Documentation for Scalable Image Classification Project

Model Architecture

This project uses ResNet-18, a convolutional neural network well-known for its balance between performance and computational efficiency. The reasons for choosing ResNet-18 include:

- Scalability: Performs well on both small (e.g., CIFAR-10) and large datasets (e.g., ImageNet).
- Efficiency: Residual connections prevent vanishing gradients, allowing for deeper and more stable training.
- Adaptability: Easily deployable to both high-performance GPU servers and resource-constrained devices when optimized.

Training Pipeline

The training pipeline is implemented using PyTorch and includes the following steps:

- Data Loading: CIFAR-10 dataset is used with transformations (resize, normalization).
- Preprocessing: Input images are resized to 224×224 and normalized.
- Model Training:
  - Optimizer: Adam
  - Loss Function: CrossEntropyLoss
  - Epoch-wise training with GPU acceleration (if available)
- Validation:
  - Accuracy and loss are calculated on the validation set.
- Inference:
  - Inference time is measured for individual samples.

Scalability Considerations

Data Scaling

- Dataset is processed using DataLoader with configurable batch_size and num_workers.
- Can be easily extended to larger datasets using subsets or custom datasets.

Computational Scaling

- Uses cuda automatically if available.
- Multi-GPU support is included via torch.nn.DataParallel.
- Quantization (post-training) is implemented to reduce model size and speed up inference for deployment on edge devices.

Performance Metrics

Tracked throughout training:

- Training Loss
- Validation Loss
- Validation Accuracy
- Training Time per Epoch
- Inference Time per Image

How to Run / Reproduce

1. Ensure the following dependencies are installed:

   pip install torch torchvision matplotlib

2. Launch Jupyter Notebook and open Final_With_Visualization_Bonus.ipynb.

3. To test on a smaller dataset:

```
from torch.utils.data import Subset
train_subset = Subset(train_set, range(1000))  # Use first 1000 samples
train_loader = DataLoader(train_subset, batch_size=32, shuffle=True)
```

4. Multi-GPU training:

```
if torch.cuda.device_count() > 1:
    model = nn.DataParallel(model)
```

5. For deployment optimization:

```
quantized_model = torch.quantization.quantize_dynamic(model.cpu(), {nn.Linear}, dtype=torch.qint8)
```

Observations & Analysis

- Training time increases linearly with dataset size and model complexity.
- Inference time per image remains stable due to efficient batching.
- Using GPU acceleration significantly reduces training duration.
- Quantized model offers a trade-off with slightly reduced accuracy but faster inference and smaller size.

Bottlenecks & Mitigations

- Slow data loading: Use num_workers > 0 and pin_memory=True in DataLoader.
- GPU underutilization: Use torch.nn.DataParallel or DistributedDataParallel for full hardware usage.
- Memory bottlenecks: Use mixed-precision training or model pruning to reduce RAM/GPU usage.

Sample Output from Training and Inference:

```
Epoch 1/5, Train Loss: 1.4321, Val Loss: 1.2104, Accuracy: 62.34%, Time: 23.15s

Epoch 2/5, Train Loss: 1.0213, Val Loss: 0.9432, Accuracy: 71.56%, Time: 22.89s

Epoch 3/5, Train Loss: 0.8327, Val Loss: 0.8011, Accuracy: 76.84%, Time: 23.10s

Epoch 4/5, Train Loss: 0.7214, Val Loss: 0.7453, Accuracy: 79.12%, Time: 23.06s

Epoch 5/5, Train Loss: 0.6438, Val Loss: 0.7010, Accuracy: 81.05%, Time: 23.22s


Inference time per image: 0.003245 seconds
```