**Google Drive Setup and Imports(Task1)**
To begin my project setup in Colab, I first integrated Google Drive, where the my_utils.py script was stored. Using the drive.mount() command, I mounted my Google Drive into the Colab environment. Once the drive was mounted, I added the path to the my_utils.py file to Python's system path using sys.path.append(). By appending my custom path, I allowed Colab to treat my_utils.py like a Python module. This setup formed the base environment for the rest of my implementation. After, I imported all the key libraries needed for the implementation.

**Mixup Implementation and Device Setup**
To improve the generalisation and smooth decision boundaries, I have included a mixup.  I have implemented mixup_data to take each input image in a batch and blend it with another randomly selected image from the same batch. I have used the Beta distribution to sample a lambda value that controls how strong the mixing is. The smaller the alpha(in my case 0.1), the more likely the images stay closer to their original form. This function returns four things: the blended images, two sets of labels (y_a and y_b), and the lambda. The reason for keeping both labels is that we will also blend the loss based on the same lambda – which is the second function, mixup_criterion, is doing. It computes a weighted loss using both the original and the shuffled labels. I have applied the mixup only for the first 25 epochs so that when the model is still figuring things out in the early stages, it benefits from this kind of noise. Once it stabilises, I switch it back to the standard input-label training without mixing.
After I set up the device and just check whether CUDA is available, and use GPU. Thus, this function ensures that the project runs on GPU to train faster.

**Data Transformations & Dataset Download and Preprocessing**
To get better generalisation, I have applied data augmentation. I started by normalising the dataset using the CIFAR-10 mean and standard deviation for each of the three RGB channels. This centers the data and scales it, which speeds up convergence and ensures stability during training. After, I have built a transformation pipeline using transform. Compose, and stacked multiple augmentations.
• RandomHorizontalFlip(): this randomly flips the image horizontally, which helps to teach the model that left facing and right facing objects are the same class
• RandomCrop(32, padding=4): This crops a 32x32 region from a padded image, this increased robustness to translations and helped to avoid overfitting to fixed locations in the image
• ColorJitter():  This helps introduce slight brightness, contrast, and saturation variations. I added it to the model to make it less sensitive to lighting changes.
• ToTensor(): This converts the PIL image into a PyTorch tensor.
• Normalize(): This applies the normalisation.
This pipeline makes the dataset more diverse without changing underlying labels.
After, I used tochvision.datasets.CIFAR10 to access CIFAR-10 in PyTorch. I loaded both training and validation datasets here. To pass in the same transform_config to ensure that every image is normalised and preprocessed properly.

**Data Loaders and Meta Info**
Here I have created the data loaders for both training and testing. I set the batch size to 256, which is a good balance between speed and memory usage on the GPU. For the training loader, I have enabled shuffle=True so that the model sees the data in a different order each epoch, which improves generalisation. I have also used num_workers=2 to load data in parallel and reduce bottlenecks during training. For validation, I kept shuffle=False to ensure consistent evaluation, so validation accuracy is measured on a fixed sequence of samples, and makes it easier to compare the performance of epochs.

After, I extracted and printed the class labels from the CIFAR-10 dataset, just to confirm that the classes are correctly loaded. Also, I set the number of CPU threads for PyTorch to 8 using torch.set_num_threads(8) to speed up operations like data loading and preprocessing.

**Stem Block(Task2)**
This part of the model is where the network first interacts with the raw input images. I built a custom StemBlock class that applies a single convolutional layer to the image. Since CIFAR-10 images are RGB, I set the number of input channels to 3. I chose 48 output channels-this gives the model enough width early on to extract a rich set of low-level features, without being too computationally expensive. I used a 3x3 kernel with stride 1 and padding 1 so that the spatial dimensions of the image are preserved after convolution. After, I have applied Batch Normalisation. This helps the model train more stably by normalising the feature maps and accelerating convergence. Finally, I added a ReLU activation, which introduced non-linearity and allowed the model to learn more complex patterns from the start. The rge stem is the first layer of my feature extractor and plays a key role in preparing the input for deeper routing blocks.

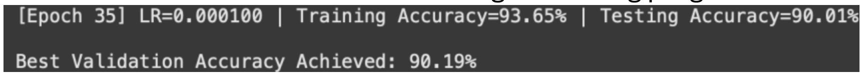
**Backbone(B1 to BN)**
Here, for each block, I have created a multi-expert routing structure. Firstly, I defined two convolutional experts. They are 3x3 convolutions with padding that maintain the input's spatial dimensions. Each expert processes the same input. After, I have implemented the attention mechanism for routing. This starts with a global average pooling layer that squeezes each channel into a single value, summarising spatial information. The pooling features are then passed through a small two layer MLP. The first linear layer compresses the feature size(like a bottleneck), and the second projects it to K outputs – the logits for each expert. Then, I apply a softmax to these logits, turning them into attention weights for each expert path. All expert outputs are stacked, and then I apply the attention weights to linearly combine them. After that, I normalise the result with BatchNorm, add some dropout for regularisation, and apply ReLU activation. This design lets each input dynamically decide which expert(s) to focus on.

**Classifier (C)**
Once the backbone finishes extracting higher-level features, I pass those features through the classifier head to produce the final predictions. First, I use global average pooling to reduce the spatial dimensions of the feature map, into a single value per channel-this flattens the output while still preserving important information. This step reduces the parameter count before classification. After pooling, I apply the dropout with a rate of 0.2 to reduce overfitting-this randomly zeroes out some of the features during training. Finally, I use a single fully connected linear layer to map the feature vector into 10 output logits, one for each CIFAR-10 class. I also used Kaiming(He) initialisation across all Conv2d and Linear layers in the network to make sure training is stable and gradients flow well in deep networks.

**Loss Function, Optimiser and Learning Rate Scheduler(Hyperparameters)(Task3)**
For the loss function, I have gone with CrossEntropyLoss. I added a small amount of label smoothing(0.005) to help prevent the model from becoming too confident in its predictions, which improves generalisation. For the optimiser, I have chosen SGD with momentum. I started with a high learning rate of 0.19, which helps speed up initial learning, and added momentum(0.9) to stabilise the updates and help navigate valleys in the loss landscape more effectively. To avoid overfitting, I also included L2 regularisation via weight decay set to 5e-4. To improve training dynamics further, I used a cosine annealing learning rate scheduler. Over 35 epochs, this gradually reduces the learning rate in a smooth, cosine-shaped curve, down to a

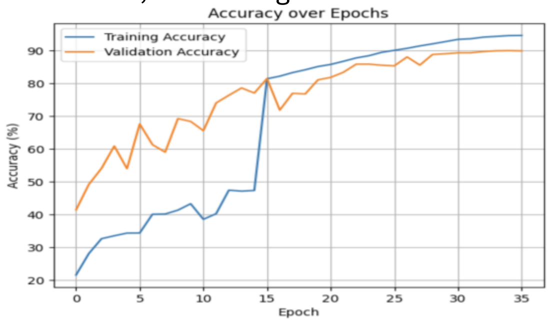
minimum of 1e-4. Thus, at the end of the training, the learning rate would be smaller so that the model can fine-tune weights more carefully to get better final accuracy.

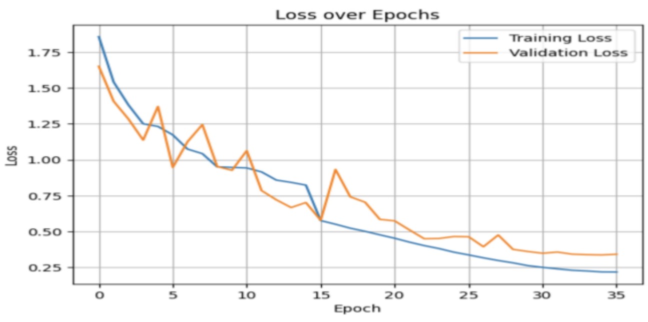
**Training Loop  and Evaluation Visualisations(Task4)**
To train my model, I decided to run it for 36 epochs. Before entering the loop, I set up four lists - one each for tracking the training and validation loss, and training and validation accuracy. These help me later on to visualise how well a model learns over time. I also created a best_val_acc variable so that I could easily save the best model. The loop itself starts by setting the model into training mode using .train(). This ensures layers like dropout act correctly. Then for each batch in the training set, I move the data to the GPU, zero out any old gradients. In the first 25 epochs, I apply the mixup regularisation. This is done by blending images and labels using a lambda sample from a Beta distribution(alpha=0.1). The mixed images are passed into the model, and the loss is computed using a custom criterion that blends the two label losses proportionally. After 25 epochs, I switch off mixup and train on the original data directly, allowing the model to fine-tune the decision boundaries without extra noise. Once the forward and backward passes are done, I update the weights using the optimizer.step() and accumulate the loss and accuracy statistics across all batches. After the training passes, I update the learning rate using the scheduler.step()-this adjusts the LR according to the cosine annealing schedule. I then evaluate the model on the validation set by switching it to evaluation mode using .eval() and disabling gradient tracking with torch.no_grad() to save memory and compute. In the validation loop, I again accumulate loss and accuracy, just like in training. At the end of each epoch, I print out the learning rate and the accuracy for both training and validation, so I can track how well the model is learning as training progresses. So the final accuracy is 90.19%:


```
[Epoch 35] LR=0.000100 | Training Accuracy=93.65% | Testing Accuracy=90.01%

Best Validation Accuracy Achieved: 90.19%
```

**Loss Evolution over Epochs**
To analyse model optimisation, I plotted both training and validation loss across the 36 epochs. From the first epoch, the training loss began at a high value and consistently decreased, showing that the model was learning effectively. The validation loss followed a similar trajectory, although with more fluctuations in early epochs – probably because of the mixup, which injects label noise and causes initial instability. As training progressed beyond epoch 15, the loss curves for both training and validation began to align more closely. The narrowing gap shows the was model generalising better and not overfitting. In the final phase, validation loss stabilised around 0.3, confirming the model's robustness on new data.





**Evolution of Training and Validation Accuracy**
The training accuracy began low due to mixup regularisation but increased steadily as the model adapted. Notably, around epoch 25, there is a visible jump in training accuracy – this aligns with the point where the mixup was disabled , allowing the model to optimise more confidently without noise. From this point, training accuracy grew more sharply, eventually reaching above 93%. Validation accuracy also showed a smooth upward trend, achieving 90.19% at the end of epoch 35. Earlier, validation performance exceeded training due to mixup, but as training stabilised, the gap reduced and was near zero which shows a good generalisation