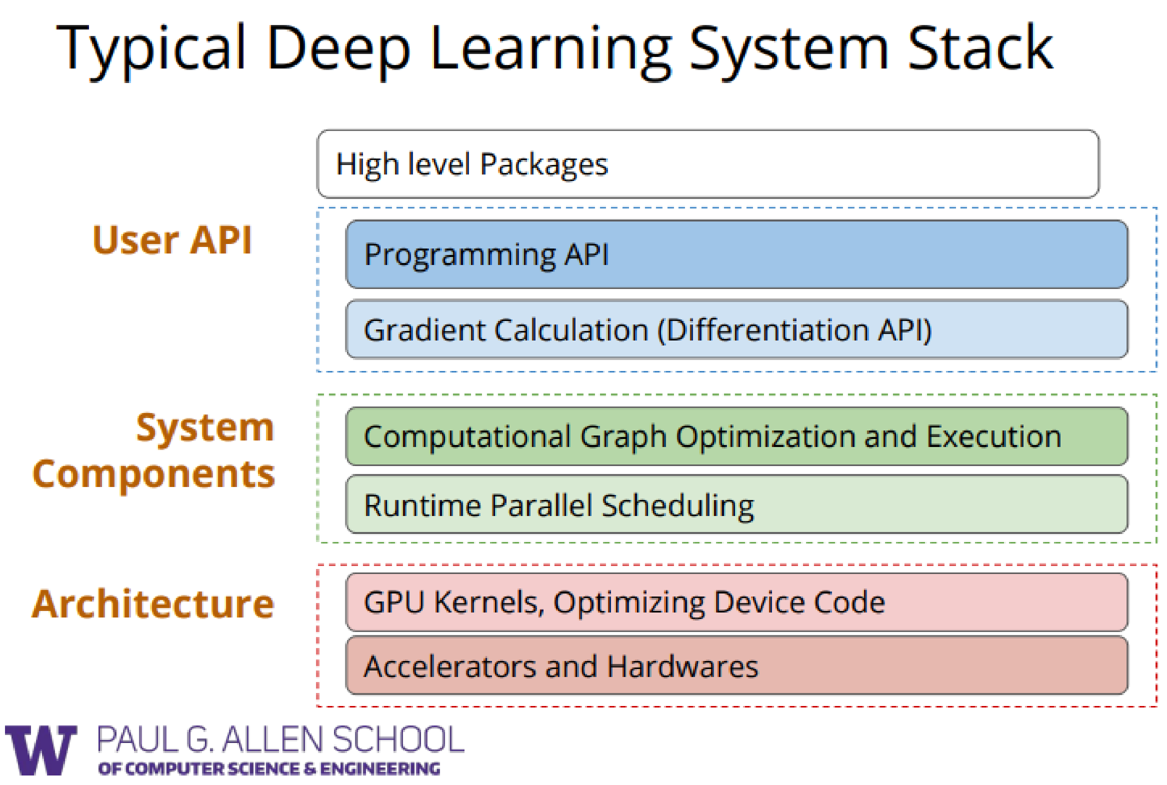
思路

内容



- 调用API
    - python

    - c++/cuda
    - 其他硬件加速方法
- 自动求导方法（autodiff）
- 内存共享优化
- 网络结构表示方法
- 计算图执行和优化

程序求导的四种方法

手动求导 Manual Derivatives

这种求导方法在传统计算机视觉模型中比较常用，也就是模型方法会定义一个能量函数之类的量。需要优化的变量则通过对能量函数进行理论求导之后再在代码中实现。很明显，这种方法几乎没有什么可拓展性。

数值微分 Numerical Differentiation

主要利用导数的定义：

$$\frac{\partial f(x)}{\partial x_i} \approx \frac{f(x + he_i) - f(x)}{h} \tag{1}$$

这样输出量$f(x)$的梯度$\nabla f = \left(\frac{\partial f}{\partial x_i}, \cdots, \frac{\partial f}{\partial x_n}\right)$，其中$e_i$是第$i$个元素为1其他为0的单位向量，$h$是一个很小的步长。这种方法比较容易实现，但是存在比较多的问题。

第一，这种方法只能近似。误差来源主要有两个，第一个是截断误差（truncate error），这是式(1)造成的，主要是由于$h \neq 0$引起的；另一个误差来源是舍入误差（round-off error），主要是由于计算机本身表示上无法完全与理论相等，$f(x + he_i)$与$f(x)$在表示时存在误差。当$h \to 0$时，截断误差趋于0，但是舍入误差占主导；而随着h增大，截断误差则占据主导。

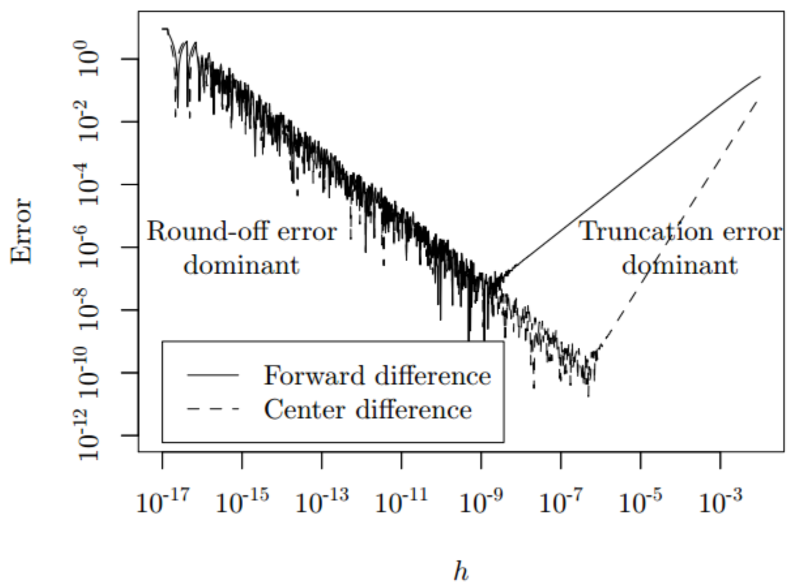


Figure 3: Error in the forward (Eq. 1) and center difference (Eq. 2) approximations as a function of step size h, for the derivative of the truncated logistic map $f(x) = 64x(1-x)(1-2x)^2(1-8x+8x^2)^2$. Plotted errors are computed using $E_{\text{forward}}(h, x_0) = \left| \frac{f(x_0+h)-f(x_0)}{h} - \frac{d}{dx} f(x)\big|_{x_0} \right|$ and $E_{\text{center}}(h, x_0) = \left| \frac{f(x_0+h)-f(x_0-h)}{2h} - \frac{d}{dx} f(x)\big|_{x_0} \right|$ at $x_0 = 0.2$.

一种改进方法是不用式(1)的前向方式，改为中心式的：

$$\frac{\partial f(x)}{\partial x_i} \approx \frac{f(x+he_i)-f(x-he_i)}{2h} + O(h^2) \tag{2}$$

这能去掉一阶截断误差（当然更高阶的截断误差仍然存在）。由式(2)，每次计算一个方向的梯度就要执行两次函数f。对于一个n维的输入变量和一个$f:\mathbb{R}^n \to \mathbb{R}^m$，计算一个雅可比矩阵需要执行$2mn$次$f$函数。

第二个问题是，各个维度的敏感度不同，步长h不能很好的确定。如果x本身的量级与h差不多，这种方法就会造成问题。

第三个问题，也是这种求导方法最主要的问题就是计算的复杂度。当n增大到成千上万时，计算这一梯度就成了主要的问题。相比于第一个误差问题，在深度学习的语境下，这种误差的容忍度较高。

符号微分 Symbolic Differentiation

符号求导在现在的一些数学软件如`Mathematica/Maple`中已经应用了，比如针对复合函数：

$$\frac{d}{dx}(f(x)+g(x)) = \frac{d}{dx}f(x) + \frac{d}{dx}g(x) \tag{3}$$

$$\frac{d}{dx}f(x)g(x) = \left(\frac{d}{dx}f(x)\right)g(x) + f(x)\left(\frac{d}{dx}g(x)\right) \tag{4}$$

$$\frac{d}{dx}\frac{f(x)}{g(x)} = \frac{f'(x)g(x)-f(x)g'(x)}{g(x)^2} \tag{5}$$

符号微分旨在为人提供一种直观的闭式解的自动微分。因此如果能将问题转化为一个纯数学符号问题，那么也就能用这类符号微分方法进行自动求解了。

符号微分自然也存在问题。其一是带求解问题必须能转化为一个数学符号表达式；其二，更重要的问题是，随着复合函数嵌套层数的增加，符号微分会遇到所谓的“表达式膨胀”（expression swell）问题：

Table 1: Iterations of the logistic map $l_{n+1} = 4l_n(1-l_n)$, $l_1 = x$ and the corresponding derivatives of l_n with respect to x, illustrating expression swell.

n	l_n	$\frac{d}{dx}l_n$	$\frac{d}{dx}l_n$ (Simplified form)
1	x	1	1
2	$4x(1-x)$	$4(1-x)-4x$	$4-8x$
3	$16x(1-x)(1-2x)^2$	$16(1-x)(1-2x)^2-16x(1-2x)^2-64x(1-x)(1-2x)$	$16(1-10x+24x^2-16x^3)$
4	$64x(1-x)(1-2x)^2(1-8x+8x^2)^2$	$128x(1-x)(-8+16x)(1-2x)^2(1-8x+8x^2)+64(1-x)(1-2x)^2(1-8x+8x^2)^2-64x(1-2x)^2(1-8x+8x^2)^2-256x(1-x)(1-2x)(1-8x+8x^2)^2$	$64(1-42x+504x^2-2640x^3+7040x^4-9984x^5+7168x^6-2048x^7)$

如果不加处理，为了计算嵌套函数的梯度，可能需要多次执行同一个表达式，这就造成实际所需的符号表达式将呈指数级增长，比如中间一列。事实上，我们可以看到$n=4$时的导数中有很多基本表达式在之前也出现过，我们可以保留一些中间结果避免再次计算。

自动微分 Auto Differentiation

自动微分技术可以看成是在执行一个计算机程序，只不过其中一步可能是对某些公式进行求导。由于所有数学计算最终都可以被分解为有限个基本操作，并且这些基本运算的梯度是已知的，通过链式法则对这些导数进行运算和组合就能计算出完整的结果。这些基本算子包括：二值逻辑运算，单元符号转换运算，超越函数（比如指数），对数函数和三角函数等。现在的深度学习框架都是使用AD方法实现自动求导的。

自动微分技术包括两种模式：前向模式（forward mode / tangent linear mode）和反向模式（reverse mode / cotangent linear mode）。假定一个函数$f(x_1,x_2)=\ln(x_1)+x_1x_2-\sin(x_2)$，并定义：

- 变量$v_{i-n}=x_i, i=1,\cdots,n$为输入变量；
- 变量$v_i\ i=1,\cdots,l$是中间变量；
- 变量$y_{m-i}=v_{l-i},\ i=m-1,\cdots,0$为输出变量。

现在通过对这一函数的求导过程来解释AD的前向和反向模式。

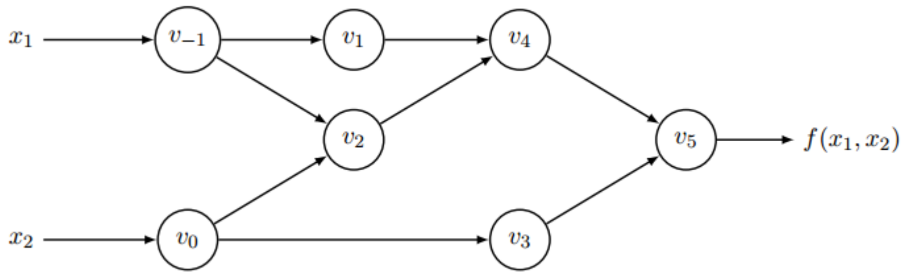


Figure 4: Computational graph of the example $f(x_1,x_2)=\ln(x_1)+x_1x_2-\sin(x_2)$. See the primal trace in Tables 2 or 3 for the definitions of the intermediate variables $v_{-1}\dots v_5$.

前向模式

前向模式的思路比较简单直接：根据计算图，我们利用链式法则自前向后逐个计算各中间变量相对于输入变量的导数：

$$\dot{v}_i=\frac{\partial v_i}{\partial x_1} \tag{6}$$

Table 2: Forward mode AD example, with $y=f(x_1,x_2)=\ln(x_1)+x_1x_2-\sin(x_2)$ evaluated at $(x_1,x_2)=(2,5)$ and setting $\dot{x}_1=1$ to compute $\frac{\partial y}{\partial x_1}$. The original forward evaluation of the primals on the left is augmented by the tangent operations on the right, where each line complements the original directly to its left.

Forward Primal Trace			Forward Tangent (Derivative) Trace		
v_{-1}	$=x_1$	$=2$	$\dot{v}_{-1}$	$=\dot{x}_1$	$=1$
v_0	$=x_2$	$=5$	$\dot{v}_0$	$=\dot{x}_2$	$=0$
v_1	$=\ln v_{-1}$	$=\ln 2$	$\dot{v}_1$	$=\dot{v}_{-1}/v_{-1}$	$=1/2$
v_2	$=v_{-1}\times v_0$	$=2\times 5$	$\dot{v}_2$	$=\dot{v}_{-1}\times v_0+\dot{v}_0\times v_{-1}$	$=1\times 5+0\times 2$
v_3	$=\sin v_0$	$=\sin 5$	$\dot{v}_3$	$=\dot{v}_0\times\cos v_0$	$=0\times\cos 5$
v_4	$=v_1+v_2$	$=0.693+10$	$\dot{v}_4$	$=\dot{v}_1+\dot{v}_2$	$=0.5+5$
v_5	$=v_4-v_3$	$=10.693+0.959$	$\dot{v}_5$	$=\dot{v}_4-\dot{v}_3$	$=5.5-0$
y	$=v_5$	$=11.652$	$\dot{\boldsymbol{y}}$	$=\dot{\boldsymbol{v}}_5$	$=\mathbf{5.5}$

给定一个数学表达式$f(x)$，它可以用一系列算子（加减乘除三角函数指数对数等）的组合表示。前向计算中每一步都对应一步函数计算和一步导数计算（即执行f和计算梯度同时进行），导数计算的依据则来自于函数。通过合适的数据表示方法，我们只需编写这些基础算子的前向计算和求导过程即可。

这一思路推广到多维数据和多维函数$f:\mathbb{R}^n\to\mathbb{R}^m$，其中$n$是输入变量的维度，$m$是输出变量的维度。求解其雅可比矩阵的每个元素时，可以在每个前向AD中设置为$\dot{\mathbf{x}}=\mathbf{e}_i$（即只有第$i$个元素为1，其他为0的单位向量）作为输入进行计算：

$$\dot{y}_j=\left.\frac{\partial y_j}{\partial x_i}\right|_{\mathbf{x}=\mathbf{a}},\quad j=1,\cdots,m \tag{7}$$

那么整个雅可比矩阵为：

$$\mathbf{J}_f = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_m}{\partial x_1} & \cdots & \frac{\partial y_m}{\partial x_n} \end{bmatrix} \Bigg|_{\mathbf{x}=\mathbf{a}} \tag{8}$$

或者可以初始化$\dot{\mathbf{x}} = \mathbf{r}$，用矩阵形式来计算：

$$\mathbf{J}_f \mathbf{r} = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_m}{\partial x_1} & \cdots & \frac{\partial y_m}{\partial x_n} \end{bmatrix} \begin{bmatrix} r_1 \\ \vdots \\ r_n \end{bmatrix} \tag{9}$$

这种前向表示对$f : \mathbb{R} \to \mathbb{R}^m$类型的函数比较高效和直接，只需要执行$f$一次即可；但对于另一种极端形式$f : \mathbb{R}^n \to \mathbb{R}$，则需要执行$n$次$f$函数的流程。对于一个$f : \mathbb{R} \to \mathbb{R}^m$的映射，求解其导数需要$n\ c\ \mathrm{ops}(f)$的运算时间（其中$c < 6$，一般取$c \sim [2, 3]$）。我们知道实际使用中，输入的维度$n$往往远大于输出的维度$m$（即$n \gg m$），所以这使得AD的前向模式并不那么好用；而AD反向模式则能使运算时间降为$m\ c\ \mathrm{ops}(f)$。

前向模式的实现：二元数求导法 Dual Number

AD的前向模式可以使用二元数求导法来方便的实现。

二元数是实数的一种推广。二元数引入了一个“二元数单位”ε，满足$\varepsilon \neq 0$且$\varepsilon^2 = 0$。每个二元数都具有$z = a + b\varepsilon$的形式（其中a和b是实数）。这种表达形式可以看成是对一般实数的一阶展开（$\varepsilon \neq 0$），更高阶的数据则被消除掉了（$\varepsilon^2 = 0$）。根据泰勒展开，函数$f(x)$可表达为：

$$f(x) = f(x_0) + f'(x_0)(x - x_0) + \cdots + \frac{f^{(n)}(x_0)}{n!}(x - x_0)^n + R_n(x) \tag{10}$$

所以如果忽略二阶项及更高阶项（$n \geq 2$），$f(x)$在$x = x_0 + \varepsilon$处满足：

$$f(x) = f(x_0) + f'(x_0)\varepsilon \tag{11}$$

二元数系数b即可看成是某函数$f(x)$在$x = a$处的导数。我们要做的就是为每个实数绑定一个二元数系数，并根据常用函数的求导法则更新该系数，就能获得任意复合函数在$x = a$处的导数了。

假定两个二元数分别为$x = a + b\varepsilon$和$y = c + d\varepsilon$，二元数的运算法则如下：

- 加法：

$$\begin{aligned} x + y &= (a + b\varepsilon) + (c + d\varepsilon) \\ &= (a + c) + (b + d)\varepsilon \end{aligned} \tag{12}$$

- 减法：

$$\begin{aligned} x - y &= (a + b\varepsilon) - (c + d\varepsilon) \\ &= (a - c) + (b - d)\varepsilon \end{aligned} \tag{13}$$

- 乘法：

$$\begin{aligned} x \times y &= (a + b\varepsilon) \times (c + d\varepsilon) \\ &= (ac + cd) + (bc + ad)\varepsilon + bd\varepsilon^2 \\ &= (ac + cd) + (bc + ad)\varepsilon \end{aligned} \tag{14}$$

- 除法：

$$\begin{aligned} \frac{x}{y} &= \frac{a + b\varepsilon}{c + d\varepsilon} \\ &= \frac{(a + b\varepsilon)(c - d\varepsilon)}{(c + d\varepsilon)(c - d\varepsilon)} \\ &= \frac{ac + (bc - ad)\varepsilon}{c^2} \\ &= \frac{a}{c} + \frac{bc - ad}{c^2}\varepsilon \end{aligned} \tag{15}$$

- 幂：

$$\begin{aligned} x^y &= (a+b\varepsilon)^{c+d\varepsilon} \\ &= a^c + \varepsilon\left(b(ca^{c-1}) + d(a^c \ln a)\right) \end{aligned} \tag{16}$$

特别的，当指数为实数时：

$$\begin{aligned} x^y &= (a+b\varepsilon)^c \\ &= a^c + (ca^{c-1})b\varepsilon \end{aligned} \tag{17}$$

当底数为实数时：

$$\begin{aligned} x^y &= a^{c+d\varepsilon} \\ &= a^c + d(a^c \ln a)\varepsilon \end{aligned} \tag{18}$$

- 三角函数：

$$\sin(a+b\varepsilon) = \sin(a) + \cos(a)b\varepsilon \tag{19}$$

$$\cos(a+b\varepsilon) = \cos(a) - \sin(a)b\varepsilon \tag{20}$$

$$\tan(a+b\varepsilon) = \tan(a) + \frac{1}{\cos(a)^2}b\varepsilon \tag{21}$$

$$\arctan(a+b\varepsilon) = \arctan(a) + \frac{1}{1+a^2}b\varepsilon \tag{22}$$

- 对数函数：

$$\log_s(a+b\varepsilon) = log_s(a) + \frac{1}{\ln(s)a}b\varepsilon \tag{23}$$

一般的，令一个实数a对应的一个二元数为$a+\varepsilon$，则复合函数$F = f_1(f_2(f_3(\ldots f_n(x)\ldots)))$在$x=a$处的导数为：

$$F'|_{x=a} = \text{Dual}\left(F\left(a+\varepsilon\right)\right) \tag{24}$$

因此，我们只需要编写一些针对二元数的基础运算法则和函数即可。需要注意的是，我们并不需要实际给ε进行赋值，只要记住它与虚数单位类似，是一个独立的单位即可。这里用python给个简单的实现：

```python
import numpy as np
import math


class DualNumber:
    def __init__(self, x, y):
        self.real = x
        self.dual = y

    def __str__(self):
        rpr = '{}+{}e'.format(self.real, self.dual)
        return rpr

    def __repr__(self):
        return self.__str__()

    def __add__(self, other):
        if isinstance(other, DualNumber):
            real = self.real + other.real
            dual = self.dual + other.dual
        elif np.isscalar(other):
            real = self.real + other
            dual = self.dual
        else:
            raise TypeError('The other operator should be a scalar or a {}'.format(self.__class__.__name__))
        return DualNumber(real, dual)

    def __radd__(self, other):
        return self.__add__(other)
```

```python

    def __sub__(self, other):
        if isinstance(other, DualNumber):
            real = self.real - other.real
            dual = self.dual - other.dual
        elif np.isscalar(other):
            real = self.real - other
            dual = self.dual
        else:
            raise TypeError('The other operator should be a scalar or a {}'.format(self.__class__.__name__))
        return DualNumber(real, dual)

    def __rsub__(self, other):
        if isinstance(other, DualNumber):
            real = other.real - self.real
            dual = other.dual - self.dual
        elif np.isscalar(other):
            real = other.real - self.real
            dual = - self.dual
        else:
            raise TypeError('The other operator should be a scalar or a {}'.format(self.__class__.__name__))
        return DualNumber(real, dual)

    def __mul__(self, other):
        if isinstance(other, DualNumber):
            real = self.real * other.real
            dual = self.dual * other.real + self.real * other.dual
        elif np.isscalar(other):
            real = self.real * other
            dual = self.dual * other
        else:
            raise TypeError('The other operator should be a scalar or a {}'.format(self.__class__.__name__))
        return DualNumber(real, dual)

    def __rmul__(self, other):
        return self.__mul__(other)

    def __truediv__(self, other):
        if isinstance(other, DualNumber):
            if other.real == 0:
                raise ValueError
            real = self.real / other.real
            dual = (self.dual - self.real / other.real * other.dual) / other.real
        elif np.isscalar(other):
            if other == 0:
                raise ValueError
            real = self.real / other
            dual = self.dual / other
        else:
            raise TypeError('The other operator should be a scalar or a {}'.format(self.__class__.__name__))
        return DualNumber(real, dual)

    def __pow__(self, power, modulo=None):
        real = math.pow(self.real, power)
        dual = self.dual * power * math.pow(self.real, power-1)
        return DualNumber(real, dual)
```

```python

    def __abs__(self):
        real = abs(self.real)
        dual = np.sign(self.real)
        return DualNumber(real, dual)

    @staticmethod
    def sin(a):
        real = math.sin(a.real)
        dual = a.dual * math.cos(a.real)
        return DualNumber(real, dual)

    @staticmethod
    def cos(a):
        real = math.cos(a.real)
        dual = - a.dual * math.sin(a.real)
        return DualNumber(real, dual)

    @staticmethod
    def tan(a):
        real = math.tan(a.real)
        x = math.cos(a.real)
        dual = a.dual / (x * x)
        return DualNumber(real, dual)

    @staticmethod
    def atan(a):
        real = math.atan(a.real)
        x = a.real
        dual = a.dual / (1. + x*x)
        return DualNumber(real, dual)

    @staticmethod
    def sqrt(a):
        real = math.sqrt(a.real)
        dual = .5 * a.dual / real
        return DualNumber(real, dual)

    @staticmethod
    def exp(a):
        real = math.exp(a.real)
        dual = a.dual * math.exp(a.real)
        return DualNumber(real, dual)

    @staticmethod
    def log(a, base=math.e):
        real = math.log(a.real, base)
        dual = 1. / a.real / math.log(base) * a.dual
        return DualNumber(real, dual)
```

反向模式

反向传播BP可以看成AD反向模式的一种特例。不同于前向模式，反向模式需要计算输出对于每个中间变量v_i的梯度伴随量：

$$\bar{v}_i = \frac{\partial y_j}{\partial v_i} \tag{25}$$

这一导数表征着输出变量y_j对于中间变量v_i的敏感程度。在BP算法中，y就是最后的损失函数值了。

在反向模式中，导数是通过一个两阶段的过程计算出来的。在第一个阶段中，我们执行函数f的计算，获得所有的中间变量v_i，并且在计算图中记录变量之间的依赖性和相关性；在第二阶段中，输出对输入的导数是通过反方向从输出到输入传播梯度伴随量得到的：

Table 3: Reverse mode AD example, with $y = f(x_1, x_2) = \ln(x_1) + x_1x_2 - \sin(x_2)$ evaluated at $(x_1, x_2) = (2, 5)$. After the forward evaluation of the primals on the left, the adjoint operations on the right are evaluated in reverse (cf. Figure 1). Note that both $\frac{\partial y}{\partial x_1}$ and $\frac{\partial y}{\partial x_2}$ are computed in the same reverse pass, starting from the adjoint $\bar{v}_5 = \bar{y} = \frac{\partial y}{\partial y} = 1$.

Forward Primal Trace		Reverse Adjoint (Derivative) Trace		
$v_{-1} = x_1$	$= 2$	$\boldsymbol{\bar{x}_1 = \bar{v}_{-1}}$		$\mathbf{= 5.5}$
$v_0 = x_2$	$= 5$	$\boldsymbol{\bar{x}_2 = \bar{v}_0}$		$\mathbf{= 1.716}$
$v_1 = \ln v_{-1}$	$= \ln 2$	$\bar{v}_{-1} = \bar{v}_{-1} + \bar{v}_1 \frac{\partial v_1}{\partial v_{-1}}$	$= \bar{v}_{-1} + \bar{v}_1 / v_{-1}$	$= 5.5$
$v_2 = v_{-1} \times v_0$	$= 2 \times 5$	$\bar{v}_0 = \bar{v}_0 + \bar{v}_2 \frac{\partial v_2}{\partial v_0}$	$= \bar{v}_0 + \bar{v}_2 \times v_{-1}$	$= 1.716$
		$\bar{v}_{-1} = \bar{v}_2 \frac{\partial v_2}{\partial v_{-1}}$	$= \bar{v}_2 \times v_0$	$= 5$
$v_3 = \sin v_0$	$= \sin 5$	$\bar{v}_0 = \bar{v}_3 \frac{\partial v_3}{\partial v_0}$	$= \bar{v}_3 \times \cos v_0$	$= -0.284$
$v_4 = v_1 + v_2$	$= 0.693 + 10$	$\bar{v}_2 = \bar{v}_4 \frac{\partial v_4}{\partial v_2}$	$= \bar{v}_4 \times 1$	$= 1$
		$\bar{v}_1 = \bar{v}_4 \frac{\partial v_4}{\partial v_1}$	$= \bar{v}_4 \times 1$	$= 1$
$v_5 = v_4 - v_3$	$= 10.693 + 0.959$	$\bar{v}_3 = \bar{v}_5 \frac{\partial v_5}{\partial v_3}$	$= \bar{v}_5 \times (-1)$	$= -1$
		$\bar{v}_4 = \bar{v}_5 \frac{\partial v_5}{\partial v_4}$	$= \bar{v}_5 \times 1$	$= 1$
$y = v_5$	$= 11.652$	$\bar{v}_5 = \bar{y}$	$= 1$	

同样以函数$f(x_1,x_2) = \ln(x_1) + x_1x_2 - \sin(x_2)$为例。前馈过程与AD前向模式中的情况一样（左列），但是求导则与之前的顺序相反，是从输出变量开始的。由于定义了$y = v_5$，所以$\bar{v}_5 = \frac{\partial y}{\partial v_5} = 1$；而$v_5$是由$v_3$和$v_4$两个变量计算得到的，并且：

$$\frac{\partial y}{\partial v_3} = \frac{\partial y}{\partial v_5}\frac{\partial v_5}{\partial v_3} = \bar{v}_5\frac{\partial v_5}{\partial v_3} \tag{26}$$

$$\frac{\partial y}{\partial v_4} = \frac{\partial y}{\partial v_5}\frac{\partial v_5}{\partial v_4} = \bar{v}_5\frac{\partial v_5}{\partial v_4} \tag{27}$$

所以通过$\bar{v}_5$和$\frac{\partial v_5}{\partial v_3}$可以计算到伴随量$\bar{v}_3$。$\bar{v}_4$类似。不难看出，这一过程就是本质上就是机器学习中的反向传播方法。只是此处输出y是变量（标量或矢量、矩阵），而不仅仅可以是机器学习中的损失函数值（标量）。另外值得一提的是，计算完$\bar{v}_3$和$\bar{v}_4$后，$\bar{v}_5$也就完成了任务，离开了其作用域（红线之间的几行为对应变量的作用域），可以在内存中释放掉。这可能也是PyTorch的`loss.backward()`实现中，一个结点完成反传后计算图被释放掉的原因。

对于输出到多个结点的中间变量，如v_0与v_2/v_3都相关，其梯度为：

$$\begin{aligned}\frac{\partial y}{\partial v_0} &= \frac{\partial y}{\partial v_2}\frac{\partial v_2}{\partial v_0} + \frac{\partial y}{\partial v_1}\frac{\partial v_1}{\partial v_0} \\ &= \bar{v}_2\frac{\partial v_2}{\partial v_0} + \bar{v}_1\frac{\partial v_1}{\partial v_0}\end{aligned} \tag{28}$$

具体实现时，一般使用多步增量模式：

$$\bar{v}_0 = 0 \tag{29}$$

$$\bar{v}_0 = \bar{v}_0 + \bar{v}_2\frac{\partial v_2}{\partial v_0} \tag{30}$$

$$\bar{v}_0 = \bar{v}_0 + \bar{v}_1\frac{\partial v_1}{\partial v_0} \tag{31}$$

上文中我们提到前向模式中，如果$f : \mathbb{R}^n \to \mathbb{R}$，那么计算所有针对输入变量的导数需要执行$n$次$f$函数流程；而在反向模式中，$f$函数流程执行次数则变为了$m$，即输出变量的维度。一次流程即可算出某个输出变量针对所有输入变量的导数：

$$\nabla y_i = \left(\frac{\partial y_i}{\partial x_1}, \cdots, \frac{\partial y_i}{\partial x_n}\right) \tag{32}$$

在$n \gg m$的情况下， AD的反向模式能够有效降低执行计算量。反向模式也可以用矩阵向量化表达为：

$$\mathbf{J}_f^T \mathbf{r} = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{y_m}{x_1} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_1}{\partial x_n} & \cdots & \frac{y_m}{x_n} \end{bmatrix} \begin{bmatrix} r_1 \\ \vdots \\ r_m \end{bmatrix} \tag{33}$$

其中初始化$\bar{\mathrm{y}} = \mathbf{r}$。

AD反向模式也有自身的缺陷，就是在最坏情况下导致计算所需内存空间增加（与反馈过程中的操作数量成比例）。如何优化和高效利用内存是一个比较热门的研究方向。

Table 4: Evaluation times of the Helmholtz free energy function and its gradient (Figure 5). Times are given relative to that of the original function with both (1) $n = 1$ and (2) n corresponding to each column. (For instance, reverse mode AD with $n = 43$ takes approximately twice the time to evaluate relative to the original function with $n = 43$.) Times are measured by averaging a thousand runs on a machine with Intel Core i7-4785T 2.20 GHz CPU and 16 GB RAM, using DiffSharp 0.5.7. The evaluation time for the original function with $n = 1$ is 0.0023 ms.

	n, number of variables							
	1	8	15	22	29	36	43	50
f, original								
Relative $n = 1$	1	5.12	14.51	29.11	52.58	84.00	127.33	174.44
∇f, numerical diff.								
Relative $n = 1$	1.08	35.55	176.79	499.43	1045.29	1986.70	3269.36	4995.96
Relative n in column	1.08	6.93	12.17	17.15	19.87	23.64	25.67	28.63
∇f, forward AD								
Relative $n = 1$	1.34	13.69	51.54	132.33	251.32	469.84	815.55	1342.07
Relative n in column	1.34	2.66	3.55	4.54	4.77	5.59	6.40	7.69
∇f, reverse AD								
Relative $n = 1$	1.52	11.12	31.37	67.27	113.99	174.62	254.15	342.33
Relative n in column	1.52	2.16	2.16	2.31	2.16	2.07	1.99	1.96

自动微分实现类型

Table 5: Survey of AD implementations. Tools developed primarily for machine learning are highlighted in bold.

Language	Tool	Type	Mode	Institution / Project	Reference	URL
AMPL	AMPL	INT	F, R	Bell Laboratories	Fourer et al. (2002)	http://www.ampl.com/
C, C++	ADIC	ST	F, R	Argonne National Laboratory	Bischof et al. (1997)	http://www.mcs.anl.gov/research/projects/adic/
	ADOL-C	OO	F, R	Computational Infrastructure for Operations Research	Walther and Griewank (2012)	https://projects.coin-or.org/ADOL-C
C++	Ceres Solver	LIB	F	Google		http://ceres-solver.org/
	CppAD	OO	F, R	Computational Infrastructure for Operations Research	Bell and Burke (2008)	http://www.coin-or.org/CppAD/
	FADBAD++	OO	F, R	Technical University of Denmark	Bendtsen and Stauning (1996)	http://www.fadbad.com/fadbad.html
	Mxyzptlk	OO	F	Fermi National Accelerator Laboratory	Ostiguy and Michelotti (2007)	
C#	AutoDiff	LIB	R	George Mason Univ., Dept. of Computer Science	Shtof et al. (2013)	http://autodiff.codeplex.com/
F#, C#	**DiffSharp**	OO	F, R	Maynooth University, Microsoft Research Cambridge	Baydin et al. (2016a)	http://diffsharp.github.io
Fortran	ADIFOR	ST	F, R	Argonne National Laboratory	Bischof et al. (1996)	http://www.mcs.anl.gov/research/projects/adifor/
	NAGWare	COM	F, R	Numerical Algorithms Group	Naumann and Riehme (2005)	http://www.nag.co.uk/nagware/Research/ad_overview.asp
	TAMC	ST	R	Max Planck Institute for Meteorology	Giering and Kaminski (1998)	http://autodiff.com/tamc/
Fortran, C	COSY	INT	F	Michigan State Univ., Biomedical and Physical Sci.	Berz et al. (1996)	http://www.bt.pa.msu.edu/index_cosy.htm
	Tapenade	ST	F, R	INRIA Sophia-Antipolis	Hascoët and Pascual (2013)	http://www-sop.inria.fr/tropics/tapenade.html
Haskell	ad	OO	F, R	Haskell package		http://hackage.haskell.org/package/ad
Java	ADiJaC	ST	F, R	University Politehnica of Bucharest	Slusanschi and Dumitrel (2016)	http://adijac.cs.pub.ro
	Deriva	LIB	R	Java & Clojure library		https://github.com/lambder/Deriva
Julia	JuliaDiff	OO	F, R	Julia packages	Revels et al. (2016a)	http://www.juliadiff.org/
Lua	**torch-autograd**	OO	R	Twitter Cortex		https://github.com/twitter/torch-autograd
MATLAB	ADiMat	ST	F, R	Technical University of Darmstadt, Scientific Comp.	Willkomm and Vehreschild (2013)	http://adimat.sc.informatik.tu-darmstadt.de/
	INTLab	OO	F	Hamburg Univ. of Technology, Inst. for Reliable Comp.	Rump (1999)	http://www.ti3.tu-harburg.de/rump/intlab/
	TOMLAB/MAD	OO	F	Cranfield University & Tomlab Optimization Inc.	Forth (2006)	http://tomlab.biz/products/mad
Python	ad	OO	R	Python package		https://pypi.python.org/pypi/ad
	autograd	OO	F, R	Harvard Intelligent Probabilistic Systems Group	Maclaurin (2016)	https://github.com/HIPS/autograd
	Chainer	OO	R	Preferred Networks	Tokui et al. (2015)	https://chainer.org/
	PyTorch	OO	R	PyTorch core team	Paszke et al. (2017)	http://pytorch.org/
	Tangent	ST	F, R	Google Brain	van Merriënboer et al. (2017)	https://github.com/google/tangent
Scheme	R6RS-AD	OO	F, R	Purdue Univ., School of Electrical and Computer Eng.		https://github.com/qobi/R6RS-AD
	Scmutils	OO	F	MIT Computer Science and Artificial Intelligence Lab.	Sussman and Wisdom (2001)	http://groups.csail.mit.edu/mac/users/gjs/6946/refman.txt
	Stalingrad	COM	F, R	Purdue Univ., School of Electrical and Computer Eng.	Pearlmutter and Siskind (2008)	http://www.bcl.hamilton.ie/~qobi/stalingrad/

目前大部分AD方法大致可分为以下几种：

- 元素型（elemental）：这类实现方法主要通过将任意函数分解为有限个基础AD算子，并用基础AD算子替代基础数学算子来实现自动微分。在没有运算重载符的语言环境中，这种方法比较适用；
- 编译和源码转换型（compilers and source code transformation）：用另一种语法或扩展语言编写运算，然后再转换到原始编程语言上，比如用数学标记来表达目标函数和约束，再用解释器或编译器分析为编程语言；
- 运算符重载型（operator overloading）：现代编程语言支持运算符重载，这使得元素型的AD方法更加容易实现。

名称	编程语言	实现方法	支持模式	地址
torch-autograd	Lua	运算符重载	反向	https://github.com/twitter/torch-autograd
autograd	Python	运算符重载	前向/反向	https://github.com/HIPS/autograd
Chainer	Python	运算符重载	反向	https://chainer.org/
PyTorch	Python	运算符重载	反向	https://pytorch.org/
Tangent	Python	源码转换	前向/反向	https://github.com/google/tangent

源码转换型AD方法示例

Tangent库通过对Python抽象语法树的修改，为部分系统数学运算以及numpy部分基础运算添加了自定义的求导函数并自动生成代码。具体代码尚未完全理解，这里自己简单记录下原理，并附上一些简单的代码辅助说明。

我们先得理解Python代码的执行过程：

> 语法分析 $\Longrightarrow$ 具体语法树 $\Longrightarrow$ 抽象语法树 $\Longrightarrow$ 控制流图 $\Longrightarrow$ 字节码 $\Longrightarrow$ 执行

Tangent库就用到了gast库（以ast库作为基础）对抽象语法树进行读取和补充。所以其中的关键就是如何利用ast和抽象语法树。

看一个简单的例子。先在代码中嵌入编写expr这一段Python代码，其中包括一个add函数，用来计算两个输入的和，然后执行并print：

```
>>> import ast
>>> expr = """
... def add(x,y):
...     return x + y
... print(add(3,4))
... """
>>> expr_ast = ast.parse(expr)
>>> expr_ast
<_ast.Module object at 0x7f61f2a58ac8>
```

expr经过ast模块解析后，得到的抽象语法树如下：

```
>>> ast.dump(expr_ast)
Module(
    body=[
        FunctionDef(
            name='add',
            args=arguments(
                args=[
                    arg(arg='x', annotation=None),
                    arg(arg='y', annotation=None)
                ],
```

```
                vararg=None,
                kwonlyargs=[],
                kw_defaults=[],
                kwarg=None,
                defaults=[]
            ),
            body=[
                Return(
                    value=BinOp(
                        left=Name(id='x', ctx=Load()),
                        op=Add(),
                        right=Name(id='y', ctx=Load()))
                )
            ],
            decorator_list=[],
            returns=None
        ),
        Expr(
            value=Call(
                func=Name(id='print', ctx=Load()),
                args=[
                    Call(
                        func=Name(id='add', ctx=Load()),
                        args=[Num(n=3), Num(n=4)],
                        keywords=[]
                    )
                ],
                keywords=[]
            )
        )
    ]
)
```

可以看到，expr中自定义的函数在抽象语法树中位于FunctionDef这个field中，而其中具体算子（+）位于FunctionDef.body.Return.value中，名为BinOp，具体操作为Add()。接下来我们通过ast的转换模块，将BinOp这个域中的Add()函数修改为乘法（ast.Mult()）。

定义一个转换类，将ast中的结点进行修改。由于目标结点是BinOp，所以在其中定义一个visit_BinOp函数，并将其中op域替换为ast.Mult()：

```
>>> class Transformer(ast.NodeTransformer):
...     def visit_BinOp(self, node):
...         node.op = ast.Mult()
...         return node
...
>>> trans = Transformer()
```

执行一下原始expr中的代码，$3+4=7$，+号执行的是加法：

```
>>> exec(compile(expr_ast, '<string>', 'exec'))
7
```

接下来我们替换掉其中的加法：

```
>>> modified = trans.visit(expr_ast)  # visit会调用所有visit_<classname>的方法
>>> ast.dump(modified)
Module(
    body=[
```

```
5         FunctionDef(
6             name='add',
7             args=arguments(
8                 args=[
9                     arg(arg='x', annotation=None),
10                    arg(arg='y', annotation=None)
11                ],
12                vararg=None,
13                kwonlyargs=[],
14                kw_defaults=[],
15                kwarg=None,
16                defaults=[]
17            ),
18            body=[
19                Return(
20                    value=BinOp(
21                        left=Name(id='x', ctx=Load()),
22                        op=Mult(),
23                        right=Name(id='y', ctx=Load())
24                    )
25                )
26            ],
27            decorator_list=[],
28            returns=None
29        ),
30        Expr(
31            value=Call(
32                func=Name(id='print', ctx=Load()),
33                args=[
34                    Call(
35                        func=Name(id='add', ctx=Load()),
36                        args=[Num(n=3), Num(n=4)],
37                        keywords=[])],
38                keywords=[]
39            )
40        )
41    ]
42 )
```

可以看到，在第22行，原来BinOp域里的op已经被替换为了x乘法。执行一下新的抽象语法树：

```
>>> exec(compile(modified, '<string>', 'exec'))
12
```

结果变成了$3 \times 4 = 12$。

这个例子说明，我们能够通过ast模块注入并修改源代码。此处再给出一个例子，调用numpy.add（也就是numpy.ndarray的加法）然后通过ast注入修改为了减法。由于numpy.add并非系统函数，所以抽象语法树有些不同：

```python
import ast
expr = """
import numpy as np
def add(x, y):
    out = np.add(x, y)
    return out
a = np.zeros((1,3))
b = np.ones((1,3))
print(add(a, b))
"""

expr_ast = ast.parse(expr)
print(ast.dump(expr_ast))
```

获得上述代码的抽象语法树为：

```
Module(
    body=[
        Import(
            names=[alias(name='numpy', asname='np')]
        ),
        FunctionDef(
            name='add',
            args=arguments(
                args=[arg(arg='x', annotation=None),
                      arg(arg='y', annotation=None)],
                vararg=None,
                kwonlyargs=[],
                kw_defaults=[],
                kwarg=None,
                defaults=[]
            ),
            body=[
                Assign(
                    targets=[Name(id='out', ctx=Store())],
                    value=Call(
                        func=Attribute(
                            value=Name(id='np', ctx=Load()),
                            attr='add',
                            ctx=Load()
                        ),
                        args=[Name(id='x', ctx=Load()),
                              Name(id='y', ctx=Load())],
                        keywords=[]
                    )
                ),
                Return(
                    value=Name(id='out', ctx=Load())
                )
            ],
            decorator_list=[],
            returns=None
        ),
        Assign(
            targets=[Name(id='a', ctx=Store())],
            value=Call(
                func=Attribute(
                    value=Name(id='np', ctx=Load()),
                    attr='ones',
                    ctx=Load()
```

```
45                 ),
46                 args=[Tuple(elts=[Num(n=1), Num(n=3)], ctx=Load())],
47                 keywords=[]
48             )
49         ),
50         Assign(
51             targets=[Name(id='b', ctx=Store())],
52             value=Call(
53                 func=Attribute(
54                     value=Name(id='np', ctx=Load()),
55                     attr='ones',
56                     ctx=Load()
57                 ),
58                 args=[Tuple(elts=[Num(n=1), Num(n=3)], ctx=Load())],
59                 keywords=[])
60         ),
61         Expr(
62             value=Call(
63                 func=Name(id='print', ctx=Load()),
64                 args=[
65                     Call(
66                         func=Name(id='add', ctx=Load()),
67                         args=[Name(id='a', ctx=Load()), Name(id='b', ctx=Load())],
68                         keywords=[]
69                     )
70                 ],
71                 keywords=[]
72             )
73         )
74     ]
75 )
```

注入目标numpy.add位于第23行Attribute.attr内，所以修改的点就在这里：

```
class EvilTransformer(ast.NodeTransformer):
    def visit_Attribute(self, node):
        if node.attr == 'add':
            node.attr = 'subtract'  # numpy中减法为numpy.subtract
        return node

trans = EvilTransformer()
new_ast = trans.visit(expr_ast)
exec(compile(new_ast, '<string>', 'exec'))
```

理论上，注入前，expr执行的结果应该是[[1.,1.,1.]]；注入后，输出结果就会变成[[-1.,-1.,-1.]]，成功将numpy.add改成了numpy.subtract。

注意，虽然trans.visit返回了“新”的抽象语法树变量new_ast，实际上该函数是对expr_ast直接进行了修改。所以new_ast和expr_ast是同一个变量的两个别名。

Tangent自动微分库

最后，简单分析下Tangent库是如何注入的。Tangent库中grads.py文件中有关键的注入几个注入装饰器函数，用于对各个算子进行注入，“自动生成”对应的导函数：

- tangent.grads.create_register

```python
def create_register(dict_):
  def register(key):
    def _(f):
      dict_[key] = f
      return f
    return _
  return register
```

这一函数用于生成一个作为装饰器的注册机。注册信息保存在dict_这一变量中。通过这类注册机，可以自定义某特定函数的微分函数。需要注意的是，**不需要该微分函数能运行，它只是作为模板用于自动生成。**

- tangent.grad
  - 用于对某个函数进行求微分；
  - 适用于$f:\mathbb{R}^n\rightarrow\mathbb{R}$类型的函数；
  - 会检查输入是否为标量。
- tangent.autodiff
  - `autodiff(f, mode='forward')`：AD前向模式，调用的关键函数：ForwardAD
  - `autodiff(f, mode='reverse')`：AD反向模式，调用的关键函数：ReverseAD

上述几个函数将会分析目标函数，并根据注册机内的微分函数模板，进行注入，然后生成一个对应的微分函数的ast抽象语法树，然后通过astor包将该抽象语法树转化回python源码。

hook技术

PyTorch使用了一种hook方法来捕捉模型在前馈和反馈时的中间数据。由于AD的设计，调用损失的backward方法后各结点的梯度逐个计算完成并释放计算图，所以无法通过模型来获得中间结果的一些数据，所以使用了**钩子**（hook）技术来抓取这些数据保存到一个新的变量中。

PyTorch中有四种钩子：

- torch.tensor.register_hook(self, hook)
  - 用于tensor；
  - 在每次计算完tensor的梯度时都会调用其中的钩子；
  - 不能修改数据，只能获得一个新的变量用于保存新的梯度（通过tensor.grad获取）；
  - 钩子签名必须为：`hook(grad) -> Tensor or None`。
- torch.nn.Module.register_forward_pre_hook(self, hook)
  - 用于Module；
  - 在每次调用forward函数**前**都会调用其中的钩子，主要用于各类Norm模块/层；
  - 可以修改输入数据；
  - 钩子签名必须为：`hook(module, input) -> None or modified input`
- torch.nn.Module.register_forward_hook(self, hook)
  - 用于Module；
  - 在每次调用forward函数后，backward之前都会调用其中的钩子；
  - 可以修改输出数据，也可以原址修改输入数据，但是不会影响前馈结果（因为执行在forward之后）；
  - 钩子签名必须为：`hook(module, input, output) -> None or modified output`
- torch.nn.Module.register_backward_hook(self, hook)
  - 用于Module；
  - 在每次计算完输入数据的梯度后都会调用其中的钩子；
  - 不能修改数据，但是可以返回一个新变量包含其中的梯度数据（通过Module.grad_input获取）；

- 钩子签名必须为：hook(module, grad_input, grad_output) -> Tensor or None；其中grad_input和grad_output可以是tuple。

我们举个例子来说明各种钩子的作用。首先，定义一个简单的模块，其中包含一个大小为3x1的参数：

In [1]:
```
import torch
import torch.nn as nn
```

In [2]:
```
class TestModule(nn.Module):
    def __init__(self):
        super(TestModule, self).__init__()
        self.w = torch.rand((3,1), requires_grad=True)

    def forward(self, x):
        return torch.matmul(x, self.w)
```

随后我们定义三个钩子函数，分别用于tensor.register_hook，Module.register_forward_hook和Module.register_backward_hook，并且读取相应的数据：

In [3]:
```
grad_list = []
def tensor_hook(grad):
    grad_list.append(grad)

forward_in_list = []
forward_out_list = []
def forward_hook(module, input, output):
    forward_in_list.extend(list(input))
    forward_out_list.extend(list(output))

backward_grad_in_list = []
backward_grad_out_list = []
def backward_hook(module, grad_input, grad_output):
    backward_grad_in_list.extend(list(grad_input))
    backward_grad_out_list.extend(list(grad_output))
```

我们定义运算引入中间变量y：

In [4]:
```
x = torch.rand((1,3), requires_grad=True)
y = x + 2
model = TestModule()
```

先来观察下各个数据：

In [5]:
```
x
```

Out[5]:
```
tensor([[0.1545, 0.0325, 0.8274]], requires_grad=True)
```

In [6]:
```
y
```

Out[6]:
```
tensor([[2.1545, 2.0325, 2.8274]], grad_fn=<AddBackward0>)
```

In [7]:
```
model.w
```

Out[7]:
```
tensor([[0.7566],
        [0.4152],
        [0.7712]], requires_grad=True)
```

注册钩子：

In [8]:
```
y.register_hook(tensor_hook)
model.register_forward_hook(forward_hook)
model.register_backward_hook(backward_hook)
```

Out[8]:
```
<torch.utils.hooks.RemovableHandle at 0x7f945eb3def0>
```

然后我们调用模块，并反传：

In [9]:
```
z = model(y)
z.backward()
```

来看下y.grad，发现是NoneType：

In [11]:
```
type(y.grad)
```

Out[11]:
```
NoneType
```

但是y的钩子函数捕捉到了数据，放在了grad_list这个列表中。各钩子捕捉到的数据：

```
In [12]: grad_list
Out[12]: [tensor([[0.7566, 0.4152, 0.7712]])]

In [13]: forward_in_list
Out[13]: [tensor([[2.1545, 2.0325, 2.8274]], grad_fn=<AddBackward0>)]

In [14]: forward_out_list
Out[14]: [tensor([4.6544], grad_fn=<SelectBackward>)]

In [15]: backward_grad_in_list
Out[15]: [tensor([[0.7566, 0.4152, 0.7712]]), tensor([[2.1545],
                 [2.0325],
                 [2.8274]])]

In [16]: backward_grad_out_list
Out[16]: [tensor([[1.]])]
```

由此可以看到，通过对不同阶段的数据使用钩子，我们可以容易得获得中间变量/模块的数值/梯度等数据，并在其他任务中进行分析和处理。

个人认为钩子函数主要适用于对中间变量、特征图的数值和梯度的提取，这在对抗样本、迁移学习等邻域可能较为常用。而对于`torch.tensor`定义的变量（比如上例中的`x`）和模块参数（上例中的`model.w`），无需使用钩子技术。

资源

内容	网址	备注
自动微分社区	http://www.autodiff.org/	有关自动微分的内容

参考资料

[1] 自动微分(Automatic Differentiation)简介

[2] Automatic Differentiation in Machine Learning: a Survey

[3] Dual Numbers & Automatic Differentiation

[4] CSE 599W： Systems for ML博客

[5] CSE 599W： Systems for ML

[6] PyTorch 学习笔记（六）： PyTorch hook 和关于 PyTorch backward 过程的理解

[7] pytorch中的钩子（Hook）有何作用?

[8] 详解Pytorch中的网络构造

[9] [python] ast模块

[10] ast --- 抽象语法树