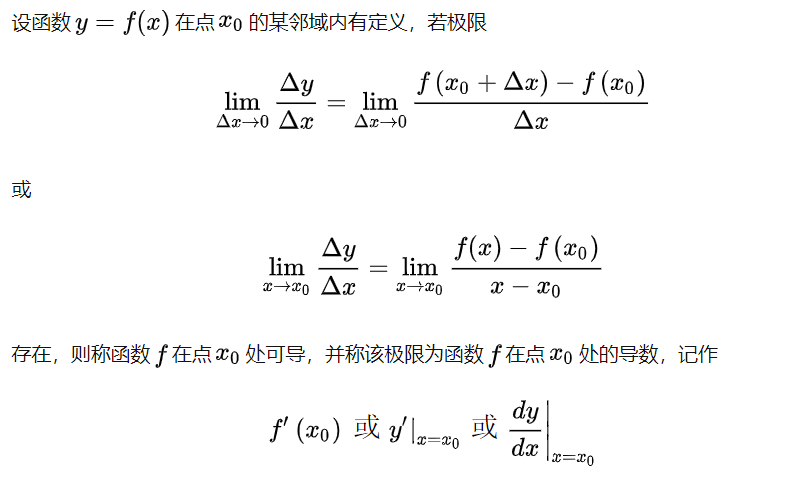
# **Pytorch教程五-张量的自动微分**

自动求导是深度学习框架的重要组成部分。自动微分为深度网络学习中的梯度的计算提供了“快车道”。因此我们本次文章来探究一下PyTorch中自动求导的相关知识。

**一、自定义实现梯度**

我们先看一下不借于PyTorch我们该怎么进行一个求导的操作，我们以一元函数的导数来举例，一元函数的导数定义大家可以参考高等数学里面的定义:



假设现在我们需要对一个函数在x=3处的导数：



那么我们怎么使用Python来实现这个工作呢？如下所示：

**def** grad\_myself(func, x, d=1e-7):  
 fc\_1, fc\_2 = func(x + d), func(x - d)  
 g = (fc\_1 - fc\_2) / (d)  
 **return** g  
f = **lambda** x: x\*\*2 + x\*\*(1/3)  
print(grad\_myself(func=f,x=3))

# 输出

12.320499891416148

可以看出这个，实际是有误的，我们把代码变一下：

g = (fc\_1 - fc\_2) / (d)

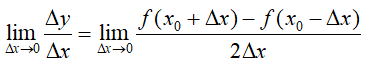
更改为：

g = (fc\_1 - fc\_2) / (2 \*d)

即可。这样最后的输出为：6.160249945708074

至于为什么多除以2，这里就留给大家自己思考了。





可以看出使用定义方法去实现一个函数的导数也是可以的。不过当我们函数变得很复杂的时候这个方法就不太好用了，我们来看一下PyTorch中是怎么实现这个梯度的计算的。

**二、PyTorch Autograd**

我们知道在深度学习中最为常见的优化算法是梯度下降类的算法，很自然的就必须要进行梯度的计算。很便捷的是PyTorch 提供了自动求导机制，我们来看一下。

在早期版本的PyTorch中 ，自动微分变量和张量是分开的，之后在0.4版本之后进行了统一，也就是说：在之后常见的变量都是可以进行自动求导的变量了：

import torch  
from torch.autograd import Variable  
X = Variable(torch.ones(3,3), requires\_grad = True)  
X\_1 = torch.ones(3,3)  
print(X)  
print(X\_1)

# 输出

tensor([[1., 1., 1.],

[1., 1., 1.],

[1., 1., 1.]], requires\_grad=True)

tensor([[1., 1., 1.],

[1., 1., 1.],

[1., 1., 1.]])

可以看出这个张量X有一个requires\_grad的属性，实际上这个张量与不带属性的requires\_grad张量是一样的。

现在我们吧这个变量X加上一个常数，并赋值给Y:

Y = X + 0.5

可以知道，此时我们Y的数据为：

print(Y)  
print(Y.data)

# 输出

tensor([[1.5000, 1.5000, 1.5000],

[1.5000, 1.5000, 1.5000],

[1.5000, 1.5000, 1.5000]], grad\_fn=<AddBackward0>)

tensor([[1.5000, 1.5000, 1.5000],

[1.5000, 1.5000, 1.5000],

[1.5000, 1.5000, 1.5000]])

可以发现，张量Y多了一个属性grad\_fn，这个grad\_fn中存储的就是<AddBackward0>信息，表示加上一个常数的操作，它表示计算图中的一个节点。

接下来我们在来一个赋值的操作：

Z = Y \* Y

注意这里的乘法表示的是元素对应相乘，可以知道Z的输出为：

tensor([[2.2500, 2.2500, 2.2500],

[2.2500, 2.2500, 2.2500],

[2.2500, 2.2500, 2.2500]], grad\_fn=<MulBackward0>)

此时grad\_fn中存储的就是<MulBackward0>,即乘法运算的信息。止于此，我们可以得到一个函数表达式：



我们改成小写：



那么怎么求解呢？我们可以进行手动的求解，那么使用自动求导该怎么进行呢？我们可以这样：

Z.backward(X)  
print(Z.grad)  
print(Y.grad)  
print(X.grad)

# 输出

None

None

tensor([[3., 3., 3.],

[3., 3., 3.],

[3., 3., 3.]])

D:\Python3\lib\site-packages\torch\\_tensor.py:1104: UserWarning: The .grad attribute of a Tensor that is not a leaf Tensor is being accessed. Its .grad attribute won't be populated during autograd.backward(). If you indeed want the .grad field to be populated for a non-leaf Tensor, use .retain\_grad() on the non-leaf Tensor. If you access the non-leaf Tensor by mistake, make sure you access the leaf Tensor instead. See github.com/pytorch/pytorch/pull/30531 for more informations. (Triggered internally at C:\actions-runner\\_work\pytorch\pytorch\builder\windows\pytorch\build\aten\src\ATen/core/TensorBody.h:475.)

return self.\_grad

这里解释一下前两行代码的含义，在PyTorch中直接对非变量求导是不允许的，所以我们backwrad()需要额外指定额外指定grad\_tensors即可。

PyTorch中规定，只有计算图上的叶子节点才可以通过**.backward()**获得梯度信息。Z和Y不是叶子节点，所以没有梯度信息，输出为None。输出的Warning信息也显示了这个信息。

那么回到最初的那个函数：



的梯度我们来计算一下：

import torch  
from torch.autograd import Variable  
X = Variable(torch.tensor([3.0]), requires\_grad = True)  
Y = X \*\* 2 + X \*\*(1/3)  
Y.backward()  
print(Y.grad)  
print(X.grad)

# 输出

None

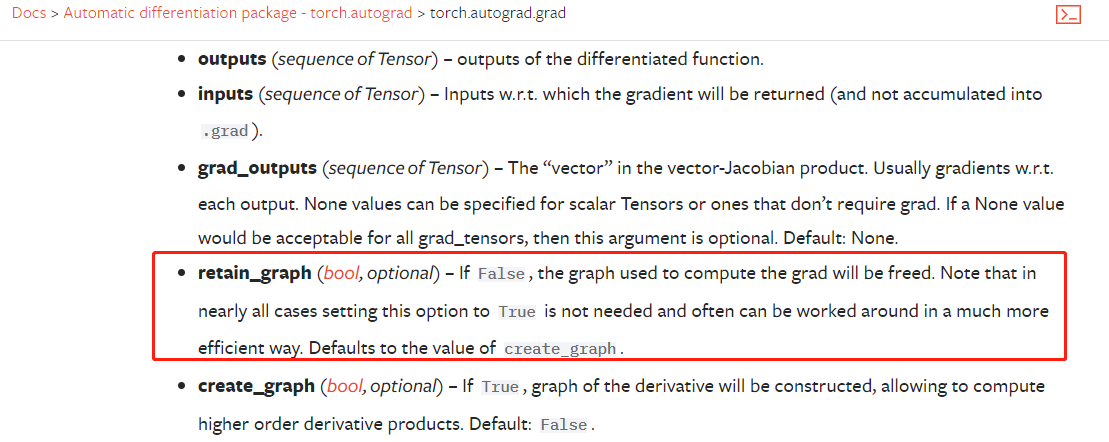
tensor([6.1602])

可以看出我们此时输出的结果与我们自己实现的求导结果是一致的。

注意点：在PyTorch中只有叶子节点才能计算grad信息，非叶子节点不能进行grad的计算。但是我们可以保留非叶子节点的梯度，下述代码的y.retain\_grad()即可保留y的梯度信息：

x = torch.tensor(5.0, requires\_grad=True)  
y = x.exp()  
y.retain\_grad()  
y.backward()  
print(y.grad)

官方文档也详细说明了这一点：



另外，目前版本1.11的Torch仅支持浮点Tensor类型（half、float、double 和 bfloat16）和复杂Tensor类型（cfloat、cdouble）的 autograd。关于底层autograd的实现感兴趣的小伙伴可以去自行学习。

1. **总结**

由此可知，在PyTorch中梯度和导数的计算可以自动化的进行，非常的方便。在今后的求导中，不管我们的目标函数是多么的复杂。只要借助我们backward()就可以完成自动求导的任务。