Final Report

1. Overview

This project uses PyTorch and the ResNet18 architecture to classify images. The process begins with loading and concatenating `.npy` files containing data and labels, followed by creating a custom dataset, applying transformations/augmentations, and finally training and validating a model. Below is a detailed explanation of each step in the workflow.

2. Data Import and Concatenation

- **Data Files and Label Files**:The script specifies multiple `.npy` files for the images (data0.npy, data1.npy, data2.npy) and corresponding label files (lab0.npy, lab1.npy, lab2.npy).
- **Loading and Merging**:Each file is loaded with NumPy's `np.load`. The resulting arrays are concatenated along the first axis. This results in two large NumPy arrays: one containing all the images (`data`) and the other containing all the labels (`labels`).
- **Shape Verification**:After concatenation, the script prints out the shapes of the `data` and `labels` arrays to confirm that everything loaded correctly.

3. Custom Dataset Class

- **Purpose**:A `CustomDataset` class, extending PyTorch's `Dataset`, is defined to handle the data in a convenient way for DataLoader.
- **Key Steps**:
  1. The dataset stores `data` and `labels`.
  2. The `__getitem__` method:

    - Reshapes the sample from (40, 168) to maintain its single-channel (grayscale) format.
    - Converts it to float32.
    - Optionally applies a transformation pipeline if provided.
3. The `__len__` method returns the total number of samples in the dataset.

4. Data Transformations (Augmentations)

- **Training Transform**:
    - Converts the NumPy array to a PIL image.
    - Resizes it to 224×224 (matching ResNet18 input size).
    - Applies random horizontal flips, random rotations, and random affine transformations to augment the dataset.
    - Converts the image back to a PyTorch tensor and normalizes pixel values (mean=0.5, std=0.5).
- **Validation Transform**:
    - Similar to the training transform, but typically without random augmentations (so the evaluation data remains consistent).

These transformations help improve generalization by exposing the model to a variety of augmented samples during training.

5. Splitting the Dataset

- **Train, Validation, and Test Sets**:
    - The first 20,000 samples are designated for training, with full augmentations.

- The next 4,000 samples (from index 20,000 to 24,000) are for validation, with a simpler transform (no random augmentations).
- All remaining samples (index 24,000 onwards) form the test set, often with either no transform or only basic normalization.

6. DataLoaders

- **Creation**:Three DataLoaders (`train_loader`, `val_loader`, and `test_loader`) wrap the custom datasets.
- **Parameters**:
  - Batch size is 32 for all sets.
  - Training data is shuffled at every epoch (`shuffle=True`), while validation and test sets are not shuffled (`shuffle=False`).

7. Model Definition (ResNet18)

- **Pretrained Model**:A pre-trained ResNet18 from `torchvision.models` is loaded with `pretrained=True`, which initializes weights from ImageNet.
- **Input Channel Modification**:The original ResNet expects 3 input channels (RGB). Here, the first convolution layer is replaced to accept a single grayscale channel.
- **Output Layer Modification**:The final fully connected layer (`fc`) is replaced with a new layer whose output size matches the number of classes in the dataset. This is determined by the unique labels.

8. Loss Function, Optimizer, and Scheduler

- **Loss Function**: `nn.CrossEntropyLoss` is used for multi-class classification.

- **Optimizer**:An Adam optimizer is chosen, starting with a learning rate of 0.001.
- **Scheduler**:A StepLR scheduler reduces the learning rate by a factor (`gamma=0.1`) every 5 epochs (set by `step_size=5`).

9. Training Loop

- **Function**:A function `train_model` encapsulates the training/validation procedure.
- **Training Phase**:
    1. `model.train()` is called to set the model to training mode.
    2. For each batch, images and labels are moved to the device (GPU/CPU).
    3. The optimizer gradients are zeroed.
    4. A forward pass computes predictions, and the cross-entropy loss is calculated.
    5. A backward pass (`loss.backward()`) updates the model's weights via `optimizer.step()`.
    6. Accumulates training loss and the number of correct predictions.
- **Validation Phase**:
    1. `model.eval()` is called, and gradients are not computed (`with torch.no_grad()`).
    2. Validation loss and accuracy are computed over the validation set.
- **Scheduler Step**:The learning rate is updated at the end of each epoch via the scheduler.
- **Saving the Best Model**:The best validation accuracy observed across epochs is stored, and the model's weights are saved to `best_model.pth` if an improvement occurs.

---

10. Model Training and Future Testing

- **Model Training**:The script trains for 10 epochs. During each epoch, it prints training/validation losses and accuracies.
- **Model Testing** :A test model function could then be applied to the test set to measure final performance. This step is commented out in the code, but can be implemented similarly to validation.