informe tp cv

Función de la Aplicación

La aplicación detecta la postura de yoga que realiza un usuario a partir de una imagen proporcionada, clasificándola en una de las cinco categorías (`downdog`, `goddess`, `plank`, `tree`, `warrior2`). Su propósito es ofrecer una herramienta básica que sirva como punto de partida para desarrollos más avanzados en el futuro.

Visión de la Aplicación

La visión a largo plazo es desarrollar una herramienta que no solo detecte posturas de yoga, sino que también ofrezca retroalimentación detallada, permita registrar un historial de poses detectadas y, eventualmente, habilite el reconocimiento en tiempo real mediante video. Este enfoque garantiza una evolución progresiva hacia una aplicación más sofisticada y útil para los usuarios.

Alcance

1. En esta etapa inicial, el proyecto se limitará a la detección de cinco posturas de yoga específicas a partir de imágenes.
2. En el futuro, se podrá expandir la funcionalidad para:
    - Registrar un historial de poses detectadas y compararlas con poses óptimas.
    - Incorporar un sistema de reconocimiento en tiempo real mediante video.

Historia de Usuario

Como practicante de yoga, **quiero** que la aplicación me diga qué postura estoy haciendo al subir una foto, **para** aprender que posicion estoy realizando.

Casos de Uso

1. **Caso básico:** Un usuario sube una imagen y recibe la postura detectada.
2. **Caso futuro:** Registro de un historial de poses detectadas y comparación con poses óptimas.

3. **Extensión futura:** Reconocimiento de poses en tiempo real mediante video.

Entorno de Trabajo

El desarrollo se realiza en Google Colab, aprovechando su capacidad de cómputo en la nube y su integración con Python para pruebas y entrenamientos. Los datos y modelos se almacenan y gestionan en GitHub, lo que facilita la colaboración y el control de versiones.

Consideraciones del Entorno:

- **Capacidad de cómputo**: Aunque Colab es útil para entrenar modelos, más adelante se evaluará cómo optimizar el proyecto para que funcione en dispositivos con pocos recursos, como celulares.
- **Tiempo real**: La función de procesamiento en tiempo real no se incluirá al inicio, pero se trabajará en ello en futuras versiones.
- **Organización**: Se seguirán reglas claras para nombrar archivos y repositorios, asegurando que todo esté bien ordenado y sea fácil de mantener.

Hipótesis sobre MediaPipe Pose

Hipótesis inicial:

MediaPipe Pose puede detectar puntos clave del cuerpo humano (landmarks) con suficiente precisión para entrenar un modelo que clasifique cinco posturas de yoga con una precisión superior al 85%.

Objetivo

- Desarrollar un pipeline en Google Colab que:
    1. Extraiga landmarks de imágenes usando MediaPipe Pose.
    2. Entrene un modelo supervisado capaz de clasificar las cinco posturas de yoga con una precisión aceptable.

Planificación

1. **Fase 1: Preparación del entorno**

- Configurar y utilizar MediaPipe Pose en Google Colab.

2. **Fase 2: Manejo de datos**
   - Crear un conjunto de datos etiquetado donde los puntos clave (landmarks) sean las entradas y las posturas sean las salidas.
   - Subir y gestionar el dataset junto con los checkpoints en un repositorio de GitHub.
3. **Fase 3: Entrenamiento del modelo**
   - Entrenar una red neuronal densa (MLP) usando los puntos clave como características.
4. **Fase 4: Pruebas y ajustes**
   - Evaluar el modelo con métricas como precisión (accuracy) y pérdida (loss).
   - Ajustar los hiperparámetros y realizar pruebas adicionales si es necesario.
5. **Fase 5: Preparación para el despliegue**
   - Adaptar el modelo para funcionar en dispositivos móviles en futuras etapas, priorizando su rapidez y eficiencia.

Librerías Instaladas y Configuración Inicial

Instalaciones de librerías necesarias

```
!pip install mediapipe
!pip install fiftyone
!pip install tensorflow
```

1. `mediapipe`: Se instala para usar MediaPipe Pose, una librería de Google que permite la detección de landmarks corporales a partir de imágenes o video. Es fundamental para extraer los puntos clave que representan las posturas de yoga.
2. `fiftyone`: Una herramienta para gestionar y analizar datasets visuales. Puede ser útil en tareas como la exploración del conjunto de datos o la

validación de predicciones del modelo.

3. `tensorflow`: Framework de aprendizaje profundo que se puede usar en futuras etapas para la creación y entrenamiento de modelos.

Importación de librerías

```
import cv2
import os
import numpy as np
import pandas as pd
import mediapipe as mp
import matplotlib.pyplot as plt
from tqdm import tqdm
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, TensorDataset
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
import requests
```

- `cv2` **(OpenCV):** Para la manipulación de imágenes y videos, como lectura, visualización y preprocesamiento.
- `os`: Proporciona acceso a funcionalidades del sistema operativo, como manipular rutas de archivos o directorios.
- `numpy`: Para realizar operaciones matemáticas y manejar arreglos, que serán útiles para manipular landmarks y otros datos.
- `pandas`: Ayuda en la gestión de datos tabulares, como la creación y análisis de tablas con etiquetas de posturas y características.
- `mediapipe`: Biblioteca principal que permite detectar landmarks corporales mediante el modelo Pose.
- `matplotlib.pyplot`: Herramienta de visualización para graficar landmarks, resultados y datos.
- `tqdm`: Para generar barras de progreso y hacer seguimiento de procesos iterativos.

- `torch`: Biblioteca para desarrollar modelos de aprendizaje profundo. Aquí se usa para definir redes neuronales, optimizadores y datasets.
- `sklearn`: Incluye herramientas para dividir datos en entrenamiento y prueba (`train_test_split`) y para codificar etiquetas categóricas (`LabelEncoder`).
- `requests`: Útil para realizar solicitudes HTTP, por ejemplo, descargar datasets o recursos adicionales.

Configuración de MediaPipe Pose

```
mp_pose = mp.solutions.pose
mp_drawing = mp.solutions.drawing_utils
pose = mp_pose.Pose()
```

1. `mp.solutions.pose`: Se inicializa el modelo **Pose** de MediaPipe, que permite detectar landmarks corporales en imágenes o videos.
2. `mp.solutions.drawing_utils`: Utilidad que permite superponer los landmarks y las conexiones entre ellos directamente en las imágenes, útil para visualización.
3. `mp_pose.Pose()`: Se crea una instancia del modelo Pose con parámetros predeterminados.

Parámetros de configuración del modelo

```
with mp_pose.Pose(min_detection_confidence=0.5, min_trackin
g_confidence=0.5) as pose:
    print("Modelo Pose inicializado, listo para procesar im
ágenes.")
```

- `min_detection_confidence=0.5`: Establece un umbral del 50% para considerar válida la detección inicial de landmarks.
- `min_tracking_confidence=0.5`: Define un umbral del 50% para mantener el seguimiento de los landmarks en secuencias de video.
- `with`: Garantiza que los recursos asociados con el modelo Pose se liberen automáticamente cuando el bloque termina de ejecutarse.

Propósito de este fragmento

Este código prepara el entorno para trabajar con **MediaPipe Pose**, asegurando que todas las librerías necesarias estén instaladas y configuradas correctamente. También define las herramientas básicas para procesar imágenes, detectar landmarks y manejar datos.

Carga de archivos desde GitHub

```
repo_owner = 'brimoresco'
repo_name = 'cv-tp-yoga-moresco'
train_dir = 'TEST'
test_dir = 'TRAIN'
branch = 'main'

# Función para obtener el listado de archivos desde el repo
sitorio de GitHub
def get_github_file_list(owner, repo, path, branch='main'):
    url = f'https://api.github.com/repos/{owner}/{repo}/con
tents/{path}?ref={branch}'
    response = requests.get(url)
    response.raise_for_status()  # Asegura que la solicitud
sea exitosa
    files = response.json()
    return files
```

1. **Descripción**: Este bloque obtiene la lista de archivos alojados en un repositorio de GitHub.
2. `get_github_file_list`:
   - Genera la URL para obtener el contenido de una carpeta en GitHub mediante la API de GitHub.
   - Utiliza `requests.get` para realizar la solicitud y asegura que sea exitosa con `response.raise_for_status()`.
   - Devuelve una lista de archivos o carpetas en el directorio especificado.

Descarga de imágenes desde URLs

```
def load_image_from_url(url):
    resp = requests.get(url)
    img = np.array(bytearray(resp.content), dtype=np.uint8)
    img = cv2.imdecode(img, -1)
    if img is None:
        print(f"Error loading image from {url}")
    return img
```

1. **Descripción**: Carga imágenes en tiempo real desde URLs proporcionadas.
2. `requests.get(url)`: Realiza una solicitud HTTP para obtener el contenido binario de la imagen.
3. `cv2.imdecode`: Decodifica los bytes descargados en una imagen legible por OpenCV.
4. **Propósito**: Garantizar que las imágenes necesarias para el entrenamiento y prueba puedan ser utilizadas sin descargarlas manualmente.

Interpolación entre Keypoints

```
def interpolate_between_keypoints(keypoints, edges, num_poi
nts=5):
    interpolated_keypoints = []
    for i, keypoint in enumerate(keypoints):
        interpolated_keypoints.append(keypoint)

    for start_idx, end_idx in edges:
        start_keypoint = np.array(keypoints[start_idx])
        end_keypoint = np.array(keypoints[end_idx])
        for t in np.linspace(0, 1, num_points + 2)[1:-1]:
            interpolated_keypoints.append((start_keypoint *
(1 - t) + end_keypoint * t).tolist())

    return interpolated_keypoints
```

- **Descripción inicial**: Esta función genera puntos adicionales entre los keypoints que están conectados por aristas (edges).
- **Evolución de la decisión**:

    - **Primera instancia**: Inicialmente, no se aplicaba interpolación; solo se usaban los puntos clave detectados directamente por MediaPipe Pose.
    - **Motivación para el cambio**: Tras analizar los resultados iniciales, se decidió aplicar interpolación para **enriquecer los datos espaciales**, aumentando la densidad de puntos clave y capturando mejor las relaciones geométricas entre ellos.
- **Cómo funciona**:
    - **Keypoints originales**: Se añaden primero los puntos detectados por MediaPipe Pose.
    - **Interpolación**: Para cada arista (conexión entre dos keypoints), se generan puntos intermedios usando interpolación lineal.
    - `np.linspace`: Divide el intervalo entre dos keypoints en segmentos iguales, generando puntos adicionales en el espacio 3D (x, y, z).
- **Propósito final**: Esta técnica mejora la capacidad del modelo para capturar las relaciones espaciales entre los puntos clave, lo que puede traducirse en un mejor rendimiento durante el entrenamiento y la clasificación de las posturas.

Carga y procesamiento de datos

```
def data_loader_to_dataframe(path, repo_owner, repo_name, n
um_points=5):
    data = []
    pose = mp.solutions.pose.Pose()
    s, f = 0, 0
    labels = [item['name'] for item in get_github_file_list
(repo_owner, repo_name, path)]

    for label in tqdm(labels):
        label_path = os.path.join(path, label)
        files = get_github_file_list(repo_owner, repo_name,
label_path)
        for file in files:
            image_url = file['download_url']
            if image_url:
                img = load_image_from_url(image_url)
```

```
                if img is not None and img.size != 0:
                    img_rgb = cv2.cvtColor(img, cv2.COLOR_B
GR2RGB)
                    results = pose.process(img_rgb)
                    if results.pose_landmarks:
                        keypoints = [[lmk.x, lmk.y, lmk.z]
for lmk in results.pose_landmarks.landmark]
                        edges = list(mp.solutions.pose.POSE
_CONNECTIONS)
                        interpolated_keypoints = interpolat
e_between_keypoints(keypoints, edges, num_points=num_point
s)
                        data.append({
                            "url_img": image_url,
                            "keypoints": interpolated_keypo
ints,
                            "pose_name": label
                        })
                        s += 1
                    else:
                        f += 1
    return pd.DataFrame(data)
```

1. **Flujo General**:
    - **Etiquetas:** Obtiene las carpetas (nombres de las poses) del repositorio.
    - **Imágenes:** Descarga cada imagen desde su URL y la procesa con **MediaPipe Pose**.
    - **Landmarks:** Extrae los puntos clave de la pose detectada en la imagen.
    - **Interpolación:** Añade puntos adicionales entre keypoints para enriquecer la información.
    - **Estructuración:** Organiza la información (imagen, keypoints, etiqueta) en un DataFrame.
2. **Columnas del DataFrame**:
    - `url_img`: URL de la imagen procesada.
    - `keypoints`: Lista de keypoints interpolados (formato `[x, y, z]`).

- `pose_name`: Nombre de la postura detectada.

3. **Indicadores de rendimiento**:
   - `s`: Número de imágenes procesadas con éxito.
   - `f`: Número de imágenes donde no se detectaron landmarks.

Generación de DataFrames de Entrenamiento y Prueba

```
train_df = data_loader_to_dataframe(train_dir, repo_owner,
repo_name, num_points=5)
test_df = data_loader_to_dataframe(test_dir, repo_owner, re
po_name, num_points=5)
```

1. **Entrenamiento y prueba**:
   - `train_dir`: Directorio de entrenamiento en el repositorio.
   - `test_dir`: Directorio de prueba en el repositorio.
2. **Propósito**: Estructurar los datos necesarios para entrenar y evaluar el modelo en dos conjuntos separados.

Análisis Exploratorio de los Datos

1. Ver las primeras filas del DataFrame

```
train_df.head()
```

- **Propósito**:
  Visualizar las primeras filas del DataFrame `train_df` para entender la estructura de los datos. Esto ayuda a confirmar que las columnas (`url_img`, `keypoints`, `pose_name`) están correctamente formadas y contienen los datos esperados.

---

2. Estadísticas descriptivas de los keypoints

```
keypoints_len = train_df['keypoints'].apply(len)
print(f"Longitud de keypoints promedio: {keypoints_len.mean
()}")
```

```
print(f"Longitud de keypoints por imagen: {keypoints_len.un
ique()}")
```

- **Propósito**:
  Analizar la longitud de la lista de keypoints para cada imagen, lo cual refleja el número de puntos clave detectados (incluyendo los interpolados).
- **Datos útiles**:
  - La longitud promedio ayuda a verificar si el número de puntos generados es consistente entre las imágenes.
  - La diversidad en las longitudes puede indicar diferencias en la calidad o la interpolación de los keypoints.
- Resultados:

```
Longitud de keypoints promedio: 208.0
Longitud de keypoints por imagen: [208]
```

- **Consistencia en el número de keypoints:**
  - La longitud promedio de keypoints es 208, y todas las imágenes tienen exactamente la misma longitud (208). Esto indica que el número de puntos clave detectados o interpolados es consistente para todas las imágenes.
- **Calidad uniforme del procesamiento:**
  - La uniformidad en las longitudes sugiere que el método utilizado para detectar o interpolar los keypoints (por ejemplo, MediaPipe Pose) está funcionando de manera consistente en todas las imágenes, sin variaciones en la calidad o cantidad de puntos generados.

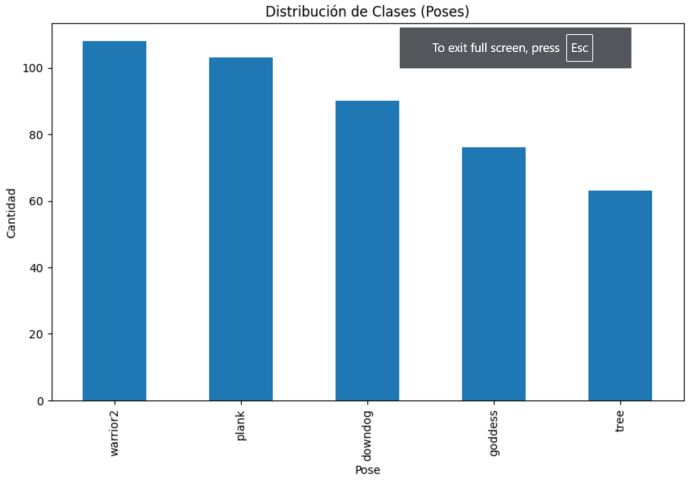
---

3. Visualización de la distribución de clases

```
plt.figure(figsize=(10, 6))
class_counts.plot(kind='bar')
plt.title("Distribución de Clases (Poses)")
plt.xlabel("Pose")
```

```
plt.ylabel("Cantidad")
plt.show()
```

- **Propósito**:
  Mostrar gráficamente el número de imágenes por clase en el conjunto de entrenamiento. Esto facilita:
  - Detectar visualmente el desbalance de clases.
  - Comunicar los resultados del análisis de forma más clara y atractiva.
- Resultado:
  - observamos un fuerte desbalance entre las clases.



4. Ver la distribución de etiquetas en los conjuntos de entrenamiento y prueba

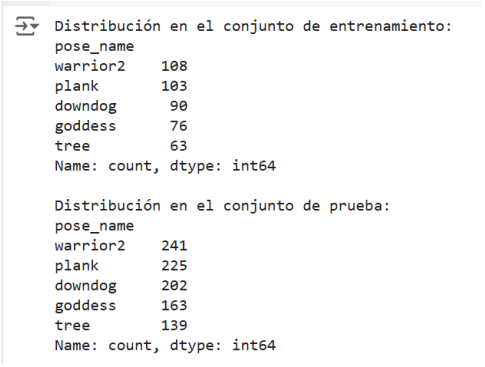
```
train_counts = pd.Series(train_df["pose_name"]).value_count
s()
test_counts = pd.Series(test_df["pose_name"]).value_counts
()

print("Distribución en el conjunto de entrenamiento:")
print(train_counts)
```

```
print("\nDistribución en el conjunto de prueba:")
print(test_counts)
```

- **Propósito**:
  Comparar la distribución de etiquetas entre los conjuntos de entrenamiento (
  `train_df`) y prueba (`test_df`). Esto asegura que ambos conjuntos estén bien representados y permitan evaluar el modelo de manera confiable.
- **Resultado**:

  Hay un claro desequilibrio en las clases, especialmente en el conjunto de entrenamiento, donde algunas etiquetas (como **tree**) tienen significativamente menos ejemplos en comparación con otras (como **warrior2**).



```
Distribución en el conjunto de entrenamiento:
pose_name
warrior2    108
plank       103
downdog      90
goddess      76
tree         63
Name: count, dtype: int64

Distribución en el conjunto de prueba:
pose_name
warrior2    241
plank       225
downdog     202
goddess     163
tree        139
Name: count, dtype: int64
```

Aumentación de Datos con Flip Horizontal

Este bloque aborda una estrategia fundamental para combatir el desbalance de clases mediante la **aumentación de datos**: duplicar las muestras de las clases minoritarias invirtiendo las imágenes y sus keypoints horizontalmente. También incluye visualización para verificar los resultados.

Función `flip_keypoints_horizontally`

```
def flip_keypoints_horizontally(keypoints, img_width):
    flipped_keypoints = []
    for x, y, z in keypoints:
        flipped_keypoints.append([1 - x, y, z])  # Invertir
```

```
la coordenada X
    return flipped_keypoints
```

1. **Propósito**:
   - Reflejar horizontalmente los keypoints manteniendo la coherencia con la imagen invertida.
   - La coordenada `x` se invierte con respecto al ancho de la imagen.
2. **Cómo funciona**:
   - Para cada keypoint (`x, y, z`), se calcula la nueva posición de `x` como `1 - x`.
   - Las coordenadas `y` y `z` permanecen sin cambios.

Función `augment_data_with_flip`

```
def augment_data_with_flip(train_df):
    # Código de la función
```

1. **Propósito**:
   - Balancear las clases duplicando las muestras de las clases minoritarias hasta igualar la cantidad de la clase mayoritaria.
   - Las muestras duplicadas se generan invirtiendo horizontalmente las imágenes y los keypoints asociados.
2. **Pasos principales**:
   - **Detectar clases desbalanceadas**: Calcula la cantidad de ejemplos por clase usando `value_counts()` y determina la clase mayoritaria.
   - **Aumentar clases minoritarias**:
     - Itera sobre cada clase con menos ejemplos que la mayoritaria.
     - Duplica muestras de esa clase aplicando:
       - Reflejo horizontal en la imagen.
       - Transformación de keypoints con `flip_keypoints_horizontally`.
   - **Combinar datos originales y aumentados**: Al final, devuelve un nuevo DataFrame que incluye ambos.

Visualización de Keypoints

```
def visualize_keypoints_on_image(img, keypoints):
    for keypoint in keypoints:
        x, y = int(keypoint[0] * img.shape[1]), int(keypoin
t[1] * img.shape[0])
        cv2.circle(img, (x, y), 5, (0, 255, 0), -1)
    return img
```

1. **Propósito**:

   Dibujar los keypoints sobre una imagen para verificar visualmente su correspondencia y ubicación correcta.

2. **Cómo funciona**:

   - Transforma las coordenadas normalizadas (`x`, `y`) de los keypoints a píxeles reales usando las dimensiones de la imagen.
   - Dibuja círculos en cada keypoint sobre la imagen con OpenCV.

Resultados Obtenidos

1. Balance de clases después de la aumentación

```
class_counts_after_flip = augmented_train_df['pose_name'].v
alue_counts()
print(f"Distribución de clases después de la aumentación co
n flip horizontal: \n{class_counts_after_flip}")
```

- Todas las clases tienen la misma cantidad de ejemplos. Esto asegura que el modelo reciba una representación equitativa durante el entrenamiento.

```
Distribución de clases después de la aumentación
pose_name
downdog     108
goddess     108
plank       108
tree        108
warrior2    108
Name: count, dtype: int64
```

2. Visualización de imágenes originales y aumentadas

```
# Visualizar imágenes originales y aumentadas (flip horizon
tal)
for i in range(3):
    # Visualizar imagen original
    img_url = train_df.iloc[i]['url_img']
    keypoints = train_df.iloc[i]['keypoints']
    img = load_image_from_url(img_url)

    if img is not None:
        img_with_keypoints = visualize_keypoints_on_image(i
mg.copy(), keypoints)
        plt.imshow(cv2.cvtColor(img_with_keypoints, cv2.COL
OR_BGR2RGB))
        plt.title(f"Original Pose: {train_df.iloc[i]['pose_
name']}")
        plt.axis('off')
        plt.show()

    # Visualizar imagen aumentada (flip horizontal)
    augmented_img_url = augmented_train_df.iloc[i * 2 + 1]
['url_img']  # Toma la imagen aumentada
    augmented_keypoints = augmented_train_df.iloc[i * 2 +
1]['keypoints']
    augmented_img = load_image_from_url(augmented_img_url)

    if augmented_img is not None:
        augmented_img_with_keypoints = visualize_keypoints_
on_image(augmented_img.copy(), augmented_keypoints)
        plt.imshow(cv2.cvtColor(augmented_img_with_keypoint
s, cv2.COLOR_BGR2RGB))
        plt.title(f"Aumentada Pose: {augmented_train_df.ilo
c[i * 2 + 1]['pose_name']}")
        plt.axis('off')
        plt.show()
```

- **Propósito**:
  Verificar que las imágenes y keypoints invertidos reflejan correctamente la pose original, pero en su versión horizontal.

Sobreescribir el DataFrame Original

```
train_df = augmented_train_df.copy()
```

- **Propósito**:
  Actualizar el DataFrame de entrenamiento con los datos aumentados para su uso en el modelo.

Separación del conjunto de entrenamiento y prueba

```
# Para el conjunto de entrenamiento
x_train_raw = augmented_train_df['keypoints']  # Característi
y_train_raw = train_df['pose_name']  # Etiquetas (posturas) d

# Para el conjunto de prueba
x_test_raw = test_df['keypoints']  # Características (keypoin
y_test_raw = test_df['pose_name']  # Etiquetas (posturas) del

# Verifica las formas de las variables
print("train dataset: ", x_train_raw.shape, y_train_raw.shape
print("test dataset: ", x_test_raw.shape, y_test_raw.shape)
```

Para el conjunto de entrenamiento y prueba:

- `x_train_raw` y `x_test_raw` almacenan las características de las muestras para los conjuntos de entrenamiento y prueba, respectivamente. En este caso, son las 'keypoints' de las imágenes que representan las características principales de cada muestra.
- `y_train_raw` y `y_test_raw` contienen las etiquetas (o clases) correspondientes a las 'posturas' (poses) de las imágenes en cada conjunto. Estas etiquetas se extraen de las columnas `pose_name` en `train_df` y `test_df`.
- imprimimos las dimensiones de `x_train_raw` y `y_train_raw`, así como las de `x_test_raw` y `y_test_raw` para verificar que el número de características y etiquetas coincide correctamente para cada conjunto de datos.

```
# Codificar las clases
from sklearn.preprocessing import LabelEncoder
le = LabelEncoder()
labels = ['downdog', 'goddess', 'plank', 'tree', 'warrior2']
le.fit(labels)
y_train_raw = le.transform(y_train_raw)
y_test_raw = le.transform(y_test_raw)
```

La codificación de las clases utiliza `LabelEncoder` de `scikit-learn` para convertir las etiquetas de las posturas de texto a números. Se ajusta el encoder (`le.fit(labels)`) a las clases para preparar la transformación. Despues, se aplican las transformaciones a las etiquetas de entrenamiento y prueba para convertirlas en valores numéricos adecuados para el modelo.

Convertir los datos de pose y etiquetas en tensores para PyTorch.

- **Propósito**: Transformar las características y las etiquetas de las muestras en un formato que PyTorch pueda manejar. Esto es esencial para realizar cálculos y entrenar el modelo con las tensores correspondientes.
- **Código**:

```
x_train = torch.FloatTensor(x_train_flat)
y_train = torch.LongTensor(y_train_flat)
x_test = torch.FloatTensor(x_test_flat)
y_test = torch.LongTensor(y_test_flat)
```

1. **Aplanar los datos de entrada**:
   - **Propósito**: Las características de entrada (`keypoints`) se representan como listas de listas de valores (vectores). Aplanarlas implica convertir estos vectores en un único arreglo 1D, lo que facilita el manejo de los datos en PyTorch.
   - **Código**:

```
x_train_flat = np.array([np.array(kp).flatten() for k
p in x_train_raw])
x_test_flat = np.array([np.array(kp).flatten() for kp
in x_test_raw])
```

- **Explicación**:
  - `np.array([np.array(kp).flatten() for kp in x_train_raw])`: Se itera sobre cada elemento en `x_train_raw` (que son los 'keypoints' para el conjunto de entrenamiento). Para cada 'keypoint', se convierte en un arreglo NumPy, se aplane (transformando el arreglo en una sola dimensión) y se almacena en una lista. La función `np.array(...)` convierte esta lista en un arreglo NumPy.
  - El mismo proceso se aplica a `x_test_raw`, produciendo `x_test_flat`.

2. **Asegúrate de que las etiquetas sean arreglos NumPy y aplánalas (si corresponde)**:
   - **Propósito**: Similar a los pasos de características, las etiquetas (`pose_name`) también se representan como listas. Aplanarlas convierte cada lista en un único número, lo que facilita el uso en modelos de aprendizaje automático.
   - **Código**:

```
y_train_flat = np.array([np.array(label).flatten() fo
r label in y_train_raw])
y_test_flat = np.array([np.array(label).flatten() for
label in y_test_raw])
```

   - **Explicación**:
     - `np.array([np.array(label).flatten() for label in y_train_raw])`: Itera sobre cada elemento en `y_train_raw` (que son las etiquetas para el conjunto de entrenamiento). Para cada etiqueta, se convierte en un arreglo NumPy y se aplana. La función `flatten()` transforma un arreglo multidimensional en un 1D.
     - El mismo proceso se aplica a `y_test_raw`, generando `y_test_flat`.

3. **Verificar las formas resultantes**:

- **Propósito**: Verificar que la transformación de las características y etiquetas ha producido los tamaños correctos de arreglo. Esto es crucial para asegurarse de que el modelo recibirá la entrada y las etiquetas en el formato adecuado.
- **Código**:

```
print("Forma de x_train_flat:", x_train_flat.shape)
print("Forma de y_train_flat:", y_train_flat.shape)
print("Forma de x_test_flat:", x_test_flat.shape)
print("Forma de y_test_flat:", y_test_flat.shape)
```

- **Explicación**: Estas líneas imprimen las dimensiones de los arreglos aplanados para los conjuntos de entrenamiento y prueba, lo que permite confirmar que el aplanamiento se realizó correctamente y que los datos tienen la longitud esperada.

4. **Convertir a tensores**:
   - **Propósito**: Transformar los datos de entrada y las etiquetas de los arreglos NumPy a tensores de PyTorch. Esto facilita su uso en modelos de PyTorch, que requieren tensores para las operaciones de cálculo y entrenamiento.
   - **Código**:

```
x_train = torch.FloatTensor(x_train_flat)
y_train = torch.LongTensor(y_train_flat)
x_test = torch.FloatTensor(x_test_flat)
y_test = torch.LongTensor(y_test_flat)
```

   - **Explicación**:
     - `torch.FloatTensor(x_train_flat)`: Convierte `x_train_flat` (las características aplanadas) de un arreglo NumPy a un tensor de PyTorch de tipo flotante.
     - `torch.LongTensor(y_train_flat)`: Convierte `y_train_flat` (las etiquetas aplanadas) de un arreglo NumPy a un tensor de PyTorch de tipo largo (entero).

- El mismo proceso se realiza para los datos de prueba (`x_test` y `y_test`).

5. **Asegúrate de que las longitudes coincidan**:
   - **Propósito**: Verificar que el número de muestras en las características (`x_train` y `x_test`) coincida con el número de etiquetas (`y_train` y `y_test`). Esto es necesario para asegurar que cada muestra de entrada tenga una etiqueta correspondiente.
   - **Código**:

```
assert len(x_train) == len(y_train), "x_train y y_tra
in no coinciden en longitud"
assert len(x_test) == len(y_test), "x_test y y_test n
o coinciden en longitud"
```

   - **Explicación**: Estas líneas verifican que el número de elementos en `x_train` coincida con el número de elementos en `y_train`, así como el número de elementos en `x_test` coincida con `y_test`. Si no coinciden, se lanza un error para indicar un problema en la preparación de los datos.

6. **Creación de DataLoaders**:
   - **Propósito**: Crear DataLoaders de PyTorch para cargar los datos en mini-batches. Los mini-batches permiten que el modelo sea entrenado de manera eficiente, reduciendo el consumo de memoria y mejorando el rendimiento computacional durante el entrenamiento.
   - **Código**:

```
from torch.utils.data import TensorDataset, DataLoade
r

# Combina las características (x) y etiquetas (y).
train_dataset = TensorDataset(x_train, y_train)
test_dataset = TensorDataset(x_test, y_test)

# Crear iteradores para cargar los datos en mini-batc
hes
train_dataloader = DataLoader(train_dataset, batch_si
```

```
ze=batch_size, drop_last=True, shuffle=True)
test_dataloader = DataLoader(test_dataset, batch_size
=1, drop_last=True, shuffle=True)
```

- **Explicación**:
  - `TensorDataset(x_train, y_train)`: Combina las características y las etiquetas del conjunto de entrenamiento en un TensorDataset, que es un contenedor que almacena datos como tensores.
  - `TensorDataset(x_test, y_test)`: Lo mismo para el conjunto de prueba.
  - `DataLoader(...)`: Crea un DataLoader que facilita la gestión de los datos. `batch_size` especifica el número de muestras por mini-batch, `drop_last=True` asegura que no se generen mini-batches de tamaño menor al especificado al final del conjunto de datos, y `shuffle=True` permite que las muestras se carguen en un orden aleatorio en cada época de entrenamiento.

7. **Imprimir las clases aprendidas**:
   - **Propósito**: Verificar las clases que el encoder aprendio y las etiquetas únicas presentes en las muestras de entrenamiento.
   - **Código**:

```
print("Clases aprendidas por el codificador:", le.cla
sses_)
print("Etiquetas únicas en y_train_raw:", np.unique(y
_train_raw))
```

   - **Explicación**:
     - `le.classes_`: Imprime las clases que el `LabelEncoder` aprendio durante la preparación de las etiquetas.
     - `np.unique(y_train_raw)`: Imprime las etiquetas únicas presentes en el conjunto de entrenamiento para asegurarse de que todas las clases están representadas correctamente.

Aprendí que transformar los datos en tensores es algo súper útil y muy práctico. Además, me di cuenta de que esto hace que el entrenamiento del modelo sea mucho más rápido, lo cual está genial para agilizar el aprendizaje y obtener mejores resultados.

Modelo : Clasificación de Posturas

1. Importar TensorFlow y otros módulos necesarios

```
import tensorflow as tf
from tensorflow.keras.layers import Input, Dense, Dropout
from tensorflow.keras.models import Model
import matplotlib.pyplot as plt
```

- **TensorFlow**: Se importa la biblioteca TensorFlow junto con otros módulos necesarios para la construcción y entrenamiento de modelos.
- `Input`, `Dense`, `Dropout`, `Model`: Estas funciones y clases de TensorFlow se utilizan para definir la arquitectura de la red neuronal y para entrenar el modelo.
- `matplotlib.pyplot`: Se importa para crear las gráficas de precisión, pérdida y validación de estas, que se utilizarán más adelante para visualizar los resultados del entrenamiento.

Se implemento diferentes funciones para la construccion del modelo para si, evaluar el que mejor daba.

- Se probo con 512 neuronas en la primera capa y 256 neuronas en la segunda capa con 200 epocas y me dio un resultado de:

```
Epoch 126/200
5/5 ━━━━━━━━━━━━━━━━━━━━ 0s 42ms/step - accuracy: 0.8207 - loss: 0.7417 - val_accuracy: 0.7814 - val_loss: 0.9006
```

El modelo muestra un buen desempeño en entrenamiento con 82.07% de exactitud y una pérdida de 0.7417. Sin embargo, en validación, la exactitud es más baja (78.14%) y la pérdida es mayor (0.9006), lo que sugiere sobreajuste.

- Se probo con 256 neuronas en la primera capa y 64 neuronas en la segunda capa, con 200 epocas y me dio un resultado de:

```
Epoch 195/200
5/5 ━━━━━━━━━━━━━━━━━━━━ 0s 26ms/step - accuracy: 0.9591 - loss: 0.1185 - val_accuracy: 0.7990 - val_loss: 0.8779
Epoch 196/200
5/5 ━━━━━━━━━━━━━━━━━━━━ 0s 20ms/step - accuracy: 0.9704 - loss: 0.0935 - val_accuracy: 0.8010 - val_loss: 0.8764
Epoch 197/200
5/5 ━━━━━━━━━━━━━━━━━━━━ 0s 22ms/step - accuracy: 0.9813 - loss: 0.0795 - val_accuracy: 0.7907 - val_loss: 0.9076
Epoch 198/200
5/5 ━━━━━━━━━━━━━━━━━━━━ 0s 30ms/step - accuracy: 0.9767 - loss: 0.0798 - val_accuracy: 0.7918 - val_loss: 0.9228
Epoch 199/200
5/5 ━━━━━━━━━━━━━━━━━━━━ 0s 20ms/step - accuracy: 0.9668 - loss: 0.0943 - val_accuracy: 0.8021 - val_loss: 0.9136
Epoch 200/200
5/5 ━━━━━━━━━━━━━━━━━━━━ 0s 21ms/step - accuracy: 0.9665 - loss: 0.1126 - val_accuracy: 0.7959 - val_loss: 0.9016
```

Tiene un rendimiento excelente en entrenamiento con una exactitud de 96.65%, pero su desempeño en validación (79.59%) es más bajo, lo que sugiere que sigue habiendo **sobreajuste**.

---

2. Función QUE MEJOR DIO para construir el modelo

```
# Función para construir el modelo basado en keypoints
def build_keypoints_model(input_shape, output_labels):
    i = Input(input_shape, dtype=tf.float32)

    # Red neuronal densa (MLP)
    x = Dense(128, activation='relu')(i)  # Más neuronas en
la primera capa
    x = Dropout(0.3)(x)
    x = Dense(64, activation='relu')(x)   # Capa adicional
con 64 neuronas
    x = Dropout(0.3)(x)
    x = Dense(output_labels, activation='softmax')(x)  # Ca
pa de salida con softmax

    return Model(inputs=[i], outputs=[x])

# Definir forma de entrada y número de clases
input_shape = (len(x_train[0]),)  # Longitud de los keypoin
ts
num_classes = len(le.classes_)  # Número de poses (etiqueta
s)
```

- `input_shape`: Se define la forma de entrada del modelo basada en la longitud de los keypoints (`len(x_train[0])`).
- `output_labels`: Número de clases o poses disponibles en las etiquetas (`len(le.classes_)`).

- `i = Input(input_shape, dtype=tf.float32)`: Define la capa de entrada de la red neuronal, donde `input_shape` especifica la dimensión de las características (keypoints) y `dtype=tf.float32` indica que se espera un tensor de punto flotante.
- **Primera capa (`Dense(128, activation='relu')`)**: Se añade una capa densa con 128 neuronas y función de activación `ReLU` para introducir no-linealidad.
- **Dropout (`Dropout(0.3)`)**: Se agrega una capa de dropout con tasa de 0.3 para regularizar el modelo y evitar sobreajuste.
- **Segunda capa (`Dense(64, activation='relu')`)**: Se añade una capa densa adicional con 64 neuronas.
- **Dropout (`Dropout(0.3)`)**: Otra capa de dropout con la misma tasa para continuar la regularización.
- **Capa de salida (`Dense(output_labels, activation='softmax')`)**: Una capa densa con un número de neuronas igual al número de clases (`output_labels`) y activación `softmax` para calcular las probabilidades de las clases.

3. Definir forma de entrada y número de clases

```
input_shape = (len(x_train[0]),)  # Longitud de los keypoin
ts
num_classes = len(le.classes_)  # Número de poses (etiqueta
s)
```

- `input_shape`: Obtiene la longitud de los keypoints de una muestra en el conjunto de entrenamiento (`x_train[0]`). Esto define cuántas características tiene cada entrada para el modelo.
- `num_classes`: Obtiene el número total de clases o etiquetas únicas (`le.classes_`). Esto se utilizará como el número de neuronas en la capa de salida.

4. Construir el modelo

```
model_keypoints = build_keypoints_model(input_shape, num_cl
asses)
```

- `model_keypoints`: Se construye el modelo usando la función `build_keypoints_model`, que define la arquitectura de la red neuronal con las características y el número de clases proporcionados.

5. Compilar el modelo

```
model_keypoints.compile(
    optimizer='adam',
    loss='sparse_categorical_crossentropy',
    metrics=['accuracy']
)
```

- `optimizer`: Se usa el optimizador `adam`, muy popular debido a su efectividad y facilidad de uso.
- `loss`: Función de pérdida `sparse_categorical_crossentropy`, adecuada para problemas de clasificación multi-clase con etiquetas enteras.
- `metrics`: Se especifica `accuracy` para evaluar el modelo durante el entrenamiento y validación.

6. Resumen del modelo

```
model_keypoints.summary()
```

- `summary()`: Muestra un resumen detallado de la arquitectura del modelo, incluyendo el número de capas, tipos de capas, neuronas en cada capa y salidas. Esto ayuda a verificar si la arquitectura se construyó correctamente y permite a los desarrolladores revisar los parámetros del modelo.

7. Entrenamiento del modelo

```
#EPOCHS = 30
EPOCHS = 100

history = model_keypoints.fit(
    x_train,
    y_train,
    validation_data=(x_test, y_test),
    epochs=EPOCHS,
```

```
    batch_size=batch_size
)
```

- `EPOCHS`: Probé entrenar el modelo con diferentes cantidades de épocas: 30, 100, 150 y 200. Aunque con menos épocas el modelo mostraba buen progreso, fue **con 100 épocas** donde obtuve los mejores resultados. Esto permitió reducir la pérdida y mejorar la precisión en validación, logrando el mejor balance entre ambos conjuntos.
- `history = model_keypoints.fit(...)`: Entrena el modelo con los datos de entrenamiento (`x_train`, `y_train`), y realiza la validación en el conjunto de prueba (`x_test`, `y_test`).
  - `validation_data`: Se utiliza para monitorizar las métricas de validación y evitar el sobreajuste.
  - `epochs`: Número total de épocas para las que se entrenará el modelo.
  - `batch_size`: Tamaño del batch, que especifica cuántas muestras se procesarán en cada iteración de entrenamiento.

```
Epoch 95/100
5/5 ━━━━━━━━━━━━━━━━━━━━ 0s 15ms/step - accuracy: 0.8915 - loss: 0.3134 - val_accuracy: 0.7887 - val_loss: 0.7295
Epoch 96/100
5/5 ━━━━━━━━━━━━━━━━━━━━ 0s 15ms/step - accuracy: 0.8904 - loss: 0.2906 - val_accuracy: 0.7856 - val_loss: 0.7103
Epoch 97/100
5/5 ━━━━━━━━━━━━━━━━━━━━ 0s 18ms/step - accuracy: 0.9214 - loss: 0.2742 - val_accuracy: 0.7907 - val_loss: 0.7125
Epoch 98/100
5/5 ━━━━━━━━━━━━━━━━━━━━ 0s 19ms/step - accuracy: 0.9038 - loss: 0.2861 - val_accuracy: 0.7711 - val_loss: 0.7243
Epoch 99/100
5/5 ━━━━━━━━━━━━━━━━━━━━ 0s 18ms/step - accuracy: 0.8932 - loss: 0.3010 - val_accuracy: 0.7845 - val_loss: 0.7205
Epoch 100/100
5/5 ━━━━━━━━━━━━━━━━━━━━ 0s 15ms/step - accuracy: 0.9056 - loss: 0.2946 - val_accuracy: 0.7804 - val_loss: 0.7206
```

Análisis del Modelo

- **Precisión en entrenamiento**: 90.56
- **Pérdida en entrenamiento**: 0.2946
- **Precisión en validación**: 78.04%
- **Pérdida en validación**: 0.7206

Conclusión:

Aunque el modelo de 200 epocas y con 256 neuronas en la primera capa tiene mejores resultados en entrenamiento, el
Modelo final seleccionado tiene un mejor balance entre entrenamiento y

validación, con una mayor capacidad de generalización. Por lo tanto, **el Modelo final** es ligeramente mejor en términos de capacidad de generalización a nuevos datos, aunque ambos modelos son buenos.

Probé usar Dropout y agregar regularización para controlar el **sobreajuste**, el mejor resultado lo conseguí sin añadir nada extra. Lo único que funcionó fue bajar el Dropout a 0.3, ya que cuando lo subí a 0.5 no dio buenos resultados.

8. Visualización de resultados

```
### Visualización de resultados

# Graficar precisión y pérdida
acc = history.history['accuracy']
val_acc = history.history['val_accuracy']
loss = history.history['loss']
val_loss = history.history['val_loss']

epochs_range = range(EPOCHS)

plt.figure(figsize=(8, 8))
plt.subplot(1, 2, 1)
plt.plot(epochs_range, acc, label='Training Accuracy')
plt.plot(epochs_range, val_acc, label='Validation Accurac
y')
plt.legend(loc='lower right')
plt.title('Training and Validation Accuracy')

plt.subplot(1, 2, 2)
plt.plot(epochs_range, loss, label='Training Loss')
plt.plot(epochs_range, val_loss, label='Validation Loss')
plt.legend(loc='upper right')
plt.title('Training and Validation Loss')
plt.show()
```

Análisis:

1. **Precisión (Accuracy)**:
   - La **precisión de entrenamiento** (en azul) crece de forma constante, alcanzando valores cercanos a 0.9 (90%), lo que indica que el modelo está aprendiendo bien de los datos de entrenamiento.
   - La **precisión de validación** (en naranja) también mejora, pero más lentamente y se estabiliza alrededor de 0.8 (80%), lo que sugiere que el modelo está logrando una buena precisión en validación, pero con algo de sobreajuste.
2. **Pérdida (Loss)**:
   - La **pérdida de entrenamiento** (en azul) disminuye rápidamente al principio y luego se estabiliza, lo que indica que el modelo sigue

aprendiendo pero a un ritmo más lento.

- La **pérdida de validación** (en naranja) sigue una tendencia similar, disminuyendo al principio y luego estabilizándose. Sin embargo, es más alta que la pérdida de entrenamiento, lo que refuerza la idea de que existe algo de sobreajuste, dado que el modelo se ajusta mejor a los datos de entrenamiento que a los de validación.

Conclusión:

El modelo muestra un buen desempeño en entrenamiento, pero existe una ligera discrepancia entre la precisión y pérdida de validación y las de entrenamiento, lo que indica **sobreajuste**.

PREDICCION POSE DE YOGA

1. **Definición de la función `predict_pose`:**
   Esta función toma como entrada una imagen, un modelo de predicción entrenado, un codificador de etiquetas (`label_encoder`) y la forma de entrada esperada del modelo.

```
def predict_pose(image_path, model, label_encoder, expected_i
    """
    Predice la pose de yoga basada en una imagen y muestra la

    Args:
        image_path (str): Ruta de la imagen a predecir.
        model (tf.keras.Model): Modelo entrenado para predeci
        label_encoder (LabelEncoder): Codificador de etiqueta
        expected_input_shape (int): Tamaño esperado de la ent

    Returns:
        str: Nombre de la pose predicha.
    """
    # Cargar la imagen
    img = cv2.imread(image_path)
    if img is None:
        raise ValueError(f"No se pudo cargar la imagen desde
```

```
    # Convertir la imagen a RGB para trabajar con MediaPipe y
    img_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)

    # Detectar keypoints usando MediaPipe
    with mp_pose.Pose(min_detection_confidence=0.5, min_track
        results = pose.process(img_rgb)
        if not results.pose_landmarks:
            raise ValueError("No se detectaron keypoints en l

        # Extraer los keypoints en un formato plano
        keypoints = [
            [landmark.x, landmark.y, landmark.z]
            for landmark in results.pose_landmarks.landmark
        ]
        keypoints_flat = np.array(keypoints).flatten()  # Apl

    # Verificar la longitud de los keypoints y ajustar si es
    if len(keypoints_flat) < expected_input_shape:
        # Rellenar con ceros si faltan valores
        keypoints_flat = np.pad(keypoints_flat, (0, expected_
    elif len(keypoints_flat) > expected_input_shape:
        # Truncar si hay demasiados valores
        keypoints_flat = keypoints_flat[:expected_input_shape

    # Ajustar la forma para el modelo
    keypoints_flat = keypoints_flat.reshape(1, -1)

    # Realizar la predicción
    predictions = model.predict(keypoints_flat)
    predicted_label = np.argmax(predictions, axis=1)  # Índic

    # Decodificar la etiqueta
    pose_name = label_encoder.inverse_transform(predicted_lab

    # Dibujar keypoints y conexiones sobre la imagen
    img_with_keypoints = img_rgb.copy()
    mp_drawing.draw_landmarks(
        img_with_keypoints, results.pose_landmarks, mp_pose.P
```

```
        mp_drawing.DrawingSpec(color=(0, 255, 0), thickness=2
        mp_drawing.DrawingSpec(color=(255, 0, 0), thickness=2
    )

    # Visualizar la imagen procesada con keypoints
    plt.figure(figsize=(8, 8))
    plt.imshow(img_with_keypoints)
    plt.title(f"Pose predicha: {pose_name}", fontsize=16, col
    plt.axis('off')
    plt.show()

    return pose_name
```

- **Carga de la imagen:**
  Se utiliza `cv2.imread()` para leer la imagen desde la ruta proporcionada. Si la imagen no se carga correctamente, se lanza un error.
- **Conversión a RGB:**
  OpenCV carga imágenes en formato BGR, pero MediaPipe y `matplotlib` esperan imágenes en formato RGB. Se usa `cv2.cvtColor()` para hacer la conversión.
- **Detección de keypoints con MediaPipe:**
  Se crea una instancia de `mp_pose.Pose` de MediaPipe, que permite detectar los puntos clave de la pose humana en la imagen. Los puntos clave se extraen de la respuesta de `pose.process()`, que devuelve un conjunto de coordenadas `x`, `y`, y `z` para cada punto clave del cuerpo. Estos puntos son luego aplanados en un array unidimensional.
- **Ajuste de la longitud de los keypoints:**
  Los keypoints aplanados pueden no coincidir con la forma de entrada esperada del modelo. Si hay menos valores, se rellenan con ceros; si hay más, se recortan.
- **Predicción de la pose:**
  El modelo se utiliza para predecir la pose con los keypoints procesados. La predicción se realiza a través de `model.predict()`, y luego se obtiene la clase predicha utilizando `np.argmax()`.

- **Decodificación de la etiqueta:**
  Se usa el codificador de etiquetas `label_encoder` para convertir el índice de la clase predicha en su nombre correspondiente.
- **Visualización de los keypoints en la imagen:**
  Se utiliza `mp_drawing.draw_landmarks()` para dibujar los puntos clave y las conexiones entre ellos en la imagen, lo que ayuda a visualizar cómo el modelo detecta los puntos clave de la pose.
- **Mostrar la imagen procesada:**
  Finalmente, se utiliza `matplotlib` para mostrar la imagen con los keypoints superpuestos.
- **Resultado:**
  La función retorna el nombre de la pose predicha.

Pruebas con imágenes de ejemplo:

Después de definir la función, se realiza la predicción para diversas imágenes de prueba. Cada imagen de prueba corresponde a una pose diferente de yoga (como "warrior2", "tree", "plank", etc.).

- Se define la ruta de la imagen.
- Se llama a la función `predict_pose()` pasando la ruta de la imagen, el modelo, el codificador de etiquetas y la forma de entrada esperada.
- Se imprime la pose predicha para cada imagen.

Conclusión del trabajo práctico:

El modelo de predicción de poses de yoga mostró un buen desempeño al predecir la pose correcta en la mayoría de las imágenes probadas. De las múltiples imágenes evaluadas, todas fueron clasificadas correctamente, excepto una. Aunque el modelo tiene una alta precisión, no alcanza el 100%, lo cual es común en modelos de visión por computadora que dependen de la calidad de la entrada y la variabilidad en las poses. Esto demuestra que, aunque el modelo puede ser útil en la mayoría de los escenarios, es importante

tener en cuenta que no siempre se alcanzará una precisión perfecta debido a los desafíos inherentes a la visión por computadora y la variabilidad humana.

Este análisis sugiere que el modelo puede ser eficaz para la clasificación de poses de yoga en condiciones controladas, pero se deben hacer mejoras continuas, como la recolección de más datos y la mejora en la precisión de la detección de keypoints, para lograr una mayor robustez y precisión.