**DATASET CODE:**

import cv2

import mediapipe as mp

import pandas as pd

import os

# Open webcam

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 = "neutral\_wall\_pushups\_final"

no\_of\_frames = 1000

def make\_landmark\_timestep(results):

"""Extract all 33 pose landmarks."""

c\_lm = []

if results.pose\_landmarks:

for lm in results.pose\_landmarks.landmark:

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 CSV file already 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\_wall\_pushups\_final.csv")

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

wrong\_df = pd.read\_csv("wrong\_wall\_pushups\_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=32, return\_sequences=True, input\_shape=(X.shape[1], X.shape[2])))

model.add(Dropout(0.3))

model.add(LSTM(units=32))

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=5, # Train for up to 15 epochs

batch\_size=2,

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\_fullbody\_wall\_pushups.keras", save\_format='keras')

print("Model saved to 'lstm\_fullbody\_wall\_pushups.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 RUNNING:**

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, QHBoxLayout

from PyQt5.QtGui import QImage, QPixmap, QFont

from PyQt5.QtCore import QTimer, Qt

# Load the trained model

model = tf.keras.models.load\_model("lstm\_fullbody\_wall\_pushups.keras")

# Define class labels

class\_labels = {0: "neutral", 1: "wall\_pushup", 2: "wrong"}

# Initialize MediaPipe Pose

mpPose = mp.solutions.pose

pose = mpPose.Pose()

mpDraw = mp.solutions.drawing\_utils

# Smoothing parameters for angle detection

angle\_smoothing = deque(maxlen=5)

def make\_landmark\_timestep(results):

"""Extract all 132 pose landmarks as a flat list."""

if results.pose\_landmarks:

landmarks = results.pose\_landmarks.landmark

return [val for lm in landmarks for val in (lm.x, lm.y, lm.z, lm.visibility)]

return []

def calculate\_angle(point1, point2, point3):

"""

Calculate the angle between three points using vector math.

point1, point2, point3: Tuples containing (x, y) coordinates.

"""

a = np.array(point1)

b = np.array(point2)

c = np.array(point3)

# Calculate vectors

ba = a - b

bc = c - b

# Calculate cosine angle

cosine\_angle = np.dot(ba, bc) / (np.linalg.norm(ba) \* np.linalg.norm(bc))

angle = np.arccos(np.clip(cosine\_angle, -1.0, 1.0)) # Clip to avoid floating-point errors

# Convert to degrees

return np.degrees(angle)

class WallPushupApp(QWidget):

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

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

self.setWindowTitle("Wall Pushup AI Trainer")

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

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

# GUI Components

self.layout = QVBoxLayout()

self.video\_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;")

self.video\_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.video\_layout.addWidget(self.feedback\_label)

self.layout.addLayout(self.video\_layout)

# Close button layout

self.button\_layout = QHBoxLayout()

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.button\_layout.addWidget(self.close\_button)

self.layout.addLayout(self.button\_layout)

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"

self.last\_y\_elbow = None # To track the Y position of the elbow

# Thresholds for detecting the pushup phases (These are adjustable)

self.DOWN\_THRESHOLD = 65 # Threshold for when the arms are "down" (bent elbow)

self.UP\_THRESHOLD = 140 # Threshold for when the arms are "up" (extended elbow)

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)

if lm: # Ensure valid landmarks are captured

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

# Focus on left arm for wall pushup

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 elbow angle

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

# Smooth the angle

angle\_smoothing.append(elbow\_angle)

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

# Repetition logic based on smoothed angle

if self.label == "wall\_pushup":

if smoothed\_angle < self.DOWN\_THRESHOLD and self.arms\_extended:

self.arms\_extended = False # Arm is going down

elif smoothed\_angle > self.UP\_THRESHOLD and not self.arms\_extended:

self.rep\_counter += 1 # Count a rep

self.arms\_extended = True # Reset to extended position

# Draw landmarks on frame

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

# Draw feedback on screen

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 = WallPushupApp()

window.show()

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