Problem2

October 7, 2023

1 Chapter 3 - Deep Learning Development with PyTorch

[ ]:
```
import torch
import torchvision
from torchvision.datasets import CIFAR10
```

1.1 Data Transforms

[ ]:
```
from torchvision import transforms

train_transforms = transforms.Compose([
  transforms.RandomCrop(32, padding=4),
  transforms.RandomHorizontalFlip(),
  transforms.ToTensor(),
  transforms.Normalize(
      (0.4914, 0.4822, 0.4465),
      (0.2023, 0.1994, 0.2010))])

train_data_transforms = CIFAR10(root="./train/",
                    train=True,
                    download=True,
                    transform=train_transforms)
```

```
Files already downloaded and verified
```

[ ]:
```
test_transforms = transforms.Compose([
  transforms.ToTensor(),
  transforms.Normalize(
      (0.4914, 0.4822, 0.4465),
      (0.2023, 0.1994, 0.2010))])

test_data_transforms = torchvision.datasets.CIFAR10(
      root="./test/",
      train=False,
      transform=test_transforms)
```

[ ]:
```
targets = [1, 3, 5, 9]
```

```
indices = [i for i, label in enumerate(train_data_transforms.targets) if label␣
 ↪in targets]
indices_t = [i for i, label in enumerate(test_data_transforms.targets) if label␣
 ↪in targets]

from torch.utils.data.dataset import Subset

train_subset = Subset(train_data_transforms, indices)
test_subset = Subset(test_data_transforms, indices_t)
```

1.2 Data Batching

[ ]:
```
trainloader = torch.utils.data.DataLoader(
                    train_subset,
                    batch_size=16,
                    shuffle=True)
```

[ ]:
```
data_batch, labels_batch = next(iter(trainloader))
print(data_batch.size())

print(labels_batch.size())
```

```
torch.Size([16, 3, 32, 32])
torch.Size([16])
```

[ ]:
```
testloader = torch.utils.data.DataLoader(
                    test_subset,
                    batch_size=16,
                    shuffle=True)
```

1.3 Model Design

1.3.1 Using Existing & Pre-trained models

[ ]:
```
from torchvision import models

vgg16 = models.vgg16(pretrained=True)
```

```
c:\Users\kaasa\AppData\Local\Programs\Python\Python310\lib\site-
packages\torchvision\models\_utils.py:208: UserWarning: The parameter
'pretrained' is deprecated since 0.13 and may be removed in the future, please
use 'weights' instead.
  warnings.warn(
c:\Users\kaasa\AppData\Local\Programs\Python\Python310\lib\site-
packages\torchvision\models\_utils.py:223: UserWarning: Arguments other than a
weight enum or `None` for 'weights' are deprecated since 0.13 and may be removed
in the future. The current behavior is equivalent to passing
`weights=VGG16_Weights.IMAGENET1K_V1`. You can also use
```

```
`weights=VGG16_Weights.DEFAULT` to get the most up-to-date weights.
  warnings.warn(msg)
```

[ ]:
```
print(vgg16.classifier)
```

```
Sequential(
  (0): Linear(in_features=25088, out_features=4096, bias=True)
  (1): ReLU(inplace=True)
  (2): Dropout(p=0.5, inplace=False)
  (3): Linear(in_features=4096, out_features=4096, bias=True)
  (4): ReLU(inplace=True)
  (5): Dropout(p=0.5, inplace=False)
  (6): Linear(in_features=4096, out_features=1000, bias=True)
)
```

[ ]:
```
import torch.nn as nn
vgg16.classifier[-1] = nn.Linear(4096,4)

print(vgg16.classifier)

device = "cuda" if torch.cuda.is_available() else "cpu"

vgg_model = vgg16.to(device = device)
```

```
Sequential(
  (0): Linear(in_features=25088, out_features=4096, bias=True)
  (1): ReLU(inplace=True)
  (2): Dropout(p=0.5, inplace=False)
  (3): Linear(in_features=4096, out_features=4096, bias=True)
  (4): ReLU(inplace=True)
  (5): Dropout(p=0.5, inplace=False)
  (6): Linear(in_features=4096, out_features=4, bias=True)
)
```

1.4 The PyTorch NN Module (torch.nn)

1.5 Training

1.5.1 Fundamental Training Loop

Code Annotations:

<1> Our training loop

<2> Need to move inputs and labels to GPU is avail.

<3> Zero out gradients before each backprop or they'll accumulate

<4> Forward pass

<5> Compute loss

<6> Backpropagation, compute gradients

<7> Adjust parameters based on gradients

<8> accumulate batch loss so we can average over epoch

[ ]:
```
from torch import optim
from torch import nn

criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(vgg_model.parameters(), # <1>
                      lr=0.001,
                      momentum=0.9)
```

[ ]:
```
N_EPOCHS = 10
for epoch in range(N_EPOCHS): # <1>

    epoch_loss = 0.0
    for inputs, labels in trainloader:
        inputs = inputs.to(device) # <2>

        labmap = {x:i for i, x in enumerate(targets)}

        tars = [labmap[label] for label in labels.tolist()]

        modified_labels = torch.tensor(tars)
        modified_labels = modified_labels.to(device)

        optimizer.zero_grad() # <3>

        outputs = vgg_model(inputs) # <4>
        loss = criterion(outputs,modified_labels ) # <5>
        loss.backward() # <6>
        optimizer.step() # <7>

        epoch_loss += loss.item() # <8>
    print("Epoch: {} Loss: {}".format(epoch,
                  epoch_loss/len(trainloader)))
```

```
Epoch: 0 Loss: 0.4555310956761241
Epoch: 1 Loss: 0.3098580519348383
Epoch: 2 Loss: 0.26245294438153505
Epoch: 3 Loss: 0.23032913172133268
Epoch: 4 Loss: 0.20355475818254054
Epoch: 5 Loss: 0.18501621563639492
Epoch: 6 Loss: 0.16868607298964636
Epoch: 7 Loss: 0.1534964678324759
Epoch: 8 Loss: 0.1456668941570446
Epoch: 9 Loss: 0.12939511819183827
```

[ ]:
```python
num_correct = 0.0

labmap = {i:x for i,x in enumerate(targets)}

for x_test_batch, y_test_batch in testloader:

    vgg_model.eval()

    y_test_batch = y_test_batch.to(device)

    x_test_batch = x_test_batch.to(device)

    y_pred_batch = vgg_model(x_test_batch)

    _, predicted = torch.max(y_pred_batch, 1)

    final_prediction =torch.tensor([labmap[label] for label in predicted.
 ↪tolist()])
    final_prediction = final_prediction.to(device=device)

    num_correct += (final_prediction == y_test_batch).float().sum()



accuracy = num_correct/(len(testloader)*testloader.batch_size)



print(len(testloader), testloader.batch_size)

print("Test Accuracy: {}".format(accuracy))
```

```
250 16
Test Accuracy: 0.906000018119812
```

[ ]:
```python
from torch import nn
import torch.nn.functional as F

class LeNet5(nn.Module):
    def __init__(self):
        super(LeNet5, self).__init__()
        self.conv1 = nn.Conv2d(3, 6, 5) # <1>
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)
```

```python
    def forward(self, x):
        x = F.max_pool2d(F.relu(self.conv1(x)), (2, 2))
        x = F.max_pool2d(F.relu(self.conv2(x)), 2)
        x = x.view(-1, int(x.nelement() / x.shape[0]))
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x

device = "cuda" if torch.cuda.is_available() else "cpu"
LeNet_model = LeNet5().to(device=device)
```

[ ]:
```python
from torch import optim
from torch import nn

criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(LeNet_model.parameters(), # <1>
                      lr=0.001,
                      momentum=0.9)
```

[ ]:
```python
N_EPOCHS = 10
for epoch in range(N_EPOCHS): # <1>

    epoch_loss = 0.0
    for inputs, labels in trainloader:
        inputs = inputs.to(device) # <2>

        labmap = {x:i for i, x in enumerate(targets)}

        tars = [labmap[label] for label in labels.tolist()]

        modified_labels = torch.tensor(tars)
        modified_labels = modified_labels.to(device)

        optimizer.zero_grad() # <3>

        outputs = LeNet_model(inputs) # <4>
        loss = criterion(outputs, modified_labels) # <5>
        loss.backward() # <6>
        optimizer.step() # <7>

        epoch_loss += loss.item() # <8>
    print("Epoch: {} Loss: {}".format(epoch,
                      epoch_loss/len(trainloader)))
```

```
Epoch: 0 Loss: 1.1889315284252167
Epoch: 1 Loss: 0.9005565319538117
Epoch: 2 Loss: 0.842382780623436
```

```
Epoch: 3 Loss: 0.8132672204732895
Epoch: 4 Loss: 0.7868075754642486
Epoch: 5 Loss: 0.760386355304718
Epoch: 6 Loss: 0.7387431738853455
Epoch: 7 Loss: 0.7251886307954788
Epoch: 8 Loss: 0.7022756109476089
Epoch: 9 Loss: 0.6788096746921539
```

[ ]:
```
num_correct = 0.0

labmap = {i:x for i,x in enumerate(targets)}

for x_test_batch, y_test_batch in testloader:

    vgg_model.eval()

    y_test_batch = y_test_batch.to(device)

    x_test_batch = x_test_batch.to(device)

    y_pred_batch = LeNet_model(x_test_batch)

    _, predicted = torch.max(y_pred_batch, 1)

    final_prediction =torch.tensor([labmap[label] for label in predicted.
 ↪tolist()])
    final_prediction = final_prediction.to(device=device)

    num_correct += (final_prediction == y_test_batch).float().sum()



accuracy = num_correct/(len(testloader)*testloader.batch_size)



print(len(testloader), testloader.batch_size)

print("Test Accuracy: {}".format(accuracy))
```

```
250 16
Test Accuracy: 0.7205000519752502
```

[ ]:
```
print(LeNet_model)
print(vgg_model)
```

```
LeNet5(
  (conv1): Conv2d(3, 6, kernel_size=(5, 5), stride=(1, 1))
  (conv2): Conv2d(6, 16, kernel_size=(5, 5), stride=(1, 1))
```

```
  (fc1): Linear(in_features=400, out_features=120, bias=True)
  (fc2): Linear(in_features=120, out_features=84, bias=True)
  (fc3): Linear(in_features=84, out_features=10, bias=True)
)
VGG(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU(inplace=True)
    (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): ReLU(inplace=True)
    (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
    (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (6): ReLU(inplace=True)
    (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (8): ReLU(inplace=True)
    (9): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
    (10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): ReLU(inplace=True)
    (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (13): ReLU(inplace=True)
    (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (15): ReLU(inplace=True)
    (16): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
    (17): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (18): ReLU(inplace=True)
    (19): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (20): ReLU(inplace=True)
    (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (22): ReLU(inplace=True)
    (23): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
    (24): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (25): ReLU(inplace=True)
    (26): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (27): ReLU(inplace=True)
    (28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (29): ReLU(inplace=True)
    (30): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(7, 7))
  (classifier): Sequential(
    (0): Linear(in_features=25088, out_features=4096, bias=True)
    (1): ReLU(inplace=True)
    (2): Dropout(p=0.5, inplace=False)
```

```
    (3): Linear(in_features=4096, out_features=4096, bias=True)
    (4): ReLU(inplace=True)
    (5): Dropout(p=0.5, inplace=False)
    (6): Linear(in_features=4096, out_features=4, bias=True)
  )
)
```