IE 534: Deep Learning, Fall 2018

Homework 3: Deep Convolution Neural Network on CIFAR10

Lecturer: Justin Sirignano *Submitted by: Rachneet Kaur, Sept. 28th, 2018*

Implementation

The objective is to **classify** the input 32×32 dimensional coloured image into the corresponding one of the 10 mutually exclusive classes namely {airplane, automobile, bird, cat, deer, dog, frog, horse, ship, truck}. The CIFAR10 data set has 50000 coloured training images and 10000 images in the test set of dimensions $3 \times 32 \times 32$.

Data Loading

To load the data set, I used *torchvision.datasets.CIFAR10* and *torch.utils.data.DataLoader* functions in Pytorch.

Hence, we have:

- Dimensions of training set input X = (50000, 3, 32, 32) and labels Y = (50000, 1)
- In my code, I store training set in $data_train$.
- Dimensions of test set input X = (10000, 3, 32, 32) and labels Y = (10000, 1)
- In my code, I store testing set in $data_test$.
- I use batch size=100 both while loading the training and the testing set.
- I shuffle the training data randomly while loading whereas I do not shuffle the data while loading it in testing phase.
- While loading the training data, I perform data augmentation step (discusses further afterwards) and normalize the data.
- While loading the testing data, no data augmentation is performed but I normalize the data.
- Since this is a $k = 10$ class classification problem, I have 10 units in the output layer.

Model Architecture

I implemented a class $CIFAR10Model$ to define the architecture. The input image is $3 \times 32 \times 32$. My architecture has 8 convolution layers and 3 fully connected layers with dropout, batch normalization and Max pooling. The detailed architecture is as follows:

- Convolution layer with input as 3 channels and output as 64 channels, with Kernel = 4×4, Stride = 1 and Padding = 2.
- ReLU activation
- Batch Normalization with 64 channels
- Convolution layer with input as 3 channels and output as 64 channels, with Kernel = 4×4, Stride = 1 and Padding = 2.
- ReLU activation
- Max pooling with stride = 2 and kernel size = 2
- Dropout with probability p = 0.5
- Convolution layer with input as 3 channels and output as 64 channels, with Kernel = 4×4, Stride = 1 and Padding = 2.
- ReLU activation
- Batch Normalization with 64 channels
- Convolution layer with input as 3 channels and output as 64 channels, with Kernel = 4×4, Stride = 1 and Padding = 2.
- ReLU activation
- Max pooling with stride = 2 and kernel size = 2
- Dropout with probability p = 0.5
- Convolution layer with input as 3 channels and output as 64 channels, with Kernel = 4×4, Stride = 1 and Padding = 2.
- ReLU activation
- Batch Normalization with 64 channels
- Convolution layer with input as 3 channels and output as 64 channels, with Kernel = 3×3, Stride = 1 and Padding = 0.
- ReLU activation

- Dropout with probability p = 0.5
- Convolution layer with input as 3 channels and output as 64 channels, with Kernel = 3×3, Stride = 1 and Padding = 0.
- ReLU activation
- Batch Normalization with 64
- Convolution layer with input as 3 channels and output as 64 channels, with Kernel = 3×3, Stride = 1 and Padding = 0.
- ReLU activation
- Batch Normalization with 64
- Dropout with probability p = 0.5
- Flattening the convolution output
- Fully connected layer with 32*32 input and 500 output features.
- ReLU activation
- Fully connected layer with 500 input and 500 output features.
- ReLU activation
- Fully connected layer with 500 input and 10 output features.

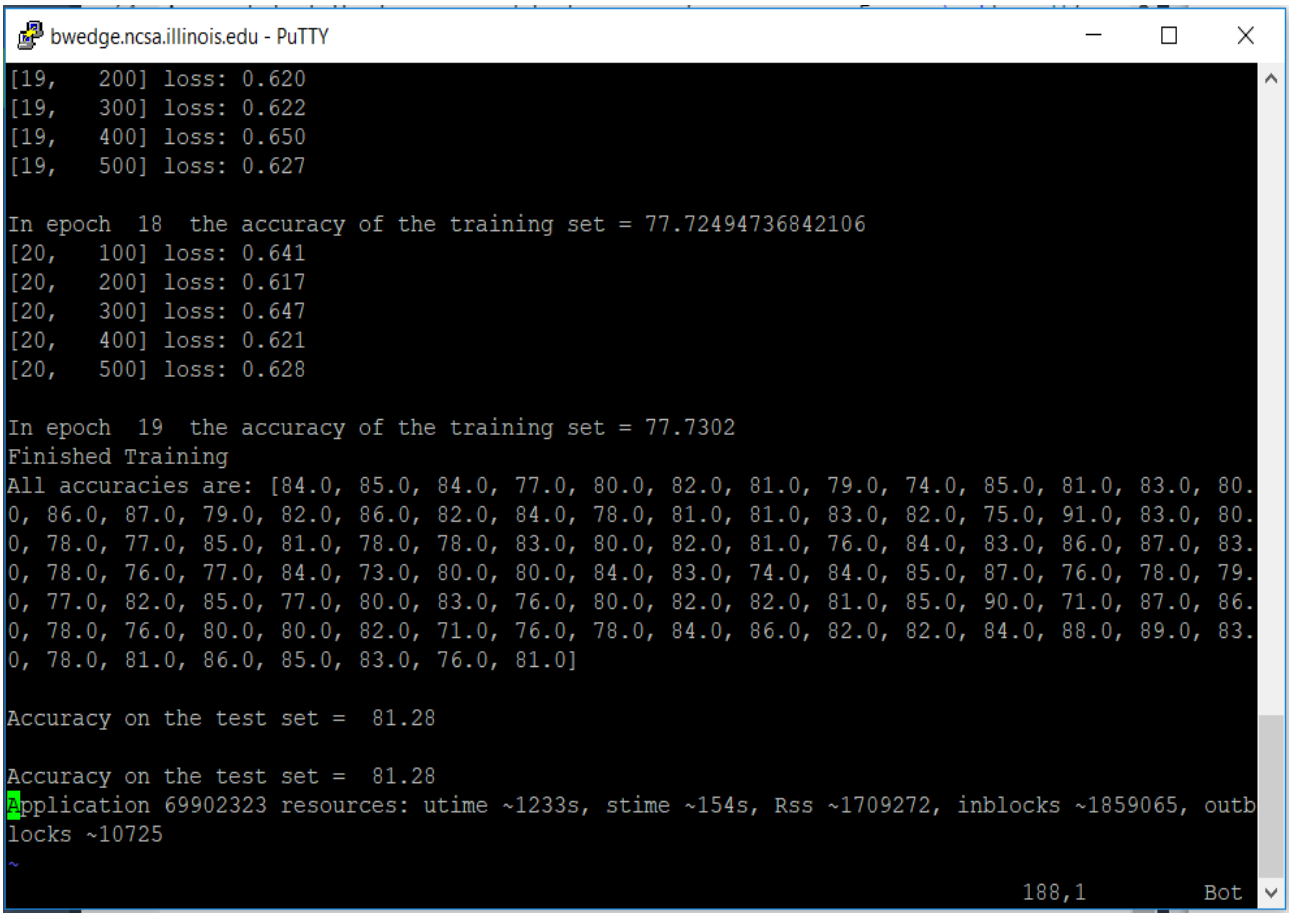
Note: No. of output units in a convolution layer = $\frac{\text{No. of input units - Filter Size + 2(Padding)}}{Stride} + 1$
Next, we calculate the predicted label as argmax$(f(x, \theta))$ for the training sample which is compared with true label y to compute the accuracy of the training set.

- For implementation in Python, I used the ReLU activation function as $\sigma(z) = max(0, z)$.
- For implementation, I trained the deep CNN model for 150 epochs.
- I used the batch size = 100.

Data Augmentation and Optimizer

I implemented the following data augmentation techniques in the training set while loading:

- Randomly flip the image horizontally with a probability of 0.5



```
[19,   200] loss: 0.620
[19,   300] loss: 0.622
[19,   400] loss: 0.650
[19,   500] loss: 0.627

In epoch  18  the accuracy of the training set = 77.72494736842106
[20,   100] loss: 0.641
[20,   200] loss: 0.617
[20,   300] loss: 0.647
[20,   400] loss: 0.621
[20,   500] loss: 0.628

In epoch  19  the accuracy of the training set = 77.7302
Finished Training
All accuracies are: [84.0, 85.0, 84.0, 77.0, 80.0, 82.0, 81.0, 79.0, 74.0, 85.0, 81.0, 83.0, 80.
0, 86.0, 87.0, 79.0, 82.0, 86.0, 82.0, 84.0, 78.0, 81.0, 81.0, 83.0, 82.0, 75.0, 91.0, 83.0, 80.
0, 78.0, 77.0, 85.0, 81.0, 78.0, 78.0, 83.0, 80.0, 82.0, 81.0, 76.0, 84.0, 83.0, 86.0, 87.0, 83.
0, 78.0, 76.0, 77.0, 84.0, 73.0, 80.0, 80.0, 84.0, 83.0, 74.0, 84.0, 85.0, 87.0, 76.0, 78.0, 79.
0, 77.0, 82.0, 85.0, 77.0, 80.0, 83.0, 76.0, 80.0, 82.0, 82.0, 81.0, 85.0, 90.0, 71.0, 87.0, 86.
0, 78.0, 76.0, 80.0, 80.0, 82.0, 71.0, 76.0, 78.0, 84.0, 86.0, 82.0, 82.0, 84.0, 88.0, 89.0, 83.
0, 78.0, 81.0, 86.0, 85.0, 83.0, 76.0, 81.0]

Accuracy on the test set =  81.28

Accuracy on the test set =  81.28
Application 69902323 resources: utime ~1233s, stime ~154s, Rss ~1709272, inblocks ~1859065, outb
locks ~10725
~
                                                                              188,1         Bot
```

Figure 1: Results

- Randomly brighten the image with a factor selected uniformly from [max(0, 1 - brightness), 1 + brightness].
- Randomly flip the image vertically with a probability of 0.5
- Normalization (Done both while training and testing as well)

I used the ADAM optimizer, implemented by *torch.optim.Adam(model.parameters(), lr=LR)* to minimize the cross entropy loss function, implemented by *nn.CrossEntropyLoss()*.

- For implementation, I used ADAM optimizer to minimize the cross entropy loss function with learning rate $\alpha = 0.001$.

A snapshot of the training and test accuracy's for dropout with the heuristic method at the testing time, is given in Figure 1.
Note that the accuracy on the training set after 150 epochs reaches approx. 0.777, but

clearly since the test accuracy is 0.812, we can notice that the model did not **overfit** the training set. It is because of the regularization of the parameters using dropout introduced in the model architecture.

- For implementation of heuristic, I used a Boolean variable *heuristic* = *True* when heuristic is implemented. In this case, I use *model.train()* at the test time to calculate $H^{(l)} = p\sigma(Z^{(l)})$ for layer l of the architecture, where $p = 0.5$ is the probability of the dropout.
- For implementation of the Monte Carlo method, I set Boolean variable *heuristic* = *False*. In this case, I do not use *model.train()* at the test time. Rather, I calculate $H^{(l)} = \sigma(Z^{(l)})$ for layer l of the architecture using a dropout selected randomly for $MonteCarlo = 100$ times, where $p = 0.5$ is the probability of the dropout. I average the probabilities of these $MonteCarlo$ trials and then make the prediction based on the averaged values of the softmax function.

Extra Credit Implementation

To compare the test accuracy's with Heuristic and Monte Carlo approach, I implement the following:

- Set a Boolean variable *heuristic*, such that *heuristic* = *True* when heuristic $H^{(l)} = p\sigma(Z^{(l)})$ need to be applied and *heuristic* = *False* when Monte Carlo method need to be applied.
- In heuristic, I used *model.train()* to use the weight scaling by $\frac{1}{1-p}$ at dropout of the units at the training time to implement use $H^{(l)} = (1-p)\sigma(Z^{(l)})$.
- In Monte Carlo method, I do not use *model.train()* at the test time. Rather, I calculate $H^{(l)} = \sigma(Z^{(l)})$ for layer l of the architecture using a dropout selected randomly for $MonteCarlo = 100$ times, where $p = 0.5$ is the probability of the dropout. I average the probabilities of these $MonteCarlo$ trials and then make the prediction based on the averaged values of the softmax function.

Results

Case 1: The results for the model when we use heuristic for dropout at the test time is as follows:

- The accuracy on the test set when dropout with heuristic is used = 81.28 %

- The estimated time to train the model for 150 epochs is about 2.5 hours.

In this case, I predict the label based on each element in test set and check if the true label is same as the predicted label to calculate the accuracy.

Case 2: The results for the model when we use Monte Carlo method for dropout at the test time is as follows:

- The accuracy on the test set when dropout with heuristic is used = 83.56 %
- The estimated time to train the model for 150 epochs is about 3.5 hours.

I predict the label based on average of results from 100 dropout architectures and check if the true label is same as the predicted label to calculate the accuracy.