Before you turn this problem in, make sure everything runs as expected. First, **restart the kernel** (in the menubar, select Kernel→Restart) and then **run all cells** (in the menubar, select Cell→Run All).

Make sure you fill in any place that says `YOUR CODE HERE` or "YOUR ANSWER HERE", as well as your name and collaborators below:

In [ ]:
```
NAME = ""
COLLABORATORS = ""
```

Differentiation options

Numpy ⟺ Pytorch

+ GPU

+ Automatic differentiation

def f($\underline{x}$): n

return

1. Numerical differentiation
2. Symbolic differentiation
3. Automatic differentiation
    A. Forward mode differentiation
    B. Reverse mode differentiation

→ check_numerical_jacobian

$$\left.\frac{\partial f}{\partial x_1}\right|_{\underline{x}} = \frac{f\left(\underline{x}+\begin{bmatrix}\varepsilon\\0\\0\end{bmatrix}\right) - f(\underline{x})}{\varepsilon} \qquad \varepsilon = 1e\text{-}6$$

$$\frac{\partial f}{\partial x_2} = \frac{f\left(\underline{x}+\begin{bmatrix}0\\\varepsilon\\0\end{bmatrix}\right) - f(\underline{x})}{\varepsilon}$$

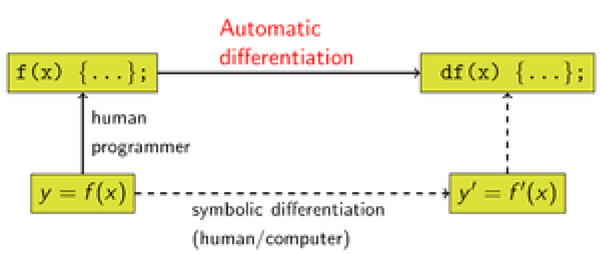
1. Numerical differentiation :

For n-dim derivative : 2n calls to function f

O(n)

2. Symbolic differentiation

3. Automatic differentiation



3.A Forward mode

Example:

$$z = f(x_1, x_2) = x_1 x_2 + \sin(x_1)$$

3.B Reverse mode

② Symbolic diff. [Automatic diff] (close)

Tensorflow
Pytorch
Jax

$x = \text{Symbol}()$

$f = x**2 + 2*x$

grad(f, x) → print a functional form of derivative

a) ~~sin~~'(~~sin~~(x)) more optimizations possible in symbolic derivatives

Diff wrt Auto diff

b) Derivative computation is done for all $\underline{x}$ input

c) f(x) has to be computed as a separate process

Auto. Diff

① Create a library of atomic function

atomic function		derivative function
$(f(x) + g(x))$	$\longleftrightarrow$	$\frac{\partial}{\partial x} f(x) + \frac{\partial}{\partial x} g(x)$
*		
**		
sin	$\longleftrightarrow$	cos
cos	$\longleftrightarrow$	−sin
exp	$\longleftrightarrow$	exp

② Write the function whose derivative you want in terms of the atomic functions

(3) Then derivative of any program written in Terms of atomic functions, can be computed by chain rule.

How?

$$f(\underline{x}) = \exp(-\underline{x}**2)$$

$$f(x) = \underbrace{\exp}\left(\underbrace{\_\_mul\_\_}_{y}\left(\underbrace{\_\_pow\_\_(x,2)}_{\text{deriv}}, -1\right)\right)$$

$$f(x) = \frac{\partial \exp(\underline{z})}{\partial \underline{z}} \cdot \frac{\partial}{\partial \underline{y}} \_\_mul\_\_(y) \cdot \frac{\partial}{\partial \underline{x}} \_\_pow\_\_(x,2)$$

Auto Diff
- Forward diff
- Reverse mode Backward diff (Backpropagation)

[computation complexity of diff.]

Example:

$x_1 = \text{ForwardDiff}(1, 1)$

$x_2 = \text{ForwardDiff}(2, 0)$

$x_1 + x_2$

$$z = f(x_1, x_2) = \underbrace{x_1 x_2}_{\text{ForwardDiff}(2,\, 2)} + \underbrace{\sin(x_1)}_{\text{ForwardDiff}(\sin(1),\, \cos(1))}$$

In [ ]:
```
import numpy as np
class ForwardDiff:
    def __init__(self, value, grad=None):
        self.value = value
        self.grad = np.zeros_like(value) if grad is None else grad


    def __add__(self, other):
        cls = type(self)
        other = other if isinstance(other, cls) else cls(other)
        out = cls(self.value + other.value,
                  self.grad + other.grad)
        return out
    __radd__ = __add__

    def __repr__(self):
        return f"{self.__class__.__name__}(data={self.value}, grad={self.gra
x = ForwardDiff(2, 1)
y = ForwardDiff(3, 0)

f = x + y
f
```

ForwardDiff

same fn

printing this class

$f(a, b) = a + b$

$\frac{d}{dy} f(a, b) = \frac{da}{dy} + \frac{db}{dy}$

$f(x, y) = x + y$

$\frac{d}{dx} f(x, y) = \underbrace{\boxed{\frac{dx}{dx}}}_{=1} + \boxed{\frac{dy}{dx}} = 0$

$f.grad = \frac{df}{dx}$

In [ ]:
```
oldFD = ForwardDiff # Bad practice: do not do it
class ForwardDiff(oldFD):
    def __mul__(self, other):
        cls = type(self)
        other = other if isinstance(other, cls) else cls(other)
        out = cls(self.value * other.value,
                  other.value * self.grad +
                  self.value * other.grad)
        return out

    __rmul__ = __mul__

x = ForwardDiff(2, 0)
y = ForwardDiff(3, 1)

f1 = x * y
f2 = 2*x + 3*y + x*y
f1, f2
```

$f(x, y) = x * y$

$\frac{d}{dz} f(x, y) = x * \frac{dy}{dz} + y * \frac{dx}{dz}$

In [ ]:
```
oldFD = ForwardDiff # Bad practice: do not do it
class ForwardDiff(oldFD):
    def log(self):
        cls = type(self)
        return cls(np.log(self.value),
                   1/self.value * self.grad)
    def exp(self):
        cls = type(self)
```

$\frac{d}{dz} \log(x) = \frac{1}{x} \cdot \frac{dx}{dz}$

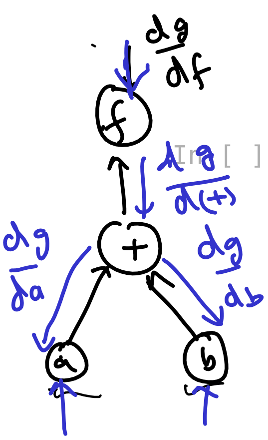
```python
        out_val = np.exp(self.value)
        return cls(out_val,
                   out_val * self.grad)
```

$$\frac{d}{dz}\exp(x) = \exp(x)\,\frac{dx}{dz}$$

```python
    def sin(self):
        cls = type(self)
        return cls(np.sin(self.value),
                   np.cos(self.value) * self.grad)

    def cos(self):
        cls = type(self)
        return cls(np.cos(self.value),
                  -np.sin(self.value) * self.grad)

    def __pow__(self, other):
        cls = type(self)
        other = other if isinstance(other, cls) else cls(other)
        return (self.log() * other).exp()

    def __neg__(self): # -self
        return self * -1

    def __sub__(self, other): # self - other
        return self + (-other)

    def __truediv__(self, other): # self / other
        return self * other**-1

    def __rtruediv__(self, other): # other / self
        return other * self**-1


x = ForwardDiff(2, 1)
y = ForwardDiff(3, 0)

f = x**y
f
```

$\frac{dx}{dz} = 1$

$\frac{dy}{dz} = 0$



vjp = vector - jacobian - product

```python
In [ ]: import numpy as np
        def add_vjp(a, b, grad):
            return grad, grad


        def no_parents_vjp(grad):
            return (grad,)

        class ReverseDiff:
            def __init__(self, value, parents=(), op='', vjp=no_parents_vjp):
                self.value = value
                self.parents = parents
                self.op = op
                self.vjp = vjp
                self.grad = None

            def backward(self, grad):
```

$\frac{dg}{df}$

$f(a,b) = a+b$

given $\frac{dg}{df}$

$\frac{dg}{db}$

$$\frac{dg}{da} = \boxed{\frac{dg}{df}\cdot\frac{df}{da}} = \frac{dg}{df}$$

$g(f)$

$f(a,b) = a+b$

arguments to the operation

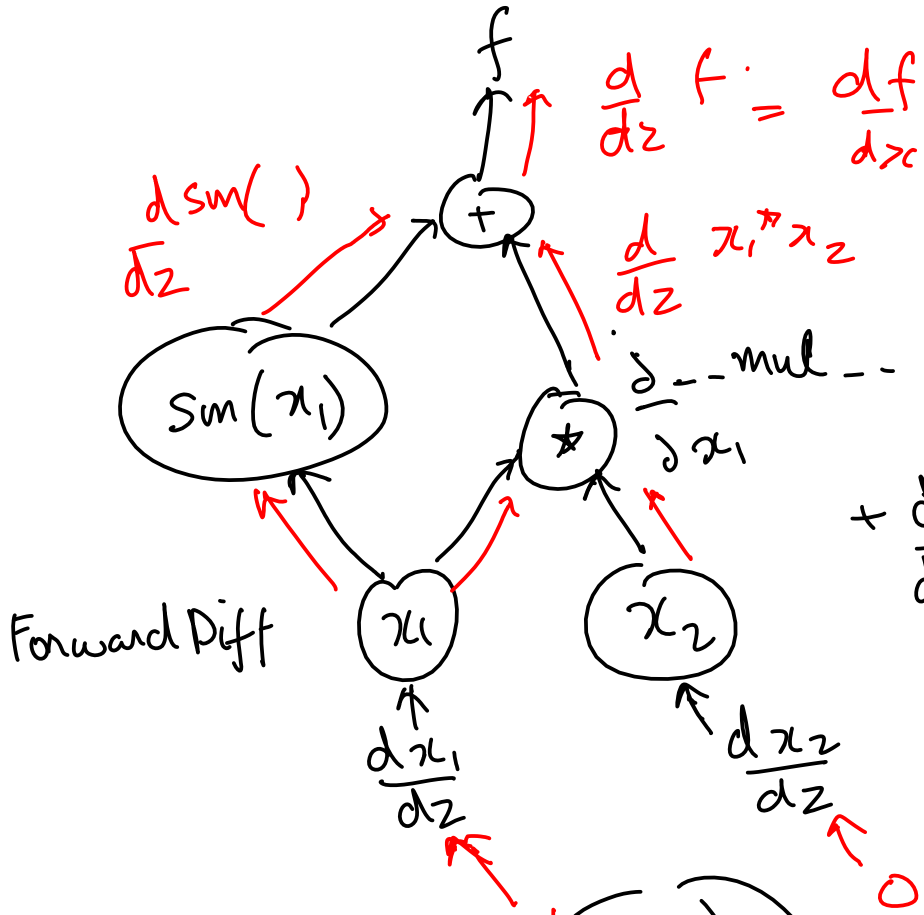
= operation

= deriv

$$\frac{dg}{db} = \frac{dg}{df}\cdot\frac{df}{db} = \frac{dg}{df}$$

Forward Diff

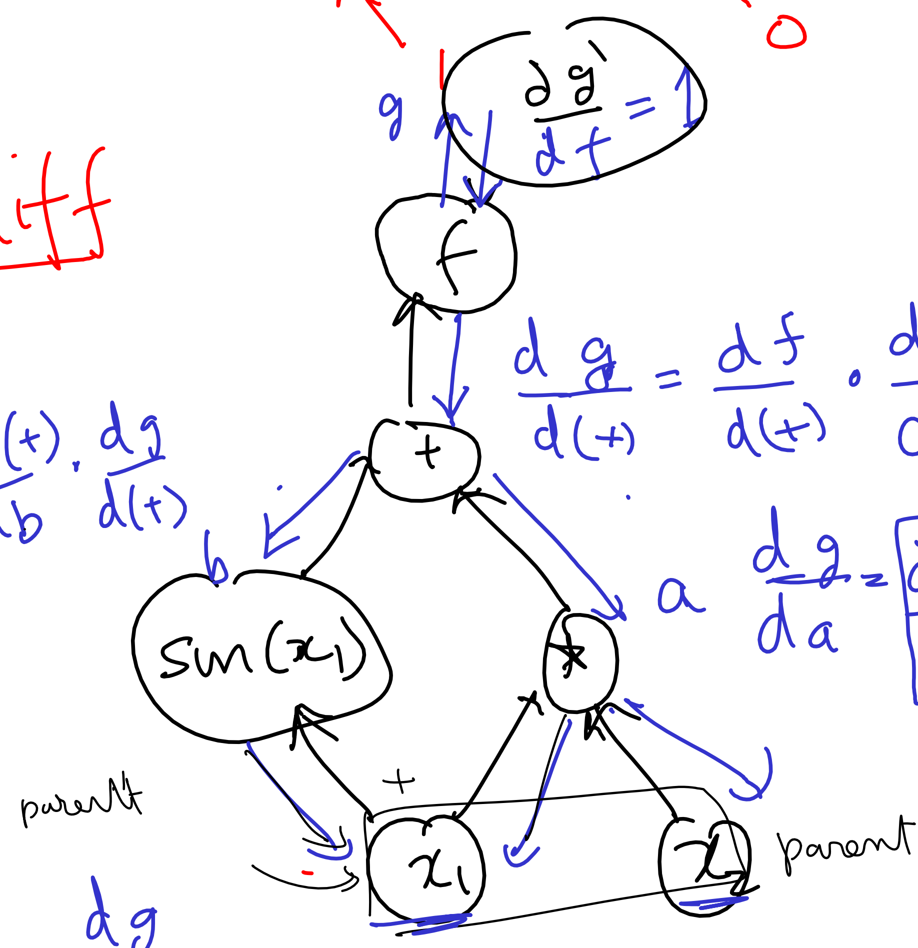
$$f(x) = x_1 * x_2 + \sin(x_1)$$



$$\frac{d}{dz} f = \frac{df}{dx_1}$$

$$\frac{\partial}{\partial x_1} \text{mul} (x_1, x_2) \cdot \frac{dx_1}{dz} + \frac{d}{dx_2} m(x_1, x_2) \frac{dx_2}{dz}$$

Reverse diff

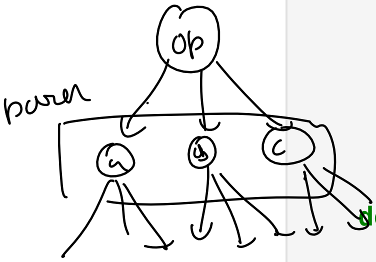


$$\frac{dg}{d(+)} = \frac{df}{d(+)} \cdot \frac{dg}{df}$$

$$\frac{dg}{db} = \frac{d(+)}{db} \cdot \frac{dg}{d(+)}$$

$$\frac{dg}{da} = \boxed{\frac{d(+)}{da}} \cdot \frac{dg}{d(+)}$$

$$\frac{dg}{dx_1} = \ldots$$




```
        self.grad = grad
        op_args = [p.value for p in self.parents]
        grads = self.vjp(*op_args, grad)
        for g, p in zip(grads, self.parents):
            p.backward(g)

    def __add__(self, other):
        cls = type(self)
        other = other if isinstance(other, cls) else cls(other)
        out = cls(self.value + other.value,
                  parents=(self, other),
                  op='+',
                  vjp=add_vjp)
        return out

    __radd__ = __add__

    def __repr__(self):
        cls = type(self)
        return f"{cls.__name__}(value={self.value}, parents={self.parents},

x = ReverseDiff(2)
y = ReverseDiff(3)

f = x + y + 3
f.backward(1)
f
x.grad, y.grad
```

+ self.grad

if self.grad is not None: grad

dg/df

x1 = ReverseDiff(2)

y = Rev

dg/df = 1

In [ ]:

```
oldRD = ReverseDiff # Bad practice: do not do it

def mul_vjp(a, b, grad):
    return grad * b, grad * a

class ReverseDiff(oldRD):
    def __mul__(self, other):
        cls = type(self)
        other = other if isinstance(other, cls) else cls(other)
        out = cls(self.value * other.value,
                 parents=(self, other),
                 op='*',
                 vjp=mul_vjp)
        return out

    __rmul__ = __mul__

x = ReverseDiff(2)
y = ReverseDiff(3)

f1 = 5*x + 7* y
f1.backward(1)
x.grad, y.grad
```

g = f1

dg/df = 1

In [ ]:

```
f2 = x*y
f2.backward(1)
```

```
x.grad, y.grad
```

$$f(g(h(x)))$$

$$\frac{\partial}{\partial x} f(g(h(x))) = \frac{\partial f}{\partial g} \cdot \frac{\partial g}{\partial h} \cdot \frac{\partial h}{\partial x}$$

Forward diff

$$\frac{\partial}{\partial x} f(g(h(x))) = \underbrace{\left(\frac{\partial f}{\partial g} \left(\frac{\partial g}{\partial h} \left(\frac{\partial h}{\partial x} \cdot \frac{dx}{dz}\right)\right)\right)}_{\text{Forward diff}}$$

$$\boxed{\frac{dx}{dz} = 1}$$

$\boxed{sum = 0}$

for i in range(10):

sum += i

Reverse mod diff

$$\frac{\partial}{\partial x} f(g(h(x))) = \underbrace{\left(\left(\left(\underbrace{\frac{\partial g}{\partial f}}_{=1} \frac{\partial f}{\partial g}\right) \frac{\partial g}{\partial h}\right) \frac{\partial h}{\partial x}\right)}_{\text{Reverse diff}}$$

Computational complexity of Forward Diff

$\propto$ number of inputs

Computational complexity of Reverse mode diffl

$\propto$ number of outputs