NEURAL NETWORKS MINI-PROJECT – CIFAR-10 CLASSIFICATION – REPORT

Preparing CIFAR-10 Dataset

In this section of the code, we are preparing the CIFAR-10 dataset for training and testing.
To improve the model's ability to generalize to new images, we apply data augmentation techniques to the training set. (Note: Hyperparameters are highlighted in yellow throughout the report).

The following transformations are applied to the training images:

- Resize: Resize images to 32x32 pixels.
- RandomAffine: Apply random rotation (up to 15 degrees), translation (up to 10%), and scaling (between 90% and 110%).
- ColorJitter: Randomly adjust brightness (±20%), contrast (±20%), saturation (±20%), and hue (±10%).
- RandomHorizontalFlip: Flip images horizontally with a 50% probability.
- ToTensor: Convert images to PyTorch tensors.
- Normalize: Normalize images by subtracting the mean (0.5) and dividing by the standard deviation (0.5) for each color channel.

For the test dataset, only the ToTensor and Normalize transformations are applied. The datasets are then loaded into DataLoader instances for batching, shuffling, and parallel processing during training and testing. For training batch_size=64 and for testing batch_size=50.

Creating the Model

1. **Block class:** Represents a building block for the CIFAR10Model
   - Adaptive average pooling: Reduces the spatial dimensions to 1x1
   - Fully connected (Linear) layer: Predicts channel-wise weights for adaptive convolutions
   - K Convolutional layers: Combined using the calculated weights to produce a single output
   - Residual connection: Allows for identity mapping, aiding in training deeper networks
2. **CIFAR10Model class:** The overall architecture
   - **Backbone:** Consists of a sequence of blocks and other layers, such as BatchNorm and LeakyReLU
     - Batch Normalization: Improves training speed and generalization by normalizing layer outputs
     - LeakyReLU: Non-linear activation function with a small slope for negative values, helping mitigate vanishing gradients
     - MaxPool2d: Reduces spatial dimensions, increasing the receptive field of the network and decreasing computation
   - **Classifier:** Processes the output of the last block
     - AdaptiveAvgPool2d: Computes the spatial average for each channel, reducing dimensions to 1x1
     - Flatten: Converts the 2D output into a 1D vector
     - Linear layer: Produces the final class scores

Definining Loss Function, Optimizer and Learning Rate Scheduler

1. **Device setup: GPU or CPU**
   - Selects the appropriate device (GPU if available, otherwise CPU) for model training
2. **Custom loss function: KLLabelSmoothingCrossEntropyLoss**
   - Combines Cross-Entropy Loss with Kullback-Leibler Divergence Loss
   - Parameters:
     - smoothing: Controls label smoothing (0.2)
     - kl_weight: Weight of KL divergence loss in the combined loss (0.1)
   - Label smoothing helps model generalization by preventing overconfidence in predictions
3. **Optimizer: Adam**
   - Adaptive learning rate optimization algorithm

- Parameters:
  - lr: Learning rate (0.001)
  - weight_decay: L2 regularization (1e-5)
- Regularization helps prevent overfitting by adding a penalty for large weights

4. **Learning rate scheduler: Cosine Annealing**
   - Adjusts the learning rate based on the training progress
   - Parameters:
     - T_max: Maximum number of training iterations (100)
     - eta_min: Minimum learning rate (1e-6)
   - Cosine Annealing helps in finding better local minima by reducing the learning rate over time

Definining Functions for Training and Validation

In this section, we define two functions, train_epoch and validate_epoch, to train and validate the model for one epoch, respectively.

The **train_epoch** function takes the following inputs:

- model: The model to be trained.
- dataloader: A DataLoader that provides training data (images and labels).
- criterion: The loss function to be used during training.
- optimizer: The optimizer responsible for updating the model parameters.
- device: The device (GPU or CPU) to be used for training.
- accumulation_steps (5): The number of mini-batches to accumulate gradients before updating the model parameters .

The function proceeds with the following steps:

1. Set the model to training mode.
2. Initialize variables for running loss, correct predictions, and total samples.
3. Iterate through the training data:
   a. Move the images and labels to the device.
   b. Perform a forward pass through the model.
   c. Calculate the loss using the given criterion.
   d. Perform backpropagation to compute gradients.
   e. Update the model parameters after accumulating gradients for 'accumulation_steps' mini-batches.
   f. Accumulate the loss and update the counters for correct predictions and total samples.
4. Calculate and return the average loss and accuracy for the epoch.

The **validate_epoch** function takes the following inputs:

- model: The model to be validated.
- dataloader: A DataLoader that provides validation data (images and labels).
- criterion: The loss function to be used during validation.
- device: The device (GPU or CPU) to be used for validation.

The function proceeds with the following steps:

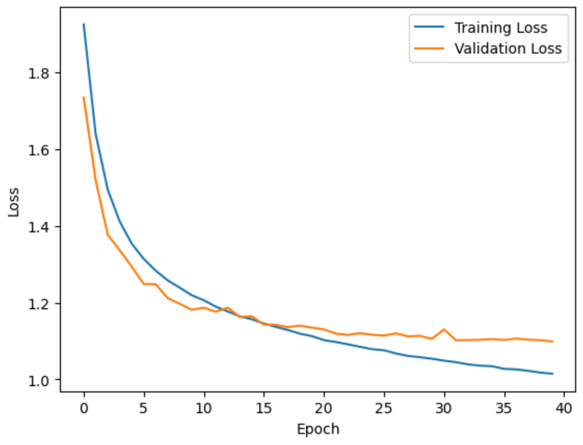
1. Set the model to evaluation mode.
2. Initialize variables for running loss, correct predictions, and total samples.
3. Iterate through the validation data with gradient calculation disabled:
   a. Move the inputs and labels to the device.
   b. Perform a forward pass through the model.
   c. Calculate the loss using the given criterion.
   d. Accumulate the loss and update the counters for correct predictions and total samples.
4. Calculate and return the average loss and accuracy for the epoch.

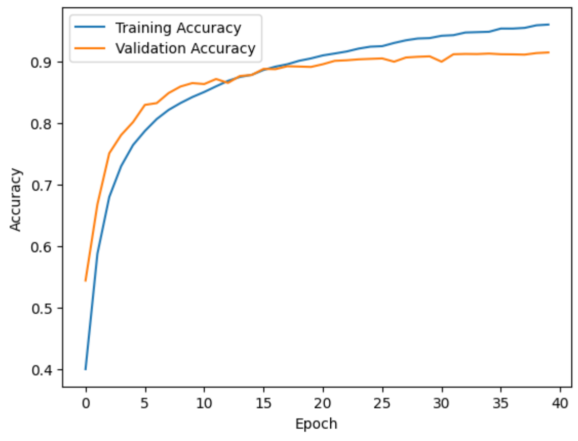
These functions are essential for training and validating the model. By utilizing these functions, we can track the model's performance during the training process and make adjustments as necessary.

Training the Model and Printing Results

In this section, we train and validate the model over multiple epochs and visualize the training progress by plotting loss and accuracy curves.

1. We define the training parameters: num_epochs is set to 40, and print_every is set to 5, meaning the training and validation losses and accuracies will be printed every 5 epochs.
2. We initialize lists to store the loss and accuracy history for both training and validation.
3. We record the start time and loop through each epoch:
   a. Train the model for one epoch using train_epoch function and calculate training loss and accuracy.
   b. Validate the model using validate_epoch function and calculate validation loss and accuracy.
   c. Update the learning rate scheduler at every epoch.
   d. Store the calculated losses and accuracies for the current epoch.
   e. Print the losses and accuracies for the current epoch if it's a multiple of 'print_every'.
4. After training, we record the end time and print the total training time.
5. We plot the loss and accuracy curves for training and validation.





The loss and accuracy curves provide insights into the model's performance throughout the training process. The training loss and accuracy curves show that the model is learning from the data, while the validation loss and accuracy curves indicate how well the model generalizes to new, unseen data.

After training the model for 40 epochs, the model achieved a training accuracy of 95.94% and a validation accuracy of 91.42%. The training loss decreased from 1.3533 to 1.0151, while the validation loss reduced from 1.2937 to 1.0988. The results indicate that the custom architecture and loss function have successfully learned to classify the images in the dataset, and the training process took 3316.40 seconds to complete. The results indicate the model's potential effectiveness on unseen data.

In this project, several techniques are used to enhance the model's performance. The 11-block architecture has adaptive convolutions for better feature extraction. Kaiming initialization, Leaky ReLU activations, and batch normalization improve training efficiency and stability. Residual connections facilitate gradient flow and mitigate degradation in deeper networks. The optimizer, loss function, and gradient accumulation also play significant roles in the model's performance. The Adam optimizer is used for its adaptiveness and efficient convergence. The custom loss function combines Cross-Entropy Loss with Kullback-Leibler Divergence Loss, which balances classification accuracy and knowledge transfer from teacher model to student model. Gradient accumulation over multiple mini-batches helps to stabilize training, reduce memory requirements, and enable the use of larger effective batch sizes for better convergence. Together with the architectural techniques, these components contribute to an efficient and high-performing model on the CIFAR-10 dataset.