Assignment Four Analysis (Deep Convolution Neural Network by PyTorch)

by Haoyu Wang (haoyuw2)

Analysis:

By implementing the Deep Convolution neural network on PyTorch, I produce 82.32% accuracy model when testing the model by CIFAR-10 and compare the result from Monte Carlo Method with Heuristics Method. To implement the model, I follow the steps proposed in the course note. I created a deepCNN object by overriding the pyTorch class and utilizing DataLoader to download CIFAR-10 from Internet. After declaring the objects in the model and defining the forward function in the deepCNN, I train the model by utilizing the model by batch size of 500. I utilize two different ways to compare the accuracy of the between using Monte Carlo Method and Heuristics Method and I realize that Monte Carlo Method usually will produce a lower accuracy on the early stage since it computes more data in a test data set. However, eventually Monte Carlo method converges to the accuracy produce by Heuristics Method

Code:

Attached in the next page.

Assignment_Deep_Convolution_Network_Pytorch

March 4, 2019

In [ ]:
```
import torch
import torchvision
import torch.nn.functional as F
import torchvision.transforms as transforms
from torch.autograd import Variable
import time

# Set up the test picture and input data to a Tensor
# Data augmentation
# Declare learning rate and epochs
Learning_rate = 0.001
epochs = 50
batch_size = 500

transform_train = transforms.Compose([
    transforms.RandomHorizontalFlip(0.2),
    transforms.RandomVerticalFlip(0.2),
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])

transform_test = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
])

# Get the training set
# CIFAR-10 is 32 by 32 numpy array as int
train_set = torchvision.datasets.CIFAR10(root='./cifardata', train=True, download=True,
# Get the test set
test_set = torchvision.datasets.CIFAR10(root='./cifardata', train=False, download=True,

# load the data from the CIFAR_10 with batch size and shuffle the data to make the model
# set num_worker to be 2 to promote the efficiency of loading data
train_loader = torch.utils.data.DataLoader(dataset=train_set, batch_size=batch_size, num
test_loader = torch.utils.data.DataLoader(dataset=test_set, batch_size=batch_size, num_w
```

```python
class deepCNN(torch.nn.Module):
    def __init__(self):
        super(deepCNN, self).__init__()
        # Declare convolution network torch.nn.Conv2d(in_channels, out_channels, kernel_
        self.cnn1 = torch.nn.Conv2d(3, 64, kernel_size=4, stride=1, padding=2)
        self.cnn2 = torch.nn.Conv2d(64, 64, kernel_size=4, stride=1, padding=2)
        self.cnn3 = torch.nn.Conv2d(64, 64, kernel_size=4, stride=1, padding=2)
        self.cnn4 = torch.nn.Conv2d(64, 64, kernel_size=4, stride=1, padding=2)
        self.cnn5 = torch.nn.Conv2d(64, 64, kernel_size=4, stride=1, padding=2)
        self.cnn6 = torch.nn.Conv2d(64, 64, kernel_size=3, stride=1, padding=0)
        self.cnn7 = torch.nn.Conv2d(64, 64, kernel_size=3, stride=1, padding=0)
        self.cnn8 = torch.nn.Conv2d(64, 64, kernel_size=3, stride=1, padding=0)

        # Declare Batch normalization with respect to 64 channels
        self.BN1 = torch.nn.BatchNorm2d(64)
        self.BN2 = torch.nn.BatchNorm2d(64)
        self.BN3 = torch.nn.BatchNorm2d(64)
        self.BN4 = torch.nn.BatchNorm2d(64)
        self.BN5 = torch.nn.BatchNorm2d(64)

        # Declare Pooling layer(Kernel size = 2, stride = 2)
        self.pool = torch.nn.MaxPool2d(kernel_size=2, stride=2)

        # Declare the drop out layer
        self.drop1 = torch.nn.Dropout2d(p=0.3)
        self.drop2 = torch.nn.Dropout2d(p=0.3)
        self.drop3 = torch.nn.Dropout2d(p=0.3)
        self.drop4 = torch.nn.Dropout2d(p=0.3)
        self.drop5 = torch.nn.Dropout2d(p=0.3)
        self.drop6 = torch.nn.Dropout2d(p=0.3)

        # Declare the linear layer
        self.linear1 = torch.nn.Linear(4 * 4 * 64, 500)
        self.linear2 = torch.nn.Linear(500, 500)
        self.linear3 = torch.nn.Linear(500, 10)

    def forward(self, input):
        # Follow the class note to finish the forward steps
        input = self.cnn1(input)
        input = self.BN1(F.relu(input))
        input = self.pool((F.relu(self.cnn2(input))))
        input = self.drop1(input)

        input = self.BN2(F.relu(self.cnn3(input)))
        input = self.pool(F.relu(self.cnn4(input)))
        input = self.drop2(input)
```

```
        input = self.BN3(F.relu(self.cnn5(input)))
        input = self.drop3(F.relu(self.cnn6(input)))

        input = self.BN4(F.relu(self.cnn7(input)))

        input = self.BN5(F.relu(self.cnn8(input)))

        input = self.drop4(input)
        # No sure. Pooling twice to form a 8 by 8 matrix
        input = input.view(-1, 4 * 4 * 64)
        input = self.drop5(F.relu(self.linear1(input)))
        input = self.drop6(F.relu(self.linear2(input)))
        input = F.relu(self.linear3(input))
        return input


# Create a object of deep CNN model
model = deepCNN()

# Move model to GPU
model.cuda()

# Declare Adam and loss function
criterion = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.RMSprop(model.parameters(), lr=Learning_rate)

# Load the loader and i in train loaders
############################################ Training Loop ##############################
for epoch in range(epochs):
    # Switch to training model
    time1 = time.time()
    model.train()
    for i, (images, classes) in enumerate(train_loader):
        # Put variable to GPU
        images = Variable(images.cuda())
        input = Variable(classes.cuda())

        # Get the result from model
        outputs = model(images)

        # Calculate the loss
        loss = criterion(outputs, classes)

        # Clear, Backpropagation and step
        optimizer.zero_grad()
        B_P = loss.backward()
        optimizer.step()
```

```python
    # No influence on the gradient, test code(set to eval model)
############################################ Heuristics ################################
    model.eval()
    with torch.no_grad():
        sum = 0.0
        counter = 0
        # Batch size number of images and labels
        for i, (images, classes) in enumerate(test_loader):
            # Put variable to GPU
            images = Variable(images.cuda())
            labels = Variable(classes.cuda())

            outputs = model(images)
            # Get the prediction of the model
            _, prediction = torch.max(outputs.data, 1)
            correctNum = float((prediction == labels).sum().item())
            counter+=1
            sum += (correctNum / float(batch_size))*100
    time2 = time.time()
    time_elapsed = time2 - time1
    print(time_elapsed, sum/float(counter))
############################################ Monte Carlo ###############################
model.train()
sum = 0
counter = 0
print("Monte Carlo")

for i, (images, classes) in enumerate(test_loader):
    p = 0.0
    images = Variable(images.cuda())
    labels = Variable(classes.cuda())
    # Should be different output eachtime
    for n in range(10):
        output = model(images)
        _, prediction = torch.max(output.data, 1)
        correctNum = float((prediction == labels).sum().item())
        p+=(correctNum / float(batch_size))*100
    p/=10.0
    counter+=1
    sum+=p
time_elapsed = time.time()-time2
print(time_elapsed, sum / float(counter))
```