**Training Details and Hyperparameters**

The CNN model was trained on the CIFAR-10 dataset with a clear aim of achieving optimal accuracy while avoiding common pitfalls such as overfitting. Below is a comprehensive list of the training details and hyperparameters utilized in this assignment. This PDF file will hopefully go in depth for Tasks 4 and 5:

**1. Dataset**

* Dataset: CIFAR-10
* Training Samples: 50,000 images
* Validation/Test Samples: 10,000 images
* Image Size: 32x32 pixels, RGB Color

**2. Data Augmentation**

Data augmentation was applied during training to enhance model generalization:

* Random Horizontal Flips: Helps the model generalize to mirrored orientations.
* Random Crops: (padding=4) Introduces slight positional variations to improve robustness.
* Normalization: Standard normalization to mean [0.5, 0.5, 0.5] and standard deviation [0.5, 0.5, 0.5], scaling inputs to a range of [-1, 1].

**3. DataLoader Configuration**

* Batch Size: 256 (chosen to balance GPU utilization and training stability)
* Shuffle: Enabled for training set; disabled for validation set
* Workers: num\_workers=2 for parallel data loading
* Pin Memory: Enabled (pin\_memory=True) to accelerate data transfer to GPU

**4. Model Hyperparameters**

* Initial Convolutional Layer Channels: 64
* Number of Backbone Blocks: 3 (each with 4 experts)
* Reduction factor in Expert Branch: reduction=4 (dimensionality reduction before expert weighting)
* Kernel Size: 3x3 for all convolutional layers
* Pooling: Max pooling layers (kernel\_size=2, stride=2)
* Classifier: Fully connected layer preceded by Global Average Pooling

**5. Training Hyperparameters**

* Number of Epochs: 50 (sufficient to achieve convergence without excessive training time)
* Optimizer: Adam optimizer
  + Learning Rate: Initially set at 0.0045
  + Weight Decay: 1e-4 (regularization to prevent overfitting)
* Loss Function: Cross-Entropy Loss (standard for multi-class classification tasks)
* Mixed Precision Training: Enabled using PyTorch's torch.cuda.amp.GradScaler for faster GPU computation and reduced memory consumption.

**6. Learning Rate Scheduler**

A learning rate scheduler was employed to further optimize training dynamics:

* Type: Step Learning Rate Scheduler (StepLR)
* Step Size: 5 epochs
* Gamma: 0.5 (reducing the learning rate by half every 8 epochs)

**Training Script and Report**

The training script is implemented inside the function train\_model(...) and contains all the necessary steps for training and evaluating the model across epochs. It handles:

* Forward and backward passes
* Loss calculation
* Accuracy tracking
* Gradient scaling for mixed precision training
* Learning rate scheduling
* Validation on test data

The key structure of the training loop is:

for epoch in range(num\_epochs):

model.train()

...

optimizer.zero\_grad()

...

scaler.scale(loss).backward()

scaler.step(optimizer)

scaler.update()

At the end of training, these metrics are visualized using matplotlib, meeting the requirement to show:

* Loss evolution curve
* Accuracy evolution curve

This visual feedback helps interpret convergence behaviour and training stability.

**Final Model Accuracy on CIFAR-10**

The model's final performance is evaluated on the CIFAR-10 validation set. Accuracy is calculated at the end of each epoch using:

with torch.no\_grad():

for inputs, labels in testloader:

outputs = model(inputs)

...

correct += predicted.eq(labels).sum().item()

The expected accuracy range using this training setup is between 85% and 90%, with 50 epochs being completed and with the given optimizer and scheduler.

A graph of a graph of a graph of a graph of a graph of a graph of a graph of a graph of a graph of a graph of a graph of a graph of a graph of

Description automatically generated

Epoch 50: Train Acc=92.47% | Test Acc=86.44%. This is the final accuracies from the model that were calculated at epoch 50. This final result may vary when tested.

**Hyperparameter Analysis and Justification**

**Learning Rate (LR = 0.0045)**

This value is moderately high — it ensures fast initial learning, while being low enough to avoid divergence. It was chosen after testing multiple LR values, as higher values (e.g. lr = 0.01) resulted in unstable loss curves and lower final accuracy. Lower values (e.g. lr = 0.001) resulted in very slow convergence, taking longer to learn key patterns in the data.

**StepLR Scheduler**

scheduler = torch.optim.lr\_scheduler.StepLR(optimizer, step\_size=8, gamma=0.5)

This reduces the learning rate by half every 8 epochs (gamma = 0.5). Early epochs explore the loss landscape broadly with a high LR. Later epochs fine-tune the model with a lower LR for better generalization.

By using a scheduler, this reduces and prevents overfitting by reducing large updates when the model is close to convergence. This encourages better performance on the validation set over time.

**Using different StepLR values:**

Longer step size (step\_size = 10): Learning rate decays too slowly, possibly causing overfitting or slower convergence. Higher gamma (gamma = 0.8): Less aggressive decay, may not fine-tune weights properly. Lower gamma (gamma = 0.1): Too aggressive; model may stop learning effectively early on.

**Number of Channels (64)**

self.conv1 = nn.Conv2d(3, 64, kernel\_size=3, stride=1, padding=1)

The initial convolution layer uses 64 channels. 64 is a good trade-off between representational capacity and computational cost. It allows the network to capture complex features like edges, textures, and colour blobs early on.

**Effect of changing it:**

* Lower (e.g. 32): Reduces capacity and may hurt accuracy.
* Higher (e.g. 128): Increases capacity but also GPU memory usage — may lead to overfitting or slower training without significant accuracy gain unless the dataset is larger.

**Number of Expert Convolutions per Block (K = 4)**

self.convs = nn.ModuleList([nn.Conv2d(...)] \* K)

Each BackboneBlock contains K = 4 convolutional experts, which are dynamically weighted by the ExpertBranch. This provides the model with adaptive flexibility to choose between different filters based on the input — helping it generalize better.

With 4 experts, there is enough diversity in learned filters without overly increasing the parameter count. Too few (e.g. K=2) would limit adaptive capacity, while too many (e.g. K=8) might lead to parameter inefficiency and slower training.