Assignment Five Analysis (Residual Neural Network)

by Haoyu Wang (haoyuw2)

Analysis:

For Assignment Five, I implemented a residual neural network in pyTorch by the HW description on the websites and Piazza for the part one and fine tune the pre-train resNet by online pre-train model, arriving 84.19% with 181 epochs (500 batch size) for the part one and 71.84% accuracy with 18 epochs (100 batch sizes) for the part two. What worth noting is that I created two classes for part one called basicBlock1 and basicBlock2, that basicBlock1 has the one by one convolution. In the resNet class overriding the pyTorch basic block, I called those two classes and reduced the unnecessary codes. Also, for the part two, for the SGD function I utilized a different way of adding the Tensor into the optimizer compared with the code on Piazza. I created dictionaries with key 'params' and pushed the 'layer4' and 'fc' into the list of dictionaries.

Code:

Part One:

```
import numpy as np
import torch
import torchvision
import time
import random
import torchvision.transforms as transforms
from torch.autograd import Variable
```

```python
import torch.nn.functional as F



transform1 = transforms.Compose([transforms.RandomVerticalFlip(p = 0.1),
                                transforms.RandomHorizontalFlip(p = 0.1),
                                 transforms.ToTensor(),
                                transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])
transform2 = transforms.Compose([transforms.ToTensor(),
                                 transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])

train_set = torchvision.datasets.CIFAR100(root='./cifardata', train=True, download=True, transform=transform1)
test_set = torchvision.datasets.CIFAR100(root='./cifardata', train=False, download=True, transform=transform2)

# Block defined in the note with one by one convolution
class basicBlockA(torch.nn.Module):
    def __init__(self, inChannel):
        super(basicBlockA, self).__init__()
        # First with input channel 3 and output channel 32, padding =1, stride =1
        self.inChannel = inChannel
        self.outChannel = 2*inChannel

        # Correct the convolution to the right shape
        self.conCor = torch.nn.Conv2d(self.inChannel, self.outChannel, kernel_size=1, stride=2)
        # Only the first convolution within basic block 1 should have a stride of 2.
        self.conv1 = torch.nn.Conv2d(self.inChannel, self.outChannel, kernel_size=3, stride=2, padding=1)
        # Rest of conv are normal convolution
        self.conv2 = torch.nn.Conv2d(self.outChannel, self.outChannel, kernel_size=3, stride=1, padding=1)

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

    def forward(self, x):
        # Remember the current residual
        residual = x
        # Correct dimension
        residual = self.conCor(residual)
```

```python
        x = self.conv1(x)
        x = F.relu(self.BN1(x))
        x = self.conv2(x)
        x = self.BN2(x)
        x+=residual
        x = F.relu(x)
        return x

# Block defined in the note with one by one convolution
class basicBlockB(torch.nn.Module):
    def __init__(self, inChannel):
        super(basicBlockB, self).__init__()
        # First with input channel 3 and output channel 32, padding =1, stride =1
        self.inChannel = inChannel
        self.outChannel = 2*inChannel

        # Normal convolution
        self.conv1 = torch.nn.Conv2d(self.inChannel, self.inChannel, kernel_size=3, stride=1, padding=1)
        # Rest of conv are normal convolution
        self.conv2 = torch.nn.Conv2d(self.inChannel, self.inChannel, kernel_size=3, stride=1, padding=1)

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

    def forward(self, x):
        # Remember the current residual
        residual = x
        x = self.conv1(x)
        x = F.relu(self.BN1(x))
        x = self.conv2(x)
        x = self.BN2(x)
        x+=residual
        x = F.relu(x)

        return x
```

```
# Residual deep Network
class ResNet(torch.nn.Module):
  def __init__(self):
    super(ResNet, self).__init__()
    # First with input channel 3 and output channel 32, padding =1, stride =1
    self.conv1 = torch.nn.Conv2d(3, 32, kernel_size=3, stride=1, padding=1)

    # Second Block with 2 basic block
    self.block2_1 = basicBlockB(32)
    self.block2_2 = basicBlockB(32)

    # Third Block with 4 basic block
    self.block3_1 = basicBlockA(32)
    self.block3_2 = basicBlockB(64)
    self.block3_3 = basicBlockB(64)
    self.block3_4 = basicBlockB(64)

    # Fourth Block with 4 basic blocks
    self.block4_1 = basicBlockA(64)
    self.block4_2 = basicBlockB(128)
    self.block4_3 = basicBlockB(128)
    self.block4_4 = basicBlockB(128)

    # Fifth Block with 2 basic blocks
    self.block5_1 = basicBlockA(128)
    self.block5_2 = basicBlockB(256)

    # Drop out
    self.drop = torch.nn.Dropout(p=0.5)

    # Max pooling
    self.pool = torch.nn.MaxPool2d(kernel_size=4, stride=4)

    # Batch Normalization
```

```python
        self.BN = torch.nn.BatchNorm2d(32)

        # fully connect network
        self.linear = torch.nn.Linear(256, 100)

    def forward(self, x):
        x = F.relu(self.BN(self.conv1(x)))
        x = self.drop(x)

        x = self.block2_1(x)
        x = self.block2_2(x)

        x = self.block3_1(x)
        x = self.block3_2(x)
        x = self.block3_3(x)
        x = self.block3_4(x)

        x = self.block4_1(x)
        x = self.block4_2(x)
        x = self.block4_3(x)
        x = self.block4_4(x)

        x = self.block5_1(x)
        x = self.block5_2(x)

        x = self.pool(x)
        x = x.view(-1, 256)
        x = self.linear(x)
        return x


model = ResNet()
model.cuda()

LR = 0.001
batch_size = 500
Num_Epochs = 300
```

```python
train_loader = torch.utils.data.DataLoader(train_set, batch_size=batch_size, num_workers=2)
test_loader = torch.utils.data.DataLoader(test_set, batch_size=batch_size, num_workers=2)

criterion = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.RMSprop(model.parameters(), lr = LR)

for epoch in range(Num_Epochs):
    time1 = time.time()
    model.train()
    for i, (images, classes) in enumerate(train_loader):
        data, target = Variable(images.cuda()), Variable(classes.cuda())
        optimizer.zero_grad()
        output = model(data)
        loss = criterion(output, target)
        loss.backward()
        optimizer.step()

    model.eval()
    # Test Loss
    counter = 0
    test_accuracy_sum = 0.0
    for i, (images, classes) in enumerate(test_loader):
        data, target = Variable(images.cuda()), Variable(classes.cuda())
        output = model(data)
        prediction = output.data.max(1)[1]
        accuracy = (float(prediction.eq(target.data).sum())/float(batch_size))*100.0
        counter += 1
        test_accuracy_sum = test_accuracy_sum + accuracy
    test_accuracy_ave = test_accuracy_sum/float(counter)
    time2 = time.time()
    time_elapsed = time2 - time1
    print(epoch, test_accuracy_ave, time_elapsed)


# SAVE THE MODEL
model.save_state_dict('mytraining.pt')
```

Part Two:

```python
import torch
import torchvision.transforms as transforms
import torchvision
import torch.nn as nn
import time
import torchvision.models as models
from torch.autograd import Variable

# Load the dataset from CIFAR100 and reshape it with respect to the ImageNet
DIM = 224
transform1 = transforms.Compose([
                    transforms.RandomResizedCrop(DIM, scale=(0.7, 1.0), ratio=(1.0, 1.0)),
                    transforms.RandomVerticalFlip(p=0.1),
                    transforms.RandomHorizontalFlip(p=0.1),
                    transforms.ToTensor(),
                    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
                    ])

transform2 = transforms.Compose([
                    transforms.Resize(DIM, interpolation=2),
                    transforms.ToTensor(),
                    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
                   ])

train_set = torchvision.datasets.CIFAR100(root='./cifardata', train=True, download=True, transform=transform1)
test_set = torchvision.datasets.CIFAR100(root='./cifardata', train=False, download=True, transform=transform2)

# Import the model and load the new linear fully connected network because CIFAR
model = models.resnet18(pretrained=True)
model.fc = nn.Linear(512,100)

# Define the hyper-parameters
```

```
LR = 0.001
batch_size = 100
Num_Epochs = 300

train_loader = torch.utils.data.DataLoader(train_set, batch_size=batch_size, num_workers=4)
test_loader = torch.utils.data.DataLoader(test_set, batch_size=batch_size, num_workers=4)

# Define the optimizer
params = []
params_dict = dict(model.named_parameters())
for key, value in params_dict.items():
    print(key)
    if key[:6] == 'layer4':
        params += [{'params':value,'lr':0.01}]
    if key[:2] == 'fc':
        params += [{'params':value,'lr':0.01}]

print(params)
optimizer = torch.optim.SGD (params,  momentum=0.9)
criterion = torch.nn.CrossEntropyLoss()
mode = model.cuda()
# Start training
for epoch in range(Num_Epochs):
    model.train()
    time1 = time.time()
    for i, (images, classes) in enumerate(train_loader):
        images, classes = Variable(images.cuda()), Variable(classes.cuda())
        optimizer.zero_grad()
        with torch.no_grad():
            h = model.conv1(images)
            h = model.bn1(h)
            h = model.relu(h)
            h = model.maxpool(h)
            h = model.layer1(h)
            h = model.layer2(h)
            h = model.layer3(h)
        h = model.layer4(h)
```

```
        h = model.avgpool(h)
        h = h.view(h.size(0), -1)
        output = model.fc(h)
        loss = criterion(output, classes)
        loss.backward()
        optimizer.step()

    model.eval()
    # Test Loss
    counter = 0
    test_accuracy_sum = 0.0
    for i, (images, classes) in enumerate(test_loader):
        images, classes = Variable(images.cuda()), Variable(classes.cuda())
        output = model(images)
        prediction = output.data.max(1)[1]
        accuracy = (float(prediction.eq(classes.data).sum()) / float(batch_size)) * 100.0
        counter += 1
        test_accuracy_sum = test_accuracy_sum + accuracy
    test_accuracy_ave = test_accuracy_sum / float(counter)
    time2 = time.time()
    time_elapsed = time2 - time1
    print(epoch, test_accuracy_ave, time_elapsed)


# SAVE THE MODEL
model.save_state_dict('mytraining.pt')
```

Result:

```
164 82.49999999999999 101.1401002407074
165 82.96 101.20789551734924
166 84.3 101.09595799446106
167 84.01000000000002 101.0277030467987
168 84.01 101.16171455383301
169 83.79000000000002 101.07029461860657
170 83.94999999999997 101.10381841659546
171 84.10999999999999 101.06384897232056
172 83.6 101.07611131668091
173 83.85 101.11192727088928
174 83.41 101.12769389152527
175 83.77 101.07614660263062
176 83.44999999999999 101.15591669082642
177 83.28999999999999 101.09584307670593
178 83.22999999999999 101.07601189613342
179 84.09 101.15588927268982
180 84.19 101.11190700531006
181 84.19000000000003 101.13590598106384
182 83.38000000000001 101.11993360519409
```

```
0 64.39 288.16154074668884
1 67.9 293.78218245506287
2 69.7 292.3714451789856
3 70.57 286.6362202167511
4 70.42 291.9415557384491
5 70.6 286.6786334514618
6 71.18 295.96221137046814
7 71.38 292.04258823394775
8 71.44 287.13219118118286
9 71.84 292.61571860313416
10 71.83 293.79099225997925
11 71.41 290.36543917655945
12 71.48 292.1228847503662
13 71.09 294.3558773994446
14 71.65 294.7798852920532
15 71.9 292.9270660877228
16 72.31 291.3274133205414
17 72.77 291.92469930648804
18 71.84 291.99394059181213
```