之前一个笔记中学习了PyTorch中的一些重要概念，如Tensor等，以及和神经网络的构建，接下来我们可以用所学到的知识来进行一个简单的实战。

上一个笔记链接：https://blog.csdn.net/whatiscode/article/details/106473811

小试牛刀：CIFAR-10分类

下面我们来尝试实现对CIFAR-10数据集的分类，步骤如下:

1. 使用torchvision加载并预处理CIFAR-10数据集
2. 定义网络
3. 定义损失函数和优化器
4. 训练网络并更新网络参数
5. 测试网络

1 CIFAR-10数据加载及预处理

CIFAR-10是一个常用的彩色图片数据集，它有10个类别: 'airplane', 'automobile', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck'。每张图片都是3 X 32 X 32，也即3-通道彩色图片，分辨率为32 X 32

CIFAR-10数据集下载地址：http://www.cs.toronto.edu/~kriz/cifar.html

```
import torch as t
import torchvision as tv
import torchvision.transforms as transforms
from torchvision.transforms import ToPILImage
show = ToPILImage() # 可以把Tensor转成Image，方便可视化

# 第一次运行程序torchvision会自动下载CIFAR-10数据集，
# 大约100M，需花费一定的时间，
# 如果已经下载有CIFAR-10，可通过root参数指定

# 定义对数据的预处理
transform = transforms.Compose([
        transforms.ToTensor(), # 转为Tensor
        transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)), # 归一化
                             ])

# 训练集
trainset = tv.datasets.CIFAR10(
                    root='data/',
                    train=True,
                    download=True,
                    transform=transform)

trainloader = t.utils.data.DataLoader(
                    trainset,
                    batch_size=4,
                    shuffle=True,  # shuffle 表示是否打乱顺序
                    num_workers=2)

# 测试集
testset = tv.datasets.CIFAR10(
                    'data/',
                    train=False,
                    download=True,
                    transform=transform)

testloader = t.utils.data.DataLoader(
                    testset,
                    batch_size=4,
                    shuffle=False,
                    num_workers=2)

classes = ('plane', 'car', 'bird', 'cat',
           'deer', 'dog', 'frog', 'horse', 'ship', 'truck')
```

Dataset对象是一个数据集，可以按下标访问，返回形如(data, label)的数据。

```
(data, label) = trainset[100]
print(classes[label])

# (data + 1) / 2是为了还原被归一化的数据
show((data + 1) / 2).resize((100, 100))
```

Dataloader是一个可迭代的对象，它将dataset返回的每一条数据拼接成一个batch，并提供多线程加速优化和数据打乱等操作。当

程序对dataset的所有数据遍历完一遍之后，相应的对Dataloader也完成了一次迭代。

```
dataiter = iter(trainloader)
images, labels = dataiter.next() # 返回4张图片及标签
print(' '.join('%11s'%classes[labels[j]] for j in range(4)))
show(tv.utils.make_grid((images+1)/2)).resize((400,100))
```

关于Dataset和Dataloader更详细的解释可以见该博客第3点：https://blog.csdn.net/whatiscode/article/details/106301541

2 定义网络

使用上一个笔记中提到的LeNet网络，修改self.conv1第一个参数为3通道，因CIFAR-10是3通道彩图。

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

class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        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)

    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(x.size()[0], -1)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x


net = Net()
print(net)
```

3 定义损失函数和优化器(loss和optimizer)

```
from torch import optim
criterion = nn.CrossEntropyLoss() # 交叉熵损失函数
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)
```

4 训练网络并更新网络参数

所有网络的训练流程都是类似的，不断地执行如下流程：

1. 输入数据
2. 前向传播+反向传播
3. 更新参数

```
t.set_num_threads(8)
for epoch in range(2):

    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):

        # 输入数据
        inputs, labels = data

        # 梯度清零
        optimizer.zero_grad()

        # forward + backward
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()

        # 更新参数
        optimizer.step()

        # 打印log信息
        # loss 是一个scalar,需要使用loss.item()来获取数值，不能使用loss[0]
        running_loss += loss.item()
```

```
        if i % 2000 == 1999: # 每2000个batch打印一下训练状态
            print('[%d, %5d] loss: %.3f' \
                  % (epoch+1, i+1, running_loss / 2000))
            running_loss = 0.0
print('Finished Training')
```

在GPU训练

就像之前把Tensor从CPU转到GPU一样，模型也可以类似地从CPU转到GPU。如果发现在GPU上并没有比CPU提速很多，实际上是因为网络比较小，GPU没有完全发挥自己的真正实力。

```
device = t.device("cuda:0" if t.cuda.is_available() else "cpu")

net.to(device)
images = images.to(device)
labels = labels.to(device)
output = net(images)
loss= criterion(output,labels)

loss
```

5 测试网络

上面仅训练了2个epoch（遍历完一遍数据集称为一个epoch），来看看网络有没有效果。将测试图片输入到网络中，计算它的label，然后与实际的label进行比较。

5.1 测试部分测试样本

```
dataiter = iter(testloader)
images, labels = dataiter.next() # 一个batch返回4张图片
print('实际的label: ', ' '.join(\
            '%08s'%classes[labels[j]] for j in range(4)))
show(tv.utils.make_grid(images / 2 - 0.5)).resize((400,100))
```

接着计算网络预测的label:

```
# 计算图片在每个类别上的分数
outputs = net(images)
# 得分最高的那个类
_, predicted = t.max(outputs.data, 1)

print('预测结果: ', ' '.join('%5s'\
            % classes[predicted[j]] for j in range(4)))
```

已经可以看出效果，准确率50%，但这只是一部分的图片，再来看看在整个测试集上的效果。

5.2 测试所有测试样本

```
correct = 0 # 预测正确的图片数
total = 0 # 总共的图片数


# 由于测试的时候不需要求导，可以暂时关闭autograd，提高速度，节约内存
with t.no_grad():
    for data in testloader:
        images, labels = data
        outputs = net(images)
        _, predicted = t.max(outputs, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum()

print('10000张测试集中的准确率为: %d %%' % (100 * correct / total))
```

2个epoch下，10000张测试集中的准确率为: 53%，训练的准确率远比随机猜测(准确率10%)好，证明网络确实学到了东西。

10个epoch下的准确率为: 61%，所以当没有发生过拟合时，epoch次数越多，准确率会越高。

6 总结

以上就完成了一个简单的小项目，是不是很简单呢？不过想要进一步提高模型的准确率，还需要学习很多调参技巧或者选择更加复杂的模型，所以我们需要继续往下学咯。