ECS659U/ECS659P – NNDL Assignment - 2022/23

220266251 | Riya Dodthi

Introduction:

CIFAR-10 is a widely used dataset for image classification tasks that consists of 50,000 32x32 colour images in 10 classes. The classes include common objects such as airplanes, cars, birds, cats, deer, dogs, frogs, horses, ships, and trucks. The task is to train a neural network to correctly classify these images into their respective classes. In this assignment, we will build a neural network model on training dataset and evaluate it on test set.

Basic Model Architecture:

We will be building a model which has N blocks, and each basic block consists of –

1. A linear/MLP layer predicting a vector a = $[a_1,\ldots,a_k]$ where a = g(SpatialAvgPool(X)W), where 'g' is a non-linear function.
2. K convolution layers which are combined using a to produce a single output:
   $O = a_1 * Conv_1(X) + \ldots.. + a_k * Conv_k(X)$

Classifier:

The output from the N^{th} blocks is then passed through **f** = SpatialAvgPool(O_N). Next, f is passed to a Linear/MLP layer.

Load Data -

We use **torchvision.datasets.CIFAR10() function in load_data_cifar10()** to load train and test dataset. **Data augmentation** techniques like rotation, resizedCrop, HorizontalFlip, ColorJitter, etc. are also used in this method. The dataset has 50,000 images, 3 channels each and each image of dimensions 32x32. For training purpose, we take a batch size of 180.

Model:

In this model a single block consists of –

1. An **AvgPool2d()** layer with kernel size = 4, stride = 2 and padding = 2 which takes an input of shape X = [180, 3, 32, 32]. We then **flatten** this output and pass it through an **MLP layer of 3 Linear layers** each having a sigmoid activation function after it. The output from the last linear layer produces a vector a of 4 elements which is then passed through a non-linear **Sigmoid** activation function.
2. Next, we have **4 convolution layers** which takes the input X and where the first 3 layers have a kernel size of 3, stride=1, padding=1 and dilation=1 and the last convolution layer has a kernel size 4 and padding = 2 with the same stride and dilation.
   After every convolution layer we have a **LeakyRelu() activation** function followed by **BatchNorm2d()** which performs batch normalization that helps to improve the performance and stability of neural networks by normalizing the activations of each layer which is lastly followed by **MaxPool()** layer of kernel = 2 and stride = 2.
3. Finally, we reshape each element from vector 'a' to multiply them with the outputs from the 4 convolution layers and produce a single output O.

Backbone:

1. This class consists of **4 blocks** where the first block takes X as input and rest of the blocks take input from their previous block.
2. Each block is followed by a **LeakyRelu()** activation function.
3. Finally, the output from the N^{th} block is passed through a **AvgPool2d()** layer with kernel size = 4, stride = 2, padding =1 whose output is then flattened and passed through an **MLP of 3 Linear layers each** having 512, 256 and 10 outputs respectively. Here too each layer is followed by a **Sigmoid activation** function.
4. Finally, we initialize weights and create an object for our model with 3 input channels and 10 no. of outputs

Loss function: CrossEntropyLoss()

The CrossEntropyLoss() function measures the difference between the predicted probability distribution and the actual probability distribution. In PyTorch, nn.CrossEntropyLoss() combines the SoftMax function and the negative log-likelihood loss.

Optimizer: SGD()

SGD optimizer works by computing the gradient of the loss function with respect to the weights and biases, and then adjusting the parameters in the opposite direction of the gradient to minimize the loss. We set the learning rate parameter to 0.1 with a momentum of 0.9

Training:

To train the model, we pass our model, training dataset, test dataset, loss function, epochs, optimizer and device to the train_model() method. In train_model() for each epoch we calculate the training loss and accuracy and test accuracy.

Train metrics:

Train metrics (loss + acc) are calculated using train_epoch_ch3 function which takes 5 arguments – net (model), train dataset, test dataset, loss function, updater (optimizer), device GPU/CPU on which to perform computation.

1. The function starts by setting the network to training mode using the train() method of the PyTorch nn.Module class if the network is an instance of that class. An Accumulator object-metric is created to accumulate the sum of training loss, sum of training accuracy, and number of examples.
2. For each X, y (feature and label) in train_iter X and y are moved to device. We get the predicted label y_hat using net(X) which takes input as X feature set. The loss between y_hat and y is calculated using the loss function.
3. If the updater is an instance of the torch.optim.Optimizer class, the gradients are computed and the network's parameters are updated using the optimizer's zero_grad(), backward(), and step() methods. The training loss, training accuracy, and number of examples are accumulated using the Accumulator object. Else, the gradients are computed, and the network's parameters are updated using the backward() method and the updater function. The training loss, training accuracy, and number of examples are accumulated using the Accumulator object.
4. At the end of the epoch, the average training loss and training accuracy are computed by dividing the sum of the training loss and accuracy by the total number of examples. The function returns the average training loss and training accuracy.

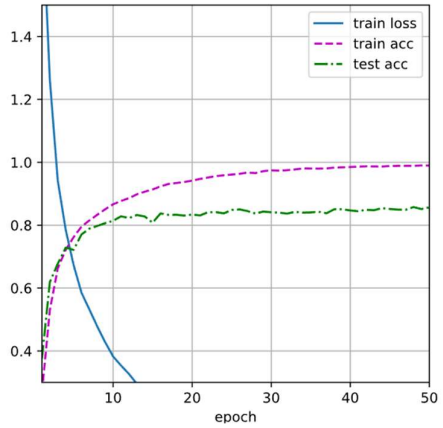
Test accuracy:

The function evaluate_accuracy_gpu() takes model, test dataset and/or device as arguments.

1. It starts by setting the model to evaluation mode using the eval() method. Then if the device is not mentioned, the function sets the device variable to the device (CPU or GPU) where the model parameters are currently located. The accumulator object accumulates the number of correct predictions and the number of examples.
2. Next, for each X, y in test set, we calculate the output of the model which is computed using the net function. The accuracy of the network's prediction for the current batch is computed using the accuracy function which takes 3 arguments – predicted label, true label, and device. If the device is mentioned, move y_hat and y to the specified device using to() method.
3. Next, if y_hat is a matrix, the argmax function returns the index with the highest value in each row. A Boolean variable is created to check if the predicted output matches the true label. Finally, the function returns the total number of correctly predicted labels.
4. Next, the number of correct predictions and number of examples are accumulated using the Accumulator object.
5. At the end of the evaluation, the function returns the average accuracy of the network over the entire evaluation dataset. Finally, animator function plots the results.

Results:

```
Train metric:  (0.02870798313478008, 0.98974)
Test metric:  0.8568
```



Train loss:
The train loss curve is observed to follow a steep downward curve and before epoch 10 it crosses 0.5.

Train accuracy:
The training accuracy curve initially follows a steep upward trend, followed by a slowed growth after reaching approx. 85-90% accuracy.

Test accuracy:
The test accuracy curve follows a similar steep upward trend, followed by much slowed growth after 80-84%.

Final Metrics:
Train loss – 0.028
Train accuracy – 98.9%
Test accuracy – 85.68%