CIFAR-10 CLASSIFICATION

Shivam Tyagi †
School of Electronic Engineering and Computer Science
Queen Mary University of London Mile End
Road, London E1 4NS, UK
s.tyagi@se22.qmul.ac.uk

Abstract

This report presents a neural network model designed using a given backbone architecture, along with additional features, to classify the CIFAR-10 dataset. The objective of the model is to accurately classify images into 10 different classes, utilizing the inherent structure of the backbone architecture and incorporating novel techniques. The network was trained on the CIFAR-10 training set and evaluated on a separate test set to assess its performance. The report describes the modified network architecture, the training process, and the evaluation results.

1 Read Dataset and Create Data loaders

- To load the CIFAR-10 dataset, the torchvision.datasets.CIFAR10 class provided by PyTorch was used. The dataset was downloaded and saved to a local directory using the root argument. The train parameter was set to True or False to indicate whether to load the training or test set. The download parameter was set to True to download the dataset if it has not already been downloaded. A transform object was passed in to apply data augmentation to the images.
- The transform object is a sequence of image transformations that are applied to each image in the dataset. A transforms.Compose method was used to define a set of transformations that includes random horizontal flipping, random cropping, color jittering, conversion to PyTorch tensors, and normalization of pixel values.
- After defining the transform object, data loaders were created for the training and test sets using the torch.utils.data.DataLoader class. The shuffle parameter was set to True for the training set to shuffle the order of the images during each epoch. The batch_size parameter was set to 150, which specifies the number of images to include in each batch during training or testing. Finally, the num_workers parameter was set to 2 to specify the number of subprocesses to use for data loading.

2 Create the Model

Backbone:

1. **ConvBlock:**

- The ConvBlock module takes three arguments, in_channels, out_channels, k_size, i.e., the number of convolution layers and the number of outputs from linear layer which is set to 5
- The first purpose of this block is to pass the input tensor X to an AdaptiveAvgPool2d((1, 1)) layer as per the given model, it is chosen so as to reduce the spatial dimensions of the input tensor to (1,1). This can be helpful because it reduces the number of parameters in the fully connected layer, which in turn can reduce the risk of overfitting the model to the training data.
- After that the result of the spatial average is fed into a linear classifier using the batch size and the number of channels and weight using xavier_normal are applied to the Linear layer generating k_szie number of outputs. By using this weight initialization method, we can avoid the problem of vanishing or exploding gradients during training.
- After this the module is used to create k_size convolution layers using same input, output channels and kernel size=3 and padding =1 thus preserving the spatial dimensions of the input feature map. xavier_normal is again used for weights along with the RelU function for Non-linear activation which is required after a convolution layer. These layers are appended together using nn.ModuleList().
- The ouput from the linear layer is passed through a softmax converting them into probabilities, the reason being we would be using the loss function as cross-entropy, and the combination of both is preferred in neural networks.

- These k_size outputs are then multiplied with corresponding layers in ModuleList() and the sum of the product is stored as sum.
- The residual connection is then used, it takes input as X, applies weights using weights from xavier_normal and ReLU activation function followed by batch normalization, and is only active when the ‘res’ flag is set to True. It is then added to the sum which is the output of the whole block. This helps to ensure that the information from the input is not lost as it passes through the convolutional layers of the network.

2. CIFAR10

- Till now the basic ConvBlock was created now the main backbone is created using CIFAR10 which works on the initial tensor X and the outputs of each block from the basic block in the following way.
- 10 blocks have been created with an increasing number of channels from 3 to 1024 with every other block using the residual connection and all of them using Batch Normalization to make the training of deep neural networks more stable and faster and LeakyRelU to help alleviate the "dying ReLU" problem.
- nn.Sequential() is used to give the output of each block as the input to the next block.

Classifier

- The classifier takes input as the output of the last layer and then computes the spatial average.
- The result from the spatial average is then fed into an MLP after flattening it with 1024 input channels 10 output channels and 512 hidden inputs.

3 Loss and Optimizer

Loss: Cross Entropy

- The cross-entropy loss function is used to measure how well the predicted class probabilities match the true class labels for each image in the dataset. The loss is then backpropagated through the model to update the weights and biases. It goes well in conjunction with softmax output which we have already used before.

Optimizer: Adam

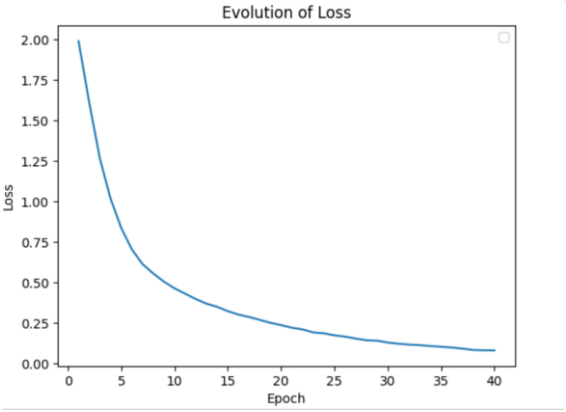
- The Adam optimizer is used to update the weights and biases of the model during training. It uses an adaptive learning rate that adjusts based on the magnitude of the gradients, making it well-suited for training deep neural networks. The lr argument is set to the desired learning rate for the optimizer and weight decay

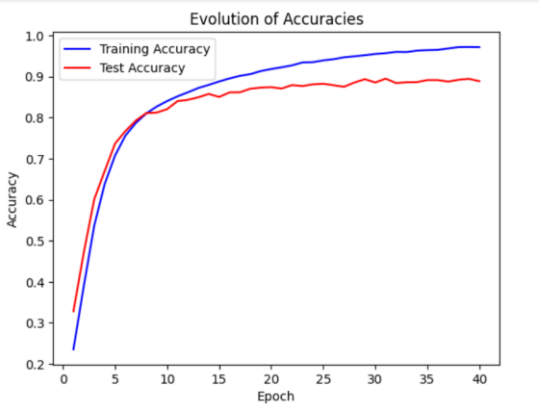
4 Training details and curves

- The model was trained based on labs from week 5-8 which is based on the following:
- The accuracy function calculates the number of correct predictions for a batch of data.
    1. y_hat: A matrix containing predicted scores for each class.
    2. y: A tensor containing true labels for each example.
- The Accumulator class is a helper class that allows us to accumulate the sum of multiple variables.
    1. n: The number of variables to accumulate sums over.
- The evaluate_accuracy function calculates the accuracy of the model on the test dataset.
    1. net: The neural network model to evaluate.
    2. data_iter: An iterator over the dataset.
    3. device: The device on which to perform computations (e.g. 'cpu' or 'cuda').

NOTE: GPU for Google Colab is chosen for quick results which are slow on CPU

- The train_epoch_ch3 function trains the model for one epoch (one pass through the entire training dataset). It computes the gradients of the loss function with respect to the model parameters, updates the parameters using an optimizer, and computes the training loss and accuracy for that epoch on the training datset.
    1. net: The neural network model to train.
    2. train_iter: An iterator over the training dataset.
    3. loss: The loss function to optimize.
    4. optimizer: The optimization algorithm to use.
    5. device: The device on which to perform computations (e.g. 'cpu' or 'cuda').
    6. batch_size: The size of each training batch.
    7. num_outputs: The number of output classes.
- The train_ch3 function trains the model for multiple epochs (specified by the num_epochs parameter). For each epoch, it calls the train_epoch_ch3 function to train the model and evaluate_accuracy function to compute the accuracy on the test dataset. It also keeps track of the training loss and accuracy and the test accuracy for each epoch.
    1. net: The neural network model to train.
    2. train_iter: An iterator over the training dataset.
    3. test_iter: An iterator over the test dataset.
    4. loss: The loss function to optimize.
    5. num_epochs: The number of training epochs.
    6. optimizer: The optimization algorithm to use.
    7. device: The device on which to perform computations (e.g. 'cpu' or 'cuda').
- The curve for the evolution of loss and accuracies are as follows:





The Hyper parameters used are:

- batch_szie: to determine the number of samples processed in each forward/backward pass during training, it is set to 150.
- in_channels: number of channels in the input tensor
- out_channels: number of channels in the output tensor
- k_size: number of convolution layers to be added in the ModuleList
- res: boolean flag to indicate if an extra convolution layer is needed for residual connection.
- learning_rate: The learning rate used by the optimizer for updating the model parameters. In this case, it is set to 0.001.
- num_epochs: the number of times the entire dataset is passed through the neural network during training, it is set to 40.

5. Model Accuracy

After training the dataset with the given backbone model and adding residual features the final results after 40 epochs are as follows:

Training Accuracy: 97.138 %

Training Loss: 0.081

Test Accuracy: 88.85%