

PyTorch

PyTorch是人工智能方面重要的工具之一，但是它的最本质功能就是自动求解梯度，因为求解梯度是绝大多数数值优化的重要组成部分。换言之，PyTorch不仅可以用于神经网络，而可以用于几乎所有的（具有梯度的）优化问题。下面我们就来看一个例子（本节以代码为主）。

举例：还是接上文的例子，我们现在有数据
$$\begin{aligned} &\mathbf{x}_1=[0.2, 0.5, 0.7]^{\text{top}} \ \&\mathbf{y}_1=[0.8, 0.8]^{\text{top}} \ \&\mathbf{x}_2=[0.4, 0.6, 0.8]^{\text{top}} \\ &\&\mathbf{y}_2=[0.7, 0.9]^{\text{top}} \ \&\mathbf{x}_3=[0.3, 0.6, 0.9]^{\text{top}} \\ &\&\mathbf{y}_3=[0.9, 0.9]^{\text{top}} \end{aligned}$$

调用PyTorch包

In [1]:

```
import torch
```

输入数据

绝大多数情况下数据不需要手动输入，它们往往会在数据采集的时候自动记录下来。这里我们用PyTorch自带的数据类型 `torch.tensor`，这个类型也可以从其他数据类型转换过来。

强调：在机器学习里，多个训练数据往往拼接到一起，而不是逐个计算，并且数据的维度往往是 $E \times M$ ，也就是每一行是一个数据，每一列是一个特征。后面我们会把数据对应这个规则来输入。

In [2]:

```
X = torch.tensor([0.2, 0.5, 0.7, 0.4, 0.6, 0.8, 0.3, 0.6, 0.9]).view(3,3)
X
```

Out[2]:

```
tensor([[0.2000, 0.5000, 0.7000],
        [0.4000, 0.6000, 0.8000],
        [0.3000, 0.6000, 0.9000]])
```

In [3]:

```
Y = torch.tensor([0.8, 0.8, 0.7, 0.9, 0.9, 0.9]).view(3,2)
Y
```

Out[3]:

```
tensor([[0.8000, 0.8000],
        [0.7000, 0.9000],
        [0.9000, 0.9000]])
```

定义参数

如上一节所示，要优化参数，就需要计算参数的梯度。在PyTorch定义tensor的时候加上 `requires_grad=True` 即可。

由于数据维度的要求，我们让 W 是一个 $M \times N$ 的矩阵，并且让 $\mathbf{y} = \mathbf{x} \cdot W$ ，这样 \mathbf{y} 的维度就是 $E \times M \cdot M \times N = E \times N$ 了。

In [4]:

```
W = torch.randn([3,2], requires_grad=True)
```

W

Out[4]:

```
tensor([[ 0.9407, -0.1843],
        [-0.5164, -1.0036],
        [ 0.7652,  0.0561]], requires_grad=True)
这里我们也可以看到 W 的最后标注了 requires_grad=True
```

定义模型的计算过程

$\hat{\mathbf{y}} = \mathbf{x} \cdot \mathbf{W}$

In [5]:

```
def f(w, x):
    return torch.matmul(x, w)
```

定义损失函数

$\mathcal{L}(\hat{\mathbf{y}}, \mathbf{y}) = \|\hat{\mathbf{y}} - \mathbf{y}\|_2^2$

In [6]:

```
def loss(yhat, y):
    return torch.norm(yhat-y)**2
```

利用 PyTorch 计算梯度

PyTorch 不能计算梯度的解析解，而是在每一个 \mathbf{W} 处计算梯度。因此需要

- 利用模型的参数 \mathbf{W} 和数据 \mathbf{X} 计算 \mathbf{y}_{hat}
- 利用 \mathbf{y}_{hat} 和数据 \mathbf{Y} 计算损失函数 \mathcal{L}
- 利用损失函数 \mathcal{L} 计算 \mathbf{W} 的梯度

可以看出，这个过程可以大致分成2部分：

- 前向传播，也就是从 $\mathbf{x} \rightarrow \hat{\mathbf{y}} \rightarrow \mathcal{L}$
- 反向传播，也就是 $\mathcal{L} \rightarrow \nabla_{\mathbf{w}}$

反向传播的代码是 `L.backward()`，求解的就是 $\nabla_{\mathbf{w}} \mathcal{L}$

查看参数的梯度使用的是 `W.grad`

In [7]:

```
# 正向传播
y_hat = f(W, X)
L = loss(y_hat, Y)

# 反向传播
L.backward()

# 显示梯度
print(W.grad)
```

```
tensor([[ -0.2943, -2.6488],
        [-0.6469, -4.9450],
        [-0.9326, -6.9814]])
```

要注意的是，反向传播 `.backward()` 所产生的梯度会不断累积：

In [8]:

```
print('再次反向传播')
loss(f(W, X), Y).backward()
print(W.grad)
```

```
print('再再次反向传播')
loss(f(W, X), Y).backward()
print(W.grad)
```

```
print('再再再次反向传播')
loss(f(W, X), Y).backward()
print(W.grad)
```

再次反向传播

```
tensor([[ -0.5886, -5.2975],
        [ -1.2937, -9.8900],
        [ -1.8652, -13.9628]])
```

再再次反向传播

```
tensor([[ -0.8828, -7.9463],
        [ -1.9406, -14.8350],
        [ -2.7978, -20.9441]])
```

再再再次反向传播

```
tensor([[ -1.1771, -10.5951],
        [ -2.5875, -19.7801],
        [ -3.7303, -27.9255]])
```

这一点有利有弊。对于刚接触PyTorch的人来说，忘记及时清除已有梯度，导致梯度错误，并且不断累积。但是这个性质也使得许多操作成为可能（有些人可能不会接触到）。

那么要清除现有梯度就需要 `W.grad.zero_()`

这样我们再来测试一次：

In [9]:

```
print('再次反向传播')
loss(f(W, X), Y).backward()
print(W.grad)
W.grad.zero_()
```

```
print('再再次反向传播')
loss(f(W, X), Y).backward()
print(W.grad)
W.grad.zero_()
```

```
print('再再再次反向传播')
loss(f(W, X), Y).backward()
print(W.grad)
W.grad.zero_()
```

```
print('梯度不再累积')
```

再次反向传播

```
tensor([[ -1.4714, -13.2439],
        [ -3.2344, -24.7251],
        [ -4.6629, -34.9069]])
```

再再次反向传播

```
tensor([[ -0.2943, -2.6488],
        [ -0.6469, -4.9450],
        [ -0.9326, -6.9814]])
```

再再再次反向传播

```
tensor([[ -0.2943, -2.6488],
        [ -0.6469, -4.9450],
        [ -0.9326, -6.9814]])
```

梯度不再累积

可以看到，这样梯度就不会累积了。这样我们就可以在每一个点计算新的梯度，然后更新参数。那么整个优化过程就变成了：

In [10]:

```
import torch
```

```
# 输入/读取数据
```

```
X = torch.tensor([0.2, 0.5, 0.7, 0.4, 0.6, 0.8, 0.3, 0.6, 0.9]).view(3,3)
```

```
Y = torch.tensor([0.8, 0.8, 0.7, 0.9, 0.9, 0.9]).view(3,2)
```

```
# 定义模型，损失函数，学习率
```

```
W = torch.randn([3,2], requires_grad=True)
```

```
def f(w, x):
```

```
    return torch.matmul(x, w)
```

```
def loss(yhat, y):
```

```
    return torch.norm(yhat-y)**2
```

```
alpha = 0.25
```

```
# 优化
```

```
for i in range(21001):
```

```
    # 正向传播
```

```
    y_hat = f(W, X)
```

```
    L = loss(y_hat, Y)
```

```
    # 反向传播
```

```
    L.backward()
```

```
    # 更新参数
```

```
    W.data = W.data - alpha * W.grad
```

```
    # 重置梯度
```

```
    W.grad.zero_()
```

```
    # 显示参数变化
```

```
    if not i%3000:
```

```
        print(f'第{i}次迭代后的参数为：\n{torch.round(W.data*100)/100}')
```

第0次迭代后的参数为：
tensor([[2.5600, 0.8600],
 [-0.7600, 0.0900],
 [0.5500, 2.1400]])
第3000次迭代后的参数为：
tensor([[-1.3000, -0.2400],
 [0.8100, 1.6400],
 [0.9100, 0.0100]])
第6000次迭代后的参数为：
tensor([[-1.4400, -0.4300],
 [1.3100, 2.2600],
 [0.6200, -0.3600]])
第9000次迭代后的参数为：
tensor([[-1.4800, -0.4800],
 [1.4400, 2.4300],
 [0.5300, -0.4600]])
第12000次迭代后的参数为：
tensor([[-1.5000, -0.4900],
 [1.4800, 2.4800],
 [0.5100, -0.4900]])
第15000次迭代后的参数为：
tensor([[-1.5000, -0.5000],
 [1.5000, 2.4900],
 [0.5000, -0.5000]])
第18000次迭代后的参数为：
tensor([[-1.5000, -0.5000],
 [1.5000, 2.5000],
 [0.5000, -0.5000]])
第21000次迭代后的参数为：
tensor([[-1.5000, -0.5000],
 [1.5000, 2.5000],
 [0.5000, -0.5000]])
得到的结果和之前的结果是一样的。

总结

从最后一个代码块可以看出，利用PyTorch，我们只需要定义前向传播的函数（模型）以及损失函数，梯度就可以自动被计算出来并且用于梯度下降，完全不需要任何公式推导。

讨论

关于 torch.tensor

可以看出，在代码中 `W` 是一个 `tensor` 的数据结构，他主要包含2个部分

- `W.data`，表示了它的数值
- `W.grad`，储存了它的梯度 也就是说，`tensor` 可以看做是一个容器，它存了很多东西在里面。

关于优化

本章节只用到了 `tensor.backward()` 用来计算参数的梯度。其实PyTorch还提供大量的功能，例如定义模型、参数更新等功能。利用这些功能，代码可以更加简洁高效。后面会继续讲解。

