mercy, and
strength have carried me through every stage of this academic
journey.*

*To my beloved parents, whose prayers, sacrifices, patience,
and unconditional love have been the foundation of everything
I have achieved. Their faith in me gave me the courage to
continue when the path became difficult.*

*To my family, whose encouragement and presence have always
reminded me that I was never alone in this journey.*

*To my teachers, friends, and everyone who supported me with
a word, an idea, or a moment of help, I offer my sincere
gratitude.*

This thesis is dedicated to all of you.

Acknowledgements

First and foremost, I thank **Allah**, the Almighty, for granting me the patience, strength, and perseverance required to complete this thesis. Every step of this work was made possible by His guidance and mercy.

I would like to express my sincere gratitude to my supervisor, **BAHACHE Mohamed**, for his guidance, support, and valuable advice throughout the preparation of this work. His feedback helped me improve the quality of the research and remain focused on the objectives of the project.

I also extend my respectful thanks to the members of the jury for accepting to examine and evaluate this thesis. Their time, remarks, and academic assessment are deeply appreciated.

My thanks go as well to the teachers and staff of the Department of Computer Science at the University of Mohamed Boudiaf - M'Sila for the knowledge, assistance, and academic environment they provided during my studies.

I am especially grateful to my parents and family for their constant prayers, encouragement, and sacrifices. Their support has been a source of motivation and confidence throughout my academic path.

I would also like to thank my friends and classmates for their encouragement, discussions, and companionship during this journey. Their presence made the difficult moments easier and the successful moments more meaningful.

Finally, I acknowledge everyone who contributed directly or indirectly to the completion of this thesis, including those whose advice, feedback, or technical assistance helped shape this work.

Contents

General Introduction	2
1 Foundations of Hand Gesture Recognition and Machine Learning	5
1.1 Introduction	5
1.2 Background of Hand Gesture Recognition	5
1.2.1 Static and Dynamic Gestures	5
1.2.2 Glove-Based, Vision-Based, and Hybrid Systems	6
1.3 General Gesture Recognition Pipeline	7
1.3.1 Acquisition	8
1.3.2 Preprocessing	8
1.3.3 Feature Extraction	8
1.3.4 Classification	9
1.4 Machine Learning and Learning Paradigms	9
1.4.1 Supervised Learning	10
1.4.2 Unsupervised Learning	10
1.4.3 Semi-Supervised Learning	10
1.4.4 Reinforcement Learning	10
1.5 Core Model Families for Gesture Classification	10
1.5.1 Multilayer Perceptrons	11
1.5.2 Convolutional Neural Networks	11
1.5.3 k-Nearest Neighbours	12
1.5.4 Support Vector Machines	13
1.5.5 Random Forests	14
1.5.6 Transformers and Graph Neural Networks	15
1.6 Activation Functions in Neural Networks	17
1.6.1 ReLU	18
1.6.2 Sigmoid	18
1.6.3 Hyperbolic Tangent	18
1.6.4 Softmax	19
1.7 Conclusion	19
2 State of the Art in Hand Gesture Recognition	21
2.1 Introduction	21

2.2	Applications and Motivation	23
2.2.1	Human-Computer Interaction	23
2.2.2	Augmented and Virtual Reality	23
2.2.3	Sign Language Recognition	23
2.2.4	Accessibility, Robotics, and Smart Environments	24
2.3	Main Technical Challenges	24
2.4	State of the Art	26
2.4.1	Glove-Based and Wearable Sensor Approaches	26
2.4.2	Vision-Based Approaches: The Processing Pipeline	27
2.4.3	Landmark-Based Pipelines	28
2.4.4	CNN-Based Approaches	29
2.4.5	Transformer-Based Approaches	29
2.4.6	Hybrid Sensor-Vision Approaches	29
2.4.7	Comparing the Main Architectures	30
2.4.8	Classification Methods and Reported Performance	31
2.5	Key Contradictions in the Literature	32
2.6	How Research in This Field Is Conducted	33
2.7	Open Research Challenges	34
2.8	The Position of This Thesis	36
2.9	Conclusion	36
3	Design, Implementation, and Experimental Evaluation	39
3.1	Introduction	39
3.2	The Dataset Problem	39
3.3	Overall Pipeline Design	40
3.4	Dataset Construction	41
3.4.1	Image Acquisition with icrawler	41
3.4.2	Hand Detection and Auto-Labeling with YOLO-HaGRID	42
3.4.3	Cropping and Storage	43
3.4.4	Summary	44
3.5	Input Representation Design	44
3.5.1	The Precomputed Landmark Architecture	45
3.5.2	Pixel-Based Input	45
3.5.3	Landmark-Based Input	45
3.6	Experimental Setup	46
3.6.1	Train, Validation, and Test Splits	46
3.6.2	Training Fractions	46
3.6.3	Total Experiments	47

3.6.4	Shared Training Defaults	47
3.6.5	What Was Measured	47
3.7	Model Design and Implementation	48
3.7.1	Models Operating on Raw Pixels	48
3.7.2	Models Operating on MediaPipe Landmarks	51
3.8	Results at the Full Training Budget	53
3.9	Speed, Size, and the Cost of Accuracy	57
3.10	Behavior Under Reduced Data	58
3.11	Error Analysis of the Selected Model	60
3.12	The Implementation Decision	62
3.13	Conclusion	62
4	Application Development and System Architecture	65
4.1	Introduction	65
4.2	The Path to the Final Stack	65
4.2.1	Phase One: Python and Tkinter	65
4.2.2	Phase Two: FastAPI and Jinja2	66
4.2.3	Phase Three: Next.js and FastAPI	67
4.2.4	The Decision to Go Desktop	68
4.3	The Final Technology Stack	68
4.4	Overall System Architecture	69
4.4.1	Data Storage	70
4.5	Authentication: Getting In	71
4.6	The Dashboard	71
4.7	Gesture Labels	72
4.8	Capturing Training Data	72
4.9	Training a Model	74
4.9.1	Configuring the Network	74
4.9.2	The Training Log	74
4.9.3	Training Results and Reports	75
4.9.4	The 3D Neural Network Visualisation	76
4.10	Translation: Using the Model	77
4.11	Development Environment	79
4.12	Conclusion	80
	General Conclusion	82
	References	84

Figure Sources	87
A Appendix A: List of Acronyms	89
Appendix A: List of Acronyms	89

List of Tables

1.1	Comparison of glove-based, vision-based, and hybrid HGR systems [2–4, 23, 25].	7
2.1	The three types of gesture: what they are, how systems typically recognise them, and the core engineering trade-off each imposes.	22
2.2	Core technical challenges in HGR: what goes wrong, what helps, and how serious each challenge remains.	24
2.3	Sensing modalities for HGR: how each captures the hand, what it costs, and where it performs best. Accuracy ranges are drawn from the reviewed literature.	26
2.4	Architecture comparison for real-time HGR on modest hardware.	30
2.5	Five direct contradictions across the reviewed literature: what is disputed, who takes which position, and what causes the disagreement.	32
2.6	Methodology types across the 42 reviewed papers: frequency, prevalence, and key limitations.	33
2.7	The five most significant research gaps identified from the reviewed literature, ordered by risk to existing conclusions.	34
3.1	Summary of the constructed dataset properties.	44
3.2	Actual training-set sizes at each evaluated fraction under the stratified sub-sampling protocol.	47
3.3	Full-split accuracy, F1 scores, training time, inference latency, and parameter count for all eleven configurations, sorted by accuracy.	54
3.5	MediaPipe + MLP performance across six training fractions from 1% to 100%.	58
4.1	Final technology stack.	68
4.2	End-to-end application flow.	79

List of Figures

1.1	Examples of glove-based and landmark-based sensing modalities in HGR systems	6
1.2	Generic hand gesture recognition pipeline across acquisition, preprocessing, extraction, and classification	8
1.3	Overview of supervised, unsupervised, and reinforcement learning paradigms	9
1.4	Overview of a multilayer perceptron architecture	11
1.5	Convolutional neural network processing from local filtering to final classification	12
1.6	Illustration of k-NN classification based on neighboring samples	13
1.7	Illustration of SVM classification with a maximum-margin hyperplane . . .	14
1.8	Random forest ensemble of decision trees for classification	15
1.9	Vision transformer pipeline using patch tokens and stacked self-attention layers	16
1.10	Graph neural network message passing through neighborhood aggregation and updates	17
1.11	Overview of common activation functions used in neural networks	18
3.1	Dataset organization and sample images. The dataset is stored as one folder per gesture class, with roughly 30,000 samples per class; the dislike folder screenshot illustrates the kind of cropped hand-gesture examples used during training.	43
3.2	Accuracy and Macro-F1 of all eleven configurations at the full training split.	55
3.3	Full-split accuracy distribution by model family.	56
3.4	Accuracy gap of each configuration relative to the best measured result. . .	56
3.5	Accuracy versus classifier-stage inference latency at the full split.	57
3.6	Training-fraction accuracy for the strongest representative of each model family.	59
3.7	Full accuracy matrix across all eleven models and six training fractions. . .	60
3.8	Confusion matrix of the selected MediaPipe + MLP model at the full training split	61
4.1	Phase one Python and Tkinter prototype.	66
4.2	Phase two FastAPI and Jinja2 prototype.	67
4.3	Desktop application architecture.	70
4.4	Authentication screen.	71

4.5	Gesture label management.	72
4.6	Training data capture.	73
4.7	Training configuration panel.	74
4.8	Live training log.	75
4.9	Training results report.	76
4.10	3D neural network visualisation.	77
4.11	Translation interface.	78

General Introduction

General Introduction

Hand Gesture Recognition (HGR) has emerged as an important research direction in computer vision and intelligent interaction because it enables machines to understand one of the most natural forms of human communication. Gestures can be used to control interfaces without physical contact, complement speech in noisy or privacy-sensitive environments, and provide an accessible interaction channel for users who benefit from visual communication. In practical terms, HGR is relevant to human-computer interaction, augmented and virtual reality, assistive technologies, robotics, industrial supervision, and smart environments [1–4].

Despite this relevance, reliable gesture recognition remains a challenging problem. Hand appearance changes continuously with viewpoint, scale, articulation, and individual morphology. Visual acquisition is further affected by illumination shifts, cluttered backgrounds, self-occlusion, motion blur, and partial visibility. These factors make it difficult to design models that remain both accurate and stable outside controlled laboratory conditions. The challenge becomes even more demanding when the target system must operate in real time with limited computational resources [5–8].

Recent advances in machine learning have produced several viable strategies for HGR. Pixel-based approaches such as Convolutional Neural Networks (CNNs) learn discriminative visual features directly from images, while transformer-based models exploit long-range interactions through attention mechanisms. At the same time, landmark-based pipelines have become especially attractive for embedded and interactive applications because they compress the visual signal into a compact geometric representation before classification. This reduction in input complexity can significantly lower computational cost without necessarily sacrificing practical recognition quality [9–17].

The central question addressed in this thesis is therefore the following: how can a custom neural network for Hand Gesture Recognition be designed and implemented so that it preserves strong classification performance while remaining lightweight enough for real-time use? To answer this question, the work combines conceptual analysis, methodological comparison, and data-driven evaluation based on a structured experimental campaign [18–20].

The study pursues the following objectives:

- to define the conceptual foundations, applications, and main challenges of Hand Ges-

ture Recognition (HGR);

- to review the principal learning paradigms and model families used for gesture classification, with special attention to landmark-based neural architectures;
- to analyze different experiments conducted across six training fractions and eleven base model configurations;
- to identify the most suitable solution for deployment by comparing accuracy, Macro-F1, training time, inference latency, and parameter count;
- to outline a modern software architecture for integrating the selected HGR pipeline into a full interactive application.

Methodologically, the thesis is organized into four complementary layers. The first focuses on methodology, including supervised and unsupervised learning perspectives, model selection, and the rationale for using hand landmarks as the primary representation. The second develops the theoretical background of gesture recognition and the evolution of relevant visual learning methods. The third is entirely data-driven: it interprets the experimental results, generates comparative tables and plots, and justifies the final design choice on empirical grounds. The last layer translates these findings into an application-level architecture based on contemporary frontend and backend technologies [18, 21, 22].

The importance of this research is both scientific and practical. From a scientific perspective, it provides a coherent comparison between heavy visual models and compact landmark-driven alternatives under a common experimental setting. From a practical perspective, it supports the design of an HGR solution that can be embedded in responsive interfaces where latency, memory footprint, and implementation simplicity matter as much as predictive quality. The final recommendation is not based on architectural preference alone, but on measurable trade-offs extracted from the available experimental evidence [14–17].

The remainder of the manuscript is organized as follows. Chapter 1 reviews the machine learning methodologies considered in the study. Chapter 2 presents the conceptual foundations and state of the art of Hand Gesture Recognition (HGR). Chapter 3 analyzes the experimental results and explains the implementation logic behind the selected solution. Chapter 4 describes the target application architecture and development environment. Finally, the general conclusion synthesizes the main contributions and highlights directions for future work.

Foundations of Hand Gesture Recognition and Machine Learning

Chapter 1

Foundations of Hand Gesture Recognition and Machine Learning

1.1 Introduction

Hand gesture recognition (HGR) studies computational methods that observe hand configurations or hand motions and assign them to semantic gesture classes. Within human-computer interaction, gestures are treated as natural input signals because they can express commands or communicative intent without conventional physical controllers. This chapter presents only the theoretical background required for the remainder of the thesis: the definition and categories of HGR, the generic recognition pipeline, the main learning paradigms in machine learning, representative model families, and the activation functions most commonly used in neural classifiers [1–4, 23].

1.2 Background of Hand Gesture Recognition

Hand gesture recognition can be defined as the process of capturing hand-related observations and mapping them to interpretable gesture labels. In practical terms, the input may be a single image, a video sequence, a depth stream, or a wearable-sensor signal, while the output is a discrete class or symbolic meaning associated with the observed gesture. The central objective is therefore to transform raw measurements of hand posture or hand motion into machine-interpretable decisions [1–3, 23].

1.2.1 Static and Dynamic Gestures

A static gesture is characterized by a hand posture observed at one instant, so recognition depends primarily on spatial configuration. A dynamic gesture is defined by a temporal evolution of hand states, which means that the order of motions across time contributes to the

gesture identity. This distinction is fundamental in HGR because static problems emphasize shape description, whereas dynamic problems require both spatial and temporal information to be represented explicitly [1, 2, 6, 23, 24].

1.2.2 Glove-Based, Vision-Based, and Hybrid Systems

HGR systems are also categorized according to the sensing modality used during acquisition. Glove-based systems rely on instrumented wearable devices that record hand articulation through sensor measurements such as flexion, joint motion, or inertial cues. Vision-based systems infer gestures from visual observations, including RGB or depth imagery captured by cameras. Hybrid systems combine two or more sensing sources within a single recognition framework so that complementary evidence from different modalities can be fused during recognition [2–4, 23, 25].

Glove-based systems are attractive because they obtain structured measurements directly from sensors and are therefore less dependent on scene conditions such as illumination or background clutter. Their principal limitation is the need for wearable hardware, which increases instrumentation cost and may reduce the naturalness of interaction. Vision-based systems, by contrast, support contact-free interaction and can exploit the rich spatial information contained in images, but they remain sensitive to occlusion, viewpoint variation, illumination changes, and the quality of hand detection or segmentation. Hybrid systems are intended to balance these trade-offs by combining complementary modalities, although this greater robustness is usually accompanied by additional hardware, calibration, synchronization, and fusion complexity [3, 4, 23, 25].

Figure 1.1 complements this comparison with representative sensing examples frequently used in the HGR literature: instrumented gloves for direct articulation sensing and landmark-based visual hand representations for camera-driven recognition [2, 3, 14].

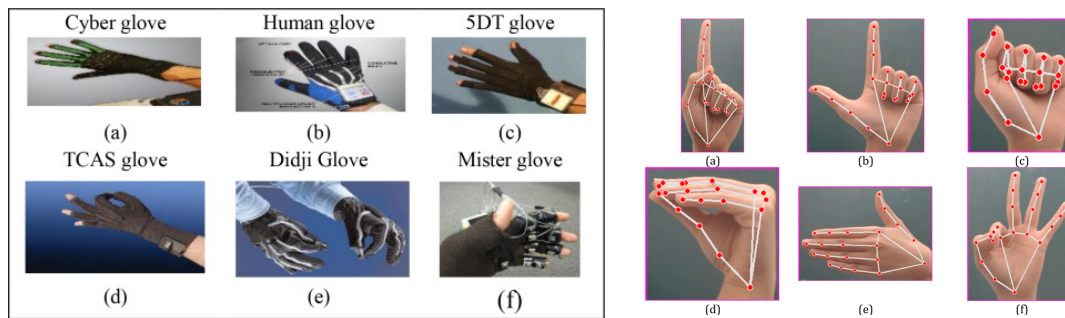


Figure 1.1: Examples of glove-based and landmark-based sensing modalities in HGR systems. *Sources:* left image from ScienceDirect [39]; right image from ResearchGate [40].

Table 1.1 summarizes the three system categories at a high level in terms of sensing principle, typical strengths, and typical limitations [2–4, 23, 25].

Table 1.1: Comparison of glove-based, vision-based, and hybrid HGR systems [2–4, 23, 25].

Aspect	Glove-based	Vision-based	Hybrid
Sensing modality	Wearable sensors (flex, IMU, bend sensors)	RGB / depth cameras	Fusion of wearable and vision sensors
Interaction type	Invasive (device worn on hand)	Non-contact	Semi-contact or multi-modal
Signal quality	High precision, structured signals	Dependent on environment conditions	Improved robustness via complementary data
Environmental sensitivity	Low (robust to lighting/background)	High (affected by occlusion, lighting, clutter)	Medium (depends on fusion strategy)
User comfort	Lower due to wearability constraints	High (natural interaction)	Moderate (depends on setup)
System complexity	Hardware-dependent but simpler processing	Heavier computation (vision processing)	Highest (fusion, synchronization, calibration)
Cost	Medium to high (specialized hardware required)	Low to medium (standard cameras)	High (combined hardware and integration cost)
Typical limitation	Requires hardware setup and calibration	Occlusion and viewpoint issues	Integration and synchronization overhead

1.3 General Gesture Recognition Pipeline

Although concrete implementations differ, gesture-recognition systems are commonly described through four generic stages: acquisition, preprocessing, feature extraction, and classification. This abstraction is broad enough to cover glove-based, vision-based, and hybrid systems while preserving the main computational steps shared across them [1–4, 23].

Figure 1.2 provides a compact visual summary of this generic HGR workflow, linking the data-acquisition stage to the successive preprocessing, representation, and decision stages described below.

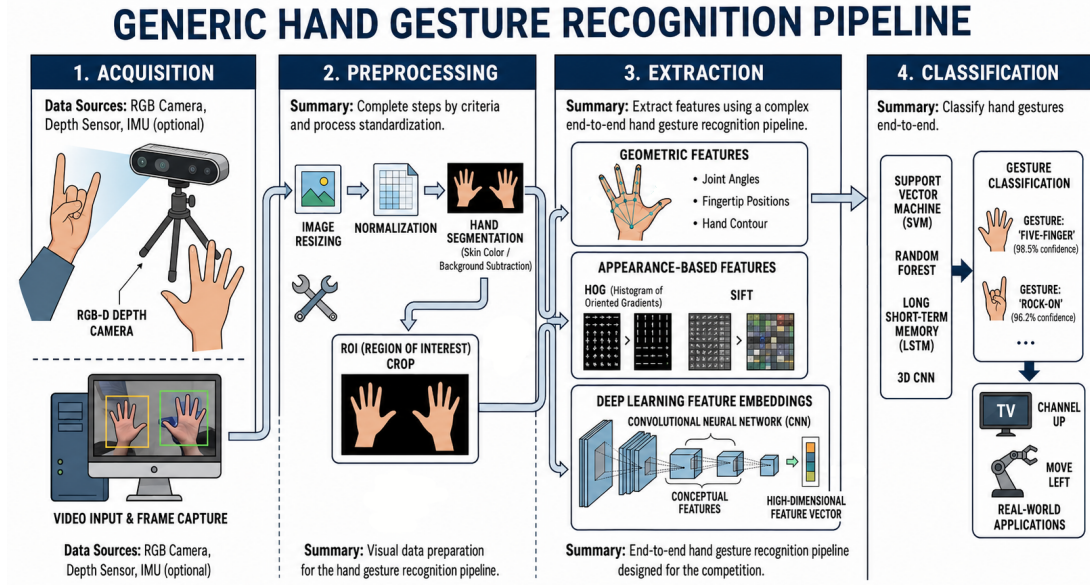


Figure 1.2: Generic hand gesture recognition pipeline across acquisition, preprocessing, extraction, and classification.

1.3.1 Acquisition

Acquisition is the stage in which raw data are captured from the sensing platform. Depending on the system design, these measurements may include image frames, depth maps, temporal video sequences, or wearable-sensor streams. The role of acquisition is not yet to interpret the gesture, but to record a measurable representation of the hand state that can be processed by later stages [1–3, 25].

1.3.2 Preprocessing

Preprocessing reduces nuisance variability before recognition. Typical operations include hand localization, segmentation, background suppression, scale and orientation normalization, temporal alignment, and noise filtering, with the exact choice determined by the sensing modality and the properties of the captured signal. The purpose of preprocessing is to improve the consistency of the input representation before features are computed [1, 3, 4, 23].

1.3.3 Feature Extraction

Feature extraction transforms preprocessed data into a representation that preserves information relevant to gesture identity. In classical systems, this stage often relies on handcrafted descriptors derived from shape, orientation, motion, or geometric structure. In learning-based systems, the representation can instead be learned directly from data, especially when neural architectures are trained end to end [5, 9, 19, 23].

1.3.4 Classification

Classification assigns the extracted representation to one of the target gesture classes. Formally, the classifier implements a decision rule that maps a feature vector or learned representation to a discrete label. The specific model may vary, but the conceptual role of this stage is stable across HGR systems: it converts informative features into final recognition decisions [18, 19, 26–28].

1.4 Machine Learning and Learning Paradigms

Machine learning studies algorithms that improve task performance by learning regularities from data or experience rather than relying solely on fixed, manually specified rules. In gesture recognition, this enables systems to either map observations to predefined labels or uncover patterns in unlabeled data [18, 19]. These capabilities are typically categorized into three fundamental learning paradigms, which structure most modern approaches. As illustrated in Figure 1.3, these are supervised learning, unsupervised learning, and reinforcement learning.

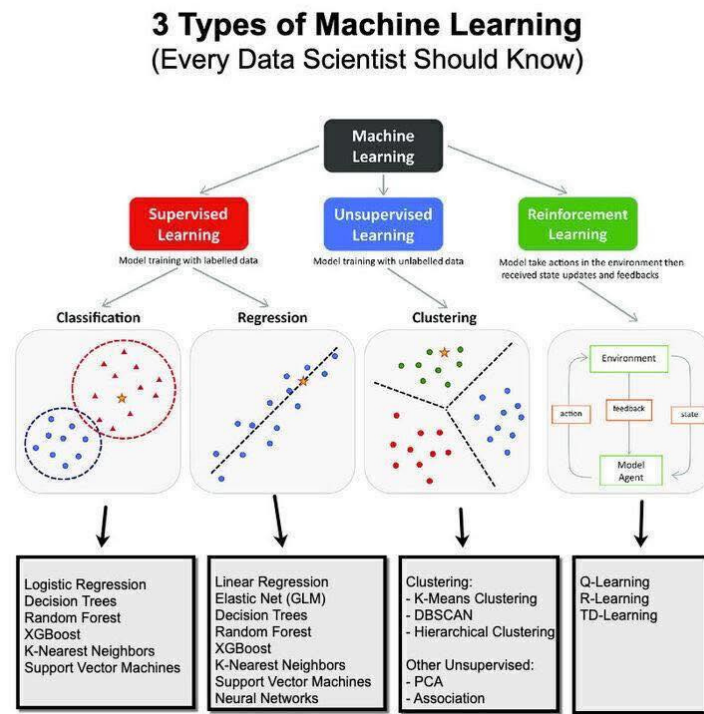


Figure 1.3: Overview of supervised, unsupervised, and reinforcement learning paradigms. *Source:* ScienceDirect [41].

1.4.1 Supervised Learning

In supervised learning, each training example is paired with a target label, and the model learns a mapping from inputs to outputs by minimizing a loss that measures prediction error on labeled data. Gesture classification is typically formulated in this way when annotated samples are available because each observation can be associated with a predefined gesture class [18, 19].

1.4.2 Unsupervised Learning

In unsupervised learning, the training data do not include target labels. The objective is instead to discover structure such as clusters, low-dimensional embeddings, or latent representations that summarize regularities in the observations. For gesture data, unsupervised learning is relevant when the goal is representation learning or exploratory structure discovery rather than direct label prediction [18, 19, 29].

1.4.3 Semi-Supervised Learning

Semi-supervised learning lies between the previous two paradigms by combining a small labeled set with a larger unlabeled set. The motivation is that unlabeled data may still constrain the learned decision boundary or improve the learned representation when annotation is costly. This setting is especially relevant in domains where acquiring labels is substantially more expensive than collecting raw observations [21, 30].

1.4.4 Reinforcement Learning

Reinforcement learning differs from classification-based paradigms because an agent interacts with an environment and learns a policy through reward signals rather than through direct class labels. The objective is to maximize cumulative reward over time, so the learning problem is defined in terms of sequential decision making instead of static supervised prediction [22].

1.5 Core Model Families for Gesture Classification

The classification stage of an HGR system can be implemented by several model families. These families differ in how they represent input structure, how they form decision boundaries, and how much inductive bias they impose on spatial, temporal, or relational data [18, 19, 23].

1.5.1 Multilayer Perceptrons

A multilayer perceptron (MLP) is a feedforward neural network composed of stacked fully connected layers. For layer ℓ , the hidden representation is commonly written as

$$\mathbf{h}^{(\ell)} = \phi(\mathbf{W}^{(\ell)}\mathbf{h}^{(\ell-1)} + \mathbf{b}^{(\ell)}), \quad (1.1)$$

where $\mathbf{W}^{(\ell)}$ and $\mathbf{b}^{(\ell)}$ are trainable parameters and ϕ is a nonlinear activation function. By composing multiple affine transformations with nonlinear activations, MLPs can learn nonlinear decision boundaries from fixed-length input vectors [18–20].

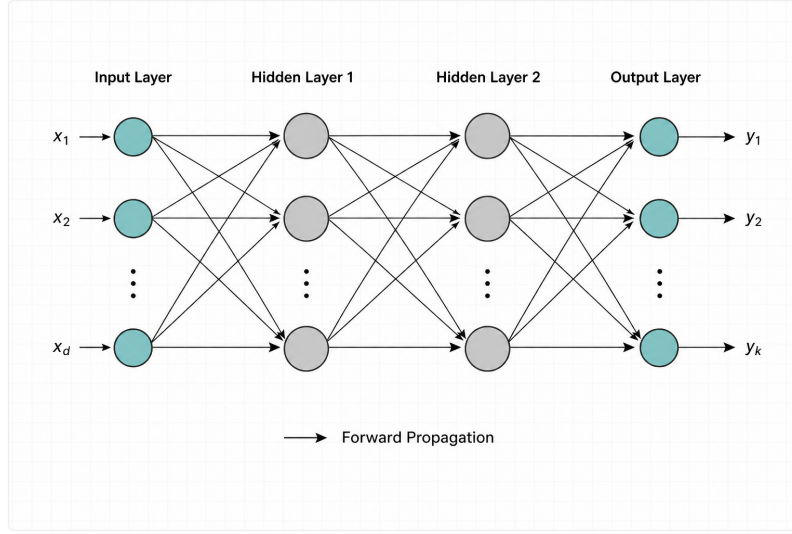


Figure 1.4: Overview of a multilayer perceptron (MLP) architecture.

1.5.2 Convolutional Neural Networks

Convolutional neural networks (CNNs) are designed for grid-structured data such as images. Their defining operations use local receptive fields and shared filters, which allows the network to learn spatially localized patterns while reusing the same parameters across positions. This inductive bias makes CNNs well suited to visual gesture data, where local shape and appearance cues must be integrated into hierarchical visual representations [9–11, 19, 31].

For input feature maps $\mathbf{X}_c^{(\ell-1)}$ and filters $\mathbf{K}_{k,c}^{(\ell)}$, a typical convolutional layer computes

$$\mathbf{X}_k^{(\ell)}(i, j) = \phi \left(\sum_{c=1}^{C_{\ell-1}} \left(\mathbf{K}_{k,c}^{(\ell)} * \mathbf{X}_c^{(\ell-1)} \right) (i, j) + b_k^{(\ell)} \right), \quad (1.2)$$

where $*$ denotes discrete convolution, k indexes output channels, and ϕ is a nonlinear activation function.

The staged local-to-global processing logic of CNNs is illustrated in Figure 1.5, where successive convolution and pooling stages progressively build more abstract visual features

before final classification [9, 19].

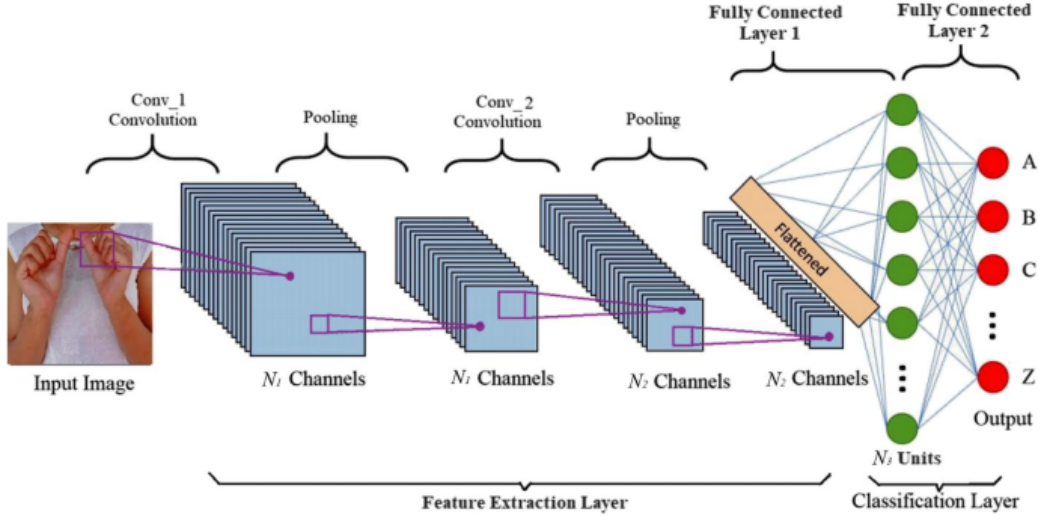


Figure 1.5: Convolutional neural network processing from local filtering to final classification. *Source:* Semantic Scholar [42].

1.5.3 k-Nearest Neighbours

k-Nearest Neighbours (k-NN) is an instance-based classifier that assigns the label of a query sample according to the labels of its nearest training examples in the feature space. The method is conceptually simple because it relies on similarity comparisons rather than an explicit parametric model, which makes it a common baseline when informative features and an appropriate distance metric are available [18, 27].

For a query sample $\mathbf{x}$, the prediction can be written as

$$\hat{y}(\mathbf{x}) = \arg \max_{c \in \mathcal{C}} \sum_{i \in \mathcal{N}_k(\mathbf{x})} \mathbf{1}[y_i = c], \quad (1.3)$$

where $\mathcal{N}_k(\mathbf{x})$ denotes the set of the k nearest training samples and $\mathbf{1}[\cdot]$ is the indicator function.

Figure 1.6 illustrates the k-NN principle, where a sample is classified from the dominant class among its neighboring points.

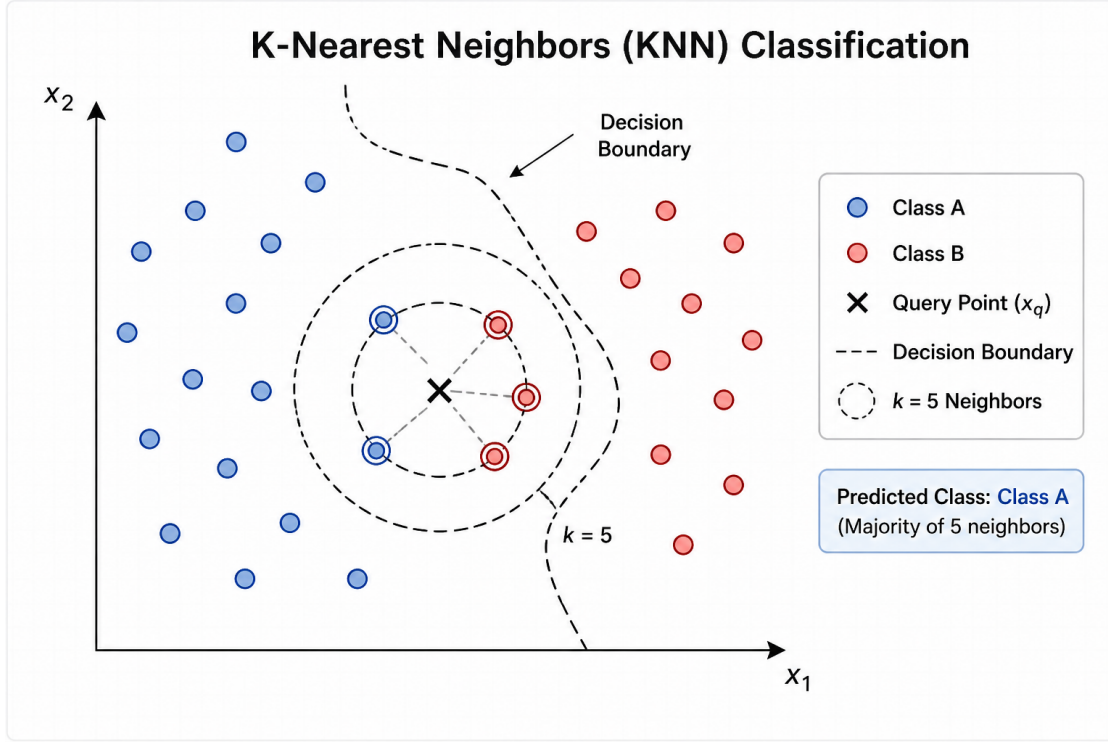


Figure 1.6: Illustration of k-NN classification based on neighboring labeled samples in the feature space.

1.5.4 Support Vector Machines

Support Vector Machines (SVMs) are margin-based classifiers that seek a separating hyperplane with maximum margin between classes. By emphasizing the decision boundary defined by the most informative training samples, SVMs often provide strong discriminative performance when the feature representation is informative and class separation can be expressed effectively in the chosen feature space [18, 26].

In the soft-margin linear case, SVM training is commonly formulated as

$$\begin{aligned}
 \min_{\mathbf{w}, b, \xi} \quad & \frac{1}{2} \|\mathbf{w}\|^2 + C \sum_{i=1}^n \xi_i \\
 \text{subject to} \quad & y_i(\mathbf{w}^\top \mathbf{x}_i + b) \geq 1 - \xi_i, \\
 & \xi_i \geq 0,
 \end{aligned} \tag{1.4}$$

where $\mathbf{w}$ and b define the separating hyperplane and ξ_i are slack variables that allow limited classification errors.

Figure 1.7 shows the SVM principle, where the classifier separates classes by maximizing the margin around the decision boundary.

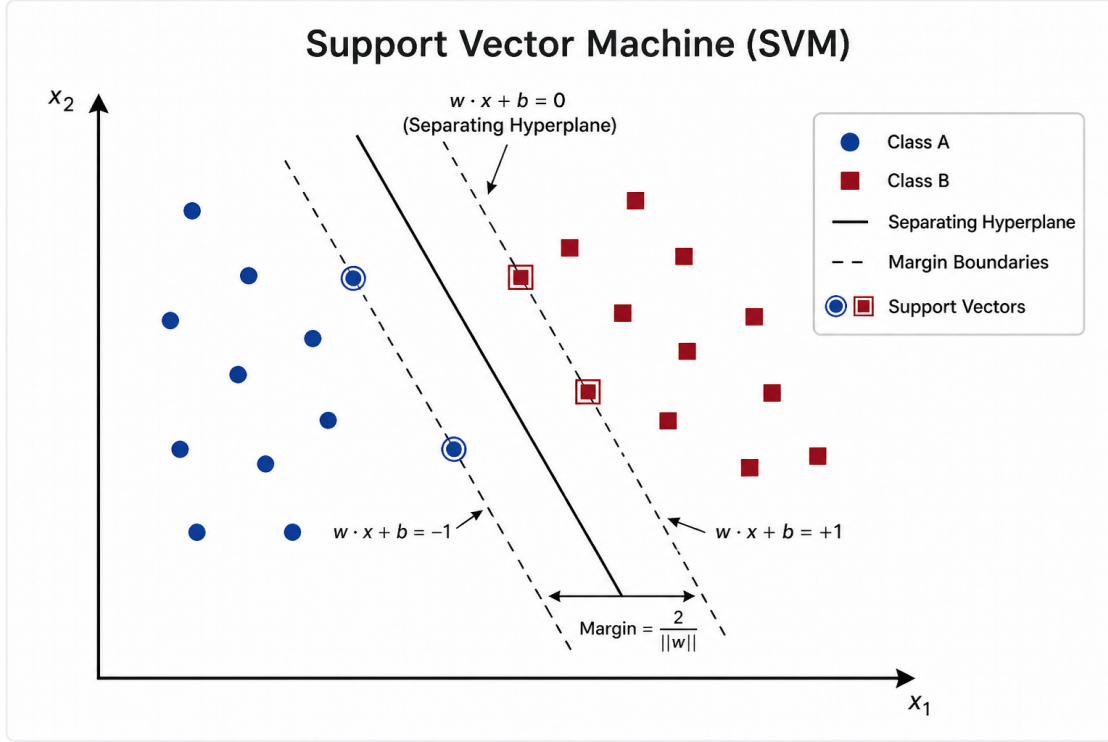


Figure 1.7: Illustration of SVM classification with a separating hyperplane and maximum-margin decision boundary.

1.5.5 Random Forests

Random forest is an ensemble learning method that combines many decision trees instead of relying on a single tree. During training, individual trees are typically constructed from bootstrapped samples of the training data, and each split considers only a random subset of features. For classification, the final decision is obtained by aggregating the predictions of all trees, commonly through majority voting. This design improves generalization by reducing the variance of individual trees while preserving the ability to model nonlinear decision boundaries and complex feature interactions [18, 28].

If $h_t(\mathbf{x})$ denotes the prediction of tree t , the ensemble decision for classification can be written as

$$\hat{y}(\mathbf{x}) = \text{mode}\{h_t(\mathbf{x})\}_{t=1}^T, \quad (1.5)$$

where T is the number of trees and the mode operation returns the majority vote across the ensemble.

Figure 1.8 illustrates the random-forest principle, where multiple decision trees contribute to a final ensemble prediction.

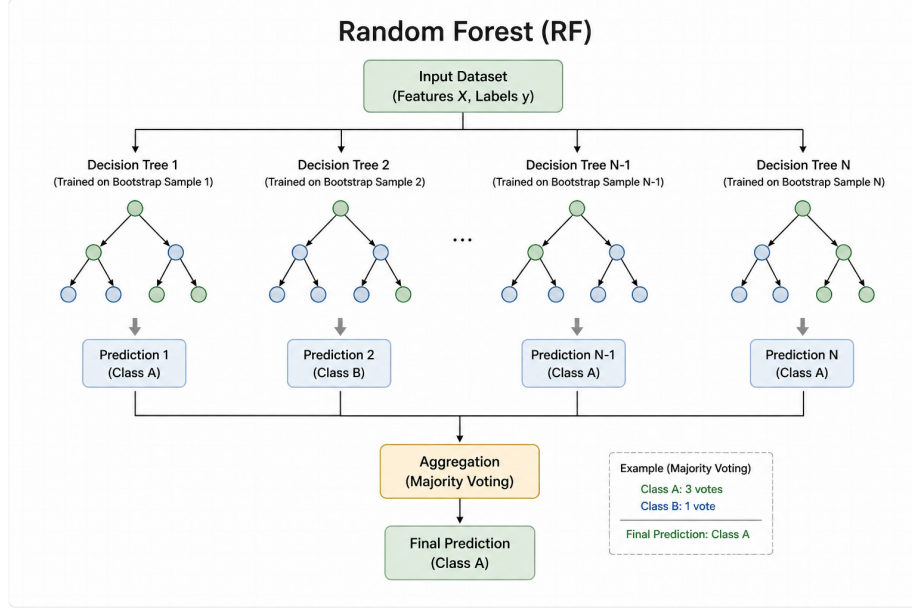


Figure 1.8: Random forest ensemble of decision trees for classification, where multiple trees contribute to a final aggregated prediction.

1.5.6 Transformers and Graph Neural Networks

Transformers model dependencies through self-attention, which allows each token representation to be updated using weighted information from other tokens in the input. In vision problems, this principle can be applied to tokenized image patches or related sequence representations so that global relationships are modeled explicitly [12, 13].

The core self-attention operation is commonly written as

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}}\right) \mathbf{V}, \quad (1.6)$$

where $\mathbf{Q}$, $\mathbf{K}$, and $\mathbf{V}$ are the query, key, and value matrices and d_k is the key dimension.

Figure 1.9 shows a representative vision-transformer workflow in which an image is decomposed into patches, embedded as tokens, and processed by stacked self-attention blocks before classification [13].

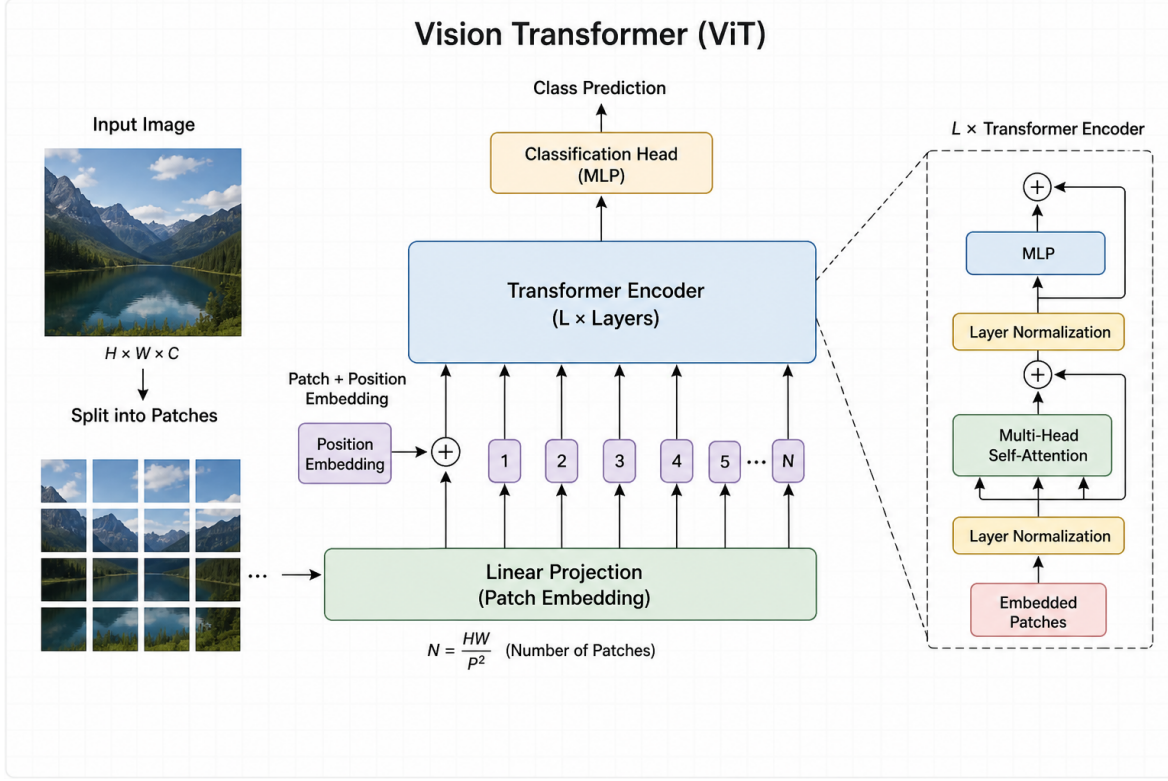


Figure 1.9: Vision transformer pipeline using patch tokens and stacked self-attention layers.

Graph neural networks (GNNs) operate on graph-structured data by propagating information along edges between nodes. This abstraction is relevant to gesture analysis when the hand is represented as a relational structure, for example through joints or landmarks connected by anatomical or spatial relations [32, 33].

A simple message-passing update for node v at layer ℓ can be expressed as

$$\mathbf{h}_v^{(\ell+1)} = \phi \left(\mathbf{W}_{\text{self}}^{(\ell)} \mathbf{h}_v^{(\ell)} + \sum_{u \in \mathcal{N}(v)} \mathbf{W}_{\text{nbr}}^{(\ell)} \mathbf{h}_u^{(\ell)} \right), \quad (1.7)$$

where $\mathcal{N}(v)$ is the neighborhood of node v and the update aggregates information from adjacent nodes.

Figure 1.10 illustrates the message-passing view of GNNs, where each node aggregates information from its neighborhood and updates its embedding before a downstream prediction stage [33].

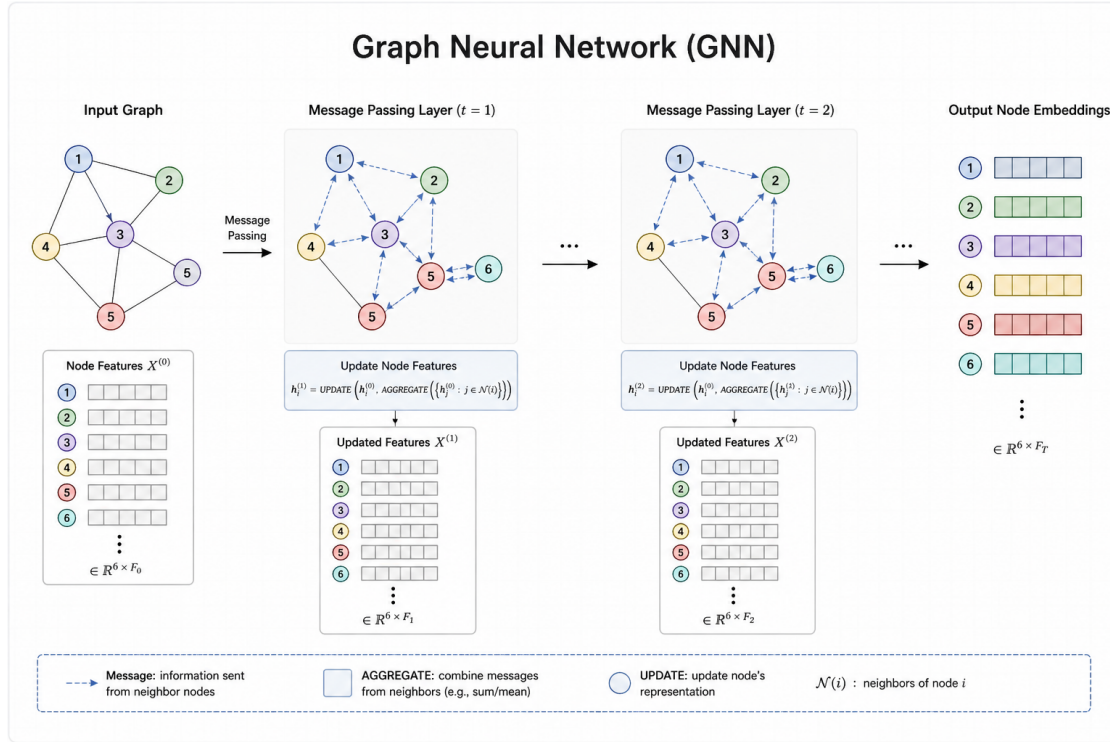


Figure 1.10: Graph neural network message passing through neighborhood aggregation and updates.

1.6 Activation Functions in Neural Networks

Activation functions introduce nonlinearity into neural networks and determine how intermediate linear combinations are transformed before being passed to the next layer. They are therefore central to the expressive power and optimization behavior of neural classifiers [18, 19].

Figure 1.11 presents the main activation functions discussed in this section, highlighting their characteristic response curves and typical output ranges.

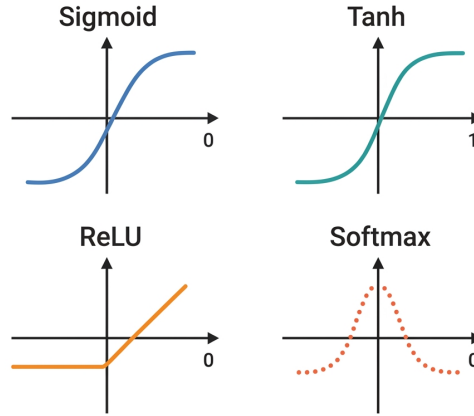


Figure 1.11: Overview of common activation functions used in neural networks, including sigmoid, hyperbolic tangent, ReLU, and softmax.

1.6.1 ReLU

The rectified linear unit is defined as

$$\text{ReLU}(u) = \max(0, u). \quad (1.8)$$

ReLU maps negative inputs to zero and preserves positive inputs unchanged. Its piecewise-linear form makes it computationally simple and effective for hidden layers in deep networks [19, 34].

1.6.2 Sigmoid

The sigmoid function is defined as

$$\sigma(u) = \frac{1}{1 + \exp(-u)}. \quad (1.9)$$

It maps real-valued inputs to the interval $(0, 1)$ and is therefore convenient when an output must be interpreted as a probability in a binary setting. Because the function saturates near its extremes, gradients can become small when activations are very positive or very negative [18, 19].

1.6.3 Hyperbolic Tangent

The hyperbolic tangent is defined as

$$\tanh(u) = \frac{\exp(u) - \exp(-u)}{\exp(u) + \exp(-u)}. \quad (1.10)$$

Like the sigmoid, it is smooth and bounded, but its output lies in $(-1, 1)$ and is zero-centered. This property often makes it easier to interpret as a hidden-layer activation than the sigmoid when signed activations are desired [18, 19].

1.6.4 Softmax

For a class-score vector $\mathbf{z} = [z_1, \dots, z_C]^\top$, the softmax function is

$$\text{softmax}(\mathbf{z})_c = \frac{\exp(z_c)}{\sum_{j=1}^C \exp(z_j)}. \quad (1.11)$$

Softmax converts arbitrary real-valued scores into a normalized probability distribution over C mutually exclusive classes. For this reason, it is the standard activation used in the output layer of multiclass classifiers [18, 19].

1.7 Conclusion

In conclusion, this chapter has provided the theoretical foundation for the remainder of the thesis by clarifying the definition of hand gesture recognition, differentiating its main gesture and sensing categories, and outlining the general stages of a recognition pipeline. It has also reviewed the principal learning paradigms in machine learning, summarized representative model families used for gesture classification, and revisited the activation functions that support neural-network-based decision making .

State of the Art in Hand Gesture Recognition

Chapter 2

State of the Art in Hand Gesture Recognition

2.1 Introduction

A hand gesture is simply a meaningful hand position or movement: a wave, a thumbs-up, or the complex shapes of sign language. The goal of Hand Gesture Recognition (HGR) is to make a computer understand those gestures from camera images or other sensors, so that users can interact with machines using their hands instead of a keyboard or mouse. Achieving this is harder than it might appear, because hands move fast, bend in many ways, overlap their own fingers, and look very different under different lighting conditions. Solving these problems reliably, at low cost, and fast enough to feel immediate is what the entire HGR research field is working toward [1–3].

Three kinds of gesture:

All gesture vocabularies fall into one of three categories. *Static gestures* are defined by a single hand shape held still, like the letter ``A" in Sign Language. *Dynamic gestures* carry meaning through movement over time, like a beckoning motion or a swipe. *Hybrid gestures* combine both: a shape that must also travel along a particular path. This distinction matters practically, because the three types demand very different recognition strategies, as Table 2.1 illustrates.

Table 2.1: The three types of gesture: what they are, how systems typically recognise them, and the core engineering trade-off each imposes.

Gesture type	What defines its meaning	Typical recognition approaches	Core trade-off
Static	A single hand shape held at one moment	SVM, KNN, MLP, CNN, landmark + classifier	Fast and simple, but cannot capture movement
Dynamic	Motion, direction, and the order of positions over time	HMM, RNN, 3D CNN, Transformer, CNN+RNN	Expressive, but needs more data, more computation, and more time to respond
Hybrid / Transitional	Shape combined with motion	Multistage or multimodal fusion pipelines	Most expressive, but hardest to build and deploy efficiently

This thesis concentrates on *static and quasi-static* recognition using compact hand descriptions computed from a standard webcam. That setting already covers a wide range of useful applications and allows thorough evaluation on modest hardware. The broader literature is reviewed here to place that focus in context.

What stands out across the field:

Reading the literature as a whole reveals a pattern worth stating upfront. Three concepts appear in almost every paper: deep convolutional neural networks (CNNs), accuracy numbers on benchmark datasets, and efficient lightweight architectures. Yet beneath this apparent agreement lie several important unresolved disputes about which sensing technology is best, what the numbers actually measure, and whether high scores on a controlled dataset predict anything about how a system will behave when deployed on real users in real environments. These tensions run through the whole chapter and motivate the choices made in this thesis.

The chapter is organised as follows. Section 2.2 covers why HGR matters and where it is used. Section 2.3 explains why it is technically difficult. Section 2.4 surveys the main approaches, from sensor gloves to deep neural networks. Section 2.5 identifies the most important disagreements in the literature. Section 2.6 examines how research in the field is conducted. Section 2.7 maps the open problems. Section 2.8 explains how this thesis is positioned relative to all of the above, and Section 2.9 summarises the key conclusions [1–3, 25].

2.2 Applications and Motivation

HGR is relevant across a wide range of domains. Understanding this diversity is important: HGR is not one problem but a family of problems, and each domain imposes its own requirements for speed, accuracy, hardware cost, and robustness to difficult conditions.

2.2.1 Human-Computer Interaction

The most familiar application is everyday computer control, replacing or supplementing a keyboard, mouse, or touchscreen with hand gestures, for example to navigate slides, control multimedia, or operate a virtual keyboard. The appeal is the absence of physical contact and a more natural interaction style. The key engineering constraint is latency: if the system takes more than a few tens of milliseconds to respond, the interaction feels broken. This pushes designers strongly toward lightweight recognition pipelines [3, 4, 14].

2.2.2 Augmented and Virtual Reality

In augmented reality (AR) and virtual reality (VR), the hand is the primary tool for manipulating virtual objects. Users expect to grab, point, and navigate in a way that feels continuous and physical. Any visible delay or unstable prediction breaks the sense of presence and degrades the experience. Recognition must therefore work in real time, under rapid hand deformation, with self-occlusion (fingers hiding each other) and cluttered backgrounds [4, 35].

An important complication: Hoppe et al. [35] found that mouse- and controller-based gesture input is a viable primary interaction method in VR, directly questioning the widespread assumption that vision-only pipelines are the only sensible design. This conflicts with the conclusion of Amin et al. [25], who argue that vision-based sensors outperform all alternatives on most benchmarks. The disagreement is discussed further in Section 2.5.

2.2.3 Sign Language Recognition

Sign language recognition (SLR) is one of the most socially important and technically demanding HGR problems. Roughly 72 million people worldwide use sign languages within a hearing-impaired population of more than 360 million [25]. Automated SLR has the potential to bridge communication barriers in education, public services, and daily life.

Sign language combines *manual* cues (hand shape, movement, location, orientation) with *non-manual* cues (facial expressions, mouth position, body posture). The hands carry most of the meaning, but fully accurate SLR ultimately requires both. A systematic review of 256 studies found that recognition accuracy depends heavily on whether the system is tested on

the same users it was trained on: across those 256 studies, accuracy averaged 87.9% when tested on known users but dropped to 79.0% on new users, a gap of roughly nine percentage points [23]. That gap is the clearest quantitative sign that generalisation across users is the central unsolved problem.

2.2.4 Accessibility, Robotics, and Smart Environments

Beyond sign language, HGR supports robot control through intuitive hand commands, touch-less interaction in hygienic or safety-critical industrial settings, and accessible interfaces for users with motor or speech impairments [3, 4]. The common thread across all these domains is the same: recognition must be accurate, fast, and practical on accessible hardware, exactly the requirements that motivate the design choices of this thesis [14–17].

2.3 Main Technical Challenges

Several interacting factors make HGR genuinely difficult. Table 2.2 presents the main technical challenges together with their practical effects, the strategies used to deal with them, and an assessment of how serious each challenge currently is.

Table 2.2: Core technical challenges in HGR: what goes wrong, what helps, and how serious each challenge remains.

Challenge	What goes wrong	Common mitigations	Current severity
Illumination variability	Changing light alters skin colour and contrast, confusing segmentation algorithms	Use YCbCr or CIE Lab colour spaces; add depth sensing; augment training data	Medium: partially solved in normal conditions; open in extreme lighting
Self-occlusion	Fingers hiding behind each other remove visual evidence from a 2D image	Depth cameras; 3D landmark estimation; using multiple frames over time	High: explicitly acknowledged as unsolved [6, 7]
Cross-user variability	Different users produce the same gesture in different ways; systems trained on some users fail on others	Larger and more diverse datasets; domain adaptation methods	High: the ≈ 9 -point signer-independent deficit across 256 studies is the primary open problem [23]

(continued on next page)

(continued from previous page)

Challenge	What goes wrong	Common mitigations	Current severity
Intra-class ambiguity	Two gestures may differ only in a small joint angle, making them nearly identical to a camera	Richer feature representations; larger balanced datasets; stronger classifiers	Medium: partly addressed by landmark pipelines and deep features
Real-time latency	The system must respond within approximately 40 ms to feel immediate	Compact descriptors; lightweight models; hardware-specific optimisation	Medium: manageable with landmark pipelines; harder for CNN or Transformer baselines [14, 15]
Dataset scarcity and bias	Most gesture datasets are small, controlled, and dominated by American Sign Language (56% of language-specific studies)	Larger multimodal datasets; synthetic data generation	High: limits generalisation across languages, demographics, and environments [23]
Demographic neutrality	No paper in the reviewed corpus explicitly tests whether systems work equally well across skin tones or cultural gesture conventions	Diverse data collection; fairness-aware benchmarking	High: an untested assumption present in virtually all experimental work

Two challenges stand out as most consequential. The first is inter-user variability. The nine-point drop in accuracy when a system is evaluated on new users, as documented consistently across 256 independent studies [23], is the single clearest evidence that the field has not yet solved the generalisation problem. The second is demographic neutrality. The implicit assumption that gesture recognition works equally for all users, regardless of skin tone or cultural background, is almost certainly false, yet no paper in the reviewed corpus tests it directly. Zhang et al. [14] acknowledge in passing that the skin-tone diversity of their training dataset is not fully disclosed, one of the few moments of partial transparency on this issue.

2.4 State of the Art

The review moves from physical sensing devices, through image-processing pipelines, to the classifier families that map features to gesture labels. A synthesis of the key technical contradictions closes the section [1–3].

2.4.1 Glove-Based and Wearable Sensor Approaches

Some of the earliest accurate HGR systems used instrumented gloves fitted with flex sensors (which measure finger bending), inertial measurement units (IMUs), or electromyography (EMG) electrodes that sense the electrical activity of muscles. Because these devices measure articulation directly from the body rather than inferring it from a camera image, they avoid the core difficulties of illumination, background clutter, and visual occlusion [1–3].

Reported accuracy figures are strong. A systematic review documents 85% accuracy on 25 Korean Sign Language words with a Fuzzy Min-Max Neural Network; 89.5% to 100% on subtasks of a 250-word Taiwanese Sign Language corpus with Hidden Markov Models; 93.1% on 72 dynamic Chinese Sign Language gestures using EMG combined with a six-axis accelerometer; and 98.6% on 20 Brazilian Sign Language (Libras) classes using a Myo armband and a Support Vector Machine [23]. These numbers are impressive. The practical drawback is equally clear: dedicated hardware is expensive, must be calibrated for each user, and is cumbersome to wear, making deployment outside a laboratory setting uncommon [1, 2].

Table 2.3 compares wearable sensors against camera-based alternatives across the dimensions that matter most for deployment.

Table 2.3: Sensing modalities for HGR: how each captures the hand, what it costs, and where it performs best. Accuracy ranges are drawn from the reviewed literature.

Modality	Device	What the signal is like	Deployment profile	Accuracy range	Best-suited domain
Glove / Wearable	Flex sensors, IMU, EMG	High quality: measures articulation directly; not affected by lighting	Expensive; requires calibration per user; must be worn	85 → 98.6%	Lab or clinical settings

(continued on next page)

(continued from previous page)

Modality	Device	What the signal is like	Deployment profile	Accuracy range	Best-suited domain
Standard RGB camera	Webcam or smartphone	Medium quality: cheap and accessible; sensitive to lighting and background	Inexpensive; contactless; runs on any device	64 → 99%	HCI, accessibility, consumer AR/VR
Depth / RGB-D camera	Kinect, Leap Motion, time-of-flight	High quality: adds a distance channel; much better hand segmentation	Higher cost; specialised hardware required	75 → 96%	Robotics, industrial settings, research
Landmark-based (MediaPipe)	21 hand keypoints extracted from a standard RGB camera	High compactness: background is discarded; loses information when fingers overlap	Very lightweight classifier; runs on commodity hardware in real time	Varies by classifier	Real-time HCI, mobile deployment
Hybrid (vision + sensor)	RGB camera plus EMG, IMU, or depth	Highest quality: complementary signals fill each other's gaps	High hardware and synchronisation cost	86% → 96% after fusion	Research settings; rarely deployed

2.4.2 Vision-Based Approaches: The Processing Pipeline

Vision-based HGR uses one or more cameras and typically passes each frame through five stages: acquisition, pre-processing, segmentation, feature extraction, and classification [3, 4]. Understanding these stages helps explain where each family of methods sits and why they make different accuracy-efficiency trade-offs.

a. Data Acquisition

Standard RGB cameras dominate because they are inexpensive and ubiquitous, but they are sensitive to lighting and lack depth information. Depth sensors such as the Kinect or Leap Motion improve segmentation and enable skeletal tracking at higher hardware cost [3, 25].

b. Pre-Processing and Segmentation

Pre-processing reduces noise and normalises the image through operations such as Gaussian smoothing and histogram equalisation. Segmentation then isolates the hand from the background using skin-colour models in colour spaces such as HSV, YCbCr, or CIE Lab, or using background subtraction and edge detectors [1, 3]. YCbCr is more stable under illumination changes than HSV; CIE Lab performs best in the most extreme lighting conditions [23].

c. Feature Extraction

Feature extraction converts the isolated hand region into a compact numerical description. The main families are: dimensionality reduction methods such as PCA and LDA; handcrafted descriptors such as Histogram of Oriented Gradients (HOG), Local Binary Patterns (LBP), and Zernike Invariant Moments; and deep features learned automatically by convolutional networks [3, 5, 19]. The shift from handcrafted to learned features has been one of the defining trends of the past decade, driven by the success of deep learning on large benchmarks [10, 11, 31].

2.4.3 Landmark-Based Pipelines

A particularly attractive modern approach avoids raw pixel features altogether. Instead of feeding the full image to a classifier, the system first locates the hand and computes 21 *landmarks*, namely the 3D positions of the main joint and palm points. Each hand is then represented as a compact geometric vector of those 21 positions. This vector can be classified by simple, fast models: a multi-layer perceptron (MLP), a support vector machine, a random forest, or a k -nearest neighbours classifier [7, 8, 14, 28].

The core advantage is efficiency: because all background pixels are discarded before classification, computation is low and the system runs in real time on a standard laptop or smartphone. The corresponding weakness is dependence on the upstream detection step: if fingers are occluded or the image is blurry, the landmark positions will be unreliable, and errors propagate directly into the classifier. Despite this, the accuracy-efficiency balance of landmark pipelines is among the best available for static gesture recognition, which is why they are at the centre of this thesis [7, 8, 14].

One methodological debate runs through this approach. Simon et al. [7] argue that the bottleneck in hand pose estimation is data: their main contribution is a bootstrapping strategy that generates large annotated datasets cheaply. Zimmermann and Brox [8], by contrast, argue the bottleneck is architecture: they show that a well-designed network generalises from single RGB images without needing depth sensors. Neither paper runs the controlled experiment needed to settle the question. Such an experiment would vary data volume and architecture independently while holding everything else fixed, so the dispute remains open.

2.4.4 CNN-Based Approaches

Convolutional neural networks (CNNs) learn to recognise gestures directly from raw images, building up an internal hierarchy of features through training on labelled examples: edges and textures first, then hand parts, then full configurations [9–11, 31]. For dynamic gestures, three-dimensional CNNs that process short video clips and two-stream networks that combine appearance with motion (optical flow) have shown particularly strong results [6]. CNNs have driven much of the progress since 2018, but at a cost: they require substantially more labelled data, memory, and computation than landmark-based models, making real-time deployment on modest hardware difficult.

Efficient CNN variants have been developed specifically to reduce this burden. MobileNet [15], MobileNetV2 [16], and EfficientNet [17] achieve near-standard accuracy at a fraction of the parameter count. All three, however, have been benchmarked exclusively on ImageNet, a large general image classification dataset. Whether their efficiency advantages transfer to gesture-specific inputs, such as temporal sequences, skeleton-based representations, or low-resolution crops, remains an open question not addressed by any paper in the reviewed corpus.

2.4.5 Transformer-Based Approaches

Vision Transformers (ViTs) replace local convolution with global self-attention: instead of looking at small neighbourhoods of pixels, the model can relate any part of the image to any other simultaneously [12, 13]. For gesture recognition, this is potentially useful when the discriminative cue is spread across non-adjacent parts of the hand, for example, a subtle relationship between the angle of the thumb and the position of the little finger. In practice, Dosovitskiy et al. [13] themselves note that ViT only matches or surpasses CNNs when trained on very large corpora, a scale unavailable in most gesture datasets.

Including a ViT baseline in this thesis is nonetheless valuable: not because it is automatically the best choice, but because it quantifies precisely how much accuracy benefit global-attention models provide over lightweight landmark approaches under the same resource constraints.

2.4.6 Hybrid Sensor-Vision Approaches

Hybrid systems pair a camera with one or more physical sensors, typically IMU, EMG, or depth, so that each modality covers the other's weaknesses. A concrete example: combining a Leap Motion camera with surface EMG raised accuracy from 86% with EMG alone to 96% after fusion [23]. These gains are real, but hybrid systems remain laboratory-oriented because the additional hardware, synchronisation overhead, and computational burden make

them impractical for everyday deployment. This reinforces the case for markerless vision pipelines as the design default in this thesis [23, 25].

2.4.7 Comparing the Main Architectures

Table 2.4 places all major architecture families side by side and assesses each against the deployment requirements of this thesis: real-time performance on commodity hardware with a single RGB camera.

Table 2.4: Architecture comparison for real-time HGR on modest hardware.

Architecture	Input features	Data needed	Compute cost	Deploy. fit	Assessment for real-time HGR
Handcrafted (HOG, LBP, ZIM)	Hand-engineered descriptors	Minimal	Low	High	Fast and interpretable; brittle when conditions change from training
Shallow CNN (LeNet-style)	Local spatial features	Moderate	Medium	Medium	Works well on simple static gestures; limited for motion
Deep CNN (VGG, ResNet)	Hierarchical spatial features	Large labelled dataset	High	Low	State of the art on controlled benchmarks; impractical on typical hardware
3D CNN / Two-stream	Spatial and temporal features from video	Large video dataset	Very high	Very low	Best for dynamic gestures; rarely deployable in real time
Vision Transformer (ViT)	Global self-attention over image patches	Very large dataset	Very high	Very low	Matches CNNs only at scale; poor data efficiency on small gesture sets

(continued on next page)

(continued from previous page)

Architecture	Input features	Data needed	Compute cost	Deploy. fit	Assessment for real-time HGR
Landmark + MLP / SVM (this thesis)	21 joint coordinates from RGB	MediaPipe on commodity hardware	Low	High	Best accuracy-efficiency balance for static real-time HGR

2.4.8 Classification Methods and Reported Performance

The final stage of any HGR pipeline maps the extracted feature vector to a gesture label. The most common classifiers are: support vector machines (SVM), k -nearest neighbours (KNN), artificial neural networks and multi-layer perceptrons (MLP), random forests, hidden Markov models (HMM) for sequences, and deep architectures (CNN, RNN, Transformer) [12, 26–28].

Representative performance figures from the literature span a wide range: DataGlove + HMM achieves 89.5% to 100% on subtasks of a 250-word Taiwanese Sign Language corpus; EMG + a six-axis accelerometer reaches 93.1% on 72 dynamic Chinese Sign Language gestures; SVM + SIFT + Bag of Features reaches 99.0% on 30 Arabic Sign Language classes; Zernike Invariant Moments + HOG + KNN/SVM reaches 96.8% on 30 Libras classes; and hybrid vision + EMG improves accuracy from 86% to 96% [23].

Four broad conclusions follow from these figures. First, wearable-sensor systems often reach the highest individual accuracy, but usually on small, controlled tasks with the same users in training and test. Second, vision-based methods show wider variance because results depend strongly on dataset size and evaluation protocol. Third, the signer-independent deficit of approximately nine percentage points is the clearest evidence that cross-user generalisation remains unsolved. Fourth, classical classifiers such as SVM, KNN, RF, and HMM still represent a substantial share of the published literature despite the rise of deep learning [23, 25].

One important caveat applies to every number above: performance on a controlled test split cannot be assumed to represent real deployment performance. The assumption that benchmark accuracy proxies real-world utility is shared by virtually every experimental paper in the corpus and is never explicitly tested. Survey-level evidence confirms that real-world performance degrades substantially outside laboratory conditions [3, 23]. A further technical problem is that accuracy and F-measure systematically misrepresent classifier quality when class distributions are unequal, as they almost always are in gesture data, where the “no

gesture" class greatly outnumbers any specific gesture class [36].

2.5 Key Contradictions in the Literature

Not all disagreements in the literature are merely matters of emphasis or dataset choice. Some are structural: two papers make claims that cannot both be true, and deciding which is right would change how other results across the field are interpreted. Table 2.5 lists the five most consequential of these contradictions.

Table 2.5: Five direct contradictions across the reviewed literature: what is disputed, who takes which position, and what causes the disagreement.

What is disputed	Position A	Position B	Why the dispute persists
Are spatial inductive biases essential for visual recognition?	LeCun et al. [9] and He et al. [11]: convolutional priors are essential for learning from images	Vaswani et al. [12] and Dosovitskiy et al. [13]: global self-attention alone is sufficient	ViT only matches CNNs with very large training data unavailable in most gesture settings
Does adding depth straightforwardly improve performance?	Simonyan & Zisserman [31]: very deep 3×3 networks consistently outperform shallower ones	He et al. [11]: depth without residual connections causes accuracy to degrade empirically	Both use ImageNet, but locate the performance bottleneck in different places
Is F-measure an adequate evaluation metric?	Sokolova & Lapalme [37]: valid and context-dependent	Powers [36]: structurally misleading regardless of context; informedness is preferred	Irreconcilable disagreement on mathematical adequacy
Does vision-based sensing outperform all alternatives?	Amin et al. [25]: vision beats wearables on most sign language benchmarks	Hoppe et al. [35]: controller-based gesture is a viable and effective primary input in VR	Different deployment contexts; different definitions of "effective"

(continued on next page)

(continued from previous page)

What is disputed	Position A	Position B	Why the dispute persists
Is data or architecture the main bottleneck in hand pose estimation?	Simon et al. [7]: the data collection strategy is the key contribution	Zimmermann & Brox [8]: architecture and loss design are the key contribution	Neither paper runs the factorial ablation needed to isolate the two factors

2.6 How Research in This Field Is Conducted

Understanding how HGR knowledge is produced is as important as understanding what that knowledge claims. Table 2.6 summarises the methodology types present in the reviewed literature, how frequently each appears, and what its inherent limitations are.

Table 2.6: Methodology types across the 42 reviewed papers: frequency, prevalence, and key limitations.

Methodology type	Count	Prevalence	Key limitation
Computational / ML experiment	≈ 25	Dominant	Single-dataset evaluation; benchmark accuracy assumed equivalent to real-world performance (never tested)
Survey / Systematic review	≈ 6	Frequent	Heterogeneous benchmarks prevent pooling of results; conclusions are narrative, not statistical
Theoretical / Formal analysis	≈ 7	Moderate	Results hold on toy tasks or asymptotic regimes; finite-sample behaviour rarely verified
Textbook / Pedagogical	3	Rare	No original experiments; provides frameworks but no new empirical claims
Computational + user study	1	Very rare	Sample size and demographics unreported; controlled lab limits applicability [35]

(continued on next page)

(continued from previous page)

Methodology type	Count	Prevalence	Key limitation
Randomised controlled trial (with real users)	0	Absent	A critical gap: HCI and SLR applications ultimately need user evaluation, but none appears in the corpus
Meta-analysis	0	Absent	Feasible given the volume of comparable accuracy reports on overlapping benchmarks, but never attempted

Computational/ML experiments dominate because the field has tacitly agreed to measure progress as benchmark improvement, a clean, quantifiable criterion that does not require users to be recruited or real environments to be instrumented. Two methodology types are conspicuously absent despite being directly relevant to the field's own stated goals. No paper in the corpus conducts a randomised controlled trial with human participants, even though several make claims about interaction usability. And no paper performs a meta-analysis, despite the volume of comparable accuracy reports on overlapping benchmarks. Hashi et al. [23] come closest with a systematic review covering 256 studies, but they do not pool effect sizes or apply formal statistical synthesis.

2.7 Open Research Challenges

Five unresolved problems of different severity emerge from the reviewed literature. Table 2.7 presents each gap, explains why it has not been closed, rates how seriously it threatens the validity of existing conclusions, and sketches what a resolution would require.

Table 2.7: The five most significant research gaps identified from the reviewed literature, ordered by risk to existing conclusions.

Research gap	Why it persists	Risk level	What would close it
Cross-dataset generalisation	No paper tests a model trained on one dataset against a different one; no shared cross-domain evaluation protocol exists [3, 23]	High	A shared multi-environment benchmark with diverse demographics; standardised cross-dataset evaluation protocol

(continued on next page)

(continued from previous page)

Research gap	Why it persists	Risk level	What would close it
Data versus architecture as the primary bottleneck	Simon et al. and Zimmermann & Brox disagree but neither runs the controlled ablation needed to isolate the two factors [7, 8]	High	A factorial experiment varying dataset size and model architecture independently on a fixed benchmark
No agreed evaluation standard	Applied papers use accuracy or F-measure without justification; Powers [36] shows both are misleading under the class imbalance endemic in gesture data	High	A domain-specific evaluation framework that accounts for class imbalance and temporal onset/offset detection errors
Efficiency trade-offs untested on gesture data	MobileNets and EfficientNet are benchmarked on ImageNet only; whether their efficiency advantages transfer to landmark or video gesture inputs is unknown [15–17]	Medium	Systematic benchmarking of lightweight architectures on a shared gesture dataset covering temporal and skeleton inputs
Semi-supervised learning never applied to HGR	Despite acknowledged data scarcity, confidence-thresholding pseudo-labelling methods have not been applied to gesture pipelines [21, 30]	Medium	Direct application of FixMatch-style pseudo-labelling to gesture video; VAE-based augmentation of sparse gesture datasets [38]

Beyond these specific gaps, the entire field rests on four foundational assumptions that are never tested but are plausibly false. (1) Benchmark performance is a valid proxy for deployment performance, an assumption shared by virtually all experimental papers. (2) More labelled data monotonically improves a model [7, 13], a claim directly contested by Sohn et al. [30], who find that low-confidence pseudo-labels actively degrade performance. (3) Accuracy on a fixed test set adequately measures system utility [36, 37]. (4) Gesture recognition is culturally and demographically neutral, a claim left untested across the entire corpus.

2.8 The Position of This Thesis

This thesis is positioned between landmark-based and full-image approaches. It compares multiple paradigms under a single shared evaluation protocol and assesses them against the practical constraint of real-time deployment on commodity hardware using only a standard RGB camera.

Four specific design choices follow directly from the literature analysis above.

1. **Vision-based and markerless.** The system uses only a standard camera, prioritising accessibility over the higher accuracy of wearable sensors.
2. **Static and quasi-static recognition.** Frame-level features are sufficient for the target application, and temporal modelling is unnecessary. This greatly reduces computational cost.
3. **Landmark representation as the primary input.** MediaPipe Hands [14] is used to extract 21 joint landmarks from each frame. This representation discards irrelevant background pixels, keeps the feature space interpretable, and is fast enough for real-time deployment.
4. **CNN and Transformer baselines included.** Heavier image-based models are tested under identical conditions to quantify exactly how much additional accuracy, if any, they provide over the lightweight landmark pipeline.

The positioning is also informed by the multimodal spatiotemporal work of Lai and Yanushkevich [24], which demonstrates the value of structured hand representations for sequential recognition. Although this thesis does not model temporal dynamics, it adopts the same underlying principle: compact neural classifiers on geometric vectors deliver speed, compactness, and interpretability.

Critically, this thesis directly addresses the evaluation gap identified in Section 2.7. By fixing the dataset, evaluation protocol, and metrics across all compared systems, it produces results that are meaningfully comparable in a way that most prior work is not. When different papers use different benchmarks, different metrics, and different data splits, cross-system comparison becomes unreliable. This thesis avoids that problem by design.

2.9 Conclusion

In conclusion, HGR is shaped by a persistent trade-off between representational power and deployability. Wearable-sensor systems are precise but intrusive. CNN- and transformer-based models are powerful but computationally expensive. Landmark-based pipelines oc-

copy the most practical middle ground: accurate enough for real applications, fast enough for real-time use, and deployable on hardware that is genuinely accessible.

The quantitative picture is clear. Across 256 studies, signer-dependent accuracy averages 87.9% (range 64% to 98%) while signer-independent accuracy averages 79.0% (range 52% to 98%), a gap of roughly nine percentage points that identifies cross-user robustness as the main unresolved challenge [23]. Continuous sign language recognition, larger and more diverse datasets, integration of non-manual cues such as facial expression, and rigorous cross-domain evaluation remain the key frontiers of the field [23, 25].

A cross-corpus analysis adds three further conclusions not present in any individual paper. First, the field's dominant research design of training and testing a model on the same controlled dataset and reporting accuracy as the terminal outcome is precisely the design least suited to answering whether any of this works outside a laboratory. Second, several direct contradictions exist in the literature (over the role of depth, the adequacy of F-measure, the primacy of vision) and are rarely acknowledged as the open questions they are. Third, a cluster of untested foundational assumptions about benchmark validity, data monotonicity, metric adequacy, and demographic neutrality underlies virtually all experimental conclusions and represents the deepest source of uncertainty in the field.

For this thesis, the best HGR solution is therefore not the most complex one, but the one that best balances accuracy with practical efficiency, evaluated under a shared, reproducible protocol that directly addresses the evaluation gap identified above.

Design, Implementation, and Experimental Evaluation

Chapter 3

Design, Implementation, and Experimental Evaluation

3.1 Introduction

The following sections document the full design and implementation journey from raw data to a trained, evaluated model. It covers three interconnected concerns: how the dataset was designed and built from scratch, how the overall recognition pipeline was architected, and how eleven model configurations were designed, implemented, and evaluated against one another.

The chapter is organized as follows. Section 3.2 explains why no existing public dataset was suitable and motivates the need for a custom one. Section 3.3 describes the end-to-end pipeline and the design decisions that shaped it. Section 3.4 documents the implementation of each pipeline stage, from image acquisition to storage. Section 3.5 covers the design of the two input representations and the architectural decision to decouple landmark extraction from model training. Section 3.6 describes the experimental protocol. Section 3.7 documents the design rationale and implementation of each evaluated model. Section 3.8 presents the full benchmark results. Section 3.9 examines efficiency and deployment constraints. Section 3.10 analyzes how models behave under progressively smaller data budgets. Section 3.11 interprets the error patterns of the selected model. Section 3.12 brings all evidence together into the final implementation decision. Section 3.13 closes the chapter.

3.2 The Dataset Problem

The first major obstacle encountered in this work was not model design. It was data. Several publicly available hand gesture datasets were downloaded and evaluated early on. The problems were immediate and fundamental.

Most were too small, ranging from a few hundred to a few thousand images per class,

and most were collected under uniform conditions: the same lighting, the same background, sometimes even the same person. A model trained and tested on data like that produces accuracy numbers that are meaningless. Perfect scores appeared effortlessly, not because the model had learned anything general, but because the test set was nearly identical to the training set. One hundred percent accuracy on data that has already been solved is not a result. It is a warning sign.

The deeper problem is the absence of diversity. Real gesture recognition must work across different hands, different skin tones, different backgrounds, different camera angles, and different lighting conditions. A dataset collected in a single session cannot expose a model to any of that variation. Whatever such a model learns, it is unlikely to transfer outside the original recording environment.

This ruled out every dataset examined and made it clear that a new one was needed: large enough and diverse enough to give the benchmark genuine validity.

3.3 Overall Pipeline Design

Before describing individual components, it is worth presenting the architecture of the system as a whole. The full recognition pipeline, from raw internet data to a trained classifier, consists of five sequential stages, each with a well-defined input, output, and responsibility.

1. **Image acquisition.** A scraping script downloads raw images from the web at scale, targeting a wide variety of people, environments, and image qualities.
2. **Hand detection and auto-labeling.** A pretrained YOLO model, fine-tuned on the HaGRID dataset, processes each downloaded image to (a) determine whether it contains a recognizable hand gesture and (b) assign it to one of eighteen gesture categories. Images that do not yield a confident detection are discarded.
3. **Cropping and storage.** Detected hand bounding boxes are used to crop out the hand region, removing background context. The resulting images are organized into per-class folders and saved in a compressed numerical format for fast repeated access.
4. **Landmark extraction.** MediaPipe Hands is applied to every cropped image, producing a 63-dimensional vector of joint coordinates per sample. These vectors are written once to a CSV file, decoupling the extraction step entirely from downstream model training.
5. **Model training and evaluation.** Eleven model configurations are trained and evaluated, drawing from either the pixel-based image files or the precomputed landmark CSV, depending on the model family.

This staged architecture was chosen deliberately. Each stage can be developed, debugged, and re-run independently. If the cropping logic is changed, only stages 3 and onward need to be re-executed. If a new model family is added, only stage 5 is touched. The separation between landmark extraction (stage 4) and training (stage 5) is particularly important: landmark extraction is computationally expensive and would be prohibitively slow if repeated at the start of every training run. Running it once and caching the result to CSV eliminates that cost entirely.

3.4 Dataset Construction

3.4.1 Image Acquisition with `icrawler`

The image acquisition stage was implemented as a Python script using the `icrawler` library. `icrawler` provides a unified interface for crawling multiple image search engines and handles pagination, request throttling, and file I/O automatically.

For each of the eighteen target gesture classes, a set of search queries was constructed. These queries combined gesture names with contextual terms to maximize the variety of returned images (for example, ``thumbs up hand gesture'', ``thumbs up sign different backgrounds'', ``thumbs up dark skin tone''). The crawler was configured to collect on the order of tens of thousands of candidate images per class, with overlapping queries across engines to reduce retrieval bias toward any single source. The raw collection reached approximately half a million images before filtering.

Several practical decisions were made during the acquisition stage:

- **Duplicate handling.** Near-identical images, which are common when the same image appears across multiple domains indexed by the same search engine, were identified by file hash and removed before any further processing.
- **Format normalization.** Downloaded files arrived in a variety of formats including JPEG, PNG, WebP, and BMP. All were converted to PNG at this stage to standardize the pipeline downstream.
- **Minimum resolution filter.** Images with a resolution below a fixed threshold in either dimension were discarded immediately, since they cannot yield usable hand crops at the target size.

The output of this stage was a large, per-class folder of raw candidate images.

3.4.2 Hand Detection and Auto-Labeling with YOLO-HaGRID

At the scale of hundreds of thousands of images, manual labeling is not feasible. The auto-labeling stage was implemented using a YOLO model pretrained on the HaGRID dataset. HaGRID (HAnd Gesture Recognition Image Dataset) is a large-scale dataset specifically designed for gesture detection, covering 18 gesture classes under diverse conditions. The YOLO model trained on it is therefore well-suited to both detecting the presence of a hand and classifying which gesture is being performed.

Each candidate image was passed through this model. For every image, the model returned either a detection with a class label and bounding box, or no detection. Images that did not yield a detection above a confidence threshold were discarded. Images that did yield a detection were assigned the predicted class label, and the bounding box coordinates were stored for the cropping stage.

This auto-labeling approach has a known limitation: some images will receive incorrect labels. However, at this scale the noise is manageable. With tens of thousands of images per class, a modest mislabeling rate shifts the label distribution only slightly and does not prevent the dataset from being usable. The goal was not a perfectly clean dataset; it was a dataset diverse and large enough that training on it would generalize better than anything publicly available. Each class ended up with roughly 30,000 images, for a total approaching 500,000 samples. The class balance is approximately uniform, which prevents models from learning to simply predict the majority class.

Figure 3.1 gives a concrete view of this dataset at the file-system level. The first screenshot shows the per-class folder structure and the near uniform sample counts visible in WizTree, while the second shows few images inside the `dislike` class folder.

Folder	% of Parent	Size	Allocated	Items	Files
D:\Dev\datasetExtractor\dataset\cropped	100.0 %	6.8 GB	7.9 GB	573,192	573,174
mute	6.0 %	422.9 MB	484.3 MB	31,460	31,460
dislike	5.8 %	408.3 MB	471.5 MB	32,021	32,021
ok	5.8 %	408.4 MB	471.0 MB	32,053	32,053
four	5.8 %	403.7 MB	468.2 MB	33,080	33,080
three2	5.7 %	402.6 MB	464.7 MB	31,888	31,888
call	5.7 %	401.3 MB	463.9 MB	31,670	31,670
two	5.7 %	394.4 MB	459.7 MB	33,223	33,223
peace2	5.7 %	397.3 MB	459.4 MB	31,600	31,600
palm	5.6 %	394.8 MB	457.5 MB	32,220	32,220
plank	5.5 %	387.3 MB	450.4 MB	32,132	32,132
three	5.5 %	385.7 MB	448.5 MB	32,155	32,155
like	5.5 %	385.9 MB	446.8 MB	31,360	31,360
peace	5.4 %	378.2 MB	440.1 MB	31,652	31,652
two2	5.3 %	373.2 MB	433.6 MB	31,072	31,072
stop	5.3 %	370.9 MB	432.6 MB	31,451	31,451
one	5.3 %	369.6 MB	432.6 MB	32,330	32,330
rock	5.3 %	368.9 MB	429.9 MB	31,237	31,237
fist	5.0 %	348.7 MB	407.5 MB	30,570	30,570



Figure 3.1: Dataset organization and sample images. The dataset is stored as one folder per gesture class, with roughly 30,000 samples per class; the dislike folder screenshot illustrates the kind of cropped hand-gesture examples used during training.

3.4.3 Cropping and Storage

The bounding box returned by the YOLO detector was used to crop each image to the hand region, discarding the surrounding background entirely. This serves two purposes. First, it ensures that models trained on pixel data cannot rely on background statistics for classification. Second, it standardizes the spatial extent of the input: every image now contains a hand, roughly centered and filling the frame.

After cropping, each image was resized to 256×256 pixels. This resolution was chosen as a balance between spatial detail and computational cost. At 256×256 , enough hand structure is preserved for both convolutional and fully connected models to operate effectively, while

keeping the per-image storage footprint small.

One practical problem arose from storage scale. The full image collection in standard format occupied approximately 13 GB on disk. Loading that volume into memory at the start of each training run was prohibitively slow and made iteration across the many experimental conditions impractical. To solve this, the pixel values for the entire dataset were extracted once and saved in a compressed numerical format using NumPy's `.npz` format. This brought the storage footprint down to approximately 2 GB without any loss of image information, and reduced load time from several minutes to a few seconds.

3.4.4 Summary

Table 3.1 summarizes the key properties of the constructed dataset.

Table 3.1: Summary of the constructed dataset properties.

Property	Value
Number of gesture classes	18
Total samples	$\approx 500,000$
Samples per class (approx.)	30,000
Image resolution	256×256 pixels
Raw dataset size	≈ 13 GB
Compressed dataset size	≈ 2 GB
Scraping library	icrawler
Auto-labeling model	YOLO (HaGRID-trained)
Landmark extraction	MediaPipe Hands

3.5 Input Representation Design

Once the dataset was built, the next design question was what form the input to each model should take. Two distinct representations were prepared. The choice between them is not arbitrary: it reflects a fundamental decision about what information should be given to the classifier, and the experimental results later validate that this choice matters more than model architecture.

3.5.1 The Precomputed Landmark Architecture

Before describing the two representations, it is important to explain the architecture that separates them from the training pipeline. For the landmark-based models, MediaPipe Hands was applied to every image in the dataset in a single offline pass. MediaPipe Hands is a production-grade hand-tracking library that detects 21 keypoints on the hand, covering fingertips, knuckle joints, and palm reference points, and returns their normalized three-dimensional coordinates. The output for each image is $21 \times 3 = 63$ floating-point values.

These 63 values were written, one row per image, to a CSV file. The schema is fixed: 63 coordinate columns followed by a class label column. During training, the loader reads this file once and parses it into a NumPy array. There is no call to MediaPipe at training time. If a row is malformed or a coordinate column is missing, it is treated as a schema error and rejected. No interpolation or imputation is applied.

This design decision, precomputing landmarks and storing them in a flat CSV rather than computing them on the fly during training, was motivated by two practical concerns. First, MediaPipe inference is not negligible in cost. Running it on 500,000 images at the start of every training experiment would have made the full experimental campaign infeasible. Second, decoupling extraction from training makes the training code simpler, faster, and fully reproducible: the same CSV produces the same input every time, regardless of library version or hardware.

3.5.2 Pixel-Based Input

Five models operate directly on pixel data. Each image is loaded from the compressed NumPy store as a 64×64 grayscale array and normalized to the range $[0, 1]$ by dividing by 255. Models that process a flat vector receive a 4,096-element array. Models that preserve spatial structure receive a $64 \times 64 \times 1$ tensor with an explicit channel dimension.

Per-dimension standardization using training-split mean and standard deviation is applied before any pixel-based model trains. This is computed on the training partition only and reused for validation and test without recomputation, which is the standard protocol for preventing data leakage.

3.5.3 Landmark-Based Input

Six models receive the 63-dimensional landmark vector from the precomputed CSV. At training time, lightweight preprocessing is applied on top of the raw coordinates:

- **Neural models** (standard MLP, deep MLP, GNN): z-score normalization is fit on the training split and applied identically to validation and test.

- **SVM and KNN:** a `StandardScaler` is wrapped in an `sklearn.pipeline.Pipeline` together with the classifier. This ensures that normalization is always computed from training data only, even when cross-validation is applied.
- **Random Forest:** raw landmark values are used directly. Tree-based methods are invariant to monotonic feature scaling, so normalization has no effect on their predictions.
- **GNN:** the flat 63-dimensional vector is reshaped into a 21×3 matrix where each row corresponds to one hand keypoint. A fixed adjacency matrix encoding the anatomical connections of the MediaPipe hand skeleton (fingertip to distal joint, distal to proximal, proximal to palm, and so on) is precomputed once and reused across all GNN training runs.

The only coordinate transformation applied on top of the raw MediaPipe output is this graph reshape for the GNN. There is no handedness canonicalization, no wrist-relative normalization, and no angular feature engineering. The downstream models work with the coordinates as MediaPipe returns them.

The key design insight behind the landmark representation is that it strips away everything irrelevant to hand shape. Background, lighting, skin tone, image quality, and camera properties all disappear. What remains is a compact geometric description of joint positions. If that description is sufficient to distinguish 18 gestures, then no complex model is needed on top of it.

3.6 Experimental Setup

3.6.1 Train, Validation, and Test Splits

The full dataset was divided into training, validation, and test partitions using a stratified approach, ensuring that each gesture class is proportionally represented in every split. The ratios used were 70% for training, 15% for validation, and 15% for testing. The split was computed class by class, with a minimum of one sample per class guaranteed in every partition. At the full training budget, the training partition contains **341,756** samples and the test set contains **73,250** samples. The validation set is used exclusively for early stopping and is never touched during final evaluation.

3.6.2 Training Fractions

To understand how each model behaves as a function of data volume, every configuration was evaluated at five smaller fractions of the training partition in addition to the full split:

1%, 10%, 25%, 50%, and 75%. Subsampling was applied only to the training portion; validation and test sets remained fixed across all fractions. This yields a view of data efficiency alongside peak performance. Table 3.2 shows the actual sample counts at each nominal fraction.

Table 3.2: Actual training-set sizes at each evaluated fraction under the stratified subsampling protocol.

Nominal fraction	Training samples
1%	3,426
10%	34,183
25%	85,448
50%	170,883
75%	256,324
100%	341,756

The differences between fractions are substantial. Going from 1% to 10% is a tenfold increase in training data. These are genuinely different data regimes.

3.6.3 Total Experiments

Eleven model configurations evaluated at six fractions each gives 66 experiments in total. The full campaign took several weeks of compute time to complete.

3.6.4 Shared Training Defaults

Neural models share a common training configuration: the same random seed, the same batch size of 64, and Stochastic Gradient Descent with a learning rate of 0.01 and momentum 0.9. Training runs for at most 200 epochs with early stopping: if the validation loss does not improve by at least 10^{-3} for six consecutive epochs, training halts and the best-performing checkpoint is restored. Deviations from these defaults are noted within the individual algorithm descriptions below. Classical models (SVM, Random Forest, KNN) have no training loop and are fitted in a single pass.

3.6.5 What Was Measured

Four performance metrics are reported for every run: classification accuracy, Macro-F1, weighted F1, and training time in seconds. Classifier-stage inference time per sample is

also recorded, since it directly determines whether a model can operate at real-time frame rates. For neural models, parameter count is reported as a proxy for model complexity and storage cost.

3.7 Model Design and Implementation

This section documents the design rationale and concrete implementation of each of the eleven evaluated configurations. For each model, the discussion covers why the architecture was chosen or designed as it was, followed by the pseudocode describing its implementation logic.

3.7.1 Models Operating on Raw Pixels

All pixel-based models receive the same 64×64 grayscale input. They represent the baseline question: how well can a model classify gestures when given only raw image data and no geometric prior? The motivation for including them is comparative: they allow the contribution of the landmark representation to be isolated and quantified.

Standard MLP

The standard MLP is the simplest neural baseline. It serves as a lower bound for what a feedforward network with minimal capacity can achieve on raw pixels, and as a reference point for assessing the value of additional depth, convolution, and attention.

The architecture consists of two hidden layers of sizes 128 and 64, chosen to give the model enough capacity to learn basic visual statistics without overfitting at small data fractions. The input is a flat 4,096-element vector, which means all spatial relationships between pixels are discarded immediately. This is a known limitation of the design, but it keeps the model simple and interpretable as a baseline.

Algorithm 1 Standard MLP on raw pixels

Require: Grayscale images resized to 64×64 , two-layer hidden stack [128, 64]

- 1: Flatten each image to a 4096-element vector and scale values to $[0, 1]$
 - 2: Apply the 70/15/15 stratified split; subsample training set to the chosen fraction
 - 3: Standardize each input dimension using the training-split mean and standard deviation
 - 4: Build the network: $\text{Input}(4096) \rightarrow \text{Dense}(128) \rightarrow \text{ReLU} \rightarrow \text{Dense}(64) \rightarrow \text{ReLU} \rightarrow \text{Dense}(18) \rightarrow \text{Softmax}$
 - 5: Train with cross-entropy loss, SGD ($\text{lr} = 0.01$, $\text{momentum} = 0.9$), batch 64, patience 6
 - 6: Restore the best validation checkpoint and evaluate on the held-out test set
 - 7: Record accuracy, Macro-F1, weighted F1, training time, inference time, and parameter count
-

Deep MLP

The deep MLP extends the standard baseline with five hidden layers of sizes [1024, 512, 256, 128, 64], batch normalization after each layer to stabilize training, and dropout at rate 0.3 to reduce overfitting. The motivation for this configuration is to test whether simply adding depth and regularization to a pixel-based feedforward network can close the gap to more structured architectures. Batch normalization was added because deep MLPs without it tend to exhibit unstable gradient flow, particularly on high-dimensional flat inputs.

Algorithm 2 Deep MLP on raw pixels

Require: Grayscale images resized to 64×64 , five-layer hidden stack [1024, 512, 256, 128, 64]

- 1: Flatten each image to a 4096-element vector and scale to $[0, 1]$
 - 2: Apply the 70/15/15 stratified split
 - 3: Standardize using training-split statistics only
 - 4: Build the network with batch normalisation and dropout (rate 0.3) after each hidden layer
 - 5: Train with cross-entropy loss, SGD defaults, patience 6
 - 6: Restore best checkpoint and evaluate
-

Convolutional Neural Network

The CNN is designed to preserve and exploit the spatial structure that the MLP baselines discard. Convolutional layers apply local filters that detect edges, textures, and shape fragments, and are therefore a natural fit for image classification. The architecture was kept deliberately lightweight: two convolutional blocks (8 and 16 filters respectively, each with a 3×3 kernel followed by 2×2 max pooling) and a single dense head of size 64. This choice reflects the input resolution: at 64×64 , a deep convolutional stack would rapidly reduce the spatial dimension to a point where additional layers add little value.

The learning rate was set to 0.012 (slightly higher than the MLP default) and patience was reduced to 4, reflecting the CNN's faster convergence relative to the fully connected models.

Algorithm 3 Convolutional Neural Network (CNN)

Require: Images shaped as $64 \times 64 \times 1$ tensors, two convolutional blocks plus one dense head

- 1: Load images, normalise to $[0, 1]$, add the channel dimension
 - 2: Apply the 70/15/15 split; standardise using training statistics
 - 3: Build the network:
 - Conv(8 filters, 3×3) $\rightarrow$ ReLU $\rightarrow$ MaxPool(2×2)
 - Conv(16 filters, 3×3) $\rightarrow$ ReLU $\rightarrow$ MaxPool(2×2)
 - Flatten $\rightarrow$ Dense(64) $\rightarrow$ ReLU $\rightarrow$ Dense(18) $\rightarrow$ Softmax
 - 4: Train with cross-entropy, SGD (lr = 0.012, momentum = 0.9), batch 64, patience 4
 - 5: Restore best checkpoint and evaluate
-

MLP-CNN Fusion Model

The fusion model was designed to test whether combining convolutional feature extraction with direct pixel statistics in a parallel branch yields any complementary benefit. The hypothesis is that convolutional filters capture local spatial patterns while the dense branch captures global pixel statistics that the CNN may suppress through pooling. Both branches share the same input image. Their outputs are concatenated before the final classification head.

The batch size was doubled to 128 relative to the standard CNN default, since the wider combined feature vector benefits from a larger gradient estimate per update step.

Algorithm 4 MLP-CNN Fusion Model

Require: Images shaped as $64 \times 64 \times 1$, two parallel branches combined before the output

- 1: Normalise images to $[0, 1]$; apply the split and standardise
 - 2: Build two branches on the same input simultaneously:
 CNN branch: Conv(8) $\rightarrow$ Pool $\rightarrow$ Conv(16) $\rightarrow$ Pool $\rightarrow$ Flatten
 Dense branch: Flatten pixels $\rightarrow$ Dense(128) $\rightarrow$ ReLU
 - 3: Concatenate both branch outputs, then Dense(96) $\rightarrow$ ReLU $\rightarrow$ Dense(18) $\rightarrow$ Softmax
 - 4: Train with cross-entropy, SGD (lr = 0.012), batch 128, patience 4
 - 5: Restore best checkpoint and evaluate
-

Vision Transformer

The Vision Transformer (ViT) was included to represent the class of attention-based architectures that have achieved state-of-the-art results in image classification. Unlike CNNs, a ViT does not impose local inductive biases: it divides the image into fixed-size patches and learns relationships between all patches simultaneously through self-attention. This global receptive field from the first layer makes ViTs potentially powerful for tasks where relevant information is distributed across the image, but also more data-hungry in general.

The configuration was scaled down substantially relative to standard ViT implementations to match the 64×64 input resolution and the available compute budget: patch size 8 (giving 64 non-overlapping patches), embedding dimension 64, 4 attention heads, and 4 Transformer layers with a feedforward hidden size of 128. Adam was used instead of SGD because ViTs are known to benefit from adaptive learning rates, and the patience was extended to 8 to allow longer convergence windows.

Algorithm 5 Vision Transformer (ViT)**Require:** Images shaped as $64 \times 64 \times 1$; patch size 8; embedding dim 64; 4 attention heads; 4 layers

- 1: Normalise and split as above
- 2: Divide each image into non-overlapping 8×8 patches; project each patch to a 64-dimensional vector
- 3: Add a learnable position encoding to each patch vector
- 4: **for** each of the 4 Transformer layers **do**
- 5: LayerNorm $\rightarrow$ multi-head self-attention (4 heads) $\rightarrow$ residual add
- 6: LayerNorm $\rightarrow$ feedforward network (hidden size 128, GELU activation) $\rightarrow$ residual add
- 7: **end for**
- 8: Average-pool all patch representations $\rightarrow$ Dense(18) $\rightarrow$ Softmax
- 9: Train with cross-entropy, Adam ($\text{lr} = 5 \times 10^{-4}$), batch 32, patience 8, max 80 epochs
- 10: Restore best checkpoint and evaluate

3.7.2 Models Operating on MediaPipe Landmarks

All six landmark-based models receive the same 63-dimensional input vector extracted from the precomputed CSV. Their purpose is to answer a different question from the pixel-based group: how much of the classification problem is already solved by the geometric structure of the hand, before any complex model is applied?

Standard and Deep MLP on Landmarks

The landmark MLP mirrors the pixel MLP in structure but operates on a 63-dimensional input rather than a 4,096-dimensional one. This makes it a much smaller model by parameter count. The standard configuration uses hidden layers [128, 64]; the deep version uses [512, 256, 128, 64] with batch normalization and dropout, for the same reasons as the pixel-based deep MLP. The comparison between these two and their pixel-based counterparts isolates the effect of the input representation at equivalent levels of model complexity.

Algorithm 6 Standard and Deep MLP on MediaPipe landmarks**Require:** 63-dimensional landmark vectors; two hidden-size options: [128, 64] (standard) or [512, 256, 128, 64] (deep)

- 1: Load landmark table and apply the 70/15/15 split
- 2: Fit z-score normalisation on the training split; apply to validation and test
- 3: Build the chosen MLP; deep version adds batch normalisation and dropout (0.3) per layer
- 4: Train with cross-entropy, SGD ($\text{lr} = 0.01$), batch 64, patience 6, max 200 epochs
- 5: Restore best checkpoint and evaluate

Graph Neural Network

The GNN was included to test whether explicitly encoding the anatomical structure of the hand as a graph provides any advantage over a flat landmark vector. The hypothesis is that the hierarchical connectivity of the hand skeleton (fingertip joints depend on proximal joints,

which connect to the palm) is informative, and that graph convolution can exploit this structure in a way that a flat MLP cannot.

The implementation constructs a graph where each of the 21 MediaPipe keypoints is a node with a 3-dimensional feature vector (its x, y, z coordinates), and edges follow the MediaPipe hand skeleton topology. Two graph convolutional layers of size 64 are applied, each using the symmetrically normalized adjacency matrix $\hat{A} = D^{-1/2}(A + I)D^{-1/2}$ where D is the degree matrix and I is the identity. Global average pooling aggregates the node representations into a fixed-size graph embedding, which is then classified by a small dense head.

Adam was used instead of SGD, and patience was extended to 8, consistent with the ViT configuration and for the same reasons: graph convolution networks are sensitive to learning rate and benefit from a longer early-stopping window.

Algorithm 7 Graph Neural Network on MediaPipe landmarks (GNN)

Require: 63-dimensional landmark vectors reshaped to 21×3 node features; fixed hand-graph edges

- 1: Apply the 70/15/15 split and fit z-score normalisation on training
 - 2: Precompute the symmetrically normalised adjacency matrix $\hat{A} = D^{-1/2}(A + I)D^{-1/2}$
 - 3: Apply: GCN(64) $\rightarrow$ GCN(64) $\rightarrow$ global average pooling $\rightarrow$ Dense(64) $\rightarrow$ Dense(18) $\rightarrow$ Softmax
 - 4: Train with cross-entropy, Adam ($\text{lr} = 5 \times 10^{-4}$), batch 32, patience 8, max 200 epochs
 - 5: Restore best checkpoint and evaluate
-

Classical Classifiers: SVM, Random Forest, and KNN

Three classical machine learning classifiers were implemented on the landmark vectors as non-neural baselines. Their inclusion tests whether the classification problem, given a good input representation, is simple enough that it does not require gradient-based learning at all.

The SVM uses a radial basis function (RBF) kernel with $C = 10$ and $\gamma = \text{scale}$. This configuration was chosen as a strong default for a normalized 63-dimensional input: the RBF kernel can model nonlinear boundaries efficiently, and $C = 10$ allows moderate margin violation without underfitting.

The Random Forest uses 200 trees with no maximum depth and a minimum of one sample per leaf. Deep trees are appropriate here because the feature space is compact and well-structured; overfitting is controlled by ensemble averaging rather than depth restrictions.

KNN uses $k = 5$ with Euclidean distance and distance-weighted voting, which reduces the influence of farther neighbors. In a 63-dimensional space where landmark vectors are z-score-normalized, Euclidean distance is a reasonable proximity measure.

Algorithm 8 Classical classifiers on MediaPipe landmarks (SVM, Random Forest, KNN)**Require:** 63-dimensional landmark vectors; estimator $E \in \{\text{SVM}, \text{RF}, \text{KNN}\}$ **Require:** SVM: $C = 10$, RBF kernel, $\gamma = \text{scale}$ **Require:** RF: 200 trees, no maximum depth, minimum 1 sample per leaf**Require:** KNN: $k = 5$, Euclidean distance, distance-weighted voting

1: Apply the 70/15/15 split

2: **if** E is SVM or KNN **then**3: Build pipeline: z-score normalisation $\rightarrow E$ 4: **else**

5: Use raw landmark values directly (Random Forest is not sensitive to scale)

6: **end if**

7: Fit the estimator on the training split

8: Predict on the test split and record accuracy, Macro-F1, weighted F1, and inference time

3.8 Results at the Full Training Budget

Table 3.3 reports the measured performance and efficiency of all eleven configurations when trained on the full available training set.

Table 3.3: Full-split accuracy, F1 scores, training time, inference latency, and parameter count for all eleven configurations, sorted by accuracy.

Model	Input	Acc. (%)	Macro F1	Wtd. F1	Train (s)	Infer. (ms)	Parameters
MediaPipe + MLP	Landmarks	95.10	0.9514	0.9515	242	0.021	17,618
MediaPipe + Deep MLP	Landmarks	94.80	0.9492	0.9493	498	0.032	209,298
MediaPipe + SVM	Landmarks	93.74	0.9376	0.9377	853	6.071	4,248,594
Vision Transformer	Pixels	92.19	0.9220	0.9220	14,680	0.920	139,346
CNN	Pixels	91.21	0.9123	0.9123	758	0.117	203,186
MLP-CNN Fusion	Pixels	91.09	0.9113	0.9113	768	0.122	840,850
Deep MLP (raw pixels)	Pixels	88.57	0.8858	0.8859	5,473	0.174	4,899,730
MediaPipe + Random Forest	Landmarks	87.95	0.8796	0.8795	105	0.013	19,779,452
MediaPipe + KNN	Landmarks	80.97	0.8110	0.8105	0	0.353	21,530,628
Standard MLP (raw pixels)	Pixels	80.51	0.8058	0.8060	448	0.052	1,084,050
MediaPipe + GNN	Hand graph	50.24	0.4998	0.4997	3,666	0.031	9,746

Several things stand out immediately.

The clear winner is **MediaPipe + MLP**. It reaches **95.10% accuracy** and a Macro-F1 of **0.9514**. What makes this striking is not just the number, but the fact that this model achieves it with only **17,618 parameters**, which is a remarkably compact model. The second-best configuration, the deep landmark MLP, uses nearly twelve times as many parameters and still falls short by almost a third of a percentage point. The design decision to use a small two-layer MLP on top of a clean geometric representation was therefore validated: additional model capacity does not help when the representation is already good.

The Vision Transformer reaches **92.19%**, which is competitive, but it required over four hours to train at the full split: roughly sixty times longer than the selected MLP. Whether that extra training cost and the added inference latency are worth a 2.9-point accuracy drop is a question the efficiency analysis below answers clearly.

The SVM on landmarks reaches **93.74%**, which is the third-best result by accuracy. But

its inference time is **6.071 ms per sample**, approximately 289 times slower than the selected MLP. For any application that needs to respond in real time, that gap rules the SVM out entirely.

The GNN is a structural failure. **50.24% accuracy** on an 18-class problem is barely above chance (a uniformly random guesser would score around 5.5%). The hypothesis that imposing an anatomical graph structure on the keypoints would help was not confirmed. Instead, the graph convolution operation appears to have introduced inductive biases that constrain the model in ways that hurt rather than help. This is discussed further in the scaling analysis.

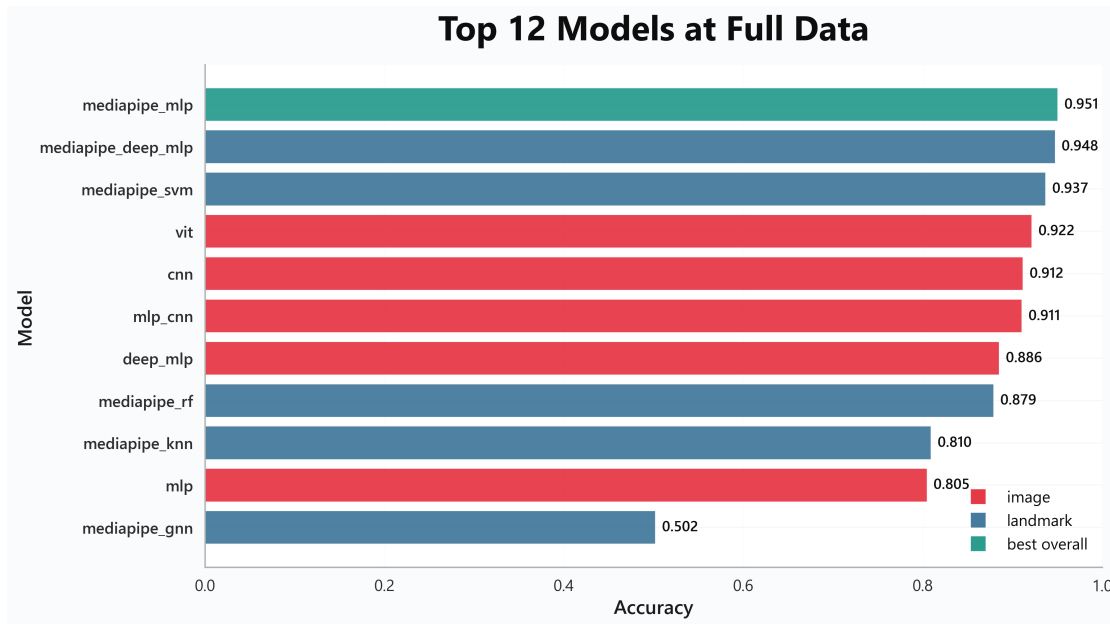


Figure 3.2: Accuracy and Macro-F1 of all eleven configurations at the full training split.

Figure 3.2 reinforces the ranking visually. The top of the chart is firmly occupied by landmark-based neural models, and the gap between them and the image-based alternatives is consistent across both accuracy and Macro-F1.

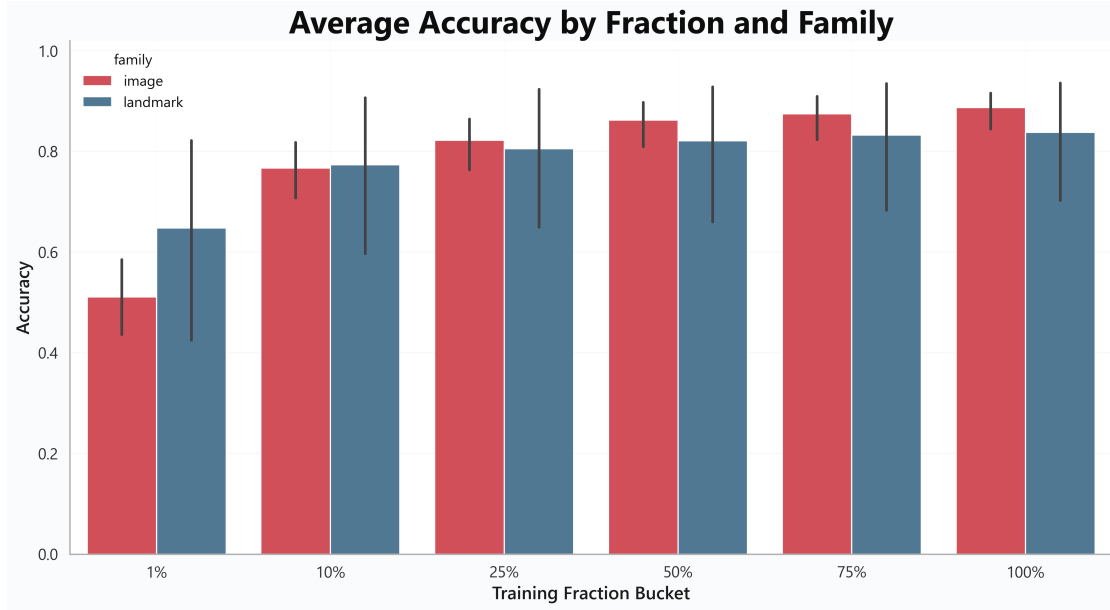


Figure 3.3: Full-split accuracy distribution by model family.

Figure 3.3 groups results by input type. Landmark-based models are not merely competitive with image-based ones: they dominate. All three of the top configurations use landmarks. The image-based models, including the ViT, fall below every landmark-neural model despite consuming substantially more resources. This is the core empirical finding: the design decision to use landmark coordinates as input, rather than raw pixels, has a larger effect on performance than any choice of model architecture.

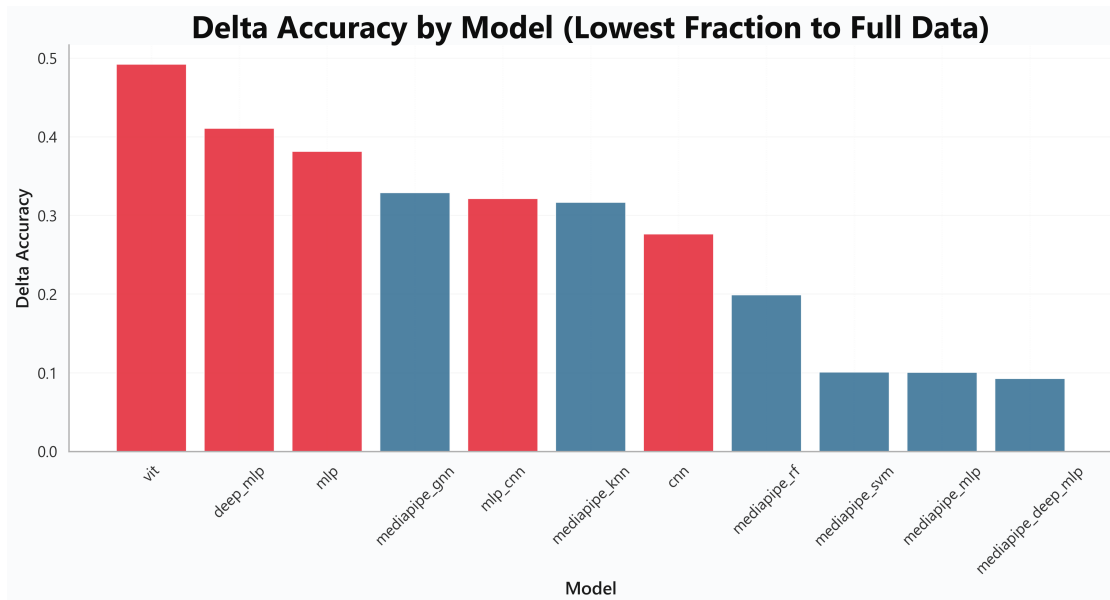


Figure 3.4: Accuracy gap of each configuration relative to the best measured result.

Figure 3.4 expresses each model as a distance from the best. The gap to the deep landmark MLP is small, around 0.3 points. The gap to the ViT is nearly 3 points. The gap to the CNN

and MLP-CNN fusion is around 4 points. And the gap to the GNN is nearly 45 points. These are not measurement noise; they are structurally meaningful differences that persist across training fractions.

3.9 Speed, Size, and the Cost of Accuracy

A high accuracy number alone is not sufficient to recommend a model for deployment. The question is whether the system can respond fast enough that a real user does not notice any delay. Figure 3.5 maps every configuration onto an accuracy-versus-latency plot, with colour encoding the parameter count.

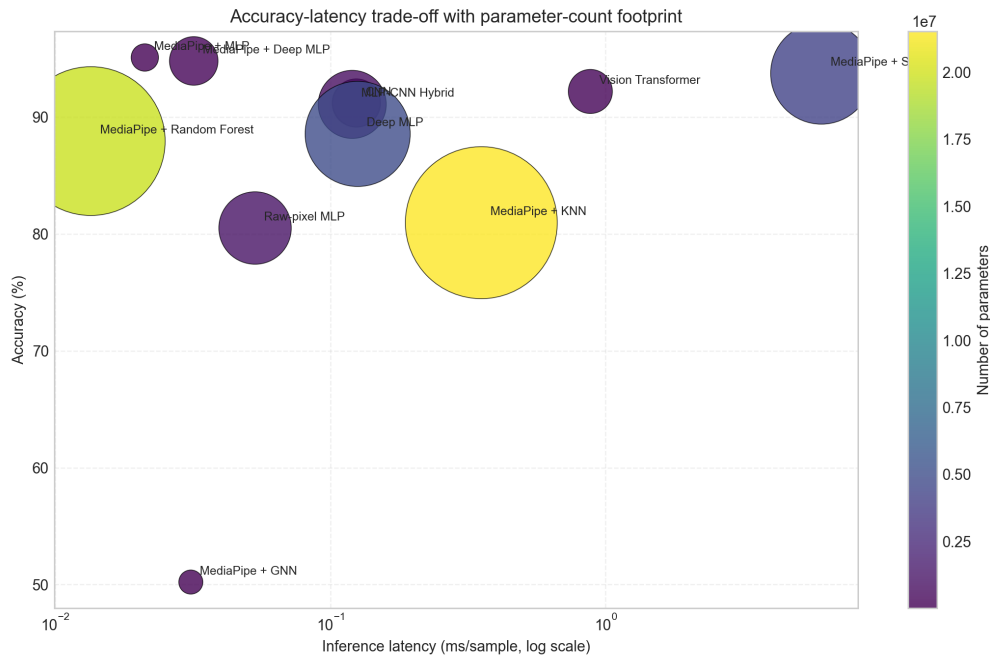


Figure 3.5: Accuracy versus classifier-stage inference latency at the full split.

The selected model occupies the most favourable position on this plot. *MediaPipe + MLP* is both the most accurate and among the fastest at inference time. Its **0.021 ms** per sample at the classifier stage is roughly 1.5 times faster than the deep landmark MLP, roughly 6 times faster than the CNN, and about 289 times faster than the SVM. The Random Forest is marginally faster at **0.013 ms**, but that speed advantage costs over seven accuracy points, a trade that makes no practical sense.

The SVM is worth pausing on. It achieves strong accuracy, but 6 ms per sample means that classifying a single frame takes six milliseconds at the classifier stage alone, before accounting for MediaPipe landmark extraction time. At real-time frame rates this accumulates into a perceptible lag. It is suitable for offline batch processing but not for responsive interactive use.

The ViT is accurate and compact in terms of learned parameters, but slow both to train and to run. It is not a practical choice for this deployment context.

3.10 Behavior Under Reduced Data

Training at the full split reveals which model is best when all available data is used. Scaling experiments across six progressively smaller data subsets reveal something more: how quickly each model learns from limited data, and how robust each design is when the training budget is constrained.

Table 3.5 shows how the selected model performs across all six fractions.

Table 3.5: MediaPipe + MLP performance across six training fractions from 1% to 100%.

Fraction	Training samples	Acc. (%)	Macro F1	Wtd. F1	Train (s)
1%	3,426	85.03	0.8503	0.8500	45
10%	34,183	93.03	0.9305	0.9306	43
25%	85,448	94.20	0.9421	0.9423	97
50%	170,883	94.68	0.9471	0.9472	145
75%	256,324	94.82	0.9487	0.9488	167
100%	341,756	95.10	0.9514	0.9515	242

The results illustrate a key property of the landmark design. At just 1% of the training data, 3,426 samples across 18 classes, the model still reaches **85.03% accuracy**. This reflects something fundamental about the landmark representation: when the input is a compact geometric description of joint positions, a small number of examples is sufficient to learn the basic separation between hand shapes. Pixel-based models, which must learn which image regions are relevant before they can classify, require far more data to reach a comparable operating point. The jump from 1% to 10% is almost 8 accuracy points. After that, returns diminish rapidly. Going from 10% to 100% gains only 2 points, and from 50% to 100% gains less than half a point. The model is effectively saturated well before the full dataset is used.

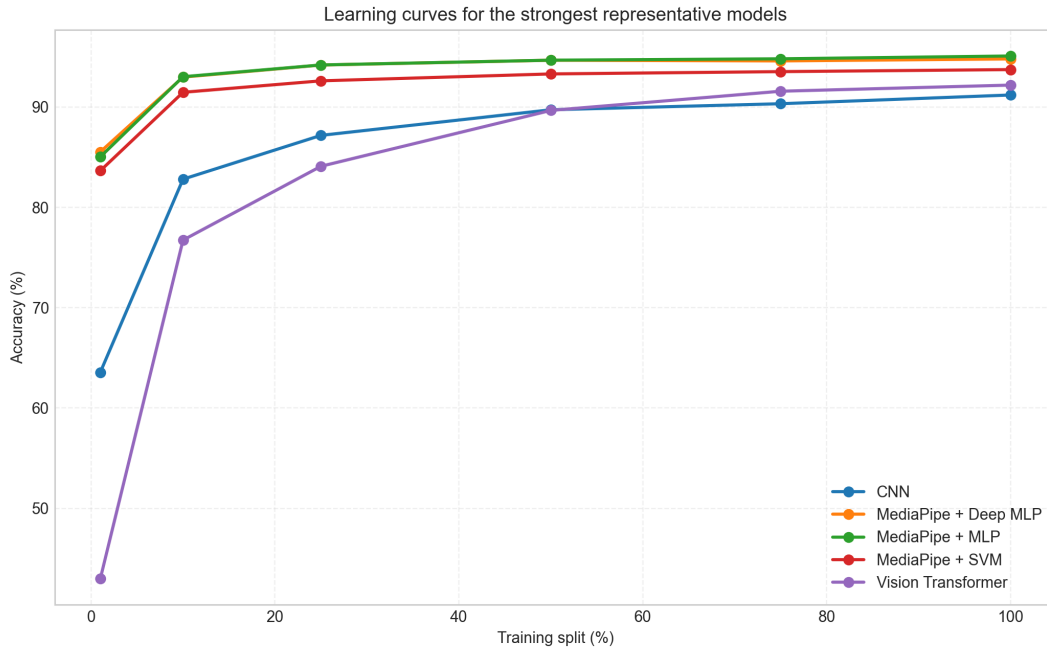


Figure 3.6: Training-fraction accuracy for the strongest representative of each model family.

Figure 3.6 plots the scaling trajectories of the best configurations together. The *MediaPipe* + *MLP* leads from the 10% split onward. Only at the smallest fraction does the deep landmark MLP occasionally edge ahead, and that advantage vanishes once data is no longer scarce. The image-based models improve more slowly and plateau lower. The GNN's trajectory is flat and far below everything else regardless of how much data it receives. This confirms that the GNN is not underperforming because of data scarcity; it is structurally mismatched to the task.

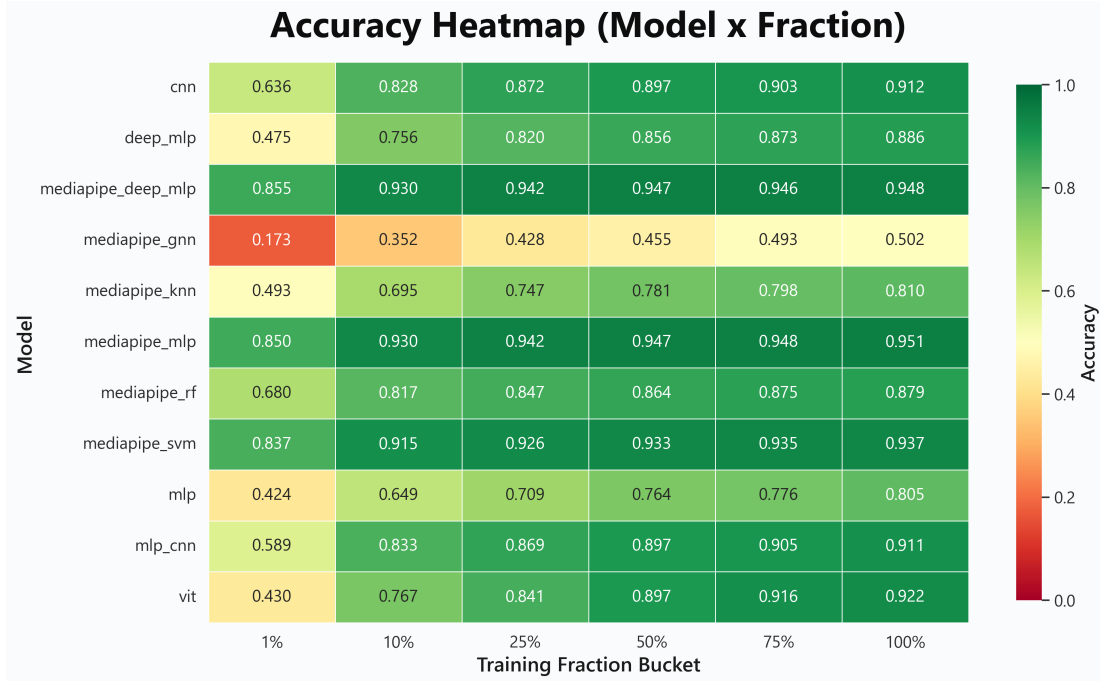


Figure 3.7: Full accuracy matrix across all eleven models and six training fractions.

Figure 3.7 shows the complete picture in a single grid. High-accuracy cells are concentrated in the landmark-based neural configurations and appear early, from the 10% fraction onward. Image-based models improve but never reach the same band. The GNN row is uniformly low regardless of how much data it receives.

3.11 Error Analysis of the Selected Model

Examining where a model fails is at least as informative as examining where it succeeds, and it is necessary for understanding the limits of the implemented design. Figure 3.8 shows the confusion matrix of *MediaPipe* + *MLP* on the full test set.

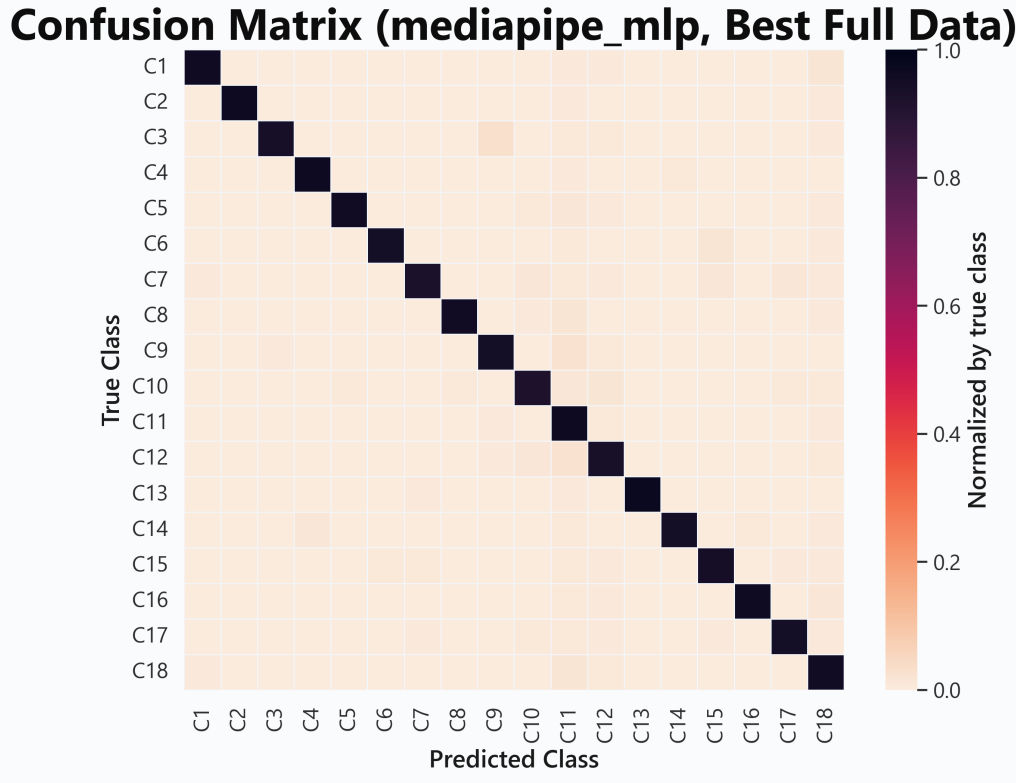


Figure 3.8: Confusion matrix of the selected MediaPipe + MLP model at the full training split

The diagonal is strongly dominant. For almost all 18 classes, the vast majority of predictions land on the correct label, and per-class recall stays consistently above 90%. The off-diagonal errors are not randomly scattered. They are concentrated in a small number of specific class pairs and tend to be directional: one class is absorbed into another more readily than vice versa.

That directionality reveals the structural limit of the landmark representation. When two gesture configurations share similar overall joint arrangements, say, when both involve the same number of extended fingers and differ only in subtle angular variations, the 63-dimensional coordinate vector may not be sufficient to distinguish them reliably. These errors are not arbitrary model failures. They are a direct consequence of representing a hand purely by joint coordinates without any higher-level angular or relational features.

This analysis points to a concrete direction for improvement. Targeted data augmentation for the most confusable gesture pairs, introducing controlled variations in joint angles specifically for those classes, could reduce residual errors without any change to the core architecture. Alternatively, augmenting the 63-dimensional input with derived features such as inter-joint angles or finger extension ratios could make the confusable pairs more separable in feature space.

3.12 The Implementation Decision

The evidence across all six evaluation sections above is consistent. **MediaPipe + MLP** is selected as the final model for deployment in the application described in Chapter 4.

The justification is cumulative across every dimension that matters for deployment:

- **Accuracy.** 95.10%, the highest measured result across all eleven configurations, including architecturally more complex ones.
- **Model size.** 17,618 parameters. The CNN uses eleven times as many. The deep landmark MLP uses twelve times as many. The standard raw-pixel MLP uses sixty times as many. None of them perform as well.
- **Inference speed.** 0.021 ms at the classifier stage. Fast enough to classify multiple frames per second without accumulating latency.
- **Data efficiency.** The model reaches a stable and competitive operating point at roughly 10% of the available training data. It does not depend on the full dataset to function well.
- **Training cost.** 242 seconds at the full split, compared to over four hours for the ViT.

This is not a close call. The only configuration that is faster at inference is Random Forest, and only marginally. KNN has lower fitting cost, but neither model approaches the accuracy of the selected one.

The broader lesson from this experimental campaign concerns representation design. The gap between landmark-based and pixel-based models is not explained by model capacity or training budget. The ViT is a substantially more powerful architecture than a two-layer MLP, and it still loses by nearly 3 percentage points. What the MLP has that the ViT does not is a compact, geometrically structured input that already encodes the information most relevant to hand shape. This observation validates the central design choice made in Section 3.5: investing engineering effort in building a good input representation is more effective than investing it in building a more complex model.

3.13 Conclusion

The preceding sections have documented the design and implementation of a complete hand gesture recognition pipeline, from the initial data acquisition scripts through to model selection and error analysis.

The dataset construction pipeline, built around `icrawler` for image acquisition and a HaGRID-trained YOLO model for auto-labeling and hand detection, produced a dataset of

approximately 500,000 labeled samples covering 18 gesture classes, with diversity in skin tone, lighting, and viewpoint that no suitable public dataset provided. The architectural decision to precompute MediaPipe landmarks into a flat CSV file and decouple that extraction step from model training was a practical design choice that made a campaign of 66 training experiments feasible.

The experimental results lead to a clear and well-supported conclusion. When evaluated across accuracy, model size, inference speed, training cost, and data efficiency, the *MediaPipe + MLP* configuration outperforms or matches all ten alternatives on every criterion that matters for practical deployment. The learning-curve analysis shows that this superiority is structural: the model reaches near-peak performance quickly and does not require a large training budget to be effective. The error analysis shows that the remaining mistakes are geometrically motivated and concentrated in a small number of visually similar gesture pairs, pointing toward targeted data augmentation or derived angular features as the most promising directions for future improvement.

With the final model selected and its behavior fully characterized, the question shifts from evaluation to deployment. The next chapter describes how this model was integrated into a concrete desktop application.

Application Development and System Architecture

Chapter 4

Application Development and System Architecture

4.1 Introduction

Building a model that achieves strong benchmark numbers is one thing. Putting it in front of a real user, running in real time, on a real machine, is quite another. This chapter documents the full journey from a first rough prototype to the final application, covering the dead ends, the stack switches, and the reasons behind each decision. Then it describes the architecture and feature set of the application that resulted.

The chapter is organised as follows. Section 4.2 walks through the three prototyping stages that preceded the final implementation. Section 4.3 explains the technology choices made for the final version. Section 4.4 describes the overall system architecture. Sections 4.5 through 4.10 document each section of the application in the order a user would encounter them. Section 4.11 describes the development environment. Section 4.12 closes the chapter.

4.2 The Path to the Final Stack

Software projects rarely arrive at their final form immediately. This one went through three distinct prototyping phases before settling into its current shape. Each phase taught something, and each switch was made for a concrete reason rather than out of preference.

4.2.1 Phase One: Python and Tkinter

The first prototype was a Python desktop application built with Tkinter, the standard GUI toolkit that ships with Python. The goal at that stage was simply to verify that the full pipeline worked end to end: camera input, landmark extraction, classification, and a visual response displayed on screen. Tkinter is not glamorous, but it is fast to set up and requires no additional

dependencies beyond what was already in the project.

This phase confirmed that the recognition pipeline itself was functional. Frame capture worked. MediaPipe ran without issues on the test machine. The MLP classifier responded in well under a millisecond per frame. But Tkinter has real limitations. Layout control is crude compared to modern toolkit alternatives. There is no clean way to build the kind of structured multi-page experience the final application needed. And the interface, even after significant effort, looked like a debugging tool rather than an application a normal user would willingly sit in front of.

The prototype served its purpose. It was time to move on.

Figure 4.1 shows the first prototype interface used during this initial validation stage.

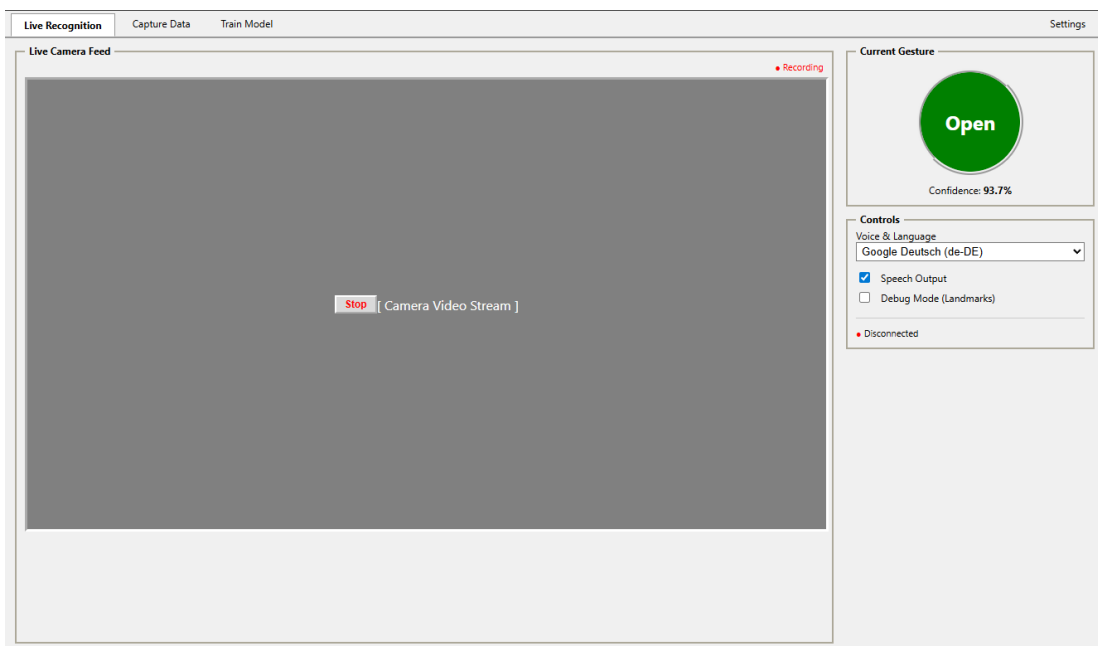


Figure 4.1: Phase one Python and Tkinter prototype.

4.2.2 Phase Two: FastAPI and Jinja2

The second attempt moved to a web-based approach. FastAPI handled the backend (model loading, inference, and route serving) while Jinja2 rendered the HTML pages server-side. The appeal was clear: a browser-based interface is immediately more flexible than Tkinter, and FastAPI is genuinely fast and ergonomic for Python developers.

This worked, to a point. Static pages loaded fine. Simple request-response interactions were clean. The problem appeared as soon as real-time gesture recognition came into scope. Showing the user their own hands being classified in real time requires a live connection between the browser and the server, a connection that needs to stream data continuously rather than send one request and wait for one response. FastAPI supports WebSockets, but getting stable, low-latency bidirectional streaming to work reliably in the browser proved

frustratingly difficult. Connection management, frame timing, and browser-side coordination all became sources of instability. The more the system was pushed toward real-time performance, the more fragile the WebSocket layer became.

Server-side rendering was not the right architecture for this problem.

Figure 4.2 shows the second prototype, where the interface moved from a desktop widget toolkit to a browser-rendered application served by FastAPI and Jinja2.

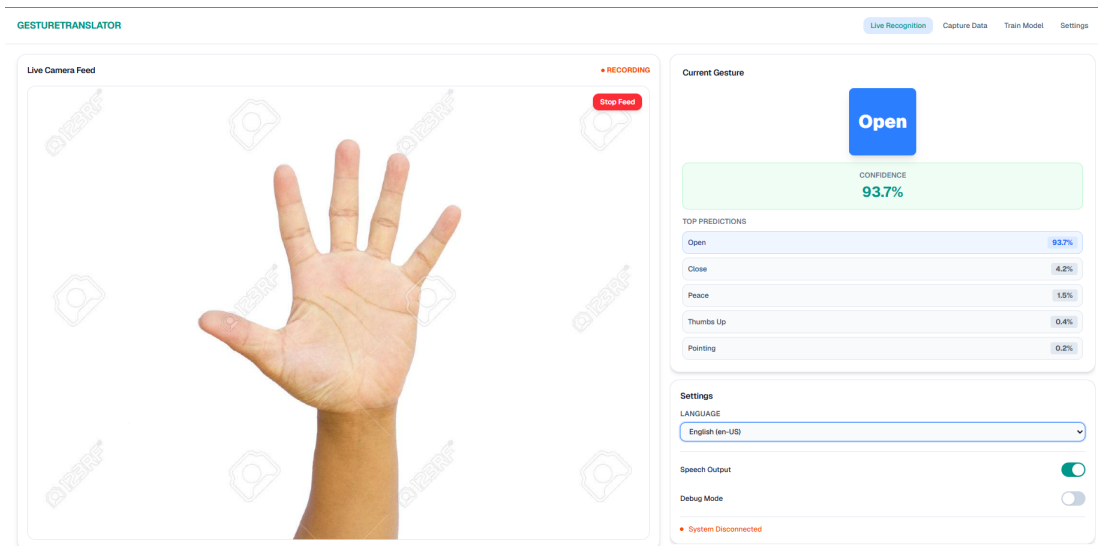


Figure 4.2: Phase two FastAPI and Jinja2 prototype.

4.2.3 Phase Three: Next.js and FastAPI

The third attempt separated the frontend more cleanly from the backend. Next.js took over the interface, which gave access to a modern React environment with full client-side capabilities, including direct access to the browser's camera API and the ability to process video frames locally in JavaScript before sending anything to the server. FastAPI remained on the backend for inference.

This was a significant improvement. The interface became far more fluid, and moving frame processing to the browser side reduced the round-trip cost considerably. But the WebSocket problem did not disappear. It became more manageable, but the fundamental issue remained: a Python web server handling a continuous stream of frames from an active camera at interactive frame rates is fighting against its own architecture. Python's concurrency model and the overhead of the FastAPI stack introduced latency that was acceptable in isolation but visible over sustained use. More practically, distributing a Next.js frontend and a FastAPI backend as a unified end-user application is not straightforward. The two services need to be started separately, port conflicts become a real concern, and packaging everything for a user who just wants to open a program and use it requires non-trivial tooling.

At this point, the question was no longer how to make the web approach work. It was whether the web approach was the right model at all.

4.2.4 The Decision to Go Desktop

A desktop application eliminates most of the distribution and real-time streaming problems at once. There is no server to start, no port to configure, no browser to open. The application is a single binary that the user launches. Camera access is direct. There is no network layer between the interface and the inference logic. And a modern desktop framework can deliver a user experience that is indistinguishable from a web application in terms of visual quality.

The chosen framework is **Tauri**: a toolkit for building desktop applications using a Rust backend and a standard web frontend. The final stack is described in detail in the next section.

4.3 The Final Technology Stack

Table 4.1 lists every major component of the final application and what role it plays.

Table 4.1: Final technology stack.

Component	Role	Why it was chosen
Tauri	Desktop application framework	Wraps a Rust backend and a web frontend into a single native binary; eliminates distribution and server-startup problems; direct OS-level camera and file access
Rust	Backend logic and inference service	Fast, safe, and predictable; no garbage-collection pauses; well suited to handling classification requests at interactive frame rates
React	Frontend UI framework	Component-based architecture; enables complex, stateful interfaces while reusing the same web technology skills from the earlier prototypes
TanStack Router	Client-side routing	Type-safe, file-based routing for multi-page navigation inside the desktop application
TanStack Query	Server-state management	Handles data fetching, caching, and synchronisation between the frontend and backend in a structured way

(continued on next page)

(continued from previous page)

Component	Role	Why it was chosen
Tailwind CSS	Styling	Utility-class styling system; fast to prototype with and consistent across components
shadcn/ui	UI component library	Prebuilt accessible components (buttons, dialogs, forms, cards, progress bars) that integrate directly with Tailwind
Three.js	3D visualisation	Renders an interactive three-dimensional view of the trained neural network, showing nodes, weighted connections, and layer structure
MediaPipe Hands	Hand landmark extraction	Same upstream tool used throughout the experiments; extracts the 21-point landmark vector from each camera frame
Neon (PostgreSQL)	Remote data storage	Serverless PostgreSQL database that stores user accounts, gesture labels, landmark samples, and trained models; accessible from any device
Web Speech API	Text-to-speech output	Converts the recognised gesture label into spoken audio in the translation interface; available natively in the browser environment Tauri wraps

The monorepo is managed with **Turborepo**, which keeps frontend code, backend code, and shared configuration in one repository with coordinated build pipelines. This makes it possible to develop both sides simultaneously without maintaining separate dependency trees or manually synchronising shared types and constants.

4.4 Overall System Architecture

Figure 4.3 shows how the application is organised at the system level.

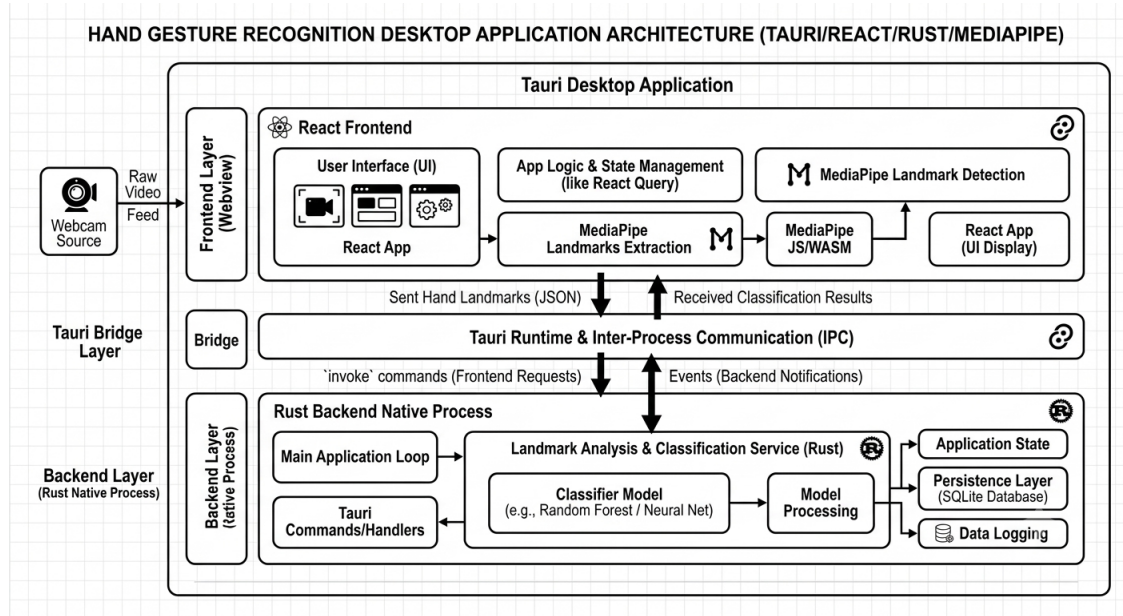


Figure 4.3: Desktop application architecture.

The Tauri shell is the envelope that holds everything together. Inside it, the React frontend runs in a lightweight embedded browser window and handles all interaction with the user. The Rust backend runs as a native process on the same machine and handles model loading, training orchestration, file management, and inference. Communication between the two sides happens through Tauri's command interface, a message-passing system that is fast, type-safe, and does not require opening any network ports.

The camera pipeline starts in the frontend. Each video frame is captured by the browser's camera API, passed to MediaPipe Hands running in WebAssembly, and the resulting 63-dimensional landmark vector is extracted entirely on the client side. That vector is what gets sent to the Rust backend for classification. This keeps the data volume exchanged between frontend and backend very small and avoids any round-trip latency that would be noticeable to the user.

The backend applies the stored normalisation statistics and runs the selected MLP classifier implemented in pure Rust, with no external machine learning library dependencies. The predicted label is returned to the frontend, which displays the result and, in the translation mode, passes the label to the speech synthesiser.

4.4.1 Data Storage

The application uses a hybrid storage model designed to support both connected and offline use. User accounts, gesture labels, landmark samples, and trained models are persisted to a **Neon** serverless PostgreSQL database, making the data accessible across sessions and devices. At the same time, the Rust backend maintains a local **SQLite** database managed with

rusqlite, which serves as the working store during an active session. This means the application remains fully functional without a network connection: the local database is the source of truth at runtime, and synchronisation with Neon handles persistence beyond the local machine.

All data is namespaced by user account. Labels, captured samples, trained models, and training history are all scoped to the authenticated user, so multiple users on the same machine maintain completely independent setups.

4.5 Authentication: Getting In

The first screen a new user sees is the authentication page.

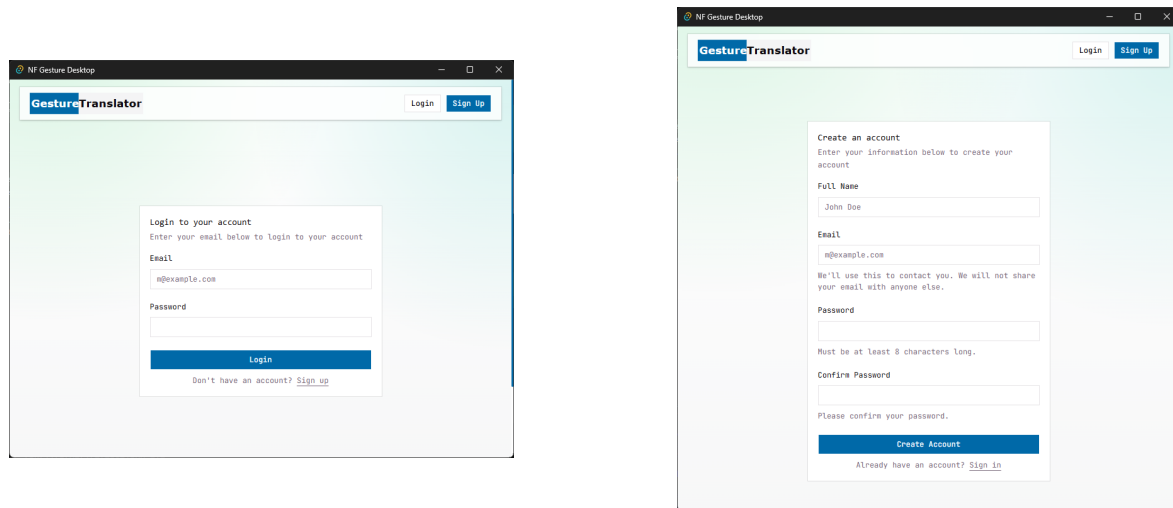


Figure 4.4: Authentication screen.

It is a simple login and registration screen. New users create an account with a username and password. Passwords are hashed with Argon2 before storage; no plaintext credentials are written to disk or the database. Returning users log in and are taken directly to the dashboard, with their session maintained via a persistent session token.

The authentication layer matters because the application is designed to be multi-user. Each account maintains its own gesture labels, training data, and trained models independently. One user's setup does not affect another's.

4.6 The Dashboard

After login, the user lands on the dashboard, which serves as the main navigation page of managing models.

At the top of the interface, three tabs are available: **Training**, **Labels**, and **Landmarks**. Each tab opens its corresponding page, allowing the user to navigate

between the core functionalities of the system. The dashboard provides a simple and organized starting point for accessing gesture label management, landmark collection, and model training features.

4.7 Gesture Labels

Before capturing any training data, a user needs to define what gestures they want the system to recognise. This is what the labels section is for.

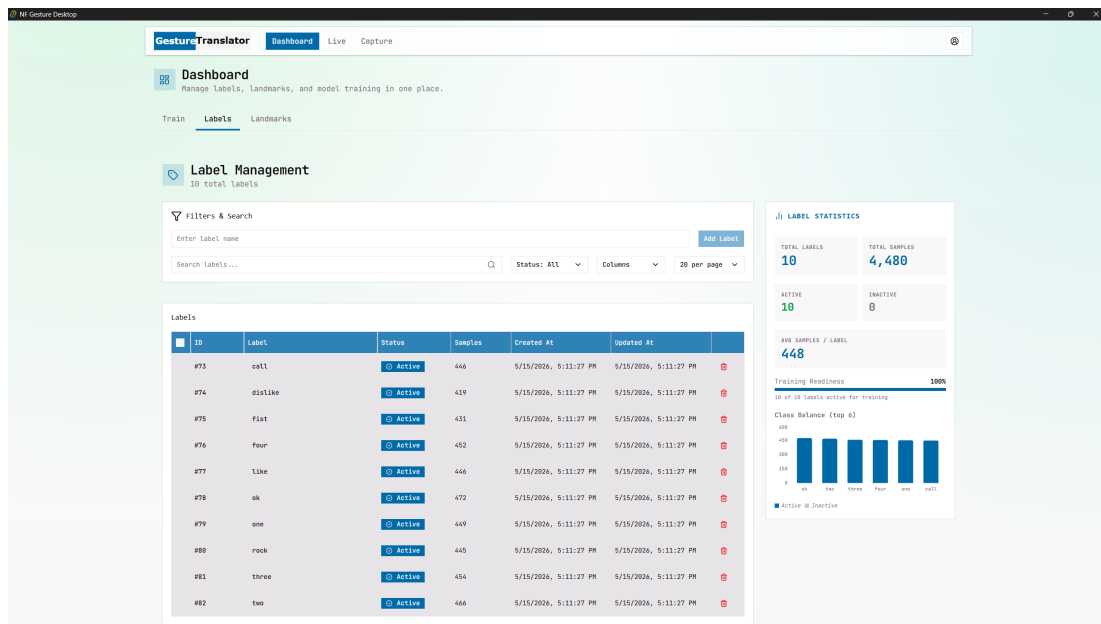


Figure 4.5: Gesture label management.

Each label is simply a name for a gesture class: *hello*, *stop*, *one*, or any label the user decides to create. Labels can be added, renamed, and deleted. They can be reordered. Deleting a label warns the user that all associated training data will be removed as well. Standard data management behaviour, but important to get right since the label set directly defines the structure of the training problem.

The label management section is intentionally simple. The only thing it needs to do is let the user maintain a clean, accurate list of the gesture categories they intend to train on.

4.8 Capturing Training Data

With labels defined, the next step is to provide training samples for each one.

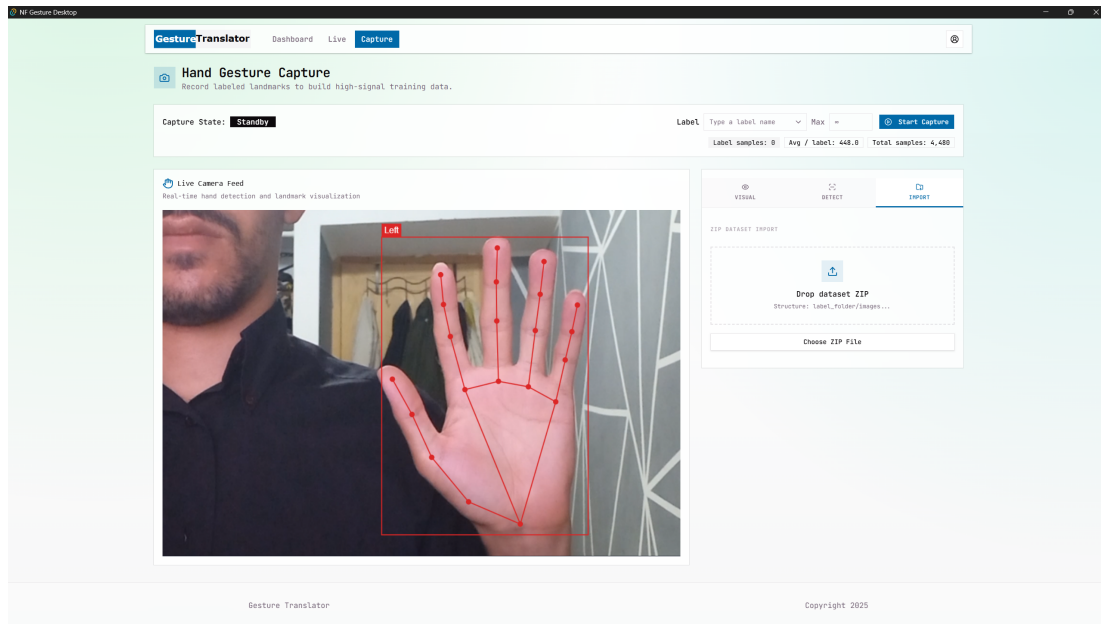


Figure 4.6: Training data capture.

There are two ways to get training data into the system. The first is live capture: the user selects a label from the list, shows the corresponding hand gesture to the camera, and captures frames directly from the live video feed, which runs at approximately 60 frames per second. For each detected hand, MediaPipe extracts the 21 keypoint positions and the frontend converts them into a normalised 63-dimensional coordinate vector. It is this landmark vector, not the raw camera image, that is stored in the database. This keeps the data compact and immediately ready for training without any further preprocessing step.

A progress indicator shows how many samples have been collected for the selected label. The application requires a minimum of **50 samples per label** before training becomes available, and it steers the user toward a balanced dataset: if the sample counts across labels are uneven, the interface highlights which labels still need more coverage before the training step is unlocked.

The second method is a drag-and-drop import. The user provides a zip archive whose internal folder structure defines the label mapping: each top-level folder inside the archive is treated as a gesture label, and the images inside that folder become the training samples for that gesture. If a folder name matches an existing label, its samples are added to that label. If a folder name does not match any existing label, the application creates the label automatically before importing the samples. This makes bulk import from an external image collection straightforward without requiring the user to manually pre-create every label.

Both methods feed into the same underlying data store. There is no functional difference in how the resulting training set is handled downstream.

4.9 Training a Model

When the user is satisfied with their dataset, they move to the training section.

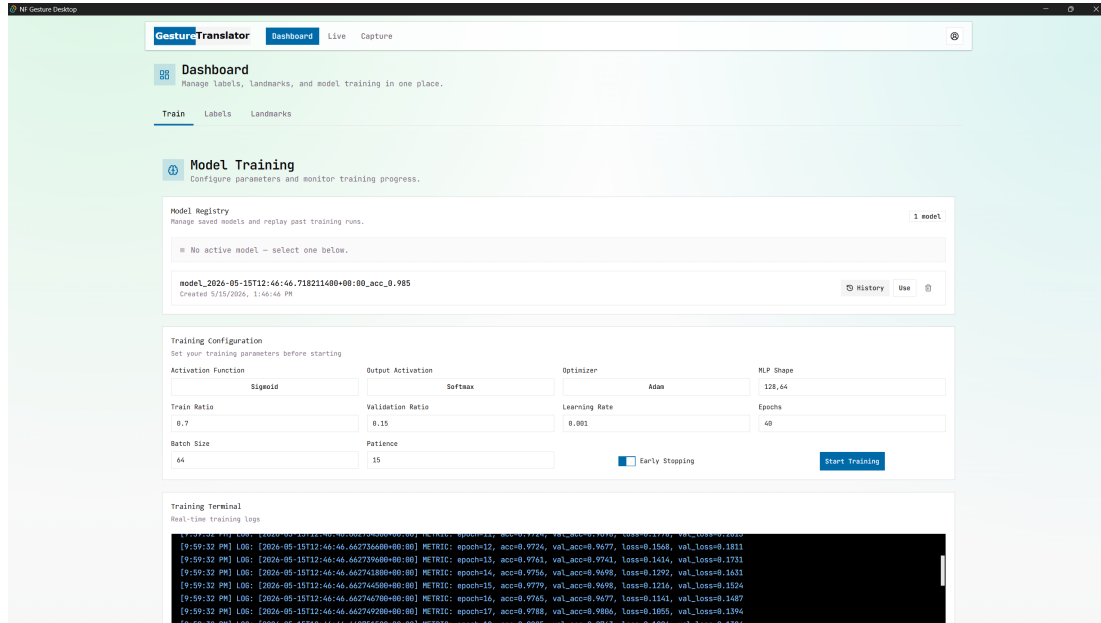


Figure 4.7: Training configuration panel.

4.9.1 Configuring the Network

The training view starts with a configuration panel. The user can set the number of hidden layers, how many nodes each layer contains, the optimiser, the learning rate, the batch size, and the number of training epochs. These are the same hyperparameters used in the experiments described in Chapter 3, exposed here in a form that a non-expert user can adjust without touching any code. The available optimisers are Adam and SGD.

Default values are pre-filled: sigmoid activation, softmax output, Adam optimiser, learning rate 0.001, batch size 32, and 200 epochs. A new user can simply click train without changing anything and get a working model. More experienced users can adjust the settings and experiment.

4.9.2 The Training Log

Once training starts, the interface transitions to a live log display.

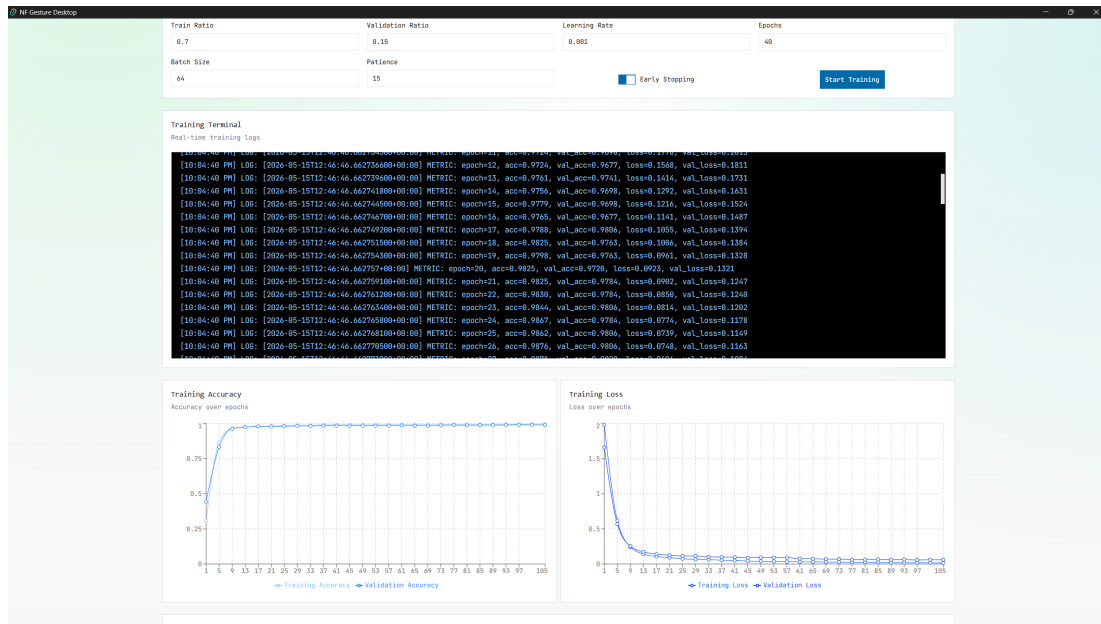


Figure 4.8: Live training log.

Training is executed entirely within the Rust backend as a custom MLP implementation with no external machine learning library. The backend spawns a dedicated training thread that emits live progress events to the frontend after each epoch, reporting the current epoch number, training loss, validation accuracy, and elapsed time. The frontend renders these as a terminal-style log that updates in real time, with a progress bar showing how far through the configured epoch limit the run currently is.

Early stopping is implemented in the training loop with a patience of 15 epochs and a minimum improvement threshold of 10^{-4} . When validation accuracy has not improved by at least that amount for 15 consecutive epochs, training halts automatically, the best checkpoint is restored, and the log reports that early stopping has triggered. The user does not need to monitor the run; training ends cleanly on its own when convergence is detected.

This kind of live feedback is deliberately included for two reasons. First, it gives the user confidence that something is actually happening. A blank screen during a training run that takes a few minutes is uncomfortable. Second, it lets the user catch problems early: if the loss is not decreasing or the accuracy stays flat from the first few epochs, there is probably a data problem, and it is better to stop and fix it than to wait for the full run to complete.

4.9.3 Training Results and Reports

When training finishes, the interface presents the results.

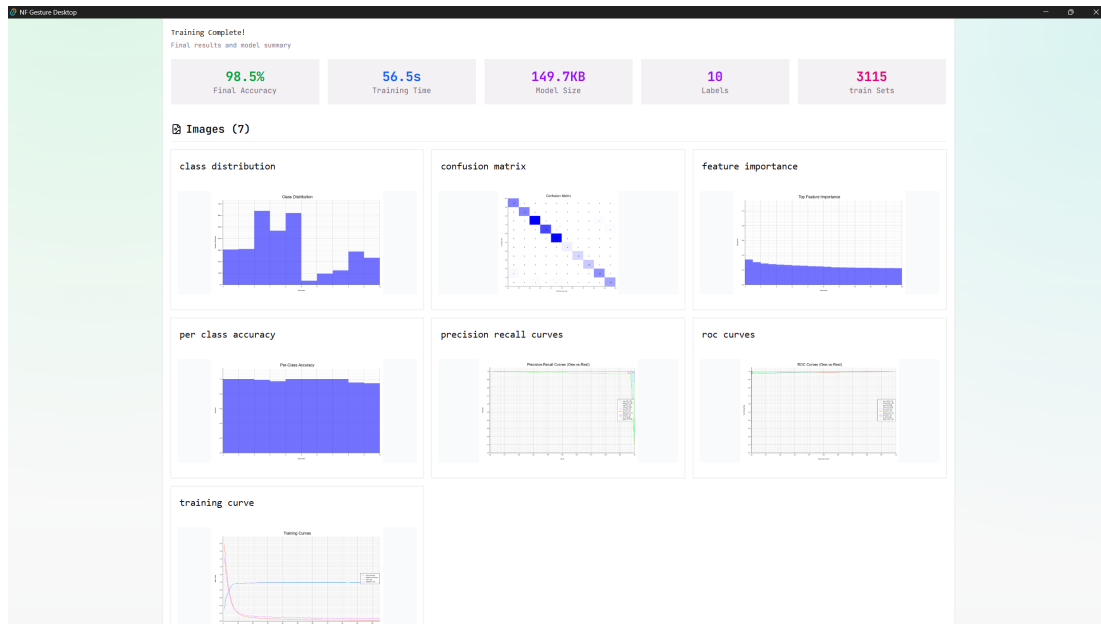


Figure 4.9: Training results report.

The report panel shows the full set of diagnostic outputs: accuracy and loss curves across epochs, precision and recall broken down by gesture class, a confusion matrix, and summary statistics. These are the same reporting elements used throughout Chapter 3. The user can look at which classes performed well, which ones confused each other, and make a judgment about whether the model is good enough to deploy or whether more data is needed for specific gestures.

4.9.4 The 3D Neural Network Visualisation

The most unusual element of the training section is the interactive three-dimensional visualisation of the trained network.

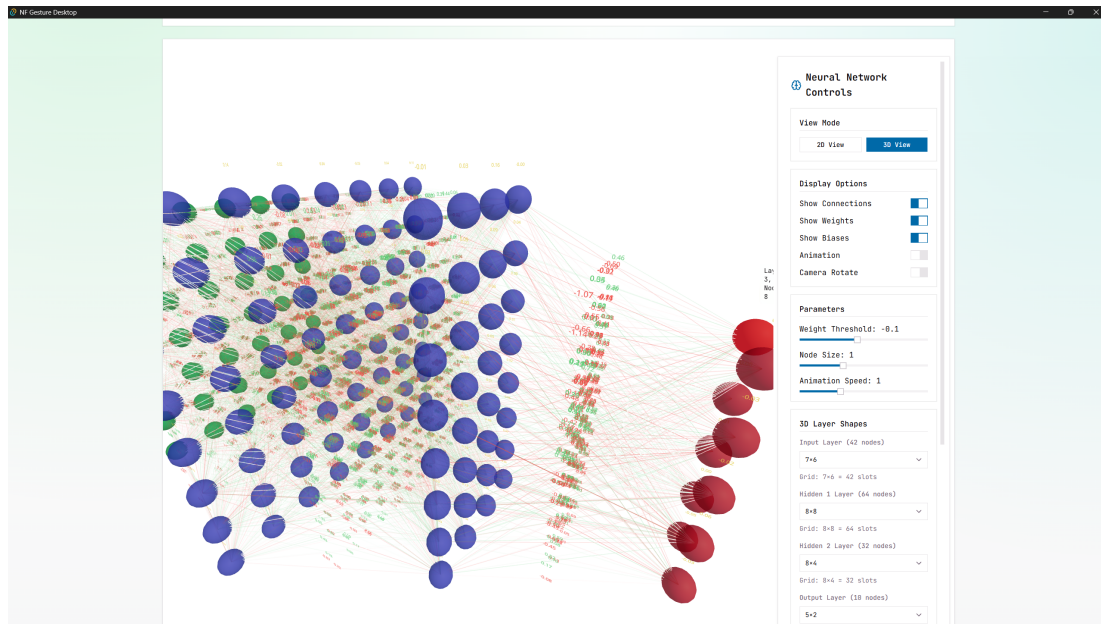


Figure 4.10: 3D neural network visualisation.

Once training completes, the network's weights are passed to a Three.js scene that renders the entire model as a three-dimensional graph. Each node in the graph represents a neuron in a layer. The connections between nodes are drawn as edges coloured according to the learned weight: positive connections are drawn in green and negative ones in red, with opacity proportional to the weight's magnitude so that near-zero connections fade into the background.

The user can orbit the scene with the mouse and zoom in and out. Clicking on a node opens a details panel showing the node's layer, index, type, and bias value, along with statistics for its incoming and outgoing connections. This makes it possible to inspect individual neurons and get a concrete sense of which connections in the network carry the most weight.

The visualisation is not a rigorous debugging tool in the analytical sense. The internal representations learned by an MLP are not easily interpretable just from looking at the weights. But it makes the model tangible. It gives the user something concrete to look at and interact with, and it communicates clearly that the trained model is a real structured object with internal organisation, not just a black box that produces outputs.

4.10 Translation: Using the Model

The final section of the application is where everything comes together. The translation page is where the trained model is put to actual use.

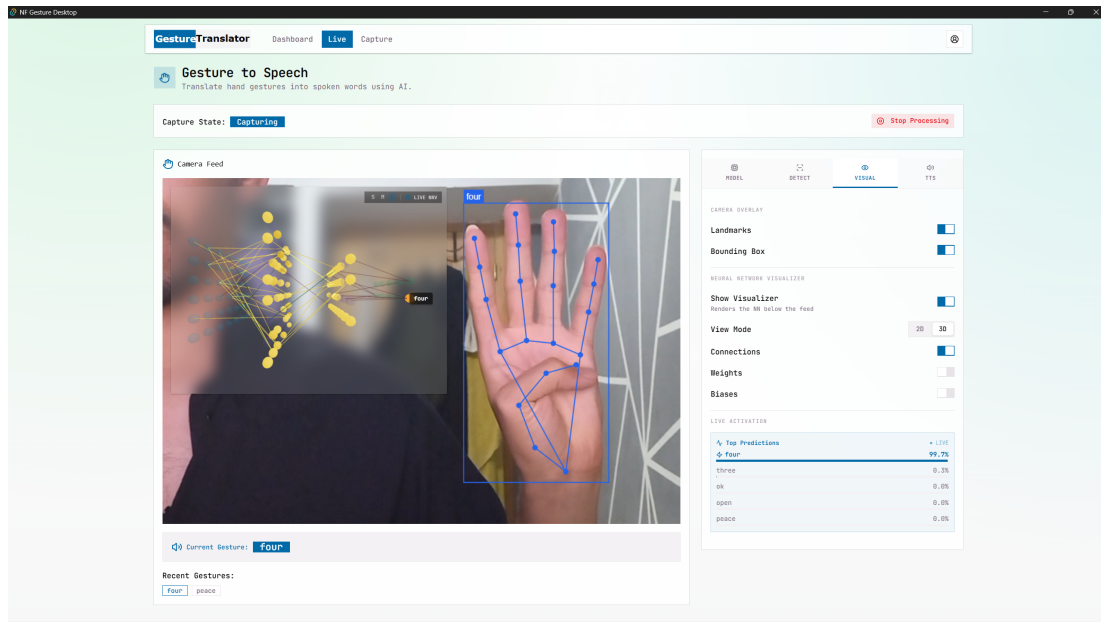


Figure 4.11: Translation interface.

The user first selects which trained model to use from a dropdown list. All previously trained models for the current account are available. Then they activate the camera. From that point, the pipeline runs continuously: each frame is processed by MediaPipe, the 63-dimensional landmark vector is extracted, the Rust backend classifies it, and the predicted gesture label is returned to the interface.

When the recognised label changes, the application immediately speaks it aloud using the text-to-speech engine and appends it to a sentence buffer displayed on screen. This buffer accumulates gesture-by-gesture, allowing the user to compose a sequence of words or phrases by performing one gesture at a time. They can clear the buffer, delete the last entry, or pause recognition at any point.

The result is a complete communication loop. Show a gesture, hear a word, see it added to the sentence. Show another, hear another. The system does not understand grammar or context. It recognises one gesture at a time and speaks it, but the sequence displayed on screen gives a visual record of what has been said, and the spoken output makes the interaction feel considerably more natural than staring at a label on a screen.

Table 4.2 summarises the full application flow from account creation to active use.

Table 4.2: End-to-end application flow.

#	Step	What happens
1	Register or log in	New users create an account; returning users authenticate and reach their dashboard
2	Review the dashboard	Overview of existing labels, captured samples, and available trained models; navigation to all sections via sidebar
3	Define gesture labels	Create the list of gesture categories the system should learn; add, rename, reorder, or delete labels as needed
4	Capture or import training data	Record samples live via the camera for each label (minimum 50 per label required), or import a structured zip archive; unknown folder names in the archive are created as new labels automatically
5	Review and curate the dataset	Browse captured samples by label, remove bad entries, and verify class coverage before training
6	Configure and start training	Set the network architecture (layers, nodes) and training parameters (optimiser, learning rate, batch size, epochs); start the training run
7	Monitor the training log	Watch epoch-by-epoch progress in the live terminal display; training stops automatically when early stopping triggers or the epoch limit is reached
8	Review training results	Inspect accuracy and loss curves, per-class metrics, confusion matrix, and the 3D neural network visualisation
9	Translate	Select a trained model, activate the camera, and use hand gestures to produce spoken words and build sentences on screen

4.11 Development Environment

The entire application was developed and tested on a single mid-range workstation with the following specifications:

- **Processor:** AMD Ryzen 5 5600G
- **Memory:** 32 GB RAM
- **Camera:** Full HD (1080p) webcam
- **Operating system:** Windows 11
- **Editor:** Visual Studio Code

This is an intentionally ordinary setup. No GPU. No specialised machine learning hardware. The fact that the full pipeline, including landmark extraction and real-time classification, runs comfortably on this machine is not accidental. It is a consequence of the model selection made in Chapter 3. The MLP classifier adds under a millisecond per frame. MediaPipe runs efficiently on CPU. The Rust backend has no garbage collector to pause at inopportune moments. The application was designed to be fast enough on hardware that a typical user is likely to actually own.

4.12 Conclusion

The development of this application did not follow a straight line. Three distinct technology stacks were tried before arriving at the final one. Each switch was forced by a real limitation: Tkinter was too crude for a usable interface, FastAPI's real-time streaming was unstable, Next.js with FastAPI solved the interface problem but not the distribution and socket problems. Moving to a desktop application with Tauri resolved all of these at once, and the resulting stack (Rust backend, React frontend, Tauri shell) turned out to be a good fit for what the application actually needs to do.

The final application is structured around a natural user progression: set up labels, collect data, train a model, use it. Each section of the application corresponds to one step in that flow. The training section goes beyond a simple train button by exposing the configuration options, showing live progress, presenting diagnostic reports, and rendering an interactive view of the trained network. The translation section closes the loop by turning the trained model into something the user can actively communicate with.

Architecturally, the choices made here are not independent of the research results in Chapter 3. The classifier is compact enough that the application does not need a heavy inference server or a GPU. The landmark-based input representation means that only a small numerical vector crosses the boundary between the frontend and the backend, keeping the communication path lightweight. The application design is, in that sense, a direct consequence of the model selection, not a separate decision made afterward.

General Conclusion

General Conclusion

This thesis examined Hand Gesture Recognition as a real-time computer vision problem and argued that a landmark-based neural pipeline can simultaneously satisfy the predictive and operational requirements of a practical interactive system. To substantiate that argument, the work combined a conceptual review of the field, a systematic comparative evaluation of eleven model families, and a concrete translation of the selected solution into a deployable application.

The domain review established that HGR sits at the intersection of human communication, computer vision, and interactive systems design, and that its difficulty comes from a combination of visual variability, inter-user diversity, and strict latency constraints. Three main approaches were surveyed: glove-based systems, full-image models, and landmark-based pipelines. Each offers a fundamentally different trade-off between representational richness, hardware dependency, and computational cost. The central conclusion was that landmark-based methods occupy a particularly favorable position for real-time deployment. They preserve the geometric structure of the hand while discarding most of the irrelevant visual context that burdens full-image models.

The central bet of the experimental work was that this representational advantage would translate into measurable results. It did. A two-layer neural network operating on 21 hand joint coordinates outperformed a Vision Transformer, a convolutional network, and every other configuration in the benchmark, on accuracy, on speed, and on model size, simultaneously. **MediaPipe + MLP** reached **95.10% accuracy** and **95.14% Macro-F1** at the full training split, with a classifier latency of **0.021 ms per sample** and only **17,618 parameters**. The selected pipeline was then packaged into a responsive desktop application built with Tauri, a React frontend, and a Rust backend, demonstrating that a system this compact can be deployed without a GPU, without a dedicated server, and without the infrastructure overhead that heavier pipelines typically demand.

What stays open is more interesting than what was closed. The confusion matrix points directly at the next problem: a handful of gesture pairs share enough geometric structure that the landmark vector cannot always separate them. Targeted augmentation for those specific pairs is the obvious next step. Beyond that, the pipeline is purely frame-level. It recognises one static pose at a time. Extending it to dynamic gestures, defined by movement and trajectory

rather than shape alone, would require temporal modelling and is an open direction worth pursuing. Cross-user robustness is another. The nine-point accuracy gap between signer-dependent and signer-independent evaluation, documented consistently across the broader literature, was not addressed here. It remains the most practically consequential unsolved problem in the field.

The system built here is a starting point. A working one, but still a starting point. The recognition vocabulary is fixed at training time, the sentences it produces have no grammar, and the user interface is a prototype rather than a finished product. All of those are solvable. What this work establishes is that the foundation is sound: the representation is efficient, the classifier is fast, the architecture is deployable, and the pieces fit together into something a real user can sit in front of and use. Everything else builds on that.

References

- [1] Vladimir I. Pavlovic, Rajeev Sharma, and Thomas S. Huang. "Visual Interpretation of Hand Gestures for Human-Computer Interaction: A Review." In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 19.7 (1997), pp. 677–695.
- [2] Sushmita Mitra and Tinku Acharya. "Gesture Recognition: A Survey." In: *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)* 37.3 (2007), pp. 311–324.
- [3] Siddharth S. Rautaray and Anupam Agrawal. "Vision Based Hand Gesture Recognition for Human Computer Interaction: A Survey." In: *Artificial Intelligence Review* 43.1 (2015), pp. 1–54.
- [4] Juan P. Wachs, Helman Stern, and Yael Edan. "Vision-Based Hand-Gesture Applications." In: *Communications of the ACM* 54.2 (2011), pp. 60–71.
- [5] William T. Freeman and Michal Roth. "Orientation Histograms for Hand Gesture Recognition." In: *International Workshop on Automatic Face and Gesture Recognition*. 1995.
- [6] Pavlo Molchanov et al. "Hand Gesture Recognition with 3D Convolutional Neural Networks." In: *IEEE Conference on Computer Vision and Pattern Recognition Workshops*. 2015.
- [7] Tomas Simon et al. "Hand Keypoint Detection in Single Images Using Multiview Bootstrapping." In: *IEEE Conference on Computer Vision and Pattern Recognition*. 2017.
- [8] Christian Zimmermann and Thomas Brox. "Learning to Estimate 3D Hand Pose from Single RGB Images." In: *IEEE International Conference on Computer Vision*. 2017.
- [9] Yann LeCun et al. "Gradient-Based Learning Applied to Document Recognition." In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324.
- [10] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. "ImageNet Classification with Deep Convolutional Neural Networks." In: *Advances in Neural Information Processing Systems*. 2012.

-
- [11] Kaiming He et al. "Deep Residual Learning for Image Recognition." In: *IEEE Conference on Computer Vision and Pattern Recognition*. 2016.
 - [12] Ashish Vaswani, Noam Shazeer, Niki Parmar, et al. "Attention Is All You Need." In: *Advances in Neural Information Processing Systems*. 2017.
 - [13] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, et al. "An Image Is Worth 16x16 Words: Transformers for Image Recognition at Scale." In: *International Conference on Learning Representations*. 2021.
 - [14] Fan Zhang, Valentin Bazarevsky, Andrey Vakunov, et al. "MediaPipe Hands: On-Device Real-Time Hand Tracking." In: *arXiv preprint arXiv:2006.10214* (2020).
 - [15] Andrew G. Howard, Menglong Zhu, Bo Chen, et al. "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications." In: *arXiv preprint arXiv:1704.04861* (2017).
 - [16] Mark Sandler et al. "MobileNetV2: Inverted Residuals and Linear Bottlenecks." In: *IEEE Conference on Computer Vision and Pattern Recognition*. 2018.
 - [17] Mingxing Tan and Quoc V. Le. "EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks." In: *International Conference on Machine Learning*. 2019.
 - [18] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.
 - [19] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.
 - [20] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. "Learning Representations by Back-Propagating Errors." In: *Nature* 323 (1986), pp. 533–536.
 - [21] Olivier Chapelle, Bernhard Schölkopf, and Alexander Zien. *Semi-Supervised Learning*. MIT Press, 2006.
 - [22] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. 2nd ed. MIT Press, 2018.
 - [23] Ikram Hashi et al. "A Systematic Review of Hand Gesture Recognition: An Update from 2018 to 2024." In: *Archives of Computational Methods in Engineering* (2024). DOI: [10.1007/s11831-024-10110-0](https://doi.org/10.1007/s11831-024-10110-0).
 - [24] Kenneth Lai and Svetlana N. Yanushkevich. "CNN+RNN Depth and Skeleton based Dynamic Hand Gesture Recognition." In: *2018 24th International Conference on Pattern Recognition (ICPR)*. 2018, pp. 3451–3456. DOI: [10.1109/ICPR.2018.8545718](https://doi.org/10.1109/ICPR.2018.8545718).
 - [25] M. A. Amin et al. "A Comparative Review on Sign Language Recognition Using Wearable and Vision-Based Sensors." In: *IEEE Access* 10 (2022), pp. 102161–102183. DOI: [10.1109/ACCESS.2022.3207726](https://doi.org/10.1109/ACCESS.2022.3207726).

-
- [26] Corinna Cortes and Vladimir Vapnik. "Support-Vector Networks." In: *Machine Learning* 20.3 (1995), pp. 273–297.
 - [27] Thomas M. Cover and Peter E. Hart. "Nearest Neighbor Pattern Classification." In: *IEEE Transactions on Information Theory* 13.1 (1967), pp. 21–27.
 - [28] Leo Breiman. "Random Forests." In: *Machine Learning* 45.1 (2001), pp. 5–32.
 - [29] Geoffrey E. Hinton and Ruslan R. Salakhutdinov. "Reducing the Dimensionality of Data with Neural Networks." In: *Science* 313.5786 (2006), pp. 504–507.
 - [30] Kihyuk Sohn, David Berthelot, Chun-Liang Li, et al. "FixMatch: Simplifying Semi-Supervised Learning with Consistency and Confidence." In: *Advances in Neural Information Processing Systems*. 2020.
 - [31] Karen Simonyan and Andrew Zisserman. "Very Deep Convolutional Networks for Large-Scale Image Recognition." In: *International Conference on Learning Representations*. 2015.
 - [32] Franco Scarselli et al. "The Graph Neural Network Model." In: *IEEE Transactions on Neural Networks* 20.1 (2009), pp. 61–80.
 - [33] Thomas N. Kipf and Max Welling. "Semi-Supervised Classification with Graph Convolutional Networks." In: *International Conference on Learning Representations*. 2017.
 - [34] Vinod Nair and Geoffrey E. Hinton. "Rectified Linear Units Improve Restricted Boltzmann Machines." In: *International Conference on Machine Learning*. 2010.
 - [35] Matthias Hoppe et al. "Mouse-Based Hand Gesture Interaction in Virtual Reality." In: *Electronics* 12.16 (2023), p. 3404. DOI: [10.3390/electronics12163404](https://doi.org/10.3390/electronics12163404).
 - [36] David M. W. Powers. "Evaluation: From Precision, Recall and F-Measure to ROC, Informedness, Markedness and Correlation." In: *Journal of Machine Learning Technologies* 2.1 (2011), pp. 37–63.
 - [37] Marina Sokolova and Guy Lapalme. "A Systematic Analysis of Performance Measures for Classification Tasks." In: *Information Processing & Management* 45.4 (2009), pp. 427–437.
 - [38] Diederik P. Kingma and Max Welling. "Auto-Encoding Variational Bayes." In: *International Conference on Learning Representations*. 2014.

Figure Sources

- [39] *Figure source: Glove-based hand gesture recognition image.* URL: <https://www.sciencedirect.com/science/article/abs/pii/S221478532202956X> (visited on 02/07/2026).
- [40] *Figure source: Landmark-based hand gesture recognition image.* URL: https://www.researchgate.net/figure/Description-of-6-hand-gestures-assigned-to-6-wheelchair-activities-a-Forward-1-b_fig3_388971598 (visited on 03/11/2026).
- [41] *Figure source: Supervised, unsupervised, and reinforcement learning paradigms.* URL: <https://www.sciencedirect.com/science/article/pii/S0955799725000207> (visited on 03/11/2026).
- [42] *Figure source: Convolutional neural network processing diagram.* URL: <https://www.semanticscholar.org/paper/Disease-Detection-in-Bombyx-Mori-Silkworm-Using-CNN-Singla-Garg/7094c2f0f18ea9f3cb67a1af0109e2b6c864fd75> (visited on 03/14/2026).

Appendix A: List of Acronyms

Appendix A

Appendix A: List of Acronyms

- **AI:** Artificial Intelligence
- **API:** Application Programming Interface
- **AR/VR:** Augmented Reality / Virtual Reality
- **CNN:** Convolutional Neural Network
- **CSV:** Comma-Separated Values
- **GNN:** Graph Neural Network
- **HCI:** Human-Computer Interaction
- **HGR:** Hand Gesture Recognition
- **KNN:** K-Nearest Neighbours
- **MLP:** Multi-Layer Perceptron
- **ML:** Machine Learning
- **RF:** Random Forest
- **ViT:** Vision Transformer