PyTorch autograd模块

`autograd`库主要提供了对Tensors上所有运算操作的自动微分功能，属于`define-by-run`;类型框架。即反向传播操作的定义是根据代码的运行方式。因此每次迭代都可以是不同的

1.张量

`torch.Tensor`设置他的属性`.requires_grad=True`就会开始追踪在该变量上的所有操作，而完成计算后，可以调用`.backward`并自动计算所有梯度，得到的梯度都保存在属性`.grad`中。

调用`detach()`方法分离出计算的历史，可以停止一个tensor变量继续追踪其历史信息，也可以防止未来的计算也被追踪。

如果希望防止跟踪历史以及使用内存，可以将代码块放在`with torch.no_grad():`内，这个做法在使用一个模型进行评估时常使用，因为模型会包含一些带有`require_grad=True`的训练参数。

> `Tensor`和`Function`两个类时有关联并建立了一个非循环的图，可以编码一个完整的计算记录。每个tensor变量都带有属性`.grad_fn`，该属性引用了创建了这个变量的`Function`（除非时是用户创建的Tensors，它们`grad_fn=None`）

如果要进行求导运算，可以调用一个`Tensor`变量的方法`.backward`如果该变量是一个标量，不需要传递任何参数给`.backward()`。当包含多个元素时，必须指定一个`gradient`参数，表示匹配尺寸大小的tensor。

代码实现：

导入必须的库：

```
improt torch
```

开始创建一个tensor，并让`requires_grad=True`来追踪该变量相关的计算操作：

```
x = torch.ones(2, 2, requires_grad=True)
print(x)
```

输出：

```
tensor([[1., 1.],
        [1., 1.]], requires_grad=True)
```

执行任意操作：

```
y = x + 2
print(y)
```

输出：

```
tensor([[3., 3.],
        [3., 3.]], grad_fn=<AddBackward>)
```

`y`是一个操作的结果，所以它带有属性`grad_fn`：

```
print(y.grad_fn)
```

输出：

```
<AddBackward object at 0x00000216D25DCC88>
```

继续对`y`进行操作：

```
z = y*y*3
out = z.mean()  # 求平均值
print('z={}'.format(z))
print('out={}'.format(out))
```

输出：

```
z= tensor([[27., 27.],
        [27., 27.]], grad_fn=<MulBackward>)

out= tensor(27., grad_fn=<MeanBackward1>)
```

一个`Tensor`变量默认`requires_grad`是`False`。可以像上述定义一个变量时候指定该属性是`True`，也可以定义变量后，调用`.requires_grad_(True)`设置为`True`。

例如：

```
a = torch.randn(2, 2)
a = ((a * 3) / (a - 1))
print(a.requires_grad)
a.requires_grad_(True)
print(a.requires_grad)
b = (a * a).sum()
print(b.grad_fn)
```

输出：

输出第一行为设置`requires_grad`的结果，接着显示调用`.requires_grad_(True)`，输出结果是`True`。

```
False
True
<SumBackward0 object at 0x00000216D25ED710>
```

2.梯度

`out`变量如上节定义，是一个标量，因此`out.backward()`相当于`out.backward(torch.tensor(1.))`。即：

```
x = torch.ones(2, 2, requires_grad=True)
y = x + 2
z = y*y*3
out = z.mean()   # 求算术平均值

out.backward()
print(x.grad)  # 输出梯度 d(out)/dx
```

输出：

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

上述过程使用数学语言进行描述：

前向过程

$$o = \frac{1}{4}\sum_i z_i$$
$$z_i = 3(x_i + 2)^2$$
$$z_i\big|_{x_i=1} = 27$$

所以计算梯度：

$$\frac{\partial o}{\partial x_i} = \frac{3}{2}(x_i + 2)$$
$$\frac{\partial o}{\partial x_i}\Big|_{x_i=1} = \frac{9}{2} = 4.5$$

> 若存在一个函数：
>
> $$\vec{y} = f(\vec{x})$$
>
> 则对应梯度为一个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.audograd`计算vector-Jacobian乘积的工具：

```
x = torch.randn(3, requires_grad=True)
print(x)
y = x * 2
while y.data.norm() < 1000:  # torch.norm()是对输入的tensor求指定维度上的范数
    y = y * 2
print(y)
```

输出：

```
tensor([-1.4212, -0.5061,  1.3746], requires_grad=True)
tensor([-727.6556, -259.1112,  703.7744], grad_fn=<MulBackward0>)
```

这里得到的变量`y`不再是一个标量，`torch.autograd`不能直接计算完整的Jacobian matrix，但可以传递向量给`backward()`作为参数得到vector-Jacobian的乘积。例：

```
v = torch.tensor([0.1, 1.0, 0.0001], dtype=torch.float)
y.backward(v)  # 此处相当于dy/dx与v做向量积， y.backward()得到dy/dx
print(x.grad)
```

输出：

```
tensor([ 102.4000, 1024.0000, 0.1024])
```

最后，加上`with torch.no_grad()`就可以停止追踪变量历史进行自动梯度计算：

```
print(x.requires_grad)
print((x ** 2).equires_grad)

with torch.no_grad():
    print((x**2).requires_grad)
```

输出：

```
True
True
False
```

注：`autograd`和`Function`官方文档介绍:https://pytorch.org/docs/stable/autograd.html