hw7_code_output

April 29, 2024

[ ]:
```
import sys
import time

import numpy as np
import matplotlib
import matplotlib.pyplot as plt
import torch

%matplotlib inline
print(f"Python version: {sys.version}\nNumpy version: {np.
 ↪__version__}\nMatplotlib version: {matplotlib.__version__}\nPyTorch version:␣
 ↪{torch.__version__}")
```

```
Python version: 3.11.8 | packaged by conda-forge | (main, Feb 16 2024, 20:49:36)
[Clang 16.0.6 ]
Numpy version: 1.26.4
Matplotlib version: 3.8.0
PyTorch version: 2.2.1
```

[ ]:
```
CIFAR_DATA_PATH = "/Users/lucah/Library/CloudStorage/OneDrive-DurhamUniversity/
 ↪Course Material & Work/SNU Year Abroad {SNU}/2-Spring Semester/Mathematical␣
 ↪Foundations of Deep Neural Networks {MFDNN}/Lectures Slides {MFDNN}/
 ↪Notebooks {MFDNN}/cifar_data"
```

1 Problem 2

Originally,

[ ]:
```
import torch
import torch.nn as nn


class AlexNet(nn.Module):
    def __init__(self, num_classes: int = 1000) :
        super(AlexNet, self).__init__()
        self.features = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=11, stride=4),
            nn.ReLU(inplace=True),
```

```
            nn.MaxPool2d(kernel_size=3, stride=2),
            nn.Conv2d(64, 192, kernel_size=5, padding=2),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=3, stride=2),
            nn.Conv2d(192, 384, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(384, 256, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(256, 256, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=3, stride=2),
        )
        self.avgpool = nn.AdaptiveAvgPool2d((6, 6))
        self.classifier = nn.Sequential(
            nn.Linear(256 * 6 * 6, 4096),
            nn.ReLU(inplace=True),
            nn.Linear(4096, 4096),
            nn.ReLU(inplace=True),
            nn.Linear(4096, num_classes),
        )

    def forward(self, x: torch.Tensor) :
        x = self.features(x)
        x = self.avgpool(x)
        x = torch.flatten(x, 1)
        x = self.classifier(x)
        return x



model = AlexNet()
```

Let's dissect it!

[ ]:
```
class AlexNetDissect(nn.Module):
    def __init__(self, num_classes: int = 1000) :
        super(AlexNetDissect, self).__init__()
        self.features1 = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=11, stride=4),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=3, stride=2),
        )
        self.features2 = nn.Sequential(
            nn.Conv2d(64, 192, kernel_size=5, padding=2),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=3, stride=2),
        )
```

```
        self.features3 = nn.Sequential(
            nn.Conv2d(192, 384, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
        )
        self.features4 = nn.Sequential(
            nn.Conv2d(384, 256, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
        )
        self.features5 = nn.Sequential(
            nn.Conv2d(256, 256, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=3, stride=2),
        )
        self.avgpool = nn.AdaptiveAvgPool2d((6, 6))
        self.classifier1 = nn.Sequential(
            nn.Linear(256 * 6 * 6, 4096),
            nn.ReLU(inplace=True),
        )
        self.classifier2 = nn.Sequential(
            nn.Linear(4096, 4096),
            nn.ReLU(inplace=True),
        )
        self.classifier3 = nn.Sequential(
            nn.Linear(4096, num_classes),
        )

    def forward(self, x: torch.Tensor) :
        print('input', x.shape)
        x = self.features1(x)
        print('after f1', x.shape)
        x = self.features2(x)
        print('after f2', x.shape)
        x = self.features3(x)
        print('after f3', x.shape)
        x = self.features4(x)
        print('after f4', x.shape)
        x = self.features5(x)
        print('after f5', x.shape)
        x = self.avgpool(x)
        print('after avg', x.shape)
        x = torch.flatten(x, 1)
        x = self.classifier1(x)
        print('after c1', x.shape)
        x = self.classifier2(x)
        print('after c2', x.shape)
        x = self.classifier3(x)
        print('after c3', x.shape)
```

```
        return x


modeld = AlexNetDissect()
```

[ ]:
```
_ = modeld(torch.rand((1,3,227,227)))
```

```
input torch.Size([1, 3, 227, 227])
after f1 torch.Size([1, 64, 27, 27])
after f2 torch.Size([1, 192, 13, 13])
after f3 torch.Size([1, 384, 13, 13])
after f4 torch.Size([1, 256, 13, 13])
after f5 torch.Size([1, 256, 6, 6])
after avg torch.Size([1, 256, 6, 6])
after c1 torch.Size([1, 4096])
after c2 torch.Size([1, 4096])
after c3 torch.Size([1, 1000])
```

2 Problem 3

2.1 Instantiate model (given)

[ ]:
```
import torch.nn as nn
from torch.utils.data import DataLoader
import torch
import torchvision
import torchvision.transforms as transforms


# Instantiate model with BN and load trained parameters
class smallNetTrain(nn.Module) :
    # CIFAR-10 data is 32*32 images with 3 RGB channels
    def __init__(self, input_dim=3*32*32) :
        super().__init__()

        self.conv1 = nn.Sequential(
            nn.Conv2d(3, 16, kernel_size=3, padding=1),
            nn.BatchNorm2d(16),
            nn.ReLU()
        )
        self.conv2 = nn.Sequential(
            nn.Conv2d(16, 16, kernel_size=3, padding=1),
            nn.BatchNorm2d(16),
            nn.ReLU()
        )
        self.fc1 = nn.Sequential(
            nn.Linear(16*32*32, 32*32),
```

```
            nn.BatchNorm1d(32*32),
            nn.ReLU()
        )
        self.fc2 = nn.Sequential(
            nn.Linear(32*32, 10),
            nn.ReLU()
        )
    def forward(self, x) :
        x = self.conv1(x)
        x = self.conv2(x)
        x = x.float().view(-1, 16*32*32)
        x = self.fc1(x)
        x = self.fc2(x)

        return x

model = smallNetTrain()
model.load_state_dict(torch.load("./smallNetSaved",map_location=torch.
 ↪device('cpu')))
```

[ ]: <All keys matched successfully>

2.2 Instantiate model without BN (given)

[ ]:
```
# Instantiate model without BN
class smallNetTest(nn.Module) :
    # CIFAR-10 data is 32*32 images with 3 RGB channels
    def __init__(self, input_dim=3*32*32) :
        super().__init__()

        self.conv1 = nn.Sequential(
            nn.Conv2d(3, 16, kernel_size=3, padding=1),
            nn.ReLU()
        )
        self.conv2 = nn.Sequential(
            nn.Conv2d(16, 16, kernel_size=3, padding=1),
            nn.ReLU()
        )
        self.fc1 = nn.Sequential(
            nn.Linear(16*32*32, 32*32),
            nn.ReLU()
        )
        self.fc2 = nn.Sequential(
            nn.Linear(32*32, 10),
            nn.ReLU()
        )
    def forward(self, x) :
```

```
        x = self.conv1(x)
        x = self.conv2(x)
        x = x.float().view(-1, 16*32*32)
        x = self.fc1(x)
        x = self.fc2(x)

        return x

model_test = smallNetTest()
```

2.3 Weight tuning

[ ]:
```
# Initialize weights of model without BN
conv1_bn_beta, conv1_bn_gamma = model.conv1[1].bias, model.conv1[1].weight
conv1_bn_mean, conv1_bn_var = model.conv1[1].running_mean, model.conv1[1].
 ↪running_var

conv2_bn_beta, conv2_bn_gamma = model.conv2[1].bias, model.conv2[1].weight
conv2_bn_mean, conv2_bn_var = model.conv2[1].running_mean, model.conv2[1].
 ↪running_var

fc1_bn_beta, fc1_bn_gamma = model.fc1[1].bias, model.fc1[1].weight
fc1_bn_mean, fc1_bn_var = model.fc1[1].running_mean, model.fc1[1].running_var

eps = 1e-05


# Initialize the following parameters
trained_w = model.conv1[0].weight
trained_b = model.conv1[0].bias
std = torch.sqrt(conv1_bn_var+eps)
model_test.conv1[0].weight.data = trained_w * conv1_bn_gamma.view(-1,1,1,1)/std.
 ↪view(-1,1,1,1)
model_test.conv1[0].bias.data = (trained_b - conv1_bn_mean) * conv1_bn_gamma/
 ↪std + conv1_bn_beta

trained_w = model.conv2[0].weight
trained_b = model.conv2[0].bias
std = torch.sqrt(conv2_bn_var+eps)
model_test.conv2[0].weight.data = trained_w * conv2_bn_gamma.view(-1,1,1,1)/std.
 ↪view(-1,1,1,1)
model_test.conv2[0].bias.data = (trained_b - conv2_bn_mean) * conv2_bn_gamma/
 ↪std + conv2_bn_beta

trained_w = model.fc1[0].weight
trained_b = model.fc1[0].bias
```

```
std = torch.sqrt(fc1_bn_var+eps)
model_test.fc1[0].weight.data = trained_w * fc1_bn_gamma.view(-1,1)/std.
 ↪view(-1,1)
model_test.fc1[0].bias.data = (trained_b - fc1_bn_mean) * fc1_bn_gamma/std +␣
 ↪fc1_bn_beta

trained_w = model.fc2[0].weight
trained_b = model.fc2[0].bias
model_test.fc2[0].weight.data = trained_w
model_test.fc2[0].bias.data = trained_b

# Verify difference between model and model_test
model.eval()
# model_test.eval()  # not necessary since model_test has no BN or dropout


test_dataset = torchvision.datasets.CIFAR10(root=CIFAR_DATA_PATH,
                                            train=False,
                                            transform=transforms.ToTensor(),␣
 ↪download = True)
test_loader = torch.utils.data.DataLoader(dataset=test_dataset, batch_size=100,␣
 ↪shuffle=False)

diff = []
with torch.no_grad():
    for images, _ in test_loader:
        diff.append(torch.norm(model(images) - model_test(images))**2)

print(max(diff)) # If less than 1e-08, you got the right answer.
```

```
Files already downloaded and verified
tensor(1.1354e-07)
```

3 Problem 5

3.1 Net1 (given)

[ ]:
```
import torch.nn as nn
import torch
import torchvision


class Net1(nn.Module):
    def __init__(self, num_classes=10):
        super(Net1, self).__init__()
        self.features = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=7, stride=1),
```

```
            nn.ReLU(),
            nn.Conv2d(64, 192, kernel_size=3, stride=1),
            nn.ReLU(),
            nn.Conv2d(192, 384, kernel_size=3, stride=1),
            nn.ReLU(),
            nn.Conv2d(384, 256, kernel_size=3, stride=1),
            nn.ReLU(),
            nn.Conv2d(256, 256, kernel_size=3, stride=1),
        )
        self.classifier = nn.Sequential(
            nn.Linear(256 * 18 * 18, 4096),
            nn.ReLU(),
            nn.Linear(4096, 4096),
            nn.ReLU(),
            nn.Linear(4096, num_classes)
        )

    def forward(self, x):
        x = self.features(x)
        x = torch.flatten(x, 1)
        x = self.classifier(x)
        return x
```

3.2 (a) Net2 and compare

[ ]:
```
class Net2(nn.Module):
    def __init__(self, num_classes=10):
        super(Net2, self).__init__()
        self.features = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=7, stride=1),
            nn.ReLU(),
            nn.Conv2d(64, 192, kernel_size=3, stride=1),
            nn.ReLU(),
            nn.Conv2d(192, 384, kernel_size=3, stride=1),
            nn.ReLU(),
            nn.Conv2d(384, 256, kernel_size=3, stride=1),
            nn.ReLU(),
            nn.Conv2d(256, 256, kernel_size=3, stride=1),  # output 18x18 (for␣
 ↪part (a))
        )
        # filled in
        self.classifier = nn.Sequential(
            nn.Conv2d(256, 4096, kernel_size=18, stride=1),  # output 1x1 (for␣
 ↪part (a))
            nn.ReLU(),
            nn.Conv2d(4096, 4096, kernel_size=1, stride=1),
            nn.ReLU(),
```

```
            nn.Conv2d(4096, num_classes, kernel_size=1, stride=1)  # so that␣
↪output is Bxnum_classesx1x1 (1 dimension removed by squeeze)
        )

    def copy_weights_from(self, net1):
        with torch.no_grad():
            for i in range(0, len(self.features), 2):
                self.features[i].weight.copy_(net1.features[i].weight)
                self.features[i].bias.copy_(net1.features[i].bias)

            for i in range(0, len(self.classifier), 2):  # skip ReLU layers
                # filled in
                linear_map = torch.clone(net1.classifier[i].weight.data)
                k1, k2 = self.classifier[i].kernel_size
                self.classifier[i].weight.data = torch.stack([row.reshape(-1,␣
↪k1, k2) for row in linear_map])
                self.classifier[i].bias.data.copy_(net1.classifier[i].bias)

    def forward(self, x):
        x = self.features(x)
        x = self.classifier(x)
        return x


model1 = Net1() # model1 randomly initialized
model2 = Net2()
model2.copy_weights_from(model1)

test_dataset = torchvision.datasets.CIFAR10(
    root=CIFAR_DATA_PATH,
    train=False,
    transform=torchvision.transforms.ToTensor()
)

test_loader = torch.utils.data.DataLoader(
    dataset=test_dataset,
    batch_size=10
)

imgs, _ = next(iter(test_loader))
diff = torch.mean((model1(imgs) - model2(imgs).squeeze()) ** 2)
print(f"Average Pixel Difference: {diff.item()}") # should be small
```

```
Average Pixel Difference: 9.708393194663781e-17
```

3.3 (b)

[ ]:
```
test_dataset = torchvision.datasets.CIFAR10(
    root=CIFAR_DATA_PATH,
    train=False,
    transform=torchvision.transforms.Compose([
        torchvision.transforms.Resize((36, 38)),
        torchvision.transforms.ToTensor()
    ]),
    download=True
)

test_loader = torch.utils.data.DataLoader(
    dataset=test_dataset,
    batch_size=10,
    shuffle=False
)

images, _ = next(iter(test_loader))
b, w, h = images.shape[0], images.shape[-1], images.shape[-2]
out1 = torch.empty((b, 10, h - 31, w - 31))
for i in range(h - 31):
    for j in range(w - 31):
        # filled in
        out1[:, :, i, j] = model1(images[:,:,i:i+32,j:j+32])
out2 = model2(images)
diff = torch.mean((out1 - out2) ** 2)

print(f"Average Pixel Diff: {diff.item()}")
```

```
Files already downloaded and verified
Average Pixel Diff: 1.8988681379406458e-16
```