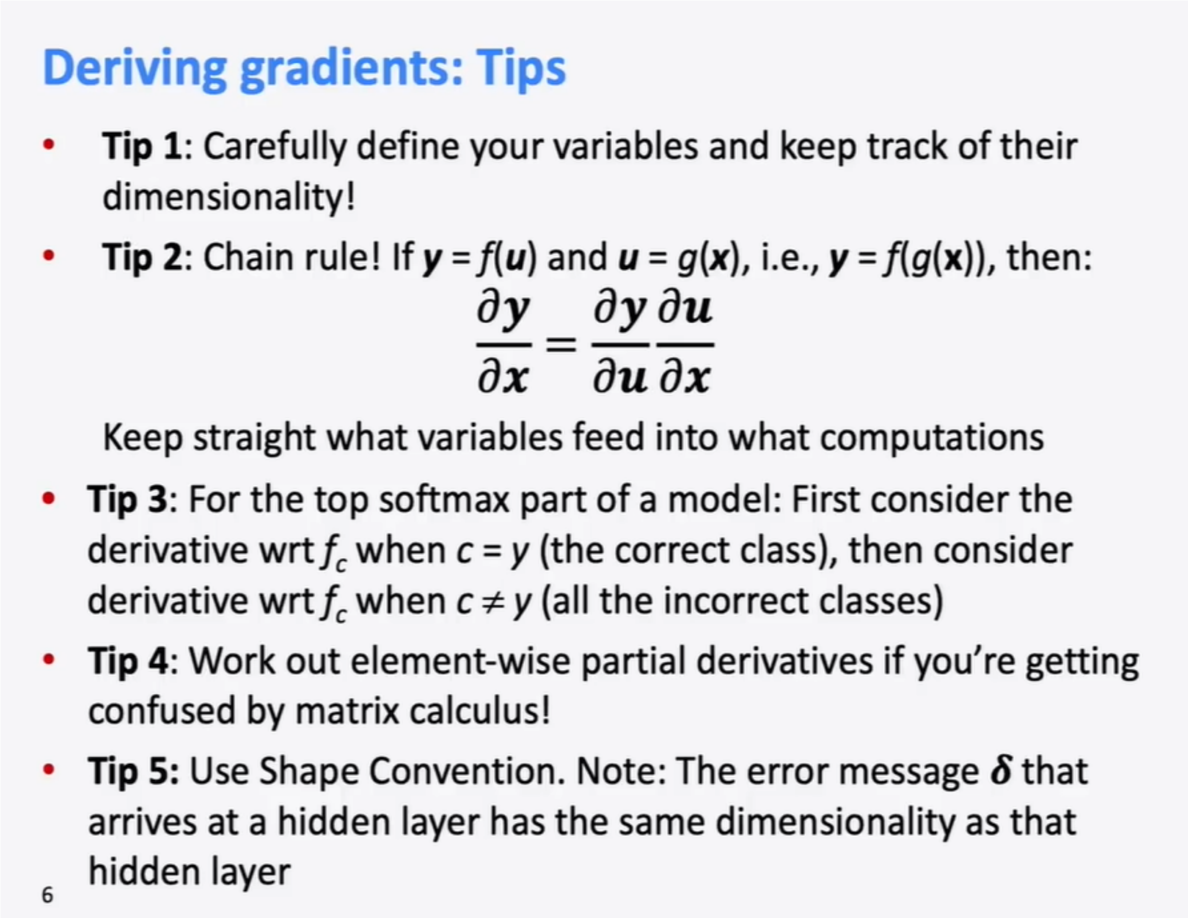
Backpropagation

手推bp的流程



Deriving gradients: Tips

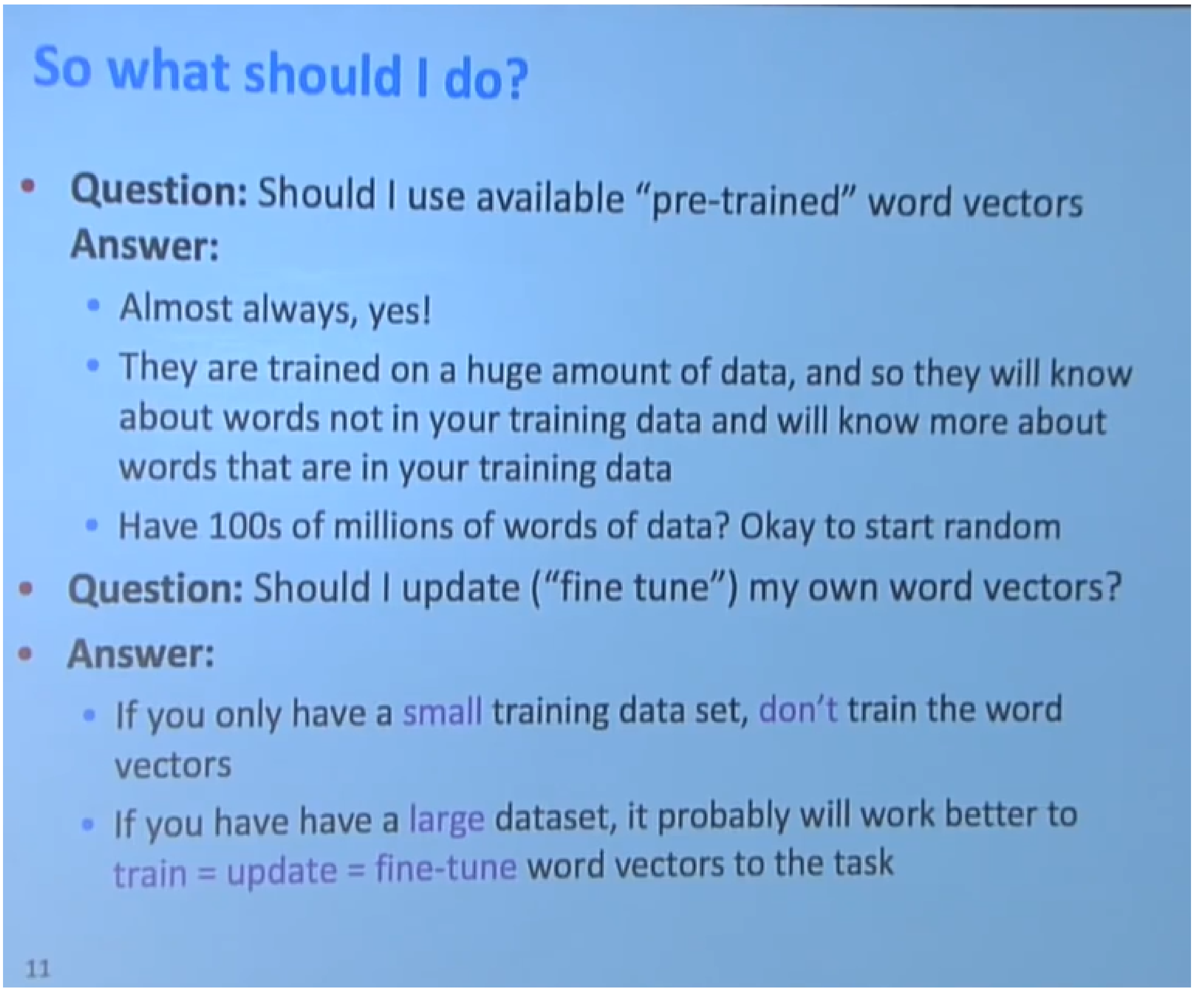
- **Tip 1**: Carefully define your variables and keep track of their dimensionality!
- **Tip 2**: Chain rule! If $\boldsymbol{y} = f(\boldsymbol{u})$ and $\boldsymbol{u} = g(\boldsymbol{x})$, i.e., $\boldsymbol{y} = f(g(\boldsymbol{x}))$, then:

  $$\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}} = \frac{\partial \boldsymbol{y}}{\partial \boldsymbol{u}} \frac{\partial \boldsymbol{u}}{\partial \boldsymbol{x}}$$

  Keep straight what variables feed into what computations
- **Tip 3**: For the top softmax part of a model: First consider the derivative wrt f_c when $c = y$ (the correct class), then consider derivative wrt f_c when $c \neq y$ (all the incorrect classes)
- **Tip 4**: Work out element-wise partial derivatives if you're getting confused by matrix calculus!
- **Tip 5:** Use Shape Convention. Note: The error message $\boldsymbol{\delta}$ that arrives at a hidden layer has the same dimensionality as that hidden layer

6

下游任务是否用word vec输入



So what should I do?

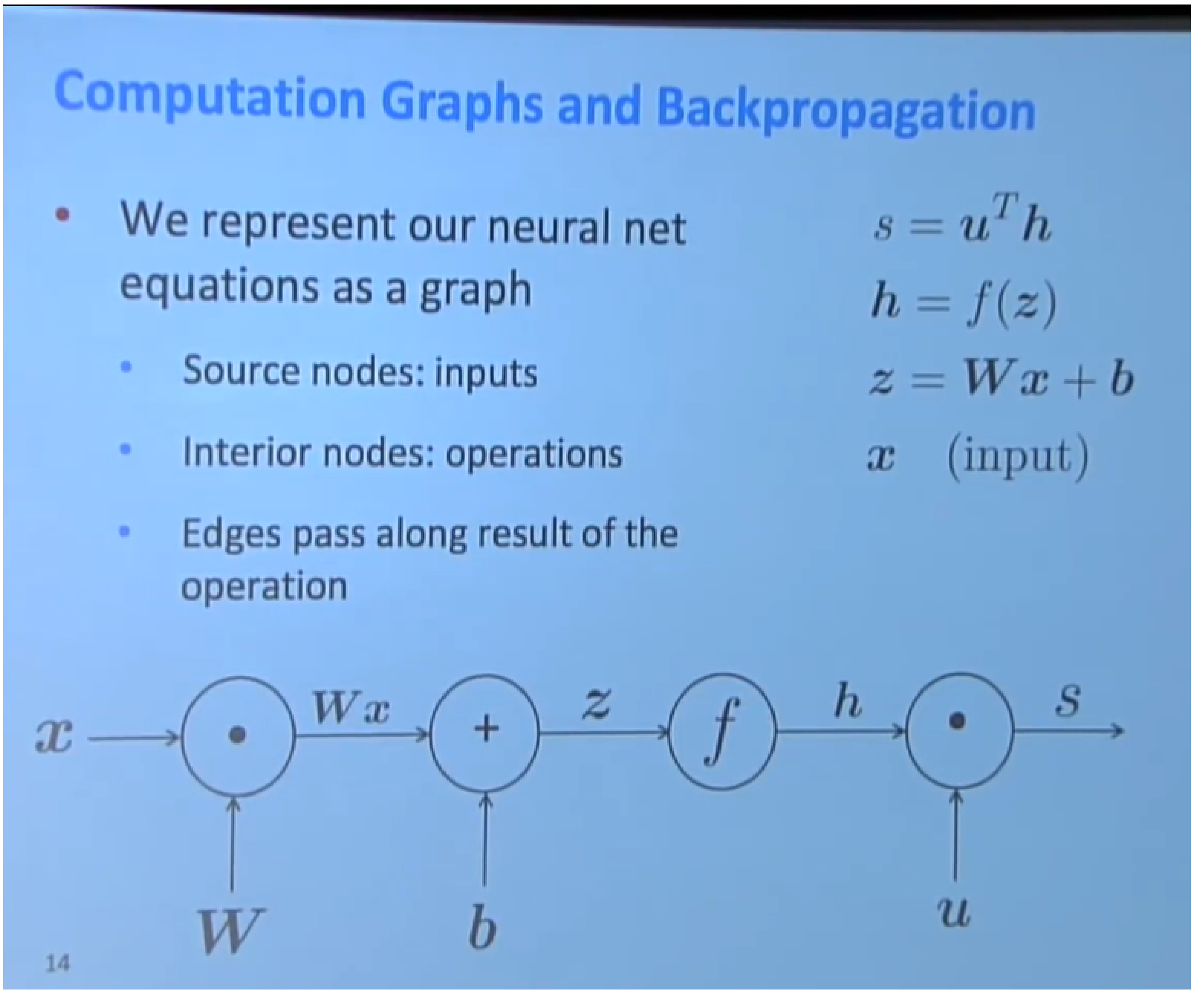
- **Question:** Should I use available “pre-trained” word vectors
  Answer:
  - Almost always, yes!
  - They are trained on a huge amount of data, and so they will know about words not in your training data and will know more about words that are in your training data
  - Have 100s of millions of words of data? Okay to start random
- **Question:** Should I update (“fine tune”) my own word vectors?
- **Answer:**
  - If you only have a small training data set, don’t train the word vectors
  - If you have have a large dataset, it probably will work better to train = update = fine-tune word vectors to the task

11

上图说明了wordvec是否应该在下游任务中更新。

Computation Graphs and Backpropagation

用图表示神经网络结构，下图是forward propagation

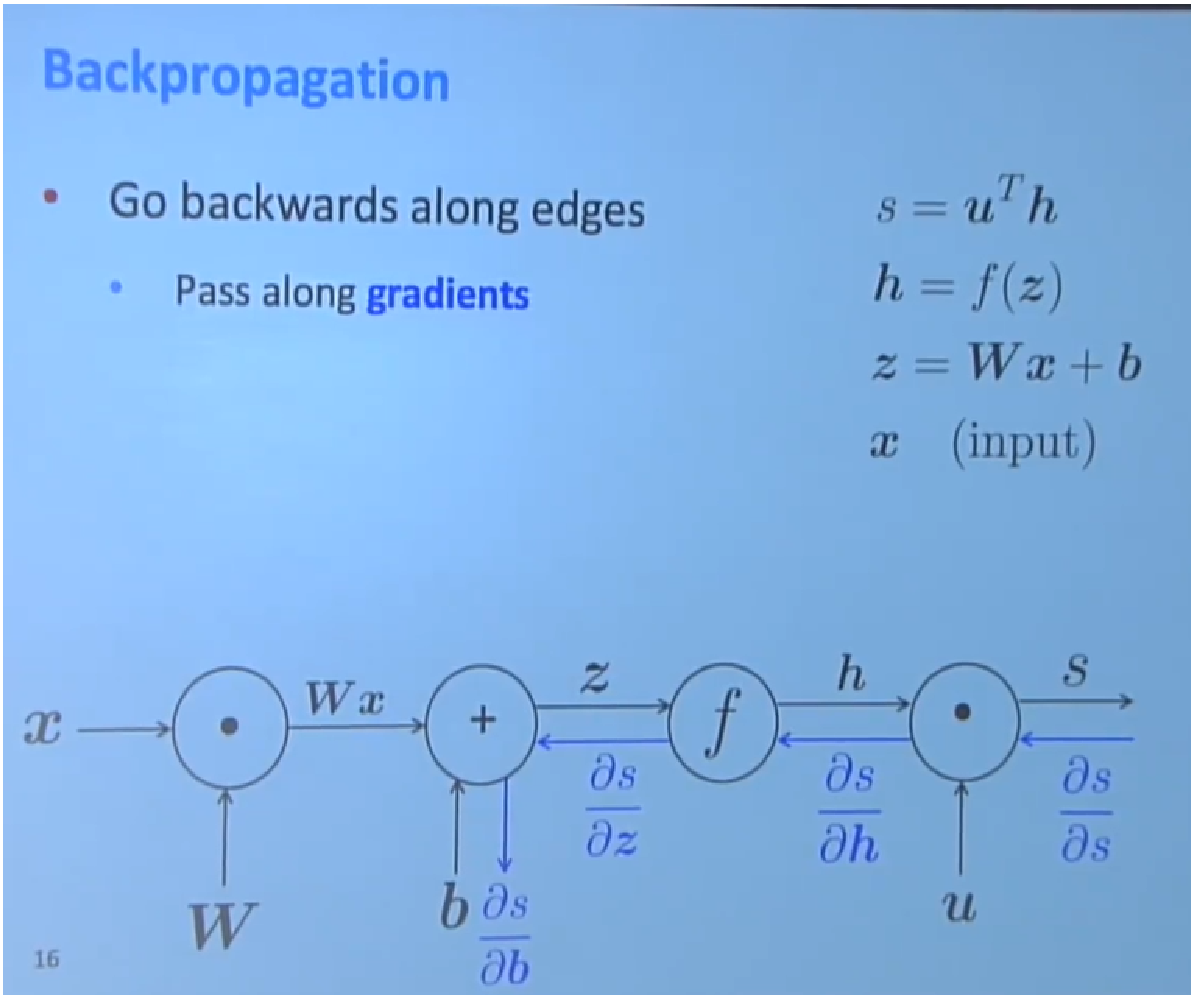


Computation Graphs and Backpropagation

- We represent our neural net equations as a graph
  - Source nodes: inputs
  - Interior nodes: operations
  - Edges pass along result of the operation

$$s = u^T h$$
$$h = f(z)$$
$$z = Wx + b$$
$$x \quad \text{(input)}$$

下图是backpropagation



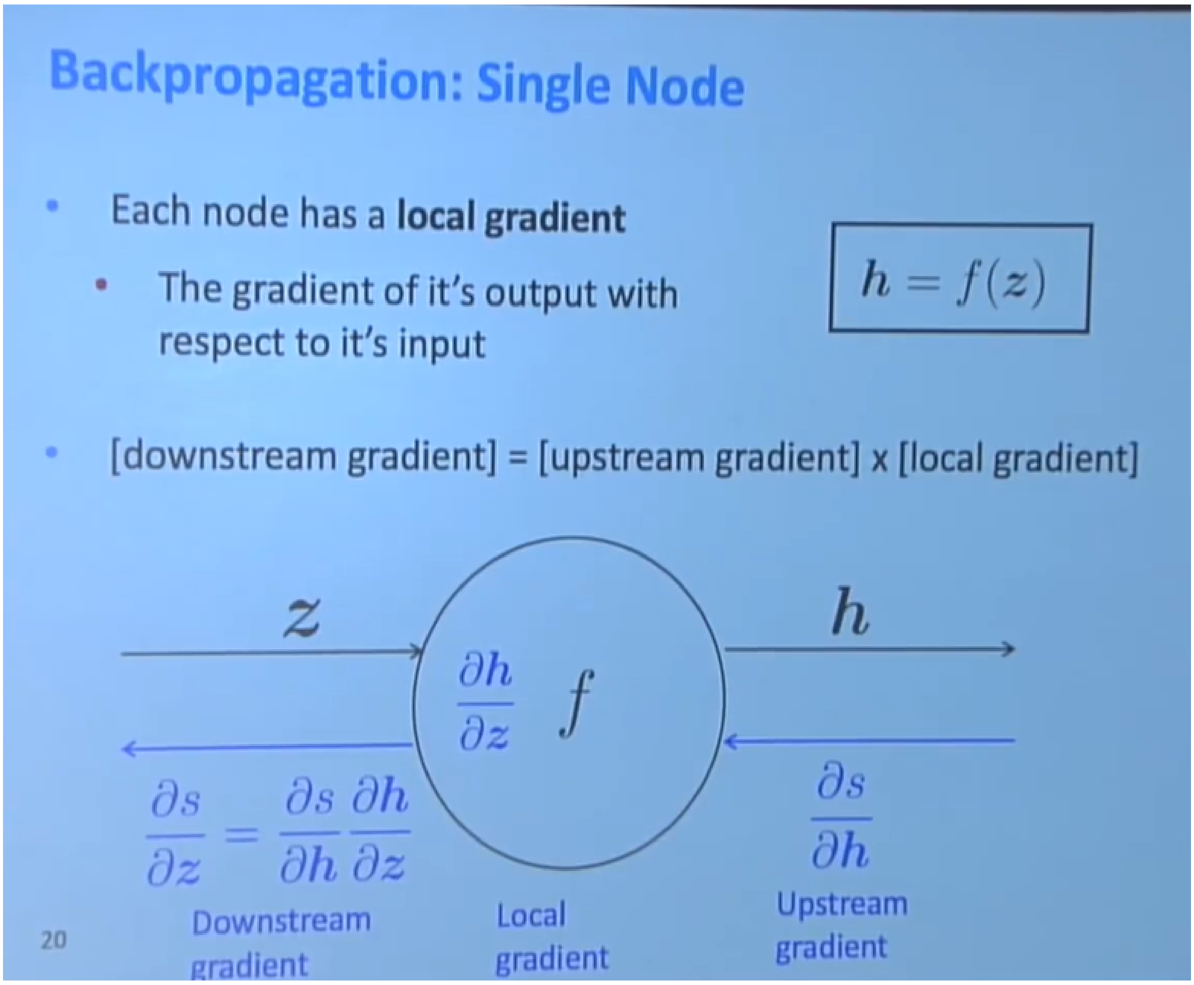
Backpropagation

- Go backwards along edges
  - Pass along **gradients**

$$s = u^T h$$
$$h = f(z)$$
$$z = Wx + b$$
$$x \quad \text{(input)}$$

x
$\cdot$
Wx
$+$
z
f
h
$\cdot$
s
$\frac{\partial s}{\partial z}$
$\frac{\partial s}{\partial h}$
$\frac{\partial s}{\partial s}$
W
b
$\frac{\partial s}{\partial b}$
u

下图是用chain rule计算多节点的梯度，重要的公式$downstream = upstream * local$



Backpropagation: Single Node

- Each node has a **local gradient**
  - The gradient of it's output with respect to it's input

$h = f(z)$

- [downstream gradient] = [upstream gradient] x [local gradient]

z

h

$\frac{\partial h}{\partial z}$ f

$\frac{\partial s}{\partial z} = \frac{\partial s}{\partial h} \frac{\partial h}{\partial z}$

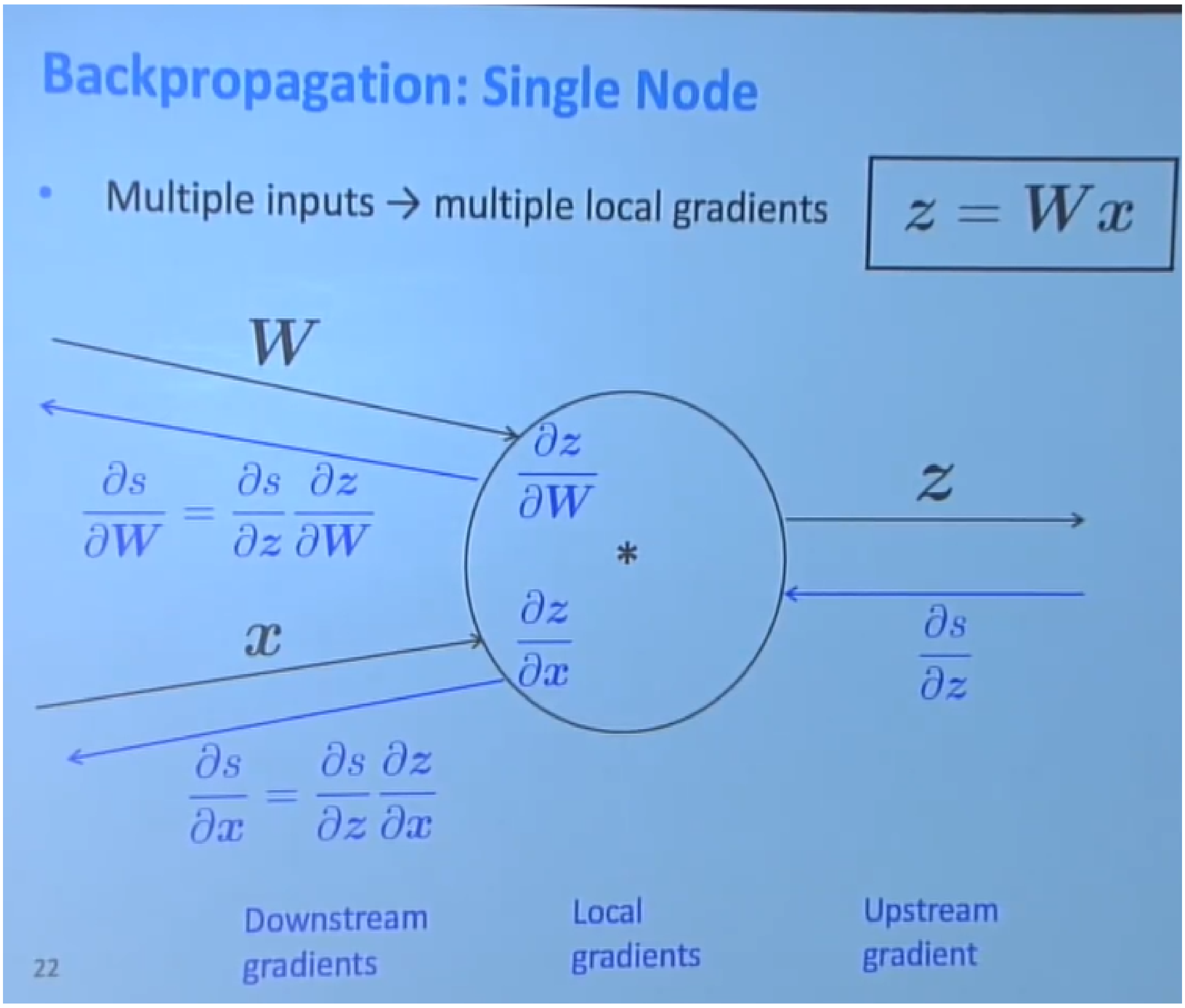
$\frac{\partial s}{\partial h}$

Downstream gradient

Local gradient

Upstream gradient

20



Backpropagation: Single Node

- Multiple inputs → multiple local gradients

$z = Wx$

W

$\frac{\partial s}{\partial W} = \frac{\partial s}{\partial z} \frac{\partial z}{\partial W}$

$\frac{\partial z}{\partial W}$

$*$

$\frac{\partial z}{\partial x}$

z

$\frac{\partial s}{\partial z}$

x

$\frac{\partial s}{\partial x} = \frac{\partial s}{\partial z} \frac{\partial z}{\partial x}$

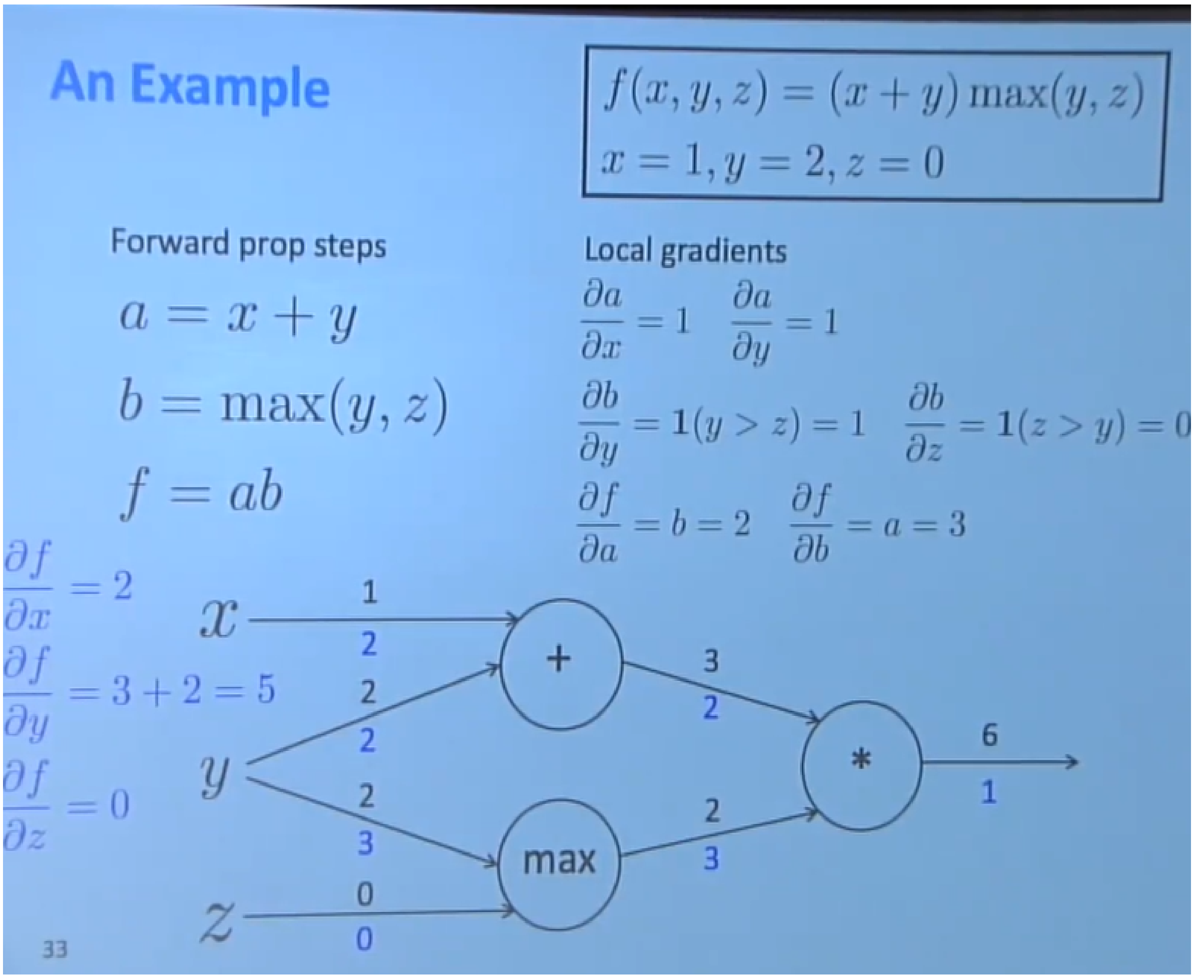
Downstream gradients

Local gradients

Upstream gradient

22

一个backpropagation例子



An Example

$$f(x, y, z) = (x + y)\max(y, z)$$

$$x = 1, y = 2, z = 0$$

Forward prop steps

$$a = x + y$$

$$b = \max(y, z)$$

$$f = ab$$

Local gradients

$$\frac{\partial a}{\partial x} = 1 \quad \frac{\partial a}{\partial y} = 1$$

$$\frac{\partial b}{\partial y} = \mathbf{1}(y > z) = 1 \quad \frac{\partial b}{\partial z} = \mathbf{1}(z > y) = 0$$

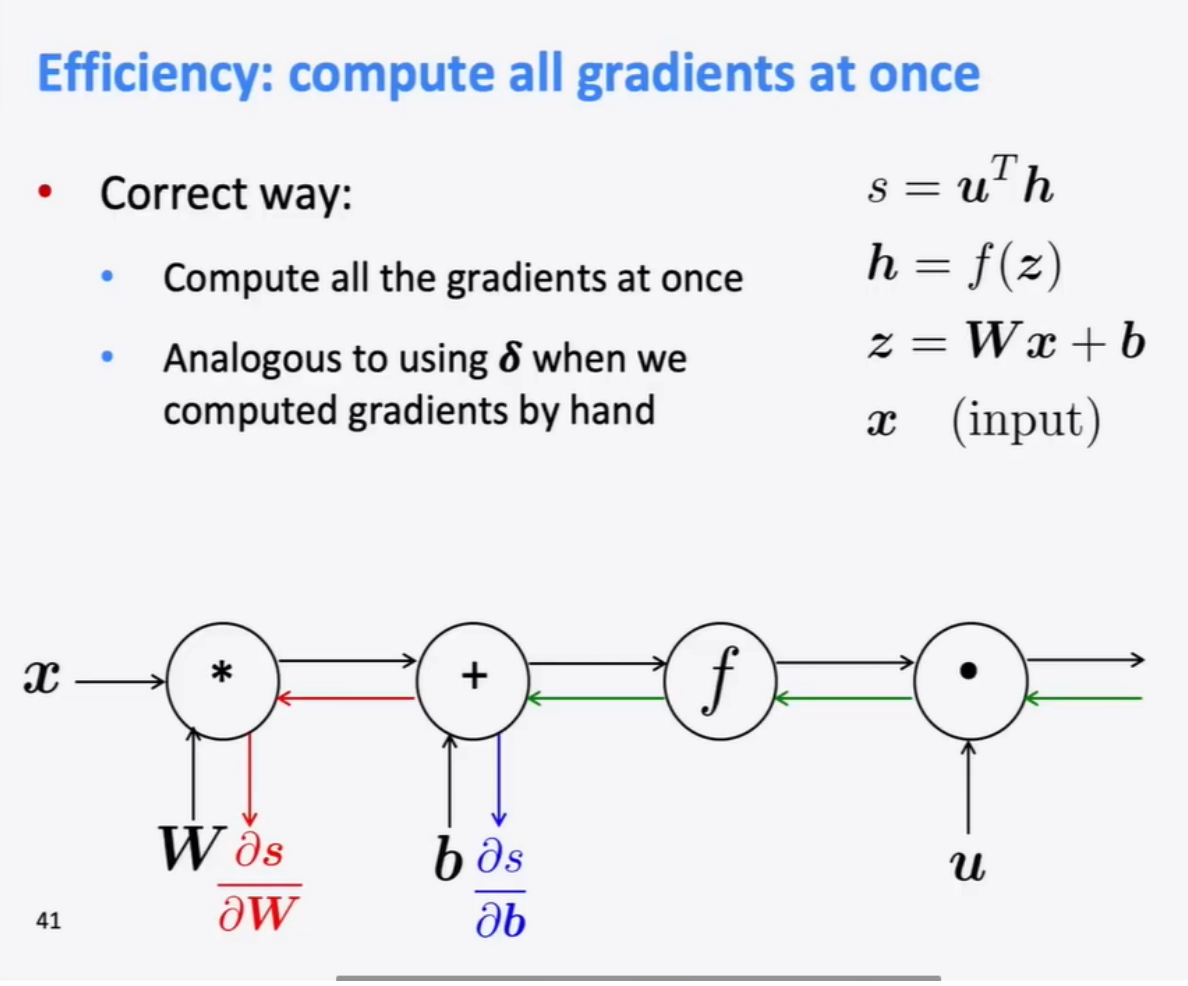
$$\frac{\partial f}{\partial a} = b = 2 \quad \frac{\partial f}{\partial b} = a = 3$$

$$\frac{\partial f}{\partial x} = 2$$

$$\frac{\partial f}{\partial y} = 3 + 2 = 5$$

$$\frac{\partial f}{\partial z} = 0$$

能算出左边的结果即可，$\frac{\partial f}{\partial x}$物理意义是x改变0.1，f就会改变0.2。



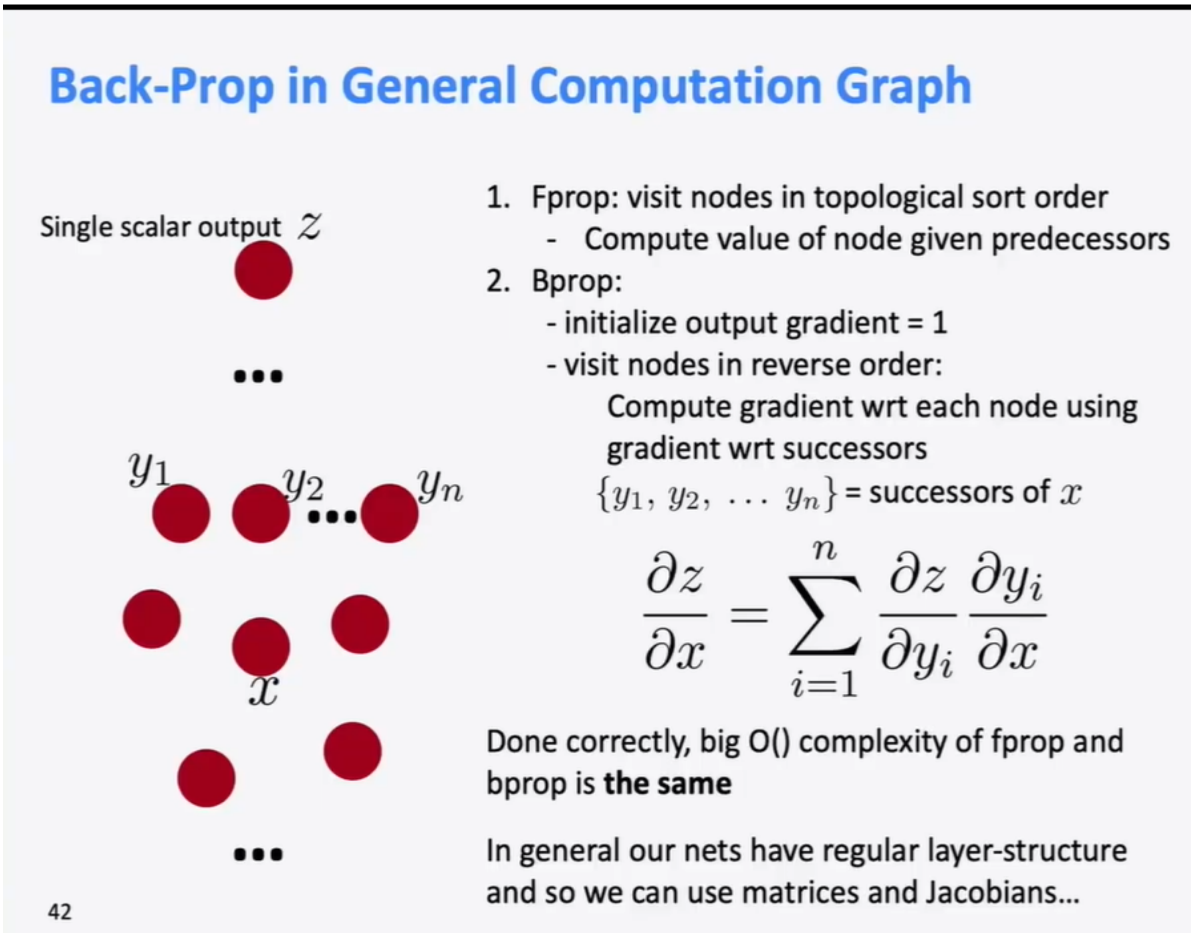
Efficiency: compute all gradients at once

- Correct way:
  - Compute all the gradients at once
  - Analogous to using $\boldsymbol{\delta}$ when we computed gradients by hand

$$s = \boldsymbol{u}^T \boldsymbol{h}$$
$$\boldsymbol{h} = f(\boldsymbol{z})$$
$$\boldsymbol{z} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}$$
$$\boldsymbol{x} \quad \text{(input)}$$

$\boldsymbol{x}$
*
+
f
•
$\boldsymbol{W}$
$\frac{\partial s}{\partial \boldsymbol{W}}$
$\boldsymbol{b}$
$\frac{\partial s}{\partial \boldsymbol{b}}$
$\boldsymbol{u}$
41

上图说明了，上游的梯度(绿色)可以保留，不用重复计算。



Back-Prop in General Computation Graph

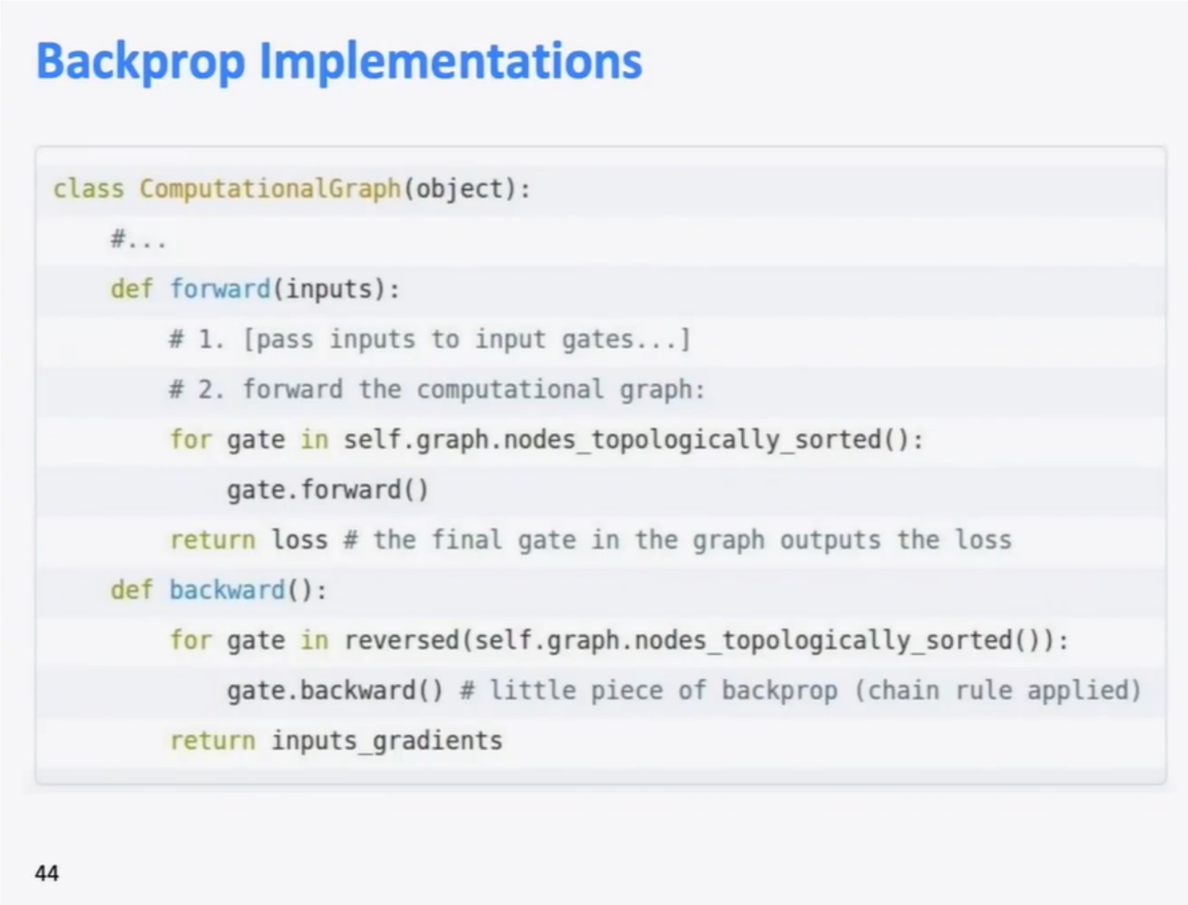
1. Fprop: visit nodes in topological sort order
   - Compute value of node given predecessors
2. Bprop:
   - initialize output gradient = 1
   - visit nodes in reverse order:
     Compute gradient wrt each node using gradient wrt successors
     $\{y_1, y_2, \ldots y_n\}$ = successors of x

$$\frac{\partial z}{\partial x} = \sum_{i=1}^{n} \frac{\partial z}{\partial y_i} \frac{\partial y_i}{\partial x}$$

Done correctly, big O() complexity of fprop and bprop is **the same**

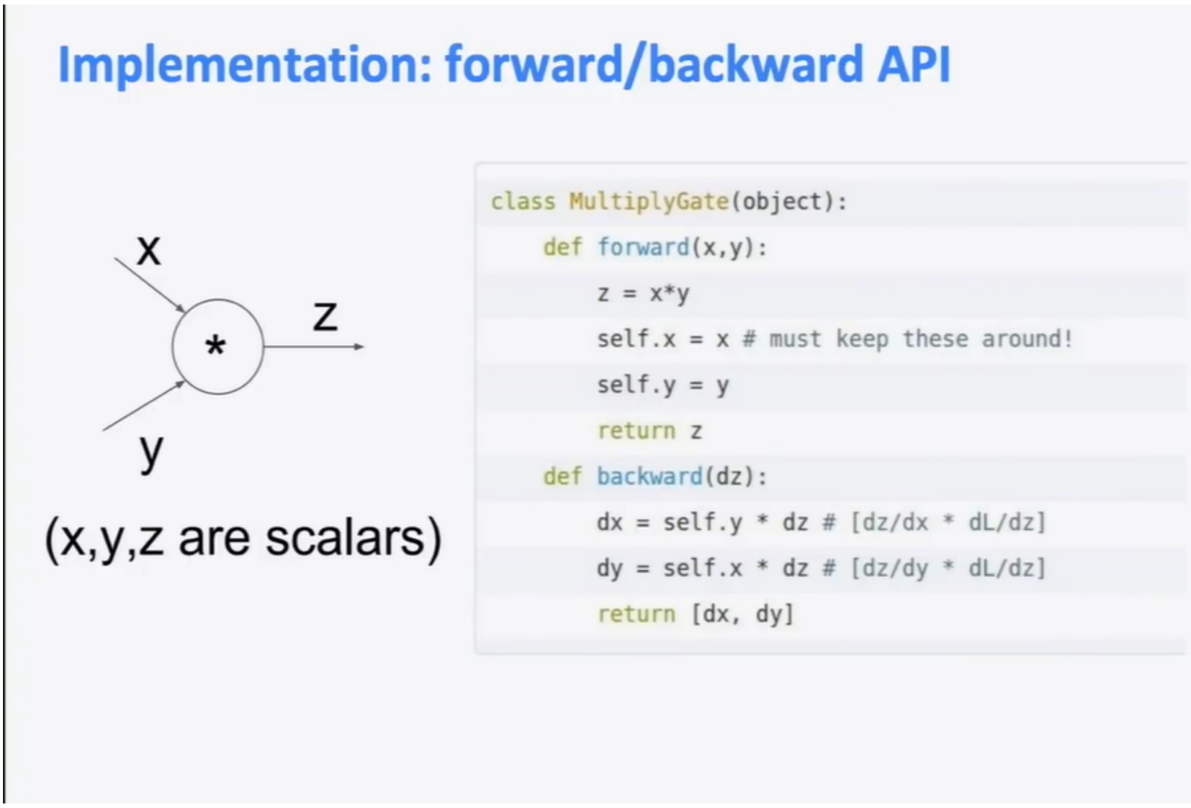
In general our nets have regular layer-structure and so we can use matrices and Jacobians...

上图是bp的构图过程和复杂度。



Backprop Implementations

```
class ComputationalGraph(object):
    #...
    def forward(inputs):
        # 1. [pass inputs to input gates...]
        # 2. forward the computational graph:
        for gate in self.graph.nodes_topologically_sorted():
            gate.forward()
        return loss # the final gate in the graph outputs the loss
    def backward():
        for gate in reversed(self.graph.nodes_topologically_sorted()):
            gate.backward() # little piece of backprop (chain rule applied)
        return inputs_gradients
```



Implementation: forward/backward API

(x,y,z are scalars)

```
class MultiplyGate(object):
    def forward(x,y):
        z = x*y
        self.x = x # must keep these around!
        self.y = y
        return z
    def backward(dz):
        dx = self.y * dz # [dz/dx * dL/dz]
        dy = self.x * dz # [dz/dy * dL/dz]
        return [dx, dy]
```

上面两张图是深度学习框架图计算的流程。