Automatic differentiation/backpropagation

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

Computational Differentiation

① Numerical Differentiation

very good test cause to verify the correctness of your implementation

$f(\underline{x})$

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

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

ε: small no $\approx 10^{-4}, 10^{-6}$

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

Disadvantages:

a) Approximate

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
code

$$\begin{tikzcd} \text{formula of a function} \arrow[r] & \boxed{SD} \arrow[r] & \text{formula for the derivative} \arrow[r] & \text{code to implement the derivative} \end{tikzcd}$$

comp. complex ?

③ Automatic differentiation (AD)

→ Operator overloading to implement (AD)

C++/Python

$a + b$

$\uparrow$ int $\quad \uparrow$ 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

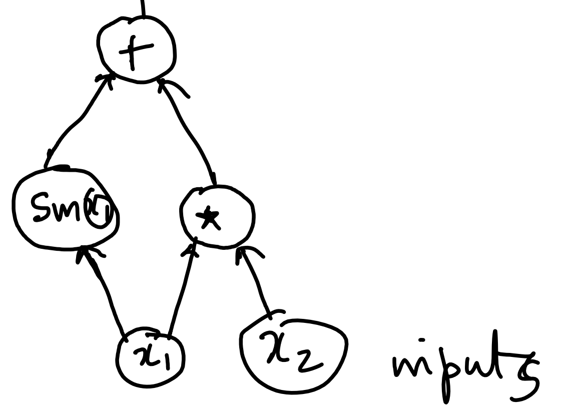
1. Forward-mode AD (Forward accumulative)

2. Reverse-mode AD (Backpropagation in NN)

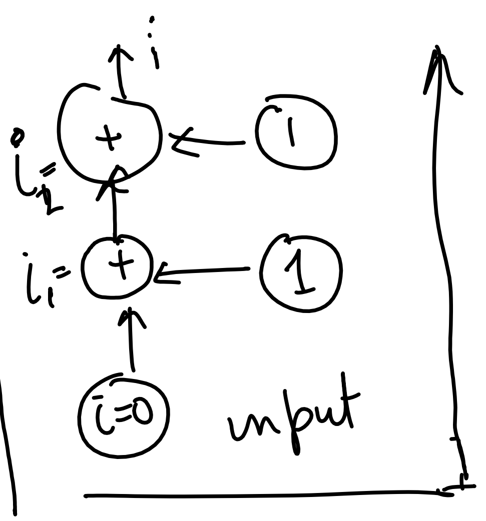
Mathematical expressions can be represented as a Directed Acyclic graph (computation graph)

Example

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



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



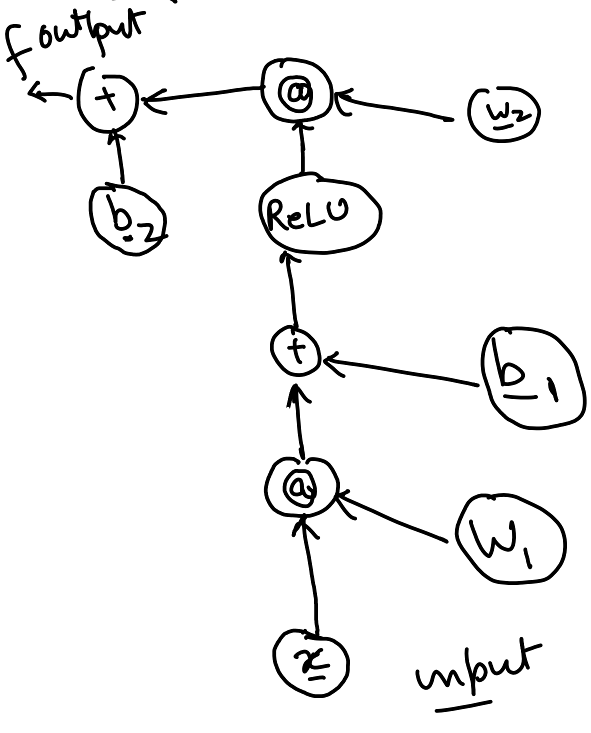
Internally Pytorch will keep a track of this graph

Computation Graph of MLP (2-layer, ReLU)

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

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

ReLU



(1) Forward-mode AD

Use operator overloading to implement AD

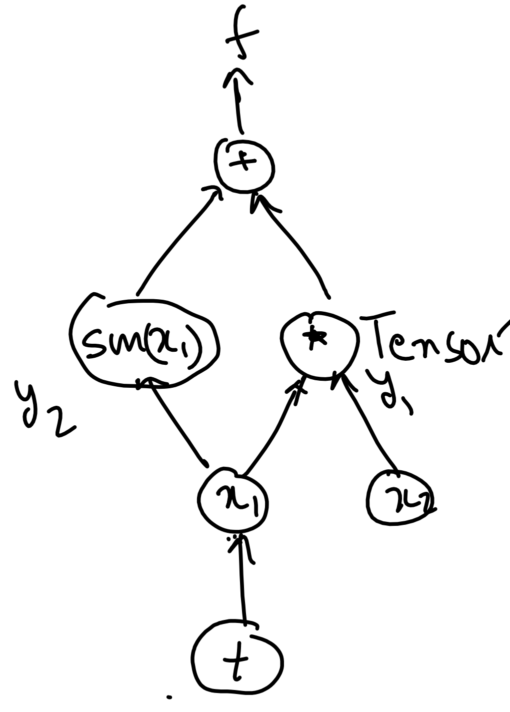
```
class Tensor( )
    def __init__(   )
        self.value =
        self.grad =

a = Tensor(init)
b = Tensor(init)

c = a + b
```

```
class Tensor

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



value

grad = $\boxed{\frac{\partial f}{\partial y_1}} = ? \cdot \boxed{\frac{\partial y_1}{\partial x_1}} = ?$

Reverse mod

Forward mode

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

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

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

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

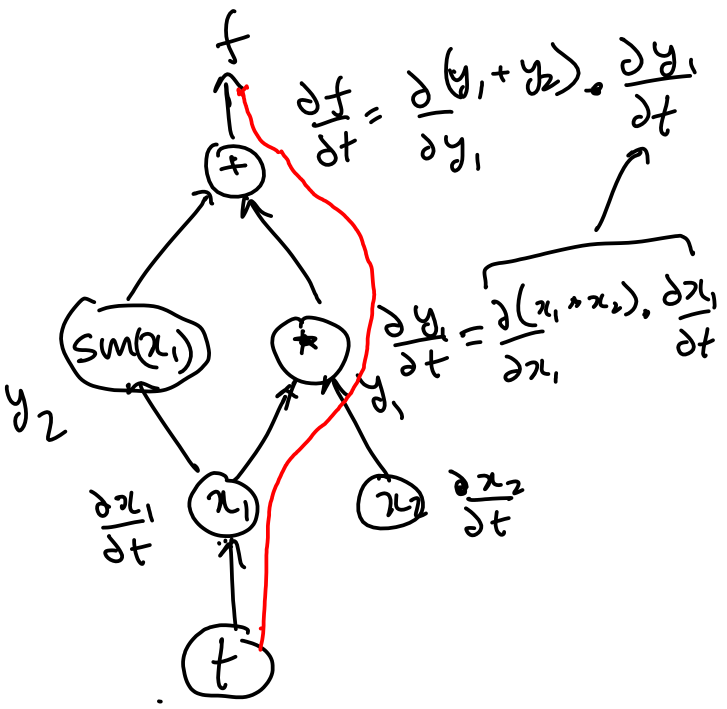
$$y_2 = \sin(x_1) \quad \rightarrow \frac{\partial y_2}{\partial t} = y_2.\text{grad} = \cos(x_1)\frac{\partial x}{\partial t} = \cos(x_1)(x_1.\text{grad})$$

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

$$= y_1.\text{grad} + y_2.\text{grad}$$

Accumulation direction

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



f
$\frac{\partial f}{\partial t} = \frac{\partial(y_1+y_2)}{\partial y_1} \cdot \frac{\partial y_1}{\partial t}$
$+$
$\sin(x_1)$
$*$
y_2
y_1
$\frac{\partial y_1}{\partial t} = \frac{\partial(x_1 * x_2)}{\partial x_1} \cdot \frac{\partial x_1}{\partial t}$
x_1
x_2
$\frac{\partial x_1}{\partial t}$
$\frac{\partial x_2}{\partial t}$
t

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

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

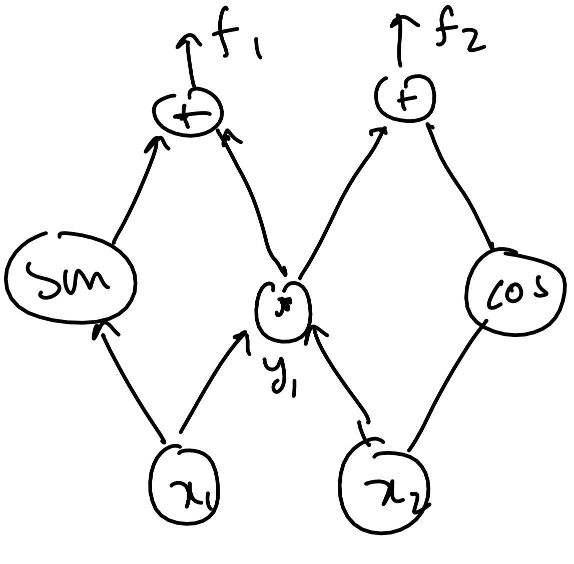
$$\longleftarrow \qquad \longleftarrow$$

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.

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

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

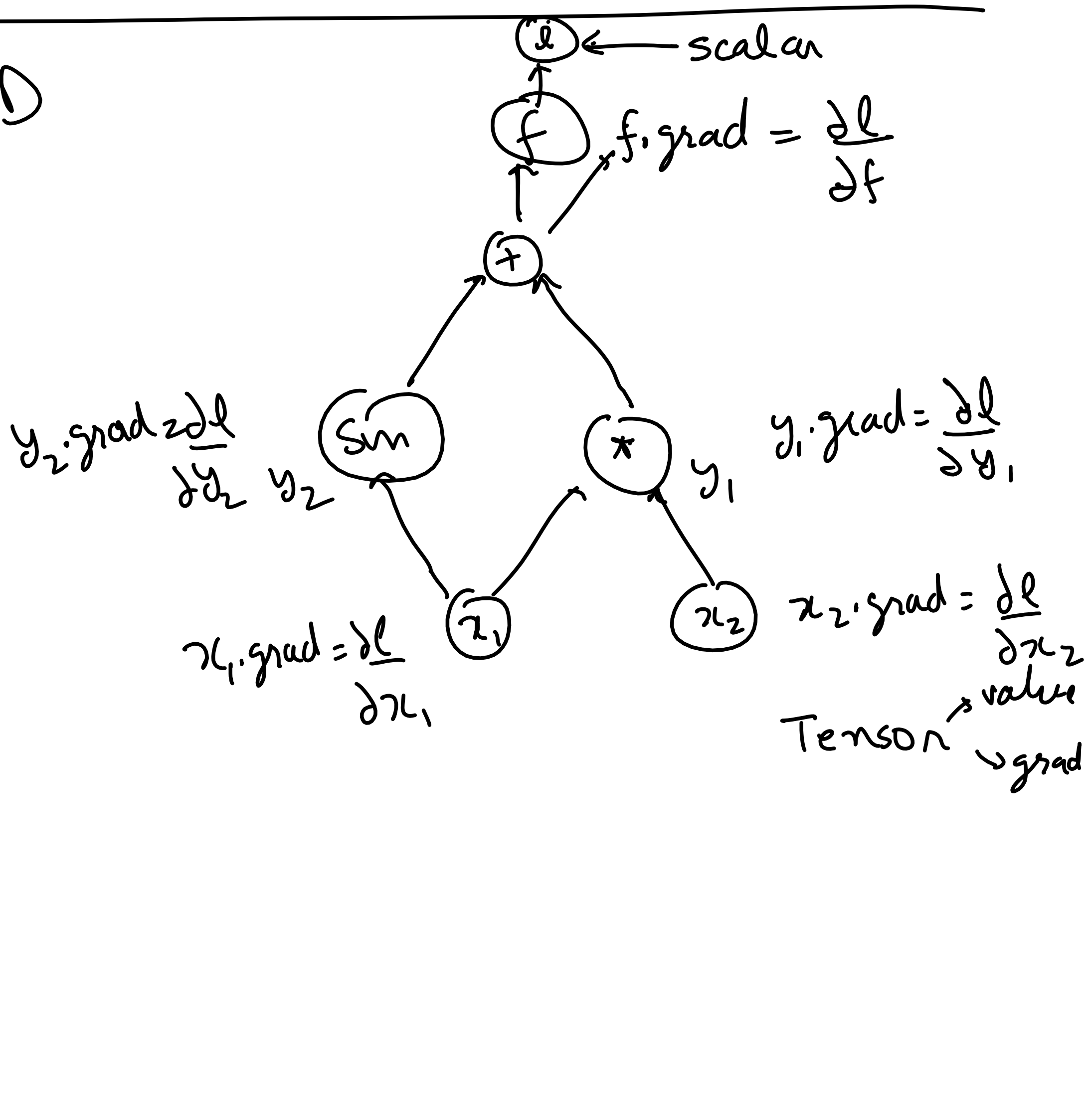


$$\frac{\partial f_1}{\partial t}, \quad \frac{\partial f_2}{\partial t}$$

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

---

② Reverse - mode AD



l
scalar
f
$f.\text{grad} = \frac{\partial l}{\partial f}$
+
sin
*
$y_2.\text{grad} = \frac{\partial l}{\partial y_2}$ y_2
y_1
$y_1.\text{grad} = \frac{\partial l}{\partial y_1}$
x_1
x_2
$x_1.\text{grad} = \frac{\partial l}{\partial x_1}$
$x_2.\text{grad} = \frac{\partial l}{\partial x_2}$
Tensor
value
grad

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 = ""
```

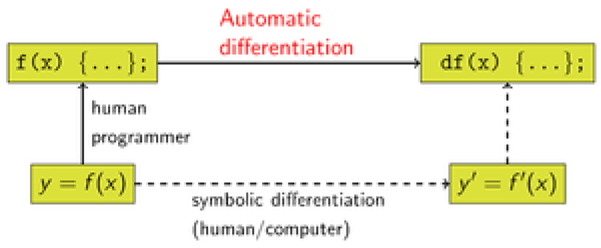
Differentiation options

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


Automatic differentiation
f(x) {...};
df(x) {...};
human programmer
y = f(x)
y' = f'(x)
symbolic differentiation (human/computer)

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)
```

```python
        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 [ ]:

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

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