Documentation for `main.py`

This Python script implements real-time pose estimation using the YOLOv8 model with the following functionality:

1. **Capture video frames from a webcam**
2. **Perform human pose estimation using the YOLOv8 model**
3. **Track individuals across frames by assigning unique IDs**
4. **Visualize detected keypoints and draw skeletons**
5. **Send keypoint data via a WebSocket to a frontend application**
6. **Buffer video frames for smoother processing**

Detailed Code Breakdown

Imports

```
from ultralytics import YOLO
import cv2
import time
import websockets
import json
import asyncio
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation
import torch
from torch.cuda.amp import autocast, GradScaler
```

- `ultralytics`: Contains the YOLO object detection model.
- `cv2` **(OpenCV)**: Used for real-time video capture and visualization.
- `time`: Used for timing operations and maintaining consistent frame rates.
- `websockets`: Enables WebSocket communication to transmit pose data to a frontend server.
- `json`: Converts Python objects to JSON strings for WebSocket communication.
- `asyncio`: Provides the asynchronous framework for WebSocket operations.
- `numpy`: Used for numerical operations, such as calculating distances between keypoints.
- `matplotlib`: (Unused in the current script but imported for potential animation of results).
- `torch`: Used for loading and processing the YOLO model on CPU/GPU.

- **`torch.cuda.amp`**: Provides mixed precision (FP16) acceleration when using GPU.

Device Initialization

```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print(f"Using device: {device}")
```

- Checks if a CUDA-compatible GPU is available; otherwise, defaults to CPU.
- Prints the selected device.

YOLO Model Initialization

```
model = YOLO('yolov11-pose.pt').to(device)
```

- Loads the YOLO pose estimation model (`yolov11-pose.pt`).
- Transfers the model to the specified `device`.

Video Capture Setup

```
cap = cv2.VideoCapture(0)
```

- Captures video feed from the first available webcam (`0` represents the device ID).

Keypoints and Skeleton Definitions

```
keypoint_names = ['nose', 'left_eye', 'right_eye', ...]
skeleton = [(0, 1), (0, 2), ...]
```

- **`keypoint_names`**: Names of body parts based on the COCO dataset (e.g., "nose", "left_eye").
- **`skeleton`**: Defines the connections (edges) between keypoints to form a skeletal structure.

Motion Data and Frame Buffers

```
motion_data = {name: [] for name in keypoint_names}
```

```
frame_data = []
buffer = []
buffer_size = 10
frame_interval = 1 / 30  # 30 FPS
```

- **`motion_data`**: Tracks the movement of each keypoint over time.
- **`frame_data`**: Stores processed frames for visualization or saving.
- **`buffer`**: Temporary storage of frames for smoother processing.
- **`frame_interval`**: Controls the target frame rate (30 FPS).

WebSocket Communication

```
async def send_coordinates(data):
    uri = "ws://localhost:8765"
    ...
```

- Sends keypoint data (`data`) as JSON over a WebSocket connection to `localhost:8765`.
- Implements error handling to ensure reliability during communication.

Keypoint Visualization

```
def draw_keypoints(frame, person_tracker, current_keypoints):
    ...
```

- Draws keypoints and the skeletal structure for each person.
- **Highlights**:
    - Uses `cv2.circle` to draw keypoints.
    - Connects keypoints with `cv2.line` to represent skeletons.
    - Displays unique IDs for tracked individuals near their nose keypoint.

Color Palette and Tracker Initialization

```
color_palette = [(255, 0, 0), (0, 255, 0), ...]
person_tracker = {}
next_person_id = 1
```

- Assigns distinct colors to individuals for clear visualization.
- `person_tracker`: Maintains data for tracking individuals (keypoints, color, and ID).
- `next_person_id`: Counter for assigning unique IDs to new individuals.

Keypoint Matching and Tracker Updates

```
def calculate_keypoint_similarity(kp1, kp2):
    ...
def update_person_tracker(keypoints_data, current_count, previous_count):
    ...
```

- `calculate_keypoint_similarity`:
  - Compares two sets of keypoints using Euclidean distance.
  - Ensures that confident keypoints (confidence > 0.5) are considered.
- `update_person_tracker`:
  - Adds new individuals if they enter the frame.
  - Removes individuals if they leave the frame.
  - Matches current keypoints to previously tracked keypoints for consistency.

Main Pose Estimation Loop

```
async def main():
    ...
```

- **Steps**:
  1. Reads frames from the webcam.
  2. Buffers frames for smoother processing.
  3. Pre-processes frames (resize, normalize, and transfer to `device`).
  4. Runs YOLO pose estimation to detect keypoints.
  5. Updates the `person_tracker` with new or removed individuals.
  6. Draws keypoints and skeletons on the frame.
  7. Displays the frame using OpenCV.
  8. Sends keypoint data to the WebSocket server.
  9. Handles frame skipping for maintaining a consistent frame rate.
- **Exit**: Releases the webcam and destroys all OpenCV windows when the program ends.

Running the Script

```
if __name__ == "__main__":
```

```
asyncio.run(main())
```

- Executes the `main` function using Python's asynchronous framework.

Important Considerations

1. **Model File**: Ensure `yolov11-pose.pt` exists in the working directory.
2. **WebSocket Server**: Confirm that the WebSocket server at `localhost:8765` is running.
3. **Performance**: Running on a GPU significantly improves performance.
4. **Buffer Size**: Adjust `buffer_size` for smoother processing on slower devices.
5. **Frame Interval**: Modify `frame_interval` to match your target frame rate.
6. **Keypoint Confidence Threshold**: Current confidence threshold is `0.5`.

Output

- **Real-time Video Feed**: Shows pose estimation results with keypoints, skeletons, and IDs.
- **WebSocket Data**: Sends JSON-encoded keypoint coordinates to the frontend.
- **Logs**: Prints the number of people detected and updates to the tracker in the console.

Development

- **Counting number of people in frame:** For tracking, we first have to count the number of people in the frame based on the maximum number of any keypoint. For example, if there are 2 right hands in the frame, then the code identifies 2 people in the frame.
- **Creating a python dictionary for tracking:** The dictionary is important for tracking each person's information - their id, color, and key points at the most recent point that a person entered or exited the frame (these do not update constantly).
- **Update person tracker:** this function is called when a person exits or enters (num_people changes). If the num people changes, the person tracker dictionary is updated. Later, we had to account for whether a person entered or exited and passed the difference in num_people as a function argument, this determined whether to add an entry to the person tracker or delete an entry.

- **Calculate average keypoint distance:** compare the key points in the tracker (the ones updated at the previous num_people change) to the current key points by getting the average distance between all the keypoints. This essentially creates a distance between centers of the different keypoint clusters (clusters as in an individual person's keypoints).
- **Distance Threshold:** added a distance threshold to use the distance between centers to have consistent tracking. If the calculated distance was below the threshold, the program considers it to be the same person.