Model Training

Exploratory Data Analysis (EDA)

Dataset Overview

- The dataset used is **CIFAR-10**, consisting of 60,000 32x32 color images across 10 classes.
- Class labels include: airplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck.

Class Distribution Analysis

- The dataset is analyzed to check the distribution of classes.
- A histogram is plotted to visualize class balance.
- Ensures no major class imbalance, making standard training approaches viable.
- Observed that all classes are equally balanced in training data

Image Visualization

- Random samples from the dataset are displayed using `matplotlib`.
- This helps understand variations in unnormalised image quality, lighting, and composition.

Feature Representation with t-SNE

- t-SNE (t-Distributed Stochastic Neighbor Embedding) is used for dimensionality reduction.
- Helps visualize feature clusters in a 2D space.
- Provides insights into the separability of different classes based on features.
- Performed t-SNE analysis on the feature maps generated by the trained model by removing the classification head, on the test data
- Observed clear separation of 10 classes, which is the reason behind good classification performance of the model

Data Preprocessing and Feature Engineering

Transformations and Augmentations

- **Training Data**: The images are augmented using:

- `RandomResizedCrop(224)`: Randomly crops the image and resizes it to 224x224.
- `RandomHorizontalFlip()`: Randomly flips the image horizontally.
- `ToTensor()`: Converts the image to a PyTorch tensor.
- `Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])`: Normalizes pixel values to align with ImageNet statistics.

- **Validation and Test Data**: The images undergo:
  - `Resize(224)`: Resizes images to 224x224.
  - `CenterCrop(224)`: Crops the central portion of the image.
  - `ToTensor()` and `Normalize()` with the same parameters as the training data.

Dataset Splitting

- The CIFAR-10 dataset is used.
- The dataset is downloaded using `datasets.CIFAR10()`.
- The validation and test sets are created using an 80-20 split from the original test set via `train_test_split()`.
- Data is loaded using `DataLoader()` with `shuffle=True` for training to introduce randomness.

Model Selection and Optimization

Model Architecture

- The chosen model is **VGG16**, a deep convolutional neural network pre-trained on ImageNet.
- The classifier's final layer is modified:
  - `model.classifier[6] = nn.Linear(model.classifier[6].in_features, num_classes)`, adapting it for 10 CIFAR-10 classes.

Training Strategy

- **Loss Function**: `CrossEntropyLoss()` for multi-class classification.
- **Optimizer**: Adam optimizer (`lr=0.0001`), which adapts learning rates dynamically.
- **Learning Rate Scheduler**: `LinearLR()` starts with a factor of 0.5 and adjusts over 4 iterations.
- **Training Loop**:
  - Iterates over epochs.
  - Computes loss and gradients using `loss.backward()` and `optimizer.step()`.

    - Evaluates validation loss and accuracy at each epoch.
- The trained model is saved as `myModel.pth`.

Explainability

Explainability using Grad-CAM (`gradio_gradcam.py`)

- Uses **Grad-CAM** to visualize the model's decision-making process.
- Extracts the **third-to-last feature layer** as the target for Grad-CAM.
- Generates a heatmap overlaying the most important regions for classification.
- Displays both the original and Grad-CAM visualized images side by side.
- Run `python gradio_gradcam.py` launch the gradio interface for prediction and Grad-CAM visualisation.

Deployment Strategy & API Usage Guide

1. Deployment Strategy

Local Deployment (Without Docker)

This method is useful for development and testing. The steps are:

1. Clone the repository.
2. Create and activate a virtual environment.
3. Install dependencies using `pip install -r requirements.txt`.
4. Start the FastAPI server with `uvicorn app:app --host 0.0.0.0 --port 8000 --reload`.
5. Launch the Streamlit frontend using `streamlit run streamlit.py`.

Containerized Deployment (Using Docker)

This method ensures a portable and consistent environment for deployment.

1. Clone the repository.
2. Build the Docker image using `docker build -t fastapi-ml-app ..`
3. Run the container with `docker run -p 8000:8000 fastapi-ml-app`.
4. For Streamlit, use `docker run -p 8501:8501 fastapi-ml-app streamlit run streamlit.py`.

Cloud Deployment (AWS, GCP, or Azure)

For scalable deployment, the service can be hosted on cloud platforms:

- **AWS EC2**: Run the container on an EC2 instance with security group configurations.
- **Google Cloud Run**: Deploy using containerized services with automatic scaling.
- **Azure App Service**: Deploy the FastAPI service using Azure’s container instances.

2. API Usage Guide

Base URL

The API runs at `http://127.0.0.1:8000` (local) or `http://<server-ip>:8000` (remote).

Endpoints

1. Model Inference

`POST /predict` – Takes input data and returns predictions.

- **Request Example:**

json
CopyEdit
```
{
  "image": "<base64_encoded_image>"
}
```

- **Response Example:**

json
CopyEdit
```
{
  "prediction": "Class_A"
}
```

3. API Documentation

- Open **Swagger UI** at `http://127.0.0.1:8000/docs`
- Open **Redoc UI** at `http://127.0.0.1:8000/redoc`

---

3. Frontend Usage (Streamlit Interface)

To interact with the model via a UI:

- Run `streamlit run streamlit.py`.
- Open `http://127.0.0.1:8501` in a browser.
- Upload an image and get model predictions in real-time.