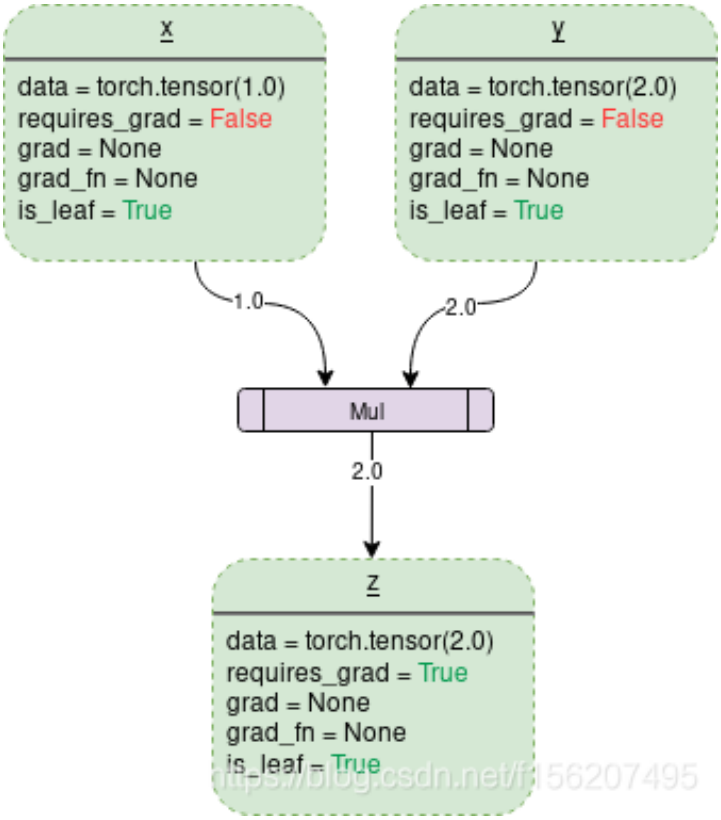
可参考网页：https://blog.csdn.net/f156207495/article/details/88727860?ops_request_misc=%257B%2522request%255Fid%2522%253A%252216264230461678026199644 7%2522%252C%2522scm%2522%253A%25220140713.130102334..%2522%257D&request_id=162642304616780261996447&biz_id=0&utm_medium=distribute.pc_search_result.none-task-blog-2~all~sobaiduend~default-1-88727860.first_rank_v2_pc_rank_v29&utm_term=pytorch+grad&spm=1018.2226.3001.4187

1 一些参数

tensor在pytorch里面是一个n维数组。我们可以通过指定参数reuqires_grad=True来建立一个反向传播图，从而能够计算梯度。



`requires_grad`：布尔变量（True/False），需要计算该Tensor梯度时，将其设置为True

`.grad_fn`：表示用于计算梯度的函数。每个`Tensor`都有一个`.grad_fn`属性。如果该Tensor是通过某些运算得到的，那么调用该属性就会返回一个与该运算相关的对象，否则返回`None`

`is_leaf`：为True或者False，表示该节点是否为叶子节点

当调用backward函数时，只有requires_grad为`true`以及is_leaf为`true`的节点才会被计算梯度，即grad属性才会被赋予值。

```
x = torch.ones(3, 3, requires_grad=True)
y = x * x
z = x + 1

print(x)
print(x.grad_fn)
print(y)
print(y.grad_fn)
print(z)
print(z.grad_fn)
```

输出

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

# y:
tensor([[1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.]], grad_fn=<MulBackward0>)
<MulBackward0 object at 0x7fe768298940>

# z:
tensor([[2., 2., 2.],
        [2., 2., 2.],
        [2., 2., 2.]], grad_fn=<AddBackward0>)
<AddBackward0 object at 0x7fd9801a0940>
```

- 可以采用in_place方式改变`requires_grad`属性

```
x = torch.ones(3, 3)
x = x * x + 2

print(x.requires_grad)
print(x.grad_fn)

x.requires_grad_(True)

print(x.requires_grad)
print(x.grad_fn)
```

输出

```
False
None
True #虽然requires_grad被修改为True
None #但是.grad_fn属性仍为None，因为x是由自己运算得到的
```

2 求梯度示例

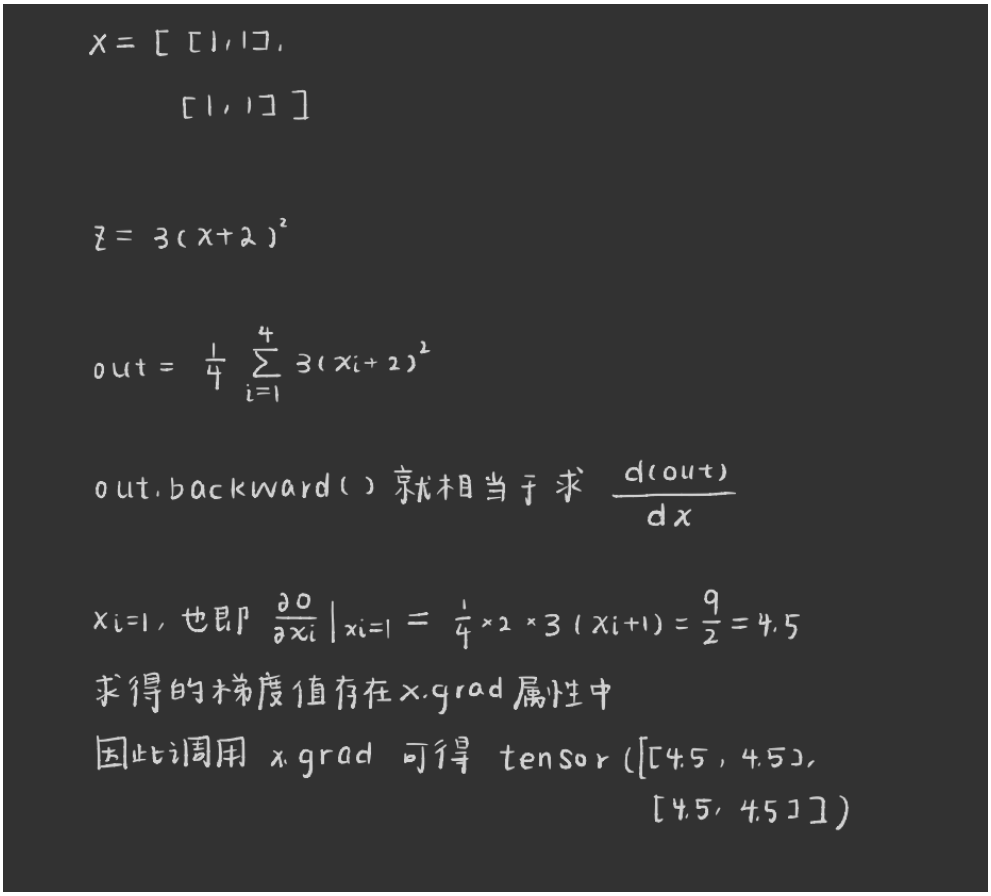
```
x = torch.ones(2, 2, requires_grad=True)
y = x + 2
z = y * y * 3
out = z.mean() #求平均值,out=tensor(27., grad_fn=<MeanBackward0>)

out.backward() #反向传播求梯度，结果存储在x.grad中
print(x.grad)
```

输出

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

解释



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 $l=g\left(\vec{y}\right)$ 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。

```
out2 = x.sum()
out2.backward()
print(x.grad)
print(out2)
```

输出

```
tensor([[5.5000, 5.5000],
        [5.5000, 5.5000]])
tensor(4., grad_fn=<SumBackward0>)
"""
可以看到，x.grad的值等于out关于x的梯度值+out2关于x的梯度值之和
4.5 + 1.0
"""
```

清零操作

`x.grad.data.zero_()`

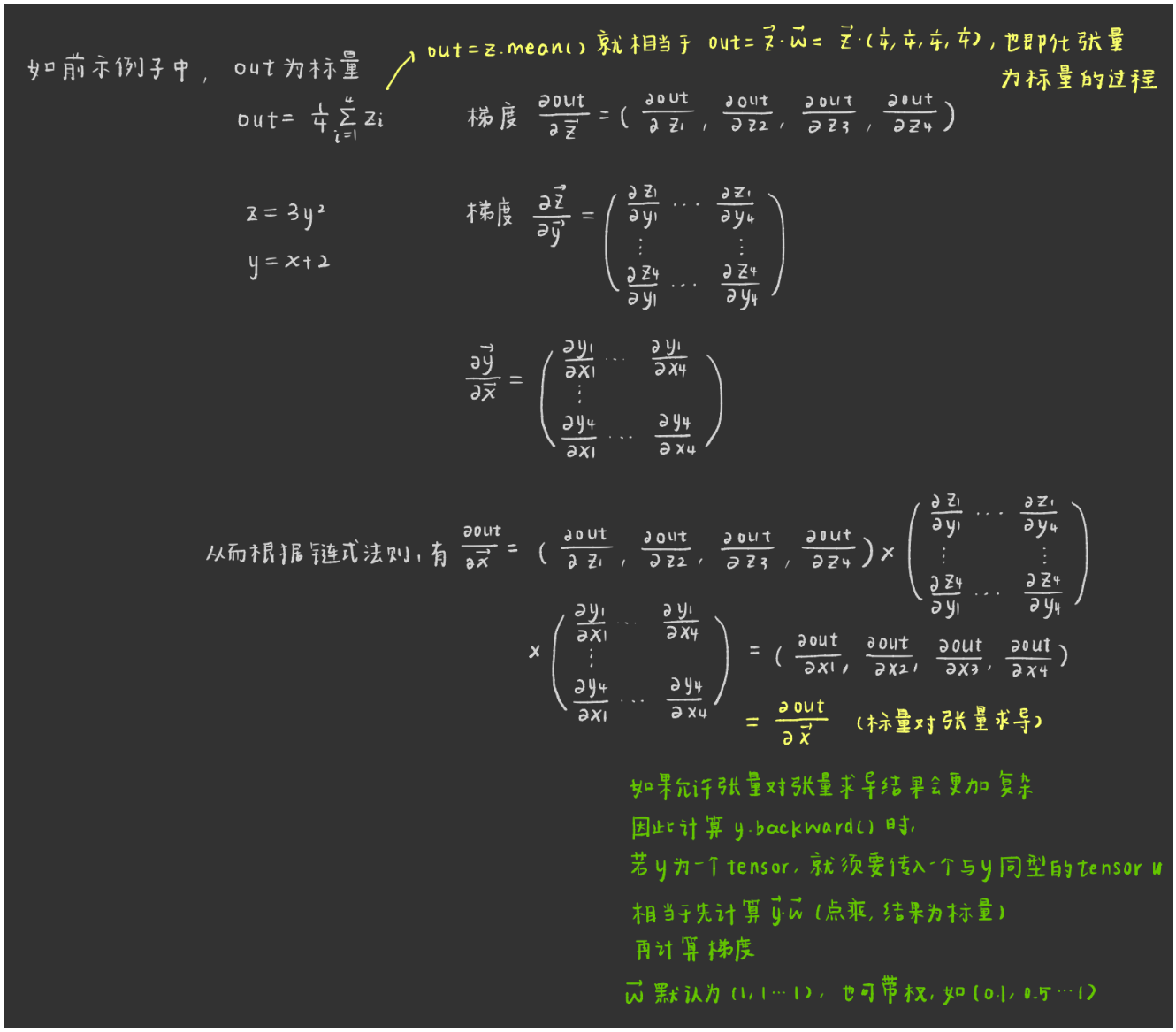
```
x.grad.data.zero_()

out2 = x.sum()
out2.backward()
print(x.grad)

#输出
tensor([[1., 1.],
        [1., 1.]])
```

- **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`。

  （只允许**标量对张量求导**，不允许张量对张量求导）




```
x = torch.tensor([1.0, 2.0, 3.0, 4.5], requires_grad=True)
y = x * x + 2
z = y.view(2, 2)
w = torch.tensor([[1.0, 0.1], [1.0, 0.01]]) #传入的与z同型的带权张量

z.backward(w)
print(x.grad)

#输出
tensor([2.0000, 0.4000, 6.0000, 0.0900])
```

- **中断梯度追踪**

  `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

```
x = torch.tensor(1., requires_grad=True)
y1 = x ** 2
with torch.no_grad():
    y2 = x ** 3
y3 = y1 + y2

y3.backward()
print(f"x.grad = {x.grad}")
print(f"y2.requires_grad = {y2.requires_grad}")

#输出
x.grad = 2.0
y2.requires_grad = False #由于被阻断了梯度追踪，因此requires_grad属性值为False
"""
如果没有中断y2的梯度追踪，那么y3=x^2+x^3,从而y3对x求导的结果为2x+3x^2,代入x=1，梯度值应为5
而由于中断了对y2的梯度追踪，从而实际上y3=x^2+1,从而y3对x的求导结果为2x,代入x=1，实际梯度值为2
"""
```

- **修改tensor数值而不影响求导**

  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.])
```