DEVELOPING A HYBRID DEEP LEARNING SYSTEM FOR ROBUST FINGER GESTURE DETECTION AND CONTROL

Submitted by

AJAY. S	**211520243002**
ANDREW SANTHOSH. C	**211520243003**
SANJAY. S	**211520243050**
VISHAL. P	**211520243060**

In partial fulfillment for the award of the degree

Of

BACHELOR OF TECHNOLOGY

IN

ARTIFICIAL INTELLIGENCE AND DATA SCIENCE

PANIMALAR INSTITUTE OF TECHNOLOGY, POONAMALLEE

ANNA UNIVERSITY: CHENNAI – 600025

MAY 2024

BONAFIDE CERTIFICATE

Certified that this project report "**DEVELOPING A HYBRID DEEP LEARNING SYSTEM FOR ROBUST FINGER GESTURE DETECTION AND CONTROL**" is the bonafide work, **AJAY. S (211520243002), ANDREW SANTHOSH. C (211520243003), SANJAY. S (211520243050)** and **VISHAL. P (211520243060)** of final year B. Tech – AI & DS during the academic year 2023 – 24 who carried out under my supervision.

SIGNATURE
Dr. T. KALAICHELVI, M. Tech., Ph.D
Professor and Head,
Department of AI & DS,
Panimalar Institute of Technology
Poonamallee, Chennai - 600123.

SIGNATURE
Dr. C. GNANAPRAKASAM, M.E., Ph.D.
Associate Professor,
Department of AI & DS
Panimalar Institute of Technology
Poonamallee, Chennai - 600123.

Certified that the above candidates were examined in the university project work viva voce examination held on **07/05/2024** at Panimalar Institute of Technology, Chennai– 600 123.

INTERNAL EXAMINER **EXTERNAL EXAMINER**

ACKNOWLEDGEMENT

A project of this magnitude and nature requires the kind cooperation and support of many, for successful completion. We wish to express our sincere thanks to all those who were involved in the completion of this project.

We would like to express our deep gratitude to **Our Beloved Secretary and Correspondent, Dr. P. CHINNADURAI, M.A., Ph.D.,** for his kind words and enthusiastic motivation which inspired us a lot in completing the project.

We also express our sincere thanks to Our **Dynamic Directors, Mrs. C. VIJAYARAJESWARI, Dr. C. SAKTHIKUMAR, M.E., Ph.D., and Dr. S. SARANYA SREE SAKTHIKUMAR, B.E., M.B.A., Ph. D.,** for Providing us with the necessary facilities for the completion of this project.

We also express our gratefulness to our **Principal Dr. T. JAYANTHY, M.E., Ph.D.,** who helped us in the completion of this project.

We wish to convey our thanks and gratitude to our **Head of the Department Dr. T. KALAICHELVI, M.E., Ph.D.,** Department of Artificial intelligence and Data science, for her support and providing us ample time to complete our project.

We express our indebtedness and gratitude to our **Associate Professor, Dr. C. Gnana Prakasam, M.E, Ph. D.,** Department of Artificial intelligence and Data science, for her guidance throughout the project.

TABLE OF CONTENTS

ABSTRACT

Interaction between humans and computers has evolved significantly with the advancement of computer vision technologies. This project aims to leverage the capabilities of OpenCV, MediaPipe, and PyTorch to develop a real-time gesture recognition system for enhancing our ability to interact with a computer system. The system utilizes a webcam to track and analyse hand gestures performed by users.

The core of the system lies in the integration of OpenCV for webcam input, MediaPipe for hand detection and landmark tracking, and PyTorch for gesture classification. Upon detecting and recognizing specific hand gestures, relevant actions are triggered, enabling seamless interaction between users and computers.

The project emphasizes the design and implementation of efficient algorithms for real-time hand gesture detection and classification. Techniques such as deep learning-based feature extraction and classification are employed to achieve high accuracy and robustness in gesture recognition.

By providing an intuitive and natural interface for users to interact with computers, this project aims to improve the overall user experience and accessibility of computing devices.

Keywords: Computer Interaction, OpenCV, MediaPipe, PyTorch, Computer Vision, Hand Tracking, Webcam, Real-time Processing

LIST OF FIGURES

LIST OF ABBREVIATIONS

Acronym	Abbreviation
ASR	Automatic Speech Recognition
ANN	Artificial Neural Network
IDE	Integrated Development Environment
GPU	Graphics Processing Unit
SSD	Solid-State Storage
CV	Computer Vision
ONNX	Open Neural Network Exchange
CNN	Convolutional Neural Network
SIANN	Space Invariant Artificial Neural Networks

INTRODUCTION

1.1. INTRODUCTION TO MEDIAPIPE / SPEECH RECOGNITION:

MediaPipe is an open-source framework created by Google for building machine learning pipelines. It simplifies the process of creating applications that use on-device machine learning, especially for computer vision tasks.

MediaPipe is particularly useful for creating on-device machine learning solutions. This means the processing of data happens directly on the device running the application, like a smartphone or a laptop, rather than needing to be sent to the cloud. This makes your applications faster and more efficient, especially when dealing with real-time data.

Speech recognition, also known as Automatic Speech Recognition (ASR) or speech-to-text, is a technology that bridges the gap between spoken language and computers. It empowers machines to understand and translate spoken words into a written format.

Speech recognition systems employ a combination of clever techniques. They analyse sound waves from your voice, converting them into a digital representation. Then, complex algorithms come into play. These algorithms identify patterns and features within the audio, attempting to match them with known words and phrases stored in a vast language database. Finally, the system translates the recognized words into written text.

1.2. OVERVIEW OF THE PROJECT:

This project develops a hybrid deep learning system for robust finger gesture detection and control. The system leverages hand landmark data captured from MediaPipe to recognize various hand postures. A custom PyTorch model, GestureNet, is designed to classify these gestures based on key features extracted from the landmarks. These features include critical angles between fingers and distances between keypoints on the hand. The extracted features are pre-processed using normalization techniques to ensure consistent model input. GestureNet utilizes a convolutional neural network architecture to efficiently process the spatial information within the hand landmarks. Additionally, separate layers handle the calculated angles and distances, allowing the model to learn from both geometric relationships and relative positions of hand elements.

The system integrates gesture recognition with real-time control actions. Each detected gesture class is mapped to specific mouse and keyboard actions using the PyAutoGUI library. This enables users to interact with their computers through hand gestures, offering an alternative and potentially more intuitive interface for various tasks. The model is exported to the ONNX format, facilitating efficient inference for real-time performance.

Beyond gesture control, the system incorporates OpenAI's Whisper model for audio transcription. Whisper transcribes speech from the user's environment in real-time, and the transcribed text can be used for keyboard input, further expanding the system's capabilities. This integration creates a multimodal interface that combines gesture recognition and voice commands for a richer user experience.

1.3. PROJECT SYNOPSIS:

This project proposes a hybrid deep learning system for robust finger gesture detection and control. It utilizes a custom PyTorch model, GestureNet, to classify hand gestures captured from MediaPipe landmarks. GestureNet extracts key features like angles and distances, achieving robust classification through convolutional neural networks. Each gesture class triggers specific mouse/keyboard actions through PyAutoGUI. Additionally, the system integrates Whisper for real-time audio transcription, enabling multimodal interaction with gestures and voice commands.

1. **Gesture Classification:**
   - A custom PyTorch model named GestureNet classifies hand gestures.
   - GestureNet utilizes MediaPipe hand landmarks to extract key features.

2. **Feature Extraction:**
   - GestureNet extracts critical angles between fingers.
   - It also calculates distances between keypoints on the hand.

3. **Real-Time Control:**
   - Each recognized gesture class triggers specific mouse or keyboard actions.
   - PyAutoGUI library facilitates the mapping between gestures and actions.

4. **Audio Transcription:**
   - OpenAI's Whisper model is integrated for real-time audio transcription.
   - The transcribed text can be used for keyboard input.

5. **Multimodal Interaction:**
   - The system combines gesture recognition and voice commands.
   - This enables a richer user experience with more intuitive interaction options.

1.4. PROJECT SCOPE:

This project aims to develop a robust and versatile system that leverages deep learning techniques and computer vision algorithms to recognize hand gestures and translate them into specific actions on a computer. The system is designed to provide an intuitive and hands-free interface, enabling users to control various applications and perform tasks efficiently.

1. **Gesture Recognition**: Implement a deep learning model capable of accurately classifying a diverse set of hand gestures, including common gestures like pointing, pinching, and swiping, as well as custom-defined gestures.

2. **Multimodal Input**: Integrate audio transcription capabilities using OpenAI's Whisper model, allowing users to combine hand gestures with voice commands for a more natural and seamless interaction experience.

3. **Action Mapping**: Establish a flexible framework to map recognized gestures and voice commands to corresponding actions, such as mouse movements, clicks, scrolling, keyboard input, and application-specific commands.

4. **Cross-Platform Compatibility**: Ensure the system functions seamlessly across various operating systems and hardware configurations, providing a consistent user experience regardless of the underlying platform.

5. **User-Friendly Interface**: Develop an intuitive and user-friendly graphical interface that displays real-time gesture recognition feedback, allows for gesture customization, and provides clear visual cues for available actions.

6. **Extensibility and Customization**: Design a modular and extensible architecture that facilitates the addition of new gestures, actions, and application-specific integrations, enabling users to tailor the system to their specific needs.

1.5. OBJECTIVE OF THE PROJECT:

- **Enhance Human-Computer Interaction**: The primary objective of this project is to revolutionize the way users interact with computers by introducing a natural and intuitive interface based on hand gestures and voice commands. This approach aims to make computer interactions more seamless, efficient, and accessible, particularly for users with physical disabilities or those in hands-free environments.

- **Improve Productivity and Accessibility**: By combining hand gestures and voice commands, the project seeks to streamline various computer tasks, such as navigating applications, executing commands, and inputting text. This multimodal approach can significantly improve productivity by reducing the reliance on traditional input devices and enabling users to multitask more effectively.

- **Develop a Robust and Accurate Recognition System**: A key objective is to develop a highly accurate and robust hand gesture recognition system that can reliably classify a wide range of gestures, even in challenging lighting conditions or with partial occlusions. This involves leveraging advanced computer vision techniques, deep learning models, and sophisticated data preprocessing methods.

- **Enable Customization and Extensibility**: The project aims to create a flexible and extensible framework that allows users to customize and define their own gestures and associated actions. This customization capability ensures that the system can adapt to individual preferences and specific use cases, making it more versatile and user-friendly.
- **Promote Accessibility and Inclusivity**: By offering an alternative input modality, the project strives to make computing more accessible to individuals with physical limitations or disabilities. The hands-free nature of the system can empower users who may have difficulty using traditional

input devices, fostering greater inclusivity and enabling them to interact with computers more effectively.

CHAPTER 2

LITERATURE SURVEY

2.1. LITERATURE SURVEY:

1. PAPER TITLE: Hand Gesture Recognition for Multi-Culture Sign Language Using Graph and General Deep

AUTHORS: Abu Saleh Musa Miah, MD. Al Mehedi Hasan, Yoichi Tomioka and Jungpil Shin

ABSTRACT:

Hand gesture-based Sign Language Recognition (SLR) is vital for communication between deaf and non-deaf individuals. Existing systems excel in recognizing cultural sign languages but struggle with multicultural sign languages (McSL). To tackle this, we propose GmTC, an end-to-end SLR system. GmTC utilizes a Graph and General deep-learning network to extract features. The graph-based stream employs superpixel values and graph convolutional network (GCN) for complex relationship features. The second stream utilizes attention-based contextual information through multi-stage, multi-head self-attention (MHSA) and CNN modules for long and short-range dependency features. Combining these features enhances understanding by translating McSL into text. Extensive experiments on five cultural SL datasets demonstrate superior accuracy compared to existing models, showcasing GmTC's effectiveness and generalizability.

MERITS:

- GmTC combines GCN, MHSA, and CNN for diverse feature extraction.
- Achieves strong performance across diverse sign language datasets.
- Shows adaptability to multi-culture sign language recognition tasks.
- Captures complex relationships while reducing computational complexity.
- Improves efficiency by preserving spatial information with superpixels.

DEMERITS:

- Multiple streams and modules increase model complexity.
- Requires significant computational resources despite efficiency efforts.
- Performance may vary based on dataset characteristics.
- Difficult to interpret learned features and patterns.
- High performance on limited datasets may lead to overfitting.

2. PAPER TITLE: Multistage spatial attention-based neural network for hand gesture recognition

AUTHORS: A. S. M. Miah, M. A. M. Hasan, J. Shin, Y. Okuyama, and Y. Tomioka

ABSTRACT:

The definition of human-computer interaction (HCI) has changed in the current year because people are interested in their various ergonomic devices' ways. Many researchers have been working to develop a hand gesture recognition system with a kinetic sensor-based dataset, but their performance accuracy is not satisfactory. In our work, we proposed a multistage spatial attention-based neural network for hand gesture recognition to overcome the challenges. We included three stages in the proposed model where each stage is inherited the CNN; where we first apply a feature extractor and a spatial attention module by using self-attention from the original dataset and then multiply the feature vector with the attention map to highlight effective features of the dataset. Then, we explored features concatenated with the original dataset for obtaining modality feature embedding. In the same way, we generated a feature vector and attention map in the second stage with the feature extraction architecture and self-attention technique. After multiplying the attention map and features, we produced the final feature, which feeds into the third stage, a classification module to predict the label of the correspondent hand gesture. Our model achieved 99.67%, 99.75%, and 99.46% accuracy for the senz3D, Kinematic, and NTU datasets.

MERITS:

- High accuracy rates on multiple datasets
- Spatial attention mechanism improves performance
- Multistage architecture captures hierarchical features
- Feature extraction and embedding stages

DEMERITS:

- Increased computational complexity
- Potential overfitting concerns
- Dataset dependency
- Interpretability challenges
- Possible real-time performance issues

3. PAPER TITLE: Dynamic hand gesture recognition using multi-branch attention-based graph and general deep learning model

AUTHORS: A. S. M. Miah, M. A. M. Hasan, J. Shin

ABSTRACT:

We propose a Multi-branch Attention-based Graph model for hand gesture recognition, utilizing spatial-temporal features extracted from hand skeleton data. Our model combines graph-based neural networks and general deep learning techniques to extract various features. Two graph-based channels generate spatial-temporal and temporal-spatial features, while a general deep learning channel extracts overall features. By concatenating these features, our model achieves high accuracy (94.12%, 92.00%, and 97.01%) on MSRA, DHG, and SHREC'17 benchmark datasets, respectively, surpassing existing methods while maintaining low computational cost

MERITS:

- The model combines graph neural networks and deep learning techniques.
- It extracts spatial-temporal and temporal-spatial features from hand skeleton data.
- The multi-branch architecture allows diverse feature extraction.
- High accuracy is achieved on MSRA, DHG, and SHREC'17 datasets.
- It surpasses existing methods in performance.
- Low computational cost is maintained.

DEMERITS:

- Multiple branches increase model complexity.
- Potential overfitting to benchmark datasets is a concern.
- Robustness to variations or noise is not mentioned.

4. PAPER TITLE: Bengali sign language alphabet recognition using concatenated segmentation and convolutional neural network

AUTHORS: A. S. M. Miah, J. Shin, M. A. M. Hasan, and M. A. Rahim

ABSTRACT:

Sign language recognition poses a significant challenge in machine learning and human-computer interaction. While many classification models exist for various sign languages, few address general performance across different datasets. Existing models often excel with small datasets but struggle to generalize to larger or diverse ones. This paper presents a novel approach for recognizing Bengali sign language (BSL) alphabets, focusing on improving generalization. Our method is evaluated on three benchmark datasets: '38 BdSL', 'KU-BdSL', and 'Ishara-Lipi'. We employ a three-step approach: segmentation using concatenated methods, augmentation with seven techniques to expand training data, and classification using a Convolutional Neural Network (CNN) called BenSignNet. Achieving accuracies of 94.00%, 99.60%, and 99.60% for the BdSL Alphabet, KU-BdSL, and Ishara-Lipi datasets, respectively, our method outperforms conventional approaches and demonstrates robust generalization across BSL datasets.

MERITS:

- Addresses the challenge of sign language recognition across different datasets.
- Focuses on improving generalization for Bengali sign language (BSL) alphabet recognition.
- Evaluates on three benchmark BSL datasets: '38 BdSL', 'KU-BdSL', and 'Ishara-Lipi'.
- Employs a three-step approach: segmentation, data augmentation, and CNN classification.
- Utilizes seven augmentation techniques to expand the training data.
- Achieves high accuracy on all three datasets: 94.00%, 99.60%, and 99.60%.
- Outperforms conventional approaches for BSL alphabet recognition.
- Demonstrates robust generalization across different BSL datasets.

DEMERITS:

- Limited to recognition of BSL alphabets, not addressing complete sign language recognition

5. PAPER TITLE: Japanese Sign Language Recognition by Combining Joint Skeleton-Based Handcrafted and Pixel-Based Deep Learning Features with Machine Learning Classification

AUTHORS: J. Shin, M. A. M. Hasan, A. S. M. Miah, K. Suzuki, and K. Hirooka

ABSTRACT:

Sign language recognition is crucial for improving communication accessibility among Deaf and hard-of-hearing communities. In Japan, about 360,000 individuals with hearing and speech disabilities rely on Japanese Sign Language (JSL). However, existing JSL recognition systems face performance limitations due to complexities. To address this, we propose a novel JSL recognition system using a fusion approach, combining handcrafted features and deep learning. Our system extracts hand and body movements in JSL gestures while capturing hierarchical representations using deep learning. By combining these features, we enhance JSL recognition accuracy and robustness. Experiments on our Lab JSL dataset and a publicly available Arabic sign language dataset confirm the effectiveness of our approach.

MERITS:

- Novel JSL dataset introduced
- Innovative fusion of handcrafted and deep learning features
- Higher accuracy and robustness
- Cost-effective using webcam
- Empirically validated

DEMERITS:

- Potentially limited dataset size
- Increased computational complexity
- Handling intricate sign complexity
- Challenges in data collection
- Interpretability vs performance trade-off

CHAPTER 3

SYSTEM ANALYSIS

3.1. EXISTING SYSTEM:

The existing systems for gesture recognition and control vary in complexity and application. Some notable examples include:

- **Leap Motion:** Leap Motion is a hand tracking device that allows users to control computers through hand and finger gestures. It uses infrared sensors to track the movement of hands and fingers in 3D space, enabling intuitive interaction with digital content without the need for physical controllers.
- **Microsoft Kinect:** Originally developed for gaming, the Kinect sensor can track body movements and gestures in real-time using depth-sensing technology. It has been used in various applications beyond gaming, including healthcare, education, and motion capture for animation.
- **Google Soli:** Soli is a radar-based gesture recognition technology developed by Google. It enables touchless interaction with devices by detecting subtle hand and finger movements in the air. Soli has been integrated into devices like the Google Pixel 4 smartphone and the Nest Hub
- **Gesture Recognition in Smart TVs:** Many smart TVs now incorporate gesture recognition technology, allowing users to control the TV interface and playback functions using hand gestures. These systems typically use cameras or sensors embedded in the TV to detect and interpret gestures.
- **Gesture-Controlled Wearables:** Some wearable devices, such as smartwatches and augmented reality glasses, incorporate gesture recognition features for hands-free interaction. Users can perform gestures like swiping, tapping, or waving to navigate menus, answer calls, or control other device functions.
- **Augmented Reality (AR) and Virtual Reality (VR) Systems:** AR and VR platforms often utilize gesture recognition to enhance immersive experiences. Users can interact with virtual objects and environments using hand gestures tracked by motion controllers or camera systems.

3.1.1. LIMITATIONS:

- **Accuracy and Reliability:** Existing systems may suffer from limitations in accurately detecting and interpreting gestures, especially in complex or dynamic environments. Factors such as lighting conditions, background clutter, and occlusions can affect the system's reliability.

- **Limited Gesture Vocabulary:** Many existing systems support a predefined set of gestures, limiting the range of interactions available to users. Adding new gestures or customizing existing ones may be challenging or impractical.

- **Hardware Requirements:** Some gesture recognition systems rely on specialized hardware, such as depth sensors or infrared cameras, which may not be readily available or affordable for all users. This can limit the accessibility and widespread adoption of the technology.

- **User Adaptation:** Users may need time to adapt to gesture-based interfaces and learn the appropriate gestures for specific commands or actions. Complex gestures or unfamiliar interactions can result in a steep learning curve, potentially leading to user frustration or abandonment of the system.

- **Privacy and Security Concerns:** Gesture recognition systems that capture and analyse user movements raise privacy and security concerns, especially if sensitive or personal information is involved. Users may be wary of sharing biometric data or having their movements tracked without their consent.

3.2. PROPOSED SYSTEM:

1. **Gesture Detection Module (GestureNet):** GestureNet incorporates a robust gesture detection module powered by convolutional neural networks (CNNs). These CNNs are trained on extensive datasets to accurately recognize various finger gestures and hand movements captured by the webcam in real-time. The module utilizes sophisticated algorithms to analyse hand landmarks and extract key features, enabling precise gesture classification.

2. **Action Implementation Module:** Upon detecting specific gestures, the system's action implementation module executes corresponding actions seamlessly. These actions may include opening applications, controlling the mouse cursor for navigation and selection, adjusting volume levels, or switching between tasks. The module is designed for efficiency and responsiveness, ensuring swift execution of user commands.

3. **Integration with OpenAI's Whisper Model:** The system integrates OpenAI's Whisper model seamlessly, enhancing accessibility and productivity. Whisper provides real-time transcription of speech from the

user's environment, converting spoken words into text input. This integration enables users to dictate commands, messages, or notes directly into text fields, expanding the system's capabilities and catering to diverse user needs.

4. **User Interface:** The system features an intuitive user interface designed for ease of interaction. It provides visual feedback on recognized gestures, displaying annotations or indicators to convey detected hand movements effectively. Additionally, the interface presents feedback on executed actions, ensuring users remain informed and in control of their interactions with the system. The user interface is customizable, allowing users to adjust settings and preferences to suit their individual preferences and workflow.

GestureNet Classes

CLASS_NAME	ACTION	DESCRIPTION
NONE	-	-
CLOSED_PALM	Closed hand	None
INDEX_POINTING	Point index finger	Mouse move
OPEN_PALM	Open hand	Mouse move
THUMB_MIDDLE_TOUCH	Pinch thumb and middle finger	Mouse left button
THUMB_RING_TOUCH	Pinch thumb and ring finger	Mouse right button
THUMB_PINKY_TOUCH	Pinch thumb and pinky	Text input with voice
3_FINGER_PINCH	Pinch thumb, index and middle	Mouse drag
5_FINGER_PINCH	Pinch with all fingers	Mouse scroll
THUMBS_UP	Thumbs Up	Volume up

THUMBS_DOWN	Thumbs Down	Volume down
MIDDLE_UP	Middle finger upwards	Copy text
MIDDLE_DOWN	Middle finger downwards	Paste text

3.2.1. ADVANTAGES:

- **High Accuracy and Responsiveness:** The integration of convolutional neural networks ensures accurate and responsive gesture detection, enhancing user experience and efficiency.

- **Versatile Action Execution:** The action implementation module allows for the execution of a wide range of actions, empowering users to perform tasks effortlessly using hand gestures.

- **Enhanced Accessibility:** Integration with OpenAI's Whisper model improves accessibility by providing real-time transcription, catering to users with diverse needs and preferences.

- **Intuitive Interaction:** The intuitive user interface offers clear feedback on recognized gestures and executed actions, enhancing user understanding and engagement with the system.

- **Productivity Enhancement:** By seamlessly integrating gesture control with real-time transcription and versatile action execution, the system enhances productivity and workflow efficiency.

3.3. HARDWARE REQUIREMENTS

The hardware requirements for this project are relatively modest, making it accessible to a wide range of users with various computing setups. The essential hardware components needed are:

- A computer or laptop with a modern CPU and sufficient RAM
- A webcam or RGB camera capable of capturing high-quality video feed
- A microphone for audio input (built-in or external)
- Sufficient storage space for data collection and model training

Optionally, for an enhanced user experience, the following hardware components can be beneficial:

- A high-resolution webcam or depth camera for improved hand tracking accuracy
- A dedicated GPU (NVIDIA or AMD) for accelerated model training and inference
- An external display or monitor for a larger viewing area and better visual feedback

While the project can run on modest hardware configurations, more powerful hardware can contribute to better performance, faster processing times, and a smoother overall experience.

REQUIREMENTS:

- 500 Gigabytes of Solid-State Storage (SSD).
- 16 Gigabytes of RAM or more.
- A Graphics Processing Unit (GPU) with CUDA Capability for faster inferencing.

3.4. SOFTWARE REQUIREMENTS

The software requirements for this project are primarily centred around various libraries, frameworks, and tools for computer vision, deep learning, and automation tasks. Here are the key software components required:

- **Programming Language**: Python (version 3.10 or later)
- **Computer Vision Libraries**:
    - **OpenCV**: A popular open-source computer vision library for image and video processing.
    - **MediaPipe**: Google's cross-platform, customizable machine learning solution for building multimodal applied ML pipelines, including hand tracking.
- **Deep Learning Frameworks**:

    - **PyTorch**: A widely-used open-source machine learning framework for building and training deep neural networks.
    - **ONNX**: An open format for representing machine learning models, used for model serialization and deployment.

- **Audio Processing**:
    - **OpenAI Whisper**: A speech recognition model for robust and accurate audio transcription.

- **Automation Libraries**:
    - **PyAutoGUI**: A cross-platform GUI automation library for controlling mouse movements, keyboard inputs, and other system interactions.

- **Data Processing and Visualization**:
    - **NumPy**: A library for scientific computing, providing support for large multi-dimensional arrays and matrices.
    - **Matplotlib**: A plotting library for creating static, animated, and interactive visualizations in Python.

- **integrated Development Environment (IDE)**:
    - Any Python-compatible IDE or text editor, such as PyCharm, Visual Studio Code, or Jupyter Notebook.

- **Operating System**:
    - The project should be compatible with major operating systems like Windows, and Linux distributions.

CHAPTER 4

SOFTWARE DESCRIPTION

4.1. PyTorch:

PyTorch is a machine learning library based on the Torch library, used for applications such as computer vision and natural language processing, originally developed by Meta AI and now part of the Linux Foundation umbrella. It is recognized as one of the two most popular machine learning libraries alongside TensorFlow, offering free and open-source software released under the modified BSD license. Although the Python interface is more polished and the primary focus of development, PyTorch also has a C++ interface.

A number of pieces of deep learning software are built on top of PyTorch, including Tesla Autopilot, Uber's Pyro, Hugging Face's Transformers, PyTorch Lightning, and Catalyst.

PyTorch provides two high-level features:

- Tensor computing (like NumPy) with strong acceleration via graphics processing units (GPU)
- Deep neural networks built on a tape-based automatic differentiation system

4.1.1. NEURAL NETWORKS:

In machine learning, a neural network (also artificial neural network or neural net, abbreviated ANN or NN) is a model inspired by the structure and function of biological neural networks in animal brains.

An ANN consists of connected units or nodes called artificial neurons, which loosely model the neurons in a brain. These are connected by edges, which model the synapses in a brain. Each artificial neuron receives signals from connected neurons, then processes them and sends a signal to other connected neurons. The "signal" is a real number, and the output of each neuron is computed by some non-linear function of the sum of its inputs, called the activation function. The strength of the signal at each connection is determined by a weight, which adjusts during the learning process.

Typically, neurons are aggregated into layers. Different layers may perform different transformations on their inputs. Signals travel from the first layer (the input layer) to the last layer (the output layer), possibly passing through multiple intermediate layers (hidden layers). A network is typically called a deep neural network if it has at least 2 hidden layers.

Artificial neural networks are used for various tasks, including predictive modelling, adaptive control, and solving problems in artificial intelligence. They can learn from experience, and can derive conclusions from a complex and seemingly unrelated set of information.

Gradient based methods such as backpropagation are usually used to estimate the parameters of the network. During the training phase, ANNs learn from labelled training data by iteratively updating their parameters to minimize a defined loss function. This method allows the network to generalize to unseen data.

ANNs have evolved into a broad family of techniques that have advanced the state of the art across multiple domains. The simplest types have one or more static components, including number of units, number of layers, unit weights and topology. Dynamic types allow one or more of these to evolve via learning. The latter is much more complicated but can shorten learning periods and produce better results. Some types allow/require learning to be "supervised" by the operator, while others operate independently. Some types operate purely in hardware, while others are purely software and run-on general-purpose computers.

- Convolutional neural networks that have proven particularly successful in processing visual and other two-dimensional data; where long short-term memory avoids the vanishing gradient problem and can handle signals that have a mix of low and high frequency components aiding large-vocabulary speech recognition, text-to-speech synthesis, and photo-real talking heads.
- Competitive networks such as generative adversarial networks in which multiple networks compete with each other, on tasks such as winning a game or on deceiving the opponent about the authenticity of an input.

4.1.2. CONVOLUTIONAL NEURAL NETWORKS

Convolutional neural network (CNN) is a regularized type of feed-forward neural network that learns feature engineering by itself via filters (or kernel)

optimization. Vanishing gradients and exploding gradients, seen during backpropagation in earlier neural networks, are prevented by using regularized weights over fewer connections.

For example, for each neuron in the fully-connected layer, 10,000 weights would be required for processing an image sized 100 × 100 pixels. However, applying cascaded convolution (or cross-correlation) kernels, only 25 neurons are required to process 5x5-sized tiles. Higher-layer features are extracted from wider context windows, compared to lower-layer features.

A convolutional neural network consists of an input layer, hidden layers and an output layer. In a convolutional neural network, the hidden layers include one or more layers that perform convolutions. Typically, this includes a layer that performs a dot product of the convolution kernel with the layer's input matrix. This product is usually the Frobenius inner product, and its activation function is commonly ReLU. As the convolution kernel slides along the input matrix for the layer, the convolution operation generates a feature map, which in turn contributes to the input of the next layer. This is followed by other layers such as pooling layers, fully connected layers, and normalization layers. Here it should be noted how close a convolutional neural network is to a matched filter.

CNNs are also known as Shift Invariant or Space Invariant Artificial Neural Networks (SIANN), based on the shared-weight architecture of the convolution kernels or filters that slide along input features and provide translation-equivariant responses known as feature maps. Counter-intuitively, most convolutional neural networks are not invariant to translation, due to the downsampling operation they apply to the input.

Convolutional networks were inspired by biological processes in that the connectivity pattern between neurons resembles the organization of the animal visual cortex. Individual cortical neurons respond to stimuli only in a restricted region of the visual field known as the receptive field. The receptive fields of different neurons partially overlap such that they cover the entire visual field.

CNNs use relatively little pre-processing compared to other image classification algorithms. This means that the network learns to optimize the filters (or kernels) through automated learning, whereas in traditional algorithms these filters are hand-engineered. This independence from prior knowledge and human intervention in feature extraction is a major advantage.

4.1.3. ONNX:

The Open Neural Network Exchange (ONNX) is an open-source artificial intelligence ecosystem of technology companies and research organizations that establish open standards for representing machine learning algorithms and software tools to promote innovation and collaboration in the AI sector.

ONNX was originally named Toffee and was developed by the PyTorch team at Facebook. In September 2017 it was renamed to ONNX and announced by Facebook and Microsoft. Later, IBM, Huawei, Intel, AMD, Arm and Qualcomm announced support for the initiative.

4.1.4. ONNX RUNTIME:

ONNX Runtime is a cross-platform inference and training machine-learning accelerator.

ONNX Runtime inference can enable faster customer experiences and lower costs, supporting models from deep learning frameworks such as PyTorch and TensorFlow/Keras as well as classical machine learning libraries such as scikit-learn, LightGBM, XGBoost, etc. ONNX Runtime is compatible with different hardware, drivers, and operating systems, and provides optimal performance by leveraging hardware accelerators where applicable alongside graph optimizations and transforms.

4.2. MULTIPROCESSING IN PYTHON:

Multiprocessing in Python allows you to leverage multiple processors in your computer by running independent processes simultaneously. This is achieved through the `multiprocessing` module, which provides tools for creating and managing these processes. Each process has its own memory space and can execute tasks concurrently, improving the performance of your program for tasks that can be divided into independent chunks.

4.3. SHARED MEMORY:

`multiprocessing.SharedMemory` is a class in Python's `multiprocessing` module that facilitates sharing memory between processes. It creates a block of memory that can be accessed and modified by multiple processes simultaneously. This allows for faster data exchange

compared to traditional methods like queues or pipes, as it avoids copying data between processes.

4.4. QUEUES:

`multiprocessing.Queue` from Python's multiprocessing module acts as a FIFO (First-In-First-Out) data channel enabling communication between processes. It works like a queue where items are added using the put method and retrieved in the order they were added using the get method. This allows processes to send and receive information in a synchronized manner. `multiprocessing.Queue` is useful for coordinating work between processes, distributing tasks, and collecting results.

4.5. TKINTER:

Tkinter is a standard Python GUI (Graphical User Interface) library that provides a set of tools and widgets to create desktop applications with graphical interfaces. Tkinter is included with most Python installations, making it easily accessible for developers who want to build GUI applications without requiring additional installations or libraries.

The name “Tkinter” comes from “Tk interface “, referring to the Tk GUI toolkit that Tkinter is based on. Tkinter provides a way to create windows, buttons, labels, text boxes, and other GUI components to build interactive applications.

Tkinter is the inbuilt python module that is used to create GUI applications. It is one of the most commonly used modules for creating GUI applications in Python as it is simple and easy to work with. You don’t need to worry about the installation of the Tkinter module separately as it comes with Python already. It gives an object-oriented interface to the Tk GUI toolkit. Among all, Tkinter is most widely used

1. **Creating windows and dialog boxes:** Tkinter can be used to create windows and dialog boxes that allow users to interact with your program. These can be used to display information, gather input, or present options to the user.
2. **Building a GUI for a desktop application:** Tkinter can be used to create the interface for a desktop application, including buttons, menus, and other interactive elements.

3. **Adding a GUI to a command-line program:** Tkinter can be used to add a GUI to a command-line program, making it easier for users to interact with the program and input arguments.
4. **Creating custom widgets:** Tkinter includes a variety of built-in widgets, such as buttons, labels, and text boxes, but it also allows you to create your own custom widgets.
5. **Prototyping a GUI:** Tkinter can be used to quickly prototype a GUI, allowing you to test and iterate on different design ideas before committing to a final implementation

Tkinter is a useful tool for creating a wide variety of graphical user interfaces, including windows, dialog boxes, and custom widgets. It is particularly well-suited for building desktop applications and adding a GUI to command-line programs.

4.6. WHISPER FROM OPENAI:

Whisper, created by OpenAI, is a powerful speech recognition and translation tool. It's like a supercharged assistant that can listen to audio (in multiple languages!) and convert it to text. Not only can it understand English, but it can also transcribe speech in various languages and even translate those languages into English. Released in September 2022, Whisper is open-source, making it a valuable resource for developers and anyone who needs to work with audio recordings.

CHAPTER 5

SYSTEM DESIGN

5.1 ARCHITECTURE DIAGRAM:

An architecture diagram is a visual representation of the structure and components of a system or application. It typically shows the relationships between the different components, as well as the inputs, outputs, and interfaces of each component.

Architecture diagrams can be used to communicate the design of a system to stakeholders, such as developers, project managers, and customers. They can also be used to document the architecture of a system for future reference and to facilitate the design and development process.

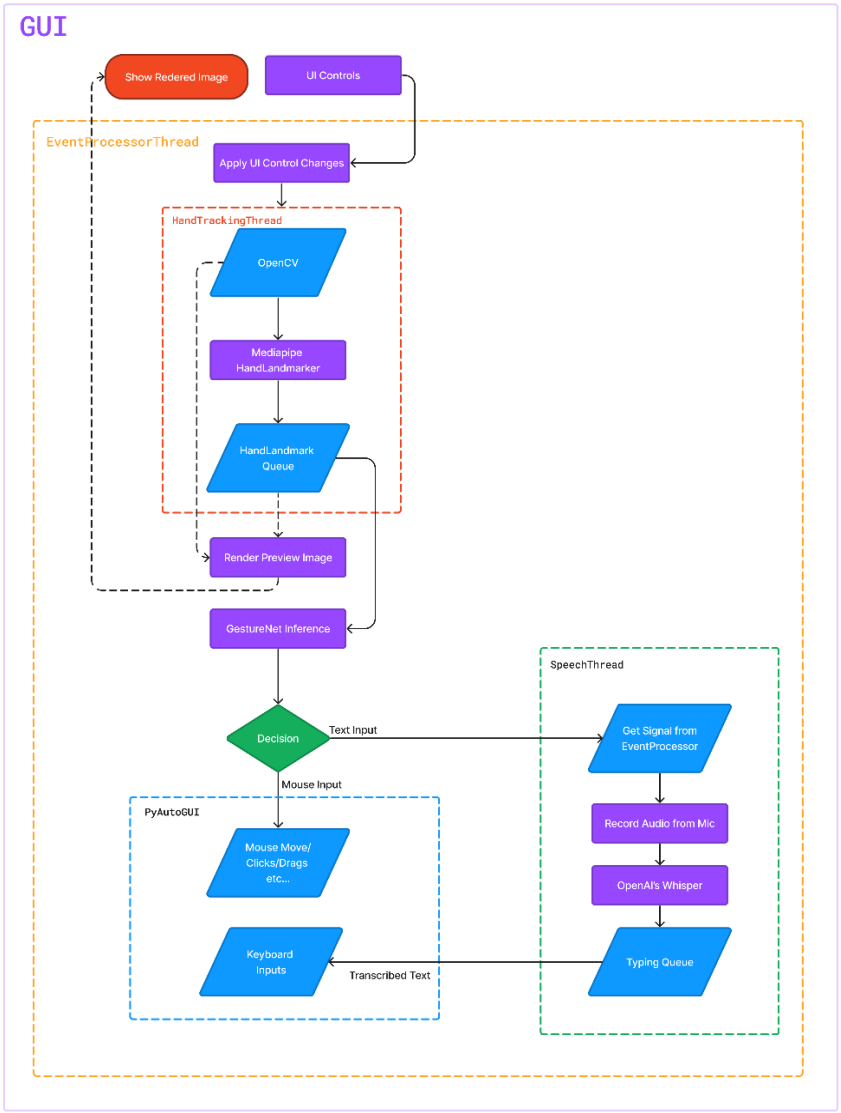


Fig 5.1 Architecture Diagram
[System Design Architecture for the proposed implementation]

5.2. WORKFLOW DIAGRAM:

A workflow diagram is a visual representation of the steps in a process or workflow. It typically shows the sequence of tasks that need to be performed, as well as the decision points and branches in the workflow.

Workflow diagrams can be used to document and communicate the steps in a process, to identify inefficiencies or bottlenecks in the workflow, and to improve the overall efficiency and effectiveness of the process.

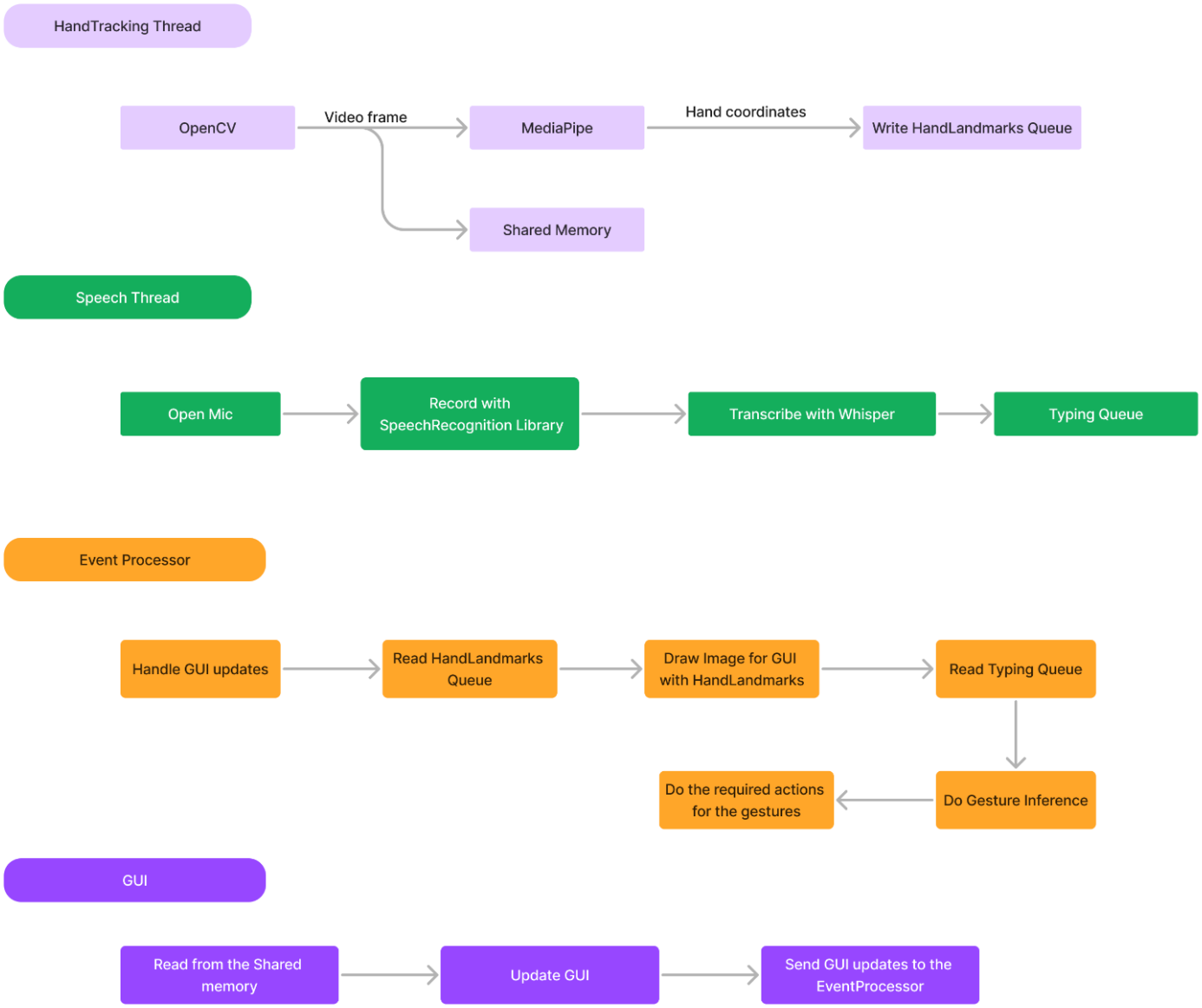


Fig 5.2 Workflow Diagram
[Working and interaction between the different components.]

5.3 UML DIAGRAM:

A UML diagram is a way to visualize systems and software using Unified Modelling Language (UML). Software engineers create UML diagrams to understand the designs, code architecture, and proposed implementation of complex software systems. UML diagrams are also used to model workflows and business processes.

5.3.1 USE-CASE DIAGRAM:

A Use Case Diagram is a type of Unified Modelling Language (UML) diagram that represents the interaction between actors (users or external systems) and a system under consideration to accomplish specific goals. It provides a high-level view of the system's functionality by illustrating the various ways users can interact with it.

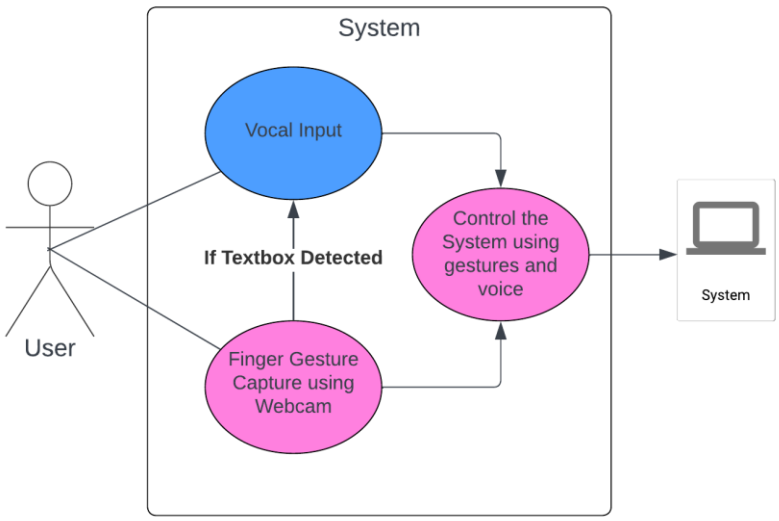


Fig 5.3 Use-Case Diagram
[Use-Case Diagram represents the interaction between actors and system]

5.3.2 CLASS DIAGRAM:

A class diagram is a type of diagram in the Unified Modelling Language (UML) that shows the structure and relationships of a system. It is used to represent the static view of a system, which is the structure of the system at a particular point in time.

A class diagram consists of:

- Class boxes, which represent the different classes in the system
- Attributes, which represent the data or properties of a class

- Operations, which represent the behaviours or methods of a class
- Relationships, which represent the connections between classes, such as inheritance, association, and aggregation

In a class diagram, classes are represented by boxes that contain the name of the class and the attributes and operations of the class. The relationships between classes are represented by lines connecting the class boxes, with arrows indicating the direction of the relationship.

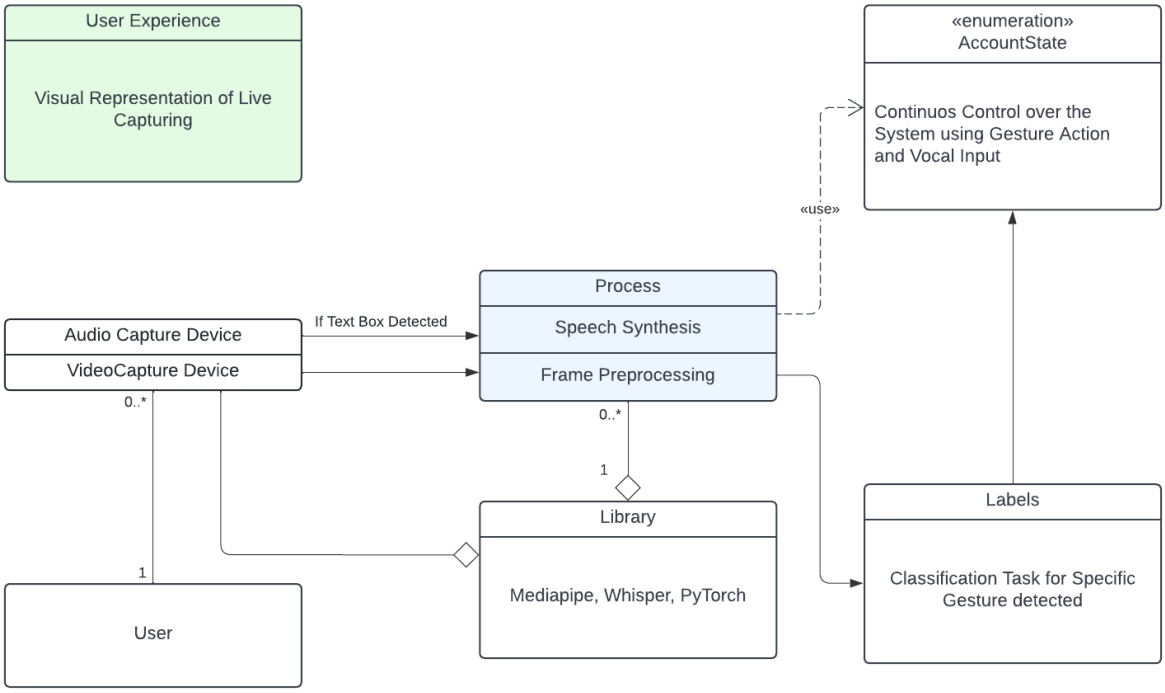


Fig 5.4 Class Diagram
[Class Diagram representing the relationship between different components]

CHAPTER 6

MODULES

This chapter dives into the individual modules that power the gesture detection and control system. It explores how MediaPipe captures hand landmarks, GestureNet classifies gestures based on extracted features, and PyAutoGUI translates gestures into real-time control actions. Additionally, it covers the optional Audio Transcription Module using Whisper for speech recognition, enabling text input and voice commands alongside gestures for a richer multimodal user experience. The chapter concludes by detailing how these modules integrate seamlessly to achieve real-time performance and customizable interaction.

6.1 GESTURE RECOGNITION MODULE

This module is responsible for classifying hand gestures captured from a user's hand.

- **MediaPipe Hand Landmarks**: This module captures real-time hand pose information from a video stream. It identifies and tracks keypoints on the hand, providing the foundation for gesture recognition.
- **Feature Extraction**: Key features are extracted from the MediaPipe hand landmarks. These features typically include:
    - Angles between specific fingers.
    - Distances between key hand points.
- **GestureNet Model**: This is a custom PyTorch model designed to classify hand gestures. GestureNet takes the extracted features as input and outputs a classification corresponding to the detected gesture (e.g., "THUMB_UP", "OPEN_PALM"). The model architecture leverages convolutional neural networks to efficiently process the spatial information within the hand landmarks. Additionally, separate layers handle the calculated angles and distances, allowing the model to learn from both geometric relationships and relative positions of hand elements.

6.2 CONTROL AND INTERACTION MODULE

The Control and Interaction Module acts as the bridge between the gesture recognition module and the user's computer. It translates the recognized gestures into real-time control actions, allowing users to interact with the computer interface using hand movements.

1. **Gesture Class Input:** This module receives the classified gesture class from the Gesture Recognition Module (Chapter 6.1). This class represents the specific hand posture the system has identified (e.g., "THUMB_UP", "OPEN_PALM").

2. **Gesture-Action Mapping:** A pre-defined mapping table associates each gesture class with a corresponding set of control actions. This mapping defines how the system should respond to each gesture. The PyAutoGUI library plays a crucial role in this process.

3. **Action Execution:** PyAutoGUI provides functionalities to simulate various user inputs like:

    - **Mouse Actions:** Clicks (left, right, double), scrolls (up, down), dragging

    - **Keyboard Actions:** Key presses, combinations (using modifiers like Ctrl, Shift)

    - **System Actions:** Volume control, copy/paste commands

4. **Action Execution with PyAutoGUI:** Based on the mapped action for the received gesture class, PyAutoGUI functions are called to perform the corresponding user input on the computer.

6.3 AUDIO TRANSCRIPTION MODULE

The Audio Transcription Module is an optional extension that equips the system with the ability to process spoken commands alongside gestures. This enhances the user experience by creating a multimodal interaction interface.

1. **Real-time Audio Capture:** The module captures audio input from the user's environment using the computer's microphone or an external device.

2. **Whisper Integration:** OpenAI's Whisper model is employed for real-time speech transcription. Whisper converts the captured audio stream into text, effectively recognizing spoken words and phrases.

3. **Text Processing (Optional):** Depending on the application's needs, this step might involve basic text processing techniques. For example, the system could remove punctuation or format the transcribed text for consistency before use.

4. **Text Input Integration:** The transcribed text is then fed into the system for further processing. There are two main approaches to utilize this text:

    - **Keyboard Input:** The system simulates keyboard typing using the transcribed text, enabling voice commands to be directly translated into on-screen actions.

    - **Command Recognition:** The system can be designed to recognize specific keywords or phrases within the transcribed text. These recognized voice commands can then trigger pre-defined actions within the application.

6.4 SYSTEM INTEGRATION

This chapter details how the individual modules described in previous sections work together to achieve the overall functionality of the hybrid deep learning system for gesture detection and control. Here, we'll explore the data flow between modules and how they collaborate to create a seamless user experience.

1. **Gesture Recognition**:

    - The system continuously captures video frames from a webcam or other video source.

    - MediaPipe within the Gesture Recognition Module (Chapter 6.1) processes each frame to identify hand landmarks and extract keypoint locations.

- Feature extraction techniques are applied to the hand landmark data, calculating critical angles between fingers and distances between keypoints.
- The GestureNet model (Chapter 6.1) takes the extracted features as input and predicts the most likely gesture class (e.g., "THUMB_UP").

2. **Control and Interaction** (Chapter 6.2):
    - The recognized gesture class from Gesture Recognition is passed to the Control and Interaction Module.
    - This module retrieves the corresponding action(s) mapped to the received gesture class from a pre-defined gesture-action table. PyAutoGUI facilitates this mapping.
    - Based on the mapped action(s), PyAutoGUI functions are called to simulate user input on the computer, such as mouse clicks, keyboard presses, or system commands.

3. **Audio Transcription** (Chapter 6.3):
    - If the Audio Transcription Module is included, the system captures audio input simultaneously with video.
    - Whisper performs real-time speech transcription, converting the audio stream into text.
    - The transcribed text can be processed further (optional) and then utilized in two main ways:
        - **Keyboard Input**: The system simulates typing the transcribed text, enabling voice commands to be translated into on-screen actions.
        - **Command Recognition**: Specific keywords or phrases within the text can trigger pre-defined actions within the application.

4. **User Interaction**:

    - The user interacts with the system through a combination of hand gestures and spoken commands (if audio transcription is enabled).
    - The system continuously monitors for gestures and audio input, processing them through the respective modules and providing real-time feedback through visual or auditory cues (optional).

CHAPTER 7

ALGORITHM AND PERFORMANCE

7.1 ALGORITHM:

1. **Read from Shared Memory (EventProcessor):** This step retrieves data from a shared memory location, likely containing information about hand landmarks detected by MediaPipe.

2. **Handle GUI changes**: This step processes any updates to the graphical user interface (GUI) of the system.

3. **Read Hand Landmarks from Queue**: This step reads hand landmark data, likely from a queue where MediaPipe or another process places it.

4. **Do gesture recognition inference**: This step performs inference on the hand landmark data using a deep learning model, likely the GestureNet model you described, to classify the hand gesture.

5. **Update GUI with gesture information (Draw Image for GUI)**: This step updates the GUI with information about the detected gesture, possibly including a visual representation of the hand and the recognized gesture class.

6. **Write gesture class to SharedMemory**: This step writes the recognized gesture class to a shared memory location, where other parts of the system can access it.

7. **Track finger movement**: This step tracks the movement of the user's hand in the video frame.

8. **Read Configuration from SharedMemory**: This step reads configuration parameters from a shared memory location.

9. **Handle Thread Events**: This step manages events triggered by different threads in the system.

10. **Update Shared Memory with HandLandmarks (EventProcessor):** This step writes the latest hand landmark data to a shared memory location, likely for use by other parts of the system.

11. **Process Speech from another thread (SpeechThread)**: This step receives a signal from the EventProcessor thread, likely indicating that speech data is available for processing.

12. **Record and transcribe speech with Whisper (record with recognizer, transcribe)**: This step records audio data from the microphone and uses Whisper to transcribe it into text.

13. **Type the transcribed text (typing with Whisper)**: This step types the transcribed text into the computer using a library like PyAutoGUI, possibly adding it to a queue for ordered processing.

14. **Do required actions for the gesture**: This step performs actions on the computer based on the recognized gesture class, likely using PyAutoGUI to simulate mouse clicks, keyboard presses, or other system interactions.

7.2 PERFORMANCE EVALUATION:

This introductory section briefly outlines the purpose of the performance evaluation. It highlights the importance of measuring the system's accuracy, robustness, and efficiency. Additionally, it can mention the specific metrics that will be used in the evaluation.

7.2.1 Webcam Resolution and FPS

This section focuses on evaluating the impact of webcam resolution and frame rate (FPS) on the system's performance.

- **Accuracy**: How do different webcam resolutions (e.g., 720p, 1080p) affect the accuracy of gesture recognition? Does higher resolution lead to more accurate classification?

- **Latency**: How does FPS impact the latency between a user performing a gesture and the system's response? Lower FPS might introduce delays in processing.

- **Resource Consumption**: Explore how varying webcam resolutions and FPS affect the computational resources required by the system. Higher resolution and FPS might consume more processing power.

This analysis helps determine the optimal balance between image quality, processing speed, and resource utilization for a smooth user experience.

7.2.2 GestureNet CrossEntropyLoss

This section dives into the evaluation of the GestureNet model's performance. It focuses on a specific metric called CrossEntropyLoss.

- **CrossEntropyLoss**: This loss function measures the difference between the model's predicted gesture probabilities and the true gesture class. Lower CrossEntropyLoss indicates better model performance, signifying the model's ability to accurately distinguish between different gestures.

- **Training Loss**: Present the CrossEntropyLoss values during the training process. Did the loss decrease steadily, indicating successful model learning?

- **Validation Loss**: Evaluate the CrossEntropyLoss on a separate validation dataset not used for training. This helps assess how well the model generalizes to unseen data.

CHAPTER 8

CODING AND TESTING

8.1 CODING:

Coding in Jupyter Notebook involves writing and executing code in cells within a Jupyter Notebook document.

To create a new cell in Jupyter Notebook, you can use the "**Insert**" menu in the toolbar, or you can use the keyboard shortcut `` `Shift+Enter` ``. By default, new cells are created as code cells, which can contain code that can be executed.

To execute a code cell, you can click the "**Run**" button in the toolbar, or you can use the keyboard shortcut `` `Shift+Enter` ``. When you execute a code cell, the code is run and any output is displayed below the cell.

You can also create cells that contain text written in the Markdown formatting language, which can be used to add explanatory text, headings, lists, and other formatting to the document. To create a Markdown cell, you can use the "**Cell**" menu in the toolbar and select "**Markdown**" from the "**Cell Type**" submenu, or you can use the keyboard shortcut `` `Esc+M` ``. Markdown cells are not executed, but rather are rendered as formatted text when the document is viewed.

In addition to code and text, Jupyter Notebook cells can also contain math written in LaTeX syntax, which is rendered as formatted math when the document is viewed. To create a cell containing math, you can use the "**Cell**" menu in the toolbar and select "**Markdown**" from the "**Cell Type**" submenu, or you can use the keyboard shortcut `` `Esc+M` ``, and then enter the math in LaTeX syntax between dollar signs `($)`.

Overall, Jupyter Notebook provides a powerful and convenient environment for writing and executing code, as well as for documenting and communicating your work.

8.2 CODING STANDARDS:

Coding standards are guidelines for writing and formatting code in a consistent and readable manner. They are used to improve the quality and maintainability of code, and to help ensure that code is easy to understand and modify by other developers.
Coding standards can cover a wide range of topics, including:

- **Naming conventions**: This includes guidelines for naming variables, functions, and other code elements in a consistent and descriptive manner.

- **Formatting conventions**: This includes guidelines for indentation, spacing, and other formatting issues that affect the readability of code.
- **Documentation standards**: This includes guidelines for documenting code, including the use of comments and documentation strings.
- **Coding style**: This includes guidelines for writing code in a clear and concise manner, and avoiding common coding pitfalls such as using global variables or writing overly complex code.
- **Testing standards**: This includes guidelines for writing and organizing tests, and for ensuring that code is thoroughly tested before it is released.

Coding standards can be specific to a particular programming language or project, and they may be enforced through automated tools such as linters and code formatting tools. Adhering to coding standards can help improve the quality and maintainability of code, and can also help to reduce the time and effort required to review and modify code.

8.3 TESTING PROCEDURE:

Testing is the process of evaluating a system or its component(s) with the intent to find whether it satisfies the specified requirements or not. Testing is an essential part of the software development process, as it helps to ensure that the system is of high quality and is fit for its intended purpose.

Here is a general outline of the testing procedure:

1. **Identify the requirements**: The first step in the testing process is to identify the requirements that the system must meet. This may involve reviewing the design and specification documents, as well as gathering input from stakeholders such as users, developers, and project managers.
2. **Plan the tests**: Based on the requirements, you can plan the tests that will be needed to evaluate the system. This may involve creating test cases, identifying the test data that will be used, and establishing the criteria for success.
3. **Set up the test environment**: Before you can begin testing the system, you need to set up the test environment, which includes installing and configuring the necessary software and hardware.
4. **Execute the tests**: Once the test environment is set up, you can begin executing the tests. This may involve running manual tests, automated tests, or a combination of both.

5. **Analyse the results**: After the tests are completed, you need to analyse the results to determine whether the system meets the specified requirements. This may involve reviewing the test logs, debugging any failed.

6. **Report the results**: After the tests are completed and the results are analysed, you need to prepare a report that summarizes the findings. This report should include an overview of the tests that were conducted, the results of the tests, and any conclusions that can be drawn from the results.

7. **Fix any issues**: If the tests reveal issues with the system, you need to fix them. This may involve modifying the code, updating the documentation, or performing additional testing.

Overall, the testing procedure is an important part of the software development process, as it helps to ensure that the system is of high quality and is fit for its intended purpose. By following a systematic testing procedure, you can identify and fix issues with the system before it is released, which can save time and resources in the long run.

8.4 UNIT TESTING:

Unit testing is a type of software testing that involves testing individual units or components of a system in isolation from the rest of the system. A unit is the smallest testable part of a system, and it is typically a function, method, or class in the code.

The goal of unit testing is to validate that each unit of the system is working correctly and meets the specified requirements. Unit tests are usually automated, and they are run as part of the build process to ensure that the code is correct and ready for integration.

To perform unit testing, you need to write test cases that exercise the units of the code and verify that they are working correctly. A test case consists of test inputs, the expected output, and any other necessary information, such as prerequisites or clean up steps.

Unit tests are typically run using a unit testing framework, which is a tool that helps to automate the testing process. The unit testing framework provides features such as assertion functions, test runners, and test result reporting.

Overall, unit testing is an important part of the software development process, as it helps to ensure the quality and reliability of the code. By writing and running unit tests, you can catch and fix issues early in the development process, which can save time and resources in the long run.

8.5 PERFORMANCE EVALUATION:

Performance testing is the process of evaluating the performance of a system or its component(s) under a specific workload. The goal of performance testing is to determine how the system behaves under normal and peak load conditions, and to identify any bottlenecks or other issues that may impact its performance.

Performance testing can be used to evaluate a wide range of characteristics of a system, including:

- **Speed**: This includes measures such as response time and throughput, which reflect how quickly the system processes requests and generates output.
- **Scalability**: This refers to the ability of the system to handle an increasing workload without a decrease in performance.
- **Stability**: This refers to the ability of the system to maintain consistent performance over a long period of time.

To perform performance testing, you need to define the workload that the system will be subjected to, and the performance criteria that you want to measure. You can then use a performance testing tool to simulate the workload and measure the performance of the system.

Performance testing is an important part of the software development process, as it helps to ensure that the system can handle the expected workload and deliver acceptable performance to users. By identifying and addressing performance issues early in the development process, you can save time and resources in the long run.

CHAPTER 9

CONCLUSION AND FUTURE ENHANCEMENTS

9.1 CONCLUSION:

In conclusion, the real-time gesture recognition system developed in this project offers an intuitive and natural interface for users to interact with computers using hand gestures. While the current system demonstrates its capabilities, future enhancements highlight the vast potential for advancements in this field, paving the way for seamless and natural human-computer interactions.

9.2 FUTURE ENHANCEMENTS:

The real-time gesture recognition system developed in this project serves as a solid foundation for further advancements and enhancements. Here are some potential future enhancements that could be explored:

- **Expanded Gesture Repertoire:** While the current system recognizes a set of predefined gestures, expanding the repertoire of recognizable gestures would broaden the applications and versatility of the system. This could involve incorporating more complex hand and finger movements, as well as integrating multi-hand gestures for advanced interactions.

- **Adaptive and Personalized Recognition:** Incorporating adaptive and personalized recognition techniques could improve the system's accuracy and user experience. By learning and adapting to individual users' gestural patterns and preferences, the system could provide a tailored experience, reducing recognition errors and enhancing overall usability.

- **Cross-Platform Compatibility:** Extending the system's compatibility across various platforms and devices, such as smartphones, tablets, and augmented/virtual reality environments, could broaden its applicability and reach. This would enable users to interact with a wide range of computing devices using the same intuitive gesture-based interface.

- **Real-World Applications and User Studies:** Conducting user studies and deploying the system in real-world scenarios could provide valuable insights into its practicality, usability, and effectiveness. Feedback from users in diverse settings could drive further improvements and refinements, ensuring that the system meets the evolving needs and preferences of its target audience.

CHAPTER 10

REFERENCES

- A.S.M. Miah, J. Shin, M.A.M. Hasan, Y. Okuyama, and A. Nobuyoshi, "Dynamic hand gesture recognition using effective feature extraction and attention based deep neural network," Proc. IEEE 16th Int. Sympos. Embedded Multicore/Many-core Systems-on-Chip, 2023, pp. 241–247.

- S. M. Miah, M. A. M. Hasan, J. Shin, Y. Okuyama, and Y. Tomioka, "Multistage spatial attention-based neural network for hand gesture recognition," Computers, vol. 12, no. 1, 2023, Art. no. 13.

- S. M. Miah, J. Shin, M. A. M. Hasan, and M. A. Rahim, "Bensignnet: Bengali sign language alphabet recognition using concatenated segmentation and convolutional neural network," Appl. Sci., vol. 12, no. 8, 2022, Art. no. 3933.

- J. Shin, M. A. M. Hasan, A. S. M. Miah, K. Suzuki, and K. Hirooka, "Japanese sign language recognition by combining joint skeleton-based handcrafted and pixel-based deep learning features with machine learning classification," Comput. Model. Eng. Sci., pp. 1–21, 2024. [Online]. Available: https://www.techscience.com/CMES/online/detail/19824

- Z. Ren, J. Yuan, J. Meng, and Z. Zhang, "Robust part-based hand gesture recognition using kinect sensor," IEEE Trans. Multimedia, vol. 15, pp. 1110–1120, 2013

- G. Yuan, X. Liu, Q. Yan, S. Qiao, Z. Wang, and L. Yuan, "Hand gesture recognition using deep feature fusion network based on wearable sensors," IEEE Sensors J., vol. 21, no. 1, pp. 539–547, Jan. 2021.

- A. Barbhuiya, R. K. Karsh, and S. Dutta, "Alexnet-CNN based feature extraction and classification of multiclass ASL hand gestures," in Proc. 5th Int. Conf. Microelectronics, Comput. Commun. Syst., 2021, pp. 77–89.

CHAPTER 11

APPENDICES

11.1 SOURCE CODE:

Client-Side:

gesture_network.py

```
import torch
import torch.nn as nn
import torch.nn.functional as F

from typin import HAND_LANDMARK_ANGLES, HAND_LANDMARK_DISTANCES

HAND_LANDMARK_DISTANCES = torch.tensor(HAND_LANDMARK_DISTANCES)
HAND_LANDMARK_ANGLES = torch.tensor(HAND_LANDMARK_ANGLES)


class GestureNet(nn.Module):
    def __init__(self, hidden_size: int, output_size: int):
        super(GestureNet, self).__init__()
        n_dim = 3
        self.coord_conv1 = nn.Conv1d(in_channels=n_dim,
out_channels=hidden_size // 8,
                                     kernel_size=n_dim, stride=1,
padding=1)
        self.coord_conv2 = nn.Conv1d(in_channels=hidden_size // 8,
out_channels=hidden_size // 4,
                                     kernel_size=n_dim, stride=1,
padding=1)
        self.coord_conv3 = nn.Conv1d(in_channels=hidden_size // 4,
out_channels=hidden_size // 2,
                                     kernel_size=n_dim, stride=1,
padding=1)
        self.pooling = nn.MaxPool1d(kernel_size=2)
        self.coord_norm = nn.BatchNorm1d(hidden_size)

        self.rad_proj = nn.Linear(len(HAND_LANDMARK_ANGLES),
hidden_size)
        self.rad_norm = nn.LayerNorm(hidden_size)

        self.dist_proj = nn.Linear(len(HAND_LANDMARK_DISTANCES),
hidden_size)
        self.dist_norm = nn.LayerNorm(hidden_size)
        self.dropout = nn.Dropout(p=0.2)
        self.down_proj = nn.Linear(hidden_size, output_size)
```

```
        self.act = F.leaky_relu

    @staticmethod
    def get_rad(landmarks: torch.Tensor) -> torch.Tensor:
        ba = landmarks[:, HAND_LANDMARK_ANGLES[:, 0]] - landmarks[:,
HAND_LANDMARK_ANGLES[:, 1]]
        bc = landmarks[:, HAND_LANDMARK_ANGLES[:, 2]] - landmarks[:,
HAND_LANDMARK_ANGLES[:, 1]]
        dot_product = torch.sum(ba * bc, dim=-1)
        norm_ba = torch.linalg.norm(ba, dim=-1)
        norm_bc = torch.linalg.norm(bc, dim=-1)
        radians = torch.acos(dot_product / (norm_ba * norm_bc))
        return radians

    @staticmethod
    def get_dist(landmarks: torch.Tensor) -> torch.Tensor:
        """
        Returns the distances between the landmarks.
        """
        dists = torch.linalg.norm(
            landmarks[:, HAND_LANDMARK_DISTANCES[:, 0]] -
landmarks[:, HAND_LANDMARK_DISTANCES[:, 1]], dim=-1)
        return dists

    def convoluted_forward(self, x: torch.Tensor) -> torch.Tensor:
        """
        Some convoluted shit is going on here...
        """
        x = x.permute(0, 2, 1)
        x = self.pooling(self.act(self.coord_conv1(x)))
        x = self.pooling(self.act(self.coord_conv2(x)))
        x = self.pooling(self.act(self.coord_conv3(x)))
        return x.view(x.size(0), -1)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        rad =
self.rad_norm(self.act(self.rad_proj(self.get_rad(x))))
        dist =
self.dist_norm(self.act(self.dist_proj(self.get_dist(x))))
        cp = self.coord_norm(self.convoluted_forward(x))
        x = self.dropout(cp + rad + dist)
        x = self.down_proj(x)
        return x
```

event_processor.py

```
import multiprocessing
import os
```

```
import time
from multiprocessing import Queue
from multiprocessing.shared_memory import SharedMemory
from typing import Optional, Tuple

import cv2
import numpy as np
import pyautogui

from constants import HEIGHT, WIDTH, EMPTY_FRAME,
DEFAULT_TRACKING_SMOOTHNESS, DEFAULT_MOUSE_SMOOTHNESS, \
    DEFAULT_SHOW_WEBCAM, DEFAULT_POINTER_SOURCE
from gesture_detector import GestureDetectorProMax
from hand import Hand
from hand_tracking import HandTrackingThread
from speech import SpeechThread
from typin import HandLandmark, HandEvent, GUIEvents
from utils import draw_landmarks_on_image, get_volume_linux,
adjust_volume_linux

pyautogui.FAILSAFE = False
SCREEN_WIDTH, SCREEN_HEIGHT = pyautogui.size()


class EventProcessor(multiprocessing.Process):
    _iteration_delay = 1 / 60

    def __init__(self, gui_event_queue: Queue, tracking_image_name:
str):
        super().__init__()
        self.gui_event_queue = gui_event_queue
        self.tracking_image = SharedMemory(tracking_image_name)
        self.show_webcam = DEFAULT_SHOW_WEBCAM

        # Mouse Control
        self.is_mouse_button_down = False
        self.last_click_time = time.time()
        self.screen_width, self.screen_height = None, None
        self.mouse_smoothness_alpha = DEFAULT_MOUSE_SMOOTHNESS
        self.prev_coords = None

        self.current_event = HandEvent.MOUSE_NO_EVENT

        self.hand = Hand(enable_smoothing=True, axis_dim=3,
smoothness=DEFAULT_TRACKING_SMOOTHNESS)

        self.gesture_detector = None
        self.current_pointer_source: HandLandmark =
DEFAULT_POINTER_SOURCE
```

```
        # Hand Tracking Thread
        self._last_video_frame_time = time.time()
        self.video_frame_shared = SharedMemory(create=True,
size=EMPTY_FRAME.nbytes)
        self.hand_landmarks_queue = Queue(maxsize=3)
        self.tracking_thread = None

        # Audio Transcription
        self.audio_thread_communication_queue = Queue(maxsize=1)
        self.typewriter_queue = Queue(maxsize=1)
        self.audio_thread = None

    def initialize_threads(self):
        start = time.time()
        self.tracking_thread =
HandTrackingThread(landmark_queue=self.hand_landmarks_queue,

video_frame_name=self.video_frame_shared.name)
        self.tracking_thread.start()
        print(f"Hand tracking thread started in {time.time() -
start:.2f} seconds")

        start = time.time()
        self.audio_thread =
SpeechThread(signal_queue=self.audio_thread_communication_queue,

typewriter_queue=self.typewriter_queue)
        self.audio_thread.start()
        print(f"Speech thread started in {time.time() - start:.2f}
seconds")

        self.gesture_detector = GestureDetectorProMax(self.hand,

model_path='./models/gesture_model.onnx',

labels_path='./gesture_rec/choices.txt')

    def terminate(self):
        if self.tracking_thread is not None:
            print("Terminating tracking thread")
            self.tracking_thread.terminate()
        if self.audio_thread is not None:
            print("Terminating audio thread")
            self.audio_thread.terminate()
        self.video_frame_shared.close()
        self.video_frame_shared.unlink()
        print("Terminating event processor")
        exit(0)
```

```
    @property
    def pointer_coordinates(self) -> Optional[Tuple[np.ndarray,
np.ndarray]]:
        # coordinates of previous and current frame
        if self.hand.coordinates is None:
            self.prev_coords = None
            return None
        curr_coords =
self.hand.coordinates_of(self.current_pointer_source)
        if self.prev_coords is None:
            self.prev_coords = curr_coords
            return None
        if np.array_equal(curr_coords, self.prev_coords):
            return None
        out = (self.prev_coords, curr_coords)
        self.prev_coords = curr_coords
        return out

    def update_tracking_frame(self):
        if self.show_webcam:
            frame = np.ndarray((HEIGHT, WIDTH, 3), dtype=np.uint8,
buffer=self.video_frame_shared.buf)
            _last_video_frame = frame.copy()
        else:
            _last_video_frame = EMPTY_FRAME.copy()
        new_time = time.time()
        fps = 1 / (new_time - self._last_video_frame_time)
        self._last_video_frame_time = new_time

        if self.hand.coordinates_2d is not None:
            frame = draw_landmarks_on_image(_last_video_frame,
self.hand.coordinates_2d)
            cv2.putText(frame, f"Event: {self.current_event.name}",
(10, 50), cv2.FONT_HERSHEY_SIMPLEX,
                        1, (255, 255, 255), 2)
            if fps is not None:
                cv2.putText(frame, f"FPS: {fps:.2f}", (10, 100),
cv2.FONT_HERSHEY_SIMPLEX,
                            1, (255, 255, 255), 2)
            self.tracking_image.buf[:frame.nbytes] = frame.tobytes()
        else:
            self.tracking_image.buf[:_last_video_frame.nbytes] =
_last_video_frame.tobytes()

    def do_mouse_movement(self, ):
        coords = self.pointer_coordinates
        if coords is None:
            return
```

```
        prev_coords, current_coords = coords
        prev_x, prev_y, prev_z = prev_coords
        current_x, current_y, current_z = current_coords
        depth_mul = np.interp(-current_z, (0.01, 0.2), (60, 70))

        # Smooth the mouse movement
        alpha = self.mouse_smoothness_alpha
        x_smoothed = prev_x * (1 - alpha) + current_x * alpha
        y_smoothed = prev_y * (1 - alpha) + current_y * alpha

        distance = ((x_smoothed - prev_x) ** 2 + (y_smoothed -
prev_y) ** 2) ** .5
        if distance < 1e-3:
            return
        multiplier = max(distance * depth_mul, 1.)

        dx = (x_smoothed - prev_x) * multiplier
        dy = (y_smoothed - prev_y) * multiplier

        # Calculate new coordinates
        current_x, current_y = pyautogui.position()
        new_x = current_x + dx * SCREEN_WIDTH
        new_y = current_y + dy * SCREEN_HEIGHT

        if 0 <= new_x <= SCREEN_WIDTH and 0 <= new_y <=
SCREEN_HEIGHT:
            pyautogui.moveTo(int(new_x), int(new_y), _pause=False)

    def pinch_scroll(self):
        coords = self.pointer_coordinates
        if coords is None:
            return
        prev_coords, current_coords = coords
        prev_x, prev_y, prev_z = prev_coords
        current_x, current_y, current_z = current_coords

        # Smooth the mouse movement
        alpha = self.mouse_smoothness_alpha
        x_smoothed = prev_x * (1 - alpha) + current_x * alpha
        y_smoothed = prev_y * (1 - alpha) + current_y * alpha

        distance = ((x_smoothed - prev_x) ** 2 + (y_smoothed -
prev_y) ** 2) ** .5
        if distance < 1e-3:
            return

        dx: float = (x_smoothed - prev_x) * 100
        dy: float = (y_smoothed - prev_y) * 100
```

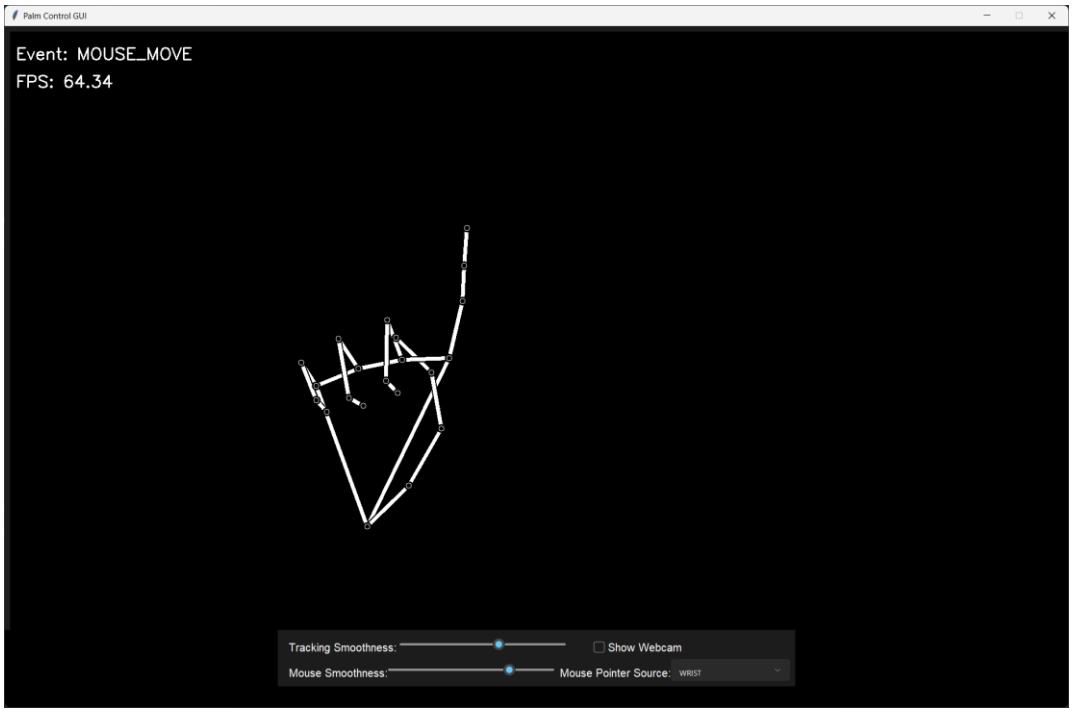
```
        if os.name == "nt":
            if abs(dx) > abs(dy):
                dx = np.interp(dx, (-5, 5), (-1000, 1000)).item()
                with pyautogui.hold('shift', _pause=False):
                    pyautogui.scroll(int(dx), _pause=False)
            else:
                dy = np.interp(dy, (-5, 5), (-1000, 1000)).item()
                pyautogui.scroll(int(dy), _pause=False)
        else:
            if abs(dx) > abs(dy):  # horizontal scroll
                pyautogui.hscroll(-dx, _pause=False)
            else:  # vertical scroll
                dy = np.interp(dy, (-5, 5), (-2.5, 2.5)).item()  #
vertical scroll is funky without this
                pyautogui.vscroll(dy, _pause=False)

    def allow_click(self):
        if time.time() - self.last_click_time > 0.5:
            self.last_click_time = time.time()
            return True
        return False

    def enable_mouse_drag(self):
        if not self.is_mouse_button_down:
            self.is_mouse_button_down = True
            pyautogui.mouseDown(button='left', _pause=False)

    def disable_mouse_drag(self):
        if self.is_mouse_button_down:
            self.is_mouse_button_down = False
            pyautogui.mouseUp(button='left', _pause=False)

    def do_lmb_click(self):
        if self.allow_click():
            pyautogui.leftClick(_pause=False)

    def do_rmb_click(self):
        if self.allow_click():
            pyautogui.rightClick(_pause=False)

    def do_copy_text(self):
        if self.allow_click():
            pyautogui.hotkey("ctrl", "c")

    def do_paste_text(self):
        if self.allow_click():
            pyautogui.hotkey("ctrl", "v")

    def increase_volume(self):
```

```
        if self.allow_click():
            if os.name == "nt":
                pyautogui.press("volumeup", _pause=False)
            else:
                if get_volume_linux() < 100:
                    adjust_volume_linux(5)

    def decrease_volume(self):
        if self.allow_click():
            if os.name == "nt":
                pyautogui.press("volumedown", _pause=False)
            else:
                if get_volume_linux() > 0:
                    adjust_volume_linux(-5)

    def handle_gui_events(self):
        while not self.gui_event_queue.empty():
            item = self.gui_event_queue.get()
            if isinstance(item, tuple):
                event, value = item
            else:
                event, value = item, None
            match event:
                case GUIEvents.EXIT:
                    self.terminate()
                case GUIEvents.SHOW_WEBCAM:
                    assert isinstance(value, bool)
                    self.show_webcam = value
                case GUIEvents.TRACKING_SMOOTHNESS:
                    assert isinstance(value, float)
                    self.hand.set_filterQ(value)
                case GUIEvents.MOUSE_SMOOTHNESS:
                    assert isinstance(value, float)
                    self.mouse_smoothness_alpha = value
                case GUIEvents.MOUSE_POINTER:
                    if isinstance(value, str):
                        value = HandLandmark[value]
                    assert isinstance(value, HandLandmark)
                    self.current_pointer_source = value
                case _:
                    pass

    def do_typing(self):
        while not self.typewriter_queue.empty():
            pyautogui.write(self.typewriter_queue.get_nowait(),
_pause=False)  # Write the text to the active window

    def update_hand_landmarks(self):
        while not self.hand_landmarks_queue.empty():
```

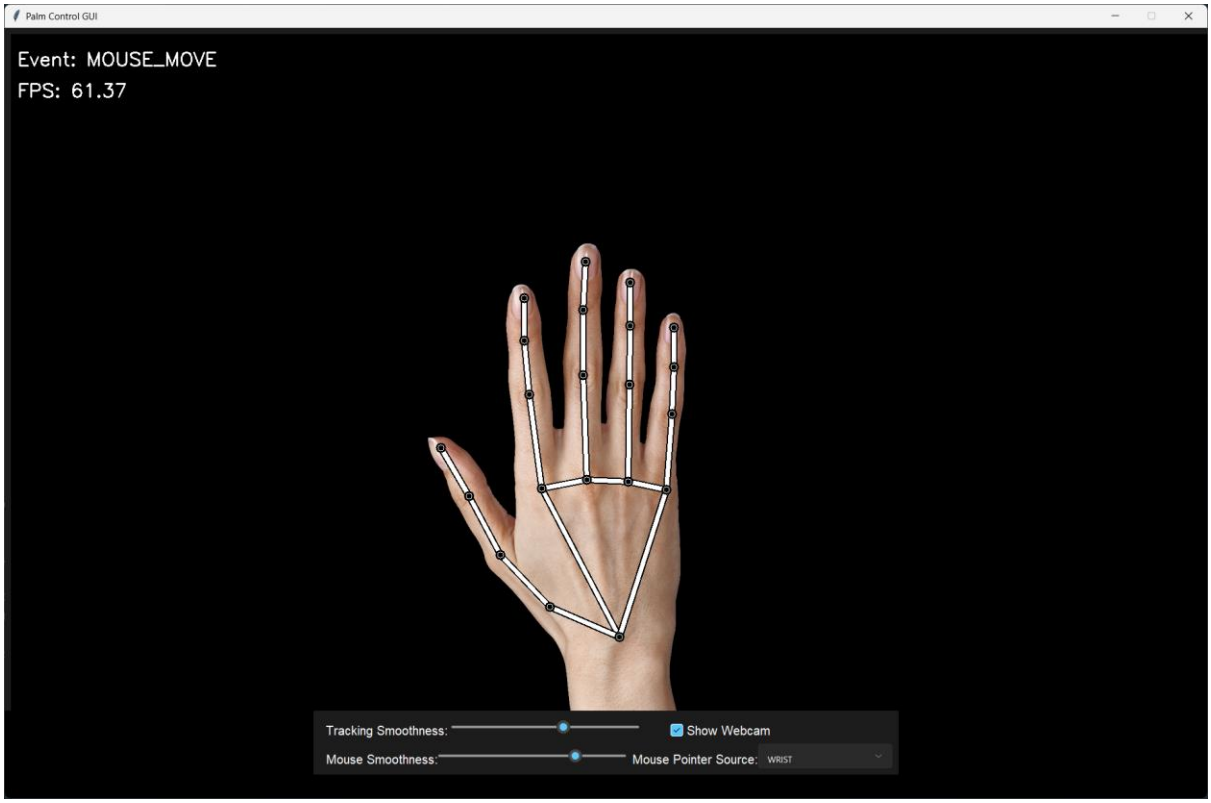
```
            self.hand.update(self.hand_landmarks_queue.get())  #
Update the hand landmarks from the queue

    def run(self):
        print(f"{self.__class__.__name__}'s PID: {os.getpid()}")
        try:
            self.initialize_threads()
        except AssertionError as e:
            print(e)
            self.terminate()
        start_time = time.time()
        try:
            while True:
                self.handle_gui_events()
                self.update_hand_landmarks()
                self.update_tracking_frame()
                self.do_typing()

                if not self.hand.is_missing:
                    # Detect the current event
                    self.current_event =
self.gesture_detector.detect()

                    if self.current_event != HandEvent.MOUSE_DRAG
and self.is_mouse_button_down:
                        self.disable_mouse_drag()
                    match self.current_event:
                        case HandEvent.MOUSE_DRAG:
                            self.enable_mouse_drag()
                            self.do_mouse_movement()
                        case HandEvent.MOUSE_CLICK:
                            self.do_lmb_click()
                            self.do_mouse_movement()
                        case HandEvent.MOUSE_RIGHT_CLICK:
                            self.do_rmb_click()
                        case HandEvent.AUDIO_INPUT:
                            if
self.audio_thread_communication_queue.empty():

self.audio_thread_communication_queue.put_nowait(True)
                        case HandEvent.MOUSE_MOVE:
                            self.do_mouse_movement()
                        case HandEvent.MOUSE_SCROLL:
                            self.pinch_scroll()
                        case HandEvent.VOLUME_UP:
                            self.increase_volume()
                        case HandEvent.VOLUME_DOWN:
                            self.decrease_volume()
                        case HandEvent.COPY_TEXT:
```

```
                        self.do_copy_text()
                    case HandEvent.PASTE_TEXT:
                        self.do_paste_text()
                    case _:
                        self.prev_coords = None
            else:
                self.current_event = HandEvent.MOUSE_NO_EVENT
                self.disable_mouse_drag()

            elapsed_time = time.time() - start_time
            remaining_time = max(self._iteration_delay -
elapsed_time, 0)
            time.sleep(remaining_time)
            start_time = time.time()
        except KeyboardInterrupt:
            self.terminate()
```

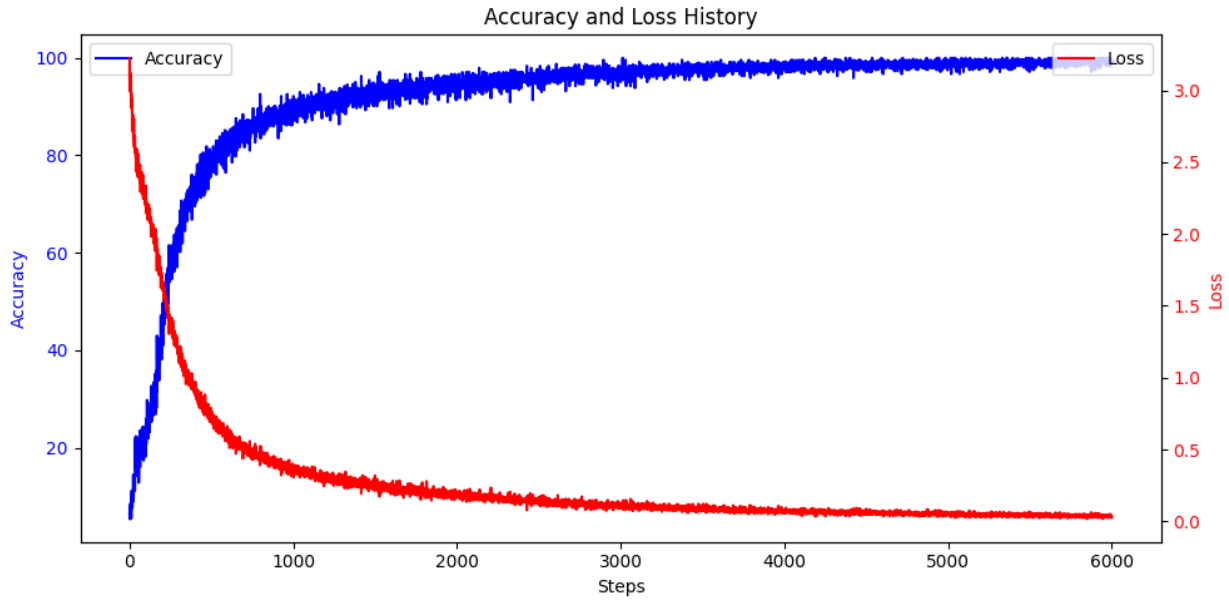
11.2 OUTPUT:



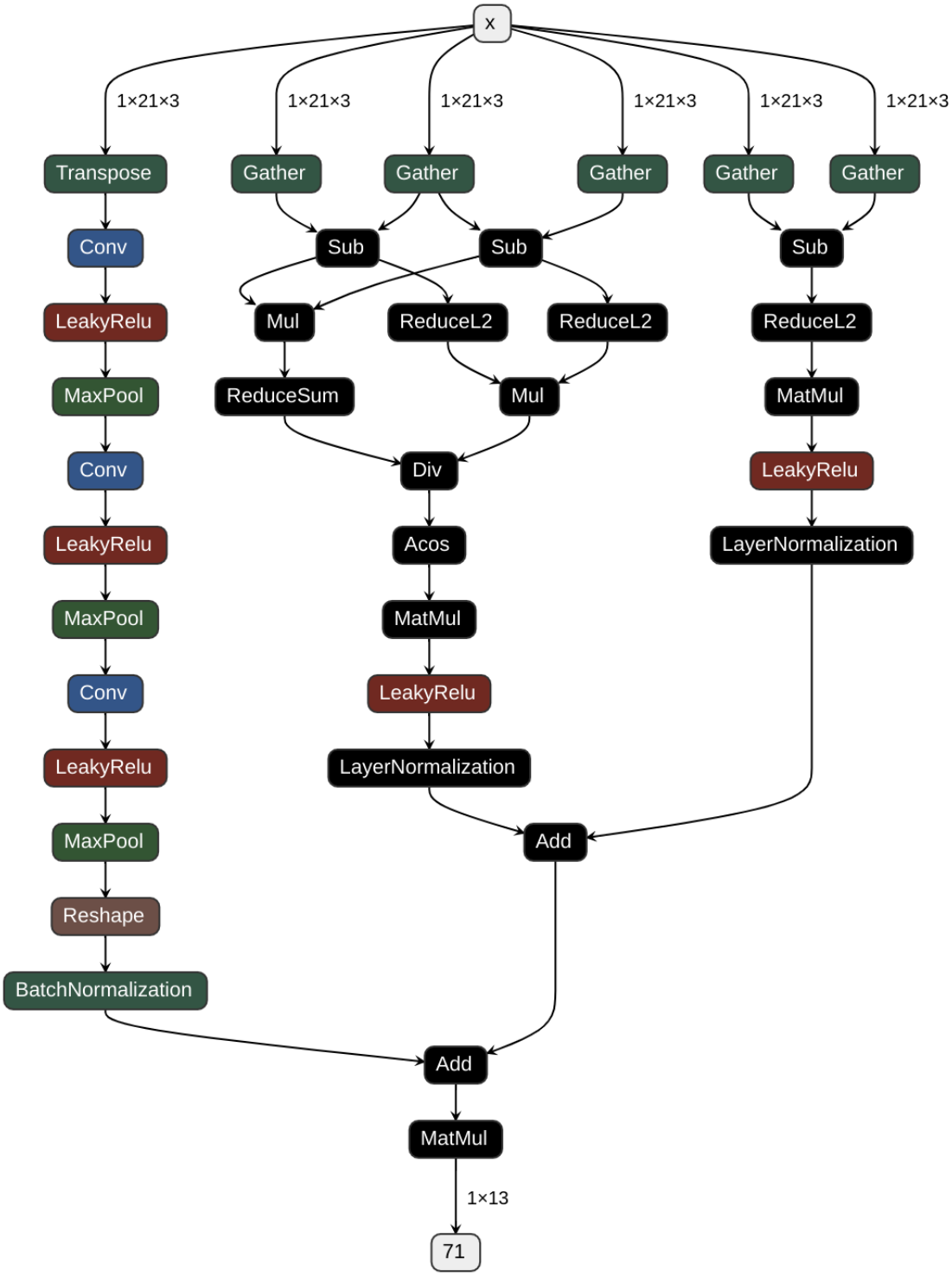
11.2.1. User Interface of Palm Control (1)



11.2.2. User Interface of Palm Control (2)



11.2.3. GestureNet Training History



11.2.4. GestureNet Architecture