**DATASET:**

import cv2

import mediapipe as mp

import pandas as pd

import os

cap = cv2.VideoCapture(0)

if not cap.isOpened():

print("Error: Could not open video capture.")

exit()

mpPose = mp.solutions.pose

pose = mpPose.Pose()

mpDraw = mp.solutions.drawing\_utils

lm\_list = []

label = "wrong\_bicep\_final"

no\_of\_frames = 1000

# Full list of landmarks for the upper body

LANDMARKS\_TO\_USE = [

mpPose.PoseLandmark.NOSE,

mpPose.PoseLandmark.LEFT\_EYE,

mpPose.PoseLandmark.RIGHT\_EYE,

mpPose.PoseLandmark.LEFT\_EAR,

mpPose.PoseLandmark.RIGHT\_EAR,

mpPose.PoseLandmark.LEFT\_SHOULDER,

mpPose.PoseLandmark.RIGHT\_SHOULDER,

mpPose.PoseLandmark.LEFT\_ELBOW,

mpPose.PoseLandmark.RIGHT\_ELBOW,

mpPose.PoseLandmark.LEFT\_WRIST,

mpPose.PoseLandmark.RIGHT\_WRIST,

mpPose.PoseLandmark.LEFT\_PINKY,

mpPose.PoseLandmark.RIGHT\_PINKY,

mpPose.PoseLandmark.LEFT\_INDEX,

mpPose.PoseLandmark.RIGHT\_INDEX,

mpPose.PoseLandmark.LEFT\_THUMB,

mpPose.PoseLandmark.RIGHT\_THUMB,

mpPose.PoseLandmark.LEFT\_HIP,

mpPose.PoseLandmark.RIGHT\_HIP,

]

def make\_landmark\_timestep(results):

"""Extract specific pose landmarks for the overhead press."""

c\_lm = []

if results.pose\_landmarks:

for idx in LANDMARKS\_TO\_USE:

lm = results.pose\_landmarks.landmark[idx]

c\_lm.append(lm.x)

c\_lm.append(lm.y)

c\_lm.append(lm.z)

c\_lm.append(lm.visibility)

return c\_lm

def draw\_landmark\_on\_image(mpDraw, results, frame):

"""Draw landmarks and connections on the frame."""

if results.pose\_landmarks:

mpDraw.draw\_landmarks(frame, results.pose\_landmarks, mpPose.POSE\_CONNECTIONS)

for lm in results.pose\_landmarks.landmark:

h, w, c = frame.shape

cx, cy = int(lm.x \* w), int(lm.y \* h)

cv2.circle(frame, (cx, cy), 3, (0, 255, 0), cv2.FILLED)

return frame

# Check if the file exists

csv\_file = label + ".csv"

file\_exists = os.path.exists(csv\_file)

while len(lm\_list) < no\_of\_frames:

ret, frame = cap.read()

if ret:

frameRGB = cv2.cvtColor(frame, cv2.COLOR\_BGR2RGB)

results = pose.process(frameRGB)

lm = make\_landmark\_timestep(results)

if lm:

lm\_list.append(lm)

frame = draw\_landmark\_on\_image(mpDraw, results, frame)

cv2.imshow("image", frame)

if cv2.waitKey(1) == ord('q'):

break

else:

print("Error: Failed to capture image.")

break

# Convert the list to a DataFrame

df = pd.DataFrame(lm\_list)

# Append data to the CSV file

df.to\_csv(csv\_file, mode='a', index=False, header=not file\_exists)

# Release resources

cap.release()

cv2.destroyAllWindows()

**MODEL TRAINING:**

# Imports

import numpy as np

import pandas as pd

import tensorflow as tf

from tensorflow.keras.layers import LSTM, Dense, Dropout

from tensorflow.keras.models import Sequential

from tensorflow.keras.callbacks import EarlyStopping

from sklearn.model\_selection import train\_test\_split

from sklearn.metrics import confusion\_matrix

import matplotlib.pyplot as plt

import seaborn as sns

# Load CSV files

neutral\_df = pd.read\_csv("neutral\_bicep\_final.csv")

correct\_df = pd.read\_csv("correct\_bicep\_final.csv")

wrong\_df = pd.read\_csv("wrong\_bicep\_final.csv")

# Parameters

X = []

y = []

no\_of\_timesteps = 20

# Data processing function

def process\_data(df, label):

datasets = df.iloc[:, :].values # Extract all columns as numpy array

n\_samples = len(datasets)

for i in range(no\_of\_timesteps, n\_samples):

X.append(datasets[i-no\_of\_timesteps:i, :])

y.append(label)

# Process each dataset

process\_data(neutral\_df, 0) # Neutral posture labeled as 0

process\_data(correct\_df, 1) # Correct overhead press labeled as 1

process\_data(wrong\_df, 2) # Wrong overhead press labeled as 2

# Convert to numpy arrays

X, y = np.array(X), np.array(y)

print("Shape of X:", X.shape) # (Samples, Timesteps, Features)

print("Shape of y:", y.shape) # (Samples,)

# Split data into training and validation sets

X\_train, X\_val, y\_train, y\_val = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# Define the model

model = Sequential()

model.add(LSTM(units=64, return\_sequences=True, input\_shape=(X.shape[1], X.shape[2])))

model.add(Dropout(0.3))

model.add(LSTM(units=64))

model.add(Dropout(0.3))

model.add(Dense(units=3, activation="softmax"))

# Compile the model

model.compile(optimizer="adam", metrics=["accuracy"], loss="sparse\_categorical\_crossentropy")

# EarlyStopping callback

early\_stopping = EarlyStopping(

monitor='val\_loss',

patience=5,

restore\_best\_weights=True

)

# Train the model

history = model.fit(

X\_train, y\_train,

epochs=15, # Train for up to 15 epochs

batch\_size=28,

validation\_data=(X\_val, y\_val),

callbacks=[early\_stopping],

verbose=1

)

# Evaluate the model

val\_loss, val\_accuracy = model.evaluate(X\_val, y\_val, verbose=0)

print(f"Validation Accuracy: {val\_accuracy \* 100:.2f}%")

print(f"Validation Loss: {val\_loss:.4f}")

# Confusion Matrix

y\_pred = model.predict(X\_val, verbose=0).argmax(axis=1)

cm = confusion\_matrix(y\_val, y\_pred)

# Save the model

model.save("lstm\_Bicep\_curls\_model.keras", save\_format='keras')

print("Model saved to 'lstm\_Bicep\_curls\_model.keras'")

# Plot Confusion Matrix

plt.figure(figsize=(8, 6))

sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=['Neutral', 'Correct', 'Wrong'], yticklabels=['Neutral', 'Correct', 'Wrong'])

plt.title('Confusion Matrix')

plt.xlabel('Predicted Labels')

plt.ylabel('True Labels')

plt.show()

# Plot Accuracy and Loss

plt.figure(figsize=(12, 6))

plt.plot(history.history['accuracy'], label="Training Accuracy")

plt.plot(history.history['val\_accuracy'], label="Validation Accuracy")

plt.title('Training and Validation Accuracy')

plt.xlabel('Epochs')

plt.ylabel('Accuracy')

plt.legend()

plt.grid()

plt.show()

plt.figure(figsize=(12, 6))

plt.plot(history.history['loss'], label="Training Loss")

plt.plot(history.history['val\_loss'], label="Validation Loss")

plt.title('Training and Validation Loss')

plt.xlabel('Epochs')

plt.ylabel('Loss')

plt.legend()

plt.grid()

plt.show()

**REAL TIME RUNNNING ON JUPTYER:**

import sys

import cv2

import mediapipe as mp

import numpy as np

import tensorflow as tf

from collections import deque

from PyQt5.QtWidgets import QApplication, QLabel, QVBoxLayout, QWidget, QPushButton

from PyQt5.QtGui import QImage, QPixmap, QFont

from PyQt5.QtCore import QTimer, Qt

# Load the trained model

model = tf.keras.models.load\_model("lstm\_Bicep\_curls\_model.keras")

# Define class labels

class\_labels = {0: "neutral", 1: "bicep\_curl", 2: "wrong"}

# Initialize MediaPipe Pose

mpPose = mp.solutions.pose

pose = mpPose.Pose()

mpDraw = mp.solutions.drawing\_utils

# Smoothing parameters

angle\_smoothing = deque(maxlen=5)

def calculate\_angle(a, b, c):

"""Calculate the angle between three points."""

a, b, c = np.array(a), np.array(b), np.array(c)

radians = np.arctan2(c[1] - b[1], c[0] - b[0]) - np.arctan2(a[1] - b[1], a[0] - b[0])

angle = np.abs(radians \* 180.0 / np.pi)

return 360.0 - angle if angle > 180.0 else angle

def make\_landmark\_timestep(results):

"""Convert pose landmarks to a flat list of selected features."""

landmarks = results.pose\_landmarks.landmark

relevant\_landmarks = [

mpPose.PoseLandmark.NOSE,

mpPose.PoseLandmark.LEFT\_EYE,

mpPose.PoseLandmark.RIGHT\_EYE,

mpPose.PoseLandmark.LEFT\_EAR,

mpPose.PoseLandmark.RIGHT\_EAR,

mpPose.PoseLandmark.LEFT\_SHOULDER,

mpPose.PoseLandmark.RIGHT\_SHOULDER,

mpPose.PoseLandmark.LEFT\_ELBOW,

mpPose.PoseLandmark.RIGHT\_ELBOW,

mpPose.PoseLandmark.LEFT\_WRIST,

mpPose.PoseLandmark.RIGHT\_WRIST,

mpPose.PoseLandmark.LEFT\_PINKY,

mpPose.PoseLandmark.RIGHT\_PINKY,

mpPose.PoseLandmark.LEFT\_INDEX,

mpPose.PoseLandmark.RIGHT\_INDEX,

mpPose.PoseLandmark.LEFT\_THUMB,

mpPose.PoseLandmark.RIGHT\_THUMB,

mpPose.PoseLandmark.LEFT\_HIP,

mpPose.PoseLandmark.RIGHT\_HIP,

]

# Extract x, y, z, and visibility for each relevant landmark

return [val for lm\_id in relevant\_landmarks for val in (

landmarks[lm\_id].x,

landmarks[lm\_id].y,

landmarks[lm\_id].z,

landmarks[lm\_id].visibility)]

class BicepCurlApp(QWidget):

def \_\_init\_\_(self):

super().\_\_init\_\_()

self.setWindowTitle("Bicep Curl AI Trainer")

self.setGeometry(100, 100, 900, 700)

self.setStyleSheet("background-color: #f0f4f8;") # Light gray background

# GUI Components

self.layout = QVBoxLayout()

# Video display

self.video\_label = QLabel(self)

self.video\_label.setAlignment(Qt.AlignCenter)

self.video\_label.setStyleSheet("border: 3px solid #0078D7; border-radius: 10px;") # Blue border

self.layout.addWidget(self.video\_label)

# Feedback label

self.feedback\_label = QLabel("Feedback: Initializing...", self)

self.feedback\_label.setFont(QFont("Arial", 16, QFont.Bold))

self.feedback\_label.setStyleSheet("color: #0078D7; padding: 10px;")

self.layout.addWidget(self.feedback\_label)

# Close button

self.close\_button = QPushButton("Close", self)

self.close\_button.setFont(QFont("Arial", 14))

self.close\_button.setStyleSheet("""

QPushButton {

background-color: #0078D7;

color: white;

padding: 10px 20px;

border: none;

border-radius: 10px;

}

QPushButton:hover {

background-color: #005A9E;

}

""")

self.close\_button.clicked.connect(self.close\_application)

self.layout.addWidget(self.close\_button)

self.setLayout(self.layout)

# Initialize video capture

self.cap = cv2.VideoCapture(0)

self.timer = QTimer()

self.timer.timeout.connect(self.update\_frame)

self.timer.start(30) # 30ms per frame

# Other Variables

self.lm\_list = []

self.rep\_counter = 0

self.arms\_extended = True

self.label = "neutral"

def update\_frame(self):

ret, frame = self.cap.read()

if not ret:

self.feedback\_label.setText("Error: Unable to access webcam.")

return

frameRGB = cv2.cvtColor(frame, cv2.COLOR\_BGR2RGB)

results = pose.process(frameRGB)

if results.pose\_landmarks:

lm = make\_landmark\_timestep(results)

self.lm\_list.append(lm)

if len(self.lm\_list) == 20: # Sequence length for LSTM

lm\_array = np.expand\_dims(np.array(self.lm\_list), axis=0)

prediction = model.predict(lm\_array)

predicted\_class = np.argmax(prediction[0])

self.label = class\_labels.get(predicted\_class, "unknown")

self.lm\_list = []

landmarks = results.pose\_landmarks.landmark

shoulder = [landmarks[mpPose.PoseLandmark.LEFT\_SHOULDER.value].x,

landmarks[mpPose.PoseLandmark.LEFT\_SHOULDER.value].y]

elbow = [landmarks[mpPose.PoseLandmark.LEFT\_ELBOW.value].x,

landmarks[mpPose.PoseLandmark.LEFT\_ELBOW.value].y]

wrist = [landmarks[mpPose.PoseLandmark.LEFT\_WRIST.value].x,

landmarks[mpPose.PoseLandmark.LEFT\_WRIST.value].y]

# Calculate the angle

elbow\_angle = calculate\_angle(shoulder, elbow, wrist)

angle\_smoothing.append(elbow\_angle)

smoothed\_angle = np.mean(angle\_smoothing)

# Repetition logic

if self.label == "bicep\_curl":

if smoothed\_angle <= 30: # Fully curled position

self.arms\_extended = False

elif not self.arms\_extended and smoothed\_angle >= 170: # Fully extended position

self.rep\_counter += 1

self.arms\_extended = True

# Draw landmarks

mpDraw.draw\_landmarks(frame, results.pose\_landmarks, mpPose.POSE\_CONNECTIONS)

# Draw feedback

feedback\_text = f"Label: {self.label} | Reps: {self.rep\_counter}"

self.feedback\_label.setText(feedback\_text)

frame = cv2.cvtColor(frame, cv2.COLOR\_BGR2RGB)

height, width, channel = frame.shape

q\_img = QImage(frame.data, width, height, channel \* width, QImage.Format\_RGB888)

self.video\_label.setPixmap(QPixmap.fromImage(q\_img))

def close\_application(self):

self.cap.release()

self.close()

if \_\_name\_\_ == "\_\_main\_\_":

app = QApplication(sys.argv)

window = BicepCurlApp()

window.show()

sys.exit(app.exec\_())

**REAL TIME RUNNING ON WEB:**

<!DOCTYPE html>

<html>

<head>

    <title>Webcam Bicep Curl Detection</title>

</head>

<body>

    <h1>Webcam Bicep Curl Detection</h1>

    <video id="webcam" autoplay></video>

    <p id="prediction">Predicted Posture: </p>

    <p id="count">Repetitions: 0</p>

    <p id="delay">Processing Delay: 0 ms</p>

    <script src="https://cdn.jsdelivr.net/npm/@mediapipe/pose"></script>

    <script src="https://cdn.jsdelivr.net/npm/@mediapipe/camera\_utils"></script>

    <script>

        const video = document.getElementById('webcam');

        const predictionElem = document.getElementById('prediction');

        const countElem = document.getElementById('count');

        const delayElem = document.getElementById('delay');

        const ws = new WebSocket('ws://127.0.0.1:8000/ws');

        let curlCount = 0;

        let isCurling = false;

        let frameTimestamp = 0;

        ws.onopen = () => {

            console.log('WebSocket connection established');

        };

        ws.onmessage = (event) => {

            const data = JSON.parse(event.data);

            console.log('Received data:', data);

            if (data.predicted\_posture) {

                predictionElem.innerText = `Predicted Posture: ${data.predicted\_posture}`;

            } else if (data.error) {

                console.error('Server error:', data.error);

            }

        };

        ws.onclose = () => {

            console.log('WebSocket connection closed');

        };

        ws.onerror = (error) => {

            console.error('WebSocket error:', error);

        };

        async function startWebcam() {

            try {

                const constraints = { video: true };

                const stream = await navigator.mediaDevices.getUserMedia(constraints);

                video.srcObject = stream;

                console.log('Webcam started');

            } catch (err) {

                console.error('Error accessing webcam:', err);

            }

        }

        function getLandmarks(poseLandmarks) {

            if (poseLandmarks) {

                return poseLandmarks.flatMap(lm => [lm.x, lm.y, lm.z, lm.visibility]);

            }

            return [];

        }

        function calculateElbowAngle(shoulder, elbow, wrist) {

            const radians = Math.atan2(wrist.y - elbow.y, wrist.x - elbow.x) -

                            Math.atan2(shoulder.y - elbow.y, shoulder.x - elbow.x);

            return Math.abs(radians \* 180 / Math.PI);

        }

        function updateCurlCount(landmarks) {

    // Left side (shoulder, elbow, wrist)

    const leftShoulder = { x: landmarks[5 \* 4], y: landmarks[5 \* 4 + 1] };

    const leftElbow = { x: landmarks[7 \* 4], y: landmarks[7 \* 4 + 1] };

    const leftWrist = { x: landmarks[9 \* 4], y: landmarks[9 \* 4 + 1] };

    // Right side (shoulder, elbow, wrist)

    const rightShoulder = { x: landmarks[6 \* 4], y: landmarks[6 \* 4 + 1] };

    const rightElbow = { x: landmarks[8 \* 4], y: landmarks[8 \* 4 + 1] };

    const rightWrist = { x: landmarks[10 \* 4], y: landmarks[10 \* 4 + 1] };

    // Calculate angles for both arms

    const leftAngle = calculateElbowAngle(leftShoulder, leftElbow, leftWrist);

    const rightAngle = calculateElbowAngle(rightShoulder, rightElbow, rightWrist);

    // Log angles for debugging

    console.log(`Left Angle: ${leftAngle}, Right Angle: ${rightAngle}`);

    // Define angle thresholds

    const curlUpThreshold = 60;  // Angle when arm is bent upward

    const curlDownThreshold = 160; // Angle when arm is extended downward

    // Check for curling motion for both arms

    if ((leftAngle < curlUpThreshold || rightAngle < curlUpThreshold) && !isCurling) {

        // Arms bent upward - Start of a curl

        isCurling = true;

    } else if ((leftAngle > curlDownThreshold && rightAngle > curlDownThreshold) && isCurling) {

        // Arms extended downward - End of a curl

        isCurling = false;

        curlCount++;

        countElem.innerText = `Repetitions: ${curlCount}`;

    }

}

        async function main() {

            const camera = new Camera(video, {

                onFrame: async () => {

                    frameTimestamp = Date.now(); // Capture timestamp when frame is sent

                    await poseDetector.send({image: video});

                },

                width: 1920,

                height: 1080

            });

            camera.start();

        }

        const poseDetector = new Pose({

            locateFile: (file) => `https://cdn.jsdelivr.net/npm/@mediapipe/pose/${file}`

        });

        poseDetector.setOptions({modelComplexity: 1});

        poseDetector.onResults((results) => {

            const processingTimestamp = Date.now(); // Capture timestamp when results are received

            const delay = processingTimestamp - frameTimestamp; // Calculate delay

            delayElem.innerText = `Processing Delay: ${delay} ms`;

            if (results.poseLandmarks) {

                const landmarks = getLandmarks(results.poseLandmarks);

                if (landmarks.length) {

                    console.log('Detected landmarks:', landmarks);

                    ws.send(JSON.stringify({ landmarks }));

                    updateCurlCount(landmarks);

                }

            } else {

                console.warn('No pose landmarks detected');

            }

        });

        startWebcam().then(main);

    </script>

</body>

</html>

from fastapi import FastAPI, WebSocket, WebSocketDisconnect

from fastapi.responses import HTMLResponse

import numpy as np

import tensorflow as tf

import uvicorn

app = FastAPI()

# Load your trained LSTM model

model = tf.keras.models.load\_model("C:/Users/BN GAMING/lstm\_Bicep\_curls\_model.keras")

# Selected landmarks for bicep curl detection (as per MediaPipe's PoseLandmark indices)

SELECTED\_LANDMARKS = [

    0,  # NOSE

    1,  # LEFT\_EYE

    2,  # RIGHT\_EYE

    3,  # LEFT\_EAR

    4,  # RIGHT\_EAR

    11, # LEFT\_SHOULDER

    12, # RIGHT\_SHOULDER

    13, # LEFT\_ELBOW

    14, # RIGHT\_ELBOW

    15, # LEFT\_WRIST

    16, # RIGHT\_WRIST

    17, # LEFT\_PINKY

    18, # RIGHT\_PINKY

    19, # LEFT\_INDEX

    20, # RIGHT\_INDEX

    21, # LEFT\_THUMB

    22, # RIGHT\_THUMB

    23, # LEFT\_HIP

    24  # RIGHT\_HIP

]

FEATURES\_PER\_LANDMARK = 4  # x, y, z, visibility

TOTAL\_SELECTED\_FEATURES = len(SELECTED\_LANDMARKS) \* FEATURES\_PER\_LANDMARK  # 76 features

def preprocess\_input(data):

    """

    Preprocess the input landmarks to extract only the selected features.

    - Extract features from SELECTED\_LANDMARKS.

    - Pad or truncate the data to match the required input size (76 features).

    """

    selected\_data = []

    for idx in SELECTED\_LANDMARKS:

        start\_idx = idx \* FEATURES\_PER\_LANDMARK

        end\_idx = start\_idx + FEATURES\_PER\_LANDMARK

        selected\_data.extend(data[start\_idx:end\_idx])

    if len(selected\_data) < TOTAL\_SELECTED\_FEATURES:

        # Pad with zeros if data is incomplete

        selected\_data.extend([0] \* (TOTAL\_SELECTED\_FEATURES - len(selected\_data)))

    elif len(selected\_data) > TOTAL\_SELECTED\_FEATURES:

        # Truncate if extra features are present

        selected\_data = selected\_data[:TOTAL\_SELECTED\_FEATURES]

    # Reshape the data to match the model's expected input

    np\_data = np.array(selected\_data).reshape(1, 1, TOTAL\_SELECTED\_FEATURES)

    return np\_data

@app.websocket("/ws")

async def websocket\_endpoint(websocket: WebSocket):

    """

    WebSocket endpoint to receive pose landmarks, preprocess them, and make predictions.

    """

    await websocket.accept()

    print("WebSocket connection opened")

    while True:

        try:

            # Receive landmarks from the client

            data = await websocket.receive\_json()

            print(f"Received data: {data}")

            landmarks = data.get("landmarks", [])

            # Preprocess input landmarks

            processed\_input = preprocess\_input(landmarks)

            # Make a prediction using the LSTM model

            prediction = model.predict(processed\_input)

            predicted\_class = np.argmax(prediction, axis=-1)

            # Map predictions to human-readable labels

            label\_map = {0: 'neutral', 1: 'bicep\_curl', 2: 'wrong'}

            predicted\_label = label\_map.get(predicted\_class[0], 'unknown')

            # Send the prediction result back to the client

            await websocket.send\_json({"predicted\_posture": predicted\_label})

            print(f"Sent prediction: {predicted\_label}")

        except WebSocketDisconnect:

            print("WebSocket connection closed")

            break

        except Exception as e:

            await websocket.send\_json({"error": str(e)})

            print(f"Error: {e}")

@app.get("/", response\_class=HTMLResponse)

async def serve\_page():

    """

    Serve the HTML page for the client.

    """

    html\_path = "index\_test\_01.html"  # Update this to the path of your HTML file

    with open(html\_path, 'r') as file:

        html\_content = file.read()

    return HTMLResponse(content=html\_content)

if \_\_name\_\_ == "\_\_main\_\_":

    uvicorn.run(app, host="127.0.0.1", port=8000)