自动微分

庞龙刚@华中师范大学

第十届华大 QCD 讲习班

学习内容

- 机器学习中梯度计算的重要性
- 解析微分和有限差分的缺陷
- 前向模式的自动微分
- 后向模式的自动微分
- 手写一个可训练的人工神经网络

机器学习中梯度计算的重要性

深度学习中需要梯度下降算法，更新神经网络的可训练参数，

$$\theta_{n+1} = \theta_n - \epsilon \frac{\partial l}{\partial \theta_n}$$

在机器学习的可解释性研究中，需要梯度上升，将一张白纸（输入 x）更新成图片，以图最大程度激活某个神经元的输出 y，

$$x_{n+1} = x_n + \epsilon \frac{\partial y}{\partial x}$$

在 Physics Informed Neural Network 或 Physics Based Deep Learning 中，

1. 神经网络往往用作可微分的泛函模块 $f(x, \theta)$, 物理的偏微分方程需要 $\frac{\partial f}{\partial x}$ 项
2. 求解逆问题时，物理学规律可以构建进可微分训练的过程

下面对比三种求梯度的方法：解析微分，有限差分和自动微分。

解析微分

手动推导，或使用计算机代数系统（Mathematica 或 Sympy）。

In [198…

```
import sympy as sym

# Step1: define the variable
x = sym.symbols("x")

f = x * sym.exp(- x**2)

dfdx = sym.diff(f, x)
```

```
dfdx
```

Out[198]: $-2x^2e^{-x^2}+e^{-x^2}$

In [6]:
```
dfdx.subs(x, 0)
```

Out[6]: 1

解析微分的缺陷

需要手动或机器推导，有严重的表达式膨胀问题。

举例：迭代函数，$l_{n+1}=4l_n(1-l_n), l_1=x$

根据迭代规则，$l_2=4x(1-x)$。

In [199…
```
ln = l1 = x

for i in range(2, 5):
    ln = 4 * ln * (1 - ln)
    sym.pretty_print(ln)
```

```
4·x·(1 - x)
16·x·(1 - x)·(-4·x·(1 - x) + 1)
64·x·(1 - x)·(-4·x·(1 - x) + 1)·(-16·x·(1 - x)·(-4·x·(1 - x) + 1) + 1)
```

In [200…
```
ln = l1 = x

for i in range(2, 5):
    ln = 4 * ln * (1 - ln)
    dldx = sym.diff(ln, x)
    sym.pretty_print(dldx)
```

```
4 - 8·x
16·x·(1 - x)·(8·x - 4) - 16·x·(-4·x·(1 - x) + 1) + 16·(1 - x)·(-4·x·(1 - x) +
1)
64·x·(1 - x)·(8·x - 4)·(-16·x·(1 - x)·(-4·x·(1 - x) + 1) + 1) + 64·x·(1 - x)·(
-4·x·(1 - x) + 1)·(-16·x·(1 - x)·(8·x - 4) + 16·x·(-4·x·(1 - x) + 1) - 16·(1 -
 x)·(-4·x·(1 - x) + 1)) - 64·x·(-4·x·(1 - x) + 1)·(-16·x·(1 - x)·(-4·x·(1 - x)
 + 1) + 1) + 64·(1 - x)·(-4·x·(1 - x) + 1)·(-16·x·(1 - x)·(-4·x·(1 - x) + 1) +
 1)
```

有限差分算法

对于 $R^n \to R$ 的映射函数，f 对 $\vec{x}$ 的第 i 个分量的偏导数等于，

$$\frac{\partial f(\vec{x})}{\partial x_i} \approx \frac{f(\vec{x}+h\vec{e}_i)-f(\vec{x})}{h}$$

其中 $\vec{e}_i$ 是第 i 个单位矢量，h 是个很小的正数。

In [201…
```
import numpy as np
import matplotlib.pyplot as plt
import mplhep as hep
hep.styles.use("ATLAS")
```

In [202…
```
def finite_difference(func, x, dx=0.01):
    return (func(x+dx) - func(x)) / dx

def f(x):
    return x * np.exp(- x**2)

dfdx = finite_difference(f, x=0)
dfdx  # 解析解 = 1，万分之一的误差
```

Out[202]:
```
0.9999000049998332
```

有限差分算法的缺陷

1. 神经网络参数众多，如果有一千万个参数，则需要把每个参数加一个小量 h，运行一次神经网络，复杂度为 $O(N)$
2. 有限差分算法的误差不好控制。

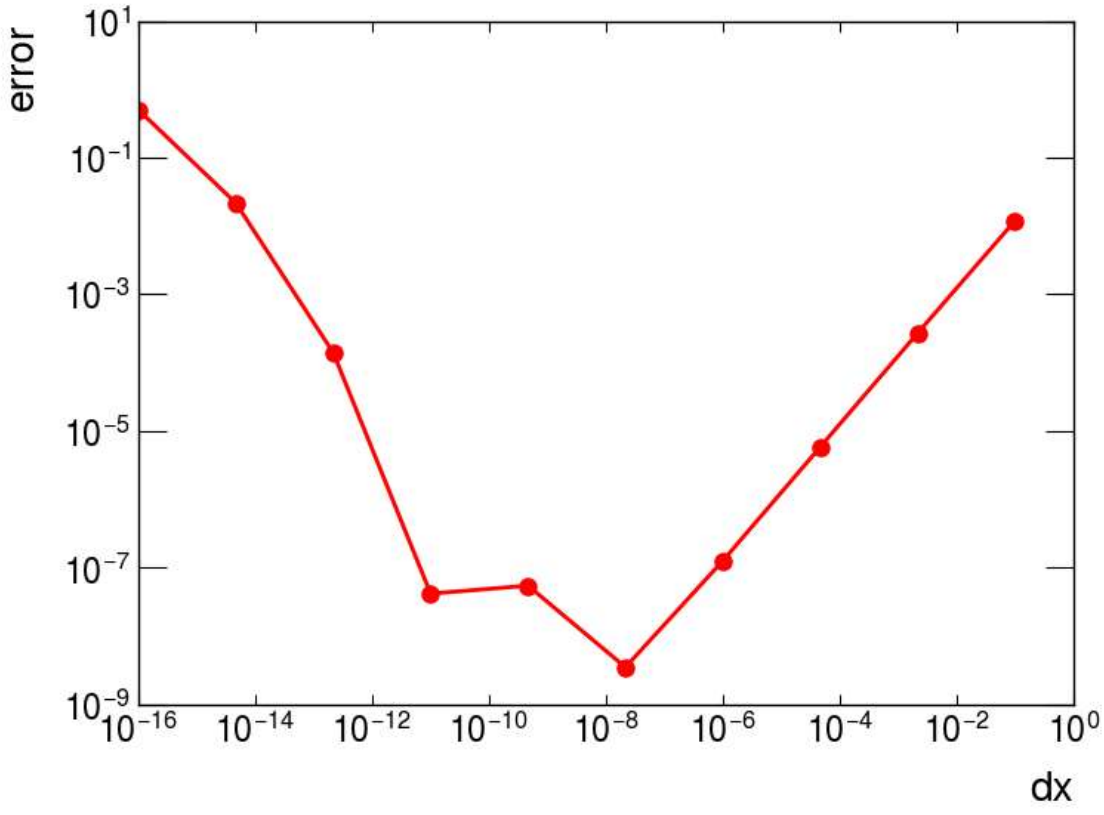
举例：$f(x) = \sqrt{x}$ 在 $x = 1.0$ 处的导数，

In [23]:
```
def check_error(dx=0.1, x0=1.0):
    num = finite_difference(np.sqrt, x0, dx)
    # 计算有限差分算法与解析解之间的误差
    ana = 0.5   # 解析解
    return np.abs(ana - num)
```

In [203…
```
dx = np.logspace(-16, -1, 10)
# dx 为从 10^{-16} 到 10^{-1} 的等比序列
err = check_error(dx)

plt.loglog(dx, err, 'ro-')
plt.xlabel("dx")
plt.ylabel("error")
```

Out[203]:
```
Text(0, 1, 'error')
```



自动微分

自动微分保留了数值微分速度快和解析微分结果精确的优点，正成为一种新的计算物理编程范式。

自动微分在人工智能时代开始流行，因为 Tensorflow, pytorch 等深度学习库封装了自动微分功能。

参考文献：

1. 王磊，刘金国《自动微分编程》
2. Automatic Differentiation in Machine Learning: a Survey
3. Reverse-mode automatic differentiation from scratch
4. Tensorflow 中的自动微分

自动微分通过在计算机上实现一些基本算术操作的微分，加上链式规则，自动从用户编写的函数 f(x)，推导出它的微分 f'(x)。

自动微分广泛应用于，

- 人工智能 $\theta = \theta - \epsilon \partial L/\partial \theta$
- 逆问题 $y = f(x) \to x = f^{-1}(y)$
- 牛顿法寻根 $x_{n+1} = x_n - f(x_n)/f'(x_n)$
- Stiff 常微分方程 $df/dt = s$

自动微分有前向和后向模式，

如果函数是 $R^1 \to R^n$ 空间的映射，前向模式更高效。

如果函数是 $R^n \to R^1$ 空间的映射，则后向模式更高效。

人工神经网络的输出一般是损失函数，为标量，神经网络的参数个数 n 巨大，函数映射为 $R^n \to R^1$，因此人工神经网络的训练使用后向模式的自动微分。

下面分别介绍自动微分的前向与后向模式，

自动微分的前向模式

方法：把所有的数加上一个对偶项 $x \to x + \dot{x}\mathbf{d}$

其中 $\mathbf{d}$ 是一个符号，像虚数的表示符号 i 一样。不同的是，$i^2 = -1$, 此处 $\mathbf{d}^2 = 0$.

可以把 $\mathbf{d}$ 理解成一个无穷小量。使用这种方法, 用户定义的函数会自动出现一个对偶的自动微分项, 以 $\mathbf{d}$ 标示。

下面举例说明,

自动微分中对偶数的代数运算 (注意 $\mathbf{d}^2 = 0$)

$$(x+\dot{x}\mathbf{d})+(y+\dot{y}\mathbf{d})=x+y+(\dot{x}+\dot{y})\mathbf{d} \tag{1}$$
$$(x+\dot{x}\mathbf{d})-(y+\dot{y}\mathbf{d})=x-y+(\dot{x}-\dot{y})\mathbf{d} \tag{2}$$
$$(x+\dot{x}\mathbf{d})*(y+\dot{y}\mathbf{d})=xy+(x\dot{y}+\dot{x}y)\mathbf{d} \tag{3}$$
$$\frac{1}{x+\dot{x}\mathbf{d}}=\frac{1}{x}-\frac{\dot{x}}{x^2}\mathbf{d}\quad(x\neq 0) \tag{4}$$

将多项式扩展到对偶数域

$$P(x)=a_0+a_1x+a_2x^2+\cdot+a_nx^n \tag{5}$$

将 x 替换成对偶数，$x\to x+\dot{x}\mathbf{d}$ 得到，

$$P(x+\dot{x}\mathbf{d})=a_0+a_1(x+\dot{x}\mathbf{d})+a_2(x+\dot{x}\mathbf{d})^2+\cdot+a_n(x+\dot{x}\mathbf{d})^n \tag{6}$$
$$=a_0+a_1x+a_2x^2+\cdot+a_nx^n+a_1\dot{x}\mathbf{d}+2a_2x\dot{x}\mathbf{d}+\cdot+na_nx^{n-1}\dot{x}\mathbf{d} \tag{7}$$
$$=P(x)+P'(x)\dot{x}\mathbf{d} \tag{8}$$

- $\dot{x}$ 可任意选择，当对 x 求导时，$\dot{x}=1$, 则对偶的项 $P'(x)$ 为 P(x) 的微分。
- 如果计算机中其它函数比如 cos, sin, log, exp 等都用多项式展开，则用户定义的任何函数都会自动生成其微分对偶。

或者直接对常用的数学函数进行重载，

$$sin(x+\dot{x}\mathbf{d})=sin(x)+cos(x)\dot{x}\mathbf{d} \tag{9}$$
$$cos(x+\dot{x}\mathbf{d})=cos(x)-sin(x)\dot{x}\mathbf{d} \tag{10}$$
$$e^{x+\dot{x}\mathbf{d}}=e^x+e^x\dot{x}\mathbf{d} \tag{11}$$
$$\log(x+\dot{x}\mathbf{d})=\log(x)+\frac{\dot{x}}{x}\mathbf{d}\quad x\neq 0 \tag{12}$$
$$\sqrt{x+\dot{x}\mathbf{d}}=\sqrt{x}+\frac{\dot{x}}{2\sqrt{x}}\mathbf{d}\quad x\neq 0 \tag{13}$$
$$(x+\dot{x}\mathbf{d})^2=x^2+2x\dot{x}\mathbf{d} \tag{14}$$

运算符重载的方法实现自动微分编程

举例：计算机如何做复数的加减乘除

$$z_1=x_1+iy_1 \tag{15}$$
$$z_2=x_2+iy_2 \tag{16}$$

底层语言只实现了实数的加减乘除，但是使用复数库，将 z_1，z_2 定义成复数，可以直接计算

- z_1+z_2
- z_1-z_2
- z_1*z_2
- z_1/z_2

复数库对 +，-，*，/ 符号做了重载。

接下来构建一个对偶数类，让其可以直接进行前向模式的自动微分。

In [82]:
```
class DNumber:
    '''自动微分中的对偶数类，对 +, -, *, /, 幂次符号做了重载'''
    def __init__(self, x, dx):
```

```
        self.val = x
        self.dval = dx

    def __repr__(self):
        '''使用 print(DNumber(1, 2)) 时输出：1 + 2 d'''
        return f'{self.val} + {self.dval} d'

    def __add__(self, other):
        '''overload a + b'''
        if isinstance(other, float) or isinstance(other, int):
            val = self.val + other
            dval = self.dval
        if isinstance(other, DNumber):
            val = self.val+other.val
            dval = self.dval + other.dval

        return DNumber(val, dval)


    def __sub__(self, other):
        '''overload a - b'''
        if isinstance(other, float) or isinstance(other, int):
            val = self.val - other
            dval = self.dval
        if isinstance(other, DNumber):
            val = self.val-other.val
            dval = self.dval - other.dval

        return DNumber(val, dval)

    def __iadd__(self, other):
        '''overload a += b'''
        self.val = self.val + other.val
        self.dval = self.dval + other.dval
        return self

    def __mul__(self, other):
        ''' overload x * y  or const * x
        (x + dx d)*(y + dy d) = x*y + (xdy + ydx) d'''
        if isinstance(other, float) or isinstance(other, int):
            val = other * self.val
            dval = other * self.dval

        if isinstance(other, DNumber):
            val = self.val * other.val
            dval = self.val * other.dval + other.val * self.dval

        return DNumber(val, dval)

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

    def __pow__(self, n):
        if isinstance(n, float) or isinstance(n, int):
            val = self.val**n
            dval = n * self.val**(n-1) * self.dval
        elif isinstance(n, DNumber) :
            raise(Exception("Pow(DNumber, DNumber) is not implemented yet"))
        return DNumber(val, dval)
```

In [205…

```
# 举例：
x = DNumber(0.1, 1)
x * x
```

Out[205]:
```
0.010000000000000002 + 0.2 d
```

In [206…
```
x**3
```

Out[206]:
```
0.0010000000000000002 + 0.030000000000000006 d
```

In [207…
```
np.cos(x)
# numpy 的 cos 函数不能作用在 DNumber 上
# 需要自己定义 cos 函数
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
AttributeError: 'DNumber' object has no attribute 'cos'

The above exception was the direct cause of the following exception:

TypeError                                 Traceback (most recent call last)
Input In [207], in <cell line: 1>()
----> 1 np.cos(x)

TypeError: loop of ufunc does not support argument 0 of type DNumber which has no c
allable cos method
```

In [208…
```
import math
def cos(x:DNumber):
    '''the cos value of DNumber'''
    return DNumber(math.cos(x.val), -math.sin(x.val)*x.dval)

def sin(x:DNumber):
    '''the sin value of DNumber'''
    return DNumber(math.sin(x.val), math.cos(x.val)*x.dval)

def sqrt(x:DNumber):
    return DNumber(math.sqrt(x.val), 0.5/math.sqrt(x.val)*x.dval)

def exp(x:DNumber):
    exp_x = math.exp(x.val)
    return DNumber(exp_x, exp_x * x.dval)

def log(x:DNumber):
    return DNumber(math.log(x.val), x.dval / x.val)
```

In [209…
```
print("x=", x)
cos(x)
```

Out[209]:
```
x= 0.1 + 1 d
0.9950041652780258 + -0.09983341664682815 d
```

多个变量的前向自动微分

上面例子中 f(x) 是 x 的单变量函数，$\dot{x}=1.$

如果是多变量函数 f(x, y), 要求 f 对 x 的偏导数，则应取

$\dot{x}=1, \dot{y}=0$，调用函数计算 $f(x,y)$,

如果还想计算 f 对 y 的导数，需要设

$\dot{x}=0, \dot{y}=1.$

重新计算一次 $f(x,y)$。

如果函数 f 是 $R^n \to R^1$ 映射，且想计算 f 对每个输入变量的偏导数， 则在前向自动微分中， f 也要计算 n 次。

举例说明：

$$f(x,y) = 3x^2y + x$$

$$\frac{\partial f}{\partial x} = 6xy + 1$$

在 (x, y) = (1, 1) 时， $f(x,y) = 4.0$, $\frac{\partial f}{\partial x} = 7.0$。

$$\frac{\partial f}{\partial y} = 3x^2$$

在 (x, y) = (1, 1) 时， $f(x,y) = 4.0$, $\frac{\partial f}{\partial y} = 3.0$。

In [213…
```
def f(x, y):
    return 3 * x**2 * y + x

x = DNumber(1, 1)
y = DNumber(1, 0)
f(x, y)
```

Out[213]:
```
4 + 7 d
```

In [215…
```
x = DNumber(1, 0)
y = DNumber(1, 1)
f(x, y)
```

Out[215]:
```
4 + 3 d
```

前向自动微分的简单应用:梯度上升法寻找极值

举例：一个铅球运动员为了获得最佳成绩，需要计算铅球出手高度为 h，速度大小为 v 时，最佳出射角度。根据

$$x(t) = vt\cos\theta \tag{17}$$

$$y(t) = h + vt\sin\theta - \frac{1}{2}gt^2 \tag{18}$$

将 $y(t) = 0$ 时的 t 代入 $x(t)$, 得到成绩 x_{max} 与 θ 的关系，

$$x_{max}(\theta) = \frac{1}{g}v\cos\theta\left(v\sin\theta + \sqrt{v^2\sin^2\theta + 2gh}\right)$$

画图可以发现对于 v=14 m/s, h=1.8 m, 当 θ 取 0.75 附近的某个数字时， x_{max} 最大。

接下来使用前向自动微分和梯度上升，寻找 θ 使得 x_{max} 最大。

In [216…
```
def xmax(theta, v=14, h=1.8, g=9.8):
    return v / g * cos(theta) * (v * sin(theta) + sqrt(v*v*sin(theta)**2 + 2 * g *

theta = DNumber(0.8, 1.0)
```

```
xmax(theta)
```

Out[216]:
```
21.72425470037264 + -0.44049923585322404 d
```

In [217…
```
lr = 0.01

theta = DNumber(0.8, 1.0)
for i in range(8):
    theta = theta + lr * xmax(theta).dval
    print(theta)
```

```
0.7581471901981273 + 1.0 d
0.747677117139374 + 1.0 d
0.7449979364087943 + 1.0 d
0.7443072794272741 + 1.0 d
0.7441288808948382 + 1.0 d
0.7440827759869065 + 1.0 d
0.7440708591255571 + 1.0 d
0.7440677788339284 + 1.0 d
```

举例：实现自动微分版本的牛顿法寻根，

$$\theta_{n+1} = \theta_n - \frac{f(\theta_n)}{f'(\theta_n)}$$

寻找 θ 使得下列方程近似满足：

$$f(\theta) = v\cos^2\theta - \sqrt{2gh + v^2\sin^2\theta}\sin\theta = 0$$

In [219…
```
def f(theta, v=14, h=1.8, g=9.8):
    return v * cos(theta)**2 - sqrt(v**2 * sin(theta)**2 + 2*g*h) * sin(theta)

theta = DNumber(0.8, 1)

for i in range(5):
    fx = f(theta)
    theta = theta - fx.val / fx.dval
    print(theta)
```

```
0.7440594805535422 + 1 d
0.744066705081479 + 1 d
0.7440667050778138 + 1 d
0.7440667050778138 + 1 d
0.7440667050778138 + 1 d
```

自动微分的后向模式 （reverse mode）

自动微分的后向模式也被称作 back propagation，它把微分从输出逐步向后传递到到给定的输入变量。

借助每个中间变量 v_i 的**伴随变量（adjoint）**，

$$\overline{v}_i = \frac{\partial y_j}{\partial v_i}$$

In [53]:
```
from IPython.display import Image
```

```
# Automatic Differentiation in Machine Learning: a survey  (Table 3)
Image("autodiff_reverse.png")
```

Out[53]:

Table 3: Reverse mode AD example, with $y = f(x_1, x_2) = \ln(x_1) + x_1 x_2 - \sin(x_2)$ 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		
$v_{-1} = x_1$		$= 2$	$\bar{x}_1 = \bar{v}_{-1}$		$= \mathbf{5.5}$
$v_0 = x_2$		$= 5$	$\bar{x}_2 = \bar{v}_0$		$= \mathbf{1.716}$
$v_1 = \ln v_{-1}$		$= \ln 2$	$\bar{v}_{-1} = \bar{v}_{-1} + \bar{v}_1 \frac{\partial v_1}{\partial v_{-1}}$	$= \bar{v}_{-1} + \bar{v}_1 / v_{-1}$	$= 5.5$
$v_2 = v_{-1} \times v_0$		$= 2 \times 5$	$\bar{v}_0 = \bar{v}_0 + \bar{v}_2 \frac{\partial v_2}{\partial v_0}$	$= \bar{v}_0 + \bar{v}_2 \times v_{-1}$	$= 1.716$
			$\bar{v}_{-1} = \bar{v}_2 \frac{\partial v_2}{\partial v_{-1}}$	$= \bar{v}_2 \times v_0$	$= 5$
$v_3 = \sin v_0$		$= \sin 5$	$\bar{v}_0 = \bar{v}_3 \frac{\partial v_3}{\partial v_0}$	$= \bar{v}_3 \times \cos v_0$	$= -0.284$
$v_4 = v_1 + v_2$		$= 0.693 + 10$	$\bar{v}_2 = \bar{v}_4 \frac{\partial v_4}{\partial v_2}$	$= \bar{v}_4 \times 1$	$= 1$
			$\bar{v}_1 = \bar{v}_4 \frac{\partial v_4}{\partial v_1}$	$= \bar{v}_4 \times 1$	$= 1$
$v_5 = v_4 - v_3$		$= 10.693 + 0.959$	$\bar{v}_3 = \bar{v}_5 \frac{\partial v_5}{\partial v_3}$	$= \bar{v}_5 \times (-1)$	$= -1$
			$\bar{v}_4 = \bar{v}_5 \frac{\partial v_5}{\partial v_4}$	$= \bar{v}_5 \times 1$	$= 1$
$y = v_5$		$= 11.652$	$\bar{v}_5 = \bar{y}$	$= 1$	

注意，从左边的前向模式看，v_0 对 y 的贡献有两项，所以

$$\frac{\partial y}{\partial v_0} = \frac{\partial y}{\partial v_2}\frac{\partial v_2}{\partial v_0} + \frac{\partial y}{\partial v_3}\frac{\partial v_3}{\partial v_0} = \overline{v}_2\frac{\partial v_2}{\partial v_0} + \overline{v}_3\frac{\partial v_3}{\partial v_0}$$

在右边逆向模式中，这两项贡献分两步累加得到，

$$\overline{v}_0 = \overline{v}_3\frac{\partial v_3}{\partial v_0} \quad and \quad \overline{v}_0 = \overline{v}_0 + \overline{v}_2\frac{\partial v_2}{\partial v_0}$$

可以看到，逆向过程只做一次，就得到了 y 对 x_1 和 x_2 两个变量的偏导数。

自动微分后向模式的实现

例子来自：Reverse-mode automatic differentiation from scratch, in Python

In [174…

```
from collections import defaultdict

class Variable:
    def __init__(self, value, local_gradients=[]):
        self.value = value
        self.local_gradients = local_gradients

    def __add__(self, other):
        return add(self, other)

    def __mul__(self, other):
        return mul(self, other)

    def __sub__(self, other):
        return sub(self, other)
```

```
    def __truediv__(self, other):
        return mul(self, inv(other))

def add(a, b):
    value = a.value + b.value
    local_gradients = ((a, 1),
                       (b, 1))
    return Variable(value, local_gradients)

def sub(a, b):
    value = a.value - b.value
    local_gradients = ((a, 1),
                       (b, -1))
    return Variable(value, local_gradients)

def mul(a, b):
    value = a.value * b.value
    local_gradients = ((a, b.value),
                       (b, a.value))
    return Variable(value, local_gradients)

def inv(a):
    value = 1 / a.value
    local_gradients = ((a, -1 / a.value**2),)
    return Variable(value, local_gradients)
```

In [175…

```
def get_gradients(variable):
    '''计算 variable 对它所有子节点的一阶导数
    :variable: 当前节点
    :adjoint: 伴随变量，输出节点 adjoint = 1'''
    gradients = defaultdict(lambda: 0)
    # 此 defaultdict 的 key 不存在时，会自动产生一个 key 并赋初值为 0
    def compute_gradients(variable, adjoint):
        # 对它的每个子变量，以及子变量的局部梯度递归
        for child_variable, local_gradient in variable.local_gradients:
            child_adjoint = adjoint * local_gradient
            gradients[child_variable] += child_adjoint
            # 逆向递归遍历计算图
            compute_gradients(child_variable, child_adjoint)

    compute_gradients(variable, adjoint=1)
    return gradients
```

In [221…

```
x1 = Variable(2)
x2 = Variable(5)

y = x1 * x2 - x1
```

In [222…

```
gradients = get_gradients(y)
gradients[x1]
```

Out[222]:

```
4
```

In [179…

```
gradients[x2]
```

Out[179]:

```
2
```

添加更多函数，比如 sin, cos, log, exp, 这样就可以计算之前的例子中 y 对 x_1 和 x_2 的导数，

$$y = f(x_1, x_2) = \log x_1 + x_1 x_2 - \sin x_2$$

In [225…
```
def cos(a):
    value = np.cos(a.value)
    local_gradients = ((a, -np.sin(a.value)),)
    return Variable(value, local_gradients)

def sin(a):
    value = np.sin(a.value)
    local_gradients = ((a, np.cos(a.value)),)
    return Variable(value, local_gradients)

def log(a):
    value = np.log(a.value)
    local_gradients = ((a, 1 / a.value),)
    return Variable(value, local_gradients)

def exp(a):
    value = np.exp(a.value)
    local_gradients = ((a, value),)
    return Variable(value, local_gradients)

def sqrt(a):
    value = np.sqrt(a.value)
    local_gradients = ((a, 0.5 / value ),)
    return Variable(value, local_gradients)
```

In [226…
```
x1 = Variable(2)
x2 = Variable(5)
y1 = log(x1) + x1 * x2 - sin(x2)
```

In [227…
```
gradients = get_gradients(y1)

print("dy/dx1 at (x1=2, x2=5)= ", gradients[x1])

print("dy/dx2 at (x1=2, x2=5)= ", gradients[x2])
```

```
dy/dx1 at (x1=2, x2=5)=  5.5
dy/dx2 at (x1=2, x2=5)=  1.7163378145367738
```

In [191…
```
x1, x2 = sym.symbols("x1, x2")
f = sym.log(x1) + x1 * x2 - sym.sin(x2)
dfdx1 = sym.diff(f, x1)
dfdx2 = sym.diff(f, x2)
print("Analytical solution: df/dx1=", dfdx1.subs(x1, 2).subs(x2, 5))
```

```
Analytical solution: df/dx1= 11/2
```

In [193…
```
print("Analytical solution: df/dx1=", dfdx2.subs(x1, 2).subs(x2, 5))
```

```
Analytical solution: df/dx1= 2 - cos(5)
```

In [194…
```
2 - np.cos(5.0)
```

Out[194]:
```
1.7163378145367738
```

注意

这个实现里没有考虑对向量、矩阵的自动微分，也没有考虑 N 阶导数。

但实现一次自动微分的后向模式可以帮助我们理解 tensorflow or Pytorch 编程。

比如，有了上面的基础，就很容易理解 Pytorch 中经常出现的这段代码

```
# zero the parameter gradients
 optimizer.zero_grad()

 # forward + backward + optimize
 outputs = net(inputs)
 loss = criterion(outputs, labels)
 loss.backward()
 optimizer.step()
```