selfresnet

August 19, 2023

[ ]:
```
import gc
import torch
import numpy as np
import torch.nn as nn
from torchvision import datasets
from torchvision import transforms
from torch.utils.data.sampler import SubsetRandomSampler
```

[ ]:
```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
```

[ ]:
```
def data_loader(data_dir, batch_size, random_seed = 42, valid_size = 0.1,␣
↪shuffle=True, test=False):
    normalize = transforms.Normalize(mean=[0.4914, 0.4822, 0.4465], std=[0.
↪2023, 0.1994, 0.2010])
    transform = transforms.Compose([transforms.Resize((224,224)), transforms.
↪ToTensor(), normalize])

    if test:
        dataset = datasets.CIFAR10(root=data_dir, train=False, download=True,␣
↪transform=transform)
        data_loader = torch.utils.data.DataLoader(dataset,␣
↪batch_size=batch_size, shuffle=shuffle)

        return data_loader

    # load the dataset
    train_dataset = datasets.CIFAR10(root=data_dir, train=True, download=True,␣
↪transform=transform)
    valid_dataset = datasets.CIFAR10(root=data_dir, train=True, download=True,␣
↪transform=transform)

    num_train = len(train_dataset)
    indices = list(range(num_train))
    split = int(np.floor(valid_size * num_train))

    if shuffle:
        np.random.seed(42)
```

```
        np.random.shuffle(indices)

    train_idx, valid_idx = indices[split:], indices[:split]
    train_sampler = SubsetRandomSampler(train_idx)
    valid_sampler = SubsetRandomSampler(valid_idx)

    train_loader = torch.utils.data.DataLoader(train_dataset,␣
↪batch_size=batch_size, sampler=train_sampler)
    valid_loader = torch.utils.data.DataLoader(valid_dataset,␣
↪batch_size=batch_size, sampler=valid_sampler)

    return (train_loader, valid_loader)


# CIFAR10 dataset
train_loader, valid_loader = data_loader(data_dir='./data',batch_size=64)
test_loader = data_loader(data_dir='./data', batch_size=64, test=True)
```

```
Files already downloaded and verified
Files already downloaded and verified
Files already downloaded and verified
```

[ ]:
```
class ResidualBlock(nn.Module):
    def __init__(self, in_channels, out_channels, stride = 1, downsample =␣
↪None) :
        super(ResidualBlock, self).__init__()       #Super is used so we don't␣
↪have problems later on in MRO
        self.conv1 = nn.Sequential(                 #Sequential just stitches␣
↪multiple steps in 1
                        nn.Conv2d(in_channels, out_channels, kernel_size=3,␣
↪stride=stride, padding=1),      #convolution step to create feature maps
                        nn.BatchNorm2d(out_channels),       #Batch Norm is dont␣
↪to normalize the distribution so that training is faster and you dont overfit
                        nn.ReLU())                          #Activation␣
↪function to make it non linear
        self.conv2 = nn.Sequential(
                        nn.Conv2d(out_channels, out_channels, kernel_size=3,␣
↪stride=1, padding=1),
                        nn.BatchNorm2d(out_channels)) #this is not immediately␣
↪followed by Relu because residual needs to be added before Relu
        self.downsample = downsample
        self.relu = nn.ReLU()
        self.out_channels = out_channels

    def forward(self, x):
        residual = x
        out = self.conv1(x)
```

```python
        out = self.conv2(out)
        if self.downsample:
            residual = self.downsample(x)
        out += residual
        out = self.relu(out)
        return out
```

[ ]:
```python
class Resnet(nn.Module):
    def __init__(self, block, layers, num_classes = 10):
        super(Resnet, self).__init__()
        self.inplanes = 64
        self.conv1 = nn.Sequential(nn.Conv2d(3, 64, kernel_size=7, stride=2,␣
↪padding=3), nn.BatchNorm2d(64), nn.ReLU())
        self.maxpool = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)
        self.layer0 = self._make_layer(block, 64, layers[0], stride=1)
        self.layer1 = self._make_layer(block, 128, layers[1], stride=2)
        self.layer2 = self._make_layer(block, 256, layers[2], stride=2)
        self.layer3 = self._make_layer(block, 512, layers[3], stride=2)
        self.avgpool = nn.AvgPool2d(7, stride=1)
        self.fc = nn.Linear(512, num_classes)

    def _make_layer(self, block, planes, blocks, stride=1):
        downsample = None
        if stride != 1 or self.inplanes != planes:
            downsample = nn.Sequential(nn.Conv2d(self.inplanes, planes,␣
↪kernel_size=1, stride=stride), nn.BatchNorm2d(planes))

        layers = []
        layers.append(block(self.inplanes, planes, stride, downsample))
        self.inplanes = planes
        for i in range(1, blocks):
            layers.append(block(self.inplanes, planes))

        return nn.Sequential(*layers)

    def forward(self, x):
        # print(x.shape)
        x = self.conv1(x)
        # print(x.shape)
        x = self.maxpool(x)
        # print(x.shape)
        x = self.layer0(x)
        # print(x.shape)
        x = self.layer1(x)
        x = self.layer2(x)
        x = self.layer3(x)
```

```
        x = self.avgpool(x)
        x = x.view(x.size(0), -1)
        x = self.fc(x)

        return x
```

[ ]:
```
# from torchsummary import summary
# input_shape = (3,224,224)
# summary(Resnet(ResidualBlock, [3, 4, 6, 3]).to(device), input_shape)
```

[ ]:
```
num_class = 10
num_epochs = 20
batch_size = 16
learning_rate = 0.01

model = Resnet(ResidualBlock, [3, 4, 6, 3]).to(device)

criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate,␣
 ↪weight_decay=0.001, momentum=0.9)

total_step = len(train_loader)
```

[ ]:
```
total_step = len(train_loader)

for epoch in range(num_epochs):
    for i, (images, labels) in enumerate(train_loader):
        images = images.to(device)
        labels = labels.to(device)

        outputs = model(images)
        loss = criterion(outputs, labels)

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        del images, labels, outputs
        torch.cuda.empty_cache()
        gc.collect()
    print('Epoch [{}/{}], Loss: {:.4f}'.format(epoch+1, num_epochs, loss.
 ↪item()))

    with torch.no_grad():
        correct = 0
        total = 0
        for images, labels in valid_loader:
            images = images.to(device)
```

```
            labels = labels.to(device)
            outputs = model(images)
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()
            del images, labels, outputs

        print('Accuracy of the network on the {} validation images: {} %'.
↪format(5000, 100 * correct / total))
```

```
Epoch [1/20], Loss: 0.8324
Accuracy of the network on the 5000 validation images: 60.86 %
Epoch [2/20], Loss: 0.4702
Accuracy of the network on the 5000 validation images: 74.36 %
Epoch [3/20], Loss: 0.7160
Accuracy of the network on the 5000 validation images: 78.68 %
Epoch [4/20], Loss: 0.6268
Accuracy of the network on the 5000 validation images: 80.18 %
Epoch [5/20], Loss: 1.6529
Accuracy of the network on the 5000 validation images: 82.38 %
Epoch [6/20], Loss: 0.4987
Accuracy of the network on the 5000 validation images: 82.14 %
Epoch [7/20], Loss: 0.3876
Accuracy of the network on the 5000 validation images: 82.3 %
Epoch [8/20], Loss: 0.2095
Accuracy of the network on the 5000 validation images: 82.78 %
Epoch [9/20], Loss: 0.6433
Accuracy of the network on the 5000 validation images: 82.98 %
Epoch [10/20], Loss: 1.8922
Accuracy of the network on the 5000 validation images: 82.8 %
Epoch [11/20], Loss: 0.5669
Accuracy of the network on the 5000 validation images: 84.18 %
Epoch [12/20], Loss: 0.3668
Accuracy of the network on the 5000 validation images: 83.36 %
Epoch [13/20], Loss: 0.2292
Accuracy of the network on the 5000 validation images: 83.14 %
Epoch [14/20], Loss: 0.8975
Accuracy of the network on the 5000 validation images: 82.72 %
Epoch [15/20], Loss: 0.0139
Accuracy of the network on the 5000 validation images: 82.66 %
Epoch [16/20], Loss: 0.1612
Accuracy of the network on the 5000 validation images: 83.62 %
Epoch [17/20], Loss: 0.0105
Accuracy of the network on the 5000 validation images: 84.34 %
Epoch [18/20], Loss: 0.0358
Accuracy of the network on the 5000 validation images: 82.42 %
Epoch [19/20], Loss: 0.8057
Accuracy of the network on the 5000 validation images: 83.92 %
```

```
Epoch [20/20], Loss: 0.1819
Accuracy of the network on the 5000 validation images: 83.96 %
```

[ ]:
```python
with torch.no_grad():
    correct = 0
    total = 0
    for images, labels in test_loader:
        images = images.to(device)
        labels = labels.to(device)
        outputs = model(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()
        del images, labels, outputs

    print('Accuracy of the network on the {} test images: {} %'.format(10000,␣
 ↪100 * correct / total))
```

```
Accuracy of the network on the 10000 test images: 83.31 %
```