Author : Lucas Berry - Student id : 220296948 - Module : ECS6P9U/P/7026P

Assignment : CIFAR-10 Classification

The CIFAR-10 dataset consists of 60,000 32x32 colour images in 10 classes (plane, car, bird, cat, deer, dog, frog, horse, ship, truck) with 6,000 images per class. This project is a complete implementation of a deep learning pipeline for image classification on the CIFAR-10 dataset using PyTorch. The pipeline includes installing required libraries, loading, and pre-processing the dataset, defining a neural network architecture, training the model, and evaluating its performance on the test set.

Part 1. Read dataset and create data loaders

The CIFAR-10 dataset is downloaded and divided into 40,000 training, 10,000 validation, and 10,000 test images. DataLoaders are created for each set with a **'batch_size'** of **64** and a **'num_workers'** equal to the available GPU devices (**1** in Google Colab). Two functions are defined to apply transformations to the training, validation, and testing sets, including random horizontal flips, crops, and conversion to tensors. To further enhance the model, data augmentation techniques such as random cropping, horizontal flipping, affine transformation, and colour jitter are applied to the training set. To evaluate the accuracy, various batch sizes (128, 256) were explored, and a trade-off was made with a batch size of **64**.

Part 2. The Model

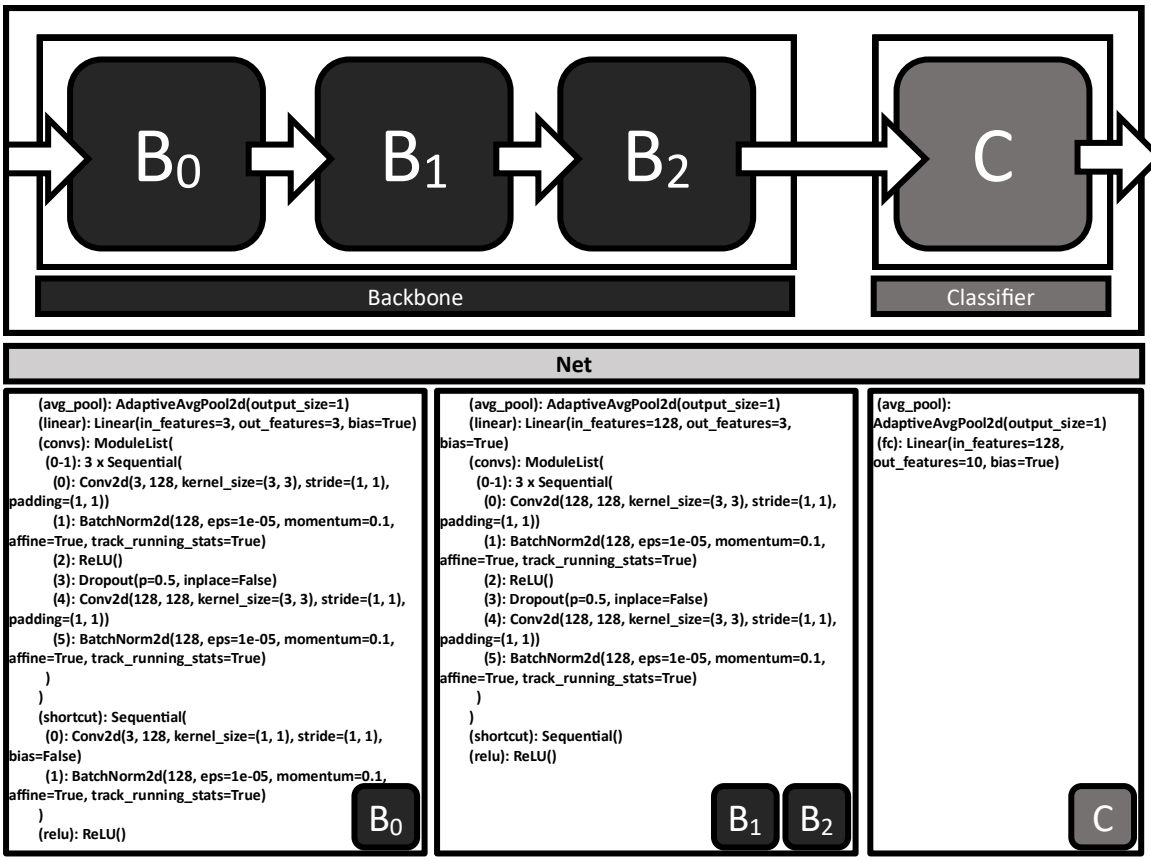

Figure 1 Model Architecture

Block: The Block class is a custom-building block for the architecture. It takes as input '**in_channels**', '**out_channels**', '**K**', and an optional '**dropout_rate**'. Within the block, it uses an adaptive averaging layer, a linear layer, and **'K'** parallel convolutional layers (with batch normalisation, ReLU activation, elimination, and another convolutional layer). The outputs of these parallel convolutional layers are combined using a weighted sum, the weights being determined by a softmax function applied to the output of the linear layer. The block also includes a shortcut connection in case the '**in_channels**' and '**out_channels**' are different.

Backbone: The Backbone class creates a sequence of '**N'** custom blocks, each with **'K'** parallel convolutional layers. It takes as inputs **'in_channels'**, **'out_channels'**, **'K'** and **'N'**. The backbone is responsible for extracting features from the input data.

Classifier: The Classifier class is a simple linear classifier that takes the features extracted by the backbone and assigns them to the desired number of classes. It consists of an adaptive averaging layer followed by a fully connected (linear) layer.

Net: The Net class combines the '**backbone'** and the **'classifier'** to create the final architecture. It takes **'K'**, **'N'** and the **'channels'** as inputs, and an optional '**num_classes'** (10 by default). The network first applies the backbone to the input data and then feeds the output into the
classifier to obtain the final output.

After conducting several tests, it has been observed that increasing the number of blocks **'N'**, the number of parallel convolutional layers **'K'**, and the number of channels does not lead to significant improvements in the model's performance. In this current configuration, the network has **K = 3** parallel convolutional layers in each block with **N = 3** blocks in the backbone, and **channels = 128** channels for each convolutional layer.

Part 3. Loss, Optimizer and Scheduler

Loss function: Cross-entropy loss is used as the criterion, which is suitable for multi-class classification problems. It measures the difference between the predicted probabilities and the true class labels.

Optimiser: The Adam optimiser is employed with a learning rate of **0.01** and no weight decay. Adam is an adaptive learning rate optimisation algorithm. It adjusts the learning rate for each parameter during training, resulting in faster convergence and improved performance.

Learning rate scheduler: The ReduceLROnPlateau scheduler is utilised to adapt the learning rate when the validation loss plateaus. It monitors the validation loss and reduces the learning rate by a factor of **0.1** if the loss does not improve for **5** consecutive epochs. This approach allows the model to escape local minima and converge to a better solution.

Part 4. Training and Evaluation

For the training and evaluation part, the project code incorporated early stopping to prevent overfitting. The model is trained for a maximum of **150** epochs, with **early stopping** implemented with a patience of **10** epochs. If the validation loss does not improve for 10 consecutive epochs, the training process is terminated early. (Occurred in Epoch **82/150**)
Lists are initialised to store training and validation loss and accuracy values for later analysis. For each epoch, the model is set to training mode and iterates through batches of the training data. The inputs and labels are moved to the target device GPU, and the model's weights are updated using backpropagation with the Adam optimizer.

After training, the model is set to evaluation mode and iterates through batches of the validation data. The inputs and labels are moved to the target device GPU, and the validation loss and accuracy are computed. The learning rate is updated using the **ReduceLROnPlateau** scheduler based on the validation loss.
The validation loss is monitored, and if it does not improve for a specified number of consecutive epochs (**patience = 10**), the training process is terminated early. This training and validation loop allows the model to learn from the training data while monitoring its performance on the validation set, helping to adjust the learning rate and prevent overfitting using early stopping.

Below, you can find a table showcasing various hyperparameter tuning combinations to achieve the best accuracy while considering the computational time. This table illustrates the impact of adjusting hyperparameters.

Epoch	Learning Rate	Batch_size	K	N	Channels	EarlyS Patience	Train Accuracy	Test Accuracy	Time
65	0.001	256	2	2	64	10	78.56%	77.26%	58m
57	0.01	256	2	4	128	10	92.71%	87.89%	2h25
65	0.01	128	2	3	128	10	91.24%	86.27%	1h35
82	0.01	64	3	3	128	10	96.78%	88.48%	3h08

Table 1 Train and test accuracy based on different hyperparameters.

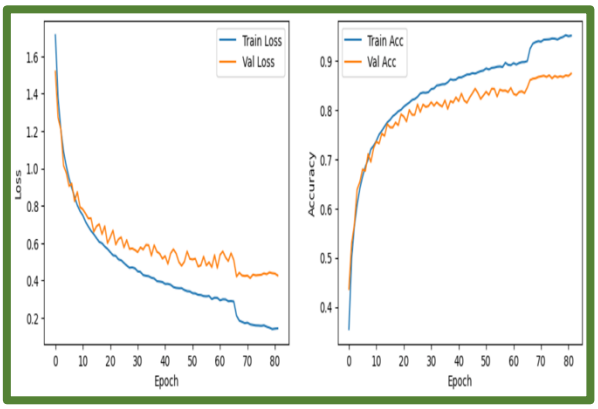

Figure 2 Evolution of loss and accuracy for train and validation test for the green line

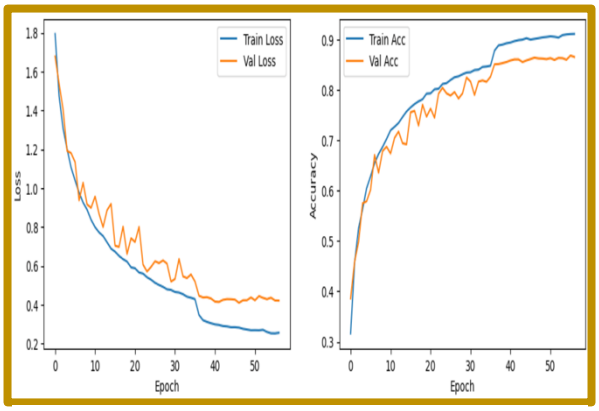

Figure 3 Evolution of loss and accuracy for train and validation test for the yellow line

Part 5. Testing

The accuracy of the model can be impacted by various hyperparameters, such as the learning rate **'lr'**, the number of **'K'** convolutional layers in the network, number of **'N'** blocks, **'batch_size', 'Channels'** and more in depth early stopping **'Patience'**. By fine-tuning these hyperparameters, it is possible to improve the model's performance on the test dataset. In our case, after exploring multiple combinations of hyperparameters, we have achieved a maximum testing accuracy of **88.48%** and a training accuracy of **96.78%**.
Each hyperparameter contributes to enhancing the model's accuracy, with the adaptive learning rate and the inclusion of a scheduler being particularly crucial for overcoming performance plateaus over time. Moreover, the channels can be considered a hyperparameter because their quantity directly influences the network's capacity to learn and extract features from the input data which may impact its performance on the given task and early stopping is particularly beneficial for preventing overfitting by halting the training process when the model's performance on the validation set no longer improves.