自动求导机制

autogrd

所在的“package”为**autogrd**，它针对tensor上面的数学运算等能进行自动微分运算；是一个“define-by-run”的架构，意味着根据你实际的运行结果来进行反向传播，每一个迭代都可以不一样。

torch.Tensor是这个包的最中心的类，设置变量的 *.requires_grad=True*时，关于这个tensor的所有操作都会被跟踪并记录；默认是False的。如果想不记录一块儿代码区域，你可以通过在代码区域上加入 *with torch.no_grad():*。当模型存在可训练参数，但是又不需要梯度时，就可以这么做。

Function

这是另一个很重要的类，Tensor和Function是相互联系的，形成一个无环图，记录完整的计算历史。每个tensor都有一个 *.grad_fn*来表示创建这个tensor的函数，但如果是用户自己手动创建的tensor，则没有这个变量。

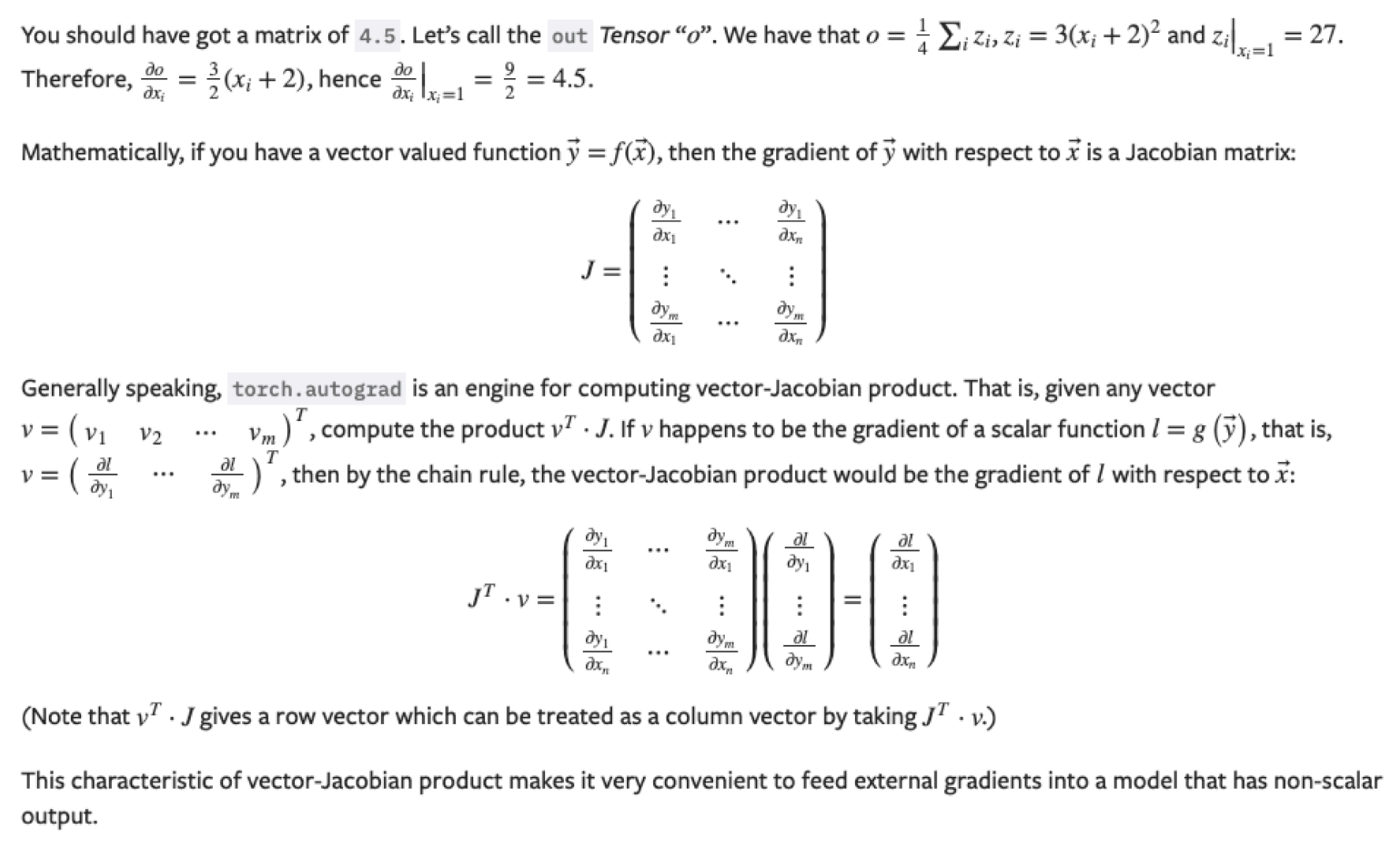
```
z = y * y * 3
out = z.mean()
print(z, out)

############
Out:

tensor([[27., 27.],
        [27., 27.]], grad_fn=<MulBackward0>) tensor(27., grad_fn=<MeanBackward0>)
```

如果想计算导数，就可以调用 *x.backward(...)* 来实现,如果x是scalar，那么就要不需要传参数,否则，需要传一个满足形状要求的梯度参数。

```
out.backward()# 和out.backward(torch.tensor(1.))是等价的,即d_out=1
```



You should have got a matrix of `4.5`. Let’s call the `out` *Tensor* “o”. We have that $o = \frac{1}{4}\sum_i z_i$, $z_i = 3(x_i+2)^2$ and $z_i\bigr\rvert_{x_i=1} = 27$. Therefore, $\frac{\partial o}{\partial x_i} = \frac{3}{2}(x_i+2)$, hence $\frac{\partial o}{\partial x_i}\bigr\rvert_{x_i=1} = \frac{9}{2} = 4.5$.

Mathematically, if you have a vector valued function $\vec{y} = f(\vec{x})$, then the gradient of $\vec{y}$ with respect to $\vec{x}$ is a Jacobian matrix:

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

Generally speaking, `torch.autograd` is an engine for computing vector-Jacobian product. That is, given any vector $v = \begin{pmatrix} v_1 & v_2 & \cdots & v_m \end{pmatrix}^T$, compute the product $v^T \cdot J$. If v happens to be the gradient of a scalar function $l = g\left(\vec{y}\right)$, that is, $v = \begin{pmatrix} \frac{\partial l}{\partial y_1} & \cdots & \frac{\partial l}{\partial y_m} \end{pmatrix}^T$, then by the chain rule, the vector-Jacobian product would be the gradient of l with respect to $\vec{x}$:

$$J^T \cdot v = \begin{pmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_m}{\partial x_1} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_1}{\partial x_n} & \cdots & \frac{\partial y_m}{\partial x_n} \end{pmatrix} \begin{pmatrix} \frac{\partial l}{\partial y_1} \\ \vdots \\ \frac{\partial l}{\partial y_m} \end{pmatrix} = \begin{pmatrix} \frac{\partial l}{\partial x_1} \\ \vdots \\ \frac{\partial l}{\partial x_n} \end{pmatrix}$$

(Note that $v^T \cdot J$ gives a row vector which can be treated as a column vector by taking $J^T \cdot v$.)

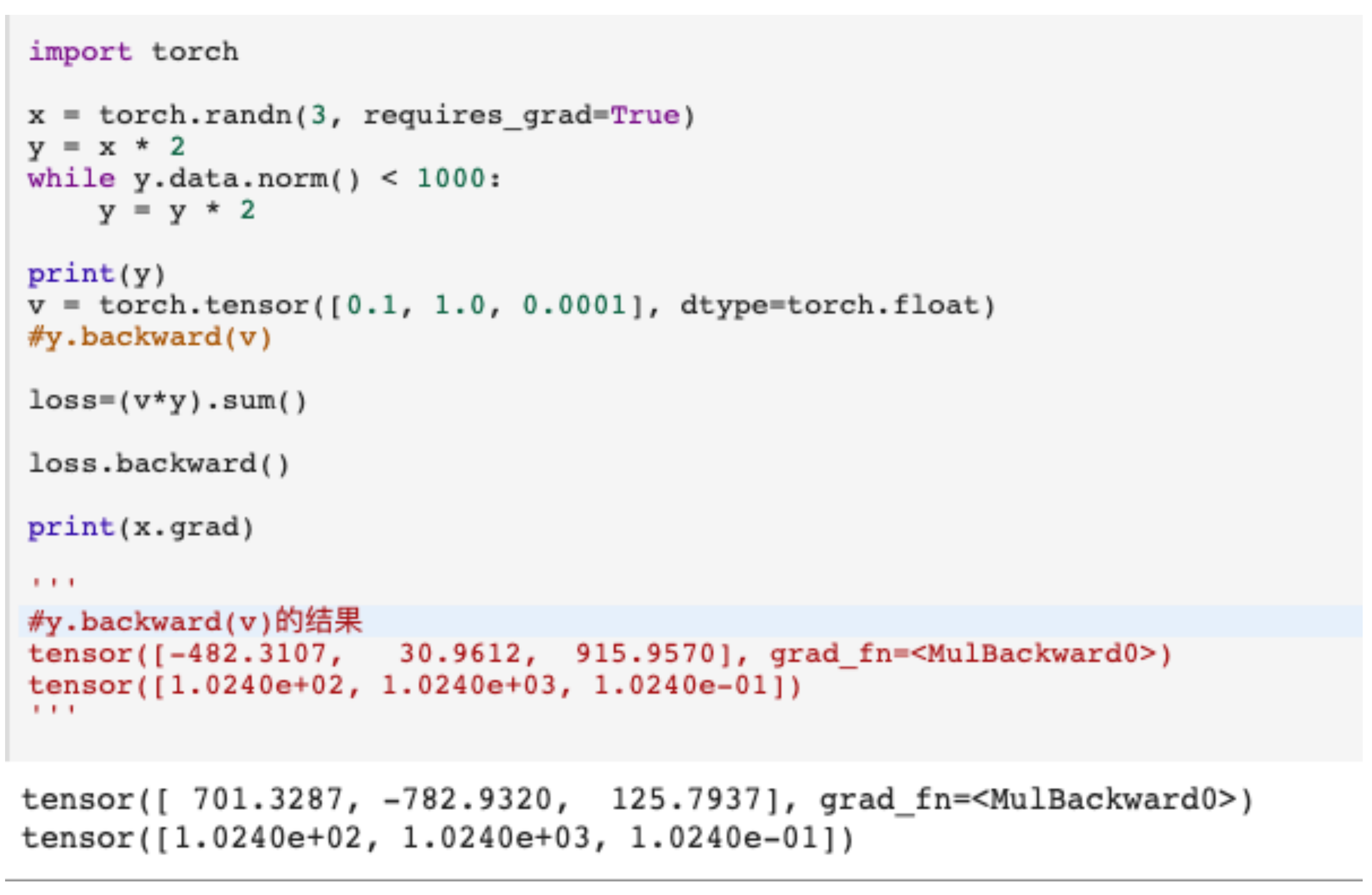
This characteristic of vector-Jacobian product makes it very convenient to feed external gradients into a model that has non-scalar output.

但是对于要求导的tensor不是scalar时，如下面例子的y，这个时候就需要传递一个参数，设置d_out:

```
x = torch.randn(3, requires_grad=True)
y = x * 2
while y.data.norm() < 1000:
    y = y * 2

print(y)
v = torch.tensor([0.1, 1.0, 0.0001], dtype=torch.float)
y.backward(v)
print(x.grad)
```

其实这个地方传递的参数v，起到的作用是设置一个d_out:
loss=0.1×y[0]+1.0×y[1]+0.0001×y[2]
而实验结果也是验证了这个知识点:



```
import torch

x = torch.randn(3, requires_grad=True)
y = x * 2
while y.data.norm() < 1000:
    y = y * 2

print(y)
v = torch.tensor([0.1, 1.0, 0.0001], dtype=torch.float)
#y.backward(v)

loss=(v*y).sum()

loss.backward()

print(x.grad)

'''
#y.backward(v)的结果
tensor([-482.3107,   30.9612,  915.9570], grad_fn=<MulBackward0>)
tensor([1.0240e+02, 1.0240e+03, 1.0240e-01])
'''
```

```
tensor([ 701.3287, -782.9320,  125.7937], grad_fn=<MulBackward0>)
tensor([1.0240e+02, 1.0240e+03, 1.0240e-01])
```