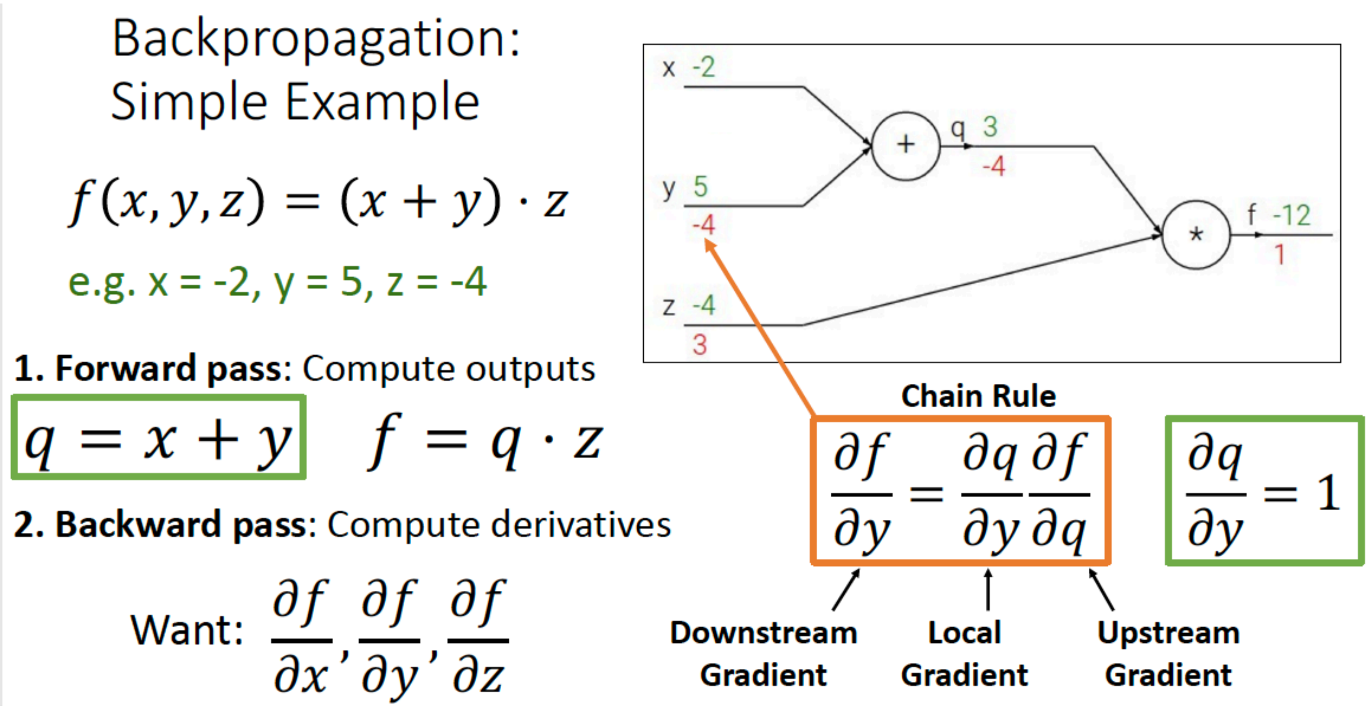
Lecture 6 - Backpropagation

1. Backpropagation

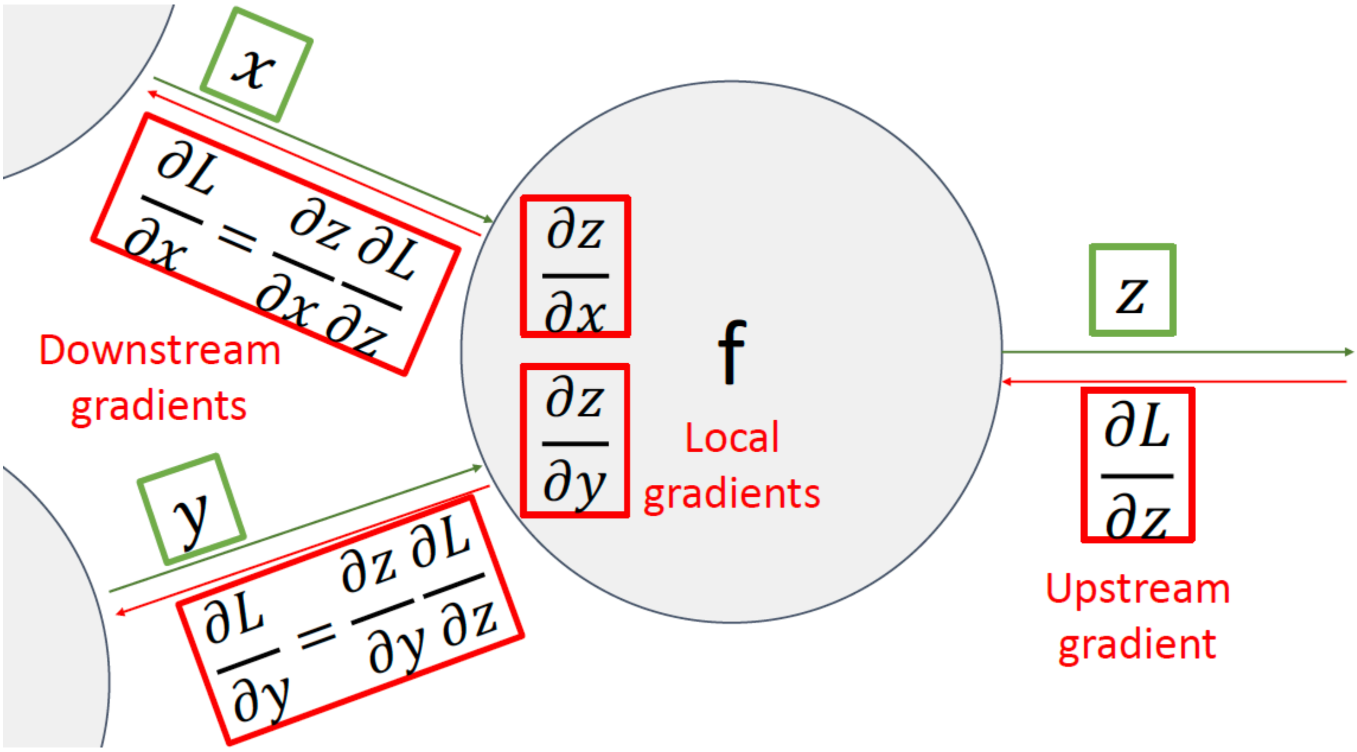
--> Computational Graphs



Downstream Gradient: the wanted gradient (need to be calculated)

Local Gradient: the direct gradient (gradient of the current function)

Upstream Gradient: close to the output (directly calculated through the output)



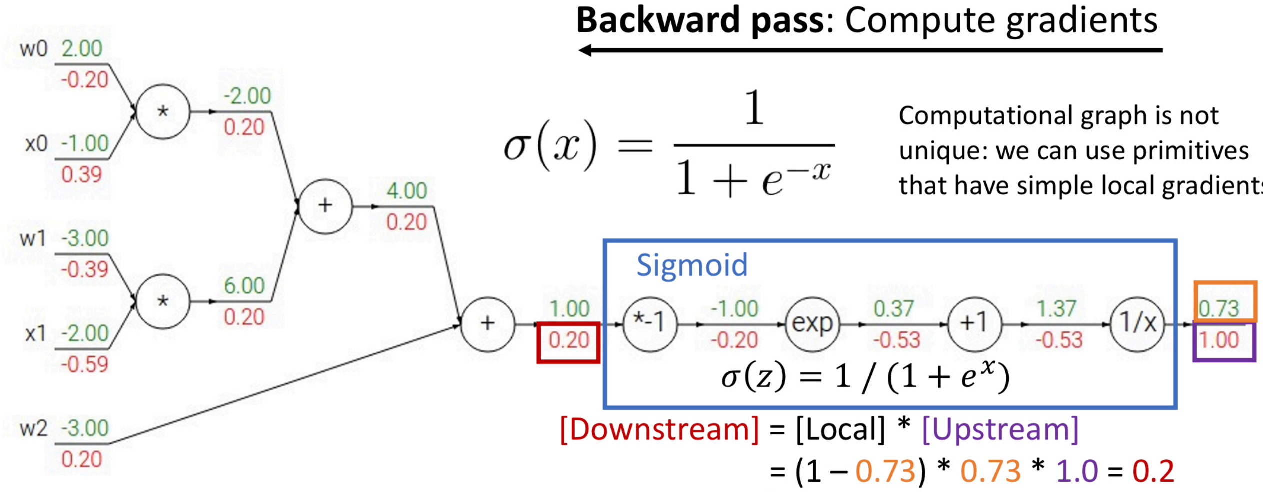
Another Example

$$f(x,w) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+w_2)}} = \boxed{\sigma(w_0x_0+w_1x_1+w_2)}$$

Backward pass: Compute gradients

$$\sigma(x) = \frac{1}{1+e^{-x}}$$

Computational graph is not unique: we can use primitives that have simple local gradients

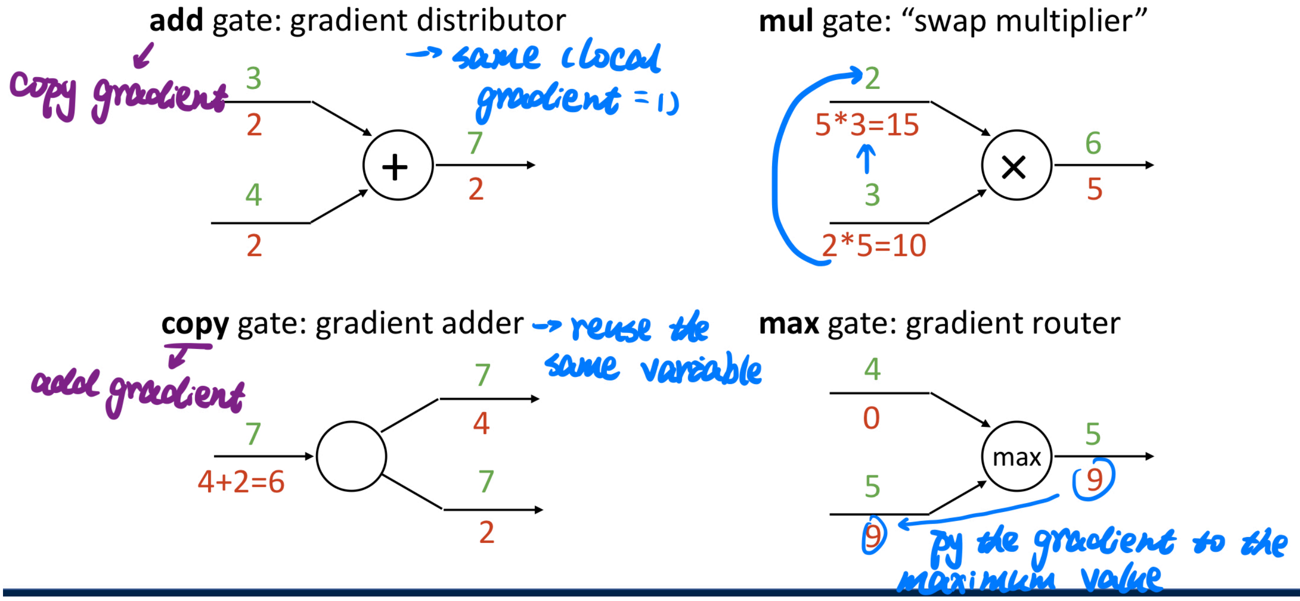


[Downstream] = [Local] * [Upstream]

= (1 – 0.73) * 0.73 * 1.0 = 0.2

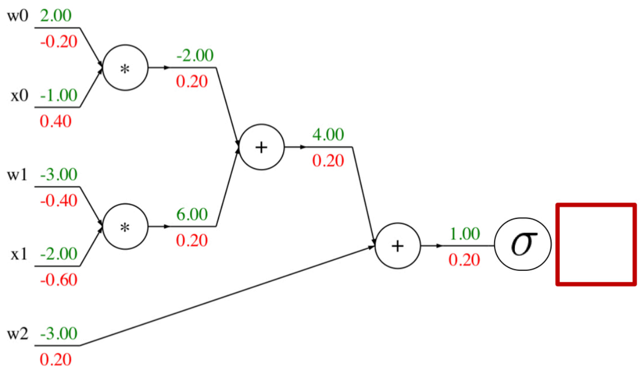
Sigmoid local gradient: $\frac{\partial}{\partial x}[\sigma(x)] = \frac{e^{-x}}{(1+e^{-x})^2} = \left(\frac{1+e^{-x}-1}{1+e^{-x}}\right)\left(\frac{1}{1+e^{-x}}\right) = (1-\sigma(x))\sigma(x)$

- **Patterns in Gradient Flow**



- **Implementation**
  - "Flat Gradient Code"

Backprop Implementation: "Flat" gradient code:

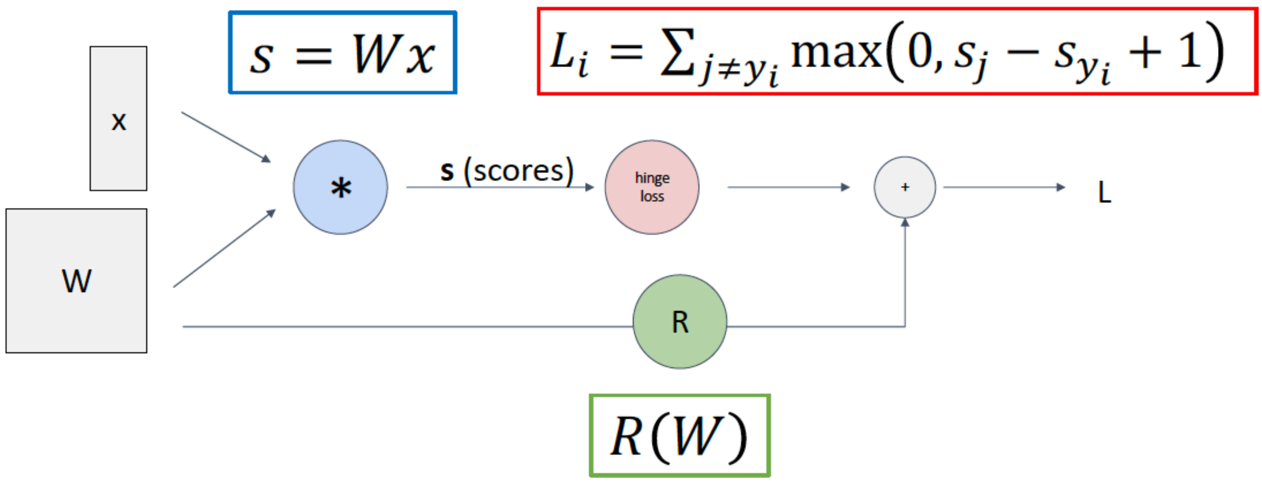


Forward pass: Compute output

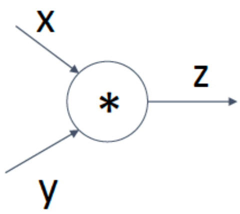
```
def f(w0, x0, w1, x1, w2):
    s0 = w0 * x0
    s1 = w1 * x1
    s2 = s0 + s1
    s3 = s2 + w2
    L = sigmoid(s3)
```

Base case

```
    grad_L = 1.0
Sigmoid ← grad_s3 = grad_L * (1 - L) * L
Add Gate  grad_w2 = grad_s3
          grad_s2 = grad_s3
Add Gate  grad_s0 = grad_s2
          grad_s1 = grad_s2
Multiply  grad_w1 = grad_s1 * x1
          grad_x1 = grad_s1 * w1
Multiply  grad_w0 = grad_s0 * x0
          grad_x0 = grad_s0 * w0
```



- Modular API



(x,y,z are scalars)

```python
class Multiply(torch.autograd.Function):
  @staticmethod
  def forward(ctx, x, y):
    ctx.save_for_backward(x, y)
    z = x * y
    return z
  @staticmethod
  def backward(ctx, grad_z):
    x, y = ctx.saved_tensors
    grad_x = y * grad_z   # dz/dx * dL/dz
    grad_y = x * grad_z   # dz/dy * dL/dz
    return grad_x, grad_y
```

Need to stash some values for use in backward

Upstream gradient

Multiply upstream and local gradients