22AIE304 : Deep Learning
GESTURE BASED
NOTES MAKING APPLICATION

A Thesis Submitted by

BATCH B : GROUP 13

P. Guna Vardhan [CB.EN.U4AIE22145]

Dheera Vikyath K [CB.EN.U4AIE22115]

J.P.N Sree Santh [CB.EN.U4AIE22125]

Y. Sudheer Kumar Chowdary [CB.EN.U4AIE22151]

in partial fulfillment for the award of the degree of

BACHELOR OF TECHNOLOGY
IN

CSE(AI)

Centre for Computational Engineering and Networking

AMRITA SCHOOL OF ARTIFICAL INTELLIGENCE

AMRITA VISHWA VIDYAPEETHAM

COIMBATORE – 641112 (INDIA)

NOVEMBER - 2024

AMRITA SCHOOL OF ARTIFICIAL INTELLIGENCE

AMRITA VISHWA VIDYAPEETHAM

COIMBATORE - 641112

BONAFIDE CERTIFICATE

This is to certify that the report entitled "GESTURE BASED NOTES MAKING APPLICATION" submitted by PULAGAM GUNA VARDHAN (CB.EN.U4AIE22145), DHEERA VIKYATH K (CB.EN.U4AIE22115), J.P.N SREE SANTH (CB.EN.U4AIE22125), Y. SUDHEER KUMAR CHOWDARY (CB.EN.U4AIE22151) for the award of the Degree of Bachelor of Technology in the "CSE(AI) " is a bonafide record of the work carried out by her under our guidance and supervision at Amrita School of Artificial Intelligence, Coimbatore.

Dr. Mithun Kumar
Project Guide

Dr. K.P.Soman

Professor and Head CEN

Submitted for the university examination held on 14/11/2024

ACKNOWLEDGEMENT

We would like to express our special thanks of gratitude to our teacher (Dr. Mithun Kumar Sir), who gave us the golden opportunity to do this wonderful project on the topic ("GESTURE BASED NOTES MAKING APPLICATION"), which also helped us in doing a lot of Research and we came to know about so many new things. We are thankful for the opportunity given. We would also like to thank our group members, as without their cooperation, we would not have been able to complete the project within the prescribed time.

AMRITA SCHOOL OF ARTIFICIAL INTELLIGENCE

AMRITA VISHWA VIDYAPEETHAM

COIMBATORE - 641112

DECLARATION

I, PULAGAM GUNA VARDHAN (CB.EN.U4AIE22145), DHEERA VIKYATH K (CB.EN.U4AIE22115), J.P.N SREE SANTH (CB.EN.U4AIE22125), Y. SUDHEER KUMAR CHOWDARY (CB.EN.U4AIE22151) hereby declare that this thesis entitled "GESTURE BASED NOTES MAKING APPLICATION", is the record of the original work done by me under the guidance of Dr. Mithun Kumar, Assistant Professor, Centre for Computational Engineering and Networking, Amrita School of Artificial Intelligence, Coimbatore. To the best of my knowledge, this work has not formed the basis for the award of any degree/diploma/ associate ship/fellowship/or a similar award to any candidate in any University.

Place: Coimbatore
Date: 14-11-2024

Name	Roll number	Signature of Students
Pulagam Guna Vardhan	CB.EN.U4AE22145	
Dheera Vikyath	CB.EN.U4AE22115	
J.P.S.N Sree Santh	CB.EN.U4AE22125	
Y. Sudheer Kumar	CB.EN.U4AIE22151	

Table Of Contents

Chapter 1 : GESTURE-BASED NOTES MAKINGAPPLICATION

1.1 Abstract

The "Hand Gesture-Based Notes Making" project introduces an innovative application that allows users to create notes using simple hand gestures in front of a webcam. This interactive system leverages state-of-the-art computer vision and deep learning techniques, including MediaPipe, OpenCV, and TensorFlow, to identify and interpret specific hand gestures that correspond to actions commonly performed in digital note-taking. The primary goal of this project is to provide an intuitive, hands-free interface for creating and editing notes, making it especially useful for people looking for accessible, interactive, and real-time writing solutions.

The project is designed to interpret three main gestures associated with note-making: writing, lifting, and clearing. Each of these gestures corresponds to a specific function within the application.

Gesture 1: Writing Mode
The application tracks the position of the user's middle finger and simulates writing on the screen in real-time. By tracking the path of the middle finger, the system accurately renders the user's writing movements, allowing for continuous input as though writing with a pen.

Gesture 2: Lift Pen Mode
This gesture signals that the user is temporarily lifting the pen without intending to write, allowing for pauses or adjustments without unintended marks appearing on the screen.

Gesture 3: Clear Screen Mode
This gesture clears all written content from the screen, enabling the user to start fresh or reset the writing space as needed.

The Hand Gesture-Based Notes Making application represents a step forward in leveraging hand gesture recognition for user-friendly and interactive note-taking. By combining gesture recognition, real-time video processing, and text extraction, this project illustrates the potential

of advanced computer vision and machine learning methods in creating innovative and practical applications. The web-based interface further enhances the user experience by providing an easy-to-navigate control panel, making it an accessible solution for interactive note-making.

1.2 Introduction

This project, titled "Hand Gesture-Based Notes Making," is designed to allow users to create digital notes using hand gestures through a webcam. By integrating computer vision and deep learning, this system recognizes three distinct gestures—writing, lifting the pen, and clearing the screen. It leverages MediaPipe and OpenCV for gesture recognition, and TensorFlow for training a model to accurately classify these gestures in real-time.

1.2.1 Proposed Outcomes

The primary outcome of this project is to provide a reliable application that:

Accurately Detects Hand Gestures:

1. The model distinguishes between writing, non-writing, and clearing gestures with high accuracy.
   Facilitates Digital Note-Making:
2. Users can interact with a virtual blackboard, writing notes simply by moving their hands.
   Text Extraction for Documentation:
3. A "Extract text" feature captures the content on the blackboard, sending it to an LLM for text extraction, making it easier to store and share notes.

1.2.2 Social Benfits

This system has significant implications, especially for enhancing accessibility and supporting innovative teaching and learning methods:

1. Educational Tools for Differently-Abled Individuals: The system could be an effective tool for individuals with physical disabilities, providing an alternative way to take notes and communicate visually.

2. Interactive Learning Platforms: Teachers and students can use this gesture-based system to create dynamic and interactive learning experiences without requiring traditional

writing tools.

3. Digital Transformation in Note-Taking: By offering a hands-free, digital note-taking experience, this project aligns with the growing demand for sustainable and paperless solutions.

1.3 Literature Survey

A. ***Research:*** *Gesture Recognition Systems*

Gesture recognition has seen extensive research in computer vision and human-computer interaction. Studies indicate that gesture-based interfaces can create intuitive and engaging ways for users to interact with technology. For instance, Mitra and Acharya (2007) discussed the application of gesture recognition systems in various domains, emphasizing the use of visual hand gestures as a natural method for human-computer interaction. Gesture recognition is especially prevalent in fields like sign language translation, gaming, and virtual reality, where intuitive, hands-free controls are beneficial. Techniques such as convolutional neural networks (CNNs) and recurrent neural networks (RNNs) are frequently used for classifying gestures, which aligns with this project's use of TensorFlow for training a hand gesture model.

B. ***Research:*** *MediaPipe for Real-Time Hand Tracking*

MediaPipe is a powerful, open-source framework developed by Google for real-time, cross-platform machine learning applications. It provides robust hand tracking capabilities that significantly improve gesture recognition tasks. According to Zhang et al. (2020), MediaPipe's hand landmark model can identify 21 hand landmarks, allowing for precise and stable hand gesture recognition, even in complex backgrounds. This project leverages MediaPipe’s hand detection to identify and classify hand positions in real-time, which forms the basis for detecting gestures like writing, lifting, and clearing.

C. ***Research:*** *Application of OpenCV in Image Processing*

OpenCV, a widely used computer vision library, provides extensive tools for image processing and real-time object detection, making it ideal for gesture-based projects. Bradski (2000) introduced OpenCV as a comprehensive library for real-time computer vision. This library has been critical in applications like object tracking and augmented reality, which share similarities with gesture recognition. In this project, OpenCV is used to process webcam video feeds, convert frames, and highlight the trajectory of the user’s gestures, making it integral to the note-making functionality.

D. **Research:** *Deep Learning for Image Classification*

In recent years, deep learning has become the primary method for high-accuracy image classification. LeCun et al. (2015) discussed the impact of convolutional networks on advancing image recognition and classification, emphasizing their ability to detect complex patterns in image data. TensorFlow, one of the most popular deep learning frameworks, is used in this project to train a custom image classification model that distinguishes between three different hand gestures: writing, non-writing, and clearing. This gesture classification enables accurate and responsive control over the digital blackboard.

E. **Research:** *Digital Note-Taking and Assistive Technologies*

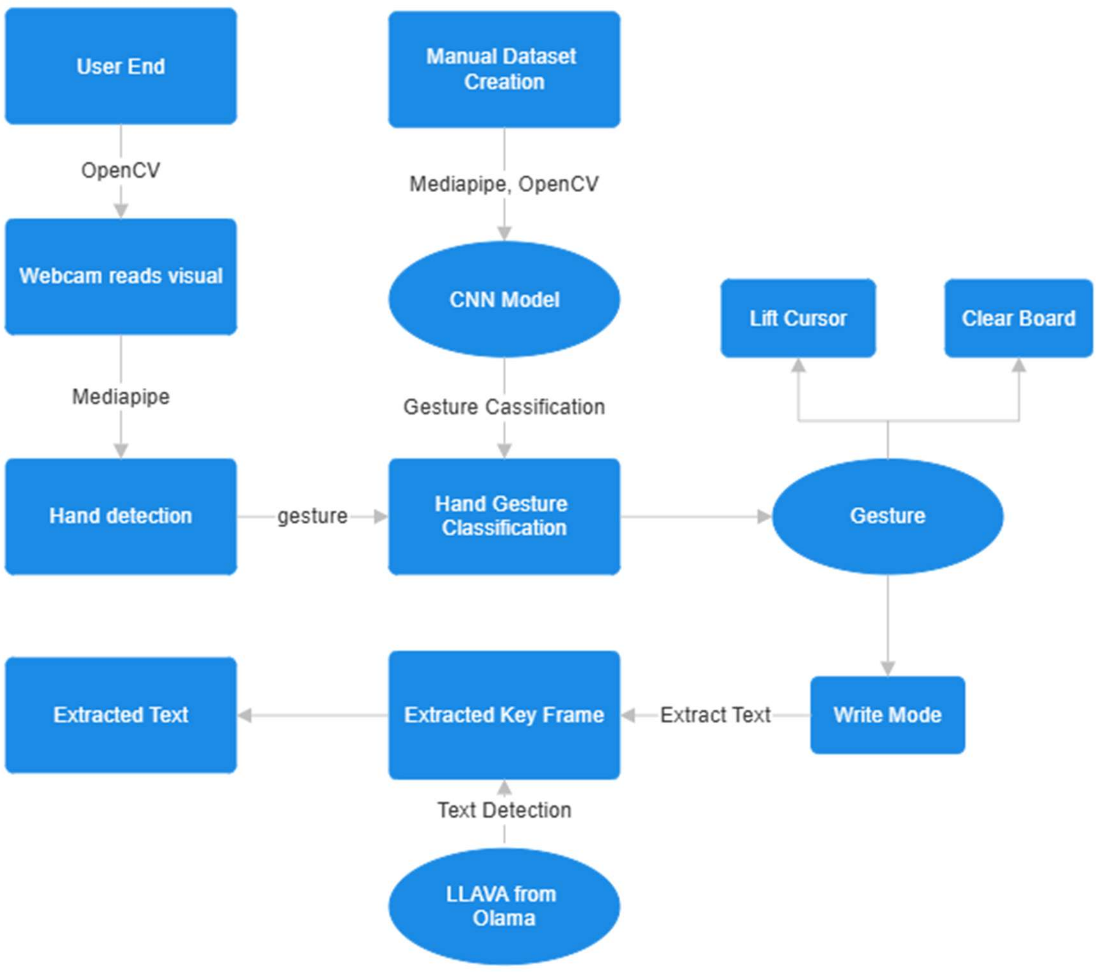
Digital note-taking technologies have become an important tool in educational and accessibility domains. *Oviatt (2013)* highlighted the importance of digital interfaces in aiding cognitive processes such as learning and memory retention. Additionally, gesture-based note-taking tools can benefit individuals with limited motor skills, providing them an accessible way to document and communicate. By capturing text through gestures and converting it into editable digital notes, this project aligns with ongoing research on assistive technologies and digital learning tools.

F. **Research:** *Large Language Models (LLMs) for Text Extraction*

With recent advancements in large language models, text extraction from images and scene text recognition has become more efficient. Models like LLAVA, used in this project, leverage neural networks to interpret and extract text from images. According to *Baek et al. (2019)*, the combination of convolutional and attention-based networks has advanced the field of scene text recognition, achieving higher accuracy in extracting readable text from complex backgrounds. By incorporating LLAVA, this project leverages state-of-the-art natural language processing to accurately transcribe handwritten gestures into digital text, enhancing the usability of the note-making system.

1.4 Methodologies

1.4.1 Work Flow Diagram



1.4.2 Dataset Creation

In this project, the dataset creation process involves capturing hand images in different gestures or modes, which are then classified into three categories: *Writing Mode*, *Moving Mode*, and *Clearing Mode*. Each of these gestures serves a specific function in the notes-making application. Below is a detailed methodology of the dataset creation process, including the use of OpenCV for video capture, MediaPipe for hand tracking, and structured file organization for storing gesture data.

1.4.2.1 Step 1: Initializiation and Setup

- **Webcam Capture and Directory Creation**

The application begins by accessing the computer's webcam to capture real-time video frames. The user is prompted to select a mode by entering `1` for Writing Mode, `2` for Moving Mode, and `3` for Clearing Mode. Based on this input, the application creates a folder (`write_images`, `move_images`, or `clear_images`) for storing the captured hand images.

- **MediaPipe Hands Initialization**

MediaPipe's *Hands* model is initialized for real-time hand detection. MediaPipe, a framework developed by Google, provides machine learning-based solutions for high-performance, cross-platform applications. The *Hands* model used here is specifically designed for hand and finger tracking. It uses a pipeline that identifies hand landmarks and tracks the movement of hands with precision, even in challenging environments. The model is set up with the following parameters:

- `static_image_mode`: Set to `False` for continuous hand tracking in video.
- `max_num_hands`: Restricted to 1 to simplify detection by focusing on a single hand.
- `min_detection_confidence`: Set to 0.4 to control the sensitivity of hand detection.

1.4.2.2 Step 2 : Hand detection and Bounding box detection

1. **Real-Time Frame Processing**
   Each frame captured from the webcam is processed in real-time. The `detect_hand_box` function (imported from `temp.py`) uses MediaPipe to locate the hand in each frame. It returns the bounding box coordinates (`xmin`, `ymin`, `xmax`, `ymax`), which define the rectangular area surrounding the detected hand.
2. **Bounding Box Validation and Cropping**
   To ensure the cropped image is within frame boundaries, the code adjusts the bounding box dimensions based on the frame's width and height. The specified area within the bounding box is then extracted as a separate image, containing only the hand gesture.

1.4.2.3 Step 3: Image Cropping and Saving

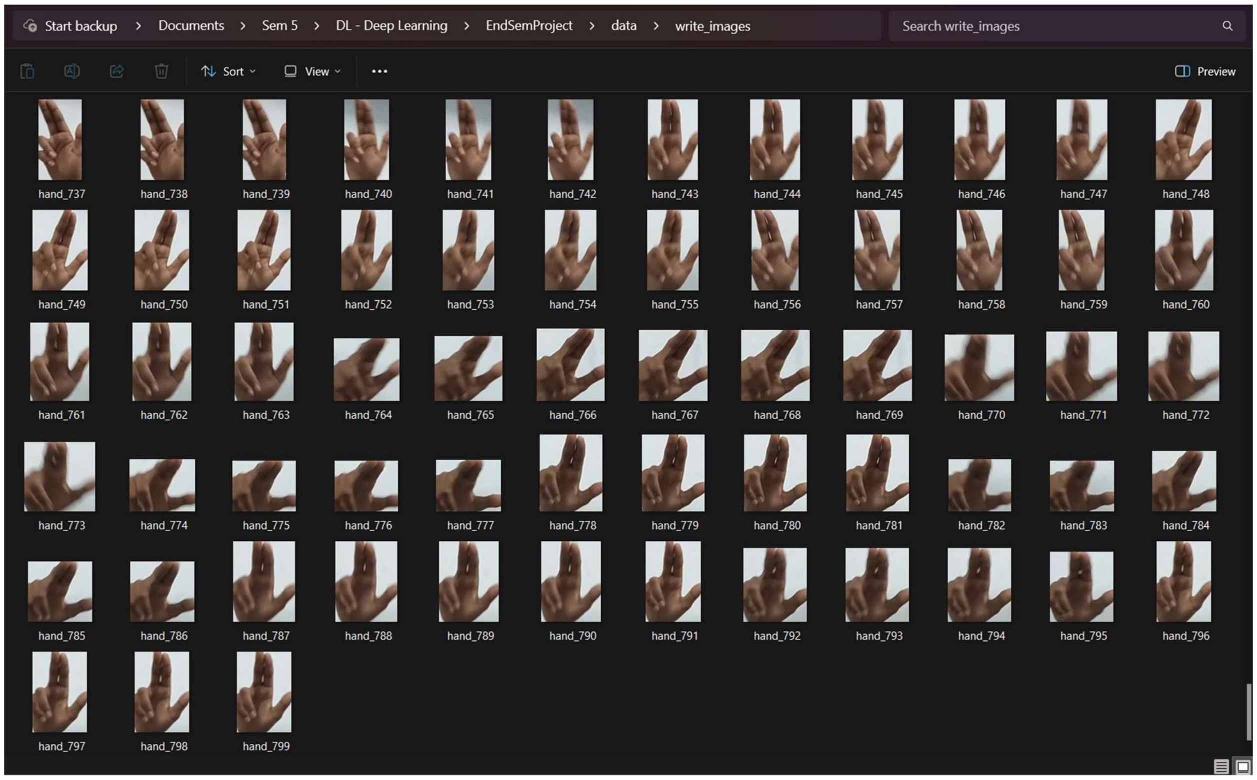
- **Saving Hand Gesture Images**
Once the hand is detected and cropped, the resulting image is saved in the appropriate folder (`write_images`, `move_images`, or `clear_images`) according to the chosen mode. The naming convention, `hand_{image_count}.jpg`, ensures that each image is uniquely identifiable within the dataset.

- **Dataset Size and Quality Control**
The program is configured to capture a maximum of 800 images per mode. Each image is checked for validity (i.e., not being empty) before being saved. If a cropped image is empty, it is skipped to maintain dataset quality.

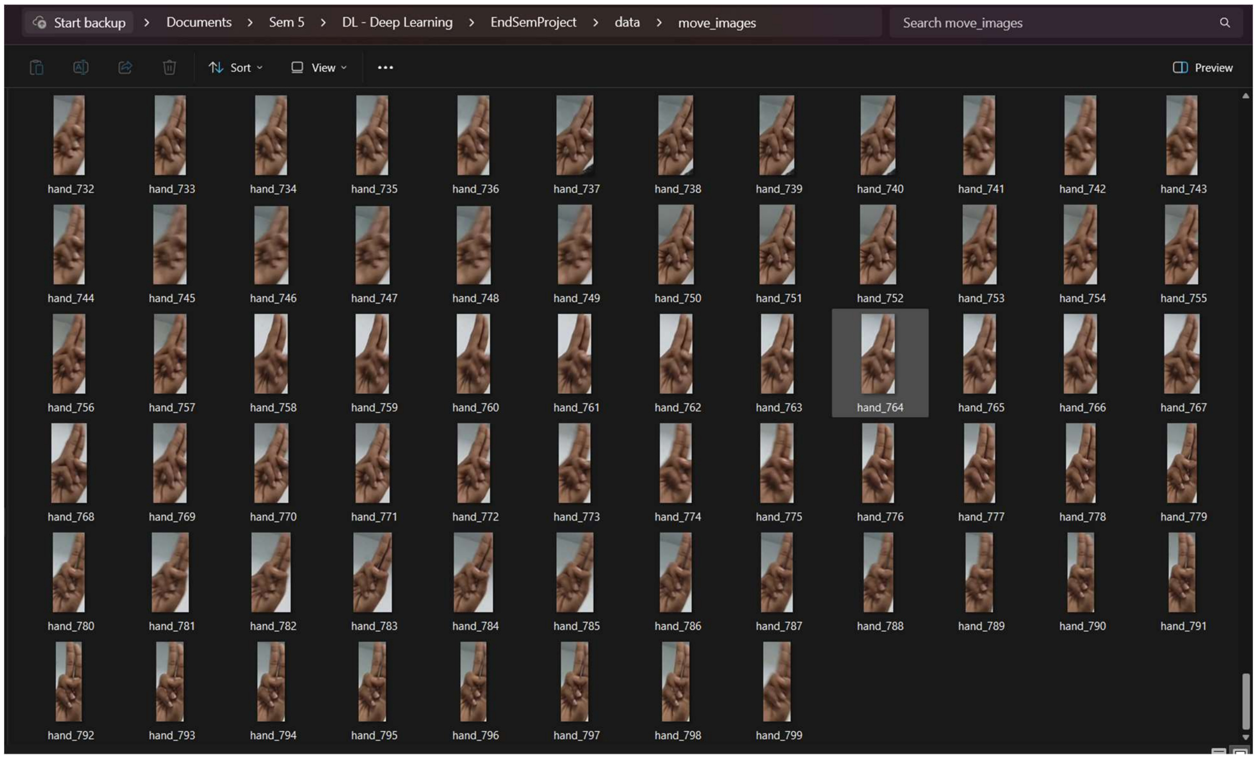
1.4.2.4 Step 4: Displaying and Monitoring the Data Collection Process

For real-time feedback, the bounding box of the detected hand is drawn onto the frame and displayed on the screen. This visual feedback allows the user to adjust their hand position to ensure accurate gesture recording.

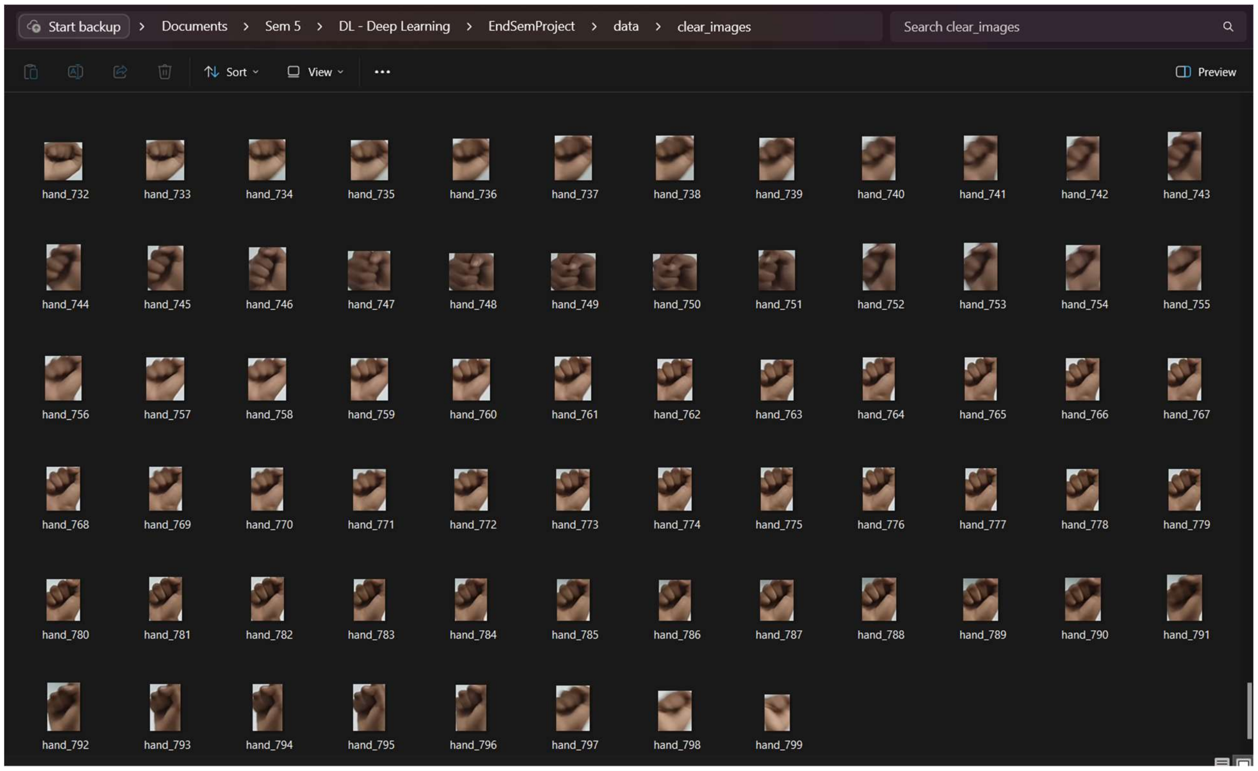
1. **Data for Write mode**


Start backup > Documents > Sem 5 > DL - Deep Learning > EndSemProject > data > write_images
Search write_images
Sort
View
Preview
hand_737 hand_738 hand_739 hand_740 hand_741 hand_742 hand_743 hand_744 hand_745 hand_746 hand_747 hand_748
hand_749 hand_750 hand_751 hand_752 hand_753 hand_754 hand_755 hand_756 hand_757 hand_758 hand_759 hand_760
hand_761 hand_762 hand_763 hand_764 hand_765 hand_766 hand_767 hand_768 hand_769 hand_770 hand_771 hand_772
hand_773 hand_774 hand_775 hand_776 hand_777 hand_778 hand_779 hand_780 hand_781 hand_782 hand_783 hand_784
hand_785 hand_786 hand_787 hand_788 hand_789 hand_790 hand_791 hand_792 hand_793 hand_794 hand_795 hand_796
hand_797 hand_798 hand_799

2. **Data for Lift Cursor**


Start backup > Documents > Sem 5 > DL - Deep Learning > EndSemProject > data > move_images
Search move_images
Sort
View
Preview
hand_732 hand_733 hand_734 hand_735 hand_736 hand_737 hand_738 hand_739 hand_740 hand_741 hand_742 hand_743
hand_744 hand_745 hand_746 hand_747 hand_748 hand_749 hand_750 hand_751 hand_752 hand_753 hand_754 hand_755
hand_756 hand_757 hand_758 hand_759 hand_760 hand_761 hand_762 hand_763 hand_764 hand_765 hand_766 hand_767
hand_768 hand_769 hand_770 hand_771 hand_772 hand_773 hand_774 hand_775 hand_776 hand_777 hand_778 hand_779
hand_780 hand_781 hand_782 hand_783 hand_784 hand_785 hand_786 hand_787 hand_788 hand_789 hand_790 hand_791
hand_792 hand_793 hand_794 hand_795 hand_796 hand_797 hand_798 hand_799

3. **Data for Clear**



1.4.2.5 Mediapipe hands model

MediaPipe Hands uses a machine learning pipeline optimized for hand detection and landmark tracking. The model applies a palm detection algorithm to locate hands in the initial frame. Once a hand is detected, it employs a set of 21 key points or landmarks to capture the hand's contours and finger positions. This landmark data is highly accurate and essential for identifying distinct hand postures reliably, which are then used to classify gestures.

1.4.3 CNN Model for Posture Classification

To classify hand gestures, we use a Convolutional Neural Network (CNN) trained on the dataset of images collected for three classes: Writing Mode, Moving Mode, and Clearing Mode. The CNN model processes and learns features from the images, enabling it to recognize each hand gesture's distinctive patterns. Below is a detailed breakdown of the CNN architecture, including each layer's role, dimensions, and formulas involved.

1.4.3.1 Step 1: Preprocessing and Data Augmentation

The images in the dataset are preprocessed using `ImageDataGenerator`, which:

1. **Rescales** pixel values by dividing by 255 to normalize them to the [0, 1] range.
2. **Applies data augmentation** techniques like shear transformation, zoom, and horizontal flipping, which increase the variety in the training dataset. This helps in reducing overfitting and makes the model more robust to variations in input images.

This preprocessed data is then fed into the model in batches of 32 images with dimensions 128x128 pixels and 3 color channels (RGB).

1.4.3.2 Step 2: Convolutional Neural Network Architecture

The CNN model consists of the following layers, each playing a role in extracting hierarchical features from the input images:

1. **Input Layer (Conv2D, 32 filters, 3x3 kernel)**
   - **Shape**: Input shape is (128, 128, 3), where 128x128 is the image size, and 3 represents the RGB color channels.
   - **Operation**: Convolution with 32 filters of size 3x3, followed by the ReLU activation function.
   - **Formula**: The convolution operation for an input image III and a filter FFF is given by:
   - $$(I * F)(x, y) = \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} I(x+i, y+j) \cdot F(i, j)$$
   - where m and n are the dimensions of the filter.
   - **Output Shape**: After the convolution, the output shape is (126, 126, 32), which is then reduced by max pooling.
2. **Pooling Layer (MaxPooling2D, 2x2)**
   - **Operation**: This layer performs max pooling with a 2x2 filter, reducing the spatial dimensions.
   - **Formula**: Max pooling selects the maximum value from each 2x2 region.
   - **Output Shape**: Reduces the output shape to (63, 63, 32)
3. **Second Convolutional Layer (Conv2D, 64 filters, 3x3 kernel)**

   - **Operation**: Another convolution layer with 64 filters and a 3x3 kernel size, followed by ReLU activation.
   - **Output Shape**: (61, 61, 64), which is again reduced by max pooling.

4. **Second Pooling Layer (MaxPooling2D, 2x2)**

   - **Operation**: Another 2x2 max pooling layer.
   - **Output Shape**: Reduces the output to (30, 30, 64).
5. **Third Convolutional Layer (Conv2D, 128 filters, 3x3 kernel)**
   - **Operation**: This layer has 128 filters with a 3x3 kernel size, followed by ReLU activation.
   - **Output Shape**: (28, 28, 128), reduced by max pooling.
6. **Third Pooling Layer (MaxPooling2D, 2x2)**
   - **Operation**: A 2x2 max pooling operation.
   - **Output Shape**: Reduces to (14, 14, 128).
7. **Flatten Layer**
   - **Operation**: Converts the 3D feature maps into a 1D vector, preparing it for the dense layers.
   - **Output Shape**: Flattens to a shape of (25088,).
8. **Dense Layer (Fully Connected, 128 units)**
   - **Operation**: This fully connected layer with 128 neurons and ReLU activation helps in learning complex representations.
   - **Output Shape**: (128,).
9. **Dropout Layer (Dropout, 0.5)**
   - **Operation**: Dropout is applied to prevent overfitting by randomly setting 50% of the nodes to zero during training.
10. **Output Layer (Dense, 3 units with Softmax)**
    - **Operation**: The final dense layer has 3 neurons, corresponding to the 3 gesture classes. The softmax activation function is applied to obtain a probability distribution over the classes.
    - **Formula**: Softmax for each class iii is given by:
    - $$\sigma(z_i) = \frac{e^{z_i}}{\sum_j e^{z_j}}$$
    - **Output Shape**: (3,), representing the probabilities of each gesture class.

1.4.3.3 Model Compilation

The model is compiled with the following settings:

- **Optimizer**: Adam, which adjusts learning rates for efficient convergence.
- **Loss Function**: Categorical crossentropy, suitable for multi-class classification. For each class ccc and sample iii, the formula is:

- $$\text{Loss} = -\sum_{c} y_{i,c} \log(\hat{y}_{i,c})$$
- where yyy is the true label, and y^\hat{y}y^ is the predicted probability.
- **Metrics**: Accuracy is tracked to measure model performance.

1.4.3.4 Training the Model

The model is trained on batches of 32 images over 30 epochs. In each epoch:

- **Forward Propagation** computes the output probabilities.
- **Backward Propagation** adjusts the model parameters to minimize the loss.

1.4.3.5 Saving the Model

After training, the model is saved as `gesture_classification_model.h5`, which can be loaded to classify new hand gestures in the application. This CNN architecture efficiently extracts features at various scales using convolution and pooling layers, captures complex patterns through dense layers, and outputs probabilities across the three gesture classes.

1.4.4 Back end application

The `app.py` file is the main script that drives the entire hand-gesture-based note-making application. This program allows users to use hand gestures detected through a webcam to write notes, lift the "pen" without writing, and clear the screen. Here's a breakdown of the entire functionality and how the application works:

1.4.4.1 Model Loading and Image Preprocessing

- The code loads a pre-trained gesture classification model (`gesture_classification_model.h5`) to recognize specific hand gestures. These gestures correspond to different "modes" for writing, moving, and clearing.

- The `preprocess_image` function prepares the captured hand region for the model by resizing it to 128x128 pixels, converting it to a suitable array format, and normalizing the pixel values to match the input requirements of the model.

1.4.4.2 Hand Detection with MediaPipe

- MediaPipe is initialized to detect hands using the `Hands` solution. It tracks the hand landmarks and positions in real-time through the webcam feed.

- MediaPipe detects and identifies the hand in each frame, enabling the application to isolate and focus on the region containing the hand.

1.4.4.3 Gesture Recognition and Mode Identification

- Once MediaPipe identifies the hand, the `detect_hand_box` function (from `temp.py`) isolates the bounding box for the hand within the frame.

- This bounding box (the "mini-frame") is preprocessed and passed to the gesture classification model to determine the current gesture (or mode):

  - Mode 0 (Clear Screen): Clears all drawn lines on the screen.

  - Mode 1 (Lift Pen): Detects when the user lifts the pen without drawing, pausing the tracking of the line.

  - Mode 2 (Writing Mode): Tracks the middle finger's position, drawing a line to simulate writing on the screen.

1.4.4.4 Drawing on the Screen

- The application tracks the middle finger's position by analyzing MediaPipe landmarks, particularly landmark 12 (the middle finger tip).

- In "Writing Mode" (Mode 2), the program draws continuous lines based on the movement of the middle finger, simulating a pen stroke.

- If the mode changes to Mode 1, the "pen" is lifted, and drawing pauses.

- The drawing positions are cleared when Mode 0 is detected, resetting the screen.

1.4.4.5 Output Frame and Frame Saving

- Each processed frame is displayed in real-time in a window titled "Finger Position Detection."

- Additionally, the application periodically saves each frame as an image file named `current_frame.jpg`. This saved image allows for text extraction and serves as the output that users can reference for notes.

1.4.4.6 Exit and Cleanup

- The application runs in a loop, capturing and processing frames continuously.

- When the user presses the "q" key, the loop breaks, releasing the webcam and closing all windows, thereby ending the application.

1.4.4.7 Application Workflow Summary

1. Setup and Initialization: Load the gesture classification model, initialize MediaPipe for hand detection, and start capturing video.

2. Real-Time Detection and Classification:

   - Detect the hand and crop the frame.

   - Preprocess and classify the hand gesture to determine the current mode.

3. Response to Gesture Modes:

   - Track finger positions and draw on the screen in writing mode.

   - Pause drawing in lift mode.

   - Clear lines in clear mode.

4. Display and Save Frames: Show the output in a real-time window and save each frame for text extraction or future reference.

5. Close Application: Gracefully exit by releasing resources and closing the display when finished.

1.4.5 Future scope : Text Extraction

The `text_extractor.py` code leverages a multimodal AI model, LLAVA (Large Language and Vision Assistant), to interpret an image input and extract individual alphabet characters. LLAVA combines visual and language processing capabilities, enabling it to understand images in response to text prompts. Here's a detailed breakdown of what this code is doing and how LLAVA's architecture enables the extraction task.

1.4.5.1 Purpose and Workflow

The main purpose of this code is to use LLAVA to:

1. **Process an image of handwritten or printed text.**
2. **Extract individual alphabets based on the prompt provided.**
3. **Return only the alphabetic content, filtering out any additional or irrelevant information.**

The steps include:

- Converting an image into a base64 format,
- Sending it to LLAVA with a prompt, and
- Receiving a response containing the extracted text.

1.4.5.2 Overview of LLAVA Architecture and Functioning

LLAVA is a deep multimodal model that typically combines components from both vision and language models. Here's how its architecture generally functions for this task:

1. **Vision Transformer (ViT) for Image Encoding:**
    - The image is processed through a Vision Transformer or similar convolutional backbone. The transformer divides the image into smaller patches and treats each as a "token," extracting features relevant to the visual content.
    - **Layers and Parameters:**
        - LLAVA's vision encoder usually consists of a stack of self-attention layers, often ranging from 12 to 24 layers (based on model size).
        - Each layer has multi-head self-attention, with 12-16 heads per layer.
        - The ViT has millions of parameters (anywhere from 86 million to 300+ million parameters for typical configurations).
2. **Token Embedding and Attention Mechanisms:**
    - The encoded image tokens are combined with textual tokens from the prompt. This enables the model to interpret the visual data in the context of the provided instructions.
    - **Parameter Count and Input/Output Sizes:**
        - The model uses an attention mechanism where the image embeddings (of size, say, 768) interact with text embeddings.
        - The embeddings are then aligned to make the image and text data interpretable together.
3. **Text Generation via Language Model (e.g., LLaMA or similar):**
    - LLAVA includes a language model, often based on models like GPT or LLaMA, which generates the output based on processed multimodal embeddings.
    - **Layers and Parameter Counts:**
        - The language model typically has 12 to 48 transformer blocks, depending on model size.
        - Each transformer block has feedforward layers, multi-head self-attention, and normalization layers.
        - Parameters here can range from 125 million (LLaMA-1) to 13 billion (LLaMA-13B), depending on the version.
    - **Input and Output Sizes:**

- The language model accepts a sequence of embeddings as input (combined image and text embeddings) and outputs a text sequence with the extracted characters.

1.4.5.3 Input and Output Sizes in LLAVA for the Task

- **Input Size (Image):** The code resizes the image to a fixed size (128x128 pixels) and converts it to an array. When passed to LLAVA, the image is tokenized into patches (e.g., 16x16 pixel patches) for the ViT. Each patch then becomes a token of size 768 or higher, depending on the architecture.
- **Output Size (Text):** The language model generates text in a sequence format. The specific characters are returned as plain text, and the output length can vary based on the model's token limit (typically up to 512-2048 tokens).

1.4.6 Front end application using Streamlit

Now we have to develop a front end application which acts as the primary interface for the **Hand Gesture-Based Note Maker** application, tying together the different components and functionalities that have been implemented. Here's a summary of what the code accomplishes and how it operates:

1.4.6.1 Purpose and Functionality Overview

1. **Launch the Hand Gesture-Based Note Maker application** using a webcam to track hand gestures for note-taking.
2. **Capture a frame** from the note-making application.
3. **Extract handwritten text** from the captured frame and display the extracted text on the web interface.

1.4.6.2 Core Functionalities and How They Work

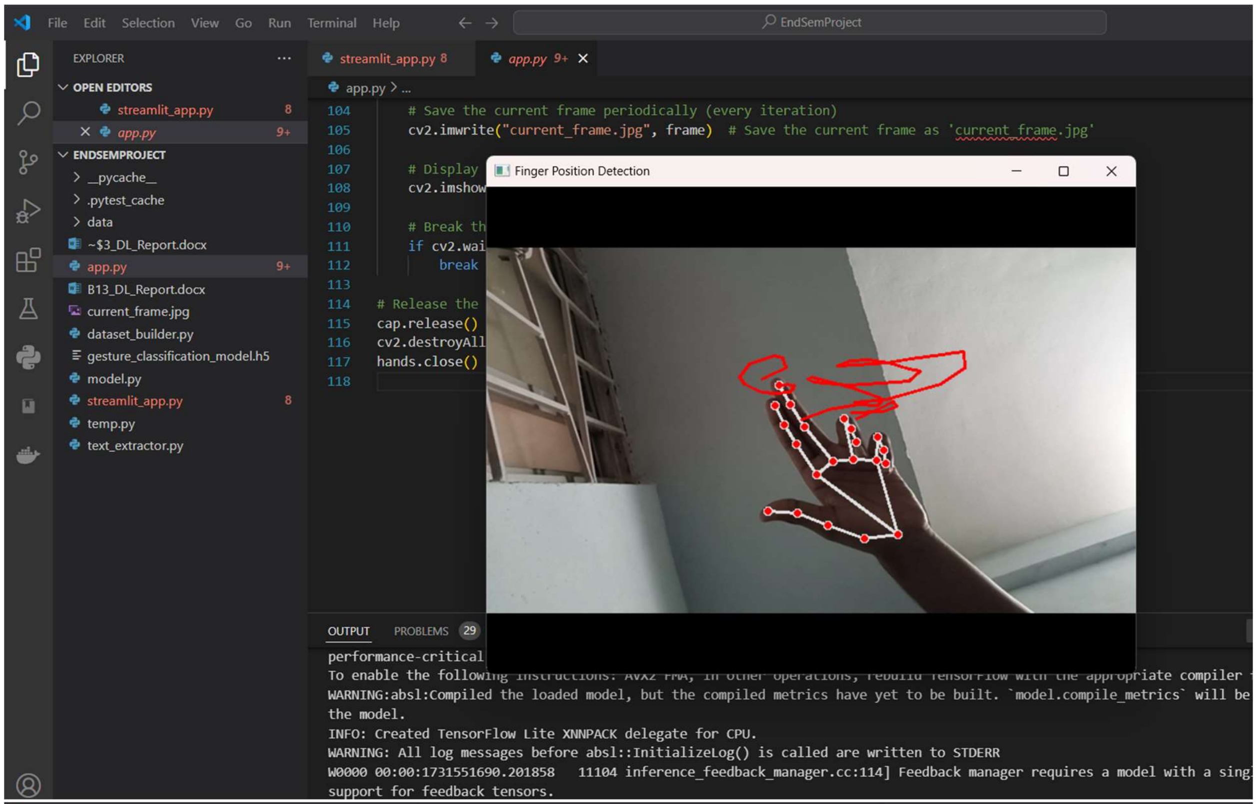
1. **Launching the Hand Gesture-Based Note Maker**:
   - The app provides a "Start window" button that, when clicked, starts the `app.py` script (the main note-making application) as a subprocess. This subprocess runs the application in a separate window, activating gesture recognition and hand-tracking functionalities to allow users to write on a virtual board using hand gestures.

    - This subprocess management allows the user to open or close the note-making window without restarting the entire Streamlit application.
2. **Closing the Note-Making Application**:
    - The "Close window" button stops the subprocess running `app.py`, effectively closing the note-making application. This enables the user to shut down the gesture-tracking window as needed without affecting the Streamlit interface itself.
3. **Extracting Text from Captured Frames**:
    - When the "Extract text" button is clicked, the application searches for a recent captured frame saved as `current_frame.jpg`. This file represents the visual notes that the user has written through gestures.
    - If a valid frame is found, the code loads and displays the image on the Streamlit interface.
    - It then uses the **LLAVA (Large Language and Vision Assistant)** model, accessed via the `text_extractor.py` module, to extract text from the image. This model processes the image to identify and isolate individual alphabet characters according to the prompt.
    - The extracted text is displayed in a text area, providing a digital record of the handwritten notes.

1.4.6.3 Technical Operation and Integration

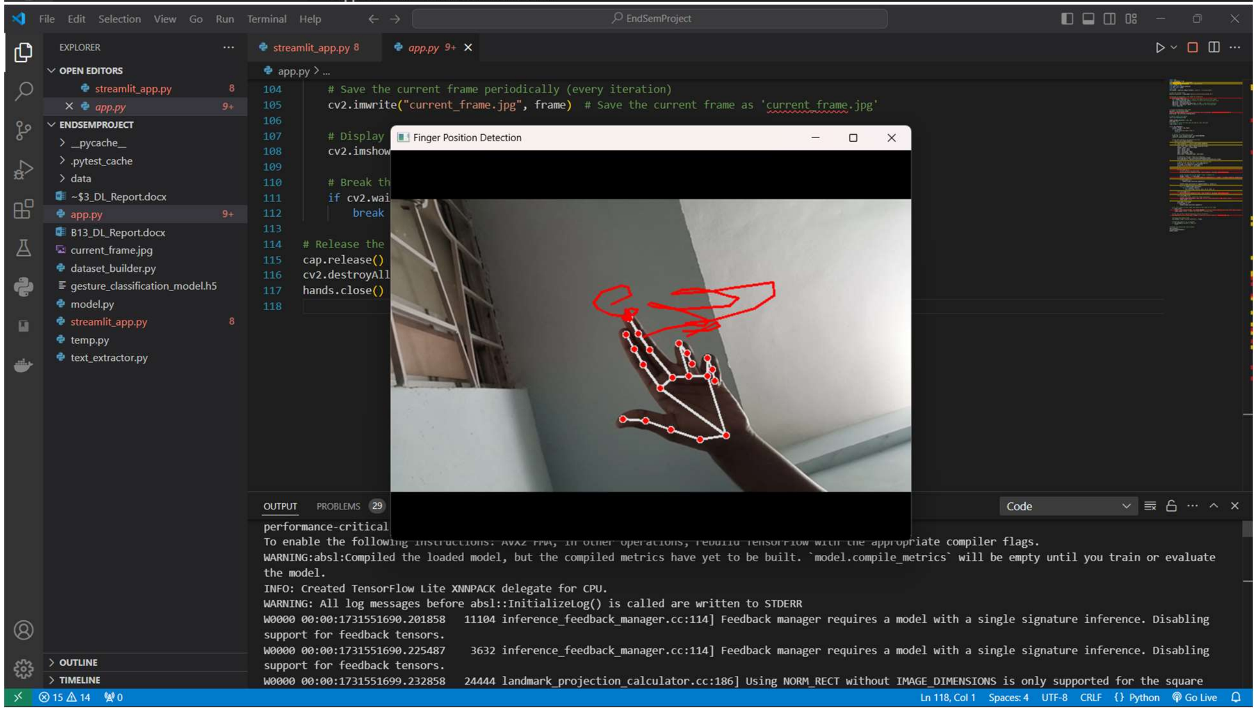
- **Session State Management**: Streamlit's `session_state` is used to manage the subprocess state and ensure that the "Start window" and "Close window" buttons correctly open and close the note-making application.
- **Image Processing and Display**: OpenCV loads the captured frame, which is then displayed through Streamlit, giving the user immediate feedback on what has been captured.
- **Text Extraction and Display**: The image is sent to LLAVA, which processes it based on the prompt. The extracted text appears on the interface, completing the note-taking process by transforming hand gestures into a readable, text-based format.

1.5 Results



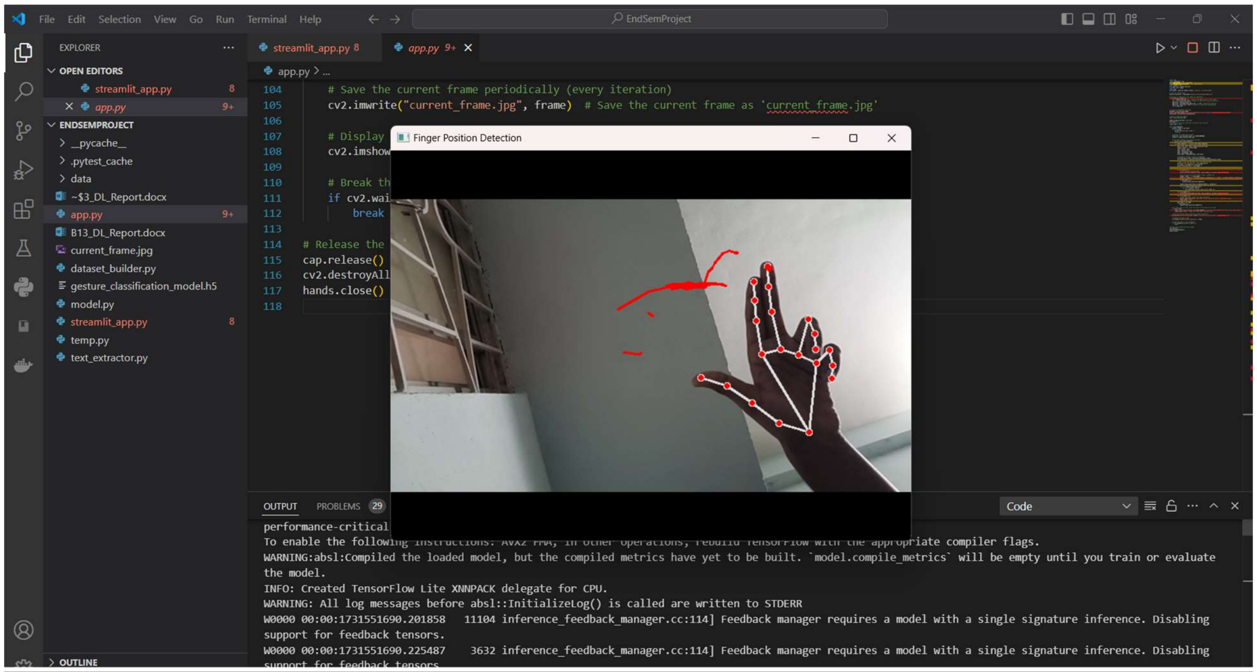
```
    # Save the current frame periodically (every iteration)
    cv2.imwrite("current_frame.jpg", frame)  # Save the current frame as 'current_frame.jpg'

    # Display
    cv2.imshow

    # Break th
    if cv2.wai
        break

# Release the
cap.release()
cv2.destroyAll
hands.close()
```

```
performance-critical
To enable the following instructions: AVX2 FMA, in other operations, rebuild TensorFlow with the appropriate compiler
WARNING:absl:Compiled the loaded model, but the compiled metrics have yet to be built. `model.compile_metrics` will be
the model.
INFO: Created TensorFlow Lite XNNPACK delegate for CPU.
WARNING: All log messages before absl::InitializeLog() is called are written to STDERR
W0000 00:00:1731551690.201858   11104 inference_feedback_manager.cc:114] Feedback manager requires a model with a sing
support for feedback tensors.
```


File Edit Selection View Go Run Terminal Help
EndSemProject
EXPLORER
OPEN EDITORS
ENDSEMPROJECT
Finger Position Detection
OUTPUT
PROBLEMS 29
Code
OUTLINE
TIMELINE
Ln 118, Col 1 Spaces: 4 UTF-8 CRLF {} Python Go Live

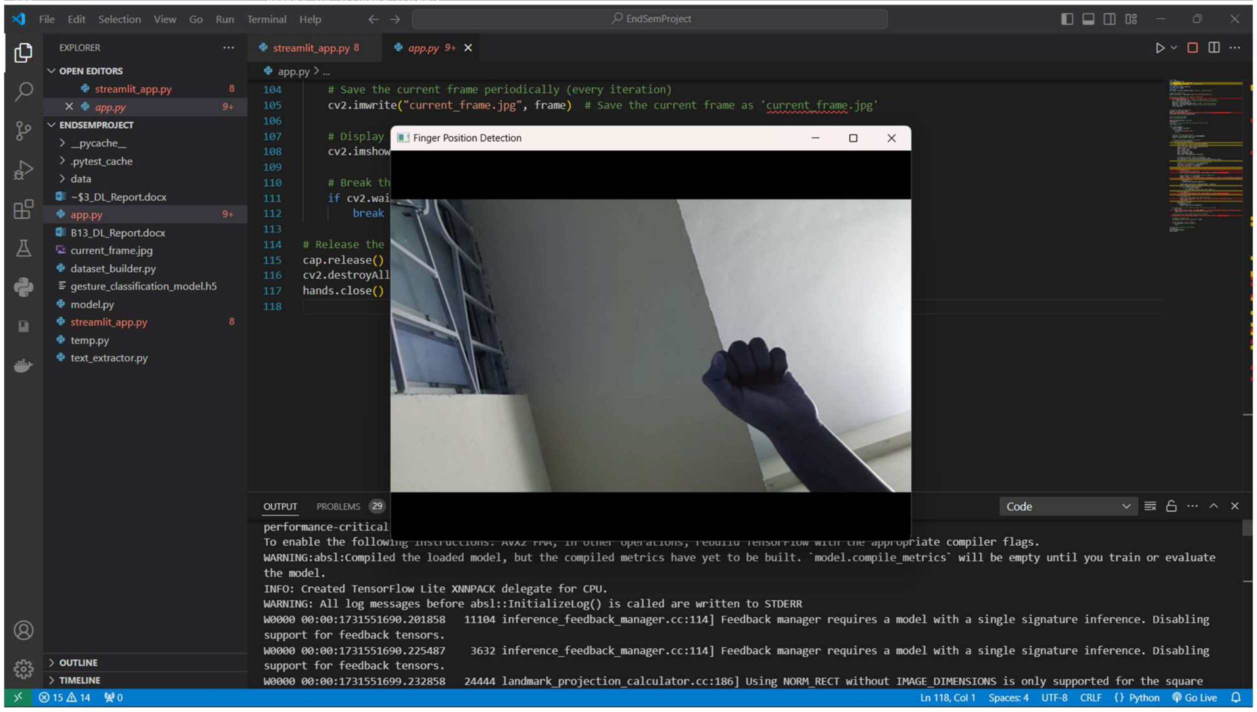
```
    # Save the current frame periodically (every iteration)
    cv2.imwrite("current_frame.jpg", frame)  # Save the current frame as 'current_frame.jpg'

    # Display
    cv2.imshow

    # Break th
    if cv2.wai
        break

# Release the
cap.release()
cv2.destroyAll
hands.close()
```

```
performance-critical
To enable the following instructions: AVX2 FMA, in other operations, rebuild TensorFlow with the appropriate compiler flags.
WARNING:absl:Compiled the loaded model, but the compiled metrics have yet to be built. `model.compile_metrics` will be empty until you train or evaluate
the model.
INFO: Created TensorFlow Lite XNNPACK delegate for CPU.
WARNING: All log messages before absl::InitializeLog() is called are written to STDERR
W0000 00:00:1731551690.201858   11104 inference_feedback_manager.cc:114] Feedback manager requires a model with a single signature inference. Disabling
support for feedback tensors.
W0000 00:00:1731551690.225487    3632 inference_feedback_manager.cc:114] Feedback manager requires a model with a single signature inference. Disabling
support for feedback tensors.
W0000 00:00:1731551699.232858   24444 landmark_projection_calculator.cc:186] Using NORM_RECT without IMAGE_DIMENSIONS is only supported for the square
```



```
    # Save the current frame periodically (every iteration)
    cv2.imwrite("current_frame.jpg", frame)  # Save the current frame as 'current_frame.jpg'

    # Display
    cv2.imshow

    # Break th
    if cv2.wai
        break

# Release the
cap.release()
cv2.destroyAll
hands.close()
```

```
performance-critical
To enable the following instructions: AVX2 FMA, in other operations, rebuild TensorFlow with the appropriate compiler flags.
WARNING:absl:Compiled the loaded model, but the compiled metrics have yet to be built. `model.compile_metrics` will be empty until you train or evaluate
the model.
INFO: Created TensorFlow Lite XNNPACK delegate for CPU.
WARNING: All log messages before absl::InitializeLog() is called are written to STDERR
W0000 00:00:1731551690.201858   11104 inference_feedback_manager.cc:114] Feedback manager requires a model with a single signature inference. Disabling
support for feedback tensors.
W0000 00:00:1731551690.225487    3632 inference_feedback_manager.cc:114] Feedback manager requires a model with a single signature inference. Disabling
support for feedback tensors.
```

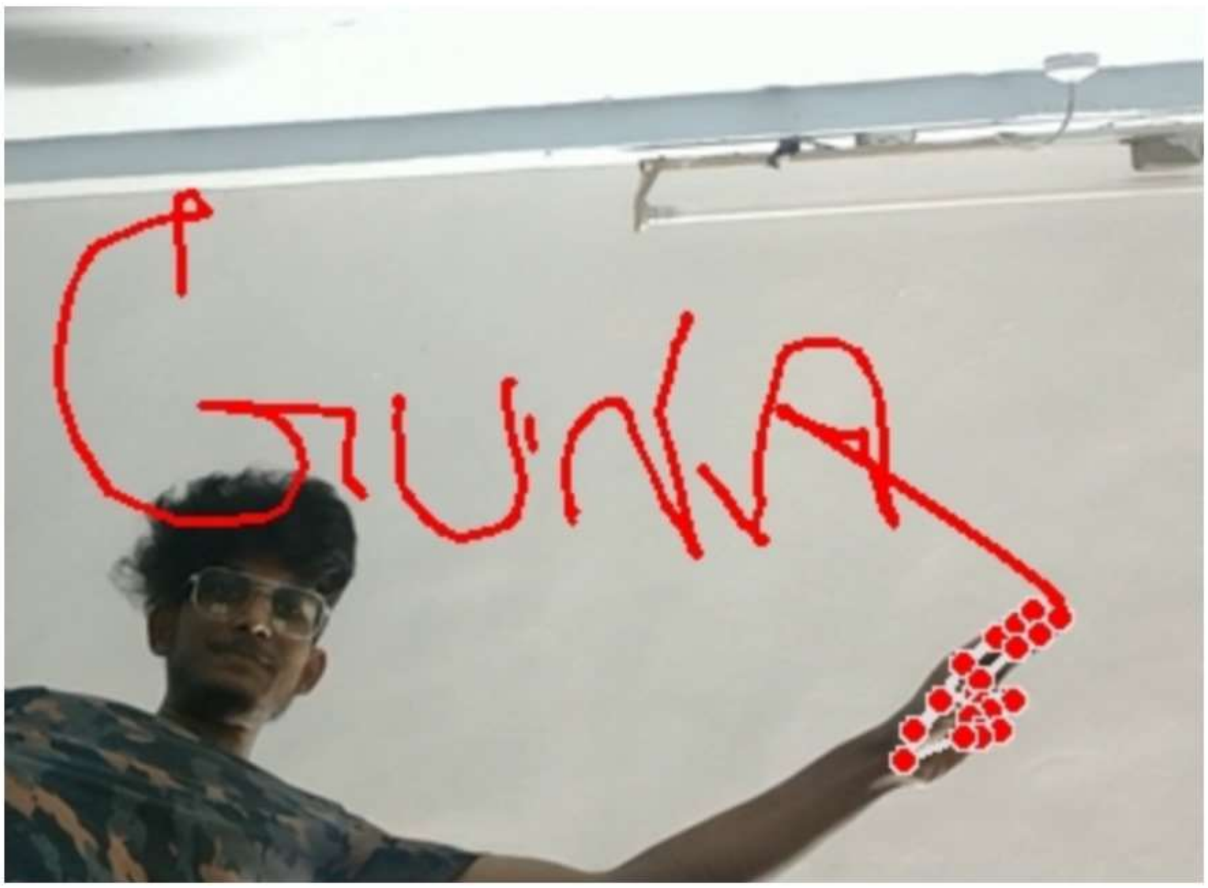


```
    # Save the current frame periodically (every iteration)
    cv2.imwrite("current_frame.jpg", frame)  # Save the current frame as 'current_frame.jpg'

    # Display
    cv2.imshow

    # Break th
    if cv2.wai
        break

# Release the
cap.release()
cv2.destroyAll
hands.close()
```

```
performance-critical
To enable the following instructions: AVX2 FMA, in other operations, rebuild TensorFlow with the appropriate compiler flags.
WARNING:absl:Compiled the loaded model, but the compiled metrics have yet to be built. `model.compile_metrics` will be empty until you train or evaluate
the model.
INFO: Created TensorFlow Lite XNNPACK delegate for CPU.
WARNING: All log messages before absl::InitializeLog() is called are written to STDERR
W0000 00:00:1731551690.201858   11104 inference_feedback_manager.cc:114] Feedback manager requires a model with a single signature inference. Disabling
support for feedback tensors.
W0000 00:00:1731551690.225487    3632 inference_feedback_manager.cc:114] Feedback manager requires a model with a single signature inference. Disabling
support for feedback tensors.
```

```
W0000 00:00:1731551699.232858   24444 landmark_projection_calculator.cc:186] Using NORM_RECT without IMAGE_DIMENSIONS is only supported for the square
```

1.6 Future Scope : Text Extraction



Uploaded Image

Math Problem Solver with Ollama (LLaVA)

Upload an image with a math problem and provide a prompt.

Upload Image

Drag and drop file here
Limit 200MB per file • PNG, JPG, JPEG

Browse files

Screenshot 2024-11-13 224305.png 352.3KB

Enter your prompt (e.g., 'Solve the math problem in this image'):

what are the individual alphabets in the image. output only those no other words

Response: GUNA

1.7 Conclusions

1.7.1 Conclusion

In the results and discussion section, the primary functionality of the project—the gesture-based note-making using accurate finger tracking—has proven to be highly effective. The system successfully interprets different hand gestures to distinguish between writing, moving, and clearing actions, enabling seamless note creation.

The integration of LLAVA for text extraction, initially planned as a future scope, represents an ambitious advancement. While the feature performs well under optimal conditions, it sometimes encounters issues, such as the "Image is not clear, try with another image" error. This likely arises from suboptimal image resolution or variations in text clarity, which is expected when pushing LLAVA's capabilities to its limits. Future improvements could focus on enhancing image preprocessing before sending data to LLAVA, potentially increasing its robustness and accuracy in text extraction.

Overall, the project has met its core objectives effectively, with promising advancements that could extend its utility even further with refinement.

1.7.2 Novelity

Everything in this project from dataset to code is created by manually.

1.7.3 References

[1] MediaPipe: A Framework for Building Perception Pipelines
Camillo Lugaresi, Jiuqiang Tang, Hadon Nash, Chris McClanahan, Esha Uboweja, Michael Hays, Fan Zhang, Chuo-Ling Chang, Ming Guang Yong, Juhyun Lee, Wan-Teh Chang, Wei Hua, Manfred Georg, Matthias Grundmann

[2] Visual Instruction Tuning
Haotian Liu, Chunyuan Li, Qingyang Wu, Yong Jae Lee

```python
import cv2
import mediapipe as mp
from keras.models import load_model
from keras.preprocessing.image import img_to_array
import numpy as np
from temp import detect_hand_box
from PIL import Image
import sys
sys.stdout = open(sys.stdout.fileno(), mode='w', encoding='utf8')

# Load the trained model
gesture_classifier = load_model('gesture_classification_model.h5')

# Function to preprocess a new image for prediction
def preprocess_image(img):
    img = Image.fromarray(img)  # Convert the NumPy array to a PIL image
    img = img.resize((128, 128))  # Resize to match the training input size
    img_array = img_to_array(img)
    img_array = np.expand_dims(img_array, axis=0)  # Add batch dimension
    img_array = img_array / 255.0  # Rescale to match the training data
    return img_array


# Initialize MediaPipe hand model
mp_hands = mp.solutions.hands
hands = mp_hands.Hands(static_image_mode=False, max_num_hands=1,
min_detection_confidence=0.4)
mp_drawing = mp.solutions.drawing_utils

# Capture video from webcam
cap = cv2.VideoCapture(0)

middle_finger_positions = [[], []]
prev_mode = 0
# Variable to store the mode when we need to clear the path
clear_lines = False

while cap.isOpened():
    success, frame = cap.read()
    if not success:
        print("Ignoring empty frame.")
        continue

    # Convert the frame color to RGB
    frame_rgb = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)
    results = hands.process(frame_rgb)

    # Draw hand landmarks and display positions
    if results.multi_hand_landmarks:

        for hand_landmarks in results.multi_hand_landmarks:

            xmin, ymin, xmax, ymax = detect_hand_box(frame)
            height, width, _ = frame.shape
            xmin = max(0, xmin)
            ymin = max(0, ymin)
            xmax = min(width, xmax)
            ymax = min(height, ymax)
            mini_frame = frame[ymin:ymax, xmin:xmax]

            # Preprocess the mini frame for prediction
            procecced_mini_frame = preprocess_image(mini_frame)
            curr_mode_matrix = gesture_classifier.predict(procecced_mini_frame)

            # Debug: Print current mode matrix and detected mode
            print("Current Mode Matrix:", curr_mode_matrix)
            curr_mode = np.argmax(curr_mode_matrix)
            print("Predicted Mode:", curr_mode)


            if curr_mode == 2:
                # Draw landmarks on the frame
                mp_drawing.draw_landmarks(frame, hand_landmarks,
mp_hands.HAND_CONNECTIONS)

                # Get the position of the middle finger (landmark 12)
                height, width, _ = frame.shape
                middle_x, middle_y = int(hand_landmarks.landmark[12].x * width),
int(hand_landmarks.landmark[12].y * height)

                if prev_mode == 1:
                    middle_finger_positions.append([])

                middle_finger_positions[-1].append((middle_x, middle_y))

                for i in middle_finger_positions:
                    for j in range(1, len(i)):
                        cv2.line(frame, i[j-1], i[j], (0, 0, 255), 2)

            elif curr_mode == 1:
                mp_drawing.draw_landmarks(frame, hand_landmarks,
mp_hands.HAND_CONNECTIONS)

            elif curr_mode == 0:
                # Clear the lines (reset the finger positions)
                middle_finger_positions = [[], []]  # This clears the path of the
middle finger
                clear_lines = True

            prev_mode = curr_mode
            if prev_mode == 1:
                middle_finger_positions.append([])

    # If clear_lines is True, reset the state of the lines on the frame
    if clear_lines:
        frame = cv2.cvtColor(frame, cv2.COLOR_RGB2BGR)  # Convert back to BGR to
avoid any color space issues
        clear_lines = False  # Reset the flag after clearing lines

    # Save the current frame periodically (every iteration)
    cv2.imwrite("current_frame.jpg", frame)  # Save the current frame as
'current_frame.jpg'

    # Display the output frame
    cv2.imshow('Finger Position Detection', frame)

    # Break the loop if 'q' is pressed
    if cv2.waitKey(1) & 0xFF == ord('q'):
        break

# Release the webcam and close windows
cap.release()
cv2.destroyAllWindows()
hands.close()
```

```
import cv2
import os
```

```python
from temp import detect_hand_box  # Import the function from temp.py
import mediapipe as mp

# Initialize MediaPipe Hands outside the function to reuse the graph
mp_hands = mp.solutions.hands


# Capture video from webcam
cap = cv2.VideoCapture(0)
image_count = 0

hands = mp_hands.Hands(static_image_mode=False, max_num_hands=1,
min_detection_confidence=0.4)

data_type = int(input("Enter 1 for enlarging write data, 2 for move and 3 for
clear data : "))

# Create a folder for saving hand images
if data_type == 1:
    output_folder = "write_images"
elif data_type == 2:
    output_folder = "move_images"
elif data_type == 3:
    output_folder = "clear_images"
else:
    print("Enter a number between 1 and 3")

os.makedirs(output_folder, exist_ok=True)

while cap.isOpened() and image_count<800:
    success, frame = cap.read()
    if not success:
        print("Ignoring empty frame.")
        continue

    # Detect hand and get bounding box
    box = detect_hand_box(frame)
    if box:
        xmin, ymin, xmax, ymax = box

        # Ensure coordinates are within frame bounds
        height, width, _ = frame.shape
        xmin = max(0, xmin)
        ymin = max(0, ymin)
        xmax = min(width, xmax)
        ymax = min(height, ymax)

        # Crop the region containing the hand
        hand_image = frame[ymin:ymax, xmin:xmax]

        # Check if hand_image is valid (not empty)
        if hand_image.size > 0:
            # Save the cropped image
            image_path = os.path.join(output_folder, f"hand_{image_count}.jpg")
            cv2.imwrite(image_path, hand_image)
            print(f"Saved hand image at {image_path}")
            image_count += 1
        else:
            print("Cropped hand image is empty, skipping save.")

    # Display the output frame with the detected bounding box (optional)
    if box:
        cv2.rectangle(frame, (xmin, ymin), (xmax, ymax), (0, 255, 0), 2)
    cv2.imshow("Hand Detection", frame)

    # Break the loop if 'q' is pressed
    if cv2.waitKey(1) & 0xFF == ord('q'):
        break

# Release the webcam and close windows
cap.release()
cv2.destroyAllWindows()
hands.close()
```

```python
import tensorflow as tf
from tensorflow.keras import layers, models
from tensorflow.keras.preprocessing.image import ImageDataGenerator
import sys
sys.stdout.reconfigure(encoding='utf-8')


# Set image dimensions
img_width, img_height = 128, 128  # Resize images to 128x128
batch_size = 32

# Directory path to your dataset
train_dir = 'data'  # e.g., 'dataset/'

# Data Augmentation and Preprocessing
train_datagen = ImageDataGenerator(
    rescale=1.0/255.0,  # Rescale pixel values to [0, 1]
    shear_range=0.2,
    zoom_range=0.2,
    horizontal_flip=True
)

train_generator = train_datagen.flow_from_directory(
    train_dir,
    target_size=(img_width, img_height),
    batch_size=batch_size,
    class_mode='categorical'  # Since we have multiple classes
)

# Build the CNN model
model = models.Sequential([
    layers.Conv2D(32, (3, 3), activation='relu', input_shape=(img_width,
img_height, 3)),
    layers.MaxPooling2D((2, 2)),

    layers.Conv2D(64, (3, 3), activation='relu'),
    layers.MaxPooling2D((2, 2)),

    layers.Conv2D(128, (3, 3), activation='relu'),
    layers.MaxPooling2D((2, 2)),

    layers.Flatten(),
    layers.Dense(128, activation='relu'),
    layers.Dropout(0.5),
    layers.Dense(3, activation='softmax')  # 3 classes, softmax for multi-class classification
])

# Compile the model
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

# Train the model
model.fit(train_generator, steps_per_epoch=train_generator.samples // batch_size, epochs=30)

# Save the model to a file
model.save('gesture_classification_model.h5')

print("Model trained and saved successfully!")
```

```python
# temp.py
import cv2
import mediapipe as mp

def detect_hand_box(frame):
    mp_hands = mp.solutions.hands
    with mp_hands.Hands(static_image_mode=False, max_num_hands=1,
min_detection_confidence=0.4) as hands:
        frame_rgb = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)
        results = hands.process(frame_rgb)

        if results.multi_hand_landmarks:
            for hand_landmarks in results.multi_hand_landmarks:
                x_coords = [lm.x for lm in hand_landmarks.landmark]
                y_coords = [lm.y for lm in hand_landmarks.landmark]
                height, width, _ = frame.shape
                xmin, xmax = int(min(x_coords) * width), int(max(x_coords) *
width)
                ymin, ymax = int(min(y_coords) * height), int(max(y_coords) *
height)
                return xmin, ymin, xmax, ymax
        return [0,0,640,480]
```

```python
import ollama
from io import BytesIO
import base64
from PIL import Image
import numpy as np

# Define the LLAVA model name and prompt
MODEL_NAME = "llava"
EXTRACTION_PROMPT = "Extract individual alphabets from this image. Give only
alphabets. Nothing else."

def extract_text_from_image(image):
    """
    Extracts text from an image using the LLAVA model from Ollama.

    Args:
        image (PIL.Image, numpy.ndarray, or file-like object): The image file
from which to extract text.

    Returns:
        str: Extracted text or an error message if extraction fails.
    """
    try:
        # Convert numpy array to PIL Image if necessary
        if isinstance(image, np.ndarray):
            image = Image.fromarray(image)

        # Convert image to base64 string for sending to Ollama
        buffered = BytesIO()
        image.save(buffered, format="JPEG")
        image_base64 = base64.b64encode(buffered.getvalue()).decode("utf-8")

        # Send the base64-encoded image and prompt to LLAVA for text extraction
        response = ollama.chat(
            model=MODEL_NAME,
            messages=[
                {
                    'role': 'user',
                    'content': EXTRACTION_PROMPT,
                    'images': [image_base64]
                }
            ]
        )
        # Return the content of the response
        return response['message']['content']
    except Exception as e:
        return f"Error: {str(e)}"
```

```python
# streamlit_app.py
import streamlit as st
import subprocess
import os
import cv2
from text_extractor import extract_text_from_image  # Import the function

# Initialize Streamlit session state variables
if 'note_maker_process' not in st.session_state:
    st.session_state['note_maker_process'] = None
if 'extract_frame' not in st.session_state:
    st.session_state['extract_frame'] = None

# Title for Streamlit interface
st.title("Hand Gesture-Based Note Maker")

# Function to start the note-making application
def start_note_maker():
    if st.session_state['note_maker_process'] is None:
        st.session_state['note_maker_process'] = subprocess.Popen(["python",
"app.py"])

# Function to stop the note-making application
def stop_note_maker():
    if st.session_state['note_maker_process']:
        st.session_state['note_maker_process'].terminate()
        st.session_state['note_maker_process'] = None

# Sidebar buttons
if st.sidebar.button("Start window"):
    start_note_maker()

if st.sidebar.button("Close window"):
    stop_note_maker()

if st.sidebar.button("Extract text"):
    if st.session_state['note_maker_process']:
        frame_path = "current_frame.jpg"
        if os.path.exists(frame_path):
            # Display the captured frame
            image = cv2.imread(frame_path)
            if image is not None:
                st.image(image, caption="Captured Frame",
use_container_width=True)
```

```python
                # Extract text using LLAVA
                extracted_text = extract_text_from_image(image)

                # Display extracted text
                st.text_area("Extracted Text:", value=extracted_text)
            else:
                st.error("Failed to load the captured frame. Please try again.")
        else:
            st.warning("No frame available for extraction.")
    else:
        st.warning("Please start the application window first.")
```