自动求导

这次课程我们会了解 PyTorch 中的自动求导机制，自动求导是 PyTorch 中非常重要的特性，能够让我们避免手动去计算非常复杂的导数，这能够极大地减少了我们构建模型的时间，这也是其前身 Torch 这个框架所不具备的特性，下面我们通过例子看看 PyTorch 自动求导的独特魅力以及探究自动求导的更多用法。

```
import torch
from torch.autograd import Variable
```

简单情况的自动求导

下面我们显示一些简单情况的自动求导，"简单"体现在计算的结果都是标量，也就是一个数，我们对这个标量进行自动求导。

```
x = Variable(torch.Tensor([2]), requires_grad=True)
y = x + 2
z = y ** 2 + 3
print(z)
```

```
Variable containing:
 19
[torch.FloatTensor of size 1]
```

通过上面的一些列操作，我们从 x 得到了最后的结果out，我们可以将其表示为数学公式

$$z = (x + 2)^2 + 3 \tag{1}$$

那么我们从 z 对 x 求导的结果就是

$$\frac{\partial z}{\partial x} = 2(x + 2) = 2(2 + 2) = 8 \tag{2}$$

如果你对求导不熟悉，可以查看以下[网址进行复习]

```
# 使用自动求导
z.backward()
print(x.grad)
```

```
Variable containing:
 8
[torch.FloatTensor of size 1]
```

对于上面这样一个简单的例子，我们验证了自动求导，同时可以发现发现使用自动求导非常方便。如果是一个更加复杂的例子，那么手动求导就会显得非常的麻烦，所以自动求导的机制能够帮助我们省去麻烦的数学计算，下面我们可以看一个更加复杂的例子。

```
x = Variable(torch.randn(10, 20), requires_grad=True)
y = Variable(torch.randn(10, 5), requires_grad=True)
w = Variable(torch.randn(20, 5), requires_grad=True)

out = torch.mean(y - torch.matmul(x, w)) # torch.matmul 是做矩阵乘法
out.backward()
```

如果你对矩阵乘法不熟悉，可以查看下面的[网址进行复习]

```
# 得到 x 的梯度
print(x.grad)
```

```
Variable containing:

Columns 0 to 9
-0.0600 -0.0242 -0.0514  0.0882  0.0056 -0.0400 -0.0300 -0.0052 -0.0289 -0.0172
-0.0600 -0.0242 -0.0514  0.0882  0.0056 -0.0400 -0.0300 -0.0052 -0.0289 -0.0172
-0.0600 -0.0242 -0.0514  0.0882  0.0056 -0.0400 -0.0300 -0.0052 -0.0289 -0.0172
-0.0600 -0.0242 -0.0514  0.0882  0.0056 -0.0400 -0.0300 -0.0052 -0.0289 -0.0172
-0.0600 -0.0242 -0.0514  0.0882  0.0056 -0.0400 -0.0300 -0.0052 -0.0289 -0.0172
-0.0600 -0.0242 -0.0514  0.0882  0.0056 -0.0400 -0.0300 -0.0052 -0.0289 -0.0172
-0.0600 -0.0242 -0.0514  0.0882  0.0056 -0.0400 -0.0300 -0.0052 -0.0289 -0.0172
-0.0600 -0.0242 -0.0514  0.0882  0.0056 -0.0400 -0.0300 -0.0052 -0.0289 -0.0172
-0.0600 -0.0242 -0.0514  0.0882  0.0056 -0.0400 -0.0300 -0.0052 -0.0289 -0.0172
-0.0600 -0.0242 -0.0514  0.0882  0.0056 -0.0400 -0.0300 -0.0052 -0.0289 -0.0172

Columns 10 to 19
-0.0372  0.0144 -0.1074 -0.0363 -0.0189  0.0209  0.0618  0.0435 -0.0591  0.0103
-0.0372  0.0144 -0.1074 -0.0363 -0.0189  0.0209  0.0618  0.0435 -0.0591  0.0103
-0.0372  0.0144 -0.1074 -0.0363 -0.0189  0.0209  0.0618  0.0435 -0.0591  0.0103
-0.0372  0.0144 -0.1074 -0.0363 -0.0189  0.0209  0.0618  0.0435 -0.0591  0.0103
-0.0372  0.0144 -0.1074 -0.0363 -0.0189  0.0209  0.0618  0.0435 -0.0591  0.0103
-0.0372  0.0144 -0.1074 -0.0363 -0.0189  0.0209  0.0618  0.0435 -0.0591  0.0103
-0.0372  0.0144 -0.1074 -0.0363 -0.0189  0.0209  0.0618  0.0435 -0.0591  0.0103
-0.0372  0.0144 -0.1074 -0.0363 -0.0189  0.0209  0.0618  0.0435 -0.0591  0.0103
```

```
-0.0372  0.0144 -0.1074 -0.0363 -0.0189  0.0209  0.0618  0.0435 -0.0591  0.0103
-0.0372  0.0144 -0.1074 -0.0363 -0.0189  0.0209  0.0618  0.0435 -0.0591  0.0103
[torch.FloatTensor of size 10x20]
```

```
# 得到 y 的的梯度
print(y.grad)
```

```
Variable containing:
1.00000e-02 *
  2.0000  2.0000  2.0000  2.0000  2.0000
  2.0000  2.0000  2.0000  2.0000  2.0000
  2.0000  2.0000  2.0000  2.0000  2.0000
  2.0000  2.0000  2.0000  2.0000  2.0000
  2.0000  2.0000  2.0000  2.0000  2.0000
  2.0000  2.0000  2.0000  2.0000  2.0000
  2.0000  2.0000  2.0000  2.0000  2.0000
  2.0000  2.0000  2.0000  2.0000  2.0000
  2.0000  2.0000  2.0000  2.0000  2.0000
  2.0000  2.0000  2.0000  2.0000  2.0000
[torch.FloatTensor of size 10x5]
```

```
# 得到 w 的梯度
print(w.grad)
```

```
Variable containing:
 0.1342  0.1342  0.1342  0.1342  0.1342
 0.0507  0.0507  0.0507  0.0507  0.0507
 0.0328  0.0328  0.0328  0.0328  0.0328
-0.0086 -0.0086 -0.0086 -0.0086 -0.0086
 0.0734  0.0734  0.0734  0.0734  0.0734
-0.0042 -0.0042 -0.0042 -0.0042 -0.0042
 0.0078  0.0078  0.0078  0.0078  0.0078
-0.0769 -0.0769 -0.0769 -0.0769 -0.0769
 0.0672  0.0672  0.0672  0.0672  0.0672
 0.1614  0.1614  0.1614  0.1614  0.1614
-0.0042 -0.0042 -0.0042 -0.0042 -0.0042
-0.0970 -0.0970 -0.0970 -0.0970 -0.0970
-0.0364 -0.0364 -0.0364 -0.0364 -0.0364
```

```
-0.0419 -0.0419 -0.0419 -0.0419 -0.0419
 0.0134  0.0134  0.0134  0.0134  0.0134
-0.0251 -0.0251 -0.0251 -0.0251 -0.0251
 0.0586  0.0586  0.0586  0.0586  0.0586
-0.0050 -0.0050 -0.0050 -0.0050 -0.0050
 0.1125  0.1125  0.1125  0.1125  0.1125
-0.0096 -0.0096 -0.0096 -0.0096 -0.0096
[torch.FloatTensor of size 20x5]
```

上面数学公式就更加复杂，矩阵乘法之后对两个矩阵对应元素相乘，然后所有元素求平均，有兴趣的同学可以手动去计算一下梯度，使用 PyTorch 的自动求导，我们能够非常容易得到 x, y 和 w 的导数，因为深度学习中充满大量的矩阵运算，所以我们没有办法手动去求这些导数，有了自动求导能够非常方便地解决网络更新的问题。

复杂情况的自动求导

上面我们展示了简单情况下的自动求导，都是对标量进行自动求导，可能你会有一个疑问，如何对一个向量或者矩阵自动求导了呢？感兴趣的同学可以自己先去尝试一下，下面我们会介绍对多维数组的自动求导机制。

```
m = Variable(torch.FloatTensor([[2, 3]]), requires_grad=True) # 构建一个 1 x 2 的矩阵
n = Variable(torch.zeros(1, 2)) # 构建一个相同大小的 0 矩阵
print(m)
print(n)
```

```
Variable containing:
 2  3
[torch.FloatTensor of size 1x2]

Variable containing:
 0  0
[torch.FloatTensor of size 1x2]
```

```
# 通过 m 中的值计算新的 n 中的值
n[0, 0] = m[0, 0] ** 2
n[0, 1] = m[0, 1] ** 3
print(n)
```

```
Variable containing:
  4  27
[torch.FloatTensor of size 1x2]
```

将上面的式子写成数学公式，可以得到

$$n = (n_0,\ n_1) = (m_0^2,\ m_1^3) = (2^2,\ 3^3) \tag{3}$$

下面我们直接对 n 进行反向传播，也就是求 n 对 m 的导数。

这时我们需要明确这个导数的定义，即如何定义

$$\frac{\partial n}{\partial m} = \frac{\partial (n_0,\ n_1)}{\partial (m_0,\ m_1)} \tag{4}$$

在 PyTorch 中，如果要调用自动求导，需要往 `backward()` 中传入一个参数，这个参数的形状和 n 一样大，比如是 $(w_0,\ w_1)$，那么自动求导的结果就是：

$$\frac{\partial n}{\partial m_0} = w_0 \frac{\partial n_0}{\partial m_0} + w_1 \frac{\partial n_1}{\partial m_0} \tag{5}$$

$$\frac{\partial n}{\partial m_1} = w_0 \frac{\partial n_0}{\partial m_1} + w_1 \frac{\partial n_1}{\partial m_1} \tag{6}$$

```
n.backward(torch.ones_like(n)) # 将 (w0, w1) 取成 (1, 1)
```

```
print(m.grad)
```

```
Variable containing:
  4  27
[torch.FloatTensor of size 1x2]
```

通过自动求导我们得到了梯度是 4 和 27，我们可以验算一下

$$\frac{\partial n}{\partial m_0} = w_0 \frac{\partial n_0}{\partial m_0} + w_1 \frac{\partial n_1}{\partial m_0} = 2m_0 + 0 = 2 \times 2 = 4 \tag{7}$$

$$\frac{\partial n}{\partial m_1} = w_0 \frac{\partial n_0}{\partial m_1} + w_1 \frac{\partial n_1}{\partial m_1} = 0 + 3m_1^2 = 3 \times 3^2 = 27 \tag{8}$$

通过验算我们可以得到相同的结果

多次自动求导

通过调用 backward 我们可以进行一次自动求导，如果我们再调用一次 backward，会发现程序报错，没有办法再做一次。这是因为 PyTorch 默认做完一次自动求导之后，计算图就被丢弃了，所以两次自动求导需要手动设置一个东西，我们通过下面的小例子来说明。

```
x = Variable(torch.FloatTensor([3]), requires_grad=True)
y = x * 2 + x ** 2 + 3
print(y)
```

```
Variable containing:
 18
[torch.FloatTensor of size 1]
```

```
y.backward(retain_graph=True) # 设置 retain_graph 为 True 来保留计算图
```

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

```
Variable containing:
 8
[torch.FloatTensor of size 1]
```

```
y.backward() # 再做一次自动求导，这次不保留计算图
```

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

```
Variable containing:
 16
[torch.FloatTensor of size 1]
```

可以发现 x 的梯度变成了 16，因为这里做了两次自动求导，所以讲第一次的梯度 8 和第二次的梯度 8 加起来得到了 16 的结果。

小练习

定义

$$x = \begin{bmatrix} x_0 \\ x_1 \end{bmatrix} = \begin{bmatrix} 2 \\ 3 \end{bmatrix} \tag{9}$$

$$k = (k_0,\ k_1) = (x_0^2 + 3x_1,\ 2x_0 + x_1^2) \tag{10}$$

我们希望求得

$$j = \begin{bmatrix} \frac{\partial k_0}{\partial x_0} & \frac{\partial k_0}{\partial x_1} \\ \frac{\partial k_1}{\partial x_0} & \frac{\partial k_1}{\partial x_1} \end{bmatrix} \tag{11}$$

参考答案:

$$\begin{bmatrix} 4 & 3 \\ 2 & 6 \end{bmatrix} \tag{12}$$

```
x = Variable(torch.FloatTensor([2, 3]), requires_grad=True)
k = Variable(torch.zeros(2))

k[0] = x[0] ** 2 + 3 * x[1]
k[1] = x[1] ** 2 + 2 * x[0]
```

```
print(k)
```

```
Variable containing:
 13
 13
[torch.FloatTensor of size 2]
```

```
j = torch.zeros(2, 2)

k.backward(torch.FloatTensor([1, 0]), retain_graph=True)
j[0] = x.grad.data

x.grad.data.zero_() # 归零之前求得的梯度

k.backward(torch.FloatTensor([0, 1]))
j[1] = x.grad.data
```

```
print(j)
```

```
 4  3
 2  6
[torch.FloatTensor of size 2x2]
```

下一次课我们会介绍两种神经网络的编程方式，动态图编程和静态图编程