Neural Networks and Deep Learning
CIFAR-10 Classification – Coursework Report

The Goal

- To implement a **specific** model to solve the CIFAR-10 classification problem and classify every image in terms of 1 out of 10 classes.
- To implement the training pipeline to train the model and achieve the highest accuracy possible.

Task 1 - Preparing the Dataset :

The primary step was to prepare the dataset for our use case. This has been achieved by applying a few transformations on the training images:

- transforms.ToTensor() : to convert the images into PyTorch tensors
- transforms.Normalize() : we normalize the tensors by subtracting the mean, which is 0.5 and dividing by the standard deviation that is 0.5 too
- degrees and translate in RandomAffine transformation, which specify the range of rotation and translation to apply randomly to the images during data augmentation.
- scale in RandomAffine transformation, which specifies the range of scaling to apply randomly to the images during data augmentation.
- brightness, contrast, saturation, and hue in ColorJitter transformation, which specify the range of color adjustments to apply randomly to the images during data augmentation.
- mean and std in Normalize transformation, which are used to normalize the pixel values of the images in both training and test datasets.

A definition of hyperparameters have also been provided in the code relating to this section, and the following are assigned or set –
Number of epochs = 35 (num_epochs)
The batch size used in the training DataLoader is 64, and the batch size used in the test DataLoader is 50.
The datasets are then loaded into DataLoader instances, enabling batching, shuffling and parallel processing during the training and testing cycles.

We are also visualizing a batch of images from the CIFAR-10 dataset by randomly selecting 10 images and their corresponding labels from the training dataset using the trainloader. The images are then de-normalized and transposed to be in the format (height, width, channels) for display. Finally, a grid of subplots with 2 rows and 5 columns is created, and each subplot is plotted with an image and its label using a for loop. The classes list contains the class labels for the CIFAR-10 dataset.

Task 2 - Model Creation :

1) The Block class defines a basic building block for the CIFAR10_NN model. Each block contains multiple convolutional layers, where the number of convolutional layers is specified by the parameter k. The block also performs adaptive convolutions by calculating channel-wise weights using a fully connected layer and then applying these weights to the

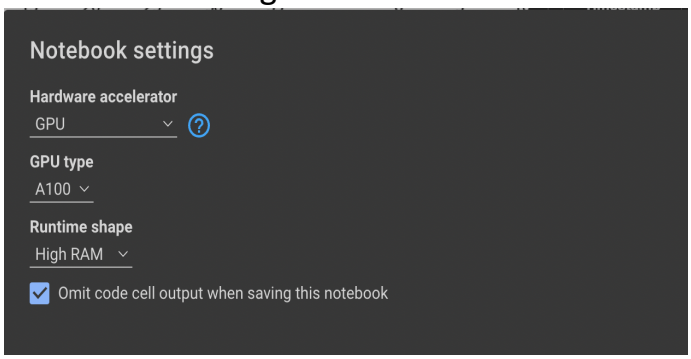
convolutional layers. Additionally, each block has a residual connection that adds the input tensor to the output tensor of the block.

2) CustomModel Class:: PyTorch neural network model class called CIFAR10_NN, which is used for image classification on the CIFAR-10 dataset.

The model architecture consists of a convolutional neural network (CNN) backbone followed by a classifier. The CNN backbone is defined using the nn.Sequential module, which is a container for a sequence of layers. The layers in the backbone include Block layers which contain convolutional, batch normalization, and ReLU activation layers. The number of channels increases as we go deeper in the network, from 64 to 512. The final layer of the backbone is a max pooling layer which reduces the spatial dimensions of the output by a factor of 2.

Task 3 - Defining Loss Function and Optimizer :

1) Enable GPU for training :
   The code enables GPU usage for training the model if it is available, else the execution will take place through a CPU.
   GPU settings



2) Loss Function : CustomLoss(): The CustomLoss class defines a custom loss function with two hyperparameters: alpha and gamma. The forward method of this class takes two arguments: input (the predicted outputs of the model) and target (the ground truth labels).
3) Optimizer : Adam optimizer for the model with a learning rate of 0.001. The optimizer will update the parameters of the model during the training process based on the gradients of the loss function with respect to the parameters
4) LR Scheduler – CosineAnnealing: a learning rate scheduler for the optimizer with the specified hyperparameters. Specifically, the scheduler is the CosineAnnealingLR function from the PyTorch optim.lr_scheduler module, which reduces the learning rate of the optimizer over time based on a cosine annealing schedule.

Task 4 – Script to Train the Model :

The two functions train_epoch and validate_epoch have been defined to perform the training and validation tasks respectively.

- *train_epoch* : function that trains a neural network model for one epoch (one pass through the entire training dataset). It takes as input the model to be trained,

dataloader containing the training dataset, criterion which is the loss function, optimizer which is the optimization algorithm, device which is the device (GPU or CPU) to use for computation, and accumulation_steps which is an integer indicating how many mini-batches of gradients to accumulate before updating the model parameters.

- *validate_epoch* : The validate_epoch function takes as input a pre-trained model, a data loader, a loss criterion, and a device. It evaluates the performance of the model on the validation set by computing the loss and accuracy for the epoch.

Task 5 – Final Model Accuracy :

We Finally visualise and check loss and model accuracy

Epoch [7/35], Train Loss: 0.0017, Train Acc: 0.8036, Val Loss: 0.0954, Val Acc: 0.8264
Epoch [14/35], Train Loss: 0.0007, Train Acc: 0.8680, Val Loss: 0.0535, Val Acc: 0.8660
Epoch [21/35], Train Loss: 0.0003, Train Acc: 0.9049, Val Loss: 0.0338, Val Acc: 0.8922
Epoch [28/35], Train Loss: 0.0001, Train Acc: 0.9286, Val Loss: 0.0316, Val Acc: 0.8989
Epoch [35/35], Train Loss: 0.0001, Train Acc: 0.9490, Val Loss: 0.0251, Val Acc**: 0.9083**
Training finished in 0.00 seconds.

We can see that the Training accuracy goes up while the training loss goes down,

The model accuracy is 90.83%