3.3 线性回归的简洁实现

随着深度学习框架的发展，开发深度学习应用变得越来越便利。实践中，我们通常可以用比上一节更简洁的代码来实现同样的模型。在本节中，我们将介绍如何使用PyTorch更方便地实现线性回归的训练。

3.3.1 生成数据集

我们生成与上一节中相同的数据集。其中 `features` 是训练数据特征， `labels` 是标签。

```
num_inputs = 2
num_examples = 1000
true_w = [2, -3.4]
true_b = 4.2
features = torch.tensor(np.random.normal(0, 1, (num_examples, num_inputs)), dtype=tor
labels = true_w[0] * features[:, 0] + true_w[1] * features[:, 1] + true_b
labels += torch.tensor(np.random.normal(0, 0.01, size=labels.size()), dtype=torch.flo
```

3.3.2 读取数据

PyTorch提供了 `data` 包来读取数据。由于 `data` 常用作变量名，我们将导入的 `data` 模块用 `Data` 代替。在每一次迭代中，我们将随机读取包含10个数据样本的小批量。

```
import torch.utils.data as Data

batch_size = 10
# 将训练数据的特征和标签组合
dataset = Data.TensorDataset(features, labels)
# 随机读取小批量
data_iter = Data.DataLoader(dataset, batch_size, shuffle=True)
```

这里 `data_iter` 的使用跟上一节中的一样。让我们读取并打印第一个小批量数据样本。

```
for X, y in data_iter:
    print(X, y)
```

```
        break
```

输出：

```
tensor([[-2.7723, -0.6627],
        [-1.1058,  0.7688],
        [ 0.4901, -1.2260],
        [-0.7227, -0.2664],
        [-0.3390,  0.1162],
        [ 1.6705, -2.7930],
        [ 0.2576, -0.2928],
        [ 2.0475, -2.7440],
        [ 1.0685,  1.1920],
        [ 1.0996,  0.5106]])
 tensor([ 0.9066, -0.6247,  9.3383,  3.6537,  3.1283, 17.0213,  5.6953, 17.6279,
         2.2809,  4.6661])
```

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
```

输出：

```
LinearNet(
  (linear): Linear(in_features=2, out_features=1, bias=True)
)
```

事实上我们还可以用 `nn.Sequential` 来更加方便地搭建网络， `Sequential` 是一个有序的容器，网络层将按照在传入 `Sequential` 的顺序依次被添加到计算图中。

```
# 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])
```

输出：

```
Sequential(
  (linear): Linear(in_features=2, out_features=1, bias=True)
)
Linear(in_features=2, out_features=1, bias=True)
```

可以通过 `net.parameters()` 来查看模型所有的可学习参数，此函数将返回一个生成器。

```
for param in net.parameters():
    print(param)
```

输出：

```
Parameter containing:
tensor([[-0.0277,  0.2771]], requires_grad=True)
Parameter containing:
tensor([0.3395], requires_grad=True)
```

回顾图3.1中线性回归在神经网络图中的表示。作为一个单层神经网络，线性回归输出层中的神经元和输入层中各个输入完全连接。因此，线性回归的输出层又叫全连接层。

> **注意： `torch.nn` 仅支持输入一个batch的样本不支持单个样本输入，如果只有单个样本，可使用 `input.unsqueeze(0)` 来添加一维。**

3.3.4 初始化模型参数

在使用 `net` 前，我们需要初始化模型参数，如线性回归模型中的权重和偏差。PyTorch在 `init` 模块中提供了多种参数初始化方法。这里的 `init` 是 `initializer` 的缩写形式。我们通过 `init.normal_` 将权重参数每个元素初始化为随机采样于均值为0、标准差为0.01的正态分布。偏差会初始化为零。

```
from torch.nn import init

init.normal_(net[0].weight, mean=0, std=0.01)
init.constant_(net[0].bias, val=0)  # 也可以直接修改bias的data: net[0].bias.data.fill_
```

> **注：如果这里的 `net` 是用3.3.3节一开始的代码自定义的，那么上面代码会报错， `net[0].weight` 应改为 `net.linear.weight` ， `bias` 亦然。因为 `net[0]` 这样根据下标访问子模块的写法只有当 `net` 是个 `ModuleList` 或者 `Sequential` 实例时才可以，详见4.1节。**

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()
```

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)
```

输出：

```
SGD (
Parameter Group 0
    dampening: 0
    lr: 0.03
    momentum: 0
    nesterov: False
    weight_decay: 0
)
```

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
```

3.3.7 训练模型

在使用Gluon训练模型时，我们通过调用 `optim` 实例的 `step` 函数来迭代模型参数。按照小批量随机梯度下降的定义，我们在 `step` 函数中指明批量大小，从而对批量中样本梯度求平均。

```
num_epochs = 3
for epoch in range(1, num_epochs + 1):
    for X, y in data_iter:
        output = net(X)
        l = loss(output, y.view(-1, 1))
        optimizer.zero_grad() # 梯度清零，等价于net.zero_grad()
        l.backward()
        optimizer.step()
    print('epoch %d, loss: %f' % (epoch, l.item()))
```

输出：

```
epoch 1, loss: 0.000457
epoch 2, loss: 0.000081
epoch 3, loss: 0.000198
```

下面我们分别比较学到的模型参数和真实的模型参数。我们从 `net` 获得需要的层，并访问其权重（ `weight` ）和偏差（ `bias` ）。学到的参数和真实的参数很接近。

```
dense = net[0]
print(true_w, dense.weight)
print(true_b, dense.bias)
```

输出：

```
[2, -3.4] tensor([[ 1.9999, -3.4005]])
4.2 tensor([4.2011])
```

小结

- 使用PyTorch可以更简洁地实现模型。
- `torch.utils.data` 模块提供了有关数据处理的工具， `torch.nn` 模块定义了大量神经网络的层，`torch.nn.init` 模块定义了各种初始化方法， `torch.optim` 模块提供了很多常用的优化算法。