自动微分及Pytorch简明要点

李扶洋

邮箱：951678201@qq.com

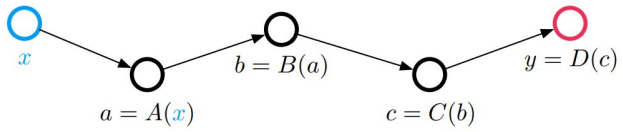
2022 年 8 月 3 日

§链式法则和Jacobians矩阵

有以下函数：

$$y = F(\mathbf{x}), \mathbf{x} \in \mathbb{R}^n, y \in \mathbb{R} \tag{1}$$

为了便于后续分析说明，进一步将函数表示为层层嵌套的函数型式（深度神经网络均为此类型）：



$$y = F(\mathbf{x}) = D(C(B(A(\mathbf{x})))) \tag{2}$$

即：$y = D(\mathbf{c})$，$\mathbf{c} = C(\mathbf{b})$，$\mathbf{b} = B(\mathbf{a})$，$\mathbf{a} = A(\mathbf{x})$

对 (2)求导：

$$\begin{aligned} F^{'}(\mathbf{x}) &= \frac{\partial y}{\partial \mathbf{x}} \\ &= \left[\frac{\partial y}{\partial x_1}\frac{\partial y}{\partial x_2}\cdots\frac{\partial y}{\partial x_n}\right]_{1\times n} \\ &= \frac{\partial y}{\partial \mathbf{c}}\frac{\partial \mathbf{c}}{\partial \mathbf{b}}\frac{\partial \mathbf{b}}{\partial \mathbf{a}}\frac{\partial \mathbf{a}}{\partial \mathbf{x}} \\ &= D^{'}(\mathbf{c})C^{'}(\mathbf{b})B^{'}(\mathbf{a})A^{'}(\mathbf{x}) \\ &= \mathbf{D}_{1\times o}\mathbf{C}_{o\times p}\mathbf{B}_{p\times q}\mathbf{A}_{q\times n} \end{aligned} \tag{3}$$

其中，$\frac{\partial y}{\partial \mathbf{c}} = D^{'}(\mathbf{c})$对应的Jcobian矩阵记作$\mathbf{D}_{1\times o}$
$\frac{\partial \mathbf{c}}{\partial \mathbf{b}} = C^{'}(\mathbf{b})$对应的Jcobian矩阵记作$\mathbf{C}_{o\times p}$
$\frac{\partial \mathbf{b}}{\partial \mathbf{a}} = B^{'}(\mathbf{a})$对应的Jcobian矩阵记作$\mathbf{B}_{p\times q}$
$\frac{\partial \mathbf{a}}{\partial \mathbf{x}} = A^{'}(\mathbf{x})$对应的Jcobian矩阵记作$\mathbf{A}_{q\times n}$

前向累积计算法：

$$
\begin{aligned}
F^{'}(\mathbf{x}) &= \frac{\partial y}{\partial \mathbf{c}}(\frac{\partial \mathbf{c}}{\partial \mathbf{b}}(\underbrace{\frac{\partial \mathbf{b}}{\partial \mathbf{a}}\frac{\partial \mathbf{a}}{\partial \mathbf{x}}})) \\
&\quad \longleftarrow \\
&= \mathbf{D}_{1\times o}(\mathbf{C}_{o\times p}(\underbrace{\mathbf{B}_{p\times q}\mathbf{A}_{q\times n}})) \\
&\quad \longleftarrow
\end{aligned}
\tag{4}
$$

Jacobian-Vector products(JVP):

$$
\begin{aligned}
F^{'}(\mathbf{x}) &= \frac{\partial y}{\partial \mathbf{c}}(\frac{\partial \mathbf{c}}{\partial \mathbf{b}}(\frac{\partial \mathbf{b}}{\partial \mathbf{a}}(\frac{\partial \mathbf{a}}{\partial \mathbf{x}}))) \\
&= \frac{\partial y}{\partial \mathbf{c}}(\frac{\partial \mathbf{c}}{\partial \mathbf{b}}(\underbrace{\frac{\partial \mathbf{b}}{\partial \mathbf{a}}}[\mathbf{a}_1, \mathbf{a}_2 \cdots \mathbf{a}_n])), \mathbf{a}_k \in \mathbb{R}^{q\times 1}, k \in (1 \ldots n) \\
&= \frac{\partial y}{\partial \mathbf{c}}(\frac{\partial \mathbf{c}}{\partial \mathbf{b}}(\underbrace{\frac{\partial \mathbf{b}}{\partial \mathbf{a}}}[\mathbf{V}_{q\times 1}, \mathbf{a}_2 \cdots \mathbf{a}_n])), \mathbf{a}_1 = \mathbf{V}_{q\times 1} \\
&= \frac{\partial y}{\partial \mathbf{c}}(\frac{\partial \mathbf{c}}{\partial \mathbf{b}}(\underbrace{\frac{\partial \mathbf{b}}{\partial \mathbf{a}}}[\mathbf{V}, \mathbf{a}_2 \cdots \mathbf{a}_n])), \mathbf{a}_1 = \mathbf{V} \\
&= \mathbf{D}_{1\times o}(\mathbf{C}_{o\times p}(\underbrace{\mathbf{B}_{p\times q}}[\mathbf{a}_1, \mathbf{a}_2 \cdots \mathbf{a}_n])) \\
&= \mathbf{D}_{1\times o}(\mathbf{C}_{o\times p}(\underbrace{\mathbf{B}_{p\times q}}[\mathbf{V}, \mathbf{a}_2 \cdots \mathbf{a}_n]))
\end{aligned}
\tag{5}
$$

前向累积算法特点：
构建Jacobian矩阵使用Jocobian-Vecter乘积（Products），一次一列按列构建，向前推进；需要n次调用才可完成Jacobian矩阵的计算

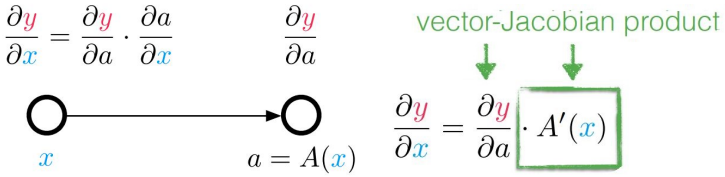
反向（后向）累积计算法：

$$
\begin{aligned}
F^{'}(\mathbf{x}) &= ((\underbrace{\frac{\partial y}{\partial \mathbf{c}}\frac{\partial \mathbf{c}}{\partial \mathbf{b}}})\frac{\partial \mathbf{b}}{\partial \mathbf{a}})\frac{\partial \mathbf{a}}{\partial \mathbf{x}} \\
&\quad \longrightarrow \\
&= ((\underbrace{\mathbf{D}_{1\times o}\mathbf{C}_{o\times p}})\mathbf{B}_{p\times q})\mathbf{A}_{q\times n} \\
&\quad \longrightarrow
\end{aligned}
\tag{6}
$$

Vector-Jacobian products(VJP):

$$
\begin{aligned}
F^{'}(\mathbf{x}) &= (((\frac{\partial y}{\partial \mathbf{c}})\frac{\partial \mathbf{c}}{\partial \mathbf{b}})\frac{\partial \mathbf{b}}{\partial \mathbf{a}})\frac{\partial \mathbf{a}}{\partial \mathbf{x}} \\
&= ((\underbrace{[\mathbf{y}_1, \mathbf{y}_2 \cdots \mathbf{y}_o]\frac{\partial \mathbf{c}}{\partial \mathbf{b}}})\frac{\partial \mathbf{b}}{\partial \mathbf{a}})\frac{\partial \mathbf{a}}{\partial \mathbf{x}}, \mathbf{y}_k \in \mathbb{R}^{1\times 1}, k \in (1 \dots o) \\
&= (((\underbrace{\mathbf{V}^T)_{1\times o}\mathbf{C}_{o\times p}})\mathbf{B}_{p\times q})\mathbf{A}_{q\times n}, [\mathbf{y}_1, \mathbf{y}_2 \cdots \mathbf{y}_o] = \mathbf{V}^T \\
&= ((\underbrace{\mathbf{V}^T\mathbf{C}_{o\times p}})\mathbf{B}_{p\times q})\mathbf{A}_{q\times n}, \mathbf{V} = [\mathbf{y}_1, \mathbf{y}_2 \cdots \mathbf{y}_o]^T \\
&= ((\underbrace{\mathbf{V}^T\mathbf{C}_{o\times p}})\mathbf{B}_{p\times q})\mathbf{A}_{q\times n}
\end{aligned}
\tag{7}
$$

反向累积算法特点：



构建Jacobian矩阵需要追踪记录程序的执行过程，跟踪使用的内存消耗和程序计算规模和深度正相关（checkpoint可以用来做平衡），使用Vecter-Jocobian乘积（Products），一次一行按行构建，反方推进；仅需1次调用才即可完成Jacobian矩阵的计算

二阶导数与高阶导数求解原理方法

部分的自动微分引擎框架可能只支持计算一阶导数，可以通过牛顿方法由一阶导数来获取得到二阶导数的近似值。在Pytorch中，反向累积算法来获取得到需要的一阶导数的计算链本身可以像前向计算过程一样被追踪，因此，高阶导数的自动微分计算就是对一阶导数计算过程对应反向累积被追踪的计算量再次进行求一阶导一样来反复计算获取得到

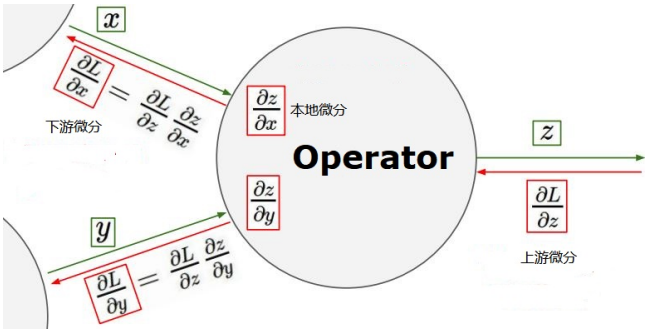
上述描述并不容易在阅读到该处时就能理解，继续学习下面的知识和案例，再去理解；另外，在本文最后简单图示了其简单情况下的实现思路和一般性洞见

反向传播(Backpropagation)

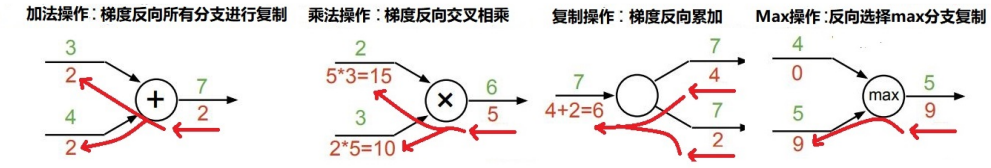
反向传播是AI中对上节中反向积累算法在该领域应用中常用称法，为了巩固该知识，通过反向传播算法的形式，再次较详细展现其计算过程，下面分3种情况来择要介绍：

第一种，标量输入标量输出：

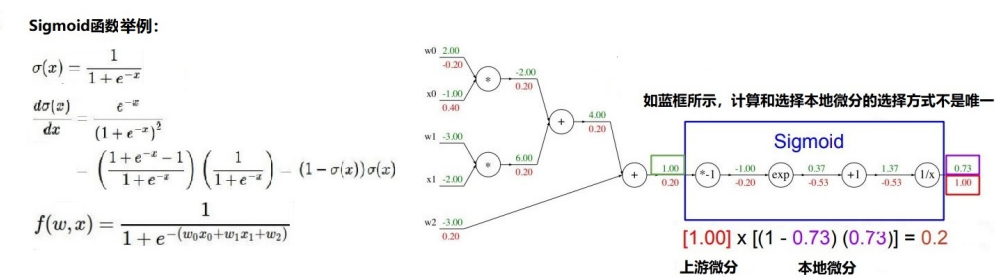
简单的使用链式法则：**下游微分=上游微分×本地微分**



典型运算操作的梯度信息传播样式：

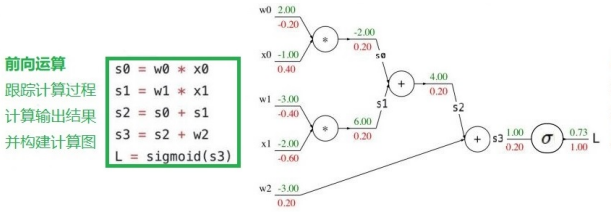


举例1：Sigmoid函数计算举例



举例2：带Sigmoid函数的正向和反向传播计算举例

反向传播算法使用举例：



前向运算

跟踪计算过程
计算输出结果
并构建计算图

```
s0 = w0 * x0
s1 = w1 * x1
s2 = s0 + s1
s3 = s2 + w2
L = sigmoid(s3)
```

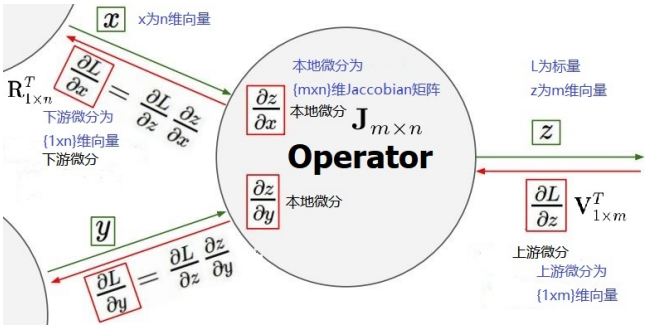
反向运算

利用计算图
和计算结果
反向计算梯度

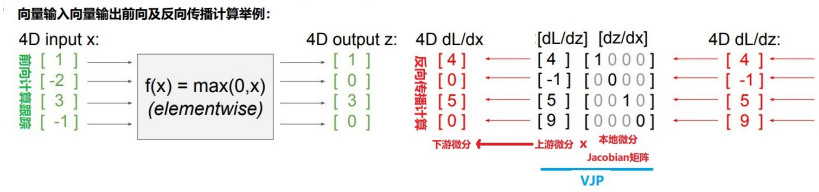
```
grad_L = 1.0
grad_s3 = grad_L * (1 - L) * L
grad_w2 = grad_s3
grad_s2 = grad_s3
grad_s0 = grad_s2
grad_s1 = grad_s2
grad_w1 = grad_s1 * x1
grad_x1 = grad_s1 * w1
grad_w0 = grad_s0 * x0
grad_x0 = grad_s0 * w0
```

第二种，向量输入向量输出：

链式法则：下游微分($\mathbf{R}_{1\times n}^T$)=上游微分($\mathbf{V}_{1\times m}^T$)×本地微分($\mathbf{J}_{m\times n}$)



举例：向量输入向量输出的前向和反向传播计算举例

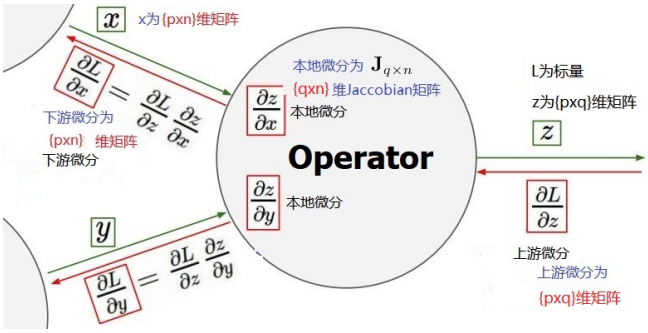


上图本地微分Jacobian矩阵只有对角线位置有非0值，属于较为典型的稀疏矩阵；满足稀疏矩阵的Jacobian在自动微分软件中一般采用稀疏算法进行处理，以上图情形为例：使用以下条件函数等式可见，基于输入维度的值是否大于0，其对应梯度直接等于上游对应梯度，而完全无需构建本地微分Jacobian矩阵

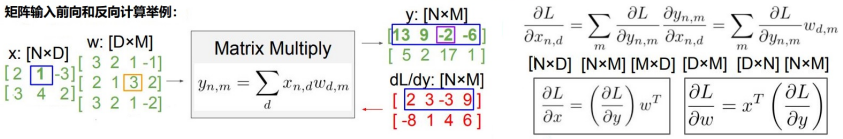
$$(\frac{\partial L}{\partial x})_i = \begin{cases} (\frac{\partial L}{\partial z})_i, if x_i > 0 \\ 0, otherwise \end{cases}$$

第三种：矩阵输入矩阵输出：

链式法则：**下游微分($\mathbf{R}_{p\times n}$)=上游微分($\mathbf{Z}_{p\times q}$)*本地微分($\mathbf{J}_{q\times n}$)**

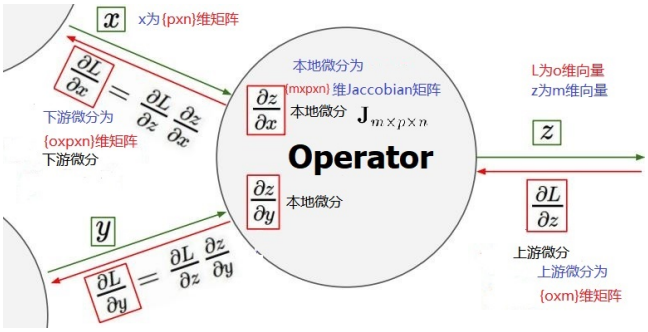


举例：矩阵输入矩阵输出前向及反向计算举例



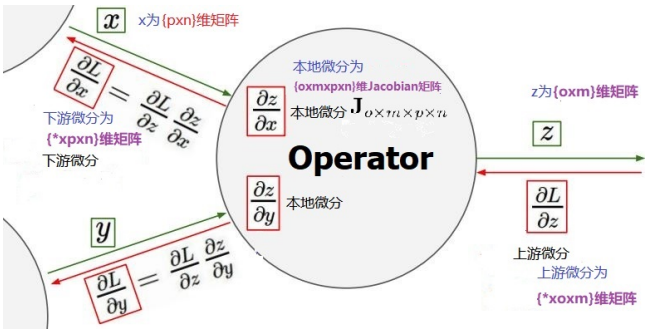
上例中，矩阵相乘情形在AI中非常典型普遍，其计算输入矩阵梯度的示意过程和最终结论弄清楚并记忆是非常有用的，这是理解Pytorch等主流自动微分软件中支持的Operators涉及的VJP代码是关键，用户书写自定义Operator，需要提供对应Operator的本地微分VJP代码也需要理解并知道该结论

链式法则：**下游微分($\mathbf{R}_{o\times p\times n}$)=上游微分($\mathbf{Z}_{o\times m}$)*本地微分($\mathbf{J}_{m\times p\times n}$)**



该图和下图中，输出L为向量和矩阵的情形在AI中少有使用，其计算方法仍是一样的，仅作参考，但这里也不进行举例

链式法则：下游微分($\mathbf{R}_{\star\times p\times n}$)=上游微分($\mathbf{Z}_{\star\times o\times m}$)*本地微分($\mathbf{J}_{o\times m\times p\times n}$)



自动微分简介

自动微分的实现通常有3种选择：

1，明确定义计算图（computation graph，Tensorflow使用该方式，但也支持第3种方式构建）；

2，内窥和分析源代码；

3，监控追踪函数执行,Pytorch，JAX等采用该方法，为接下来介绍重点

Pytorch自动微分简介

要点1：前向计算过程，构建计算**图**(这里的图指数据结构中的图，例如，前向计算过程构建有向图)

1）运行请求的操作，计算出结果Tensor

2）通过DAG（有向无环图）维护操作的梯度函数信息

要点2：反向传播过程，利用前向计算过程中构建的图进行反向追踪计算，得到需要的任意梯度值

1）反向过程起始于对backward()的在DAG根节点的调用，然后自动微分引擎计算每个操作函数的梯度

2）反向累积梯度信息到对于Tensor的grad属性参数中

3）使用链式法则，将梯度信息反向传播到叶子Tensor节点

要点3：自带Operators及自定义Operators

Pytorch自带Operators点击下面链接查看了解：

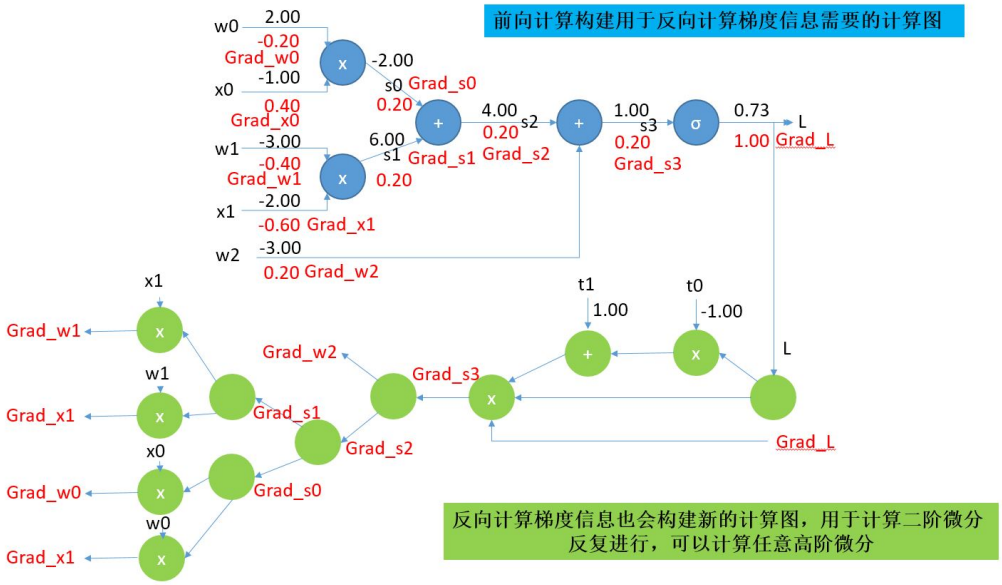
`https://github.com/pytorch/pytorch/tree/master/aten/src/ATen/native`

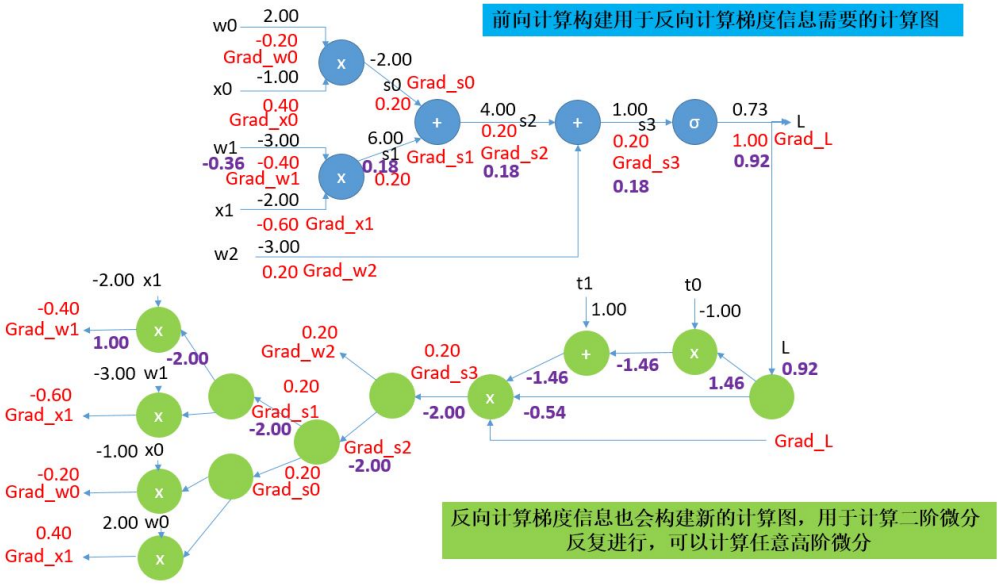
搞清楚本文档内容的基础上，可以自行编写自定义的Operators，参考如下链接

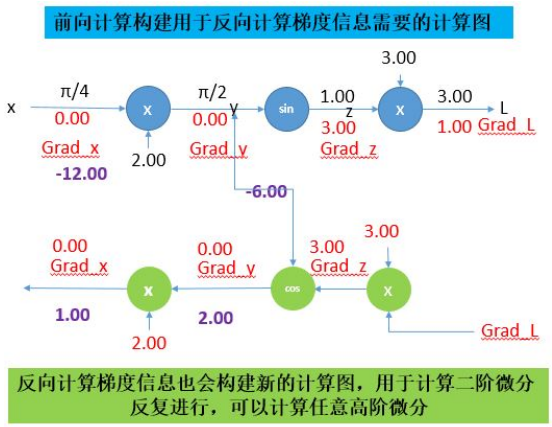
`https://pytorch.org/tutorials/advanced/torch_script_custom_ops.html`

要点4：二级及高阶自动微分

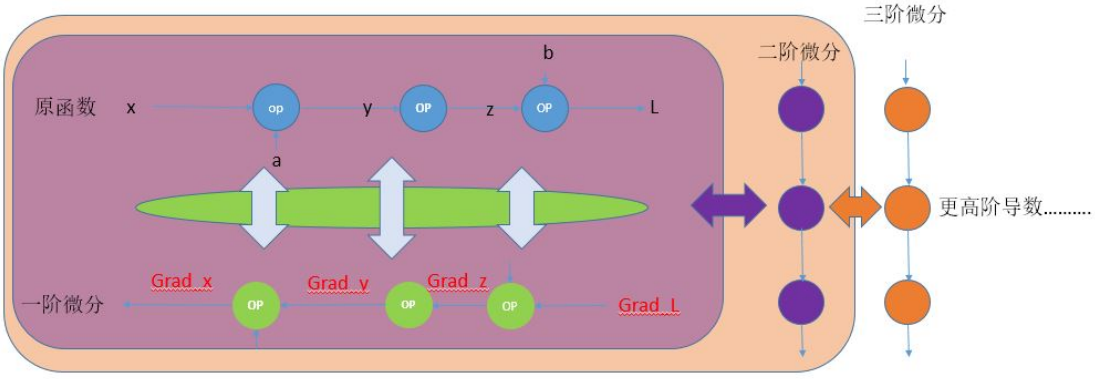
如图下所示，前向计算追踪构建的计算图可以用来计算一阶微分，一阶微分的反向计算过程也可以追踪形成新的计算图，进一步计算二阶微分，以此类推，可以计算任意高阶微分。Pytorch支持计算任意阶微分。以下是举例， 并由两个例子，得到一般的情况：


前向计算构建用于反向计算梯度信息需要的计算图
w0
2.00
-0.20
Grad_w0
x0
-1.00
0.40
Grad_x0
w1
-3.00
-0.40
Grad_w1
x1
-2.00
-0.60 Grad_x1
w2
-3.00
0.20 Grad_w2
-2.00
s0 Grad_s0
0.20
6.00
s1 Grad_s1
0.20
4.00
s2
0.20
Grad_s2
1.00
s3
0.20
Grad_s3
0.73
L
1.00 Grad_L
t1
1.00
t0
-1.00
Grad_w1
Grad_x1
Grad_w0
Grad_x1
Grad_w2
Grad_s3
Grad_s1
Grad_s2
Grad_s0
Grad_L
反向计算梯度信息也会构建新的计算图，用于计算二阶微分
反复进行，可以计算任意高阶微分


前向计算构建用于反向计算梯度信息需要的计算图
w0
2.00
-0.20
Grad_w0
x0
-1.00
0.40
Grad_x0
w1
-3.00
-0.36
-0.40
Grad_w1
x1
-2.00
-0.60 Grad_x1
w2
-3.00
0.20 Grad_w2
-2.00
s0 Grad_s0
0.20
6.00
s1 Grad_s1
0.18
0.20
4.00
s2
0.20
Grad_s2
0.18
1.00
s3
0.20
Grad_s3
0.18
0.73
L
1.00 Grad_L
0.92
t1
1.00
t0
-1.00
-2.00 x1
-0.40
Grad_w1
1.00
-2.00
-3.00 w1
-0.60
Grad_x1
-1.00 x0
-0.20
Grad_w0
2.00 w0
0.40
Grad_x1
0.20
Grad_w2
0.20
Grad_s3
-1.46
-2.00
-0.54
0.20
Grad_s1
-2.00
Grad_s2
-2.00
0.20
Grad_s0
-1.46
1.46
L
0.92
Grad_L
反向计算梯度信息也会构建新的计算图，用于计算二阶微分
反复进行，可以计算任意高阶微分



高阶微分计算图的一般情形



更高阶微分的计算图总是和其前面所有低阶计算图相关，并用到它们；这里面是否有模式可循以提升高阶微分计算效率，是可以探索的开放问题。