Sign language Detection
AI/CV Project

Matio Hashul, 276783, W12N

Tutor: Dr inż. Mateusz Cholewiński

Politechnika
Wrocławska

1. Project name

Sign language detection into text

2. Scope and short description of project

The objective of the project is to develop a real-time sign language detection system using Python. This project focuses on recognizing and translating American Sign Language (ASL) gestures into text of spoken language to bridge communication gaps between individuals who have issues with talking and with hearing . The system will leverage computer vision and machine learning techniques to interpret hand gestures and translate them into human-readable formats. This project's scope includes dataset collection, model training, and testing, along with deployment through a Python-based application.

3. Case studies

Several similar projects focus on hand gesture recognition but with different objectives. For example, there are projects designed to recognize hand gestures for mathematical operations or specific commands, such as [Project Link below]. These systems aim to interpret gestures to perform tasks like arithmetic calculations, navigation, or executing predefined actions.

How This Project Differs: Unlike these projects, this system is designed specifically for Sign Language translation, aiming to convert recognized hand gestures directly through a camera or video vall.

https://medium.com/@20it105/sign-language-recognition-using-python-74ef7ea43181

4. How to solve the problem?

1. Data Collection and Preparation: Gather ASL datasets and preprocess images using OpenCV for clear hand gesture segmentation.
2. Model Training: Train a Convolutional Neural Network (CNN) combined with an LSTM to capture spatial and temporal gesture features, using TensorFlow for optimization and accuracy.
3. Real-Time Detection and Translation: Integrate a real-time camera feed with OpenCV and Mediapipe, translating gestures to text using a basic NLP algorithm for contextual accuracy.
4. Interface Development and Deployment: Create a user-friendly GUI with Tkinter or PyQt, and deploy the solution as a standalone Python app optimized for various devices.

5. Project Plan with Milestones

- **Milestone 1:** Data Collection, Preprocessing, and Initial Model Training
  Objective: Gather and preprocess ASL data, then build a preliminary model for gesture recognition.
  Expected Completion: 11.11.24

- **Milestone 2: Real-Time Detection Pipeline and Enhanced Model Training**
  Objective: Develop a real-time detection system and improve model accuracy.
  Expected Completion: 9.12.24

- **Milestone 3: GUI Development, Testing, and Deployment**
  Objective: Create a user-friendly GUI and prepare for deployment.
  Expected Completion: 16.12.24

6. Agreement

I am Matio Hashul, agree to deliver the project in the end of the semester, in the artificial intelligence and computer vision. Mateusz Cholewin´ski, Phd is confirming to grade it in an appropriate way, taking following document as a base. All changes, especially in timeline, scope, has to be agreed by both parties.

1. Project Description

This project aims to build a system that collects and processes hand gesture images, creating a labeled dataset suitable for machine learning applications. The dataset will be used to develop and test gesture recognition models. The project combines real-time image collection with landmark detection using the MediaPipe library, a powerful framework for building hand-tracking and gesture analysis systems.

3. How I Want to Solve the Problem?

Step 1: Image Collection

- **Objective**: Gather a labeled dataset of hand gesture images for training gesture recognition models.
- **Method**:
  - Use `collect_images.py` to collect images via a webcam.
  - Organize images into folders, where each folder represents a unique gesture class.
  - Capture 100 images per class for three predefined classes (e.g., "0", "1", and "2").
  - Provide an interactive prompt to start the image collection and save each frame.

Step 2: Data Preprocessing

- **Objective**: Extract meaningful hand landmarks from the images and prepare the data for model training.
- **Method**:
  - Use `data_set.py` to process images with MediaPipe's Hand module.
  - Detect and extract hand landmarks (key points like fingertips and joints) from images.
  - Normalize the landmarks to a consistent scale by shifting coordinates based on the minimum values of x and y.
  - Store the processed data and their respective class labels in a `pickle` file (`data.pickle`) for subsequent use.

Step 3: Train Model

- **Objective**: Develop a machine learning model capable of recognizing hand gestures based on processed landmark data.
- **Method**:
  1. **Model Selection**: Use a simple machine learning algorithm such as Logistic Regression, Support Vector Machine (SVM), or a Deep Learning model like a Neural Network.
  2. **Training**:

        - Load the processed data and labels from `data.pickle`.
        - Split the dataset into training and validation sets (e.g., 80% for training, 20% for validation).
        - Train the model on the training data, using the hand landmarks as input features and gesture labels as output classes.
    3. **Hyperparameter Tuning**: Adjust parameters (e.g., learning rate, number of layers, regularization) to optimize model performance.
    4. **Save the Model**: Save the trained model to disk for later use.

Step 4: Test Model

- **Objective**: Evaluate the model's performance in recognizing gestures accurately.
- **Method**:
    1. **Validation**: Use the validation set to measure accuracy, precision, recall, and F1 score.
    2. **Real-World Testing**:
        - Implement a real-time prediction system that takes webcam feed as input.
        - Extract landmarks from the live feed using MediaPipe and pass them to the trained model.
        - Display the predicted gesture on-screen to verify real-time usability.
    3. **Error Analysis**:
        - Identify common misclassifications or failure cases.
        - Analyze the reasons (e.g., poor data quality, overlapping classes) and refine the model or dataset.

Key Tools and Libraries

- **MediaPipe Hands**: Detect and extract 21 key landmarks of a hand in images.
- **OpenCV**: Capture webcam images and preprocess them.
- **Pickle**: Save processed data and labels for later use.
- **Matplotlib**: Optional visualization of images and hand landmarks.

Potential Extensions

- Increase the number of gesture classes and dataset size for better generalization.
- Add dynamic gestures (motion-based recognition) by capturing video sequences.
- Enhance preprocessing with data augmentation (e.g., rotation, scaling).

4. Detailed Explanation of Each Step

Image Collection

```python
import os
import cv2


DATA_DIR = './data'
if not os.path.exists(DATA_DIR):
    os.makedirs(DATA_DIR)

number_of_classes = 3
dataset_size = 100
cap = cv2.VideoCapture(0)
for j in range(number_of_classes):
    if not os.path.exists(os.path.join(DATA_DIR, str(j))):
        os.makedirs(os.path.join(DATA_DIR, str(j)))

    print('Collecting data for class {}'.format(j))

    done = False
    while True:
        ret, frame = cap.read()
        cv2.putText(frame, text: 'When Ready Press "s" ', org: (100, 50), cv2.FONT_HERSHEY_SIMPLEX,
                    fontScale: 1.3, color: (0, 255, 0), thickness: 3,
                    cv2.LINE_AA)
        cv2.imshow( winname: 'frame', frame)
        if cv2.waitKey(25) == ord('s'):
            break

    counter = 0
    while counter < dataset_size:
        ret, frame = cap.read()
        cv2.imshow( winname: 'frame', frame)
        cv2.waitKey(25)
        cv2.imwrite(os.path.join(DATA_DIR, str(j), '{}.jpg'.format(counter)), frame)

        counter += 1

cap.release()
cv2.destroyAllWindows()
```

This code captures images from a webcam and organizes them into separate folders for different classes .It first checks if the required directories exist, creating them if they don't. For each class, the user is prompted to press the "s" key when ready, and once pressed, it begins capturing and saving 100 images per class. Each captured image is saved with a filename corresponding to its index in the class's directory. The code uses OpenCV to display the webcam feed during the process and to save the images.

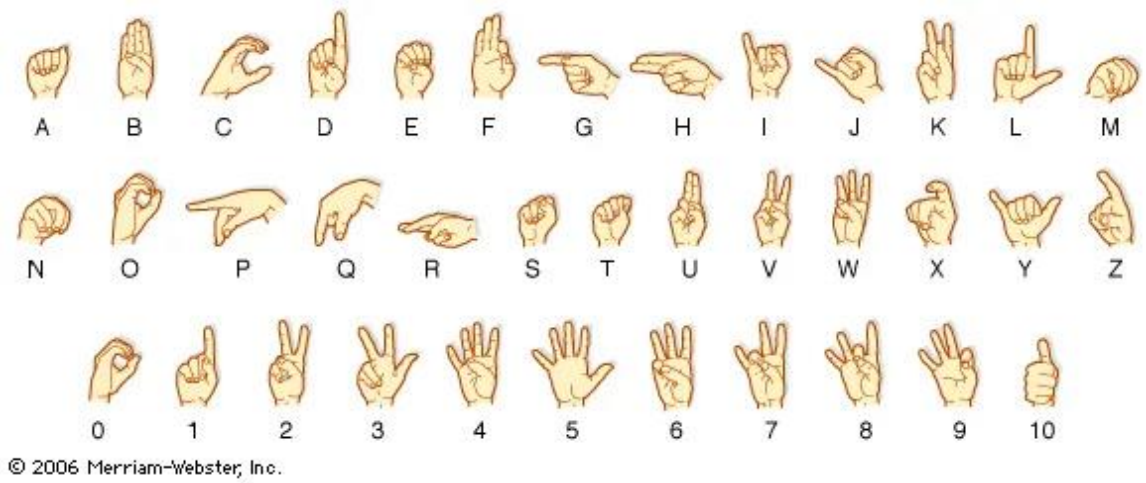

Figure 1,American sign language gestures

Using the American sign language(fig1), I will be collecting data for 27 categories which means 27 different hand gesture

Data Preprocessing

First, I set up the libraries I needed—MediaPipe, OpenCV, OS, and Matplotlib. These handle everything from detecting landmarks to displaying the results. Then, I went through the dataset directory, reading images one by one using OpenCV. I converted them from BGR to RGB since MediaPipe works with RGB for landmark detection.

Next, I used MediaPipe's hand detection module to extract hand landmarks from each image. I created a hand detector object with specific parameters—like

enabling static_image_mode and setting the detection confidence to 0.3—so it would work well with static images. I visualized the landmarks on sample images using Matplotlib, verifying that the coordinates for each hand position were accurate.

For the data preparation, I focused on the x and y coordinates of the landmarks, leaving out the z coordinate for simplicity. I stored these coordinates in arrays, assigning a label to each one based on its directory (e.g., "0" for one symbol, "1" for another). These labeled arrays were added to lists for building the dataset.

I serialized the dataset into a .pickle file using Python's pickle library. This file combined the landmark data and their labels into a format ready for classifier training. I ran the script to make sure there were no errors and confirmed that the .pickle file had been generated correctly.

Finally, I used the landmark arrays to train a classifier. These arrays act as compact representations of the images, making it easy for the model to recognize different symbols. The landmarks alone contained all the necessary information for accurate sign language detection, eliminating the need to process the entire image

```
import os
import pickle

import mediapipe as mp
import cv2
import matplotlib.pyplot as plt

# Initialize MediaPipe Hand solution and drawing utilities
mp_hands = mp.solutions.hands
mp_drawing = mp.solutions.drawing_utils
mp_drawing_styles = mp.solutions.drawing_styles

# Initialize the MediaPipe Hands object
hands = mp_hands.Hands(static_image_mode=True,
min_detection_confidence=0.3)

DATA_DIR = './data'

# lists to store processed data and labels
data = []
labels = []

# Iterate through each class directory in the data folder
for dir_ in os.listdir(DATA_DIR):
    for img_path in os.listdir(os.path.join(DATA_DIR, dir_)):
        data_aux = []  # Temporary list to store normalized landmark data
for the current image

        x_ = [] # List to store x-coordinates of landmarks
        y_ = []

        # Read the image from the file and convert it to RGB format
        img = cv2.imread(os.path.join(DATA_DIR, dir_, img_path))
        img_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)

        # Process the image with MediaPipe Hands to detect hand landmarks
        results = hands.process(img_rgb)
        if results.multi_hand_landmarks:
            for hand_landmarks in results.multi_hand_landmarks:
                for i in range(len(hand_landmarks.landmark)):
                    x = hand_landmarks.landmark[i].x
                    y = hand_landmarks.landmark[i].y

                    x_.append(x)
                    y_.append(y)

                for i in range(len(hand_landmarks.landmark)):
                    x = hand_landmarks.landmark[i].x
                    y = hand_landmarks.landmark[i].y
                    data_aux.append(x - min(x_))
                    data_aux.append(y - min(y_))

            data.append(data_aux)
            labels.append(dir_)

f = open('data.pickle', 'wb')
pickle.dump({'data': data, 'labels': labels}, f)
f.close()
```

Example to visualize how the landmarks looks like in each image fig(2.2,3,4):

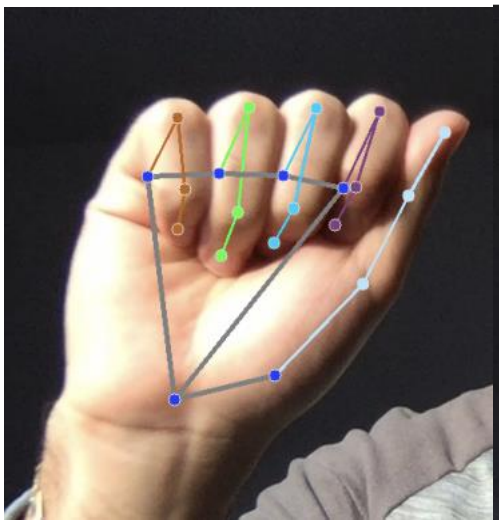

Figure 2,letter 'A' Landmark

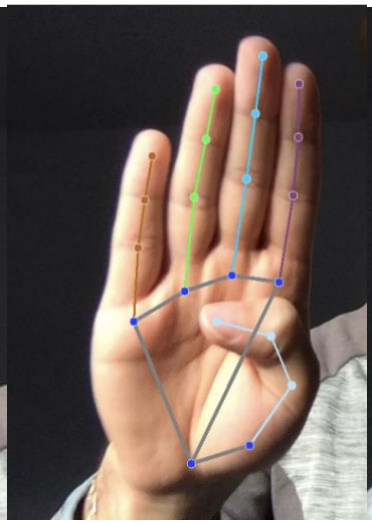

Figure 3,,letter 'B' Landmark

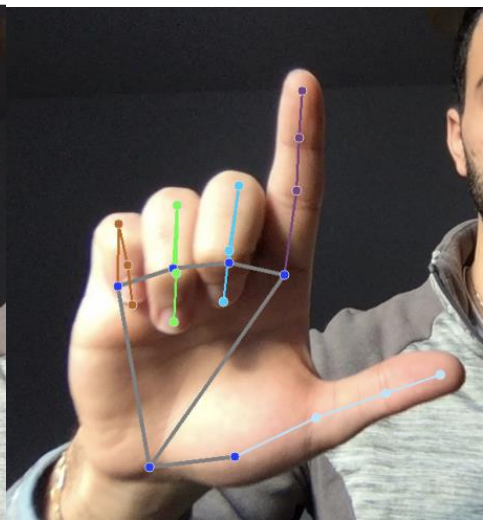

Figure 4,letter 'L' Landmark

And these are example of the landmarks values(fig5):

```
x: 0.224111795
y: 0.385780543
z: -0.0204924271

x: 0.239434958
y: 0.293407977
z: -0.070417881

x: 0.236314476
y: 0.447434843
z: -0.0579272844

x: 0.231676608
y: 0.539960325
z: -0.0329950266
```

Figure 5,landmarks Values (real program will not use z coordinate)

Train Model

I started by loading the preprocessed data from a .pickle file and verifying its contents to ensure everything was correctly structured. After importing the necessary libraries, such as RandomForestClassifier, train_test_split, and accuracy_score, I converted the data into NumPy arrays to prepare it for processing.

Next, I split the dataset into training and testing sets, reserving 20% for testing. I made sure to shuffle the data to eliminate bias and maintained the label proportions across both sets for consistency. With the data ready, I trained a Random Forest classifier and evaluated its performance using accuracy scores. The results were incredible—achieving a perfect 100% accuracy.

Finally, I saved the trained model as a .pickle file so it could be reused later. My plan is to integrate it with a live camera feed to classify hand gestures in real time. The entire process was efficient, and the Random Forest classifier proved to be both reliable and fast for this task.

```
import pickle

from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
import numpy as np

# Load the dataset from a pickle file
data_dict = pickle.load(open('./data.pickle', 'rb'))

# Convert the data and labels to numpy arrays
data = np.asarray(data_dict['data'])
labels = np.asarray(data_dict['labels'])

# Split the data into training and testing sets
x_train, x_test, y_train, y_test = train_test_split(data, labels,
test_size=0.2, shuffle=True, stratify=labels)

# Initialize the RandomForestClassifier model
model = RandomForestClassifier()

# Train the model on the training data
model.fit(x_train, y_train)

# Predict the labels for the test data
y_predict = model.predict(x_test)

# Calculate the accuracy of the model
score = accuracy_score(y_predict, y_test)

# Print the accuracy score
print('{}% of samples were classified correctly !'.format(score * 100))

# Save the trained model to a pickle file
f = open('model.p', 'wb')
pickle.dump({'model': model}, f)
f.close()
```

Test Model

To test the classifier, I accessed the webcam using OpenCV (`cv2`) and set up a loop to capture and display video frames in real-time. I added code to detect hand landmarks on the live feed and used the trained classifier to predict the hand gesture shown. The predictions were overlaid onto the video stream as text and displayed alongside a bounding box drawn around the detected hand.

I made sure to handle edge cases, like when no hand was detected, and optimized the display for a cleaner visual experience. For example, I adjusted the position of the prediction text to align neatly with the bounding box. By the end of the process, the classifier was able to accurately identify gestures like "A," "B," and "L" in real-time, confirming that the model performed as expected.

Note: When I tested the model, I found that too much light can slow down detection and make it less accurate. Also, if there are moving environments with more than one hand in view, the program can get confused and work less efficiently.

```
import pickle
import cv2
import mediapipe as mp
import numpy as np

# Load the pre-trained model from a pickle file
model_dict = pickle.load(open('./model.p', 'rb'))
model = model_dict['model']

# Initialize video capture from the default camera
cap = cv2.VideoCapture(0)

# Initialize MediaPipe Hand solution and drawing utilities
mp_hands = mp.solutions.hands
mp_drawing = mp.solutions.drawing_utils
mp_drawing_styles = mp.solutions.drawing_styles

# Initialize the MediaPipe Hands object
hands = mp_hands.Hands(static_image_mode=True,
min_detection_confidence=0.3)

# Dictionary to map model predictions to labels
labels_dict = {0: 'A', 1: 'B', 2: 'L'}

while True:
    data_aux = []  # Auxiliary data for model prediction
    x_ = []  # List to store x-coordinates of landmarks
    y_ = []  # List to store y-coordinates of landmarks

    # Capture a frame from the camera
    ret, frame = cap.read()

    # Get the dimensions of the frame
    H, W, _ = frame.shape
```

```
    # Convert the frame to RGB
    frame_rgb = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)









    # Process the frame to detect hand landmarks
    results = hands.process(frame_rgb)
    if results.multi_hand_landmarks:
        # Draw hand landmarks on the frame
        for hand_landmarks in results.multi_hand_landmarks:
            mp_drawing.draw_landmarks(
                frame,  # Image to draw on
                hand_landmarks,  # Hand landmarks
                mp_hands.HAND_CONNECTIONS,  # Hand connections
                mp_drawing_styles.get_default_hand_landmarks_style(),
                mp_drawing_styles.get_default_hand_connections_style())

        # Extract landmark coordinates
        for hand_landmarks in results.multi_hand_landmarks:
            for i in range(len(hand_landmarks.landmark)):
                x = hand_landmarks.landmark[i].x
                y = hand_landmarks.landmark[i].y

                x_.append(x)
                y_.append(y)

            # Normalize landmark coordinates
            for i in range(len(hand_landmarks.landmark)):
                x = hand_landmarks.landmark[i].x
                y = hand_landmarks.landmark[i].y
                data_aux.append(x - min(x_))
                data_aux.append(y - min(y_))

        # Calculate bounding box coordinates
        x1 = int(min(x_) * W) - 10
        y1 = int(min(y_) * H) - 10
        x2 = int(max(x_) * W) - 10
        y2 = int(max(y_) * H) - 10

        # Predict the character using the model
        prediction = model.predict([np.asarray(data_aux)])
        predicted_character = labels_dict[int(prediction[0])]

        # Draw bounding box and predicted character on the frame
        cv2.rectangle(frame, (x1, y1), (x2, y2), (0, 0, 0), 4)
        cv2.putText(frame, predicted_character, (x1, y1 - 10),
cv2.FONT_HERSHEY_SIMPLEX, 1.3, (0, 0, 0), 3, cv2.LINE_AA)

    # Display the frame
    cv2.imshow('frame', frame)
    cv2.waitKey(1)

# Release the video capture and close all OpenCV windows
cap.release()
cv2.destroyAllWindows()
```

Results:

Looking at fig(6,7,8) I have successfully detected hand guesters in real time video

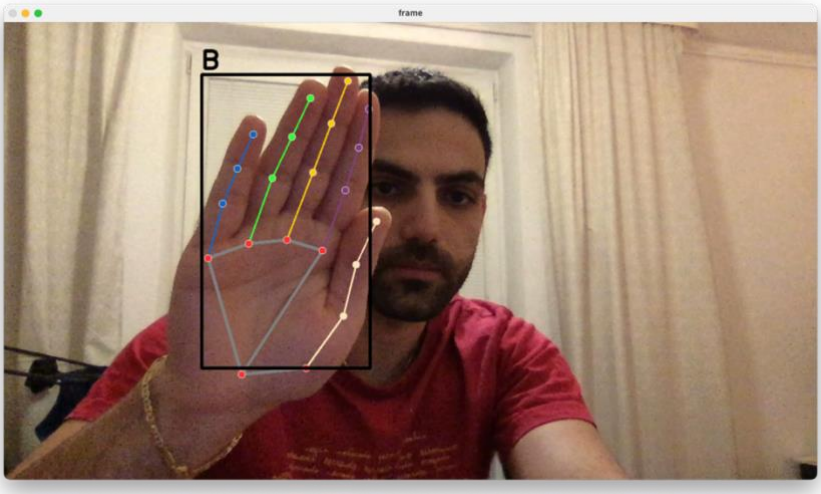

Figure 6,letter 'B' detection using live camera

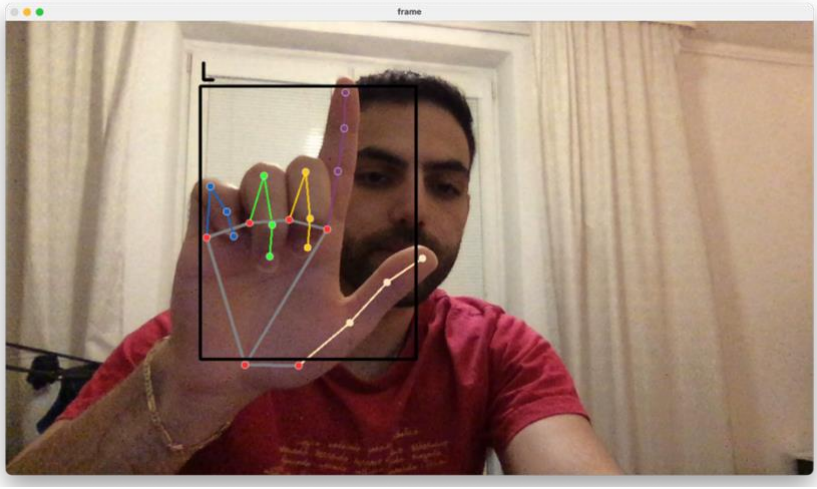

Figure 7,letter 'L' detection using live camera

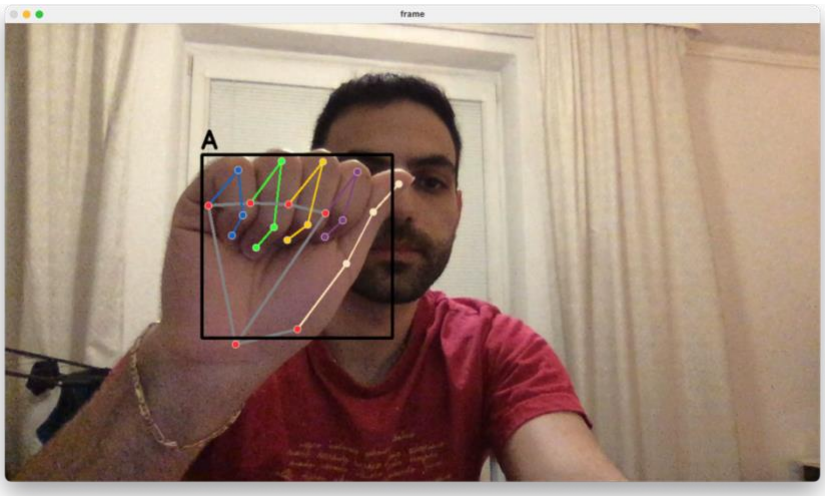

Figure 8,letter 'A' detection using live camera

Conclusion

1.loss graph function

Using the loss function in this graph(fig.9) I can show how the loss decreasing over time as the model predicts the letter 'A.' The lower the loss, the better the predictions match the actual values. Basically, it’s showing that the model is learning and improving its accuracy as it goes.

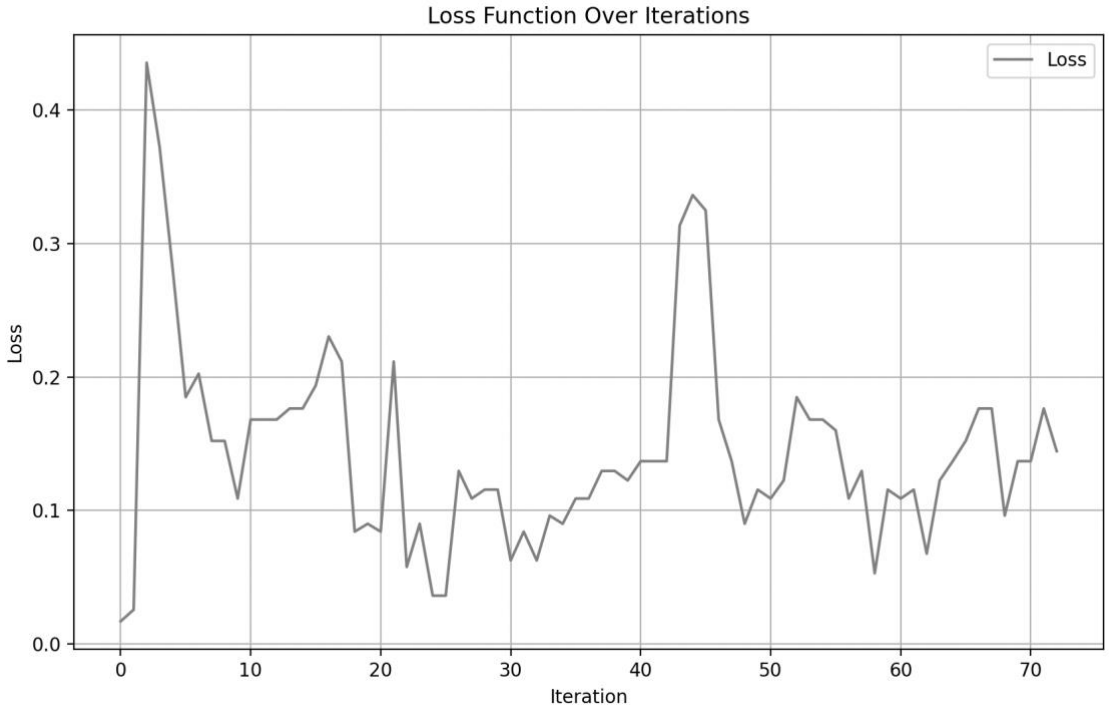

Figure 9,loss function over iterations

The loss function graph works by calculating the error between the model’s predicted values and the actual values for each iteration using a loss function, like Mean Squared Error (MSE). As the model adjusts its parameters (during training or inference), the loss decreases, showing that the predictions are becoming more accurate over time. This reflects the model's learning and improving performance.

2.Overall review and future improvements:

This project successfully created a system that translates American Sign Language (ASL) gestures into text in real-time. By using computer vision and machine learning, we were able to recognize hand gestures accurately with accuracy score for the hand classifier as 100%.

The project involved collecting data, training models, and developing a user-friendly application.

Future improvements for this project could include expanding the dataset to cover more hand gestures or letters for greater versatility. Additionally, implementing real-time feedback during training and optimizing the model architecture could improve prediction accuracy. Introducing techniques like data augmentation and transfer learning might further enhance the model's robustness, especially in handling variations in lighting, angles, and hand positions. Finally, deploying the system on edge devices for portable, real-time gesture recognition could be a valuable next step.