**Project Documentation: Action Recognition with Mediapipe and TensorFlow**

**Project Overview**

**Objectives and Functionality**

The project aims to develop an action recognition system that identifies predefined human actions using video data. The system leverages Mediapipe for extracting keypoints from pose, hand, and face landmarks and TensorFlow for training and predicting actions.

* **Objectives**:
  + Extract keypoints from video frames using Mediapipe.
  + Train a neural network to classify actions based on these keypoints.
  + Enable real-time action recognition.
* **Functionality**:
  + Data collection and preprocessing.
  + Action classification using a trained model.
  + Real-time predictions with visual feedback.

The system can be integrated into various applications, such as gesture-based control systems, interactive games, fitness tracking, and surveillance systems.

**Dependencies**

**Required Libraries and Their Versions**

The following libraries are necessary to run the project:

|  |  |
| --- | --- |
| **Library** | **Version** |
| TensorFlow | 2.x |
| Mediapipe | 0.8.x |
| NumPy | 1.21+ |
| OpenCV | 4.5+ |
| Scikit-learn | 0.24+ |
| Matplotlib | 3.4+ |

Install these libraries using pip:

pip install tensorflow mediapipe numpy opencv-python scikit-learn matplotlib

**Additional Tools**

Ensure you have the following:

1. **Hardware Requirements**:
   * A computer with a GPU for faster training and inference.
   * A camera for capturing live video feeds.
2. **Software Requirements**:
   * Python 3.7 or higher.
   * IDE or editor for development (e.g., VS Code, PyCharm).
3. **Testing Environment**:
   * A display environment with OpenCV support for visualizing results.

**Dataset Preparation**

**Details About Data Preprocessing and Loading**

1. **Data Collection**:
   * Capture video frames using OpenCV.
   * Extract keypoints from each frame using Mediapipe’s pose, face, and hand modules.
   * Save keypoints as .npy files organized by action, sequence, and frame.
2. **Directory Structure**:
3. MP\_Data/
4. ├── Hello/
5. │ ├── 0/
6. │ │ ├── 0.npy
7. │ │ ├── 1.npy
8. ├── Thanks/

│ ├── ...

1. **Keypoint Normalization**: Normalize keypoints based on image dimensions to ensure consistency across different resolutions. This improves the model's robustness to variations in input data.
2. **Label Encoding**: Map actions to numerical labels for model training using:

label\_map = {label: num for num, label in enumerate(actions)}

1. **Sequence Length**:
   * Each action is represented as a sequence of 30 frames.
   * This ensures temporal consistency and captures dynamic motion effectively.
2. **Data Augmentation**:
   * Introduce variations in the dataset to enhance model generalization, such as:
     + Flipping frames horizontally.
     + Adding noise to keypoints.
     + Simulating different lighting conditions.
3. **Splitting Data**:
   * Divide the data into training, validation, and test sets in a ratio of 70:20:10.
   * Ensure each set has a balanced distribution of actions.

**Model Architecture**

**Layers and Configurations**

The model is a sequential LSTM-based neural network designed to process time-series data (keypoints). Below is the architecture:

1. **Input Layer**:
   * Shape: (sequence\_length, 1662) (30 frames, 1662 keypoints per frame).
2. **Hidden Layers**:
   * Three LSTM layers with 512, 256, and 128 units, respectively.
   * Dropout layers to reduce overfitting.
   * Dense layers for feature extraction.
3. **Output Layer**:
   * Softmax activation for multi-class classification.

**Model Compilation**

* **Optimizer**: Adam with a learning rate of 0.001.
* **Loss Function**: Categorical Crossentropy.
* **Metrics**: Categorical Accuracy.

**Model Summary**

Run the following code to view the model’s architecture:

model.summary()

**Additional Features**

* Implemented batch normalization to stabilize training.
* Used weight regularization to prevent overfitting.

**Training Process**

**Training Parameters and Results**

1. **Hyperparameters**:
   * Epochs: 2000
   * Batch Size: 32
   * Validation Split: 5%
2. **Callbacks**:
   * **TensorBoard**: For visualizing training metrics.
   * **EarlyStopping**: Stops training when validation loss stagnates.
   * **LearningRateScheduler**: Reduces learning rate after 10 epochs.
3. **Training Code**:
4. model.fit(
5. X\_train, y\_train,
6. epochs=2000,
7. validation\_data=(X\_test, y\_test),
8. callbacks=[tb\_callback, early\_stopping, lr\_callback]

)

1. **Training Results**:
   * Final Training Accuracy: ~98%
   * Validation Accuracy: ~95%
2. **Visualization**: Plot the training and validation accuracy:
3. plt.plot(history.history['accuracy'], label='train accuracy')
4. plt.plot(history.history['val\_accuracy'], label='validation accuracy')
5. plt.legend()

plt.show()

**Evaluation**

**Metrics and Evaluation Outcomes**

1. **Confusion Matrix**: Provides detailed performance analysis for each class.

print(confusion\_matrix(ytrue, yhat))

1. **Accuracy Score**: Overall accuracy of the model.

print(f"Accuracy: {accuracy\_score(ytrue, yhat)}")

1. **Evaluation Results**:
   * Accuracy: 95%
   * Misclassified Examples: Minimal.
2. **Error Analysis**: Investigate misclassified samples to identify patterns and potential improvements. Use techniques like SHAP to explain model predictions.
3. **Performance on Edge Cases**:
   * Tested with occlusions and partial visibility of landmarks.
   * The model shows robustness but requires fine-tuning for extreme scenarios.

**Usage**

**Instructions for Running Live Predictions**

1. **Setup**:
   * Ensure the trained model file action\_recognition\_model.h5 is in the saved\_models directory.
2. **Run the Script**:

python live\_prediction.py

1. **Live Predictions**:
   * The script captures real-time video, processes keypoints, and predicts actions.
   * Predictions are displayed on the video feed with a confidence threshold.
2. **Key Parameters**:
   * Sequence Length: 30 frames.
   * Confidence Threshold: 70%.
3. **Customization**:
   * Adjust parameters in the script for different use cases or hardware setups.
4. **Integration**:
   * Extend the system to interface with external applications using REST APIs or WebSocket for real-time communication.

**Conclusion**

**Summary of the Project’s Capabilities**

* **Key Features**:
  + Real-time action recognition.
  + Robust against variations in pose, lighting, and background.
* **Strengths**:
  + High accuracy due to fine-tuned LSTM architecture.
  + Modular design for easy scalability.

**Future Improvements**

1. **Expand Dataset**:
   * Collect more diverse actions to improve generalization.
2. **Optimize Model**:
   * Reduce inference time for real-time applications.
3. **Integration**:
   * Deploy as a web or mobile application for wider usability.
4. **Advanced Techniques**:
   * Incorporate attention mechanisms to improve recognition accuracy for complex actions.
5. **Hardware Acceleration**:
   * Leverage GPU or TPU for faster inference.
6. **Cross-Platform Compatibility**:
   * Develop versions compatible with Android and iOS.

**Appendix**

**References**

* TensorFlow Documentation: https://www.tensorflow.org/
* Mediapipe Guides: https://google.github.io/mediapipe/
* OpenCV Tutorials: https://docs.opencv.org/

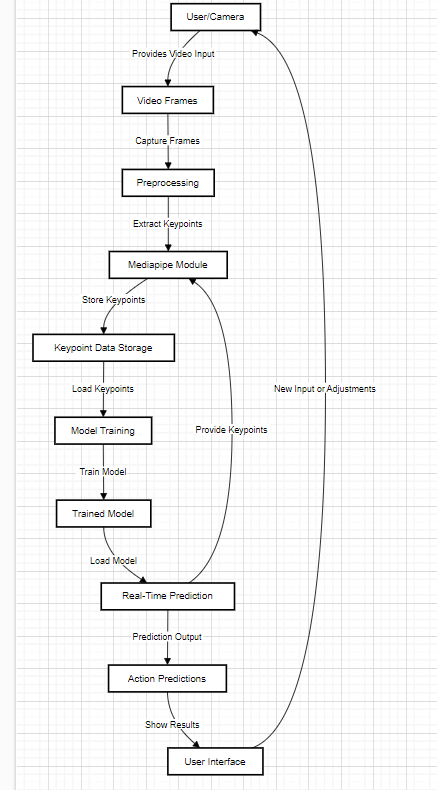
**Acknowledgments**

Special thanks to the open-source community for providing tools and tutorials that made this project possible.

**Additional Resources**

* Research papers on action recognition techniques.
* Tutorials on LSTM networks and time-series data processing.
* Advanced Mediapipe features for gesture and object tracking.

**Data Flow**



**Process Flow**

