**Tensor.backward(retain\_graph)**

进行反向传播，计算动态计算图中叶节点的梯度

**retain\_graph参数解释**

每次 backward 时，默认会把整个计算图free掉。一般情况下是每次迭代，前向运算forward和反向传播backward是成对存在的，只需一次 forward 和一次 backward即可，因此默认retain\_graph=False  
但是在某些情况下，由于自定义loss等的复杂性，需要一次forward，多个不同loss的backward来累积同一个网络的grad来更新参数。于是，若在当前backward后，不执行forward，而是执行另一个backward，需要在当前backward时，指定保留计算图，即令retain\_graph=True

import torch

x = torch.randn((1,4),requires\_grad=True)

y = x \*\* 2*#x中各个元素平方*

z = x + y

loss1 = z.mean()

loss2 = z.sum()

loss1.backward()*#这个代码执行正常，但是执行完中间变量都free*了*，所以下一个出现了问题*

loss2.backward()*#计算图x-y-z结构被释放了，而计算loss2的backward仍然试图利用x-y-z的结构，此时会引发错误*

Error

修改最后两行

loss1.backward(retain\_graph=True)

loss2.backward()

Running Successfully

a = torch.randn((1,4),dtype=torch.float32,requires\_grad=True)

b = a \*\* 2

c = b \* 4

d = c + b

e = d / 3

f = e - 6

g = f + 2

loss1 = e.mean()

loss2 = g.sum()

loss1.backward()

loss2.backward()*#生成loss2的计算图的局部构被释放了，而计算loss2的backward仍然试图利用整个计算图，此时也会引发错误*

Error

修改最后两行

loss1.backward(retain\_graph=True)

loss2.backward()

Running Successfully