Project: Video Frame Edge Detection and Multi-Class Classification

This project extracts frames from videos at specified intervals, computes edge images using Canny edge detection, and trains a custom convolutional neural network (CNN) model to perform multi-class classification on the edge images.

Important Note:

Potential Bias: Due to the presence of an example video with a maximum length in the training data, the model might exhibit bias or be shifted towards features prevalent in that particular video. This could lead to less accurate predictions for videos with significantly different characteristics.

Getting Started

Prerequisites: Python 3.x, pandas, NumPy, OpenCV (cv2), Pillow (PIL), PyTorch, torchvision

Installation: Ensure you have the required libraries installed using pip install <library\_name>.

Data: The code expects video files to be located in the /content/drive/MyDrive/Internshala/Videos directory and a CSV file containing ground truth labels to be located in /content/drive/MyDrive/internship.csv. Make sure to replace these paths with your actual data locations.

Usage: Execute the Python script (assuming it's named main.py) using python main.py.

Development

Folder Structure:

The code is likely located in a single Python script (main.py). However, if you have a more complex structure, consider adding a brief overview of the folders and their contents.

Development Workflow:

Feel free to describe your recommended development workflow, including testing procedures, code formatting conventions, and any version control practices you follow (e.g., Git).

Code Overview

Video Frame Extraction:

The code uses cv2.VideoCapture to read videos.

Frames are extracted at a specified interval using frame rate and interval seconds.

Edge Image Generation:

Grayscale conversion is performed using cv2.cvtColor.

Gaussian blurring reduces noise using cv2.GaussianBlur.

Canny edge detection identifies edges using cv2.Canny.

Data Preparation:

The CSV file is loaded using pd.read\_csv.

Data is split into training, cross-validation, and test sets based on your assumptions about the CSV file format (clarify how data is split in the CSV file).

Data is converted to PyTorch tensors for efficient deep learning processing.

Custom CNN Model Definition:

The CustomModel class defines the CNN architecture.

It uses convolutional layers for feature extraction followed by max pooling for dimensionality reduction.

The model has four output branches, each performing linear classification for a specific feature.

Consider adding comments to explain the rationale behind the chosen architecture.

Training:

A training loop iterates over epochs and minibatches.

The CrossEntropyLoss function calculates the loss for multi-class classification.

The Adam optimizer updates model weights.

Consider including hyperparameter tuning strategies (learning rate, batch size) if applicable.

Model Evaluation (Optional):

The code currently focuses on training. You can add a section for model evaluation on the test set, including metrics like accuracy, precision, recall, or F1-score.

Model Saving and Loading:

The model's state dictionary is saved using torch.save for persistence.

The code demonstrates how to load the saved model using torch.load for future predictions.

Additional Notes

Remember to tailor this README to your specific project and data.

Consider including comments throughout your code to enhance readability and maintainability.

Explore unit testing frameworks (e.g., unittest or pytest) to ensure code quality.

Visualizations (e.g., sample input/output images, training/validation loss curves) can enhance understanding.

Mitigating Bias:

Here are some strategies to consider for reducing potential bias in your model:

Potential Bias: Due to the presence of an example video with a maximum length in the training data, the model might exhibit bias or be shifted towards features prevalent in that particular video. This could lead to less accurate predictions for videos with significantly different characteristics. It's important to be aware of this potential bias and consider mitigation strategies (see below).

Data Augmentation: Artificially create variations of your existing video data (e.g., cropping, flipping, color jittering) to increase the diversity of features the model learns from.

Collect More Representative Data: If possible, gather video samples with a wider range of lengths and characteristics to provide a more balanced foundation for training.