2.3 自动求梯度

在深度学习中，我们经常需要对函数求梯度（gradient）。PyTorch提供的**autograd**包能够根据输入和前向传播过程自动构建计算图，并执行反向传播。本节将介绍如何使用autograd包来进行自动求梯度的有关操作。

2.3.1 概念

上一节介绍的 `Tensor` 是这个包的核心类，如果将其属性 `.requires_grad` 设置为 `True` ，它将开始追踪(track)在其上的所有操作（这样就可以利用链式法则进行梯度传播了）。完成计算后，可以调用 `.backward()` 来完成所有梯度计算。此 `Tensor` 的梯度将累积到 `.grad` 属性中。

> **注意在 `y.backward()` 时，如果 `y` 是标量，则不需要为 `backward()` 传入任何参数；否则，需要传入一个与 `y` 同形的 `Tensor` 。解释见 2.3.2 节。**

如果不想要被继续追踪，可以调用 `.detach()` 将其从追踪记录中分离出来，这样就可以防止将来的计算被追踪，这样梯度就传不过去了。此外，还可以用 `with torch.no_grad()` 将不想被追踪的操作代码块包裹起来，这种方法在评估模型的时候很常用，因为在评估模型时，我们并不需要计算可训练参数（ `requires_grad=True` ）的梯度。

`Function` 是另外一个很重要的类。 `Tensor` 和 `Function` 互相结合就可以构建一个记录有整个计算过程的有向无环图（DAG）。每个 `Tensor` 都有一个 `.grad_fn` 属性，该属性即创建该 `Tensor` 的 `Function` ，就是说该 `Tensor` 是不是通过某些运算得到的，若是，则 `grad_fn` 返回一个与这些运算相关的对象，否则是None。

下面通过一些例子来理解这些概念。

2.3.2 `Tensor`

创建一个 `Tensor` 并设置 `requires_grad=True` ：

```
x = torch.ones(2, 2, requires_grad=True)
print(x)
print(x.grad_fn)
```

输出：

```
tensor([[1., 1.],
        [1., 1.]], requires_grad=True)
None
```

再做一下运算操作：

```
y = x + 2
print(y)
print(y.grad_fn)
```

输出：

```
tensor([[3., 3.],
        [3., 3.]], grad_fn=<AddBackward>)
<AddBackward object at 0x1100477b8>
```

注意x是直接创建的，所以它没有 `grad_fn` ，而y是通过一个加法操作创建的，所以它有一个为 `<AddBackward>` 的 `grad_fn` 。

像x这种直接创建的称为叶子节点，叶子节点对应的 `grad_fn` 是 `None` 。

```
print(x.is_leaf, y.is_leaf) # True False
```

再来点复杂度运算操作：

```
z = y * y * 3
out = z.mean()
print(z, out)
```

输出：

```
tensor([[27., 27.],
        [27., 27.]], grad_fn=<MulBackward>) tensor(27., grad_fn=<MeanBackward1>)
```

通过 `.requires_grad_()` 来用in-place的方式改变 `requires_grad` 属性：

```
a = torch.randn(2, 2) # 缺失情况下默认 requires_grad = False
a = ((a * 3) / (a - 1))
print(a.requires_grad) # False
a.requires_grad_(True)
print(a.requires_grad) # True
b = (a * a).sum()
print(b.grad_fn)
```

输出：

```
False
True
<SumBackward0 object at 0x118f50cc0>
```

2.3.3 梯度

因为 `out` 是一个标量，所以调用 `backward()` 时不需要指定求导变量：

```
out.backward() # 等价于 out.backward(torch.tensor(1.))
```

我们来看看 `out` 关于 `x` 的梯度 $\frac{d(out)}{dx}$：

```
print(x.grad)
```

输出：

```
tensor([[4.5000, 4.5000],
        [4.5000, 4.5000]])
```

我们令 `out` 为 o, 因为

$$o = \frac{1}{4}\sum_{i=1}^{4} z_i = \frac{1}{4}\sum_{i=1}^{4} 3(x_i+2)^2$$

所以

$$\frac{\partial o}{\partial x_i}\Big|_{x_i=1} = \frac{9}{2} = 4.5$$

所以上面的输出是正确的。

数学上，如果有一个函数值和自变量都为向量的函数 $\vec{y} = f(\vec{x})$, 那么 $\vec{y}$ 关于 $\vec{x}$ 的梯度就是一个雅可比矩阵（Jacobian matrix）：

$$J = \begin{pmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_m}{\partial x_1} & \cdots & \frac{\partial y_m}{\partial x_n} \end{pmatrix}$$

而 `torch.autograd` 这个包就是用来计算一些雅克比矩阵的乘积的。例如，如果 v 是一个标量函数的 $l = g(\vec{y})$ 的梯度：

$$v = \begin{pmatrix} \frac{\partial l}{\partial y_1} & \cdots & \frac{\partial l}{\partial y_m} \end{pmatrix}$$

那么根据链式法则我们有 l 关于 $\vec{x}$ 的雅克比矩阵就为:

$$vJ = \begin{pmatrix} \frac{\partial l}{\partial y_1} & \cdots & \frac{\partial l}{\partial y_m} \end{pmatrix} \begin{pmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_m}{\partial x_1} & \cdots & \frac{\partial y_m}{\partial x_n} \end{pmatrix} = \begin{pmatrix} \frac{\partial l}{\partial x_1} & \cdots & \frac{\partial l}{\partial x_n} \end{pmatrix}$$

注意：grad在反向传播过程中是累加的(accumulated)，这意味着每一次运行反向传播，梯度都会累加之前的梯度，所以一般在反向传播之前需把梯度清零。

```
# 再来反向传播一次，注意grad是累加的
out2 = x.sum()
out2.backward()
print(x.grad)

out3 = x.sum()
x.grad.data.zero_()
out3.backward()
print(x.grad)
```

输出：

```
tensor([[5.5000, 5.5000],
        [5.5000, 5.5000]])
tensor([[1., 1.],
        [1., 1.]])
```

> **现在我们解释2.3.1节留下的问题，为什么在 `y.backward()` 时，如果 `y` 是标量，则不需要为 `backward()` 传入任何参数；否则，需要传入一个与 `y` 同形的 `Tensor` ？简单来说就是为了避免向量（甚至更高维张量）对张量求导，而转换成标量对张量求导。举个例子，假设形状为 `m x n` 的矩阵 X 经过运算得到了 `p x q` 的矩阵 Y，Y 又经过运算得到了 `s x t` 的矩阵 Z。那么按照前面讲的规则，dZ/dY 应该是一个 `s x t x p x q` 四维张量，dY/dX 是一个 `p x q x m x n` 的四维张量。问题来了，怎样反向传播？怎样将两个四维张量相乘？？？这要怎么乘？？？就算能解决两个四维张量怎么乘的问题，四维和三维的张量又怎么乘？导数的导数又怎么求，这一连串的问题，感觉要疯掉……为了避免这个问题，我们不允许张量对张量求导，只允许标量对张量求导，求导结果是和自变量同形的张量。所以必要时我们要把张量通过将所有张量的元素加权求和的方式转换为标量，举个例子，假设 `y` 由自变量 `x` 计算而来， `w` 是和 `y` 同形的张量，则 `y.backward(w)` 的含义是：先计算 `l = torch.sum(y * w)` ，则 `l` 是个标量，然后求 `l` 对自变量 `x` 的导数。 参考**

来看一些实际例子。

```
x = torch.tensor([1.0, 2.0, 3.0, 4.0], requires_grad=True)
y = 2 * x
z = y.view(2, 2)
print(z)
```

输出：

```
tensor([[2., 4.],
        [6., 8.]], grad_fn=<ViewBackward>)
```

现在 `z` 不是一个标量，所以在调用 `backward` 时需要传入一个和 `z` 同形的权重向量进行加权求和得到一个标量。

```
v = torch.tensor([[1.0, 0.1], [0.01, 0.001]], dtype=torch.float)
z.backward(v)
print(x.grad)
```

输出：

```
tensor([2.0000, 0.2000, 0.0200, 0.0020])
```

注意， `x.grad` 是和 `x` 同形的张量。

再来看看中断梯度追踪的例子：

```
x = torch.tensor(1.0, requires_grad=True)
y1 = x ** 2
with torch.no_grad():
    y2 = x ** 3
y3 = y1 + y2

print(x.requires_grad)
print(y1, y1.requires_grad) # True
print(y2, y2.requires_grad) # False
print(y3, y3.requires_grad) # True
```

输出：

```
True
tensor(1., grad_fn=<PowBackward0>) True
tensor(1.) False
tensor(2., grad_fn=<ThAddBackward>) True
```

可以看到，上面的 `y2` 是没有 `grad_fn` 而且 `y2.requires_grad=False` 的，而 `y3` 是有 `grad_fn` 的。如果我们将 `y3` 对 `x` 求梯度的话会是多少呢？

```
y3.backward()
print(x.grad)
```

输出：

```
tensor(2.)
```

为什么是2呢？$y_3 = y_1 + y_2 = x^2 + x^3$，当 $x=1$ 时 $\frac{dy_3}{dx}$ 不应该是5吗？事实上，由于 y_2 的定义是被 `torch.no_grad():` 包裹的，所以与 y_2 有关的梯度是不会回传的，只有与 y_1 有关的梯度才会回传，即 x^2 对 x 的梯度。

上面提到， `y2.requires_grad=False` ，所以不能调用 `y2.backward()` ，会报错：

```
RuntimeError: element 0 of tensors does not require grad and does not have a grad_fn
```

此外，如果我们想要修改 `tensor` 的数值，但是又不希望被 `autograd` 记录（即不会影响反向传播），那么我么可以对 `tensor.data` 进行操作。

```
x = torch.ones(1,requires_grad=True)

print(x.data) # 还是一个tensor
print(x.data.requires_grad) # 但是已经是独立于计算图之外

y = 2 * x
x.data *= 100 # 只改变了值，不会记录在计算图，所以不会影响梯度传播

y.backward()
print(x) # 更改data的值也会影响tensor的值
print(x.grad)
```

输出：

```
tensor([1.])
False
tensor([100.], requires_grad=True)
tensor([2.])
```