项目实战——利用 PyTorch 构建 RNN模型

本文将通过一个实战项目带大家使用PyTorch 搭建 RNN 模型。

本项目将构建一个 RNN 模型，来对 MNIST 手写数据集进行分类。可能有同学会有疑问，MNIST 数据集不是图片吗，而 RNN 是处理序列信号的。为什么图片识别也能使用 RNN 模型呢？其实，这里我们可以把图片看成是序列信号，例如下面是 MNIST 数据集的一张图片：

MNIST 数据集中所有的图片都是 28x28 的。按行来看，图片的每一行都包含 28 个像素点，一共有 28 行。因此，我们可以把每一行的 28 个像素点当成 RNN 的一个输入 $x^{<t>}$。总共有 28 行，则 $T_x = 28$。图片的分割方式如下图所示：

输入已经确定了，对于输出，因为是分类问题，识别 0～9 数字，因此，RNN 模型应该有 10 个输出，即 $T_y = 10$。此例中，$T_x \neq T_y$。

确定了基本结构和输入输出之后，我们开始使用 PyTorch 构建 RNN 。首先，还是导入 MNIST 数据集。

导入数据集

下面代码实现了 MNIST 数据集的导入。

```
import torch
import torchvision
import torchvision.transforms as transforms
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import matplotlib.pyplot as plt
import numpy as np

transform = transforms.Compose(
    [transforms.ToTensor()])

# 训练集
trainset = torchvision.datasets.MNIST(root='./data',      # 选择数据的根目录
                                      train=True,
```

```
                                        download=False,     # 不从网络上download图片
                                        transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=4,
                                          shuffle=True, num_workers=2)
# 测试集
testset = torchvision.datasets.MNIST(root='./data',      # 选择数据的根目录
                                     train=False,
                                     download=False,     # 不从网络上download图片
                                     transform=transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=4,
                                         shuffle=False, num_workers=2)
```

上述代码中，我们也可以设置 `download=True`，表示在线下载数据集。因为我已提前下载完成，这里的 download 设置为 False，即从本地导入数据集。我们设置 `batch_size=4`，`shuffle=True` 表示每次 epoch 都重新打乱训练样本，`num_workers=2` 表示使用两个子进程加载数据。

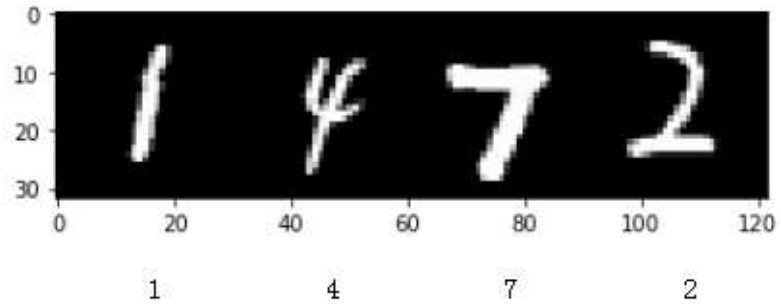
下面程序展示了 Mini-batch 训练样本图片并标注正确标签的过程。

```
def imshow(img):
    npimg = img.numpy()
    plt.imshow(np.transpose(npimg, (1, 2, 0)))

# 选择一个 batch 的图片
dataiter = iter(trainloader)
images, labels = dataiter.next()

# 显示图片
imshow(torchvision.utils.make_grid(images))
plt.show()
# 打印 labels
print(' '.join('%11s' % labels[j].numpy() for j in range(4)))
```

运行结果，如下图所示：



我们可以来看一下训练集和测试集的维度。

```
print(trainset.train_data.size())
print(testset.test_data.size())
```

运行结果如下：

```
torch.Size([60000, 28, 28])
torch.Size([10000, 28, 28])
```

训练集包含 60000 张图片，测试集包含 10000 张图片，每张图片大小为 28x28。

定义 RNN 模型

与 CNN 类似，我们可以使用 PyTorch 直接搭建 RNN 模型，首先定义 RNN 类。

```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.rnn = nn.LSTM(           # 使用 LSTM 结构
            input_size = 28,          # 输入每个元素的维度，即图片每行包含 28 个像素点
            hidden_size = 84,         # 隐藏层神经元设置为 84 个
            num_layers=1,             # 隐藏层数目，单层
            batch_first=True,         # 是否将 batch 放在维度的第一位，(batch, time_step, input_size)
        )
        self.out = nn.Linear(84, 10) # 输出层，包含 10 个神经元，对应 0～9 数字

```

```
    def forward(self, x):
        r_out, (h_n, h_c) = self.rnn(x, None)
        # 选择图片的最后一行作为 RNN 输出
        out = self.out(r_out[:, -1, :])
        return out
```

以上代码是构建 RNN 的核心部分。我们发现 PyTroch 中构建 RNN 模型非常简单，只需简单的几行语句。下面对上面的部分代码做下重点讲解。

代码中 `input_size = 28` 表示每个输入元素 $x^{<t>}$ 的维度，即图片每行包含 28 个像素点。`hidden_size = 84` 将隐藏层神经元设置为 84 个，`num_layers=2` 表示有两层隐藏层。`self.out = nn.Linear(84, 10)` 表示输出层，输出为 0～9 数字。

`r_out, (h_n, h_c) = self.rnn(x, None)` 中，`(h_n, h_c)` 为 LSTM 的记忆单元，`r_out` 为输出，每个 $x^{<t>}$ 的输出都会累加在 `r_out` 中。None 表示最初的隐藏层记忆单元为 0。`out = self.out(r_out[:, -1, :])` 表示选择图片最后一行的结果作为输出。

接下来我们可以建立一个 Net 对象，并打印出来，看看其网络结构。

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

运行结果，如下：

```
Net(
   (rnn): LSTM(28, 84, num_layers=2, batch_first=True)
   (out): Linear(in_features=84, out_features=10, bias=True)
 )
```

非常直观，可以完整清晰地查看我们构建的 RNN 模型结构。

定义损失函数

正如之前利用 PyTorch 构建 CNN 模型的实战过程，我们仍使用 Adam 梯度优化算法，即：

```
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(net.parameters(), lr=0.0001)
```

训练网络

接下来就是最有趣的地方了。我们只需循环遍历数据迭代器，放入网络的输入层并优化即可。

```
num_epoches = 5     # 设置 epoch 数目
cost = []      # 损失函数累加

for epoch in range(num_epoches):

    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        # 输入样本和标签
        inputs, labels = data
        inputs = inputs.view(-1, 28, 28)  # 设置 RNN 输入维度为 (batch, time_step, input_size)

        # 每次训练梯度清零
        optimizer.zero_grad()

        # 正向传播、反向传播和优化过程
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        # 打印训练情况
        running_loss += loss.item()
        if i % 2000 == 1999:    # 每隔2000 mini-batches, 打印一次
            print('[%d, %5d] loss: %.3f' %
                  (epoch + 1, i + 1, running_loss / 2000))
            cost.append(running_loss / 2000)
            running_loss = 0.0
```

上述代码中需要注意的是，每次迭代训练时都要先把所有梯度清零，即 `optimizer.zero_grad()`。否则，梯度会累加，造成训练错误和失效。PyTorch 中的 `.backward()` 能自动完成所有梯度计算。我们发现，PyTorch 中 RNN 的训练代码与 CNN 十分类似，只不过 RNN 的输入维度由 `inputs = inputs.view(-1, 28, 28)` 语句重新设置

为`(batch, time_step, input_size)`，即（4, 28, 28）。

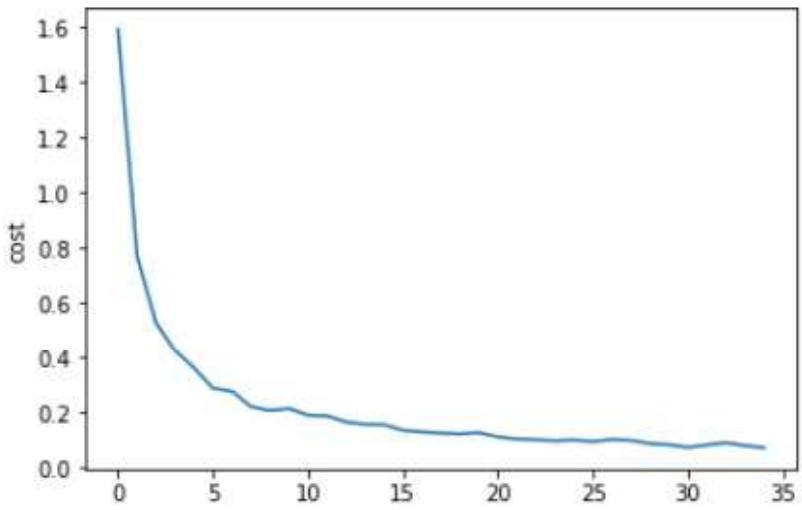
打印的结果如下：

```
[1,  2000] loss: 1.594
[1,  4000] loss: 0.772
[1,  6000] loss: 0.528
[1,  8000] loss: 0.425
[1, 10000] loss: 0.362
[1, 12000] loss: 0.287
[1, 14000] loss: 0.276
[2,  2000] loss: 0.221
[2,  4000] loss: 0.206
[2,  6000] loss: 0.214
[2,  8000] loss: 0.189
[2, 10000] loss: 0.187
[2, 12000] loss: 0.165
[2, 14000] loss: 0.157
[3,  2000] loss: 0.155
[3,  4000] loss: 0.135
[3,  6000] loss: 0.129
[3,  8000] loss: 0.125
[3, 10000] loss: 0.122
[3, 12000] loss: 0.126
[3, 14000] loss: 0.111
[4,  2000] loss: 0.103
[4,  4000] loss: 0.100
[4,  6000] loss: 0.097
[4,  8000] loss: 0.100
[4, 10000] loss: 0.094
[4, 12000] loss: 0.101
[4, 14000] loss: 0.098
[5,  2000] loss: 0.087
[5,  4000] loss: 0.083
[5,  6000] loss: 0.073
[5,  8000] loss: 0.082
[5, 10000] loss: 0.090
[5, 12000] loss: 0.080
[5, 14000] loss: 0.072
```

将所有 Loss 趋势绘制成图，如下所示：

```
plt.plot(cost)
```

```
    plt.ylabel('cost')
    plt.show()
```



显然，随着迭代训练，Loss 逐渐减小。

测试数据

让我们来看一下网络模型在整个测试数据集上的训练效果。

```
correct = 0
total = 0
with torch.no_grad():
    for data in testloader:
        images, labels = data
        images = images.view(-1, 28, 28)
        outputs = net(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

print('Accuracy of the network on the 10000 test images: %.3f %%' %
     (100 * correct / total))
```

执行结果如下：

> Accuracy of the network on the 10000 test images: 97.790 %

结果显示模型在测试集上的准确率达到了 97.790 %。说明我们训练的 RNN 模型性能还是不错的。