Neural Networks and Deep Learning Coursework - 220465290

Task 1: Dataset Loading and Data Loaders

To begin our solution, we use the CIFAR-10 dataset available through the torchvision.datasets module. We apply a set of data transformations to the dataset to prepare it for training and evaluation. For the training set, we use data augmentation techniques such as random horizontal flipping and random cropping, which help improve model generalisation by introducing variability in the input data. For the test set, we apply only normalisation to ensure fair evaluation on unaltered data. The data is loaded using PyTorch's DataLoader with batch sizes of 128 for training and 100 for testing.

Task 2: Model Architecture

The core of our model is a custom convolutional neural network built with PyTorch, structured into 3 main components: a stem, a backbone, and a classifier.

The stem layer is a standard convolutional layer that processes the raw input images and expands the feature space from 3 channels to 64. It includes a convolution layer followed by batch normalization and ReLU activation.

The backbone of the model is composed of two Expert Blocks, each integrating a dynamic routing mechanism through the use of expert branches and parallel convolutional experts. Each Expert Block contains two main components: an expert branch and a set of convolutional experts. The expert branch receives an input tensor — either from the stem (in the first block) or the preceding block (in subsequent ones) — and processes it to generate a soft attention vector. This is done by applying global average pooling, then passing the pooled features through two fully connected layers with a reduction ratio, separated by a non-linear ReLU activation. The final output is passed through a softmax function to produce the attention weights. These attention weights are then used to dynamically combine the outputs of parallel convolutional experts. Each expert is a small convolutional subnetwork processing the same input, and their outputs are weighted by the elements. This results in the final output of the block, computed as a weighted sum. This architecture allows the model to adaptively specialize across different patterns in the data by softly routing input features through different expert paths.

The classifier consists of an adaptive average pooling layer followed by flattening, dropout for regularization, and a final linear layer mapping to the 10 output classes. We apply a dropout of 0.2 to reduce overfitting.

This modular architecture with expert routing is designed to enhance representational capacity while keeping the overall model relatively compact and efficient.

Task 3: Loss Function and Optimiser

We use the standard cross-entropy loss for multiclass classification. For optimisation, we employ the Adam optimiser, which adapts learning rates per parameter and is well-suited for problems with sparse gradients. The initial learning rate is set to 0.001.

To improve convergence and help escape plateaus, we also apply a learning rate scheduler — specifically, PyTorch's StepLR. The scheduler reduces the learning rate by a factor of 0.5 every 20 epochs, allowing the model to settle into local minima more effectively over time.

Task 4: Training Script and Implementation Details

Our training loop includes two main phases for each epoch: training and evaluation. In each training epoch, the model processes all batches in the training dataset, computes loss, performs backpropagation, and updates its weights. Accuracy and average loss are recorded. After training, the model is evaluated on the test set, and the corresponding metrics are logged.

We store all losses and accuracies for both training and testing over the full 100-epoch training cycle. At the end of training, we visualize the evolution of the loss and accuracy using line plots.

Hyperparameters used:

➢ Optimiser: Adam

➢ Learning rate: 0.001

➢ Learning rate scheduler: StepLR with step size 20, dividing LR by 2 per step

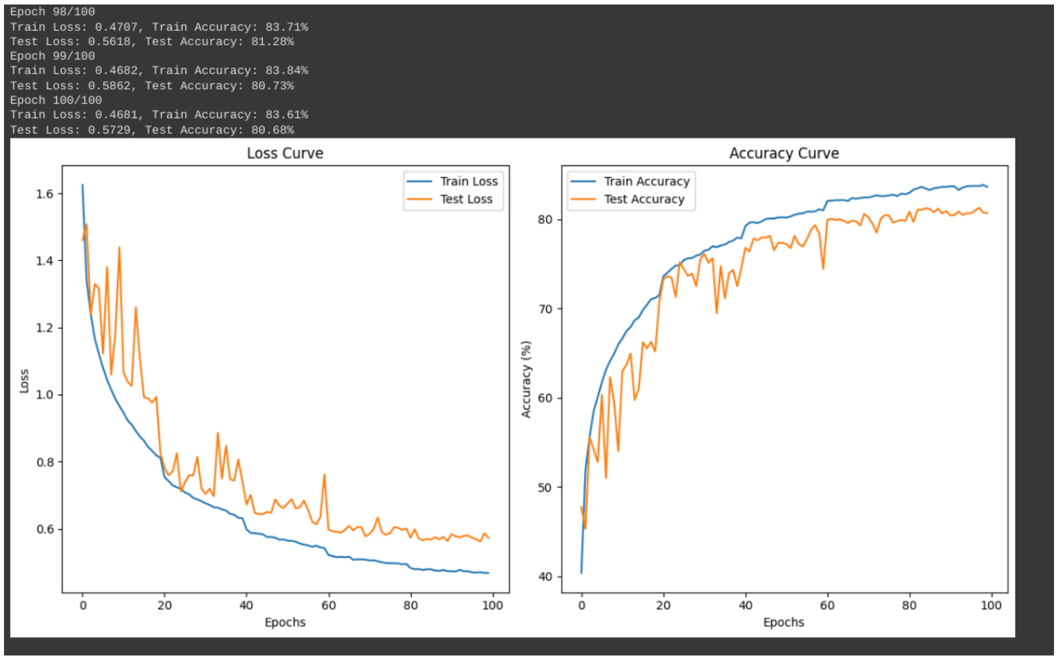
➢ Epochs: 100

➢ Batch size: 128 (train), 100 (test)

➢ Dropout: 0.2

➢ Number of experts per block: 4

➢ Reduction ratio (ExpertBranch): 4

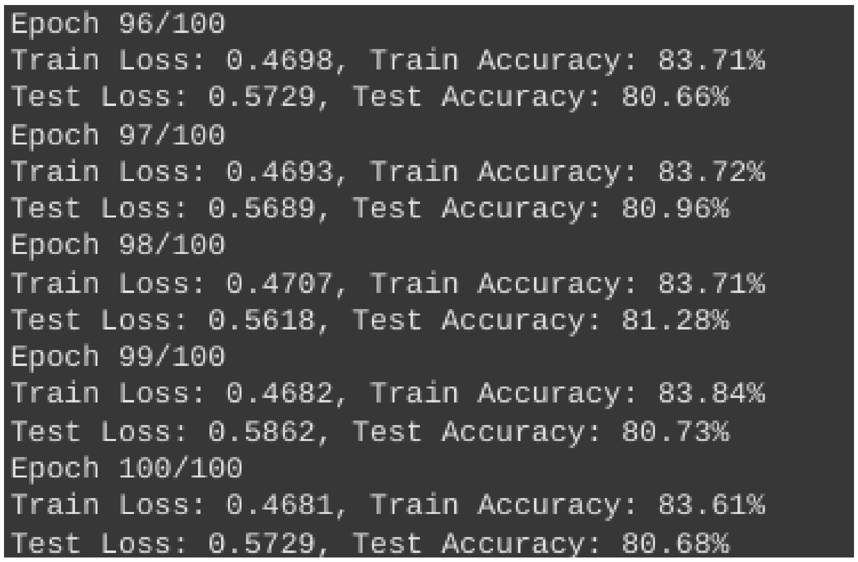


```
Epoch 98/100
Train Loss: 0.4707, Train Accuracy: 83.71%
Test Loss: 0.5618, Test Accuracy: 81.28%
Epoch 99/100
Train Loss: 0.4682, Train Accuracy: 83.84%
Test Loss: 0.5862, Test Accuracy: 80.73%
Epoch 100/100
Train Loss: 0.4681, Train Accuracy: 83.61%
Test Loss: 0.5729, Test Accuracy: 80.68%
```

These plots show that the model steadily improves over time. The learning rate scheduler helps prevent the model from plateauing too early, and the use of data augmentation and dropout improves generalisation, reducing overfitting.

Task 5: Final Accuracy and Evaluation

After training for 100 epochs, our custom model achieved results between 80% and 85%. This performance indicates the model effectively learned to distinguish between the 10 classes, and the expert routing architecture contributed to its flexibility in capturing diverse patterns in the data. Throughout training, we observed a general upward trend in both training and test accuracy, though the improvement became gradually slower over time. In later epochs, the model experienced some minor plateaus, suggesting that while it continued to learn, the rate of progress diminished. This behavior is typical as the model converges, and it highlights the importance of strategies like learning rate scheduling to sustain meaningful improvements in performance.



```
Epoch 96/100
Train Loss: 0.4698, Train Accuracy: 83.71%
Test Loss: 0.5729, Test Accuracy: 80.66%
Epoch 97/100
Train Loss: 0.4693, Train Accuracy: 83.72%
Test Loss: 0.5689, Test Accuracy: 80.96%
Epoch 98/100
Train Loss: 0.4707, Train Accuracy: 83.71%
Test Loss: 0.5618, Test Accuracy: 81.28%
Epoch 99/100
Train Loss: 0.4682, Train Accuracy: 83.84%
Test Loss: 0.5862, Test Accuracy: 80.73%
Epoch 100/100
Train Loss: 0.4681, Train Accuracy: 83.61%
Test Loss: 0.5729, Test Accuracy: 80.68%
```

While the final test accuracy falls within a suitable range, there are several avenues for further improvement. Enhancing the model's depth or experimenting with additional expert blocks could increase its representational power. Incorporating more advanced regularisation techniques, may also help combat overfitting. Additionally, fine-tuning the learning rate schedule might further boost performance.

.