**DATASET CODE:**

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 = "correct\_Overhead\_press\_final\_1"

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\_Overhead\_press\_final\_1.csv")

correct\_df = pd.read\_csv("correct\_Overhead\_press\_final\_1.csv")

wrong\_df = pd.read\_csv("wrong\_Overhead\_press\_final\_1.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, validation, and test sets

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

X\_val, X\_test, y\_val, y\_test = train\_test\_split(X\_temp, y\_temp, test\_size=0.5, 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=10, # Train for up to 15 epochs

batch\_size=8,

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

callbacks=[early\_stopping],

verbose=1

)

# Evaluate the model on validation data

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}")

# Evaluate the model on test data

test\_loss, test\_accuracy = model.evaluate(X\_test, y\_test, verbose=0)

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

print(f"Test Loss: {test\_loss:.4f}")

# Confusion Matrix for test data

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

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

# Save the model

model.save("lstm\_Overhead\_press\_model\_test.keras", save\_format='keras')

print("Model saved to 'lstm\_Overhead\_press\_model\_test.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()

# 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 KFold

from sklearn.metrics import confusion\_matrix

import matplotlib.pyplot as plt

import seaborn as sns

# Load CSV files

neutral\_df = pd.read\_csv("neutral\_Overhead\_press\_final\_1.csv")

correct\_df = pd.read\_csv("correct\_Overhead\_press\_final\_1.csv")

wrong\_df = pd.read\_csv("wrong\_Overhead\_press\_final\_1.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,)

# K-Fold Cross-Validation setup

kf = KFold(n\_splits=5, shuffle=True, random\_state=42)

fold\_no = 1

# Store results for each fold

fold\_accuracies = []

fold\_losses = []

confusion\_matrices = []

history\_all\_epochs = [] # Store history for epoch-wise accuracy/loss plotting

for train\_index, val\_index in kf.split(X):

print(f"\n--- Fold {fold\_no} ---")

# Split data into training and validation sets

X\_train, X\_val = X[train\_index], X[val\_index]

y\_train, y\_val = y[train\_index], y[val\_index]

# 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=6,

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

callbacks=[early\_stopping],

verbose=1

)

# Store history for visualization

history\_all\_epochs.append(history)

# Evaluate the model

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

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

print(f"Validation Loss for Fold {fold\_no}: {val\_loss:.4f}")

# Save the results for this fold

fold\_accuracies.append(val\_accuracy)

fold\_losses.append(val\_loss)

# Confusion Matrix

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

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

confusion\_matrices.append(cm)

# Save the model for this fold

model\_save\_path = f"lstm\_Overhead\_press\_Fold\_{fold\_no}.keras"

model.save(model\_save\_path, save\_format='keras')

print(f"Model for Fold {fold\_no} saved to {model\_save\_path}")

# Increment fold number

fold\_no += 1

# Summary of results

mean\_accuracy = np.mean(fold\_accuracies)

mean\_loss = np.mean(fold\_losses)

print("\n--- Cross-Validation Results ---")

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

print(f"Mean Loss: {mean\_loss:.4f}")

# Plot Confusion Matrix for the last fold

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

sns.heatmap(confusion\_matrices[-1], annot=True, fmt='d', cmap='Blues', xticklabels=['Neutral', 'Correct', 'Wrong'], yticklabels=['Neutral', 'Correct', 'Wrong'])

plt.title(f'Confusion Matrix for Fold {fold\_no - 1}')

plt.xlabel('Predicted Labels')

plt.ylabel('True Labels')

plt.show()

# Visualize Validation Accuracy for all folds

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

plt.plot(range(1, len(fold\_accuracies) + 1), [acc \* 100 for acc in fold\_accuracies], marker='o', label="Fold Accuracy")

plt.title('Validation Accuracy for Each Fold')

plt.xlabel('Fold Number')

plt.ylabel('Accuracy (%)')

plt.grid()

plt.legend()

plt.show()

# Plot Accuracy and Loss for 20 Epochs Across Folds

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

for fold, hist in enumerate(history\_all\_epochs):

plt.plot(hist.history['val\_accuracy'], label=f'Fold {fold + 1} Accuracy')

plt.title('Validation Accuracy for 20 Epochs Across Folds')

plt.xlabel('Epoch')

plt.ylabel('Accuracy')

plt.legend()

plt.grid()

plt.show()

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

for fold, hist in enumerate(history\_all\_epochs):

plt.plot(hist.history['val\_loss'], label=f'Fold {fold + 1} Loss')

plt.title('Validation Loss for 20 Epochs Across Folds')

plt.xlabel('Epoch')

plt.ylabel('Loss')

plt.legend()

plt.grid()

plt.show()

**REAL TIME RUNNING 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\_Overhead\_press\_model\_test.keras")

# Define class labels

class\_labels = {0: "neutral", 1: "overhead\_press", 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 OverheadPressApp(QWidget):

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

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

self.setWindowTitle("Overhead Press 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 = False

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 for the overhead press (shoulder-elbow-wrist angle)

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

angle\_smoothing.append(shoulder\_elbow\_wrist\_angle)

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

# Repetition logic for overhead press

if self.label == "overhead\_press":

if 160 < smoothed\_angle <= 180: # Top position (arms fully extended)

self.arms\_extended = True

elif self.arms\_extended and 60 <= smoothed\_angle <= 100: # Bottom position (elbows bent)

self.rep\_counter += 1

self.arms\_extended = False

# 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 = OverheadPressApp()

window.show()

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