ASL Recognition System:
Bridging Gaps in Communication Accessibility

A Thesis Submitted in Partial Fulfillment of the Requirements of the
Renée Crown University Honors Program at Syracuse University

Carlo F. Pisacane

Candidate for Bachelor's of Science in Computer Science
and Renée Crown University Honors
May 2025

Honors Thesis Faculty Advisor: Dr. Nadeem Ghani

Honors Thesis Reader: Dr. João Paulo Marum

Abstract

This thesis addresses a gap in communication accessibility for Deaf and hard-of-hearing people by developing a real-time American Sign Language (ASL) recognition system that recognizes and classifies static ASL signs with high accuracy. Current solutions often suffer from limited vocabularies, reliance on specialized hardware, or fail to generalize to real-world scenarios, but our solution leverages accessible computer vision pipelines and machine learning models to address these issues.

Specifically, we employ MediaPipe Hands to extract landmarks efficiently and compare four classification models: a Multilayer Perceptron (MLP), a Graph Convolutional Network (GCN), a Convolutional Neural Network (CNN), and a combined CNN–GCN architecture. Our results validate that fusing both spatial (CNN) and relational (GCN) insights yields the most accurate result, surpassing other standalone approaches. The system provides real-time prediction, smooth visualization, and minimal calibration needs.

With its focus on accessibility and user-centered design, this work contributes a scalable and robust framework for ASL recognition. Beyond static letters, the system has great potential for expansion into dynamic gestures and the integration of nonmanual markers like facial expressions. Ultimately, the presented ASL recognition tool illustrates how AI techniques can enable inclusive communication, fostering broader societal and educational benefits.

Executive Summary

This thesis presents the design, development, and evaluation of a real-time American Sign Language (ASL) recognition system aimed at improving communication accessibility for Deaf and hard-of-hearing individuals. Motivated by gaps in current ASL technologies, such as limited vocabulary, reliance on specialized hardware, and poor generalization, this work leverages accessible, camera-based computer vision and machine learning techniques to build a robust and user-friendly solution.

The system employs MediaPipe Hands and OpenCV for landmark extraction and image processing, producing 63-dimensional vectors representing key hand joint positions. These vectors are fed into and compared across four machine learning models:

- Multilayer Perceptron (MLP)
- Graph Convolutional Network (GCN)
- Convolutional Neural Network (CNN)
- Hybrid CNN and GCN model

Comprehensive evaluation using the ASL Alphabet dataset (approximately 124,000 images) demonstrated the hybrid CNN and GCN model to be the most effective, achieving 99.01% validation accuracy. The combination of CNN's spatial feature extraction and GCN's structural understanding of hand joint relationships outperformed all other architectures. Key innovations include a modular preprocessing pipeline, real-time gesture stability logic, and a minimal-calibration interface optimized for accessibility.

While this thesis focuses on static sign recognition (letters A to Z plus "space", "del", and "nothing"), the architecture is extensible to dynamic gestures and nonmanual markers like facial expressions. Limitations include the absence of sequence modeling and the system's current focus on isolated signs rather than continuous ASL.

This project highlights the power of combining ML techniques with inclusive design principles, laying the groundwork for future work in real-world ASL interpretation tools, interactive language learning systems, and accessible human-computer interaction platforms.

Contents

Preface

This thesis represents the culmination of my academic, personal, and technical journey at Syracuse University. It is the result of countless hours exploring the intersection of computer vision, machine learning, and human-centered technology, and it reflects my dedication to making communication more accessible for Deaf and hard-of-hearing individuals.

As someone passionate about technology and accessibility, I've long been fascinated by the potential of technology to bridge gaps between people. Taking an American Sign Language (ASL) class deepened my appreciation for the language's richness and helped conceptualize my idea. This project became an opportunity not just to apply technical skills, but to contribute meaningfully to creating a working ASL recognition system.

Throughout this experience, I've strengthened my abilities in machine learning, image processing, and real-time system development. From building neural network models to designing preprocessing pipelines, each stage of the process has challenged me to grow as a computer scientist and researcher.

More importantly, this project has taught me the value of inclusive design. Informed by input from ASL users and accessibility research, I've learned to think critically about how systems can better serve those often left out of mainstream technological advancements. This project is more than a technical solution. It is a reflection of my belief that innovation should empower everyone.

Completing this thesis has cemented my commitment to using my skills to create technologies that are impactful. I am proud of what this work represents and hopeful about the possibilities it opens for future research and as I begin my career in software development and computer science.

Acknowledgments

I am deeply grateful for the support and guidance I have received throughout my academic journey at Syracuse University. I extend my sincere thanks to my advisor, Prof. Nadeem Ghani, whose mentorship was instrumental in shaping my research and academic pursuits. I also thank Prof. João Marum, for his valuable insights and our collaboration in VR research.

I am particularly thankful to Robin Smith, my Honors advisor, for encouraging and guiding me during my time in the Honors program and throughout the thesis development process. Their support helped me overcome challenges and remain focused on my goals.

Finally, I am forever indebted to my friends and family, whose encouragement and belief in me have been a constant source of inspiration.

Chapter 1

Introduction

Communication is a fundamental human need, and for Deaf and hard-of-hearing individuals, American Sign Language (ASL) serves as a primary mode of expression. ASL is a rich and complex visual language based on hand shape, movement, and nonmanual markers such as facial expressions. Studies emphasizing Deaf-centric design have shown that effective ASL tools must respect the cultural and linguistic practices of the Deaf community (Hibbard et al., 2020 [1]).

Technological advancements have improved accessiblility through captioning services, text-based messaging, and video relay services. However, these solutions often rely on real-time interpreters or a shared written language, which can be limiting in spontaneous, in-person conversations.

In recent years, advances in computer vision and machine learning have opened new avenues for automated sign language recognition. By leveraging the power of advanced algorithms and increasingly pervasive hardware such as mobile phone cameras and webcams, engineers and researchers aim to create machines capable of recognizing ASL hand gestures in real time. While some progress has been made in recognizing static signs or alphabets (Debnath and Joe, 2024 [3]), challenges remain, particularly with vocabulary expansion, nonmanual features, dynamic signs, and real-world performance (Falvo et al., 2020 [2]). Although these are crucial frontiers for the field, this thesis focuses instead on evaluating and comparing recognition methods for static signs using various model architectures.

This work presents the development of a real-time ASL recognition system using MediaPipe and OpenCV for landmark detection, along with multiple machine learning models including MLP, CNN, GCN, and a hybrid model. The system demonstrates potential for real-world accessibility applications through its real-time performance and robust model architecture.

1.1 Research Problem

Current ASL recognition systems have several limitations that make them impractical. Some systems recognize only a limited set of signs, which restricts real-world applicability. Others do not account for nonmanual signs, such as facial expressions or head tilts, which are essential in ASL for the expression of tone, grammatical markers, and emotional context. Additionally, some tools demand specialized sensors that are expensive or inconvenient, thus deterring widespread use.

There is also a broad gap in designing user interfaces that are responsive to the needs and desires of Deaf individuals. Inaccurate calibration procedures, variability of performance under changing lighting conditions, or excessive latency can lower the usability of a system. These barriers compound to inhibit the real-world deployment potential of ASL recognition technology in everyday communication (Falvo et al., 2020 [2]).

1.2 Research Objectives

The primary objective of this thesis is to design a real-time ASL recognition system using computer vision and machine learning techniques that achieves high recognition accuracy. Secondary goals include:

- Ensuring system robustness across diverse lighting and background conditions commonly encountered in real-world environments.
- Building an extensible framework that can later support additional static signs, dynamic gestures, and nonmanual features.

By focusing on these objectives, the system aims to lay a foundation for broader applications in education, assistive technology, and inclusive communication.

1.3 Significance of the Study

This project holds the potential to significantly reduce communication barriers for Deaf and hard-of-hearing individuals while also providing valuable tools for hearing individuals who wish to learn ASL. An effective real-time ASL recognition system can facilitate communication more effectively in public places, schools, and workplaces, particularly where access to interpreters may not be readily available. It can also serve as an interactive learning tool for ASL learners, offering instant feedback on handshapes to facilitate language learning. The proposed framework can further be extended to encompass the full richness of sign languages, thereby enabling future research studies. By prioritizing usability and involving Deaf/ASL communities in the design, this research underscores the importance of user-centered solutions.

Chapter 2

Literature Review

This literature review surveys various methods and technologies employed in American Sign Language (ASL) recognition systems. The goal is to establish how our approach, aimed at real-time recognition and user-friendly interaction fits into the existing body of work. By examining both hardware-based and vision-based solutions, we highlight the fundamental challenges and opportunities in creating accessible tools for Deaf and hard-of-hearing communities.

2.1 Overview of Existing ASL Recognition Systems

Recent developments in sign language recognition often revolve around three main technological streams:

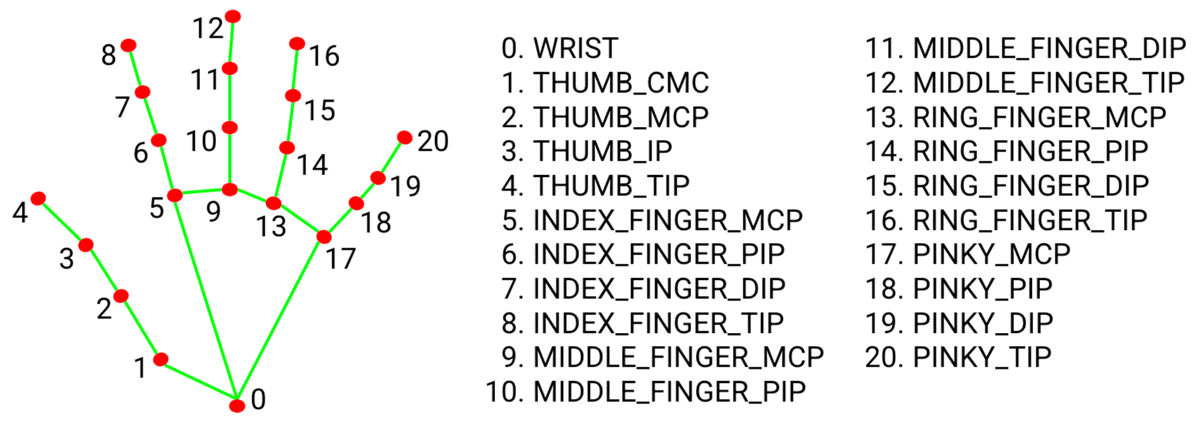
- Computer Vision Approaches
- Depth Sensor Approaches
- Wearable Technology Approaches

Each of these streams has strengths and limitations related to cost, accuracy, ease of deployment, and user comfort. This section reviews notable research in these areas and provides the groundwork for our own system's design choices.

2.1.1 Computer Vision Approaches

A substantial portion of ASL recognition research leverages standard RGB cameras and a variety of computer vision techniques. Traditional pipelines often involve skin detection, feature extraction, and hand segmentation before proceeding to classification. More recently, robust libraries like OpenCV and frameworks such as MediaPipe have significantly simplified and improved hand detection and tracking.

One notable example is the *MediaPipe Hands* solution, an open-source tool by Google[1]. MediaPipe Hands tracks 21 key landmarks on each hand, as illustrated in Figure 1. This framework provides real-time tracking even under challenging conditions such as varied lighting or partial occlusions, which is crucial for the fluid and rapid gestures of sign language.



0. WRIST
1. THUMB_CMC
2. THUMB_MCP
3. THUMB_IP
4. THUMB_TIP
5. INDEX_FINGER_MCP
6. INDEX_FINGER_PIP
7. INDEX_FINGER_DIP
8. INDEX_FINGER_TIP
9. MIDDLE_FINGER_MCP
10. MIDDLE_FINGER_PIP
11. MIDDLE_FINGER_DIP
12. MIDDLE_FINGER_TIP
13. RING_FINGER_MCP
14. RING_FINGER_PIP
15. RING_FINGER_DIP
16. RING_FINGER_TIP
17. PINKY_MCP
18. PINKY_PIP
19. PINKY_DIP
20. PINKY_TIP

Figure 1: MediaPipe Hands landmarks. Each point corresponds to a specific joint or fingertip.

Researchers have successfully integrated MediaPipe's tracking features into sign language systems to reduce the complexity of manual feature engineering. For instance, Debnath and Joe [3] demonstrate a real-time ASL detection setup where MediaPipe is employed to extract hand pose information, thereby reliably differentiating between common ASL hand shapes and movements.

2.1.2 Depth Sensor Approaches

Beyond standard RGB cameras, depth sensors capture three-dimensional information about hand positions. Notably, Avola et al. [4] illustrate how the Leap Motion Controller can track the skeletal structure of the hand with fine-grained accuracy, enabling precise measurement of joint angles. This capability is beneficial for distinguishing similar signs that differ only

[1] `https://github.com/google-ai-edge/mediapipe/blob/master/docs/solutions/hands.md`

slightly in finger placement or orientation.

2.1.3 Wearable Technology Approaches

Finally, some research efforts rely on wearable devices, such as glove-based sensors, to record hand movements. These systems, often equipped with inertial measurement units (IMUs) or bend sensors, can capture motion data in three dimensions. For example, Stefanidis et al. [5] discuss the use of 3D technologies in sign language applications through wearable platforms. While potentially more accurate, such devices can be costly or cumbersome, limiting their broad adoption for everyday use.

2.1.4 Summary

In summary, contemporary ASL recognition solutions range from purely vision-based approaches, such as MediaPipe Hands combined with OpenCV for live detection, to more specialized hardware solutions that utilize depth sensors and wearable gloves. Among these, live vision-based detection using MediaPipe and OpenCV offers the optimal balance of high accuracy, cost-effectiveness, and ease of integration, making it the best choice for designing robust, real-time ASL recognition systems that can be easily adopted in real-world contexts.

2.2 Machine Learning Approaches for Gesture Recognition

Advances in machine learning have been essential in enhancing sign language recognition by addressing both spatial and temporal complexities. In our work and as demonstrated in previous studies (Papastratis et al., 2021 [9] and Ru and Sebastian, 2023 [10]), a combination of Convolutional Neural Networks (CNNs), Recurrent Neural Networks (RNNs), and Long Short-Term Memory (LSTM) networks has been used to capture static hand features as well as the dynamic information inherent in continuous gestures. More recently, Graph Convolutional Networks (GCNs) have been introduced as powerful alternatives, especially for modeling spaital relationships between hand landmarks in non-Euclidean space (Sarkar et al., 2024[13], Bronstein et al. 2021[15]).

2.2.1 Convolutional Neural Networks (CNNs) for Static Sign Recognition

CNNs excel at extracting spatial features from images or individual video frames. Their layered architecture enables the learning of hierarchical representations that can effectively distinguish subtle differences in hand shape and orientation. In the context of ASL recognition, CNNs have been successfully used to classify isolated, static signs before further processing is applied (Debnath and Joe, 2024 [3]).

2.2.2 Recurrent Neural Networks (RNNs) and Long Short-Term Memory (LSTMs) for Dynamic Sign Recognition

Dynamic signs which involve motion and transitions require models that can capture temporal dependencies. Traditional RNNs are naturally suited for sequential data, but they often encounter difficulties when processing long input sequences (Avola et al., 2019 [4]; Liu et al., 2016 [6]). LSTM networks, introduced to overcome these limitations, incorporate memory cells and gating mechanisms that enable them to retain and update information over extended periods (Hochreiter and Schmidhuber, 1997 [7]; Colah's Blog, 2015 [8]). These capabilities make LSTMs particularly effective for recognizing continuous ASL gestures.

Hybrid architectures that combine CNNs and LSTMs have also proven effective. In such systems, CNNs first extract deep features from each frame, and these features are then processed by LSTMs to capture the sequential dynamics of the gesture (Papastratis et al., 2021 [9]; Ru and Sebastian, 2023 [10]). This two-stage approach enhances accuracy, particularly in real-world scenarios where both spatial and temporal information are crucial.

2.2.3 Graph Convolutional Networks (GCNs) for Hand Landmark Modeling

While CNNs and LSTMs have been central to prior gesture recongition system, GCNs have emerged as particularly powerful for modeling hand landmark data. Unlike CNNs, which operate on regular grid-like structures, GCNs process graph structured data and are thus well-suited for representing the complex spatial relationships between the 21 key hand landmakrs extracted by frameworks like MediaPipe (Sarkar et al., 2024[13], Dwivedi, 2021[14]).

In recent studies, GCNs integrated with residual connections have been shown to outperform traditional approaches in static ASL recognition, achieving state-of-the-art validation accuracy. These models benefit from spatial graph construction, normalization techniques, and machine learning enhancements such as dropout and batch normalization, which allow for robust generalization across varied hand postures.

2.2.4 Hybrid CNN-GCN Model

In our work, we integrate CNNs and GCNs to leverage their complementary strenghts. CNNs are used to extract global spatial features from landmark vectors or image-like inputs, while GCNs are used to model the graph-based structure of hand landmarks and their spatial dependencies. This hybrid approach has been increasingly explored in the literature and demonstrates superior accuracy, stability, and generalizability when applied to ASL recognition (Sarkar et al., 2024[13], Bronstein et al. 2021[15]).

2.2.5 Handling Static vs. Dynamic Signs

The choice of model architecture is dictated by the nature of the sign:

- **Static Signs:** When the sign consists of a fixed hand posture, CNNs and GCNs can be used alone to capture the necessary spatial features.
- **Dynamic Signs:** For gestures that involve motion, models such as RNNs and, more effectively, LSTMs are essential. Their ability to model long-term dependencies allows them to interpret the temporal evolution of a gesture accurately. Many systems adopt a two-stage approach, first applying CNNs for frame-wise feature extraction, followed by LSTMs to model the sequential aspects of the gesture (Lee et al., 2021 [11]).

2.3 Base Repository for Implementation

Our ASL recognition system builds upon multiple open-source repositories that provide foundational components for real-time gesture recognition.

The first is K. Takahashi's "hand-gesture-recognition-mediapipe" [12], which leverages the MediaPipe library for accurate and efficient real-time hand landmark detection. This repository provided a robust starting point for prototyping our gesture capture and preprocessing pipeline.

Additionally, we referenced a more recent project by Anindyadeep, "SignLangGNN" (GitHub, 2023) [16], which implements a GCN-based ASL classifier using landmark data. This work demonstrated the feasibility and performance benefits of GCNs for ASL recognition and was influential in the architectural design of our hybrid CNN-GCN system.

By drawing from these repositories, our system integrates fast and reliable landmark detection with machine learning models that specialize in spatial reasoning and gesture classification.

2.4 Challenges and Gaps in Current Systems

Despite significant advances in ASL recognition technologies, several critical challenges and gaps remain that hinder widespread adoption in real-world settings:

- **Limited Vocabulary and Generalization:** Many existing systems are designed to recognize only a restricted set of signs, making it difficult to scale to larger vocabularies. This limitation often stems from insufficient training data and models that are tuned for a narrow set of gestures (Debnath and Joe, 2024 [3]). Consequently, these systems struggle to generalize across different signers or adapt to variations in signing styles.

- **Difficulty Capturing Nonmanual Signals:** Effective sign language communication relies not only on hand gestures but also on nonmanual signals such as facial expressions and head tilts. Current approaches often neglect these critical features due to the complexities involved in accurately capturing and processing them. As a result, essential grammatical and emotional nuances are frequently lost, limiting the overall accuracy and naturalness of the interpretation (Avola et al., 2019 [4]).

- **Poor Performance in Real-World Environments:** Many systems demonstrate high accuracy under controlled conditions; however, performance typically degrades in real-world scenarios. Factors such as variable lighting conditions, dynamic backgrounds, and ambient noise can adversely affect both the feature extraction process and the robustness of the model (Liu et al., 2016 [6]).

- **User Interface and Accessibility Shortcomings:** Beyond the technical aspects of recognition, user experience plays a pivotal role in the adoption of ASL systems. Current implementations often suffer from complex calibration procedures, unintuitive interfaces, or hardware requirements that are not easily accessible to the Deaf community. Such issues can discourage user engagement and limit the practical utility of the technology (Hibbard et al., 2020 [1]).

2.5 Conclusion

In summary, while state-of-the-art ASL recognition systems have made commendable progress, significant gaps remain, particularly in vocabulary generalization, the integration of nonmanual signals, robustness in real-world conditions, and user accessibility. Our literature review highlights the evolution of model architectures from CNN and LSTM frameworks to more recent and effective GCN-based models.

This thesis proposes a hybrid architecture that combines CNNs and GCNs to address the limitations of both approaches. CNNs handle general spatial feature extraction, while GCNs effectively model the relationships between hand landmarks for fine-grained classification. By leveraging both models, our system aims to offer:

- **Real-Time Performance:** Ensuring quick and accurate recognition through efficient machine learning models and streamlined data processing pipelines.

- **Robust Detection:** Enhancing accuracy across a wide range of signs, including those with subtle spatial variations, by integrating graph-based reasoning with CNN-based feature extraction.

- **User-Friendly Interface:** Prioritizing accessibility and ease of use, with minimal calibration requirements and an intuitive design that caters to the needs of Deaf and hard-of-hearing individuals.

This foundation sets the stage for the methodology chapter, where we detail the architecture, training process, and evaluation protocols of the implemented ASL recognition system.

Chapter 3

Software and Application

This chapter describes the software design, implementation, and integration of the factors that make up the ASL recognition system. The system applies computer vision techniques with machine learning techniques to enable real-time sign language recognition. This chapter also dicusses the research process, including data collection, the algorithms and techniques employed, evaluation metrics, user engagement, and limitations and future direction.

3.1 Research Approach

The entire process integrates computer vision techniques via OpenCV and MediaPipe with a machine learning classifier in TensorFlow. Video frames are grabbed by an OpenCV webcam, which is then processed using MediaPipe's Hands solution to identify the 21 hand landmarks of the hands as shown in Figure 1 (each of x, y, and z coordinates). These 63-dimensional vectors are fed into the neural network model.

MediaPipe hand landmark detection provides speed and resilience for real-time usages, according to the MediaPipe Hands GitHub documentation[1]. OpenCV 3.4 takes care of frame capturing, image preparation, and placing the drawings on top of them for visualizing purposes, where each process stage is as efficient as effective[2].

[2]https://docs.opencv.org/3.4/index.html

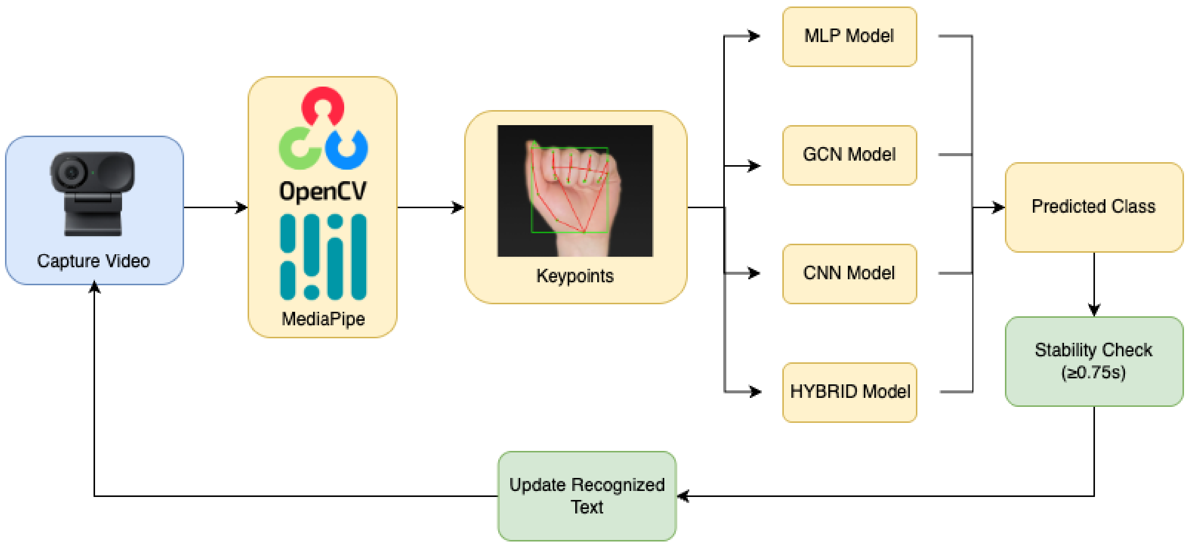


Figure 2: System flowchart: from webcam input through OpenCV preprocessing, landmark extraction via MediaPipe, model inference with TensorFlow, to output display with overlays.

3.2 Data Collection

The primary training data come from the ASL Alphabet Dataset on Kaggle[3]. The dataset consists of letter class-labeled images and contains additional classes of "space," "del," and "no_gesture" to simulate actual text input scenarios.

Horizontal flipping is applied to augment the dataset, introducing variability and allowing the model to generalize over different hand orientations. Literature on tackling data variability and augmentation techniques in sign language recognition (Papastratis et al., 2021 [9]; Ru and Sebastian, 2023 [10]) confirm this approach.

3.3 Data Preprocessing

The success of a machine learning pipeline depends greatly on how well raw data is preprocessed. In this project, the script `image_processing.py` handles the processing of ASL image data into model-ready features. This involves extracting MediaPipe landmarks, op-

[3]`https://www.kaggle.com/datasets/grassknoted/asl-alphabet`

tionally augmenting images, encoding labels, and serializing outputs for future reuse.

The preprocessing pipeline includes the following key steps:

- **Directory Traversal:** The script traverses a dataset directory in which each subfolder corresponds to an ASL class label, such as `A` or `space`. Each image is read and its label inferred from its parent folder.

```python
for label in os.listdir(data_dir):
    class_folder = os.path.join(data_dir, label)
    if os.path.isdir(class_folder):
        print(f"Processing images for class '{label}'")
        image_files = os.listdir(class_folder)
        for img_name in image_files:
            img_path = os.path.join(class_folder, img_name)
            image = cv.imread(img_path)
```

Excerpt 1: Directory Traversal and Image Loading

- **Landmark Extraction:** Images are resized to a fixed resolution, converted from BGR to RGB, and passed through MediaPipe's hand tracker. If a hand is detected, 21 landmarks (x, y, z) are extracted and flattened into a 63-dimensional feature vector.

```python
def extract_keypoints(image):
    """ Extract hand keypoints from an image using MediaPipe """
    try:
        # Resize image -> better performance
        image = cv.resize(image, (640, 480))
        # (640, 480) is default webcam resolution
        # Consistent input size will help improve speed
        # and stability of landmark detection

        # Convert image to RGB spectrum
        image_rgb = cv.cvtColor(image, cv.COLOR_BGR2RGB)
        # OpenCV images are by default BGR format
```

```
            results = hands.process(image_rgb)
            # Process image with MediaPipe

            if results.multi_hand_landmarks:
                landmarks = results.multi_hand_landmarks[0].landmark
                # Get hand's landmark
                # returns a list of 21 landmark points for a single
                    hand detected
                # Each landmark has an x, y, and z coordinate
                keypoints = []
                for landmark in landmarks:
                    keypoints.append(landmark.x)  # x-coord
                    keypoints.append(landmark.y)  # Y-coord
                    keypoints.append(landmark.z)  # Z-coord (depth)
                return keypoints
```

Excerpt 2: MediaPipe Keypoint Extraction

- **Data Augmentation:** If enabled, the script also horizontally flips each image and re-extracts landmarks, increasing dataset variability and model robustness.

```
    if augment:
        flipped_img = cv.flip(image, 1)  # Flip horizontally
        flipped_keypoints = extract_keypoints(flipped_img)
        if flipped_keypoints:
            images.append(flipped_keypoints)
            labels.append(label)
```

Excerpt 3: Image Augmentation with Horizontal Flip

- **Label Encoding:** Labels collected from folder names are encoded into integer class indices using `LabelEncoder` from scikit-learn to prepare for classification tasks.

```
    label_encoder = LabelEncoder()
    labels = label_encoder.fit_transform(labels)
```

Excerpt 4: Encoding String Labels to Integers

- **Serialization:** After processing, the features (`X`), encoded labels (`y`), and label encoder object are serialized into a pickle file for fast loading in future training or evaluation runs.

```
    with open("processed_train.pkl", "wb") as f:
        pickle.dump((X_train, y_train, le_train), f)
```

Excerpt 5: Saving Preprocessed Data

This modular and reusable pipeline promises standard class labels and input dimensions while augmenting the training data to increase robustness. By serializing the processed output, subsequent model training phases avoid duplicate computation, enhancing development efficiency.

3.4 Model Architectures

This thesis implements and evaluates four different model architectures, each optimized to classify 29 ASL gesture classes from 63-dimensional hand landmark vectors. The models employed are a Multilayer Perceptron (MLP), Convolutional Neural Network (CNN), Graph Convolutional Network (GCN), and a hybrid CNN-GCN model.

3.4.1 Multilayer Perceptron (MLP)

The MLP is employed as the base model. It is a fully connected feedforward neural network with two hidden layers of 128 and 64 units, respectively. ReLU activation functions are used throughout, and dropout layers are included to prevent overfitting. The final output layer uses a softmax activation to predict one of the 29 gesture classes.

```python
model = Sequential([
    # First Dense layer with input dimension
    Dense(128, input_dim=X_train.shape[1], activation='relu'),
    # Dropout layer to reduce overfitting
    Dropout(0.5),
    # Second Dense layer
    Dense(64, activation='relu'),
    # Another Dropout layer
    Dropout(0.5),
    # Final output layer with 'softmax' activation
    Dense(len(train_le.classes_), activation='softmax')
])
```

Excerpt 6: MLP Model Definition

This model is lightweight and fast, making it ideal for early experimentation and rapid iteration, although it lacks the spatial awareness of more advanced models.

3.4.2 Graph Convolutional Network (GCN)

The Graph Convolutional Network (GCN) is a graph with 21 nodes, where each node corresponds to a MediaPipe landmark, and edges representing anatomical connections such as those between a fingertip and its associated joint. This type of structure enables the model to learn spatial and relational dependencies among parts of the hand, which is required for distinguishing gestures with subtle local differences.

Adjacency Matrix Construction

We define the hand graph using MediaPipe's anatomical hand connections. These are used to construct an undirected adjacency matrix, with additional self-loops added to preserve each node's identity.

```python
A = np.zeros((21, 21), dtype=np.float32)
edges = [
    (0, 1), (1, 2), (2, 3), (3, 4),         # Thumb
    (0, 5), (5, 6), (6, 7), (7, 8),         # Index finger
    (0, 9), (9, 10), (10, 11), (11, 12),    # Middle finger
    (0, 13), (13, 14), (14, 15), (15, 16), # Ring finger
    (0, 17), (17, 18), (18, 19), (19, 20)  # Pinky finger
]
for i, j in edges:
    A[i, j] = A[j, i] = 1
for i in range(21):
    A[i, i] = 1  # Add self-loops
```

Excerpt 7: Adjacency Matrix Definition

Graph Normalization

To perform spectral graph convolution, we normalize the adjacency matrix using symmetric normalization:

$$\hat{A} = D^{-1/2} A D^{-1/2}$$

where D is the degree matrix of A. This step balances each node's influence across its neighbors during feature propagation.

```
D = np.diag(np.sum(A, axis=1))
D_inv_sqrt = np.linalg.inv(np.sqrt(D))
A_norm = D_inv_sqrt @ A @ D_inv_sqrt  # shape: (21, 21)
A_norm_tf = tf.constant(A_norm, dtype=tf.float32)
```

Excerpt 8: Normalization of Adjacency Matrix

GCN Architecture

The model architecture consists of two custom `GCNLayer` components that apply the normalized adjacency matrix in order to spread features across the graph. This is followed by dense layers for classification.

```
inputs = tf.keras.Input(shape=(num_nodes, 3))          # shape: (21, 3)
x = GCNLayer(64, activation='relu')(inputs, A_norm_tf) # First GCN layer
x = GCNLayer(64, activation='relu')(x, A_norm_tf)      # Second GCN layer
x = tf.keras.layers.Flatten()(x)
x = tf.keras.layers.Dense(128, activation='relu')(x)
x = tf.keras.layers.Dropout(0.5)(x)
outputs = tf.keras.layers.Dense(num_classes, activation='softmax')(x)
model = tf.keras.Model(inputs=inputs, outputs=outputs)
```

Excerpt 9: GCN Model Architecture

Why GCNs Matter

GCNs are particularly effective for structured data like human hands, where relationships between joints matter. Unlike MLPs that treat all features equally, GCNs learn to emphasize local connections, such as those between finger joints, and model spatial hierarchies. This is especially useful for distinguishing gestures that are globally similar but locally distinct, such as `M`, `N`, and `T`.

3.4.3 Convolutional Neural Network (CNN)

The CNN treats the 21 hand landmarks as a pseudo-image of shape $(21, 3, 1)$. Two convolutional layers extract local spatial features, followed by max pooling layers to reduce dimensionality. The resulting feature maps are flattened and passed through dense layers.

```
    model = Sequential([
        # Convolution #1: preserve the shape with same padding
        Conv2D(32, (3, 3), activation='relu', padding='same',
               input_shape=(21, 3, 1)),
        # Pool over the "landmark dimension" only
        MaxPooling2D(pool_size=(2, 1)),
        # Convolution #2: again use same padding
        Conv2D(64, (3, 3), activation='relu', padding='same'),
```

```python
        MaxPooling2D(pool_size=(2, 1)),
        # Convolution #3: optional
        # Conv2D(128, (3, 3), activation='relu', padding='same'),
        # MaxPooling2D(pool_size=(2, 1)),
        Flatten(),
        Dense(128, activation='relu'),
        Dropout(0.5),
        Dense(num_classes, activation='softmax')
    ])
```

Excerpt 10: CNN Model Definition

CNNs are well-suited for capturing fine-grained spatial patterns. While they don't model graph relationships directly, they can still detect unique positional characteristics of hand gestures.

3.4.4 Combined CNN-GCN Model

To leverage the strengths of both spatial and structural modeling, a hybrid model is constructed with two parallel branches: one CNN and one GCN. Each branch processes the same input data (reshaped landmark vectors) and extracts distinct feature types. The outputs are concatenated and passed through fully connected layers for final classification.

```python
def build_combined_model(num_classes, adjacency_matrix):
    inputs = Input(shape=(21, 3), name='input_landmarks')

    # GCN branch
    gcn_out = GCNLayer(64, activation='relu')(inputs, adjacency_matrix)
    gcn_out = GCNLayer(64, activation='relu')(gcn_out, adjacency_matrix)
    gcn_out = Flatten()(gcn_out)
    gcn_out = Dense(64, activation='relu')(gcn_out)

    # CNN branch
    cnn_input = Reshape((21, 3, 1))(inputs)
    x = Conv2D(32, (3, 3), activation='relu', padding='same')(cnn_input)
    x = MaxPooling2D(pool_size=(2, 1))(x)
    x = Conv2D(64, (3, 3), activation='relu', padding='same')(x)
    x = MaxPooling2D(pool_size=(2, 1))(x)
    x = Flatten()(x)
    x = Dense(64, activation='relu')(x)

    # Merge both branches
    merged = Concatenate()([gcn_out, x])
    merged = Dense(128, activation='relu')(merged)
    merged = Dropout(0.5)(merged)
    outputs = Dense(num_classes, activation='softmax')(merged)

    model = Model(inputs=inputs, outputs=outputs)
    model.compile(
        optimizer='adam',
        loss='sparse_categorical_crossentropy',
        metrics=['accuracy']
    )
    return model
```

Excerpt 11: Hybrid CNN-GCN Model

This model exhibits superior performance in terms of validation accuracy and generalization. It combines the CNN's spatial filtering capabilities with the GCN's structural understanding of the hand's anatomy.

3.5 Evaluation Criteria

All models are evaluated using the following metrics:

- **Validation Accuracy:** Measures generalization to unseen data using a held-out 20% validation set.

- **Training Loss and Accuracy:** Tracked per epoch to monitor convergence and detect overfitting.

```
    history = model.fit(
        X_train, y_train,
        validation_data=(X_val, y_val),
        epochs=10,
        batch_size=32
        )
```

Excerpt 12: Model Training with Validation

- **Frames Per Second (FPS):** Measured using `cvfpscalc.py` during live webcam inference. Real-time performance aims for 15–30 FPS.

- **Gesture Stability Logic:** A prediction is considered valid only if the same gesture is held for 0.75–1.00 seconds, reducing noise and transient misclassifications.

```
    if detected_letter == stable_letter:
        if time.time() - stable_start_time >= 0.75:
            recognized_text += detected_letter
```

Excerpt 13: Gesture Stability Check

- **User Responsiveness:** Informally assessed through hands-on testing across different lighting conditions, background noise, and camera angles.

These criteria provide a comprehensive view of each model's usability and real-world effectiveness.

3.6 Summary

This chapter presented the implementation of the real-time ASL recognition system, covering preprocessing, model training, and system deployment. By combining landmark-based feature extraction with multiple machine learning architectures, the system provides robust static sign recognition and real-time feedback.

In the next chapter, a comparative analysis will be conducted to evaluate how each model fares under different conditions, highlighting trade-offs in accuracy, speed, and responsiveness.

Chapter 4

Data Analysis

This chapter evaluates the performance of the American Sign Language (ASL) recognition models presented in Chapter 3. Our main goal is to compare each architecture's ability to classify static ASL signs accurately and efficiently, using the same training dataset and 20% held-out validation set. We present both quantitative and qualitative findings, including accuracy metrics, confusion matrices, and notable misclassification trends.

4.1 Overview of the Dataset and Evaluation Procedure

Following the methodology in Chapter 3, we processed the *ASL Alphabet Dataset* to obtain 63-dimensional hand-landmark features per image using *MediaPipe Hands*. Overall:

- **29 classes:** Letters A–Z plus "del," "nothing," and "space."
- $\sim$ 124,000 **samples:** After including horizontal-flip augmentation.
- **Train–Validation Split:** 80% for training, 20% for validation (stratified), ensuring enough samples of each class.

All models, including the Multilayer Perceptron (MLP), Graph Convolutional Network (GCN), Convolutional Neural Network (CNN), and a combined CNN and GCN hybrid, were trained on the same dataset split for a fair comparison. Key metrics include:

- **Accuracy** on the validation set (top-1 classification).
- **Loss** (cross-entropy), measuring average prediction error.
- **Confusion Matrix Analysis** to identify commonly confused sign pairs.

4.2 Multilayer Perceptron (MLP)

Architecture and Training. The MLP takes each sample as a 63-dimensional feature vector and passes it through two fully connected layers of sizes 128 and 64 (both with `ReLU` activations), sepearted by dropout layers for regularization, and concludes with a final 29-way softmax layer. After 10 epochs of training, the model typically converged around **90–91%** validation accuracy. Figure 3 shows how the training and validation accuracy/loss evolve over these 10 epochs.

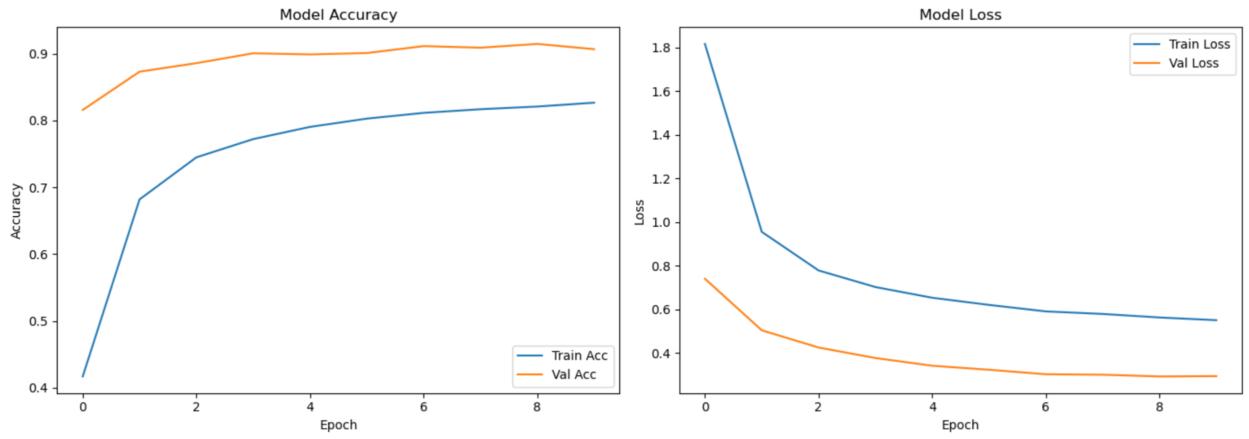


Figure 3: MLP Model Accuracy and Loss Curves

Validation Performance. With a final validation accuracy of approximately **90.80%** and a validation loss near **0.29**, the MLP successfully identifies many distinct letters. However, it struggles with signs whose landmark positions differ only subtly, indicating the limitations of a fully connected approach without explicit spatial-awareness mechanisms.

Confusion Matrix Analysis. Figure 4 presents the MLP's confusion matrix, where rows represent the true class and columns represent the predicted class. Though the diagonal is generally dark (indicating correct predictions), off-diagonal errors remain. Notably, M is commonly mistaken for N, P overlaps with Q, and R, U, V can be confused with each other. The letters T, X, and Z also pose challenges. These patterns highlight that very similar handshapes still cause misclassifications for an MLP that does not explicitly incorporate the spatial or relational structure of the hand.

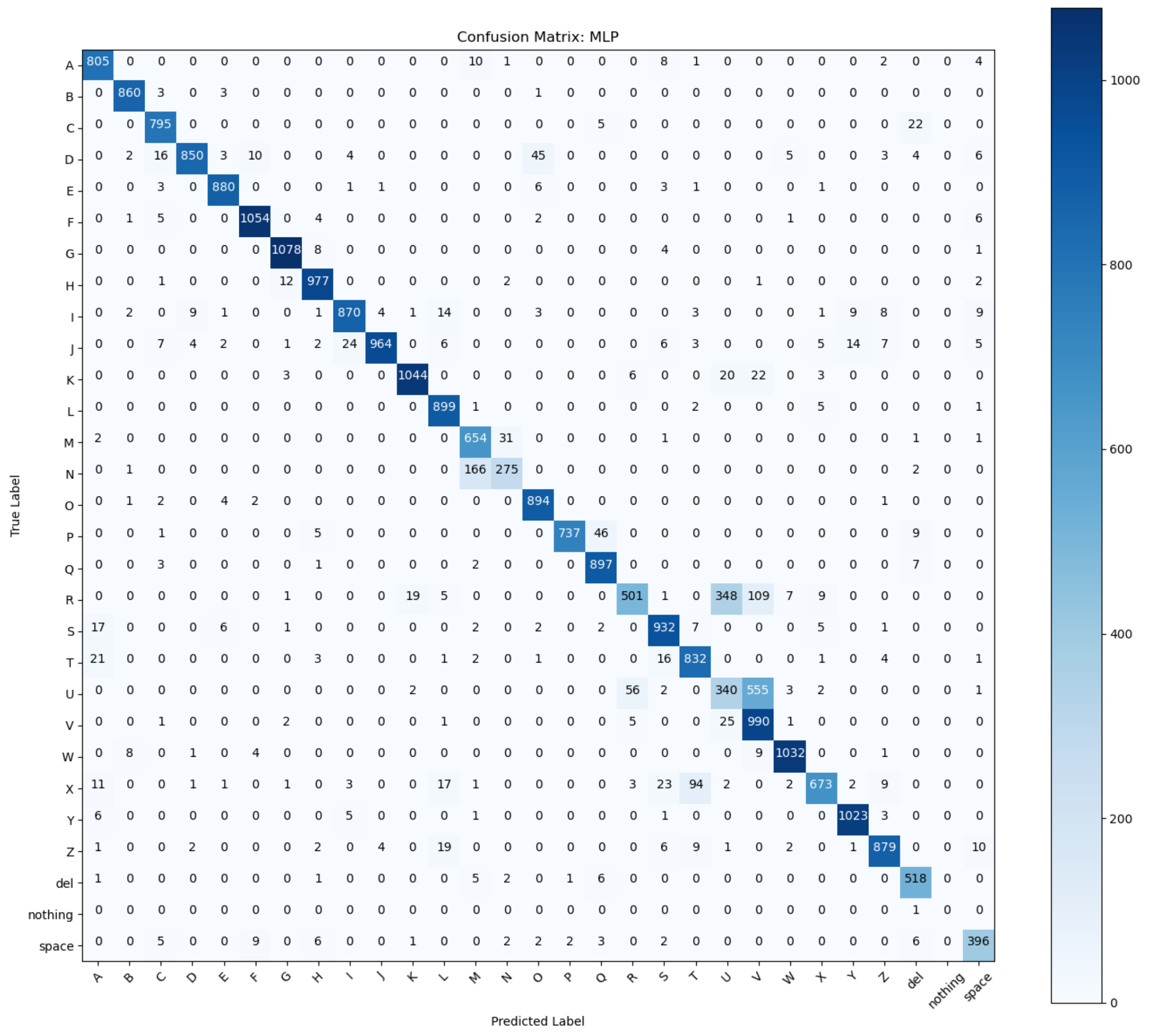


Figure 4: Confusion Matrix for the MLP Model

4.3 Graph Convolutional Network (GCN)

Architecture and Training. The GCN encodes hand structure by modeling each of the 21 landmarks as nodes in a graph, with edges reflecting finger-joint connectivity. Two graph convolutional layers (each followed by nonlinearity and dropout) process these relationships before a final softmax classification. This approach is designed to capture local joint-to-joint features that are often lost in a fully connected network. After about 10 epochs of training, the validation accuracy converged around **91–92%**. Figure 5 shows the training and validation accuracy/loss curves.

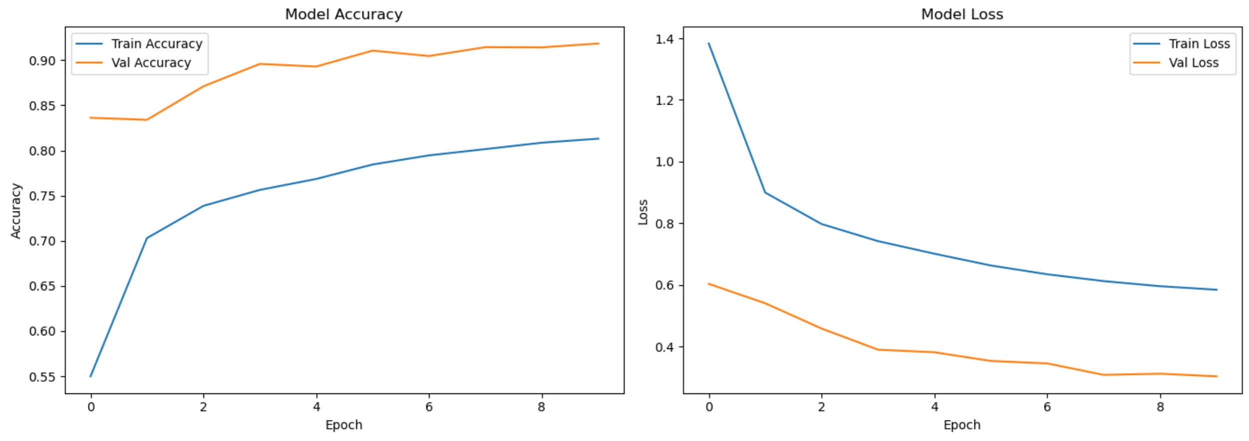


Figure 5: GCN Model Accuracy and Loss Curves

Validation Performance. Compared to the MLP, the GCN exhibits a modest but consistent improvement, finalizing near **91.85%** validation accuracy and around **0.30** validation loss. By leveraging joint adjacency, the GCN better distinguishes subtle landmark configurations.

Confusion Matrix Analysis. Figure 6 shows the GCN's confusion matrix, which still has misclassifications even though the diagonal is more pronounced than in the MLP. The most notable errors include J often misread as *space*, persistent confusion between M and N, and difficulties separating U and V or R when finger positions are not clearly distinct. While T shows improvement over the MLP, a small subset of samples is still incorrectly labeled as M, and Z occasionally appears as M. Despite these errors, the matrix demonstrates that incorporating relational cues reduces overall confusion, particularly for classes previously misclassified by the MLP.

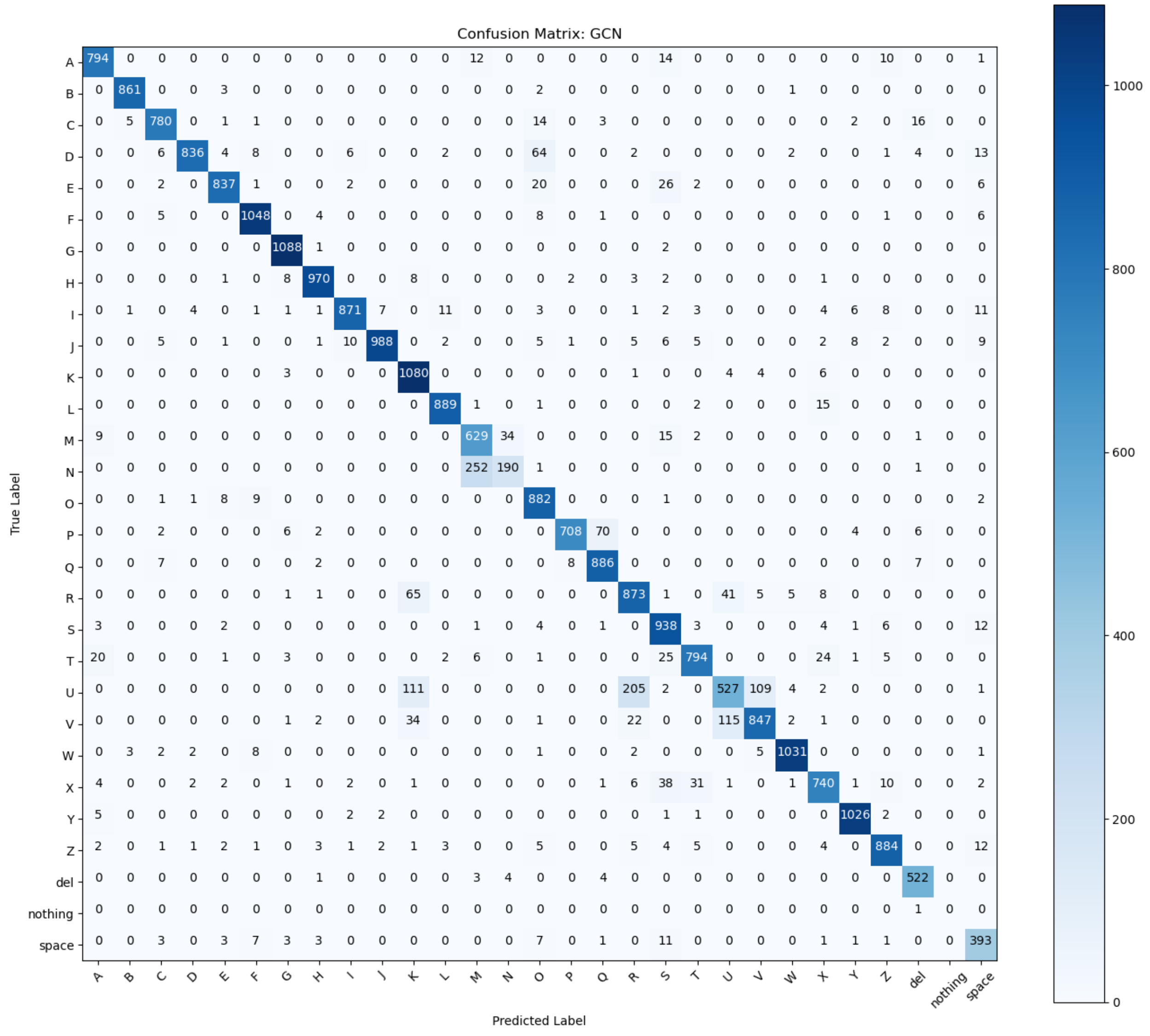


Figure 6: Confusion Matrix for the GCN Model

4.4 Convolutional Neural Network (CNN)

Architecture and Training. By treating the (21×3) landmark matrix as a “pseudo-image” (height=21, width=3, channels=1), the CNN applies 2D convolutions and pooling layers to learn local spatial features automatically. These features are then flattened and passed through fully connected layers to produce a final classification over 29 classes. After approximately 10 epochs, the model converged to a high accuracy, indicating that 2D filters are well-suited to the relatively small (21×3) input. Figure 7 shows the training and validation accuracy/loss curves, which quickly plateau at near-optimal levels.

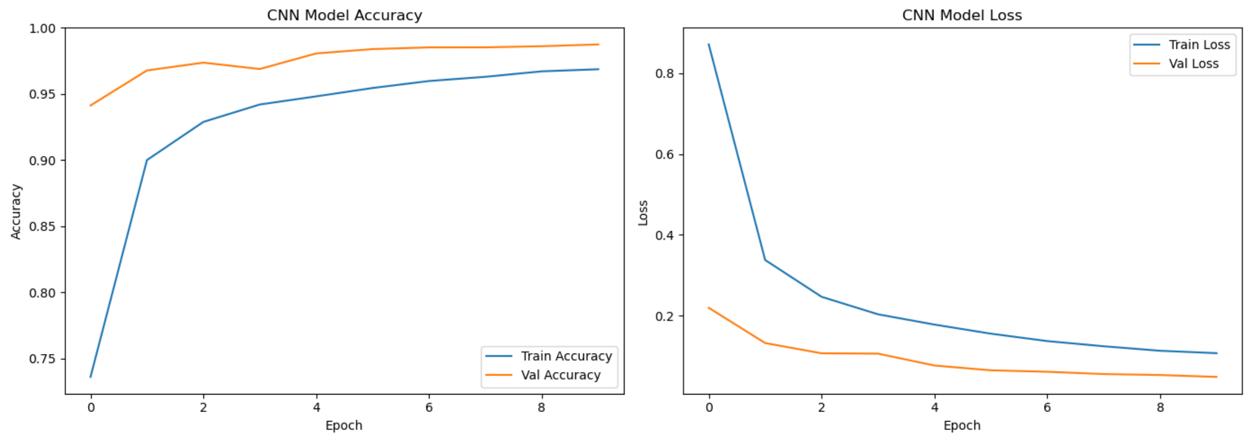


Figure 7: CNN Model Accuracy and Loss Curves

Validation Performance. Compared to both MLP and GCN, the CNN achieves notably higher performance, hovering around **98.75%** validation accuracy with a validation loss near **0.05**. These results reflect the CNN's ability to capture both fine-grained local structure and overall hand configuration via learned convolutional filters. Although many classes are correctly identified nearly all of the time, a small set of signs still poses challenges.

Confusion Matrix Analysis. Figure 8 illustrates the CNN's confusion matrix, which is close to an identity matrix but not entirely free of errors. Despite the diagram showing minimal confusion, the model can still struggle with angles when distinguishing G and H, especially if the thumb or index finger orientation is unclear. J also remains problematic, as small differences in the curved finger pose can cause misclassification. Additionally, T sometimes fails to be detected entirely, suggesting that certain finger overlaps or partial occlusions confuse the CNN. Nonetheless, these issues are relatively isolated compared to those seen in MLP and GCN, and overall confusion is significantly reduced, reflecting how 2D convolutions excel at extracting robust spatial patterns from the pseudo-image representation.

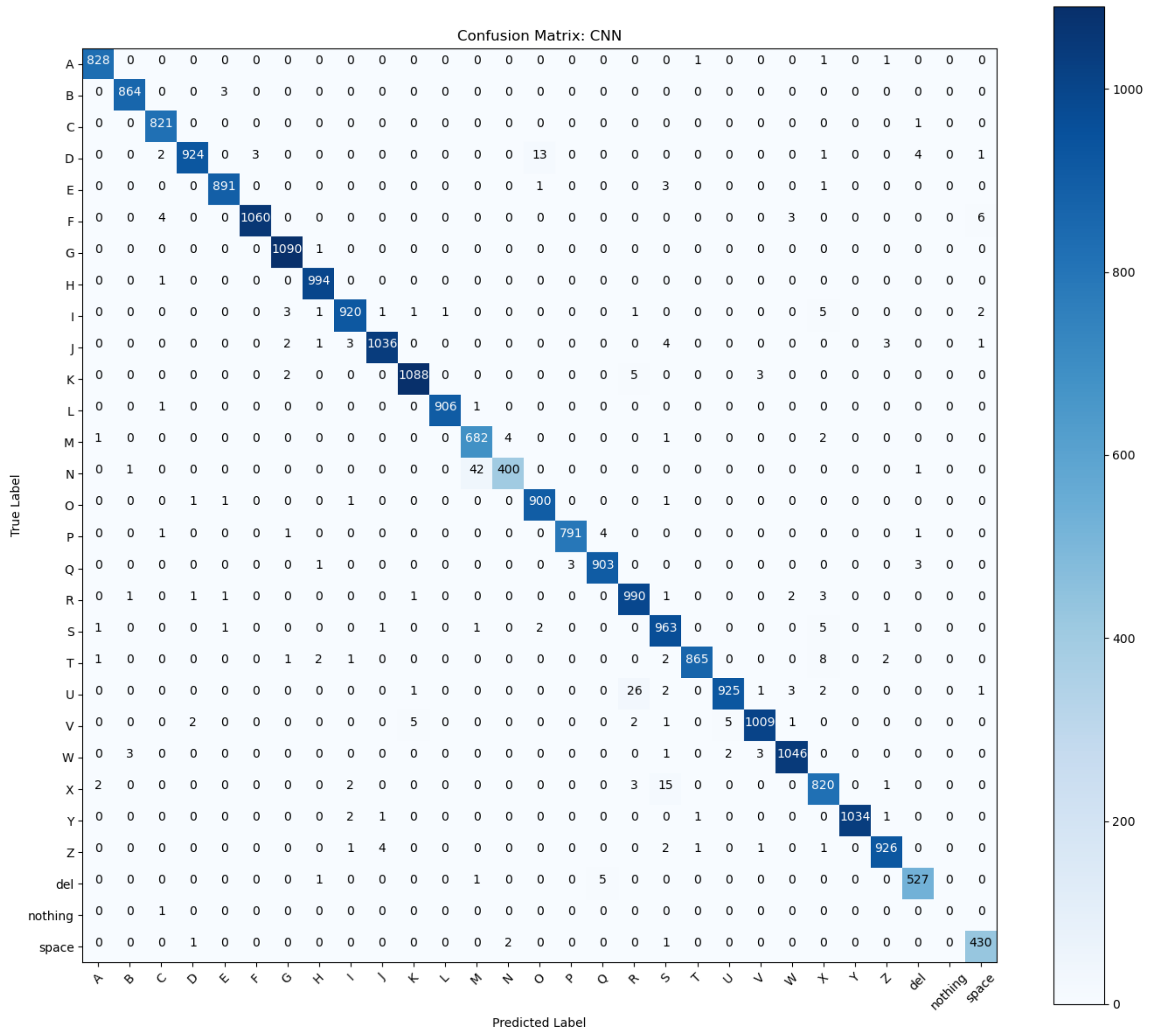


Figure 8: Confusion Matrix for the CNN Model

4.5 Combined CNN–GCN Model

Architecture and Training. To exploit both convolutional feature extraction (*local spatial patterns*) and graph-based relational reasoning (*joint connectivity*), the combined approach passes the same (21×3) landmark matrix into parallel CNN and GCN branches. The CNN processes a pseudo-image representation, while the GCN leverages an adjacency matrix capturing finger-joint connections. The outputs of both branches are concatenated and fed through fully connected layers before the final softmax classification. After roughly 10 epochs, the model converges to a higher accuracy than either the standalone CNN or GCN. Figure 9 illustrates the training and validation curves, showing rapid convergence toward minimal loss and near-perfect accuracy.

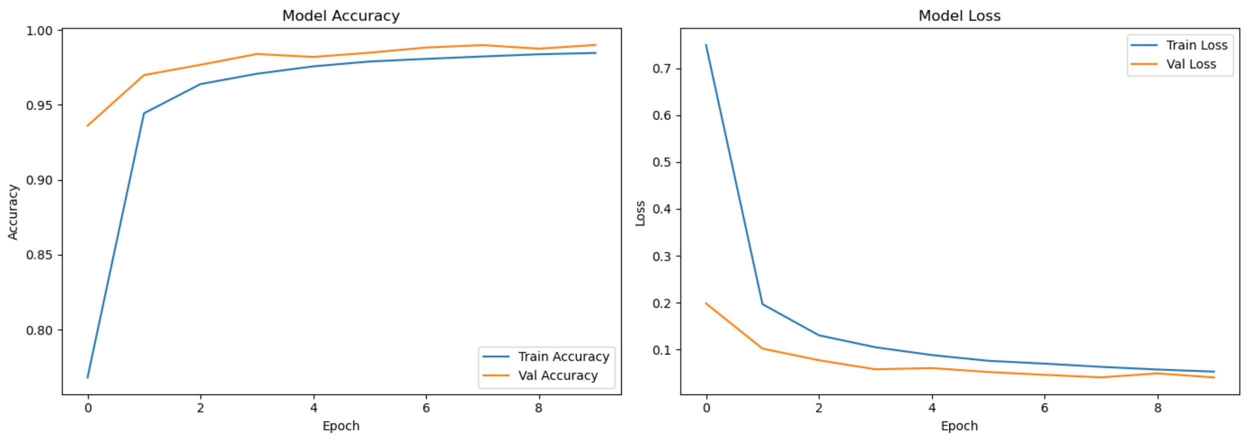


Figure 9: Combined CNN–GCN Model Accuracy and Loss Curves

Validation Performance. With a final validation accuracy of approximately **99.01%** and a validation loss around **0.04**, the combined architecture achieves the best overall performance. By merging the CNN's capacity to learn visual/spatial patterns with the GCN's emphasis on relational cues, this hybrid design can resolve many edge cases left unaddressed by the individual models.

Confusion Matrix Analysis. Figure 10 displays the confusion matrix for the combined model. Overall, it is nearly an identity matrix, reflecting minimal misclassifications. The few remaining errors are extremely sparse, often resulting from subtle angles or momentary landmark overlaps, such as those seen in signs that occasionally challenge the CNN or GCN. Nevertheless, the synergy of CNN and GCN features greatly reduces confusion across all letters. In practice, this approach almost fully disambiguates similar hand configurations such as *M* vs. *N* and captures curved poses like *J*. The ability to integrate both local and relational details helps push classification performance close to the upper limit on this dataset.

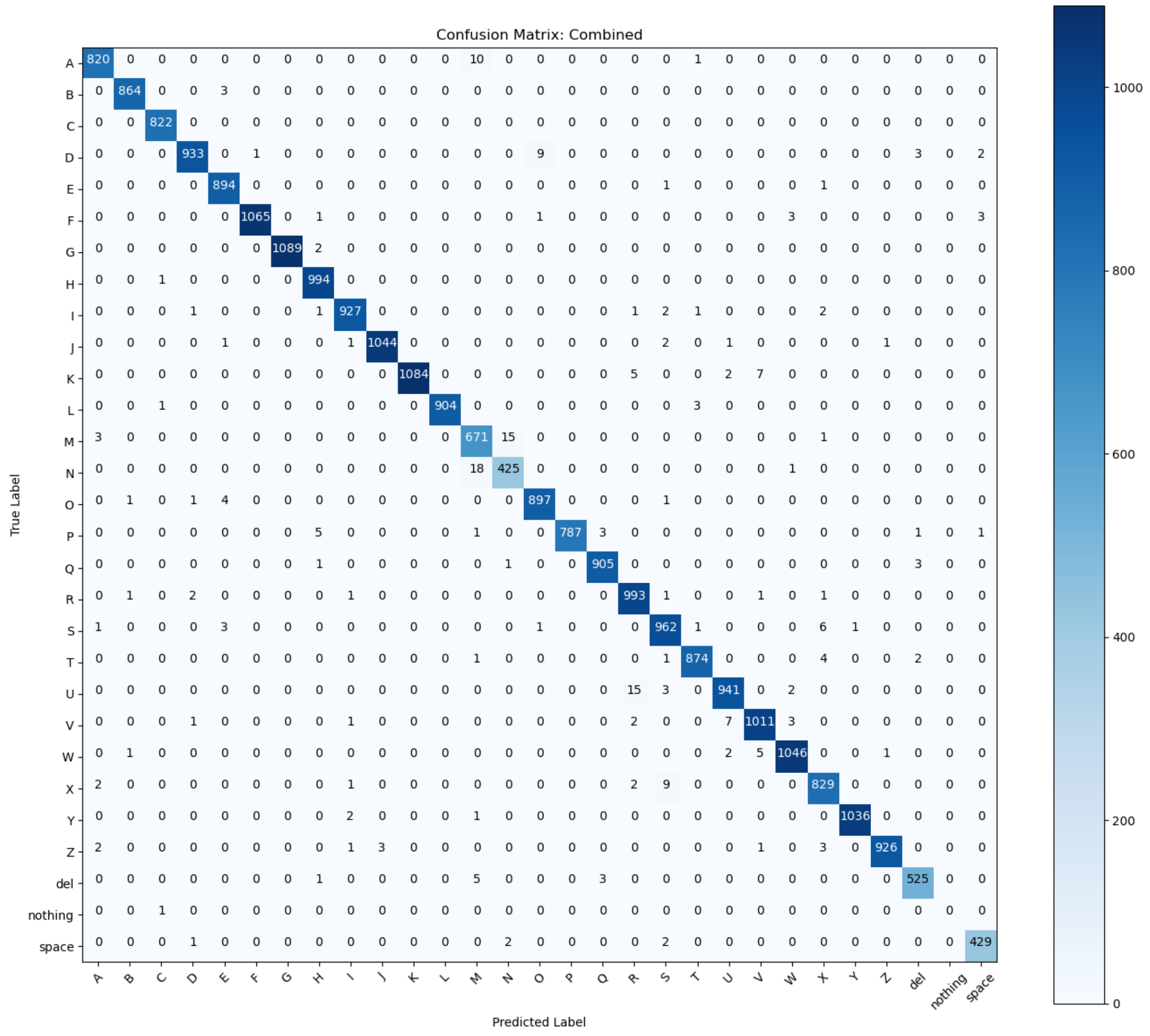


Figure 10: Confusion Matrix for the Combined CNN–GCN Model

4.6 Summary of Findings

Table 1 provides a concise comparison of each model's validation accuracy and loss. The MLP, while simple, struggles with subtle handshape differences; the GCN modestly improves classification by incorporating hand-joint connectivity; the CNN shows a major leap in performance via learned 2D filters on the landmark matrix; and the combined CNN–GCN achieves the highest accuracy by fusing convolutional and graph-based representations.

Table 1: Summary of Validation Performance for Each Model

Model	Validation Accuracy	Validation Loss
MLP	90.80%	0.29
GCN	91.85%	0.30
CNN	98.75%	0.05
Combined CNN–GCN	**99.01%**	**0.04**

Across all models, confusion occurs mainly among letters with highly similar handshapes (*M* vs. *N*, *R* vs. *U* vs. *V*, etc.) or signs that require precise angles (*G*, *H*, *J*, *Z*, etc.). While the CNN and combined CNN–GCN handle these subtleties more effectively than MLP or GCN alone, occasional errors persist, particularly in ambiguous poses or images with poor landmark extraction. Overall, the results underscore the importance of leveraging both local spatial patterns and relational cues for robust static ASL sign recognition.

4.7 Conclusions

In this chapter, we compared four machine learning architectures: MLP, GCN, CNN, and a hybrid CNN and GCN model for static ASL sign classification. The MLP provided a useful baseline but struggled with subtle or complex hand configurations. The GCN demonstrated that encoding graph-like relationships among joints helps distinguish challenging classes, though the improvements were modest compared to the MLP. In contrast, the CNN's 2D convolutions on the landmark matrix delivered a substantial increase in accuracy by effectively learning spatial filters. Finally, the combined CNN and GCN model outperformed all others by fusing local convolutional features with graph-based hand structure cues, achieving the highest validation accuracy. These findings suggest that incorporating both spatial and relational insights is key to performance on static ASL letter recognition and lays a solid foundation for more advanced sequence-based or real-time applications in future work.

Chapter 5

Conclusion

The design and implementation of a real-time American Sign Language (ASL) recognition system represents a meaningful step toward reducing communication barriers for Deaf and hard-of-hearing individuals. This thesis systematically compares four model architectures: MLP, GCN, CNN, and a combined CNN and GCN variant. It demonstrates that leveraging both spatial and relational features is pivotal in accurately classifying ASL signs. MediaPipe Hands, integrated with OpenCV, offered a reliable and efficient method for extracting critical hand landmarks in real time, enabling robust performance across varied signing conditions.

The comparative analysis yielded valuable insights into model performance. Whereas the MLP and GCN alone captured high-level distinctions, the CNN's spatial filtering bolstered feature extraction for complex gestures. The hybrid CNN–GCN model, merging local convolutional features with graph-based joint relationships, ultimately achieved the highest accuracy and reduced misclassifications for challenging sign pairs like *M* vs. *N* or *V* vs. *U*. These findings underscore the importance of incorporating both pixel-level patterns and anatomical constraints for robust recognition.

5.1 Limitations and Future Directions

Despite the promising results, this system mainly addresses static ASL letters, leaving dynamic signs and facial expressions for future exploration. Incorporating nonmanual cues such as raised eyebrows and head tilts, along with continuous gesture recognition, will require more advanced temporal modeling and possibly multi-modal sensor inputs. Additionally, expanding the vocabulary to include word-level or phrase-level signs will demand richer datasets, improved data augmentation techniques, and more sophisticated sequence models. Collaboration with Deaf and hard-of-hearing communities will be essential to guide interface design and validate real-world usability across diverse environments.

In sum, this thesis provides a solid foundation for automated ASL recognition by showcasing how contemporary AI frameworks can be brought together to solve a critical accessibility challenge. The work opens avenues for extending robust recognition to continuous signing scenarios and integrating additional linguistic nuances. Ultimately, these innovations promise to enhance communication channels, foster inclusive learning environments, and pave the way for wider adoption of ASL technologies in everyday life.

Bibliography

[1] E. Hibbard *et al.*, "Getting a Sign in Edgewise: User-Centered Design Considerations in Creating a Signed Language Mentoring Management System," *Sign Language Studies*, vol. 20, no. 2, pp. 264–300, 2020. Available: `https://www.jstor.org/stable/26983963`

[2] V. Falvo, L. P. Scatalon, and E. F. Barbosa, "The Role of Technology to Teaching and Learning Sign Languages: A Systematic Mapping," in *2020 IEEE Frontiers in Education Conference (FIE)*, Uppsala, Sweden, 2020, pp. 1–9, doi: `10.1109/FIE44824.2020.9274169`. Available: `https://ieeexplore-ieee-org.libezproxy2.syr.edu/document/9274169`

[3] J. Debnath and P. J. I R, "Real-Time Gesture Based Sign Language Recognition System," in *2024 International Conference on Advances in Data Engineering and Intelligent Computing Systems (ADICS)*, Chennai, India, 2024, pp. 01–06, doi: `10.1109/ADICS58448.2024.10533518`. Available: `https://ieeexplore-ieee-org.libezproxy2.syr.edu/document/10533518`

[4] D. Avola, M. Bernardi, L. Cinque, G. L. Foresti, and C. Massaroni, "Exploiting Recurrent Neural Networks and Leap Motion Controller for the Recognition of Sign Language and Semaphoric Hand Gestures," *IEEE Transactions on Multimedia*, vol. 21, no. 1, pp. 234–245, Jan. 2019, doi: `10.1109/TMM.2018.2856094`. Available: `https://ieeexplore.ieee.org/document/8410764`

[5] Stefanidis, Kiriakos, Dimitrios Konstantinidis, Thanassis Kalvourtzis, Kosmas Dimitropoulos, and Petros Daras. "3D technologies and applications in sign language." 2020. Available: `https://www.researchgate.net/publication/340966069_3D_technologies_and_applications_in_sign_language`

[6] T. Liu, W. Zhou, and H. Li, "Sign language recognition with long short-term memory," in *2016 IEEE International Conference on Image Processing (ICIP)*, Phoenix, AZ, USA, 2016, pp. 2871–2875, doi: `10.1109/ICIP.2016.7532884`. Available: `https://ieeexplore-ieee-org.libezproxy2.syr.edu/document/7532884`

[7] S. Hochreiter and J. Schmidhuber, "Long Short-Term Memory," in *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, Nov. 15, 1997, doi: `10.1162/neco.1997.9.8.1735`. Available: `https://ieeexplore.ieee.org/document/6795963`

[8] C. Olah, “Understanding LSTM Networks,” Understanding LSTM Networks, Aug. 17, 2015. [Online]. Available: https://colah.github.io/posts/2015-08-Understanding-LSTMs/

[9] I. Papastratis *et al.*, “Artificial Intelligence Technologies for Sign Language,” *Sensors (Basel, Switzerland)*, vol. 21, no. 17, p. 5843, Aug. 30, 2021, doi: 10.3390/s21175843. Available: https://pmc.ncbi.nlm.nih.gov/articles/PMC8434597/

[10] J. T. S. Ru and P. Sebastian, “Real-Time American Sign Language (ASL) Interpretation,” in *2023 2nd International Conference on Vision Towards Emerging Trends in Communication and Networking Technologies (ViTECoN)*, Vellore, India, 2023, pp. 1–6, doi: 10.1109/ViTECoN58111.2023.10157157. Available: https://ieeexplore-ieee-org.libezproxy2.syr.edu/document/10157157

[11] C. K. M. Lee, K. K. H. Ng, C.-H. Chen, H. C. W. Lau, S. Y. Chung, and T. Tsoi, “American sign language recognition and training method with recurrent neural network,” *Expert Systems with Applications*, vol. 167, pp. 114403, 2021, ISSN 0957-4174. Available: https://doi.org/10.1016/j.eswa.2020.114403

[12] K. Takahashi, “hand-gesture-recognition-mediapipe,” GitHub repository, Available: https://github.com/kinivi/hand-gesture-recognition-mediapipe (Accessed: Feb. 26, 2025)

[13] U. Sarkar, A. Chakraborti, T. Samanta, S. Pal, and A. Das, “Enhancing ASL Recognition with GCNs and Successive Residual Connections,” *arXiv preprint arXiv:2408.09567*, Aug. 2024. Available: https://arxiv.org/abs/2408.09567

[14] C. Dwivedi, “Graph Convolutional Networks: Introduction to GNNs,” *Medium*, Apr. 7, 2021. [Online]. Available: https://medium.com/data-science/graph-convolutional-networks-introduction-to-gnns-24b3f60d6c95

[15] M. Bronstein *et al.*, “A Gentle Introduction to Graph Neural Networks,” *Distill*, 2021. [Online]. Available: https://distill.pub/2021/gnn-intro/

[16] Anindyadeep, *SignLangGNN: Graph-based Sign Language Recognition using GCNs*, GitHub repository, 2023. Available: https://github.com/Anindyadeep/SignLangGNN

Appendices

List of Figures

List of Excerpts

Model Evaluation Notebook

Dataset processing, in `image_processing.py`

Imports

```python
import os
import time
import pickle
import numpy as np
import cv2 as cv
import pandas as pd
import mediapipe as mp
import tensorflow as tf
from sklearn.preprocessing import LabelEncoder
from collections import Counter
import matplotlib.pyplot as plt
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import (
    Input, Dense, Dropout, Flatten, Conv2D, MaxPooling2D, Concatenate,
Reshape
)
from tensorflow.keras.models import Model
from tensorflow.keras.optimizers import Adam
from sklearn.model_selection import train_test_split
from tensorflow.keras.models import load_model
from sklearn.metrics import confusion_matrix
```

MediaPipe Setup

```python
# Configure MediaPipe hands
mp_hands = mp.solutions.hands
hands = mp_hands.Hands(
    static_image_mode=True,
    max_num_hands=1,
    min_detection_confidence=0.5,
    min_tracking_confidence=0.5
)
```

```
WARNING: All log messages before absl::InitializeLog() is called are
written to STDERR
I0000 00:00:1743573969.017603 8171148 gl_context.cc:369] GL version:
2.1 (2.1 Metal - 89.3), renderer: Apple M1 Pro

INFO: Created TensorFlow Lite XNNPACK delegate for CPU.
W0000 00:00:1743573969.027181 8171413
inference_feedback_manager.cc:114] Feedback manager requires a model
with a single signature inference. Disabling support for feedback
tensors.
W0000 00:00:1743573969.033136 8171418
```

```
inference_feedback_manager.cc:114] Feedback manager requires a model
with a single signature inference. Disabling support for feedback
tensors.
```

Hand keypoint extraction function `extract_keypoints`

```
def extract_keypoints(image):
    """ Extract hand keypoints from an image using MediaPipe """
    try:
        # Resize image -> better performance
        image = cv.resize(image, (640, 480)) # (640, 480) is default
webcam resolution
                                             # Consistent input size
will help improve speed
                                             # and stability of
landmark detection

        # Convert image to RGB spectrum
        image_rgb = cv.cvtColor(image, cv.COLOR_BGR2RGB) # OpenCV
images are by default BGR format
        results = hands.process(image_rgb) # Process image with
MediaPipe

        if results.multi_hand_landmarks:
            landmarks = results.multi_hand_landmarks[0].landmark # Get
hand's landmark
                                                                 #
returns a list of 21 landmark points for a single hand detected
                                                                 #
Each landmark has an x, y, and z coordinate
            keypoints = []
            for landmark in landmarks:
                keypoints.append(landmark.x)  # x-coord
                keypoints.append(landmark.y)  # Y-coord
                keypoints.append(landmark.z)  # Z-coord (depth)
            return keypoints
    except Exception as e:
        print(f"Error processing image: {e}")
    return None

    # Expected output is a list of 63 values (21 landmarks x 3
coordinates) for an image/hand
```

Dataset processing function `process_dataset`

```
def process_dataset(data_dir, augment=False):
    """
    Process images from a dataset.
    For training: data_dir should have subfolders for each class.
    """
    images = []
    labels = []
    total_images = 0
    skipped_images = 0

    start_time = time.time()

    for label in sorted(os.listdir(data_dir)):
        class_folder = os.path.join(data_dir, label)
        if os.path.isdir(class_folder):
            print(f"Processing images for class '{label}'")
            for img_name in os.listdir(class_folder):
                img_path = os.path.join(class_folder, img_name)
                image = cv.imread(img_path)
                total_images += 1
                if image is None:
                    skipped_images +=1
                    continue

                # Extract keypoints from the original image
                keypoints = extract_keypoints(image)
                if keypoints:
                    images.append(keypoints)
                    labels.append(label)

                    # If augment is True, also flip the image
horizontally and extract again
                    if augment:
                        flipped_img = cv.flip(image, 1)  # Flip
horizontally
                        flipped_keypoints =
extract_keypoints(flipped_img)
                        if flipped_keypoints:
                            images.append(flipped_keypoints)
                            labels.append(label)

    duration = time.time() - start_time
    print(f"\nDataset processed in {duration:.2f} seconds")
    print(f"Total valid samples: {len(images)}")
    print(f"Skipped or unreadable images: {skipped_images}")

    # Convert lists to numpy arrays
    images = np.array(images)
    labels = np.array(labels)

    print(f"Total images processed (including augmentation if used):
{len(images)}")
```

```
    # Encode labels as integers
    label_encoder = LabelEncoder()
    labels = label_encoder.fit_transform(labels)

    return images, labels, label_encoder
```

Define dataset path and process images

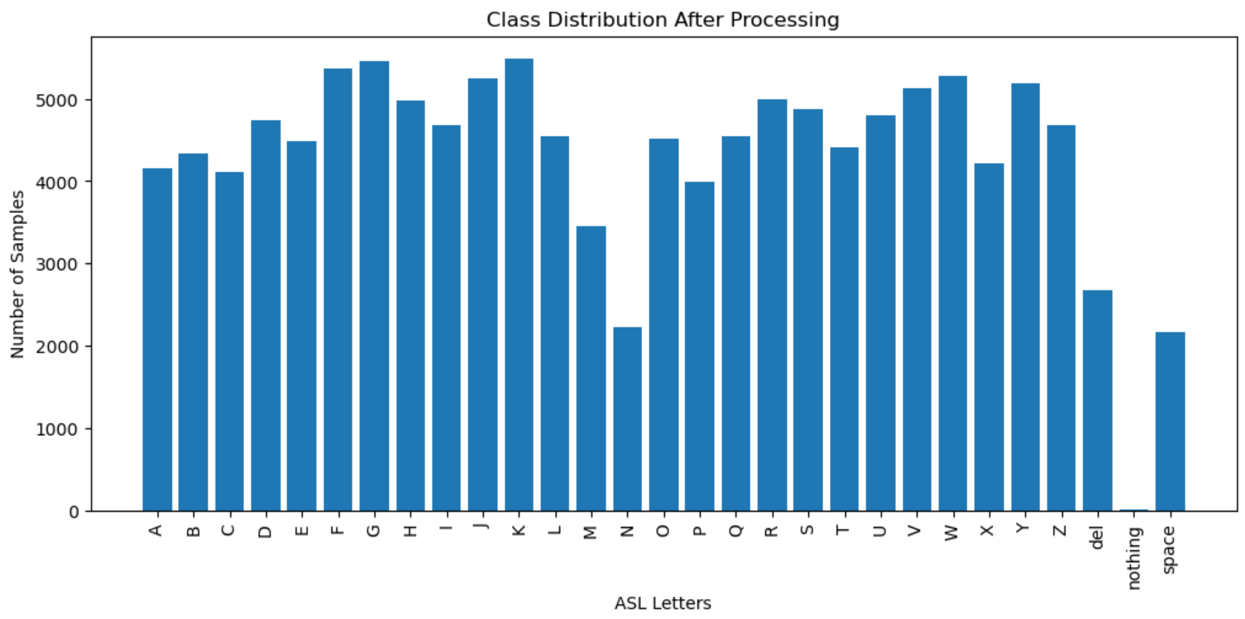
```
base_data_dir =
"/Users/carlopisacane/Desktop/SU/Honors/HonorsThesis/data"
train_dir = os.path.join(base_data_dir, "asl_alphabet_train")

# Process training data with augmentation enabled
X_train, y_train, le_train = process_dataset(train_dir, augment=True)
```

```
Processing images for class 'A'

W0000 00:00:1743573976.708578 8171416
landmark_projection_calculator.cc:186] Using NORM_RECT without
IMAGE_DIMENSIONS is only supported for the square ROI. Provide
IMAGE_DIMENSIONS or use PROJECTION_MATRIX.

Processing images for class 'B'
Processing images for class 'C'
Processing images for class 'D'
Processing images for class 'E'
Processing images for class 'F'
Processing images for class 'G'
Processing images for class 'H'
Processing images for class 'I'
Processing images for class 'J'
Processing images for class 'K'
Processing images for class 'L'
Processing images for class 'M'
Processing images for class 'N'
Processing images for class 'O'
Processing images for class 'P'
Processing images for class 'Q'
Processing images for class 'R'
Processing images for class 'S'
Processing images for class 'T'
Processing images for class 'U'
Processing images for class 'V'
Processing images for class 'W'
Processing images for class 'X'
Processing images for class 'Y'
Processing images for class 'Z'
Processing images for class 'del'
Processing images for class 'nothing'
Processing images for class 'space'
```

```
Dataset processed in 3773.73 seconds
Total valid samples: 124723
Skipped or unreadable images: 0
Total images processed (including augmentation if used): 124723
```

Stats summary

```
print(f"\nProcessed dataset shape: {X_train.shape}")
print(f"Number of labels: {len(y_train)}")
print(f"Number of classes: {len(le_train.classes_)}")

# Visualize class distribution
label_counts = Counter(le_train.inverse_transform(y_train))
plt.figure(figsize=(12, 5))
plt.bar(label_counts.keys(), label_counts.values())
plt.xticks(rotation=90)
plt.title("Class Distribution After Processing")
plt.xlabel("ASL Letters")
plt.ylabel("Number of Samples")
plt.show()


Processed dataset shape: (124723, 63)
Number of labels: 124723
Number of classes: 29
```



```
with open("processed_train.pkl", "wb") as f:
    pickle.dump((X_train, y_train, le_train), f)
```

```
print("Training data saved to 'processed_train.pkl'")

Training data saved to 'processed_train.pkl'
```

Multilayer perceptron (MLP) model, in `train_model_mlp.py`

Load the preprocessed ASL data

```
def load_data():
    """
    Loads the processed ASL training data:
    - X: Features
    - y: Labels (integers)
    - label_encoder: Maps labels to integers
    """
    with open("processed_train.pkl", "rb") as f:
        X, y, label_encoder = pickle.load(f)
    return X, y, label_encoder

# Load data
X, y, label_encoder = load_data()
```

Split the data into train and validation sets

```
X_train, X_val, y_train, y_val = train_test_split(
    X, y, test_size=0.2, random_state=42
)
print("Training samples:", X_train.shape[0])
print("Validation samples:", X_val.shape[0])

Training samples: 99778
Validation samples: 24945
```

Build a simple **MLP** model

```
model = Sequential([
    # First Dense layer with input dimension
    Dense(128, input_dim=X_train.shape[1], activation='relu'),
    # Dropout layer to reduce overfitting
    Dropout(0.5),
    # Second Dense layer
    Dense(64, activation='relu'),
    # Another Dropout layer
    Dropout(0.5),
    # Final output layer with 'softmax' activation
    Dense(len(le_train.classes_), activation='softmax')
])

/opt/anaconda3/lib/python3.12/site-packages/keras/src/layers/core/
dense.py:87: UserWarning: Do not pass an `input_shape`/`input_dim`
argument to a layer. When using Sequential models, prefer using an
`Input(shape)` object as the first layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer,
**kwargs)
```

Compile the model with optimizer, loss, and metrics

```
model.compile(
    optimizer='adam',
    loss='sparse_categorical_crossentropy',
    metrics=['accuracy']
)
```

View summary of model architecture

```
model.summary()

Model: "sequential"

┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━
━━━━━┓
┃ Layer (type)                     ┃ Output Shape            ┃
Param # ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━
━━━━━┩
│ dense (Dense)                    │ (None, 128)             │
8,192 │
├──────────────────────────────────┼─────────────────────────┼──────────
─────┤
│ dropout (Dropout)                │ (None, 128)             │
0 │
├──────────────────────────────────┼─────────────────────────┼──────────
─────┤
│ dense_1 (Dense)                  │ (None, 64)              │
8,256 │
├──────────────────────────────────┼─────────────────────────┼──────────
─────┤
│ dropout_1 (Dropout)              │ (None, 64)              │
0 │
├──────────────────────────────────┼─────────────────────────┼──────────
─────┤
│ dense_2 (Dense)                  │ (None, 29)              │
1,885 │
└──────────────────────────────────┴─────────────────────────┴──────────
─────┘
```

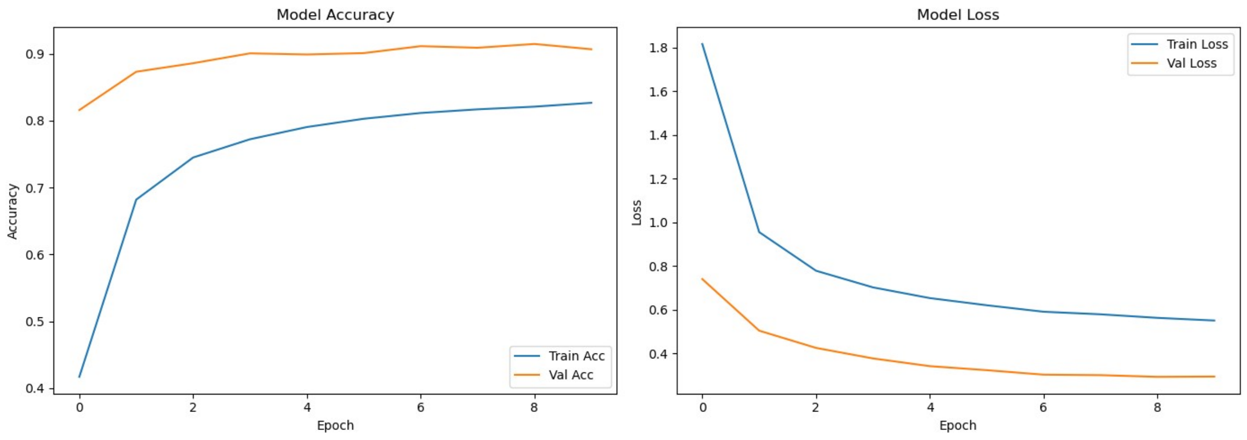
```
 Total params: 18,333 (71.61 KB)

 Trainable params: 18,333 (71.61 KB)

 Non-trainable params: 0 (0.00 B)
```

Train the model using the training data

```
history = model.fit(
    X_train, y_train,
    epochs=10,
    validation_data=(X_val, y_val)
)

Epoch 1/10
3119/3119 ━━━━━━━━━━━━━━━━━━━━ 3s 791us/step - accuracy: 0.2459 -
loss: 2.4812 - val_accuracy: 0.8158 - val_loss: 0.7401
Epoch 2/10
3119/3119 ━━━━━━━━━━━━━━━━━━━━ 2s 708us/step - accuracy: 0.6578 -
loss: 1.0208 - val_accuracy: 0.8731 - val_loss: 0.5040
Epoch 3/10
3119/3119 ━━━━━━━━━━━━━━━━━━━━ 2s 721us/step - accuracy: 0.7327 -
loss: 0.8095 - val_accuracy: 0.8859 - val_loss: 0.4250
Epoch 4/10
3119/3119 ━━━━━━━━━━━━━━━━━━━━ 2s 706us/step - accuracy: 0.7672 -
loss: 0.7261 - val_accuracy: 0.9007 - val_loss: 0.3770
Epoch 5/10
3119/3119 ━━━━━━━━━━━━━━━━━━━━ 2s 702us/step - accuracy: 0.7864 -
loss: 0.6639 - val_accuracy: 0.8989 - val_loss: 0.3416
Epoch 6/10
3119/3119 ━━━━━━━━━━━━━━━━━━━━ 2s 717us/step - accuracy: 0.7988 -
loss: 0.6269 - val_accuracy: 0.9010 - val_loss: 0.3232
Epoch 7/10
3119/3119 ━━━━━━━━━━━━━━━━━━━━ 2s 711us/step - accuracy: 0.8078 -
loss: 0.5976 - val_accuracy: 0.9114 - val_loss: 0.3026
Epoch 8/10
3119/3119 ━━━━━━━━━━━━━━━━━━━━ 2s 704us/step - accuracy: 0.8163 -
loss: 0.5820 - val_accuracy: 0.9090 - val_loss: 0.3004
Epoch 9/10
3119/3119 ━━━━━━━━━━━━━━━━━━━━ 2s 717us/step - accuracy: 0.8191 -
loss: 0.5613 - val_accuracy: 0.9147 - val_loss: 0.2927
Epoch 10/10
3119/3119 ━━━━━━━━━━━━━━━━━━━━ 2s 707us/step - accuracy: 0.8266 -
loss: 0.5515 - val_accuracy: 0.9068 - val_loss: 0.2940
```

Plot training performance

```
# Plot accuracy and loss
def plot_training_history(history):
    plt.figure(figsize=(14, 5))
    # Accuracy
    plt.subplot(1, 2, 1)
    plt.plot(history.history['accuracy'], label='Train Acc')
    plt.plot(history.history['val_accuracy'], label='Val Acc')
    plt.title('Model Accuracy')
    plt.xlabel('Epoch')
    plt.ylabel('Accuracy')
    plt.legend()

    # Loss
    plt.subplot(1, 2, 2)
    plt.plot(history.history['loss'], label='Train Loss')
    plt.plot(history.history['val_loss'], label='Val Loss')
    plt.title('Model Loss')
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.legend()

    plt.tight_layout()
    plt.show()

plot_training_history(history)
```



Save the trained model to disk

```
model.save('model/asl_mlp_model.h5')
print("Model saved to 'model/asl_mlp_model.h5'")

WARNING:absl:You are saving your model as an HDF5 file via
`model.save()` or `keras.saving.save_model(model)`. This file format
is considered legacy. We recommend using instead the native Keras
format, e.g. `model.save('my_model.keras')` or
`keras.saving.save_model(model, 'my_model.keras')`.

Model saved to 'model/asl_mlp_model.h5'
```

Evaluate the model on the validation set

```
test_loss, test_accuracy = model.evaluate(X_val, y_val, verbose=0)
print(f"Validation Accuracy: {test_accuracy * 100:.2f}%")
```

```
Validation Accuracy: 90.68%
```

Graph Convolutional Network (GCN) model, in `train_model_gcn.py`

Load the processed data from `processed_train.pkl`

```
with open("processed_train.pkl", "rb") as f:
    X, y, label_encoder = pickle.load(f)

# Reshape X: (N, 63) -> (N, 21, 3)
# Each of the 21 landmarks has 3 coordinates: (x, y, z).
X = X.reshape(-1, 21, 3)
num_samples = X.shape[0]
print("Total samples:", num_samples)
```

```
Total samples: 124723
```

View label classes

```
num_classes = len(label_encoder.classes_)
print("Number of classes:", num_classes)
print("Classes:", label_encoder.classes_)
```

```
Number of classes: 29
Classes: ['A' 'B' 'C' 'D' 'E' 'F' 'G' 'H' 'I' 'J' 'K' 'L' 'M' 'N' 'O'
'P' 'Q' 'R'
 'S' 'T' 'U' 'V' 'W' 'X' 'Y' 'Z' 'del' 'nothing' 'space']
```

Split into train/validation sets

```
X_train, X_val, y_train, y_val = train_test_split(
    X,
    y,
    test_size=0.2,
    random_state=42,
    stratify=y
)
print(f"Training samples: {X_train.shape[0]}, Validation samples:
{X_val.shape[0]}")
```

```
Training samples: 99778, Validation samples: 24945
```

Build the hand landmark graph (Adjacency Matrix)

```
# Here we define edges according to MediaPipe hand connections
num_nodes = 21
A = np.zeros((num_nodes, num_nodes), dtype=np.float32)
edges = [
    (0, 1), (1, 2), (2, 3), (3, 4),       # Thumb
    (0, 5), (5, 6), (6, 7), (7, 8),       # Index finger
    (0, 9), (9, 10), (10, 11), (11, 12),  # Middle finger
    (0, 13), (13, 14), (14, 15), (15, 16),# Ring finger
    (0, 17), (17, 18), (18, 19), (19, 20)# Pinky finger
]
for i, j in edges:
    A[i, j] = 1
    A[j, i] = 1

# Add self-loops for each node
for i in range(num_nodes):
    A[i, i] = 1
```

Compute the normalize the Adjacency Matrix

```
#    Ĥ = D^(-1/2) * A * D^(-1/2)
D = np.diag(np.sum(A, axis=1))
D_inv_sqrt = np.linalg.inv(np.sqrt(D))
A_norm = D_inv_sqrt @ A @ D_inv_sqrt  # shape: (21, 21)
A_norm_tf = tf.constant(A_norm, dtype=tf.float32)
```

Define a custom **GCN** layer

```
class GCNLayer(tf.keras.layers.Layer):
    def __init__(self, output_dim, activation=None, **kwargs):
        super(GCNLayer, self).__init__(**kwargs)
        self.output_dim = output_dim
        self.activation = tf.keras.activations.get(activation)

    def build(self, input_shape):
        # input_shape: (batch_size, num_nodes, input_dim)
        input_dim = input_shape[-1]
        self.w = self.add_weight(
            shape=(input_dim, self.output_dim),
            initializer='glorot_uniform',
            trainable=True,
            name='w'
        )
        super(GCNLayer, self).build(input_shape)

    def call(self, inputs, adj):
        # inputs: (batch_size, num_nodes, input_dim)
        # Multiply inputs by the weight matrix
        x = tf.matmul(inputs, self.w)  # => (batch_size, num_nodes,
output_dim)
        # Propagate through the graph
        x = tf.matmul(adj, x)  # => (batch_size, num_nodes,
output_dim)
        if self.activation is not None:
            x = self.activation(x)
        return x
```

Build the **GCN** model architecture

```
inputs = tf.keras.Input(shape=(num_nodes, 3))          # shape: (21,
3)
x = GCNLayer(64, activation='relu')(inputs, A_norm_tf) # First GCN
layer
x = GCNLayer(64, activation='relu')(x, A_norm_tf)      # Second GCN
layer
x = tf.keras.layers.Flatten()(x)
x = tf.keras.layers.Dense(128, activation='relu')(x)
x = tf.keras.layers.Dropout(0.5)(x)
outputs = tf.keras.layers.Dense(num_classes, activation='softmax')(x)

model = tf.keras.Model(inputs=inputs, outputs=outputs)
```

Compile the model

```
model.compile(
    optimizer='adam',
    loss='sparse_categorical_crossentropy',
    metrics=['accuracy']
)

model.summary()

Model: "functional_1"

┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━
━━━━━┓
┃ Layer (type)                     ┃ Output Shape            ┃
Param # ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━
━━━━━┩
│ input_layer_1 (InputLayer)       │ (None, 21, 3)           │
0 │
├──────────────────────────────────┼─────────────────────────┼──────────
─────┤
│ gcn_layer (GCNLayer)             │ (None, 21, 64)          │
192 │
├──────────────────────────────────┼─────────────────────────┼──────────
─────┤
```

```
│ gcn_layer_1 (GCNLayer)           │ (None, 21, 64)          │
4,096 │
├──────────────────────────────────┼─────────────────────────┼──────────
──────┤
│ flatten (Flatten)                │ (None, 1344)            │
0 │
├──────────────────────────────────┼─────────────────────────┼──────────
──────┤
│ dense_3 (Dense)                  │ (None, 128)             │
172,160 │
├──────────────────────────────────┼─────────────────────────┼──────────
──────┤
│ dropout_2 (Dropout)              │ (None, 128)             │
0 │
├──────────────────────────────────┼─────────────────────────┼──────────
──────┤
│ dense_4 (Dense)                  │ (None, 29)              │
3,741 │
└──────────────────────────────────┴─────────────────────────┴──────────
──────┘

 Total params: 180,189 (703.86 KB)

 Trainable params: 180,189 (703.86 KB)

 Non-trainable params: 0 (0.00 B)
```

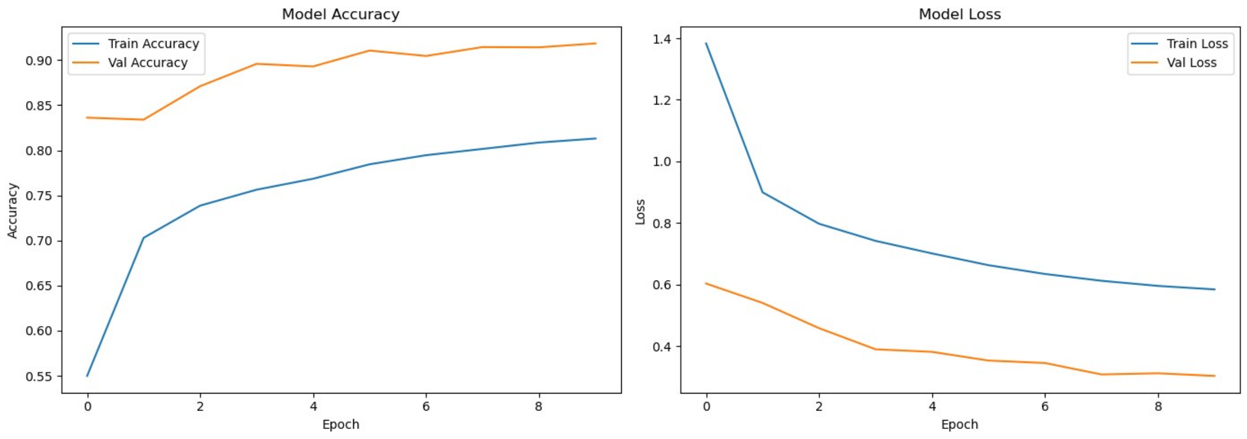
Train the **GCN** model

```
epochs = 10
history = model.fit(
    X_train, y_train,
    epochs=epochs,
    validation_data=(X_val, y_val)
)

Epoch 1/10
3119/3119 ━━━━━━━━━━━━━━━━━━━━ 9s 3ms/step - accuracy: 0.3956 - loss:
1.9346 - val_accuracy: 0.8362 - val_loss: 0.6032
Epoch 2/10
3119/3119 ━━━━━━━━━━━━━━━━━━━━ 9s 3ms/step - accuracy: 0.6926 - loss:
0.9358 - val_accuracy: 0.8340 - val_loss: 0.5403
Epoch 3/10
3119/3119 ━━━━━━━━━━━━━━━━━━━━ 9s 3ms/step - accuracy: 0.7315 - loss:
0.8117 - val_accuracy: 0.8710 - val_loss: 0.4582
Epoch 4/10
3119/3119 ━━━━━━━━━━━━━━━━━━━━ 9s 3ms/step - accuracy: 0.7528 - loss:
0.7540 - val_accuracy: 0.8959 - val_loss: 0.3897
Epoch 5/10
3119/3119 ━━━━━━━━━━━━━━━━━━━━ 9s 3ms/step - accuracy: 0.7650 - loss:
```

```
0.7180 - val_accuracy: 0.8930 - val_loss: 0.3813
Epoch 6/10
3119/3119 ━━━━━━━━━━━━━━━━━━━━ 9s 3ms/step - accuracy: 0.7788 - loss:
0.6768 - val_accuracy: 0.9106 - val_loss: 0.3531
Epoch 7/10
3119/3119 ━━━━━━━━━━━━━━━━━━━━ 9s 3ms/step - accuracy: 0.7926 - loss:
0.6391 - val_accuracy: 0.9046 - val_loss: 0.3451
Epoch 8/10
3119/3119 ━━━━━━━━━━━━━━━━━━━━ 9s 3ms/step - accuracy: 0.7973 - loss:
0.6242 - val_accuracy: 0.9145 - val_loss: 0.3080
Epoch 9/10
3119/3119 ━━━━━━━━━━━━━━━━━━━━ 9s 3ms/step - accuracy: 0.8110 - loss:
0.5835 - val_accuracy: 0.9142 - val_loss: 0.3117
Epoch 10/10
3119/3119 ━━━━━━━━━━━━━━━━━━━━ 8s 3ms/step - accuracy: 0.8119 - loss:
0.5851 - val_accuracy: 0.9185 - val_loss: 0.3032
```

Plot training accuracy and loss

```python
def plot_training_history(history):
    plt.figure(figsize=(14, 5))

    plt.subplot(1, 2, 1)
    plt.plot(history.history['accuracy'], label='Train Accuracy')
    plt.plot(history.history['val_accuracy'], label='Val Accuracy')
    plt.title('Model Accuracy')
    plt.xlabel('Epoch')
    plt.ylabel('Accuracy')
    plt.legend()

    plt.subplot(1, 2, 2)
    plt.plot(history.history['loss'], label='Train Loss')
    plt.plot(history.history['val_loss'], label='Val Loss')
    plt.title('Model Loss')
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.legend()

    plt.tight_layout()
    plt.show()

plot_training_history(history)
```



Evaluate the model on the validation set

```
val_loss, val_accuracy = model.evaluate(X_val, y_val, verbose=0)
print(f"Validation Accuracy: {val_accuracy * 100:.2f}%")

Validation Accuracy: 91.85%
```

Save trained **GCN** model

```
os.makedirs("model", exist_ok=True)
model.save("model/asl_gcn_model.h5")
print("GCN model training complete. Saved to
'model/asl_gcn_model.h5'.")

WARNING:absl:You are saving your model as an HDF5 file via
`model.save()` or `keras.saving.save_model(model)`. This file format
is considered legacy. We recommend using instead the native Keras
format, e.g. `model.save('my_model.keras')` or
`keras.saving.save_model(model, 'my_model.keras')`.

GCN model training complete. Saved to 'model/asl_gcn_model.h5'.
```

Convolutional Neural Network (CNN) model, in `train_model_cnn.py`

Load the processed data from `processed_train.pkl`

```
def load_data():
    """
    Load processed keypoint data from 'processed_train.pkl'.
    That file should contain (X, y, label_encoder),
    where X is shape (num_samples, 63) => (21 landmarks x 3 coords).
    """
    with open("processed_train.pkl", "rb") as f:
        X, y, label_encoder = pickle.load(f)
    return X, y, label_encoder

# Load the data (Mediapipe keypoints) from 'processed_train.pkl'
X, y, train_le = load_data()
print("Original X shape:", X.shape)  # (num_samples, 63)

Original X shape: (124723, 63)
```

Reshape X: (N, 63) -> (N, 21, 3), then expand dims -> (N, 21, 3, 1)

```
X = X.reshape(-1, 21, 3)
X = np.expand_dims(X, axis=-1)  # shape: (N, 21, 3, 1)
print("Reshaped X for CNN:", X.shape)

Reshaped X for CNN: (124723, 21, 3, 1)
```

Check labels & classes

```
num_classes = len(train_le.classes_)
print("Number of classes:", num_classes)
print("Classes:", train_le.classes_)

Number of classes: 29
Classes: ['A' 'B' 'C' 'D' 'E' 'F' 'G' 'H' 'I' 'J' 'K' 'L' 'M' 'N' 'O'
'P' 'Q' 'R'
 'S' 'T' 'U' 'V' 'W' 'X' 'Y' 'Z' 'del' 'nothing' 'space']
```

Train/validation split

```
X_train, X_val, y_train, y_val = train_test_split(
    X,
    y,
    test_size=0.2,
    random_state=42,
    stratify=y
)
print("Training samples:", X_train.shape[0])
print("Validation samples:", X_val.shape[0])

Training samples: 99778
Validation samples: 24945
```

Build a **CNN** that won't shrink the 3-dimension to 0

```
# Use (3,3) conv with padding='same' and
# MaxPooling2D with pool_size=(2,1) so only the "landmark dimension"
is halved.

model = Sequential([
    # Convolution #1: preserve the shape with same padding
    Conv2D(32, (3, 3), activation='relu', padding='same',
           input_shape=(21, 3, 1)),
    # Pool over the "landmark dimension" only
    MaxPooling2D(pool_size=(2, 1)),

    # Convolution #2: again use same padding
    Conv2D(64, (3, 3), activation='relu', padding='same'),
    MaxPooling2D(pool_size=(2, 1)),

    # Conv2D(128, (3, 3), activation='relu', padding='same'),
    # MaxPooling2D(pool_size=(2, 1)),

    Flatten(),
    Dense(128, activation='relu'),
    Dropout(0.5),
    Dense(num_classes, activation='softmax')
])
```

```
/opt/anaconda3/lib/python3.12/site-packages/keras/src/layers/
convolutional/base_conv.py:107: UserWarning: Do not pass an
`input_shape`/`input_dim` argument to a layer. When using Sequential
models, prefer using an `Input(shape)` object as the first layer in
the model instead.
  super().__init__(activity_regularizer=activity_regularizer,
**kwargs)
```

Compile the model

```
model.compile(
    optimizer=Adam(learning_rate=1e-3),
    loss='sparse_categorical_crossentropy',
    metrics=['accuracy']
)

model.summary()
```

```
Model: "sequential_1"

┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━
━━━━━┓
┃ Layer (type)                    ┃ Output Shape           ┃
Param # ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━
━━━━━┩
│ conv2d (Conv2D)                 │ (None, 21, 3, 32)      │
320 │
├─────────────────────────────────┼────────────────────────┼──────────
─────┤
│ max_pooling2d (MaxPooling2D)    │ (None, 10, 3, 32)      │
```

```
0 │
├─────────────────────────────────┼─────────────────────────┼──────────
─────┤
│ conv2d_1 (Conv2D)               │ (None, 10, 3, 64)       │
18,496 │
├─────────────────────────────────┼─────────────────────────┼──────────
─────┤
│ max_pooling2d_1 (MaxPooling2D)  │ (None, 5, 3, 64)        │
0 │
├─────────────────────────────────┼─────────────────────────┼──────────
─────┤
│ flatten_1 (Flatten)             │ (None, 960)             │
0 │
├─────────────────────────────────┼─────────────────────────┼──────────
─────┤
│ dense_5 (Dense)                 │ (None, 128)             │
123,008 │
├─────────────────────────────────┼─────────────────────────┼──────────
─────┤
│ dropout_3 (Dropout)             │ (None, 128)             │
0 │
├─────────────────────────────────┼─────────────────────────┼──────────
─────┤
│ dense_6 (Dense)                 │ (None, 29)              │
3,741 │
└─────────────────────────────────┴─────────────────────────┴──────────
─────┘

 Total params: 145,565 (568.61 KB)

 Trainable params: 145,565 (568.61 KB)

 Non-trainable params: 0 (0.00 B)
```

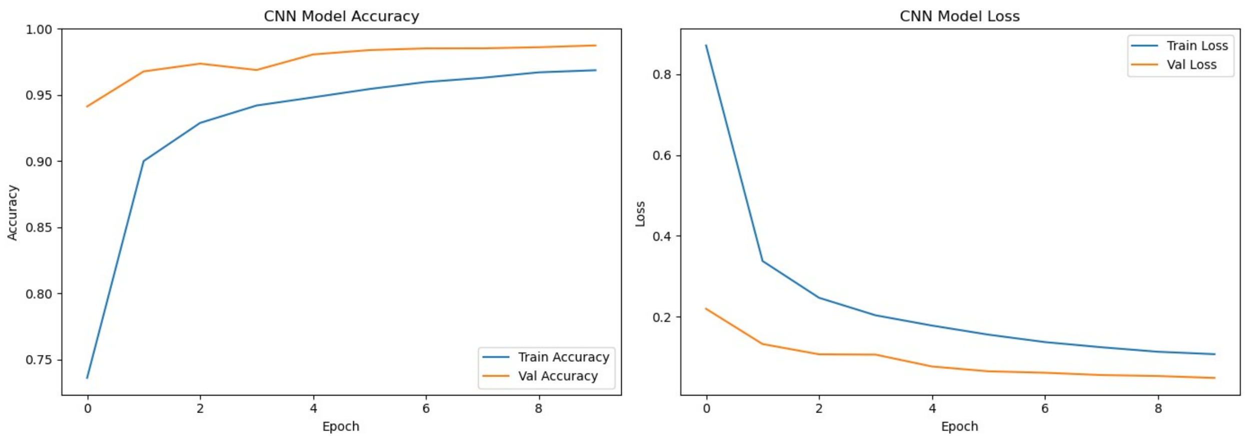
Train

```
epochs = 10
history = model.fit(
    X_train, y_train,
    epochs=epochs,
    validation_data=(X_val, y_val),
    batch_size=32
)

Epoch 1/10
3119/3119 ━━━━━━━━━━━━━━━━━━━━ 11s 3ms/step - accuracy: 0.5378 - loss:
1.5466 - val_accuracy: 0.9413 - val_loss: 0.2196
Epoch 2/10
3119/3119 ━━━━━━━━━━━━━━━━━━━━ 12s 4ms/step - accuracy: 0.8887 - loss:
0.3746 - val_accuracy: 0.9678 - val_loss: 0.1325
```

```
Epoch 3/10
3119/3119 ━━━━━━━━━━━━━━━━━━━━ 13s 4ms/step - accuracy: 0.9252 - loss: 
0.2628 - val_accuracy: 0.9737 - val_loss: 0.1072
Epoch 4/10
3119/3119 ━━━━━━━━━━━━━━━━━━━━ 12s 4ms/step - accuracy: 0.9410 - loss: 
0.2130 - val_accuracy: 0.9689 - val_loss: 0.1064
Epoch 5/10
3119/3119 ━━━━━━━━━━━━━━━━━━━━ 13s 4ms/step - accuracy: 0.9469 - loss: 
0.1819 - val_accuracy: 0.9807 - val_loss: 0.0770
Epoch 6/10
3119/3119 ━━━━━━━━━━━━━━━━━━━━ 13s 4ms/step - accuracy: 0.9535 - loss: 
0.1587 - val_accuracy: 0.9840 - val_loss: 0.0651
Epoch 7/10
3119/3119 ━━━━━━━━━━━━━━━━━━━━ 13s 4ms/step - accuracy: 0.9591 - loss: 
0.1383 - val_accuracy: 0.9852 - val_loss: 0.0615
Epoch 8/10
3119/3119 ━━━━━━━━━━━━━━━━━━━━ 13s 4ms/step - accuracy: 0.9625 - loss: 
0.1262 - val_accuracy: 0.9853 - val_loss: 0.0559
Epoch 9/10
3119/3119 ━━━━━━━━━━━━━━━━━━━━ 12s 4ms/step - accuracy: 0.9662 - loss: 
0.1145 - val_accuracy: 0.9861 - val_loss: 0.0535
Epoch 10/10
3119/3119 ━━━━━━━━━━━━━━━━━━━━ 12s 4ms/step - accuracy: 0.9680 - loss: 
0.1067 - val_accuracy: 0.9875 - val_loss: 0.0488
```

Visualize Training Progress

```python
def plot_training_history(history):
    plt.figure(figsize=(14, 5))

    plt.subplot(1, 2, 1)
    plt.plot(history.history['accuracy'], label='Train Accuracy')
    plt.plot(history.history['val_accuracy'], label='Val Accuracy')
    plt.title('CNN Model Accuracy')
    plt.xlabel('Epoch')
    plt.ylabel('Accuracy')
    plt.legend()

    plt.subplot(1, 2, 2)
    plt.plot(history.history['loss'], label='Train Loss')
    plt.plot(history.history['val_loss'], label='Val Loss')
    plt.title('CNN Model Loss')
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.legend()

    plt.tight_layout()
    plt.show()

plot_training_history(history)
```



Evaluate

```
val_loss, val_accuracy = model.evaluate(X_val, y_val, verbose=0)
print(f"Validation Accuracy: {val_accuracy * 100:.2f}%")

Validation Accuracy: 98.75%
```

Save

```
os.makedirs("model", exist_ok=True)
model.save("model/asl_cnn_model.h5")
print("CNN model (using keypoints) saved to 'model/asl_cnn_model.h5'")

WARNING:absl:You are saving your model as an HDF5 file via
`model.save()` or `keras.saving.save_model(model)`. This file format
is considered legacy. We recommend using instead the native Keras
format, e.g. `model.save('my_model.keras')` or
`keras.saving.save_model(model, 'my_model.keras')`.

CNN model (using keypoints) saved to 'model/asl_cnn_model.h5'
```

Combined GCN + CNN model, in `train_model_combined.py`

Load the processed data from `processed_train.pkl`

```
with open("processed_train.pkl", "rb") as f:
    X, y, label_encoder = pickle.load(f)
```

Reshape X: (N, 63) -> (N, 21, 3), then expand dims -> (N, 21, 3, 1)

```
X = X.reshape((-1, 21, 3))
num_samples = X.shape[0]
print("Total samples:", num_samples)
```

```
Total samples: 124723
```

View label classes

```
num_classes = len(label_encoder.classes_)
print("Number of classes:", num_classes)
print("Classes:", label_encoder.classes_)
```

Train/validation split

```
X_train, X_val, y_train, y_val = train_test_split(
    X,
    y,
    test_size=0.2,
    random_state=42,
    stratify=y
)
print(f"Training samples: {X_train.shape[0]}")
print(f"Validation samples: {X_val.shape[0]}")
```

```
Training samples: 99778
Validation samples: 24945
```

Build the hand landmark graph (Adjacency Matrix)

```
num_nodes = 21
A = np.zeros((num_nodes, num_nodes), dtype=np.float32)
edges = [
    (0, 1), (1, 2), (2, 3), (3, 4),         # Thumb
    (0, 5), (5, 6), (6, 7), (7, 8),         # Index
    (0, 9), (9, 10), (10, 11), (11, 12),    # Middle
    (0, 13), (13, 14), (14, 15), (15, 16), # Ring
    (0, 17), (17, 18), (18, 19), (19, 20)  # Pinky
]
for i, j in edges:
    A[i, j] = 1
    A[j, i] = 1

for i in range(num_nodes):
    A[i, i] = 1
```

Compute the normalize the Adjacency Matrix

```
D = np.diag(np.sum(A, axis=1))
D_inv_sqrt = np.linalg.inv(np.sqrt(D))
A_norm = D_inv_sqrt @ A @ D_inv_sqrt
A_norm_tf = tf.constant(A_norm, dtype=tf.float32)
```

Define a custom **GCN** layer

```
class GCNLayer(tf.keras.layers.Layer):
    def __init__(self, output_dim, activation=None, **kwargs):
        super(GCNLayer, self).__init__(**kwargs)
        self.output_dim = output_dim
        self.activation = tf.keras.activations.get(activation)

    def build(self, input_shape):
        input_dim = input_shape[-1]
        self.w = self.add_weight(
            shape=(input_dim, self.output_dim),
            initializer='glorot_uniform',
            trainable=True,
            name='gcn_weight'
        )
        super(GCNLayer, self).build(input_shape)

    def call(self, inputs, adj):
        x = tf.matmul(inputs, self.w)   # shape: (batch, 21,
output_dim)
        x = tf.matmul(adj, x)           # graph propagation
        if self.activation is not None:
            x = self.activation(x)
        return x
```

Build the **combined** model that merges **GCN** + **CNN** branches, `build_combined_model` function

```
def build_combined_model(num_classes, adjacency_matrix):
    inputs = Input(shape=(21, 3), name='input_landmarks')

    # GCN branch
    gcn_out = GCNLayer(64, activation='relu')(inputs,
adjacency_matrix)
    gcn_out = GCNLayer(64, activation='relu')(gcn_out,
adjacency_matrix)
    gcn_out = Flatten()(gcn_out)
    gcn_out = Dense(64, activation='relu')(gcn_out)

    # CNN branch
    cnn_input = Reshape((21, 3, 1))(inputs)
    x = Conv2D(32, (3, 3), activation='relu', padding='same')
(cnn_input)
    x = MaxPooling2D(pool_size=(2, 1))(x)
    x = Conv2D(64, (3, 3), activation='relu', padding='same')(x)
    x = MaxPooling2D(pool_size=(2, 1))(x)
    x = Flatten()(x)
    x = Dense(64, activation='relu')(x)
    # Merge both branches
    merged = Concatenate()([gcn_out, x])
    merged = Dense(128, activation='relu')(merged)
    merged = Dropout(0.5)(merged)
    outputs = Dense(num_classes, activation='softmax')(merged)

    model = Model(inputs=inputs, outputs=outputs)
    model.compile(
        optimizer='adam',
        loss='sparse_categorical_crossentropy',
        metrics=['accuracy']
    )
    return model
```

Build the model

```
model = build_combined_model(num_classes, A_norm_tf)
model.summary()

Model: "functional_3"

┏━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━┳━━━━━━━━━━━━━━
━━━━━┓
┃ Layer (type)        ┃ Output Shape       ┃     Param # ┃ Connected to
┃
┡━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━╇━━━━━━━━━━━━━━
━━━━━┩
│ input_landmarks     │ (None, 21, 3)      │           0 │ -
│
│ (InputLayer)        │                    │             │
│
├─────────────────────┼────────────────────┼─────────────┼──────────────
─────┤
│ reshape (Reshape)   │ (None, 21, 3, 1)   │           0 │
input_landmarks[… │
├─────────────────────┼────────────────────┼─────────────┼──────────────
─────┤
│ conv2d_2 (Conv2D)   │ (None, 21, 3, 32)  │         320 │ reshape[0][0]
│
├─────────────────────┼────────────────────┼─────────────┼──────────────
─────┤
│ max_pooling2d_2     │ (None, 10, 3, 32)  │           0 │ conv2d_2[0]
[0]     │
│ (MaxPooling2D)      │                    │             │
│
├─────────────────────┼────────────────────┼─────────────┼──────────────
─────┤
│ gcn_layer_2         │ (None, 21, 64)     │         192 │
input_landmarks[… │
│ (GCNLayer)          │                    │             │
```

```
│
├───────────────────────┼────────────────────┼─────────────┼──────────────
──────┤
│ conv2d_3 (Conv2D)     │ (None, 10, 3, 64)  │      18,496 │
max_pooling2d_2[… │
├───────────────────────┼────────────────────┼─────────────┼──────────────
──────┤
│ gcn_layer_3           │ (None, 21, 64)     │       4,096 │
gcn_layer_2[0][0] │
│ (GCNLayer)            │                    │             │
│
├───────────────────────┼────────────────────┼─────────────┼──────────────
──────┤
│ max_pooling2d_3       │ (None, 5, 3, 64)   │           0 │ conv2d_3[0]
[0]     │
│ (MaxPooling2D)        │                    │             │
│
├───────────────────────┼────────────────────┼─────────────┼──────────────
──────┤
│ flatten_2 (Flatten) │ (None, 1344)       │           0 │
gcn_layer_3[0][0] │
├───────────────────────┼────────────────────┼─────────────┼──────────────
──────┤
│ flatten_3 (Flatten) │ (None, 960)        │           0 │
max_pooling2d_3[… │
├───────────────────────┼────────────────────┼─────────────┼──────────────
──────┤
│ dense_7 (Dense)       │ (None, 64)         │      86,080 │ flatten_2[0]
[0]    │
├───────────────────────┼────────────────────┼─────────────┼──────────────
──────┤
│ dense_8 (Dense)       │ (None, 64)         │      61,504 │ flatten_3[0]
[0]    │
├───────────────────────┼────────────────────┼─────────────┼──────────────
──────┤
│ concatenate           │ (None, 128)        │           0 │ dense_7[0]
[0],     │
│ (Concatenate)         │                    │             │ dense_8[0][0]
│
├───────────────────────┼────────────────────┼─────────────┼──────────────
──────┤
│ dense_9 (Dense)       │ (None, 128)        │      16,512 │
concatenate[0][0] │
├───────────────────────┼────────────────────┼─────────────┼──────────────
──────┤
│ dropout_4 (Dropout) │ (None, 128)        │           0 │ dense_9[0][0]
│
├───────────────────────┼────────────────────┼─────────────┼──────────────
──────┤
```

```
│ dense_10 (Dense)      │ (None, 29)          │       3,741 │ dropout_4[0]
[0]    │
└───────────────────────┴─────────────────────┴─────────────┴───────────────
──────┘

 Total params: 190,941 (745.86 KB)

 Trainable params: 190,941 (745.86 KB)

 Non-trainable params: 0 (0.00 B)
```

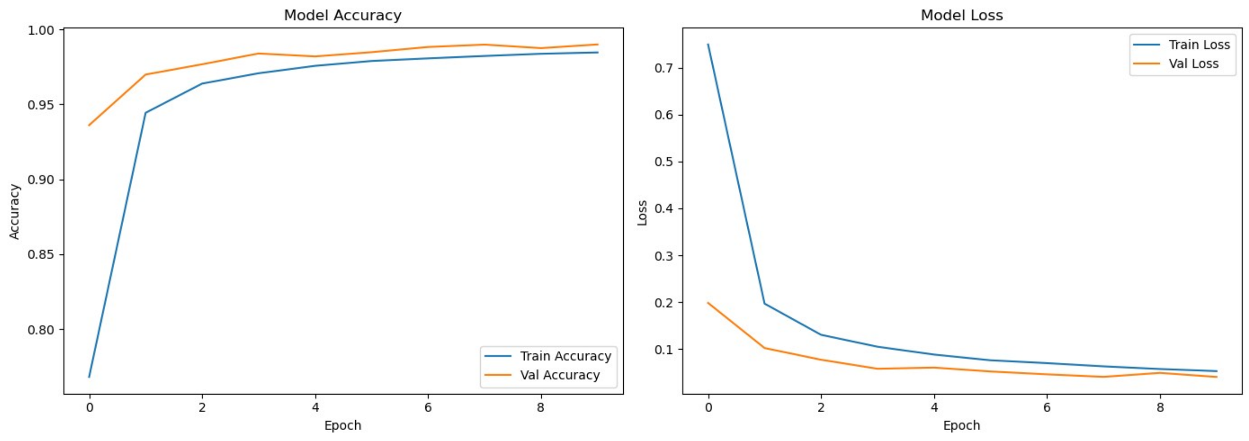
Train

```
epochs = 10
history = model.fit(
    X_train, y_train,
    validation_data=(X_val, y_val),
    epochs=epochs,
    batch_size=32
)

Epoch 1/10
3119/3119 ━━━━━━━━━━━━━━━━━━━━ 16s 5ms/step - accuracy: 0.5578 - loss:
1.4398 - val_accuracy: 0.9361 - val_loss: 0.1985
Epoch 2/10
3119/3119 ━━━━━━━━━━━━━━━━━━━━ 15s 5ms/step - accuracy: 0.9348 - loss:
0.2267 - val_accuracy: 0.9699 - val_loss: 0.1023
Epoch 3/10
3119/3119 ━━━━━━━━━━━━━━━━━━━━ 15s 5ms/step - accuracy: 0.9631 - loss:
0.1356 - val_accuracy: 0.9768 - val_loss: 0.0774
Epoch 4/10
3119/3119 ━━━━━━━━━━━━━━━━━━━━ 15s 5ms/step - accuracy: 0.9695 - loss:
0.1083 - val_accuracy: 0.9840 - val_loss: 0.0581
Epoch 5/10
3119/3119 ━━━━━━━━━━━━━━━━━━━━ 15s 5ms/step - accuracy: 0.9749 - loss:
0.0900 - val_accuracy: 0.9820 - val_loss: 0.0606
Epoch 6/10
3119/3119 ━━━━━━━━━━━━━━━━━━━━ 14s 5ms/step - accuracy: 0.9783 - loss:
0.0778 - val_accuracy: 0.9849 - val_loss: 0.0523
Epoch 7/10
3119/3119 ━━━━━━━━━━━━━━━━━━━━ 14s 5ms/step - accuracy: 0.9804 - loss:
0.0715 - val_accuracy: 0.9883 - val_loss: 0.0462
Epoch 8/10
3119/3119 ━━━━━━━━━━━━━━━━━━━━ 14s 4ms/step - accuracy: 0.9816 - loss:
0.0650 - val_accuracy: 0.9899 - val_loss: 0.0409
Epoch 9/10
3119/3119 ━━━━━━━━━━━━━━━━━━━━ 14s 4ms/step - accuracy: 0.9834 - loss:
0.0581 - val_accuracy: 0.9876 - val_loss: 0.0493
Epoch 10/10
```

```
3119/3119 ━━━━━━━━━━━━━━━━━━━━ 14s 4ms/step - accuracy: 0.9847 - loss:
0.0535 - val_accuracy: 0.9901 - val_loss: 0.0407
```

Visualize

```
def plot_training_history(history):
    plt.figure(figsize=(14, 5))

    plt.subplot(1, 2, 1)
    plt.plot(history.history['accuracy'], label='Train Accuracy')
    plt.plot(history.history['val_accuracy'], label='Val Accuracy')
    plt.title('Model Accuracy')
    plt.xlabel('Epoch')
    plt.ylabel('Accuracy')
    plt.legend()

    plt.subplot(1, 2, 2)
    plt.plot(history.history['loss'], label='Train Loss')
    plt.plot(history.history['val_loss'], label='Val Loss')
    plt.title('Model Loss')
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.legend()

    plt.tight_layout()
    plt.show()

plot_training_history(history)
```



Evaluate

```
val_loss, val_acc = model.evaluate(X_val, y_val, verbose=0)
print(f"Validation Accuracy: {val_acc * 100:.2f}%")

Validation Accuracy: 99.01%
```

Save

```
os.makedirs("model", exist_ok=True)
model.save("model/asl_combined_model.h5")
print("Saved combined GCN+CNN model to 'model/asl_combined_model.h5'")
```

```
WARNING:absl:You are saving your model as an HDF5 file via
`model.save()` or `keras.saving.save_model(model)`. This file format
is considered legacy. We recommend using instead the native Keras
format, e.g. `model.save('my_model.keras')` or
`keras.saving.save_model(model, 'my_model.keras')`.

Saved combined GCN+CNN model to 'model/asl_combined_model.h5'
```

Analysis and Results

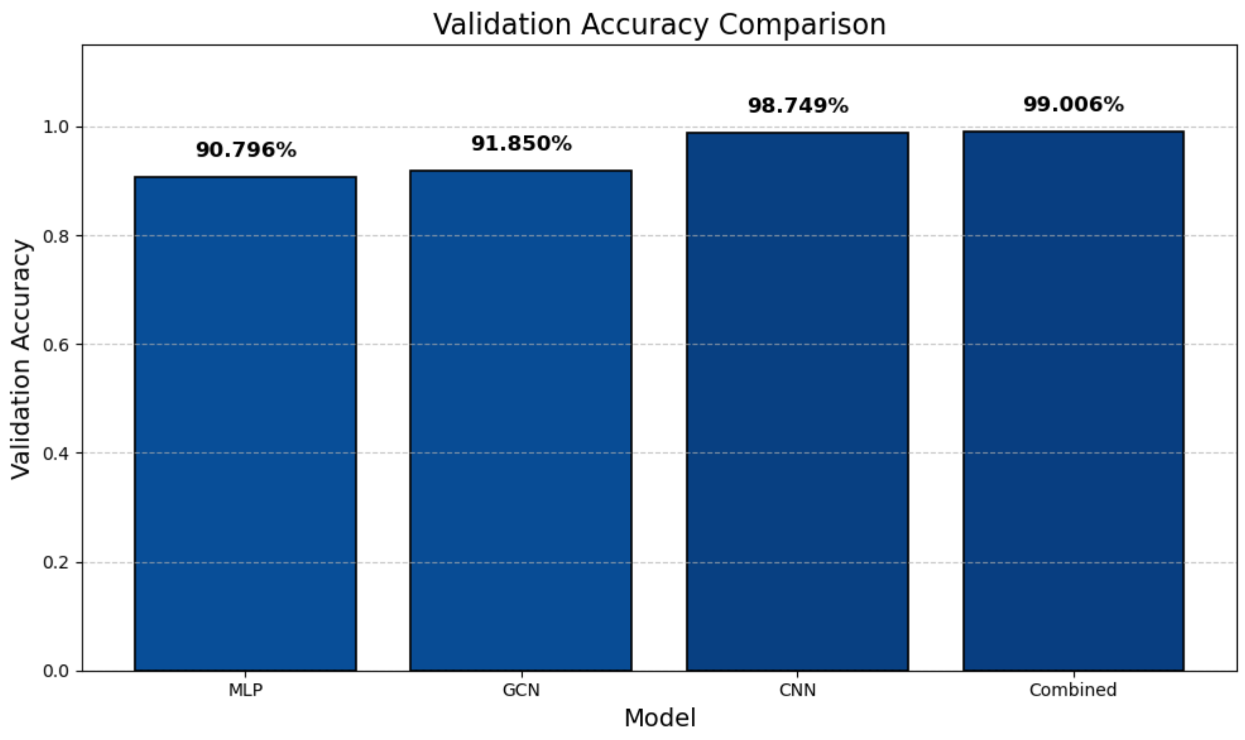
```
# 1. Define the Custom GCNLayer
class GCNLayer(tf.keras.layers.Layer):
    def __init__(self, output_dim, activation=None, **kwargs):
        super(GCNLayer, self).__init__(**kwargs)
        self.output_dim = output_dim
        self.activation = tf.keras.activations.get(activation)

    def build(self, input_shape):
        input_dim = input_shape[-1]
        self.w = self.add_weight(
            shape=(input_dim, self.output_dim),
            initializer='glorot_uniform',
            trainable=True,
            name='w'
        )
        super(GCNLayer, self).build(input_shape)

    def call(self, inputs, adj):
        # Multiply inputs by the weight matrix
        x = tf.matmul(inputs, self.w)
        # Propagate through the graph using the adjacency matrix
        x = tf.matmul(adj, x)
        if self.activation is not None:
            x = self.activation(x)
        return x

# Create a dictionary for custom objects
custom_objects = {'GCNLayer': GCNLayer}

# 2. Load Processed Data
with open("processed_train.pkl", "rb") as f:
    X, y, label_encoder = pickle.load(f)
```

```
# Use a common train/validation split (80/20, stratified for
consistency)
X_train_mlp, X_val_mlp, y_train, y_val = train_test_split(
    X, y, test_size=0.2, random_state=42, stratify=y
)

# 3. Prepare Validation Data for Each Model
# MLP: expects (N, 63)
val_data_mlp = X_val_mlp

# GCN & Combined: expects (N, 21, 3)
val_data_gcn = X_val_mlp.reshape(-1, 21, 3)
val_data_combined = val_data_gcn  # Same shape as GCN

# CNN: expects (N, 21, 3, 1)
val_data_cnn = X_val_mlp.reshape(-1, 21, 3)
val_data_cnn = np.expand_dims(val_data_cnn, axis=-1)

# 4. Define Model Information
models_info = {
    "MLP": {
        "file": "model/asl_mlp_model.h5",
        "input_data": val_data_mlp
    },
    "GCN": {
        "file": "model/asl_gcn_model.h5",
        "input_data": val_data_gcn
    },
    "CNN": {
        "file": "model/asl_cnn_model.h5",
        "input_data": val_data_cnn
    },
    "Combined": {
        "file": "model/asl_combined_model.h5",
        "input_data": val_data_combined
    }
}

# 5. Evaluate Each Model and Compute Metrics
model_results = {}

for model_name, info in models_info.items():
    print(f"Loading {model_name} model from {info['file']}...")
    model = load_model(info["file"], custom_objects=custom_objects,
compile=False)
    # Compile the model to ensure evaluation works properly
    model.compile(optimizer='adam',
loss='sparse_categorical_crossentropy', metrics=['accuracy'])

    # Evaluate model on the validation set
    val_loss, val_accuracy = model.evaluate(info["input_data"], y_val,
verbose=0)

    # Generate predictions and compute confusion matrix
    y_pred_prob = model.predict(info["input_data"], verbose=0)
    y_pred = np.argmax(y_pred_prob, axis=1)
    cm = confusion_matrix(y_val, y_pred)

    model_results[model_name] = {
        "val_accuracy": val_accuracy,
        "val_loss": val_loss,
        "confusion_matrix": cm
    }

    print(f"{model_name} - Validation Accuracy: {val_accuracy*100:.2f}
%")

# 6. Create a Summary DataFrame
summary_data = []
for model_name, metrics in model_results.items():
    summary_data.append({
        "Model": model_name,
        "Validation Accuracy": metrics["val_accuracy"],
        "Validation Loss": metrics["val_loss"]
    })
results_df = pd.DataFrame(summary_data)
print("\nModel Performance Summary:")
print(results_df)

# 7. Improved Validation Accuracy Comparison Chart with Lighter Colors
plt.figure(figsize=(10, 6))
cmap = plt.get_cmap("Blues")
colors = [cmap(0.25 + 0.7 * acc) for acc in results_df["Validation
Accuracy"]]

bars = plt.bar(results_df["Model"], results_df["Validation Accuracy"],
               color=colors, edgecolor='black', linewidth=1.2)

plt.xlabel("Model", fontsize=14)
plt.ylabel("Validation Accuracy", fontsize=14)
plt.title("Validation Accuracy Comparison", fontsize=16)
plt.ylim([0, 1.15])
plt.grid(axis='y', linestyle='--', alpha=0.7)

# Annotate each bar with its accuracy percentage (rounded to three
decimals)
for i, v in enumerate(results_df["Validation Accuracy"]):
    plt.text(i, v + 0.03, f"{(v*100):.3f}%", ha='center', va='bottom',
fontsize=12, fontweight='bold')
```

```
plt.tight_layout()
plt.show()

# 8. Prepare Data for the Table
results_df_table = results_df.copy()
results_df_table["Validation Accuracy"] =
(results_df_table["Validation Accuracy"] * 100).round(3).astype(str) +
'%'
results_df_table["Validation Loss"] = results_df_table["Validation
Loss"].round(3)

# 9. Create a Table for Model Performance Summary
fig, ax = plt.subplots(figsize=(8, 3))
ax.axis('tight')
ax.axis('off')
table = ax.table(cellText=results_df_table.values,
                 colLabels=results_df_table.columns,
                 cellLoc='center',
                 loc='center')

# Better spacing
table.auto_set_font_size(False)
table.set_fontsize(12)
table.scale(1.3, 1.5)

plt.title("Model Performance Summary", fontsize=16)
plt.show()

# 10. Define a Function to Plot Confusion Matrix
def plot_confusion_matrix(cm, classes, title):
    plt.figure(figsize=(14,12))
    plt.imshow(cm, interpolation='nearest', cmap=plt.cm.Blues)
    plt.title(title)
    plt.colorbar()
    tick_marks = np.arange(len(classes))
    plt.xticks(tick_marks, classes, rotation=45)
    plt.yticks(tick_marks, classes)

    thresh = cm.max() / 2.0
    for i in range(cm.shape[0]):
        for j in range(cm.shape[1]):
            plt.text(j, i, format(cm[i, j], 'd'),
                     horizontalalignment="center",
                     color="white" if cm[i, j] > thresh else "black")
    plt.tight_layout()
    plt.ylabel("True Label")
    plt.xlabel("Predicted Label")
    plt.show()

# 11. Plot Confusion Matrices for Each Model
```

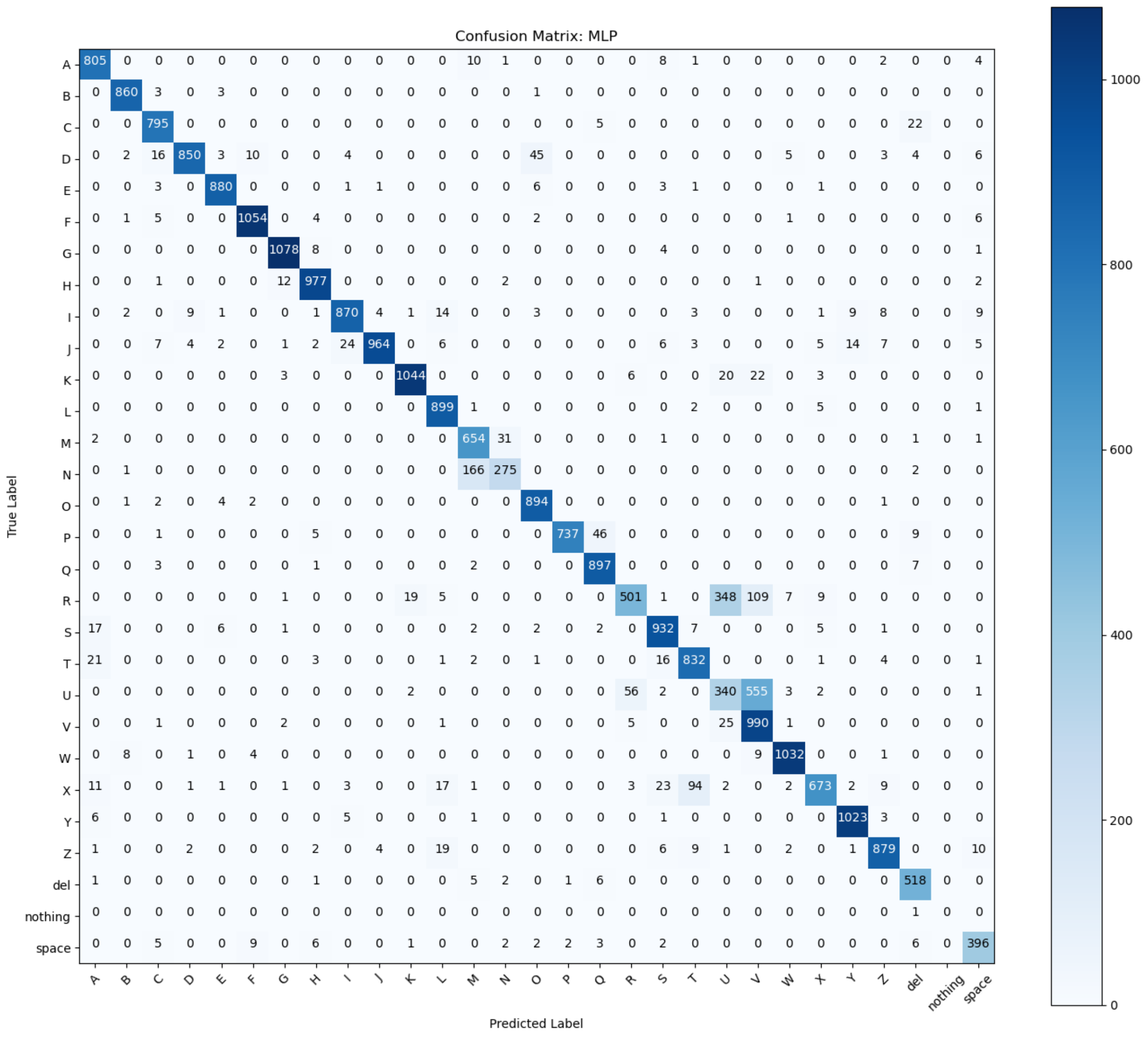
```
classes = label_encoder.classes_
for model_name, metrics in model_results.items():
    plot_confusion_matrix(metrics["confusion_matrix"], classes,
title=f"Confusion Matrix: {model_name}")
```

```
Loading MLP model from model/asl_mlp_model.h5...
MLP - Validation Accuracy: 90.80%
Loading GCN model from model/asl_gcn_model.h5...
GCN - Validation Accuracy: 91.85%
Loading CNN model from model/asl_cnn_model.h5...
CNN - Validation Accuracy: 98.75%
Loading Combined model from model/asl_combined_model.h5...
Combined - Validation Accuracy: 99.01%

Model Performance Summary:
      Model  Validation Accuracy  Validation Loss
0       MLP             0.907957         0.292693
1       GCN             0.918501         0.303174
2       CNN             0.987493         0.048753
3  Combined             0.990058         0.040724
```

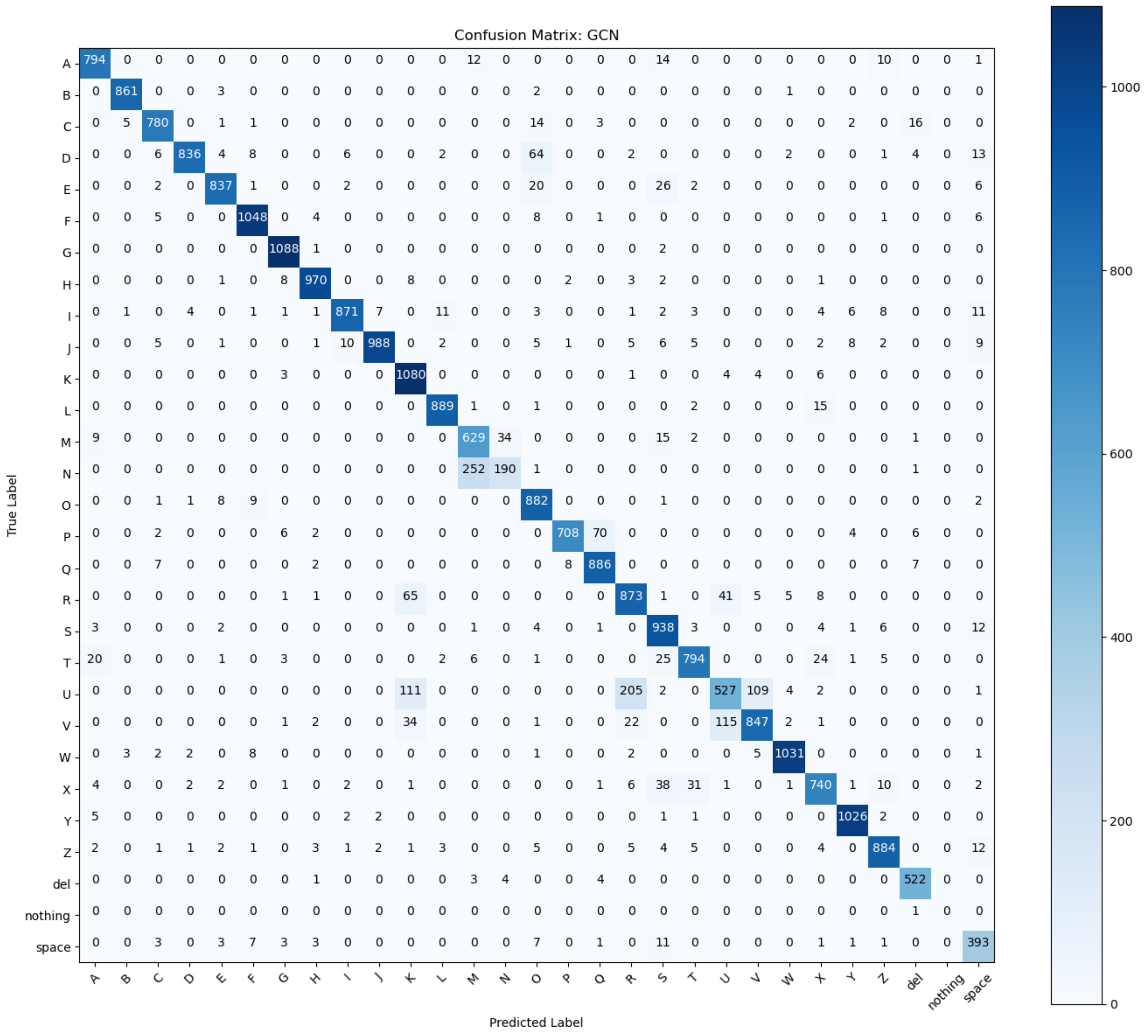


Model Performance Summary

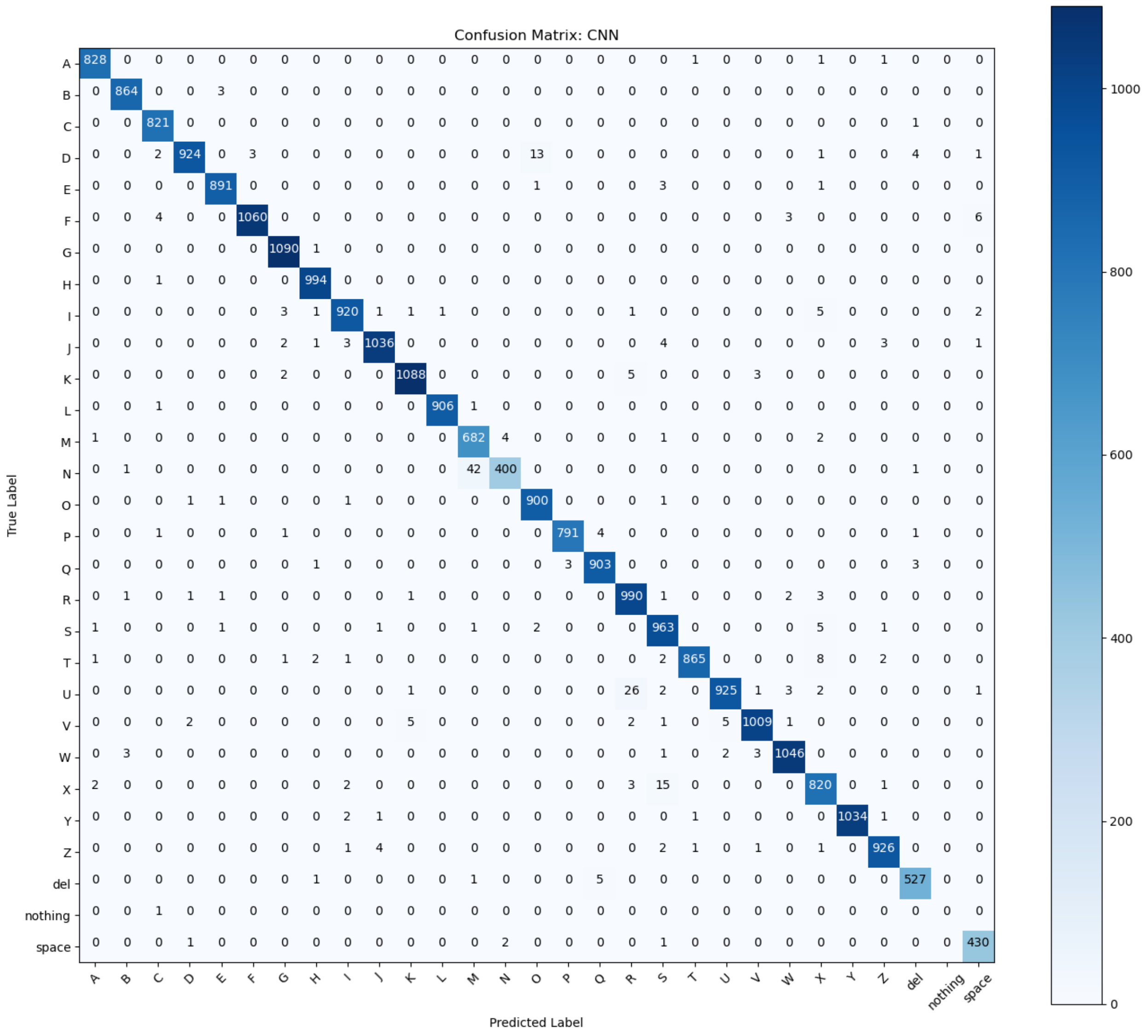
Model	Validation Accuracy	Validation Loss
MLP	90.796%	0.293
GCN	91.85%	0.303
CNN	98.749%	0.049
Combined	99.006%	0.041



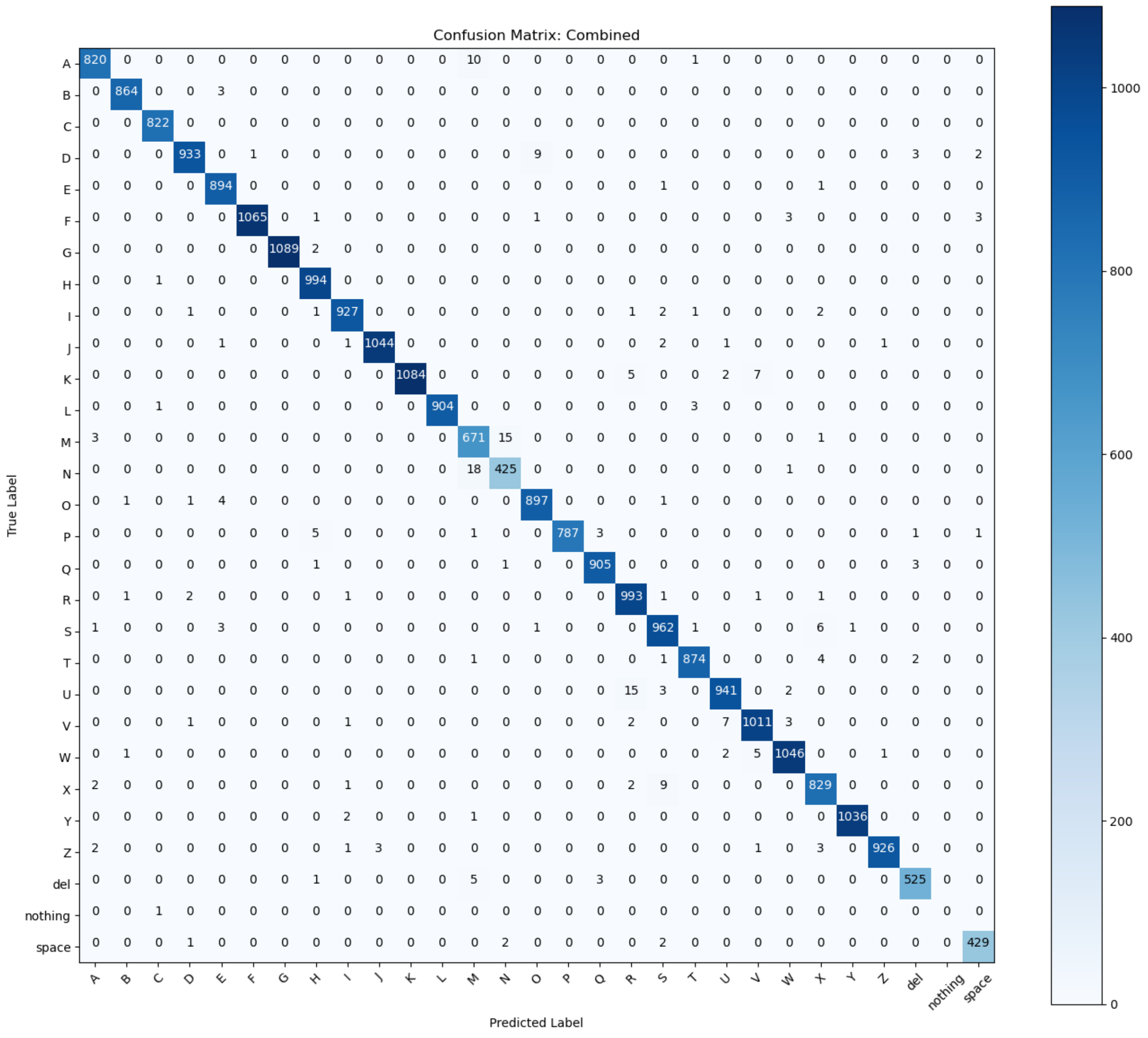
True \ Predicted	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	del	nothing	space
A	805	0	0	0	0	0	0	0	0	0	0	0	10	1	0	0	0	0	8	1	0	0	0	0	0	2	0	0	4
B	0	860	3	0	3	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
C	0	0	795	0	0	0	0	0	0	0	0	0	0	0	0	0	5	0	0	0	0	0	0	0	0	0	22	0	0
D	0	2	16	850	3	10	0	0	4	0	0	0	0	0	45	0	0	0	0	0	0	0	5	0	0	3	4	0	6
E	0	0	3	0	880	0	0	0	1	1	0	0	0	0	6	0	0	0	3	1	0	0	0	1	0	0	0	0	0
F	0	1	5	0	0	1054	0	4	0	0	0	0	0	0	2	0	0	0	0	0	0	0	1	0	0	0	0	0	6
G	0	0	0	0	0	0	1078	8	0	0	0	0	0	0	0	0	0	0	4	0	0	0	0	0	0	0	0	0	1
H	0	0	1	0	0	0	12	977	0	0	0	0	0	2	0	0	0	0	0	0	0	1	0	0	0	0	0	0	2
I	0	2	0	9	1	0	0	1	870	4	1	14	0	0	3	0	0	0	0	3	0	0	0	1	9	8	0	0	9
J	0	0	7	4	2	0	1	2	24	964	0	6	0	0	0	0	0	0	6	3	0	0	0	5	14	7	0	0	5
K	0	0	0	0	0	0	3	0	0	0	1044	0	0	0	0	0	0	6	0	0	20	22	0	3	0	0	0	0	0
L	0	0	0	0	0	0	0	0	0	0	0	899	1	0	0	0	0	0	0	2	0	0	0	5	0	0	0	0	1
M	2	0	0	0	0	0	0	0	0	0	0	0	654	31	0	0	0	0	1	0	0	0	0	0	0	0	1	0	1
N	0	1	0	0	0	0	0	0	0	0	0	0	166	275	0	0	0	0	0	0	0	0	0	0	0	0	2	0	0
O	0	1	2	0	4	2	0	0	0	0	0	0	0	0	894	0	0	0	0	0	0	0	0	0	0	1	0	0	0
P	0	0	1	0	0	0	0	5	0	0	0	0	0	0	0	737	46	0	0	0	0	0	0	0	0	0	9	0	0
Q	0	0	3	0	0	0	0	1	0	0	0	0	2	0	0	0	897	0	0	0	0	0	0	0	0	0	7	0	0
R	0	0	0	0	0	0	1	0	0	0	19	5	0	0	0	0	0	501	1	0	348	109	7	9	0	0	0	0	0
S	17	0	0	0	6	0	1	0	0	0	0	0	2	0	2	0	2	0	932	7	0	0	0	5	0	1	0	0	0
T	21	0	0	0	0	0	0	3	0	0	0	1	2	0	1	0	0	0	16	832	0	0	0	1	0	4	0	0	1
U	0	0	0	0	0	0	0	0	0	0	2	0	0	0	0	0	0	56	2	0	340	555	3	2	0	0	0	0	1
V	0	0	1	0	0	0	2	0	0	0	0	1	0	0	0	0	0	5	0	0	25	990	1	0	0	0	0	0	0
W	0	8	0	1	0	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	9	1032	0	0	1	0	0	0
X	11	0	0	1	1	0	1	0	3	0	0	17	1	0	0	0	0	3	23	94	2	0	2	673	2	9	0	0	0
Y	6	0	0	0	0	0	0	0	5	0	0	0	1	0	0	0	0	0	1	0	0	0	0	0	1023	3	0	0	0
Z	1	0	0	2	0	0	0	2	0	4	0	19	0	0	0	0	0	0	6	9	1	0	2	0	1	879	0	0	10
del	1	0	0	0	0	0	0	1	0	0	0	0	5	2	0	1	6	0	0	0	0	0	0	0	0	0	518	0	0
nothing	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
space	0	0	5	0	0	9	0	6	0	0	1	0	0	2	2	2	3	0	2	0	0	0	0	0	0	0	6	0	396



True \ Predicted	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	del	nothing	space
A	794	0	0	0	0	0	0	0	0	0	0	0	12	0	0	0	0	0	14	0	0	0	0	0	0	10	0	0	1
B	0	861	0	0	3	0	0	0	0	0	0	0	0	0	2	0	0	0	0	0	0	0	1	0	0	0	0	0	0
C	0	5	780	0	1	1	0	0	0	0	0	0	0	0	14	0	3	0	0	0	0	0	0	0	2	0	16	0	0
D	0	0	6	836	4	8	0	0	6	0	0	2	0	0	64	0	0	2	0	0	0	0	2	0	0	1	4	0	13
E	0	0	2	0	837	1	0	0	2	0	0	0	0	0	20	0	0	0	26	2	0	0	0	0	0	0	0	0	6
F	0	0	5	0	0	1048	0	4	0	0	0	0	0	0	8	0	1	0	0	0	0	0	0	0	0	1	0	0	6
G	0	0	0	0	0	0	1088	1	0	0	0	0	0	0	0	0	0	0	2	0	0	0	0	0	0	0	0	0	0
H	0	0	0	0	1	0	8	970	0	0	8	0	0	0	0	2	0	3	2	0	0	0	0	1	0	0	0	0	0
I	0	1	0	4	0	1	1	1	871	7	0	11	0	0	3	0	0	1	2	3	0	0	0	4	6	8	0	0	11
J	0	0	5	0	1	0	0	1	10	988	0	2	0	0	5	1	0	5	6	5	0	0	0	2	8	2	0	0	9
K	0	0	0	0	0	0	3	0	0	0	1080	0	0	0	0	0	0	1	0	0	4	4	0	6	0	0	0	0	0
L	0	0	0	0	0	0	0	0	0	0	0	889	1	0	1	0	0	0	0	2	0	0	0	15	0	0	0	0	0
M	9	0	0	0	0	0	0	0	0	0	0	0	629	34	0	0	0	0	15	2	0	0	0	0	0	0	1	0	0
N	0	0	0	0	0	0	0	0	0	0	0	0	252	190	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0
O	0	0	1	1	8	9	0	0	0	0	0	0	0	0	882	0	0	0	1	0	0	0	0	0	0	0	0	0	2
P	0	0	2	0	0	0	6	2	0	0	0	0	0	0	0	708	70	0	0	0	0	0	0	0	4	0	6	0	0
Q	0	0	7	0	0	0	0	2	0	0	0	0	0	0	0	8	886	0	0	0	0	0	0	0	0	0	7	0	0
R	0	0	0	0	0	0	1	1	0	0	65	0	0	0	0	0	0	873	1	0	41	5	5	8	0	0	0	0	0
S	3	0	0	0	2	0	0	0	0	0	0	0	1	0	4	0	1	0	938	3	0	0	0	4	1	6	0	0	12
T	20	0	0	0	1	0	3	0	0	0	0	2	6	0	1	0	0	0	25	794	0	0	0	24	1	5	0	0	0
U	0	0	0	0	0	0	0	0	0	0	111	0	0	0	0	0	0	205	2	0	527	109	4	2	0	0	0	0	1
V	0	0	0	0	0	0	1	2	0	0	34	0	0	0	1	0	0	22	0	0	115	847	2	1	0	0	0	0	0
W	0	3	2	2	0	8	0	0	0	0	0	0	0	0	1	0	0	2	0	0	0	5	1031	0	0	0	0	0	1
X	4	0	0	2	2	0	1	0	2	0	1	0	0	0	0	0	1	6	38	31	1	0	1	740	1	10	0	0	2
Y	5	0	0	0	0	0	0	0	2	2	0	0	0	0	0	0	0	0	1	1	0	0	0	0	1026	2	0	0	0
Z	2	0	1	1	2	1	0	3	1	2	1	3	0	0	5	0	0	5	4	5	0	0	0	4	0	884	0	0	12
del	0	0	0	0	0	0	0	1	0	0	0	0	3	4	0	0	4	0	0	0	0	0	0	0	0	0	522	0	0
nothing	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
space	0	0	3	0	3	7	3	3	0	0	0	0	0	0	7	0	1	0	11	0	0	0	0	1	1	1	0	0	393



True \ Predicted	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	del	nothing	space
A	828	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0	1	0	0	0
B	0	864	0	0	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
C	0	0	821	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
D	0	0	2	924	0	3	0	0	0	0	0	0	0	0	13	0	0	0	0	0	0	0	0	1	0	0	4	0	1
E	0	0	0	0	891	0	0	0	0	0	0	0	0	0	1	0	0	0	3	0	0	0	0	1	0	0	0	0	0
F	0	0	4	0	0	1060	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3	0	0	0	0	0	6
G	0	0	0	0	0	0	1090	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
H	0	0	1	0	0	0	0	994	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
I	0	0	0	0	0	0	3	1	920	1	1	1	0	0	0	0	0	1	0	0	0	0	0	5	0	0	0	0	2
J	0	0	0	0	0	0	2	1	3	1036	0	0	0	0	0	0	0	0	4	0	0	0	0	0	0	3	0	0	1
K	0	0	0	0	0	0	2	0	0	0	1088	0	0	0	0	0	0	5	0	0	0	3	0	0	0	0	0	0	0
L	0	0	1	0	0	0	0	0	0	0	0	906	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
M	1	0	0	0	0	0	0	0	0	0	0	0	682	4	0	0	0	0	1	0	0	0	0	2	0	0	0	0	0
N	0	1	0	0	0	0	0	0	0	0	0	0	42	400	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
O	0	0	0	1	1	0	0	0	1	0	0	0	0	0	900	0	0	0	1	0	0	0	0	0	0	0	0	0	0
P	0	0	1	0	0	0	1	0	0	0	0	0	0	0	0	791	4	0	0	0	0	0	0	0	0	0	1	0	0
Q	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	3	903	0	0	0	0	0	0	0	0	0	3	0	0
R	0	1	0	1	1	0	0	0	0	0	1	0	0	0	0	0	0	990	1	0	0	0	2	3	0	0	0	0	0
S	1	0	0	0	1	0	0	0	0	1	0	0	1	0	2	0	0	0	963	0	0	0	0	5	0	1	0	0	0
T	1	0	0	0	0	0	1	2	1	0	0	0	0	0	0	0	0	0	2	865	0	0	0	8	0	2	0	0	0
U	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	26	2	0	925	1	3	2	0	0	0	0	1
V	0	0	0	2	0	0	0	0	0	0	5	0	0	0	0	0	0	2	1	0	5	1009	1	0	0	0	0	0	0
W	0	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	2	3	1046	0	0	0	0	0	0
X	2	0	0	0	0	0	0	0	2	0	0	0	0	0	0	0	0	3	15	0	0	0	0	820	0	1	0	0	0
Y	0	0	0	0	0	0	0	0	2	1	0	0	0	0	0	0	0	0	0	1	0	0	0	0	1034	1	0	0	0
Z	0	0	0	0	0	0	0	0	1	4	0	0	0	0	0	0	0	0	2	1	0	1	0	1	0	926	0	0	0
del	0	0	0	0	0	0	0	1	0	0	0	0	1	0	0	0	5	0	0	0	0	0	0	0	0	0	527	0	0
nothing	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
space	0	0	0	1	0	0	0	0	0	0	0	0	0	2	0	0	0	0	1	0	0	0	0	0	0	0	0	0	430



	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	del	nothing	space
A	820	0	0	0	0	0	0	0	0	0	0	0	10	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
B	0	864	0	0	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
C	0	0	822	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
D	0	0	0	933	0	1	0	0	0	0	0	0	0	0	9	0	0	0	0	0	0	0	0	0	0	0	3	0	2
E	0	0	0	0	894	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	1	0	0	0	0	0
F	0	0	0	0	0	1065	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	3	0	0	0	0	0	3
G	0	0	0	0	0	0	1089	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
H	0	0	1	0	0	0	0	994	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
I	0	0	0	1	0	0	0	1	927	0	0	0	0	0	0	0	0	1	2	1	0	0	0	2	0	0	0	0	0
J	0	0	0	0	1	0	0	0	1	1044	0	0	0	0	0	0	0	0	2	0	1	0	0	0	0	1	0	0	0
K	0	0	0	0	0	0	0	0	0	0	1084	0	0	0	0	0	0	5	0	0	2	7	0	0	0	0	0	0	0
L	0	0	1	0	0	0	0	0	0	0	0	904	0	0	0	0	0	0	0	3	0	0	0	0	0	0	0	0	0
M	3	0	0	0	0	0	0	0	0	0	0	0	671	15	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
N	0	0	0	0	0	0	0	0	0	0	0	0	18	425	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
O	0	1	0	1	4	0	0	0	0	0	0	0	0	0	897	0	0	0	1	0	0	0	0	0	0	0	0	0	0
P	0	0	0	0	0	0	0	5	0	0	0	0	1	0	0	787	3	0	0	0	0	0	0	0	0	0	1	0	1
Q	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	905	0	0	0	0	0	0	0	0	0	3	0	0
R	0	1	0	2	0	0	0	0	1	0	0	0	0	0	0	0	0	993	1	0	0	1	0	1	0	0	0	0	0
S	1	0	0	0	3	0	0	0	0	0	0	0	0	0	1	0	0	0	962	1	0	0	0	6	1	0	0	0	0
T	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	1	874	0	0	0	4	0	0	2	0	0
U	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	15	3	0	941	0	2	0	0	0	0	0	0
V	0	0	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0	2	0	0	7	1011	3	0	0	0	0	0	0
W	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	5	1046	0	0	1	0	0	0
X	2	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	2	9	0	0	0	0	829	0	0	0	0	0
Y	0	0	0	0	0	0	0	0	2	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1036	0	0	0	0
Z	2	0	0	0	0	0	0	0	1	3	0	0	0	0	0	0	0	0	0	0	0	1	0	3	0	926	0	0	0
del	0	0	0	0	0	0	0	1	0	0	0	0	5	0	0	0	3	0	0	0	0	0	0	0	0	0	525	0	0
nothing	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
space	0	0	0	1	0	0	0	0	0	0	0	0	0	2	0	0	0	0	2	0	0	0	0	0	0	0	0	0	429