AI深度学习

深度学习介绍

Introduction of Deep Learning

手写数字识别

1. 使用torchvision加载并对MNIST数据集进行预处理
2. 定义网络结构
3. 定义损失函数和优化器
4. 训练神经网络并更新网络参数
5. 测试网络

```
  from fcnet import FcNet
1 import torch
2 import torchvision
3
4 trans = torchvision.transforms.Compose([
5             torchvision.transforms.ToTensor()])
6
7 train_data = torchvision.datasets.MNIST('./data',
8                             transform = trans,
9                             train=True,
10                            download = True)
11
12
13 test_data = torchvision.datasets.MNIST('./data',
14                            transform = trans, train=False, download=True)
15
16 train_data_loader = torch.utils.data.DataLoader(train_data,
17                                                 batch_size=200,
18                                                 shuffle=True,
19                                                 num_workers=4)
20
21 test_data_loader = torch.utils.data.DataLoader(test_data,
22                                                batch_size=200,
23                                                shuffle=False,
24                                                num_workers=4)
25
26
27 net = FcNet()
28 # net = FcNet2()
training.py" 112L, 2916C                                          1,1           Top
```

下一次读取数据时，数据的顺序都会被打乱，然后再进行下一次，从而两次数据读取到的顺序都是不同的

在实践中，数据读取经常是训练的性能瓶颈，特别当模型较简单或者计算硬件性能较高时。PyTorch的`DataLoader`中一个很方便的功能是允许使用多进程来加速数据读取。这里我们通过参数`num_workers`来设置4个进程读取数据。

```
batch_size = 256
if sys.platform.startswith('win'):
    num_workers = 0  # 0表示不用额外的进程来加速读取数据
else:
    num_workers = 4
train_iter = torch.utils.data.DataLoader(mnist_train,
batch_size=batch_size, shuffle=True, num_workers=num_workers)
test_iter = torch.utils.data.DataLoader(mnist_test,
batch_size=batch_size, shuffle=False, num_workers=num_workers)
```

我们将获取并读取Fashion-MNIST数据集的逻辑封装在`d2lzh_pytorch.load_data_fashion_mnist`函数中供后面章节调用。该函数将返回`train_iter`和`test_iter`两个变量。随着本书内容的不断深入，我们会进一步改进该函数。它的完整实现将在5.6节中描述。

3.3.3 定义模型

在上一节从零开始的实现中，我们需要定义模型参数，并使用它们一步步描述模型是怎样计算的。当模型结构变得更复杂时，这些步骤将变得更繁琐。其实，PyTorch提供了大量预定义的层，这使我们只需关注使用哪些层来构造模型。下面将介绍如何使用PyTorch更简洁地定义线性回归。

首先，导入 `torch.nn` 模块。实际上，“nn”是neural networks（神经网络）的缩写。顾名思义，该模块定义了大量神经网络的层。之前我们已经用过了 `autograd`，而 `nn` 就是利用 `autograd` 来定义模型。`nn` 的核心数据结构是 `Module`，它是一个抽象概念，既可以表示神经网络中的某个层（layer），也可以表示一个包含很多层的神经网络。在实际使用中，最常见的做法是继承 `nn.Module`，撰写自己的网络/层。一个 `nn.Module` 实例应该包含一些层以及返回输出的前向传播（forward）方法。下面先来看看如何用 `nn.Module` 实现一个线性回归模型。

```
class LinearNet(nn.Module):
    def __init__(self, n_feature):
        super(LinearNet, self).__init__()
        self.linear = nn.Linear(n_feature, 1)
    # forward 定义前向传播
    def forward(self, x):
        y = self.linear(x)
        return y

net = LinearNet(num_inputs)
print(net) # 使用print可以打印出网络的结构
```

```python
import torch.nn as nn

class FcNet1(nn.Module):

    def __init__(self, **kwargs):
        super(FcNet1, self).__init__(**kwargs)

        self.hidden = nn.Linear(784, 256)
        self.active = nn.ReLU()
        self.output = nn.Linear(256, 10)


    def forward(self, img):
        act_out = self.act(self.hidden(img))
        output = self.output(act_out)
        return output
```

1,1 All

```
jqlg@X1:~/dl/fc$ python print_net.py
FcNet1(
  (hidden): Linear(in_features=784, out_features=256, bias=True)
  (active): ReLU()
  (output): Linear(in_features=256, out_features=10, bias=True)
)
jqlg@X1:~/dl/fc$
```

```
from fcnet1 import FcNet1

net = FcNet1()
print(net)
```

```
"print_net.py" 4L, 54C                                        1,1           All
```

```python
import torch.nn as nn

class FcNet3(nn.Module):

    def __init__(self, **kwargs):
        super(FcNet3, self).__init__(**kwargs)

        self.hidden1 = nn.Linear(784, 500)
        self.active1 = nn.ReLU()
        self.hidden2 = nn.Linear(500, 300)
        self.active2 = nn.ReLU()
        self.hidden3 = nn.Linear(300, 100)
        self.active3 = nn.ReLU()
        self.output = nn.Linear(100, 10)


    def forward(self, img):
        img = img.view(-1, 784);
        hid_out1 = self.hidden1(img)
        act_out1 = self.active1(hid_out1)
        hid_out2 = self.hidden2(act_out1)
        act_out2 = self.active2(hid_out2)
        hid_out3 = self.hidden3(act_out2)
        act_out3 = self.active3(hid_out3)
        output = self.output(act_out3)
        return output
```

26,1 All

```
jqlg@X1:~/dl/fc$ python print_net3.py
FcNet3(
  (hidden1): Linear(in_features=784, out_features=500, bias=True)
  (active1): ReLU()
  (hidden2): Linear(in_features=500, out_features=300, bias=True)
  (active2): ReLU()
  (hidden3): Linear(in_features=300, out_features=100, bias=True)
  (active3): ReLU()
  (output): Linear(in_features=100, out_features=10, bias=True)
)
jqlg@X1:~/dl/fc$
```

```python
# 写法一
net = nn.Sequential(
    nn.Linear(num_inputs, 1)
    # 此处还可以传入其他层
    )

# 写法二
net = nn.Sequential()
net.add_module('linear', nn.Linear(num_inputs, 1))
# net.add_module ......

# 写法三
from collections import OrderedDict
net = nn.Sequential(OrderedDict([
          ('linear', nn.Linear(num_inputs, 1))
          # ......
        ]))

print(net)
print(net[0])
```

手写数字识别

1. 使用torchvision加载并对MNIST数据集进行预处理
2. 定义网络结构
3. 定义损失函数和优化器
4. 训练神经网络并更新网络参数
5. 测试网络

3.3.5 定义损失函数

PyTorch在`nn`模块中提供了各种损失函数，这些损失函数可看作是一种特殊的层，PyTorch也将这些损失函数实现为`nn.Module`的子类。我们现在使用它提供的均方误差损失作为模型的损失函数。

```
loss = nn.MSELoss()
```

```
loss_func = torch.nn.CrossEntropyLoss()
```

3.3.6 定义优化算法

同样，我们也无须自己实现小批量随机梯度下降算法。`torch.optim`模块提供了很多常用的优化算法比如SGD、Adam和RMSProp等。下面我们创建一个用于优化`net`所有参数的优化器实例，并指定学习率为0.03的小批量随机梯度下降（SGD）为优化算法。

```
import torch.optim as optim

optimizer = optim.SGD(net.parameters(), lr=0.03)
print(optimizer)
```

我们还可以为不同子网络设置不同的学习率，这在finetune时经常用到。例：

```
optimizer =optim.SGD([
                # 如果对某个参数不指定学习率，就使用最外层的默认学习率
                {'params': net.subnet1.parameters()}, # lr=0.03
                {'params': net.subnet2.parameters(), 'lr': 0.01}
            ], lr=0.03)
```

有时候我们不想让学习率固定成一个常数，那如何调整学习率呢？主要有两种做法。一种是修改 `optimizer.param_groups` 中对应的学习率，另一种是更简单也是较为推荐的做法——新建优化器，由于optimizer十分轻量级，构建开销很小，故而可以构建新的optimizer。但是后者对于使用动量的优化器（如Adam），会丢失动量等状态信息，可能会造成损失函数的收敛出现震荡等情况。

```
# 调整学习率
for param_group in optimizer.param_groups:
    param_group['lr'] *= 0.1 # 学习率为之前的0.1倍
```

```
test_data = torchvision.datasets.MNIST('./data',
                           transform = trans, train=False, download=True)

train_data_loader = torch.utils.data.DataLoader(train_data,
                                                batch_size=200,
                                                shuffle=True,
                                                num_workers=4)

test_data_loader = torch.utils.data.DataLoader(test_data,
                                               batch_size=200,
                                               shuffle=False,
                                               num_workers=4)


net = FcNet3()

loss_func = torch.nn.CrossEntropyLoss()

optimizer = torch.optim.SGD(net.parameters(), lr=0.1)

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


def test():
    test_acc_sum = 0.0
    with torch.no_grad():
        for images, labels in test_data_loader:

            output = net(images)
```

学习率一般就是0.1或者0.01，看具体效果

按照10的幂往小调，尝试一下，看结果

15,1 17%

```
8
 1      for images, labels in train_data_loader:
 2
 3          images.to(device)
 4          labels.to(device)
 5
 6          #print(images.shape, labels)
 7
 8          #forward
 9          output = net(images)
10
11          loss = loss_func(output, labels)
12
13          #gradient clear
14          optimizer.zero_grad()
15
16          #backward
17          loss.backward()
18
19          #update weight
20          optimizer.step()
21
22          #train_loss_sum += loss.item()
23          #GPU
24          #item()
25          train_loss_sum += loss.cpu().item()
26
27          #
                                                    58,0-1          71%
```

注意：images的shape

```
        #update weight
        optimizer.step()

        #train_loss_sum += loss.item()
        #GPU
        #item() convert Tersor to Python number
        #
        train_loss_sum += loss.cpu().item()

        #print(output.argmax(dim=1) == labels)
        #print((output.argmax(dim=1) == labels).float())

        train_acc_sum += (output.argmax(dim=1) == labels).float().sum().cpu().item()
        batch_count += 1

    #print(train_acc_sum)
    train_acc = train_acc_sum / len(train_data)
    batch_loss = train_loss_sum / batch_count

    test_acc = test()
    print("epoch %d, loss %.4f, train accuracy %.3f, test accuracy %.3f" %
          (curr_epoch, batch_loss, train_acc, test_acc))


def main():

    epoch = 30
    for e in range(epoch):
        training(e)
                                                              105,9         93%
```

3.6.6 计算分类准确率

给定一个类别的预测概率分布`y_hat`，我们把预测概率最大的类别作为输出类别。如果它与真实类别`y`一致，说明这次预测是正确的。分类准确率即正确预测数量与总预测数量之比。

为了演示准确率的计算，下面定义准确率`accuracy`函数。其中`y_hat.argmax(dim=1)`返回矩阵`y_hat`每行中最大元素的索引，且返回结果与变量`y`形状相同。相等条件判断式`(y_hat.argmax(dim=1) == y)`是一个类型为`ByteTensor`的`Tensor`，我们用`float()`将其转换为值为0（相等为假）或1（相等为真）的浮点型`Tensor`。

```
def accuracy(y_hat, y):
    return (y_hat.argmax(dim=1) == y).float().mean().item()
```

让我们继续使用在演示`gather`函数时定义的变量`y_hat`和`y`，并将它们分别作为预测概率分布和标签。可以看到，第一个样本预测类别为2（该行最大元素0.6在本行的索引为2），与真实标签0不一致；第二个样本预测类别为2（该行最大元素0.5在本行的索引为2），与真实标签2一致。因此，这两个样本上的分类准确率为0.5。

```
print(accuracy(y_hat, y))
```

输出：

```
        1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 0., 0., 1., 1., 1.,
        1., 1., 1., 0., 1., 1., 1., 0., 1., 1., 1., 0., 1., 1., 1., 1., 1.,
        0., 1., 0., 1., 1., 1., 0., 1., 1., 1., 0., 1., 1., 0., 1., 1., 0., 1.,
        1., 0.])
tensor([1., 1., 0., 0., 1., 1., 0., 1., 1., 1., 0., 1., 1., 0., 1., 1., 1., 1.,
        1., 1., 1., 0., 0., 1., 1., 1., 1., 1., 1., 0., 1., 0., 0., 0., 1., 1.,
        1., 0., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 0., 1., 1., 1., 1.,
        1., 1., 0., 0., 0., 1., 1., 1., 1., 1., 1., 1., 1., 1., 0., 1., 0.,
        1., 1., 1., 1., 1., 1., 0., 1., 1., 0., 1., 1., 1., 1., 1., 1., 1., 0.,
        1., 1., 1., 0., 0., 1., 0., 1., 1., 1., 1., 1., 0., 1., 1., 1., 0., 1.,
        0., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 0., 0., 1., 1., 1., 0., 0.,
        1., 1., 1., 1., 1., 1., 0., 1., 0., 1., 1., 1., 1., 1., 1., 1., 0., 1.,
        1., 1., 1., 0., 1., 1., 1., 1., 0., 1., 1., 1., 1., 1., 0., 1., 1., 0.,
        1., 1., 1., 1., 1., 0., 1., 1., 1., 1., 1., 1., 1., 0., 1., 1., 1., 0.,
        1., 0., 1., 0., 0., 0., 1., 1., 1., 1., 1., 1., 1., 0., 1., 0., 1., 0.,
        1., 1.])
tensor([1., 1., 1., 1., 1., 0., 1., 1., 1., 0., 1., 0., 0., 1., 1., 1., 1., 1.,
        1., 1., 1., 0., 0., 1., 0., 1., 1., 1., 0., 0., 0., 0., 1., 0., 1., 1.,
        1., 1., 0., 1., 1., 0., 1., 1., 1., 0., 0., 1., 1., 1., 1., 1., 1., 1.,
        1., 1., 1., 1., 1., 1., 1., 1., 1., 0., 1., 1., 1., 1., 1., 0., 1., 1.,
        0., 1., 1., 0., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 0., 1.,
        1., 1., 0., 1., 1., 1., 1., 0., 1., 1., 0., 1., 1., 1., 1., 0., 1., 1.,
        1., 1., 1., 1., 0., 1., 0., 1., 0., 1., 1., 1., 1., 0., 0., 1., 1., 0.,
        1., 1., 1., 1., 1., 1., 1., 0., 1., 1., 0., 1., 1., 1., 0., 1., 1., 1.,
        1., 0., 1., 1., 1., 0., 0., 0., 1., 1., 0., 1., 1., 1., 1., 1., 1., 1.,
        1., 1., 1., 0., 1., 0., 1., 0., 1., 0., 0., 0., 0., 1., 1., 1., 1., 1.,
        0., 1., 0., 1., 1., 1., 1., 0., 1., 1., 0., 1., 1., 1., 0., 1., 1., 1.,
        0., 1.])
```

```
net = FcNet3()

loss_func = torch.nn.CrossEntropyLoss()

optimizer = torch.optim.SGD(net.parameters(), lr=0.1)

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


def test():
    test_acc_sum = 0.0
    with torch.no_grad():
        for images, labels in test_data_loader:

            output = net(images)

            test_acc_sum += (output.argmax(dim=1) == labels).sum().cpu().item()

        acc = test_acc_sum / len(test_data)
    return acc
```

```
        #
        train_loss_sum += loss.cpu().item()

        #print(output.argmax(dim=1) == labels)
        #print((output.argmax(dim=1) == labels).float())

        train_acc_sum += (output.argmax(dim=1) == labels).float().sum().cpu().item()
        batch_count += 1

    #print(train_acc_sum)
    train_acc = train_acc_sum / len(train_data)
    batch_loss = train_loss_sum / batch_count

    test_acc = test()
    print("epoch %d, loss %.4f, train accuracy %.3f, test accuracy %.3f" %
          (curr_epoch, batch_loss, train_acc, test_acc))


def main():

    epoch = 30
    for e in range(epoch):
        training(e)

    #save model weight
    torch.save(net.state_dict(), 'params.pkl')
```

2. 保存和加载整个模型

保存：

```
torch.save(model, PATH)
```

加载：

```
model = torch.load(PATH)
```

我们采用推荐的方法一来实验一下：

```
X = torch.randn(2, 3)
Y = net(X)

PATH = "./net.pt"
torch.save(net.state_dict(), PATH)

net2 = MLP()
net2.load_state_dict(torch.load(PATH))
Y2 = net2(X)
Y2 == Y
```

课后作业

网络结构784-32-16-10
1 epoch：94.31%
2 epoch：94.1%
3 epoch：95.2%
……
10 epoch：96.2%

网络结构784-128-32-10
1 epoch：95.6%
2 epoch：96.6%
3 epoch：95.9%
……
10 epoch：97.9%