

Computing Derivatives & Autograd Lab 4

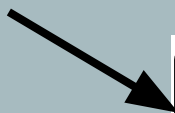
By John Liu & Michael Kireeff

Raise your hand if you dislike a lot of unnecessary derivatives!!!!!!

Raise your hand if you like points!

Raise your hand if you like
****bonus****👁👁👁 points?

We can combine all those interests!!!!



Start: Jan 17 at 11:59pm | Due: Apr 25 at 11:59pm

HW1 Autograd

Start: Jan 17 at 11:59pm | Due: Apr 25 at 11:59pm

Recap

Method 1: Hand-Coding Derivatives

- We use the definitions in calculus to calculate $\frac{\partial y}{\partial w_2}$, $\frac{\partial y}{\partial w_1}$, $\frac{\partial y}{\partial b_1}$, etc.
- Use the chain rule
- Pros:
 - Exact
 - Fast if well-implemented
- Cons:
 - Error-prone
 - Wasteful in terms of generalizability
 - Time consuming to hand calculate and code.

Method 2: Numerical Differentiation

- Approximate derivatives with using finite differences
- $$\frac{\partial y}{\partial w} \approx \frac{y(w+\epsilon) - y(w-\epsilon)}{2\epsilon}$$

Consideration: epsilon value changes accuracy

- Pros:
 - Easier to code (sorta)
- Cons:
 - Computationally expensive for higher-dimensional problems
 - Can accumulate a lot of error.

Method 3: Auto Differentiation :)

- Start with input x
- Use Primitive Operations to incrementally create differentiation graph
- Automatically track computations
- Continuously Update parameters during gradient calculations
- Pytorch uses an autodifferentiation engine called autograd that you have used!
- Pros:
 - Backward is automatically if computation graph is correctly formed
 - Exact calculations
- Cons:
 - Must be very careful while implementing the forward functions.
 - Makes brains hurt

Let's start small

Take for example, $f = b * (a + b)$

Our goal is to figure out how a and b affect f

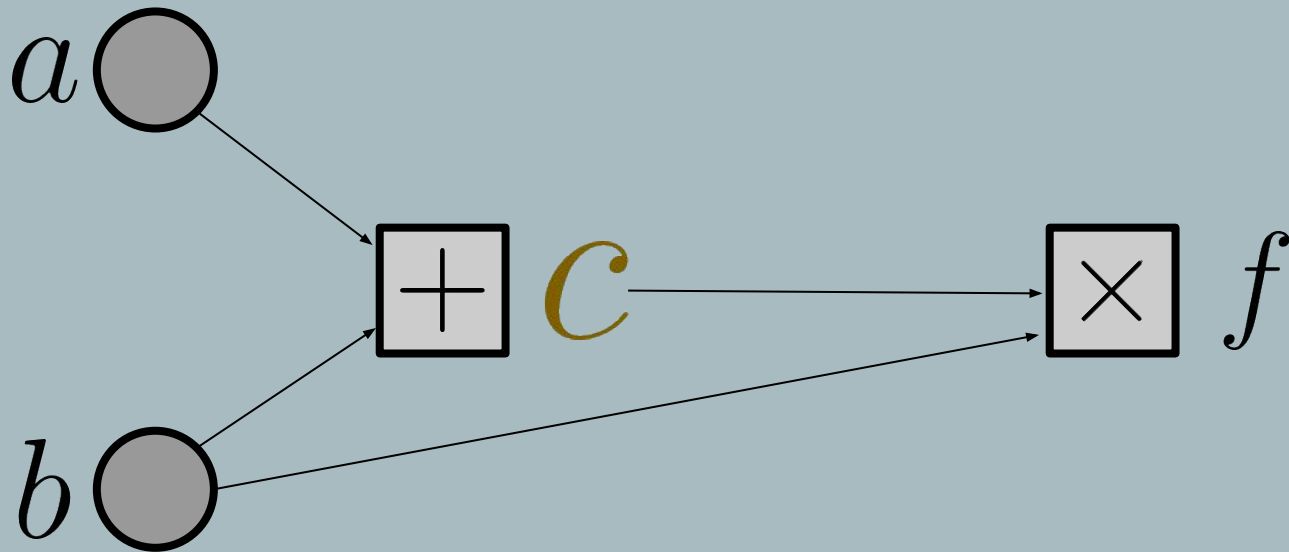
So, we're looking for $\frac{\partial f}{\partial a}$, and $\frac{\partial f}{\partial b}$

$$\frac{\partial f}{\partial b} = \frac{\partial(b * (a + b))}{\partial b} = b + (a + b)$$

$$\frac{\partial f}{\partial a} = \frac{\partial(b * (a + b))}{\partial a} = b$$

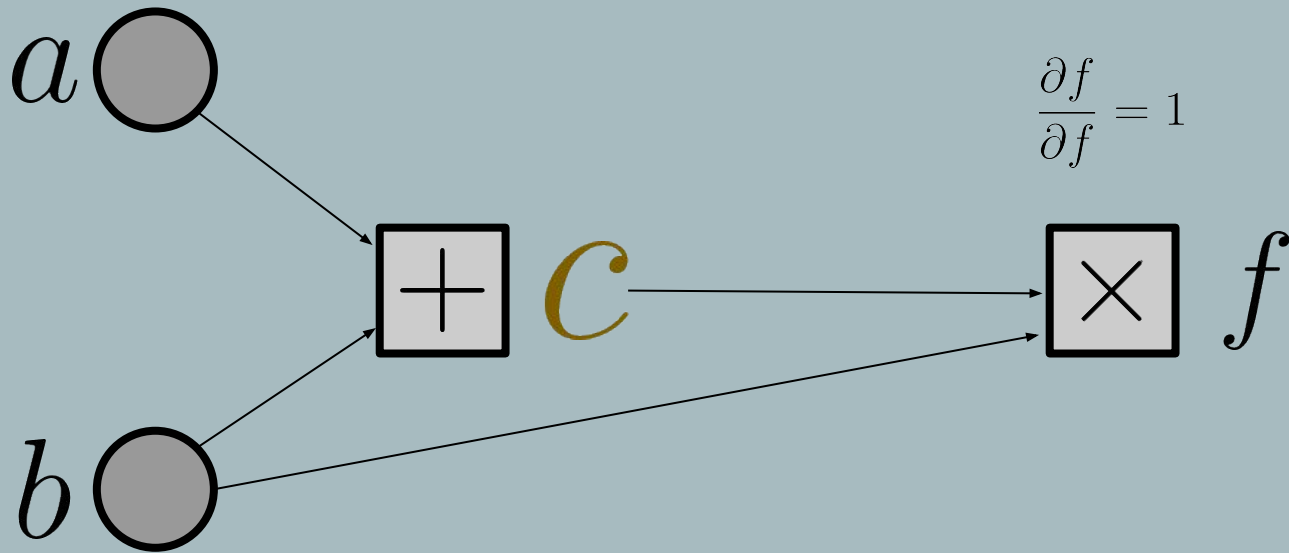
Let's make a graph
instead!

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



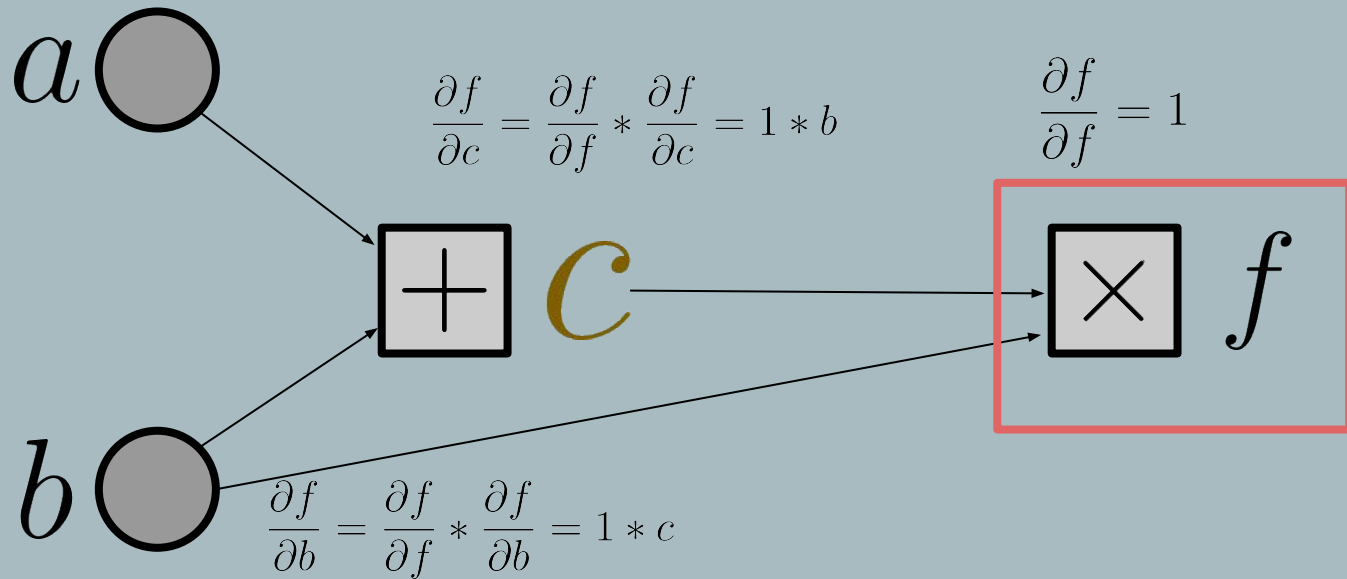
Let's analyze how
this works

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



Let's analyze how
this works

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



Let's analyze how this works

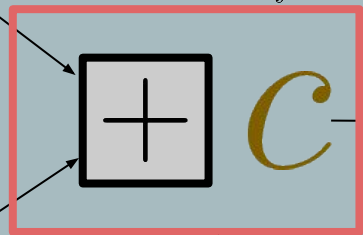
$$f = b * (a + b)$$

$$\frac{\partial f}{\partial a} = \frac{\partial f}{\partial f} * \frac{\partial f}{\partial c} \frac{\partial c}{\partial a} = 1 * b * 1 = b$$

a

$$\frac{\partial f}{\partial c} = \frac{\partial f}{\partial f} * \frac{\partial f}{\partial c} = 1 * b$$

$$\frac{\partial f}{\partial f} = 1$$



$\times$ f

b

$$\frac{\partial f}{\partial b} = \frac{\partial f}{\partial f} * \frac{\partial f}{\partial b} = 1 * c$$

What about dbdc?

Let's analyze how this works

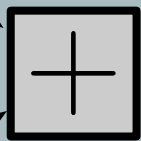
$$f = b * (a + b)$$

$$\frac{\partial f}{\partial a} = \frac{\partial f}{\partial f} * \frac{\partial f}{\partial c} \frac{\partial c}{\partial a} = 1 * b * 1 = b$$

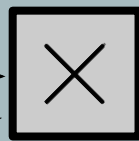
a

$$\frac{\partial f}{\partial c} = \frac{\partial f}{\partial f} * \frac{\partial f}{\partial c} = 1 * b$$

$$\frac{\partial f}{\partial f} = 1$$



c



f

b

$$\frac{\partial f}{\partial b} = \frac{\partial f}{\partial f} * \frac{\partial f}{\partial b} + \frac{\partial f}{\partial c} * \frac{\partial c}{\partial b} = 1 * c + 1 * b$$

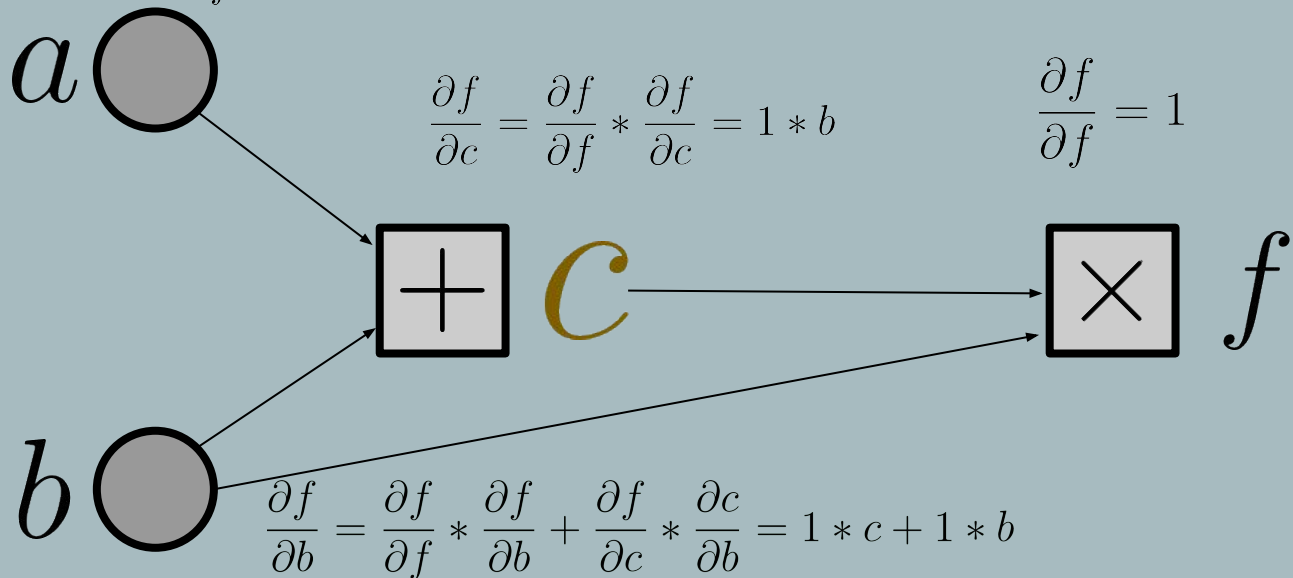
Just add it on

Let's analyze how this works

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

$$\frac{\partial f}{\partial a} = \frac{\partial f}{\partial f} * \frac{\partial f}{\partial c} \frac{\partial c}{\partial a} = 1 * b * 1 = b$$

a



$$\frac{\partial f}{\partial c} = \frac{\partial f}{\partial f} * \frac{\partial f}{\partial c} = 1 * b$$

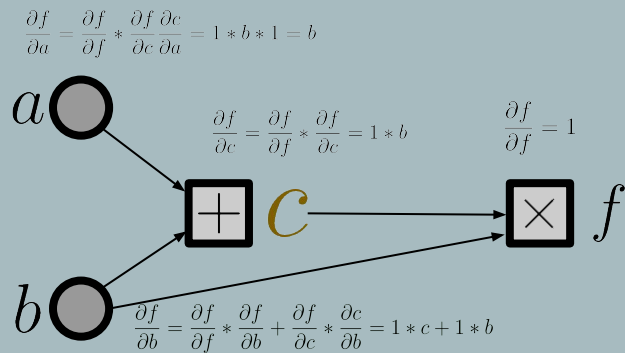
$$\frac{\partial f}{\partial f} = 1$$

b

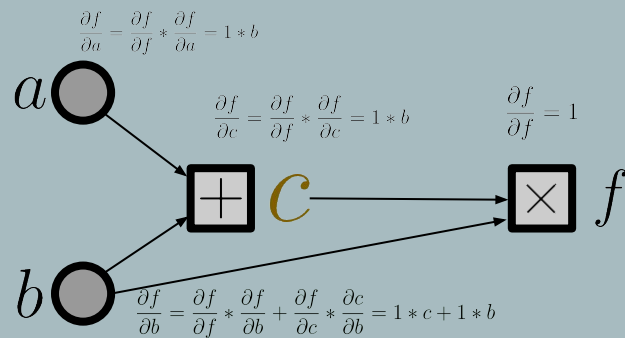
$$\frac{\partial f}{\partial b} = \frac{\partial f}{\partial f} * \frac{\partial f}{\partial b} + \frac{\partial f}{\partial c} * \frac{\partial c}{\partial b} = 1 * c + 1 * b$$

Let's understand the autograd framework

- Node class initialization:
 - Contains some data
 - I.e. **node a** contains its data input **a**
 - (tip: make this a numpy array holding data)
 - It also holds its gradient during backprop
 - Initialize it to a zero numpy array of the shape of the data.
 - tip: use `np.zeros_like()`
 - Also keep track of the parents of the node
 - Keep this as a **set** of parent nodes (so we don't duplicate parent nodes accidentally)
 - I.e. node C has to know its parents are nodes a and b



Let's understand the autograd framework



```
class Node():
    def __init__(self, data, _parents=()):
        self.data = np.array(data) # Store data as NumPy array
        self.grad = np.zeros_like(self.data) # Gradient initialized to zero
        self._parents = set(_parents) # Parent nodes in the computation graph
```

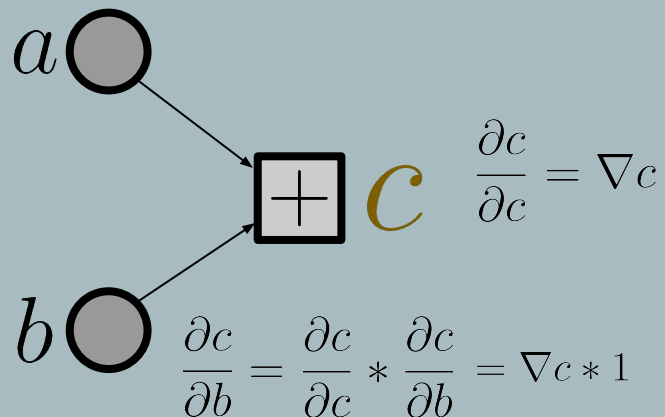
Let's implement primitives!

- Reminder: autograd works with a graph to incrementally perform backward operations.
- We need to implement operation primitives to facilitate this.
- Examples:
 - Addition
 - Multiplication
 - Matrix Multiplication
 - Subtraction
 - Log
 - Exponential
 - Division

Primitives: Addition

$$c = a + b$$

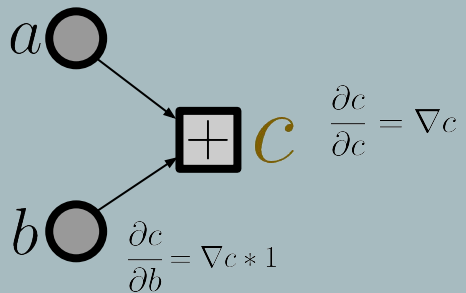
$$\frac{\partial c}{\partial a} = \frac{\partial c}{\partial c} * \frac{\partial c}{\partial a} = \nabla c * 1$$



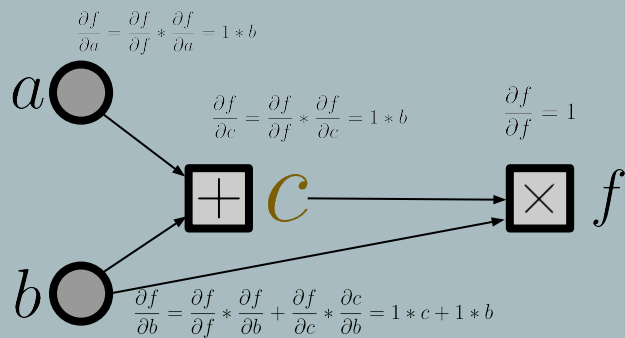
Primitives: Addition

- Addition primitive function:
 - Create a new **out node**
 - Add the data from the input nodes
 - Keep track of parents
 - Define a backward function
 - Manipulate the input nodes' gradients from within the function
 - Set the node's backward function to track it.

$$\frac{\partial c}{\partial a} = \nabla c * 1$$



Primitives: Addition

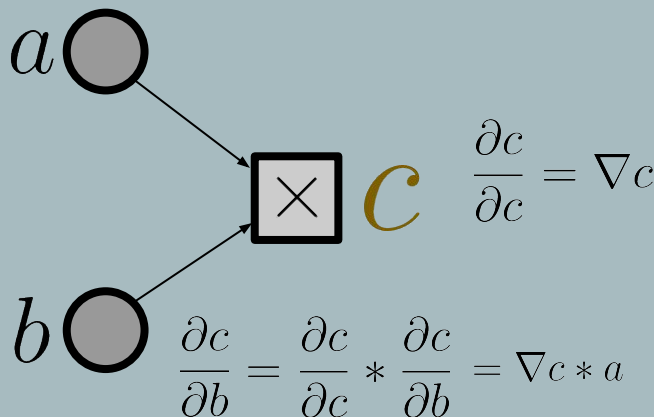


```
def add_node(a_node, b_node):  
    out = Node(a_node.data + b_node.data, (a_node, b_node))  
    def add_backward():  
        a_node.grad += out.grad # dL/dA = dL/dC  
        b_node.grad += out.grad # dL/dB = dL/dC  
    out.backward_fn = add_backward()  
    return out
```

Primitives: Multiplication

$$c = ab$$

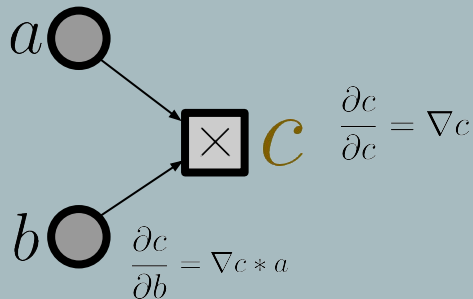
$$\frac{\partial c}{\partial a} = \frac{\partial c}{\partial c} * \frac{\partial c}{\partial a} = \nabla c * b$$



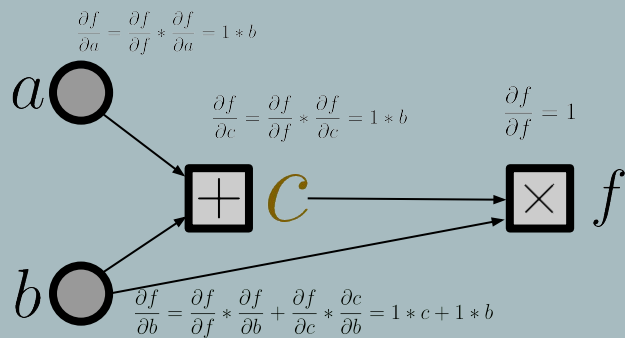
Primitives: Multiplication

- Addition primitive function:
 - Create a new **out node**
 - *Multiply* the data from the input nodes
 - Keep track of parents
 - Define a backward function
 - Manipulate the input nodes' gradients from within the function
 - Set the node's backward function to track it.

$$\frac{\partial c}{\partial a} = \nabla c * b$$



Primitives: Multiplication



```
def mul_node(a_node, b_node):  
    out = Node(a_node.data * b_node.data, (a_node, b_node))  
    def mul_backward():  
        a_node.grad += b_node.data * out.grad # dL/dA = B * dL/dC  
        b_node.grad += a_node.data * out.grad # dL/dB = A * dL/dC  
    out.backward_fn = mul_backward()  
    return out
```

Primitives: Mat Mul (Challenge)

$$c = a @ b$$

$$\frac{\partial c}{\partial a} = \nabla C \cdot b^T$$

$$\frac{\partial c}{\partial b} = a^T \nabla C$$

Notes:

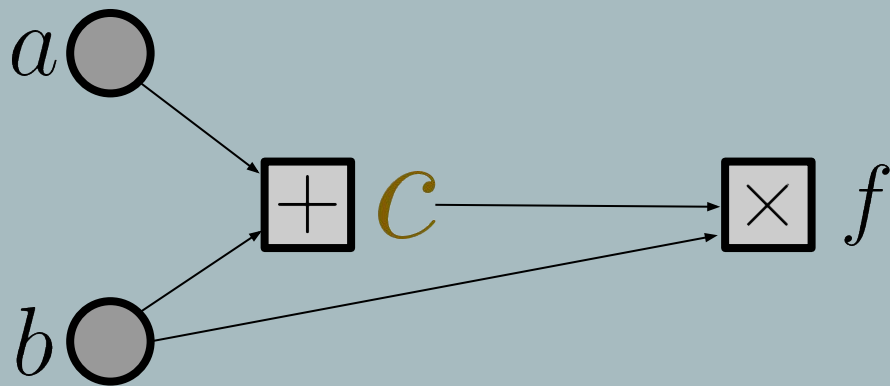
- Use @ for matmul in np
- Use np.dot for dot product multiplication in np

Primitives: Mat Mul (Challenge)

```
def matmul_node(a_node, b_node):  
    out = Node(a_node.data @ b_node.data, (a_node, b_node))  
    def matmul_backward():  
        a_node.grad += np.dot(out.grad, b_node.data.T) # Gradient wrt a  
        b_node.grad += np.dot(a_node.data.T, out.grad) # Gradient wrt b  
    out.backward_fn = matmul_backward  
    return out
```

The Backward Method

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



backward_list:

- []

parents:

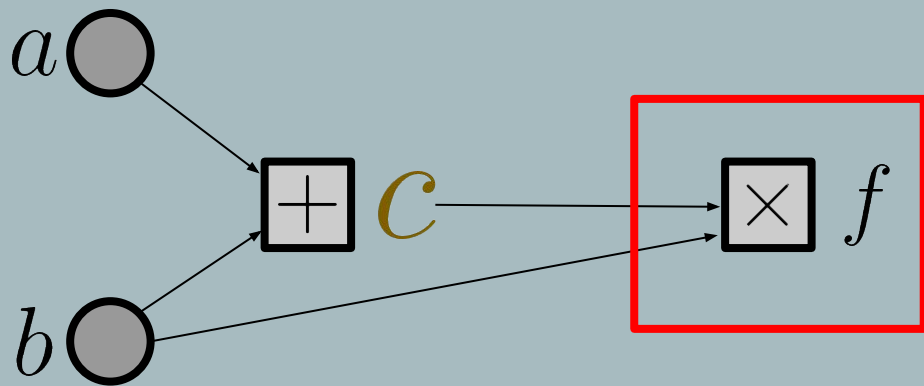
- None

Current:

- None

The Backward Method

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



backward_list:

- [f]

parents:

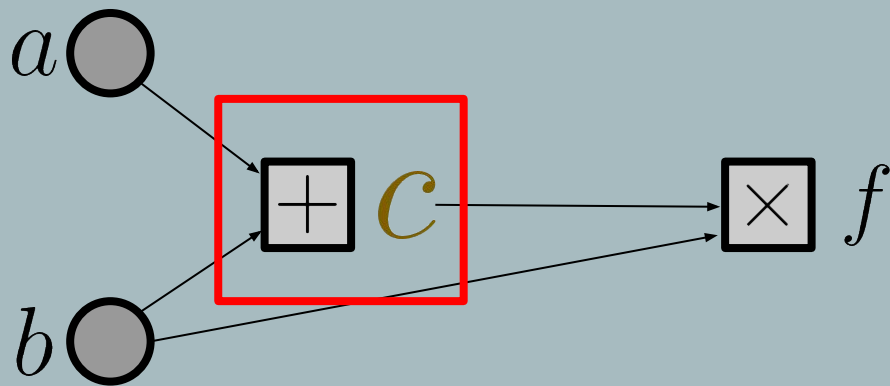
- c, b

Current:

- f

The Backward Method

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



backward_list:

- $[f, c]$

parents:

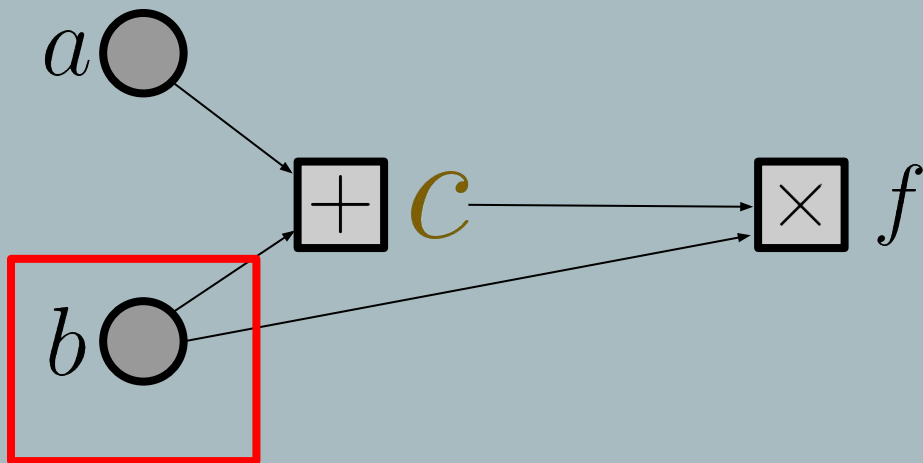
- a, b

Current:

- c, b (for later)

The Backward Method

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



backward_list:

- [f, c, b]

parents:

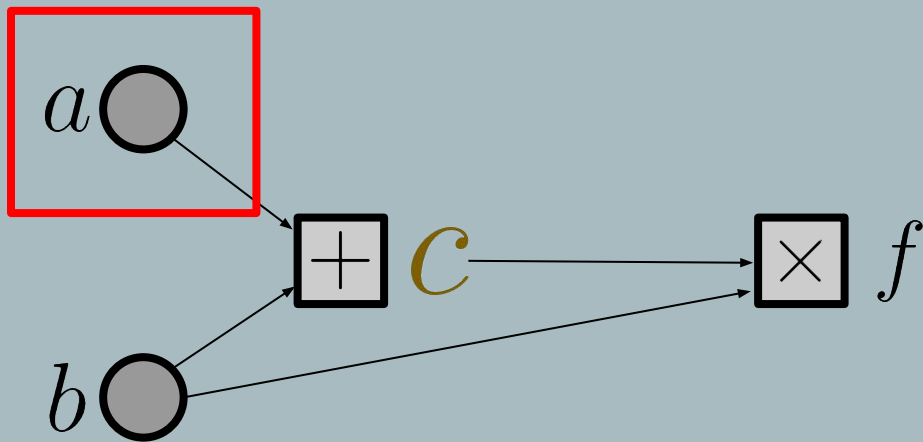
- None

Current:

- **b**, a (for later)

The Backward Method

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



backward_list:
• [f, c, b, a]
parents:
• None
Current:
• a

The Backward Method

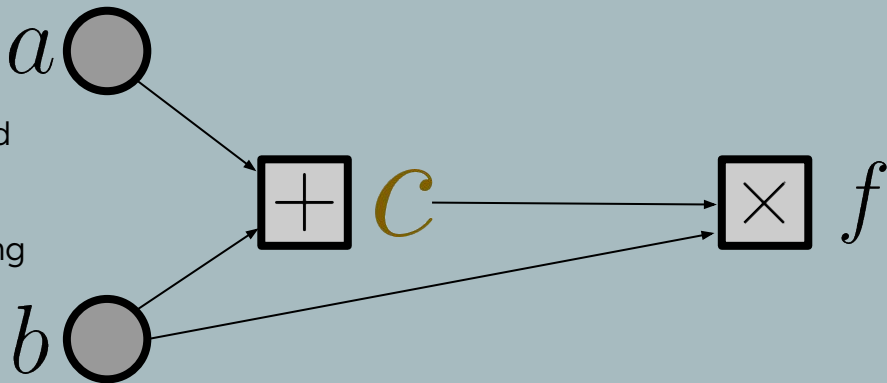
$$f = b * (a + b)$$

backward_list:

- [f, c, b, a]

Iterate through the backward list.

Note how we're properly going backward now!



Backward Method Pseudocode

Initialize the current node's grad to an array of all 1's as a placeholder

initialize backward_list as an empty list

Recursively take in a node

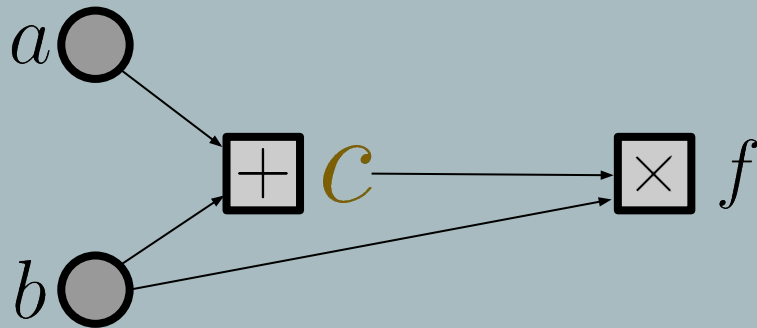
If the node is already in the backward_list, skip

Iterate through the node's parents

recursively build the list from the parent nodes

Iterate through all the nodes in backward_list:

if the node has parents, call node.backward_fn()



Backward Method Code

```
class Node(Node):
    def backward(self):
        backward_list = []
        self.grad = np.ones_like(self.data)
        def build_backward_list(n):
            if n not in backward_list:
                backward_list.append(n)
                for parent in n._parents:
                    build_backward_list(parent)
        build_backward_list(self)

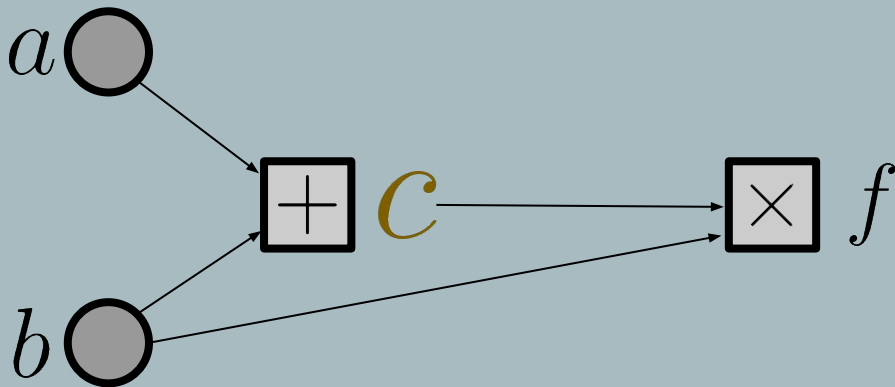
        for node in backward_list:
            if node._parents:
                node.backward_fn()
```

Let's make this computational graph now!

Set a to 7.0

Set b to 8.0

And call backward on it



How to use our basic autodiff engine

```
A= Node(7)
B = Node(8)

C = add_node(A, B)
F = mul_node(C, B)

F.grad = 1.0
F.backward()

print(f"a_grad: {A}")
print(f"b_grad: {B}")
print(f"c_grad: {C}")
print(f"f_grad: {F}")
```

Activation Functions

- For larger operations, you will want to use your primitives
- All you simply have to do is call `out.backward()` and all the derivatives will be computed for you!
- If you manually create new nodes, they must NOT rely directly on the data of the input.
 - If you want to use the data directly from the input, you must use the primitives so the input gets chained in with the rest of the graph.
 - You MAY create other matrices of a similar shape as the input. That's fine!

SoftPlus

$$f(x) = \ln(1 + e^x)$$

Tips:

- From our current framework, you must create a numpy array of 1's with `np.ones_like()` to be a similar shape to `x`
- We have provided you the `ln` and `exp` primitives :)

SoftPlus Node

```
def softplus_node(x):  
    exp_x = exp_node(x)  
  
    one_exp_x = add_node(Node(np.ones_like(x.data)), exp_x)  
  
    out = log_node(one_exp_x)  
  
    return out
```

Leaky ReLU

$$\text{LeakyReLU}(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha x & \text{if } x \leq 0 \end{cases}$$

Tips:

- Create an node with a matrix of alpha values which has a similar shape to the input data
- Create a mask of all the 0/positive values with np.where
- Create a mask of all the negative values with np.where

Leaky ReLU Node

```
def leaky_relu_node(input, alpha=0.01):  
    alpha_node = Node(np.ones_like(input.data) * alpha)  
  
    positives_mask_node = Node(np.where(input.data >= 0., 1., 0.))  
    negatives_mask_node = Node(np.where(input.data < 0., 1., 0.))  
  
    alpha_mask_node = mul_node(alpha_node, negatives_mask_node)  
  
    positive_part = mul_node(input, positives_mask_node)  
    negative_part = mul_node(input, alpha_mask_node)  
  
    out = add_node(positive_part, negative_part)  
    return out
```

Tanh

$$f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

Tips:

- Incrementally build the graph for the numerator and denominator
- Note how the numerator and denominator both have e^x
 - We recommend building **separate graphs/separate instances** of the intermediate variables needed for the numerator and denominator
 - This is because in our backward method, the gradient for intermediate values (such as $-x$ and e^x) might only be updated by the numerator, or only be updated by the denominator. Thus, if you do them separately, they will update properly.
 - You can just do `neg_x1` and `negx_2`, or `exp_x1` and `exp_x2`, and just make the same node twice.

Tanh Node

```
def tanh_node(x):  
    neg_one = Node(-1.0 * np.ones_like(x.data))  
  
    neg_x1 = mul_node(neg_one, x)  
    exp_x1 = exp_node(x)  
    exp_neg_x1 = exp_node(neg_x1)  
    neg_exp_neg_x1 = mul_node(exp_neg_x1, neg_one)  
  
    num = add_node(exp_x1, neg_exp_neg_x1)  
  
    neg_input2 = mul_node(neg_one, x)  
    exp_x2 = exp_node(x)  
    exp_neg_x2 = exp_node(neg_input2)  
  
    den = add_node(exp_x2, exp_neg_x2)  
  
    out = div_node(num, den)  
    return out
```

Moving to Gradient
Buffers (more like
the homework
bonuses ;)