Automatic differentiation/backpropagation

- loss.backward() $\longrightarrow$ param.grad $\leftarrow$ what grad? why? How?

Computational Differentiation

① Numerical Differentiation

very good test case to verify the correctness of your implementations

$f(\underline{x})$ $\quad \frac{\partial f(\underline{x})}{\partial \underline{x}} = \left[\frac{\partial f(\underline{x})}{\partial x_1}, \ldots \frac{\partial f(\underline{x})}{\partial x_n} \right]$

Disadvantages:

a) Approximate

$$\frac{\partial f(\underline{x})}{\partial x_i} = \frac{f(\underline{x} + \underline{e}_i \varepsilon) - f(\underline{x})}{\varepsilon}$$

ε: small no $= 10^{-4}, 10^{-6}$

$\underline{e}_i = \begin{bmatrix} 0 \\ \vdots \\ 1 \\ \vdots \\ 0 \end{bmatrix}$ — 1 at i^{th} place, 0 everywhere

b) Computationally expensive:

How many times do you have to call $f(\underline{x})$ to compute $\frac{\partial f(\underline{x})}{\partial \underline{x}}$ if $\underline{x} \in \mathbb{R}^n$?

$n+1$ times

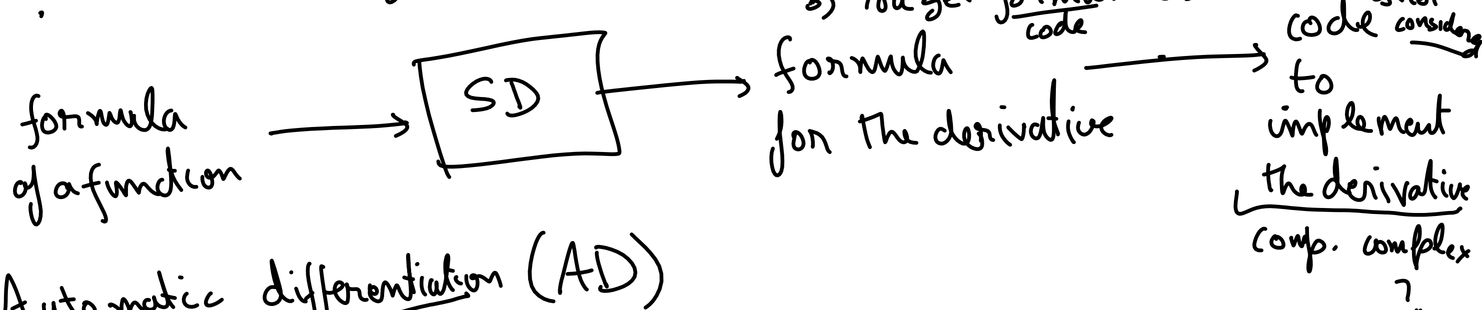
② Symbolic Differentiation (SD)

e.g. sympy / Matlab

Disadvantages

a) you need a new language to communicate your formulas

b) You get formula as output. Efficiency is not considered



③ Automatic differentiation (AD)

$\rightarrow$ Operator overloading to implement (AD)

C++ / Python

a + b (int, int)

a = Person( )
b = Person( )

a + b $\quad$ Person.__add__

a@b $\quad$ Person.__matmul__

a = Tensor( )
b = Tensor( )

```
class Tensor( )
    def __init__(   )
        self.value =
        self.grad =

a = Tensor(int)
b = Tensor(int)
```

```
c = a + b
c.value = a.value + b.value
c.grad = ~~~~~~

a = Sympy.Variable("a")
b = "  "  ("b")

c = a + b

c.expression = "a+b"
```

AD

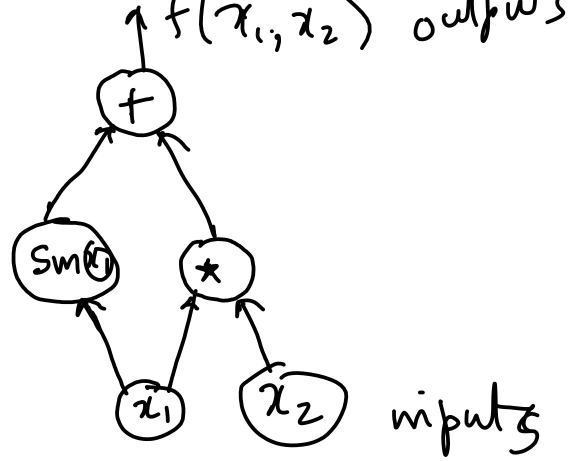
1. Forward-mode AD (Forward accumulative)

2. Reverse-mode AD (Backpropagation in NN)

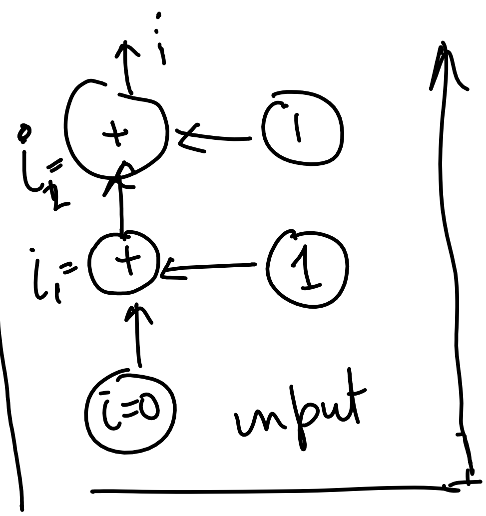
Mathematical expressions can be represented as a <u>Directed Acyclic graph</u> (computation graph)

Example

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



```
i = Tensor(0)
for i in range(10)
    i = i + 1
```

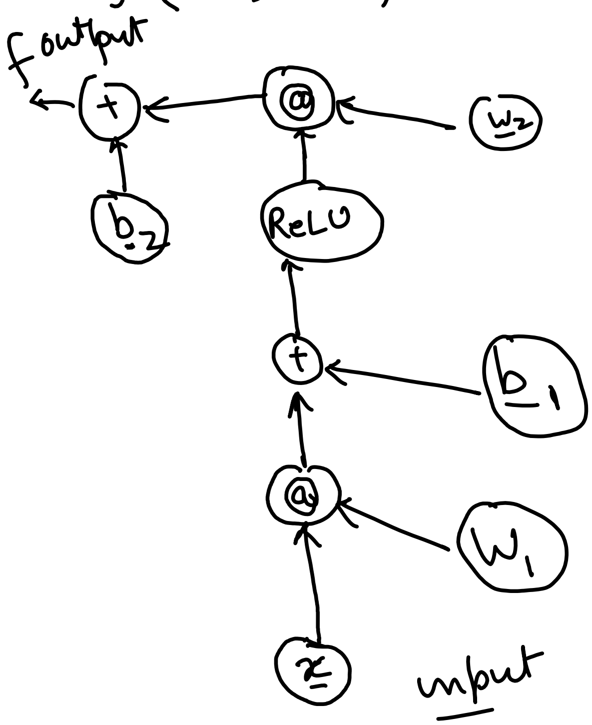


Internally Pytorch will keep a track of this graph

Computation Graph of MLP (2-layer, ReLU)

$$f(\underline{x}; \underline{W}_1, \underline{w}_2, \underline{b}_1, \underline{b}_2) = \underline{w}_2^T \, \mathrm{ReLU}(W_1 \underline{x} + \underline{b}_1) + b_2$$

$$= \underline{w}_2^T (W_1 \underline{x} + b_1)_{\underset{\text{ReLU}}{+}} + \underline{b}_2$$



(1) Forward-mode AD

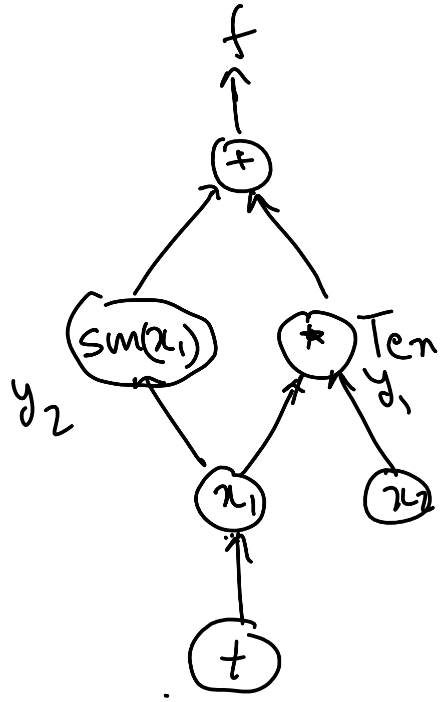
Use operator overloading to implement AD

```
class Tensor( )
    def __init__(  )
        self.value =
        self.grad =

a = Tensor(int)
b = Tensor(int)

c = a + b
```

```
class Tensor
    def __add__(s, b)
        c = Tensor()
        c.value = self.value + b.value
        c.grad = self.grad + b.grad
```



Tensor → value, grad $= \boxed{\dfrac{\partial f}{\partial y_1}} = ?$ Reverse mod

$\boxed{\dfrac{\partial y_1}{\partial x_1}} = ?$ Forward mode

In forward mode $\quad x_1.\text{grad} = \dfrac{\partial x_1}{\partial t} = 1$

$x_2.\text{grad} = \dfrac{\partial x_2}{\partial t} = 0$

$y_1 = x_1 * x_2 \Rightarrow y_1.\text{grad} = \dfrac{\partial}{\partial t}(x_1 * x_2) = x_1 \dfrac{\partial x_2}{\partial t} + x_2 \dfrac{\partial x_1}{\partial t}$

$$\Rightarrow y_1.grad = x_1 * (x_2.grad) + x_2 * (x_1.grad)$$

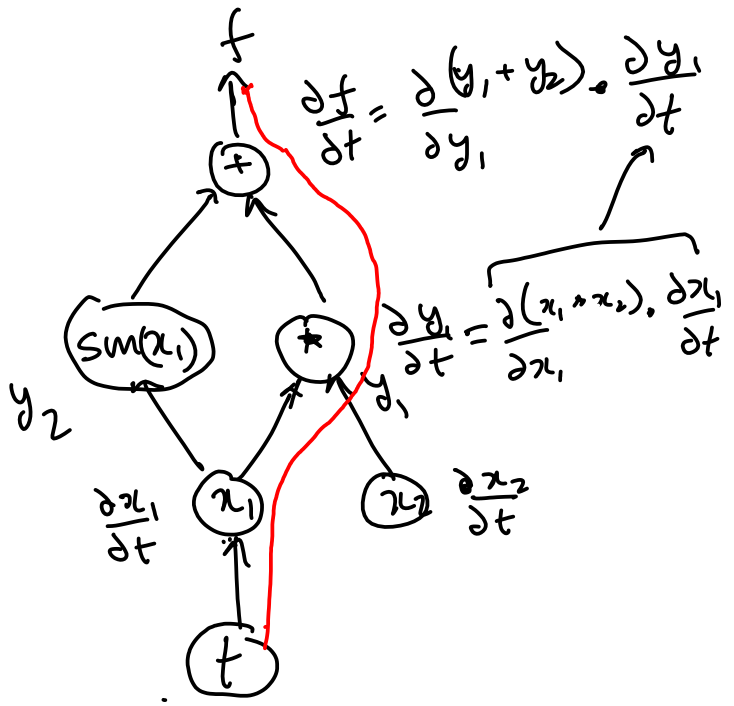
$$y_2 = \sin(x_1) \quad \rightarrow \frac{\partial y_2}{\partial t} = y_2.grad = \cos(x_1) \frac{\partial x}{\partial t} = \cos(x_1)(x_1.grad)$$

$$f = y_1 + y_2 \quad \Rightarrow \frac{\partial f}{\partial t} = f.grad = \frac{\partial y_1}{\partial t} + \frac{\partial y_2}{\partial t}$$

$$= y_1.grad + y_2.grad$$

Accumulation direction

$$\frac{\partial f}{\partial t} = \left[\frac{\partial}{\partial y_1}(y_1 + y_2) . \underbrace{\left[\frac{\partial (x_1 * x_2)}{\partial x_1} . \frac{\partial x_1}{\partial t}\right]}_{\text{This accumulation happens first}}\right]$$



$$f = h(g(k(l(x))))$$

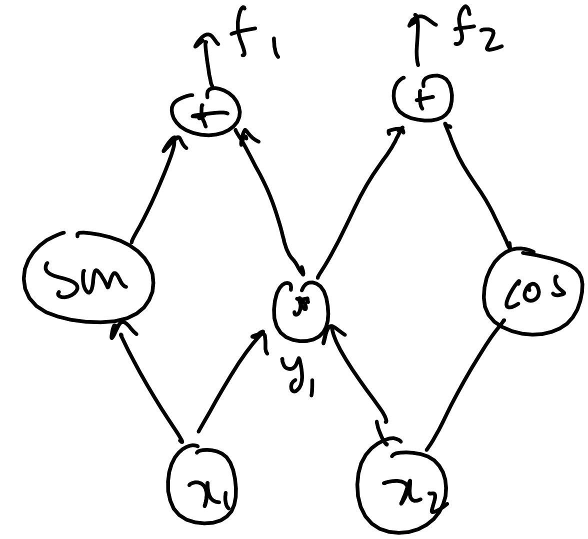
$$\frac{\partial f}{\partial x} = \left(\frac{\partial h}{\partial g}\left(\frac{\partial g}{\partial k}\left(\frac{\partial k}{\partial l} \frac{\partial l}{\partial x}\right)\right)\right)$$

(1) In forward mode differentiation, the accumulation of chain rule derivatives happens from the input side to the output side (forward direction).

(2) In forward mode differentiation, you have to re-propagate the derivatives for each input.

$$f_1(x_1, x_2) = x_1 x_2 + \sin(x_1)$$

$$f_2(x_1, x_2) = \underbrace{x_1 \cdot x_2} + \cos(x_2)$$



$$\frac{\partial f_1}{\partial t}, \quad \frac{\partial f_2}{\partial t}$$

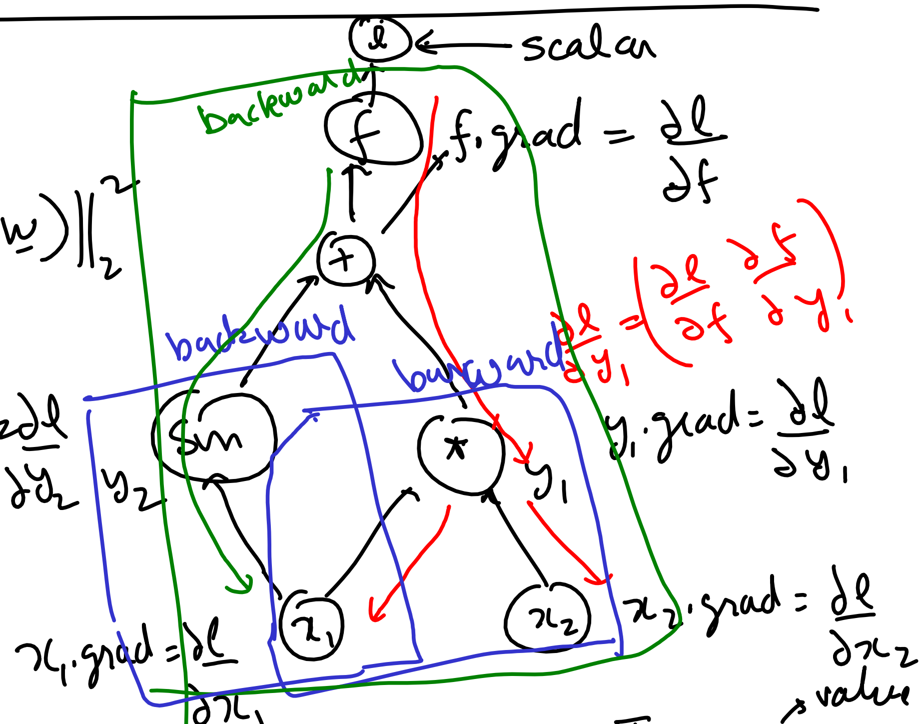
③ we dont have to recompute derivatives for the common part of the graph.

---

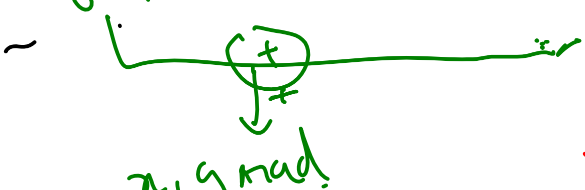
② Reverse - mode AD

$$L(D: \underline{w}) = \sum_{D} \|\underline{y} - \underline{f}(\underline{x}, \underline{w})\|_2^2$$

scalar



$$\frac{\partial l}{\partial x_1} = \frac{\partial l}{\partial f} \frac{\partial f}{\partial y_2} \frac{\partial f}{\partial x_1}$$

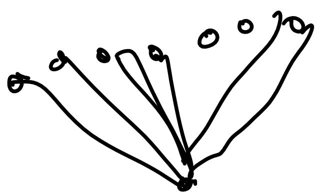


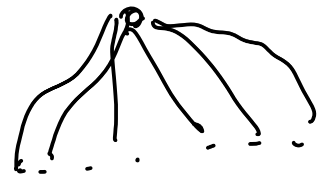
$$\frac{\partial l}{\partial x_1} = \frac{\partial l}{\partial y_1} \cdot \frac{\partial y_1}{\partial x_1}$$

$$\frac{\partial l}{\partial x_1} = \left(\left(\frac{\partial l}{\partial f} \frac{\partial f}{\partial y_1}\right) \cdot \frac{\partial y_1}{\partial x_1}\right)$$

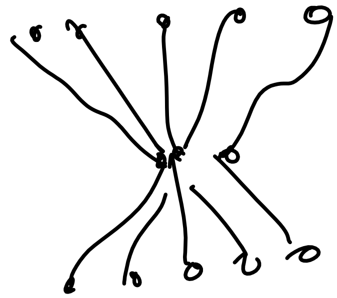
① Reverse mode is more efficient the forward mode when # of output < # of inputs

Forward mode AD





Reverse mode AD



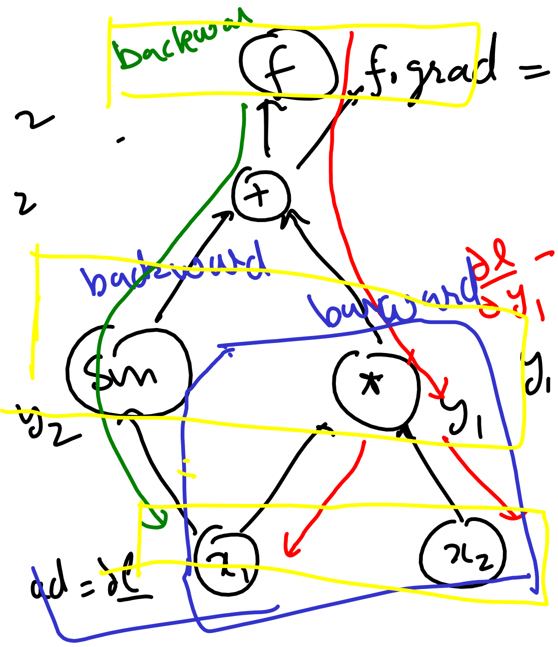
NP-hard

$f = f = \cdot$

chain graph

$\underline{h} = \begin{bmatrix} \sin(x_1) \\ x_1 * x_2 \end{bmatrix}$

$\underline{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$



$$\frac{\partial f}{\partial \underline{x}} = \underbrace{\frac{\partial f}{\partial \underline{h}}}_{\text{vector}} \underbrace{\frac{\partial \underline{h}}{\partial \underline{x}}}_{\substack{\text{Matrix} \\ \text{Jacobian}}}$$

$$\underline{f} = \underline{f}(\underline{g}(\underline{h}(\underline{k}(\underline{x}))))$$

$$\frac{\partial f}{\partial \underline{x}} = \frac{\partial f}{\partial \underline{g}} \cdot \frac{\partial \underline{g}}{\partial \underline{h}} \cdot \frac{\partial \underline{h}}{\partial \underline{k}} \cdot \frac{\partial \underline{k}}{\partial \underline{x}}$$

FD $\qquad\qquad$ RD

$$\frac{\partial f}{\partial \underline{x}} = \left(\frac{\partial f}{\partial \underline{g}}\left(\frac{\partial \underline{g}}{\partial \underline{h}}\left(\frac{\partial \underline{h}}{\partial \underline{k}} \cdot \frac{\partial \underline{k}}{\partial \underline{x}}\right)\right)\right)$$

$$\frac{\partial f}{\partial x} = \left(\left(\left(\frac{\partial f}{\partial \underline{g}} \cdot \frac{\partial g}{\partial \underline{h}}\right) \cdot \frac{\partial h}{\partial k}\right) \frac{\partial \underline{k}}{\partial \underline{x}}\right)$$

O(matrix size)

\# output < \# inputs

$$O_{FD}(\ ) > O_{RD}(\ )$$

$$\underline{k}(\underline{x}) : \overset{\underline{x}}{\mathbb{R}^n} \to \overset{\underline{k}(\underline{x})}{\mathbb{R}^m}$$

$$\frac{\partial \underline{k}}{\partial \underline{x}} \in \mathbb{R}^{? \times ?}$$

What is the dim of the Jacobian matrix?

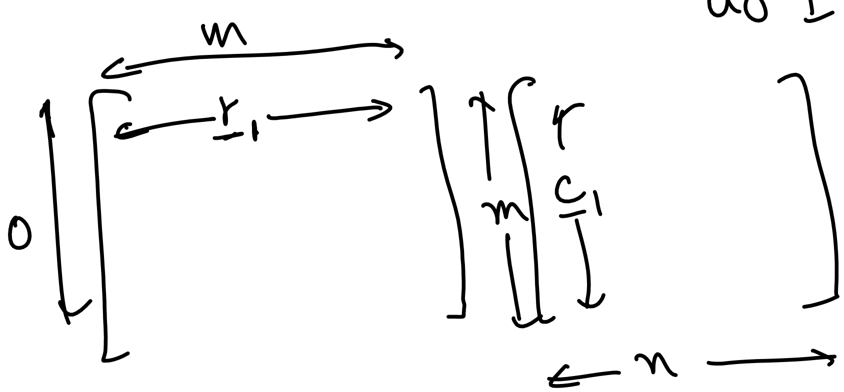
$$\frac{\partial \underline{k}}{\partial \underline{x}} = \begin{bmatrix} \frac{\partial k_1}{\partial x_1}, & \frac{\partial k_1}{\partial x_2} & \cdots & , \frac{\partial k_1}{\partial x_n} \\ \vdots & & & \\ \frac{\partial k_m}{\partial x_1}, & \cdots & & , \frac{\partial k_m}{\partial x_n} \end{bmatrix}_{m \times n}$$

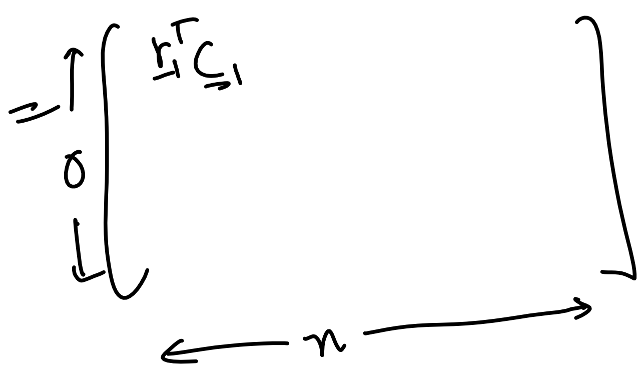
$$\frac{\partial f}{\partial \underline{x}} = \frac{\partial \underline{f}^{\,q}}{\partial \underline{g}} \cdot \frac{\partial \underline{g}^{\,p}}{\partial \underline{h}} \cdot \left(\frac{\partial \underline{h}^{\,o}}{\partial \underline{k}} \cdot \frac{\partial \underline{k}^{\,m}}{\partial \underline{x}_{\,n}} \right) \qquad f = \underline{f}(\underline{g}(\underline{h}(\underline{k}(\underline{x}))))$$

$$\frac{\partial \underline{h}}{\partial \underline{k}} \in \mathbb{R}^{o \times m} \qquad \frac{\partial \underline{k}}{\partial \underline{x}} \in \mathbb{R}^{m \times n}$$

$$\underset{o \times m}{\frac{\partial \underline{h}}{\partial \underline{k}}} \ \underset{m \times n}{\frac{\partial \underline{k}}{\partial \underline{x}}}$$

How many addition and multiplication operations do I need for this matmul?





How many mul? = m
$+$
add? = m

Total number of operations?

$= 2mno$

$$\boxed{\text{Comp. complexity of mat mul} = O(mno)}$$

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

q, p, o, m, n

Reverse mode - AD

Comp. complexity of RD ?

$q \times p \quad p \times o \quad O(qpo)$

$q \times o \quad o \times m \quad O(qom)$

$q \times m \quad m \times n \quad O(qmn)$

$\cancel{q p^2 o^2 m^2 n^2}$ ×

$O(\boxed{q}(po + om + mn))$ ✓

output dim = 1 for scalar Loss

Forward mode - AD : Comp. complexity?

$O(qpn + pon + omn)$ ✓

$O((qp + po + om)\boxed{n})$

input dim

When output dim < Input dim

use Reverse mode AD ✓

Otherwise

use Forward mode AD

$$\underline{f}(\underline{a}, \underline{b}) = \underline{a} + \underline{b} \in \mathbb{R}^n$$

we are given $\dfrac{\partial l}{\partial \underline{f}} = \left[\dfrac{\partial l}{\partial f_1}, \ldots \ldots \dfrac{\partial l}{\partial f_n}\right]$

Find $\dfrac{\partial l}{\partial \underline{a}}, \dfrac{\partial l}{\partial \underline{b}}$?

$$\underbrace{l(\underline{g}(\underline{h}(\underline{f}(\underline{x}))}$$

$$\underbrace{\frac{\partial l}{\partial \underline{f}} \cdot \frac{\partial \underline{f}}{\partial}}_{\frac{\partial l}{\partial \underline{x}}}$$

$$\frac{\partial l}{\partial \underline{a}} = \underbrace{\frac{\partial l}{\partial \underline{f}}} \cdot \underbrace{\frac{\partial \underline{f}}{\partial \underline{a}}}$$

$$\frac{\partial \underline{f}}{\partial \underline{a}} = \frac{\partial}{\partial \underline{a}}(\underline{a} + \underline{b}) = I_{n \times n}$$

$$\frac{\partial \underline{f}}{\partial \underline{b}} = \frac{\partial}{\partial \underline{b}}(\underline{a} + \underline{b}) = I_{n \times n}$$

$$\frac{\partial}{\partial \underline{a}}(\underline{a} + \underline{b}) = \frac{\partial \underline{a}}{\partial \underline{a}} + \frac{\partial \underline{b}}{\partial \underline{a}} \to 0$$

$$\frac{\partial \underline{a}}{\partial \underline{a}} = \begin{bmatrix} \frac{\partial a_1}{\partial a_1} \to 1 & \frac{\partial a_1}{\partial a_2} \to 0 & \cdots & \frac{\partial a_1}{\partial a_n} \\ \frac{\partial a_2}{\partial a_1} & \frac{\partial a_2}{\partial a_2} \to 1 & & \\ \vdots & \vdots & & \\ \frac{\partial a_n}{\partial a_1} & \frac{\partial a_n}{\partial a_2} & \cdots & \frac{\partial a_n}{\partial a_n} \end{bmatrix} = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & & 0 \\ \vdots & & & \vdots \\ 0 & 0 & \cdots & 1 \end{bmatrix}$$

$$= I_{n \times n}$$

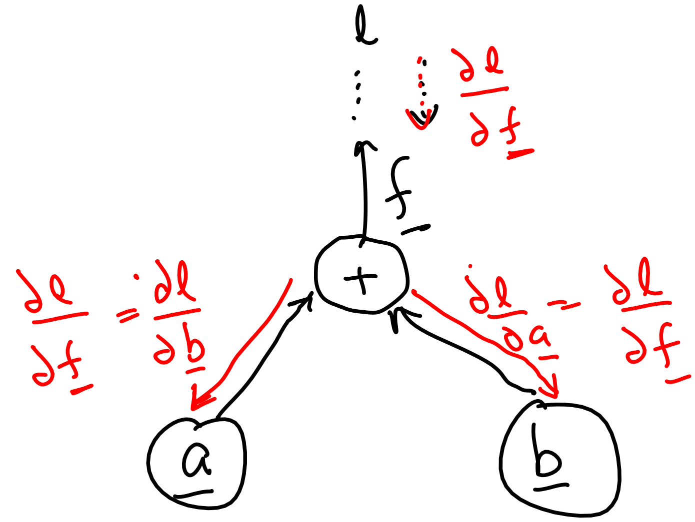
① $$\frac{\partial}{\partial \underline{x}} \underline{x}^T A \underline{x} = x^T (A + A^T)$$

② $$\frac{\partial}{\partial \underline{x}} \underline{b}^T \underline{x} = b^T$$

③ $$\frac{\partial}{\partial \underline{x}} A \underline{x} = A \Rightarrow \frac{\partial \underline{x}}{\partial \underline{x}} = \frac{\partial}{\partial \underline{x}} I \underline{x} = I$$

$$\frac{\partial l}{\partial \underline{a}} = \frac{\partial l}{\partial \underline{f}} \frac{\partial \underline{f}}{\partial \underline{a}} = \frac{\partial l}{\partial \underline{f}} \cdot I_{n \times n} = \frac{\partial l}{\partial \underline{f}}$$

$$\frac{\partial l}{\partial \underline{b}} = \frac{\partial l}{\partial \underline{f}}$$



Multiplication(s)

① Scalar-vector multiplication

$\underline{f}(\alpha, \underline{x}) = \alpha \underline{x}$, we are given $\frac{\partial l}{\partial \underline{f}}$

Find $\frac{\partial l}{\partial \alpha}$ and $\frac{\partial l}{\partial \underline{x}}$ (Answer in terms of α, $\underline{x}$ and $\frac{\partial l}{\partial \underline{f}}$)

$$\frac{\partial l}{\partial \alpha} = \frac{\partial l}{\partial \underline{f}} \cdot \frac{\partial \alpha \underline{x}}{\partial \alpha} = \frac{\partial l}{\partial f} \underline{x}$$

$$\frac{\partial l}{\partial \underline{x}} = \frac{\partial l}{\partial \underline{f}} \frac{\partial (\alpha \underline{x})}{\partial \underline{x}} = \frac{\partial l}{\partial \underline{f}} \left(\alpha I_{n \times n} \right) = \alpha \frac{\partial l}{\partial \underline{f}}$$

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

1. Numerical differentiation

2. Symbolic differentiation

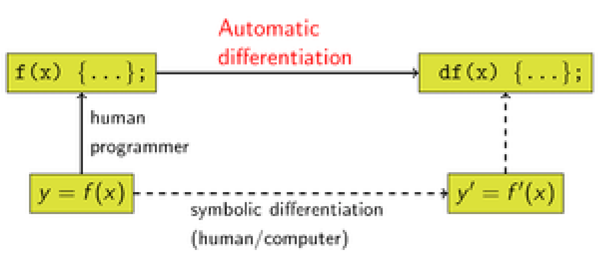
3. Automatic differentiation

   A. Forward mode differentiation
   B. Reverse mode differentiation

1. Numerical differentiation

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

Example:

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

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
```

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
```

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)
```

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

    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
```

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):
```

```python
        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
```

In [ ]:

```python
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
```

In [ ]:

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

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