Natural Language Processing with Deep Learning

CS224N/Ling284

Christopher Manning

Lecture 4: Backpropagation and computation graphs

Lecture Plan

Lecture 4: Backpropagation and computation graphs

1. Matrix gradients for our simple neural net and some tips [15 mins]
2. Computation graphs and backpropagation [40 mins]
3. Stuff you should know [15 mins]
    a. Regularization to prevent overfitting
    b. Vectorization
    c. Nonlinearities
    d. Initialization
    e. Optimizers
    f. Learning rates

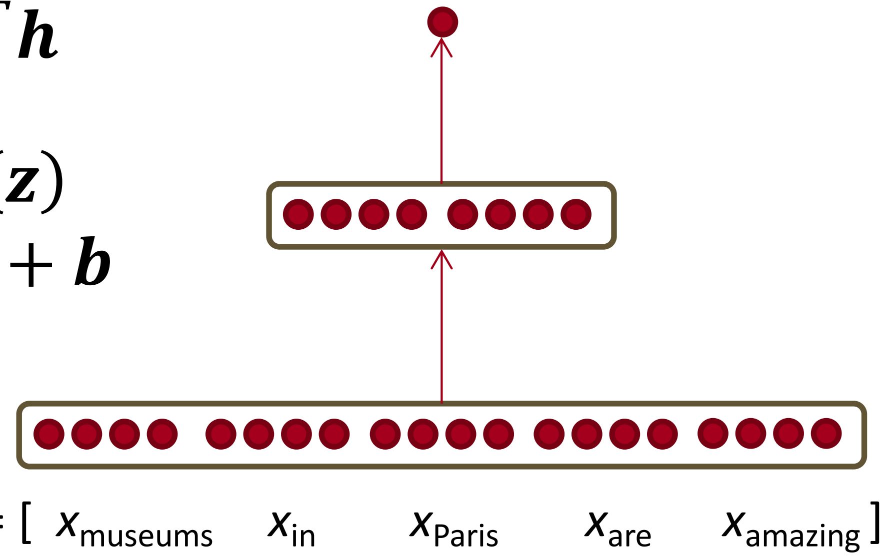
1. Derivative wrt a weight matrix

- Let's look carefully at computing $\frac{\partial s}{\partial \boldsymbol{W}}$
  - Using the chain rule again:

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

$$s = \boldsymbol{u}^T \boldsymbol{h}$$

$$\boldsymbol{h} = f(\boldsymbol{z})$$
$$\boldsymbol{z} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}$$



Deriving gradients for backprop

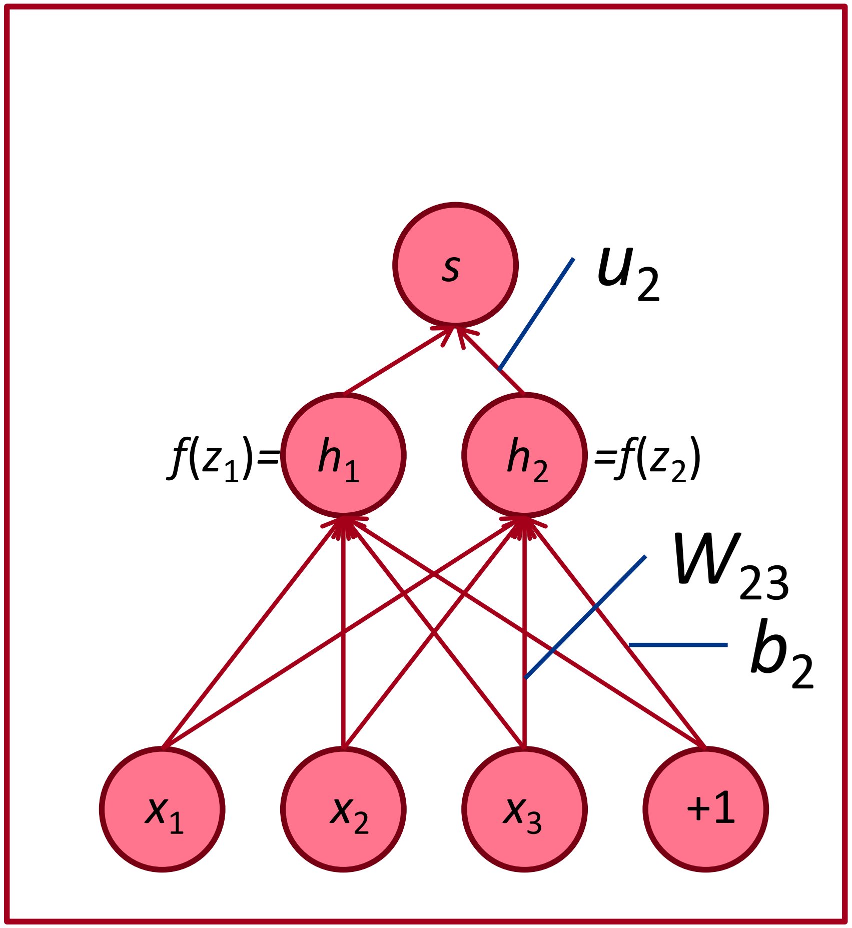
- For this function (following on from last time):

$$\frac{\partial s}{\partial \boldsymbol{W}} = \boldsymbol{\delta}\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{W}} = \boldsymbol{\delta}\frac{\partial}{\partial \boldsymbol{W}}\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}$$

- Let's consider the derivative of a single weight W_{ij}
- W_{ij} only contributes to z_i
  - For example: W_{23} is only used to compute z_2 not z_1

$$\frac{\partial z_i}{\partial W_{ij}} = \frac{\partial}{\partial W_{ij}}\boldsymbol{W}_{i\cdot}\boldsymbol{x} + b_i$$

$$= \frac{\partial}{\partial W_{ij}}\sum_{k=1}^{d} W_{ik}x_k = x_j$$



Deriving gradients for backprop

- So for derivative of single W_{ij} :

$$\frac{\partial s}{\partial W_{ij}} = \underbrace{\delta_i}_{\text{Error signal from above}} \underbrace{x_j}_{\text{Local gradient signal}}$$

- We want gradient for full ***W*** – but each case is the same
- Overall answer: Outer product:

$$\frac{\partial s}{\partial \boldsymbol{W}} = \boldsymbol{\delta}^T \quad \boldsymbol{x}^T$$

$$[n \times m] \quad [n \times 1][1 \times m]$$

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 $\boldsymbol{u} = g(\mathbf{x})$, i.e., $\boldsymbol{y} = f(g(\mathbf{x}))$, 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

Deriving gradients wrt words for window model

- The gradient that arrives at and updates the word vectors can simply be split up for each word vector:
- Let $\nabla_x J = W^T \delta = \delta_{x_{window}}$
- With x_{window} = [$x_{museums}$ x_{in} x_{Paris} x_{are} $x_{amazing}$]
- We have

$$\delta_{window} = \begin{bmatrix} \nabla_{x_{museums}} \\ \nabla_{x_{in}} \\ \nabla_{x_{Paris}} \\ \nabla_{x_{are}} \\ \nabla_{x_{amazing}} \end{bmatrix} \in \mathbb{R}^{5d}$$

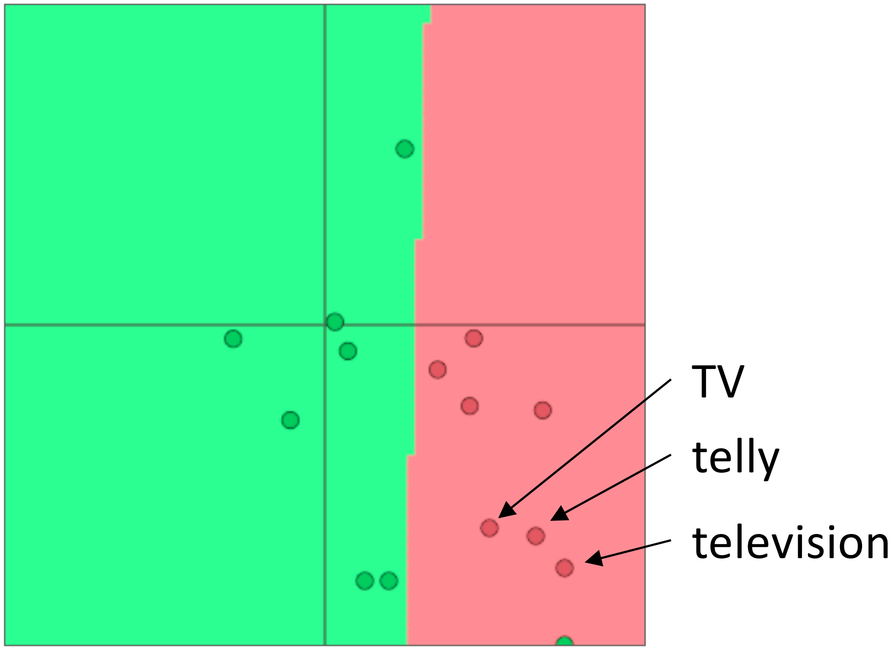
Updating word gradients in window model

- This will push word vectors around so that they will (in principle) be more helpful in determining named entities.

- For example, the model can learn that seeing x_{in} as the word just before the center word is indicative for the center word to be a location

A pitfall when retraining word vectors

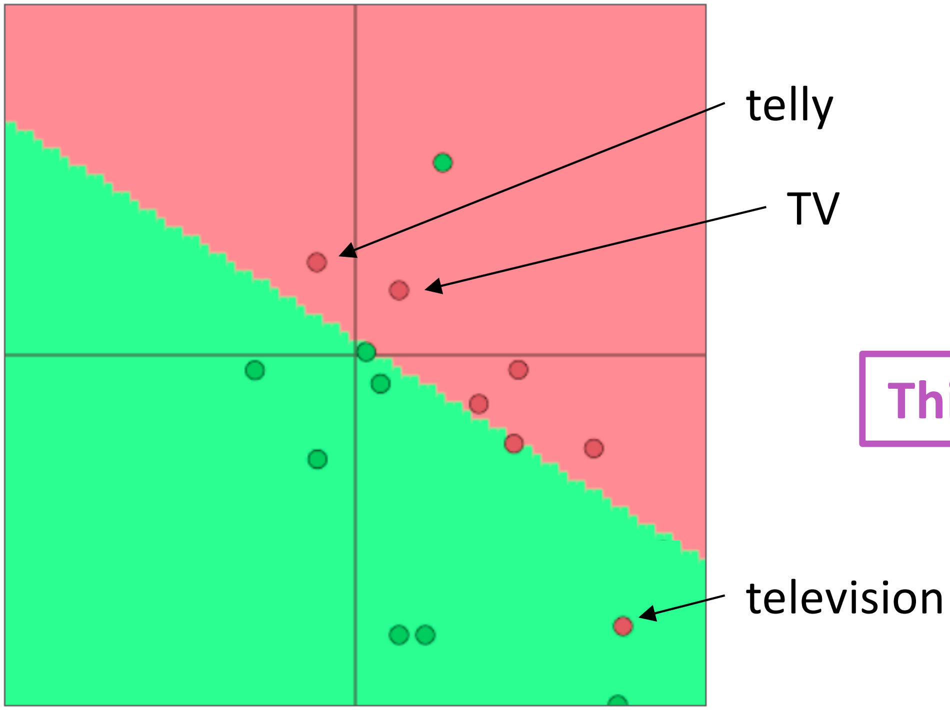
- **Setting:** We are training a logistic regression classification model for movie review sentiment using single words.
- In the training data we have "TV" and "telly"
- In the testing data we have "television"
- The pre-trained word vectors have all three similar:



- **Question: What happens when we update the word vectors?**

A pitfall when retraining word vectors

- **Question:** What happens when we update the word vectors?
- **Answer:**
  - Those words that are in the training data move around
    - "TV" and "telly"
  - Words not in the training data stay where they were
    - "television"



This can be bad!

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

Backpropagation

We’ve almost shown you backpropagation

It’s taking derivatives and using the (generalized) chain rule

Other trick: we **re-use** derivatives computed for higher layers in computing derivatives for lower layers so as to minimize computation

2. Computation Graphs and Backpropagation

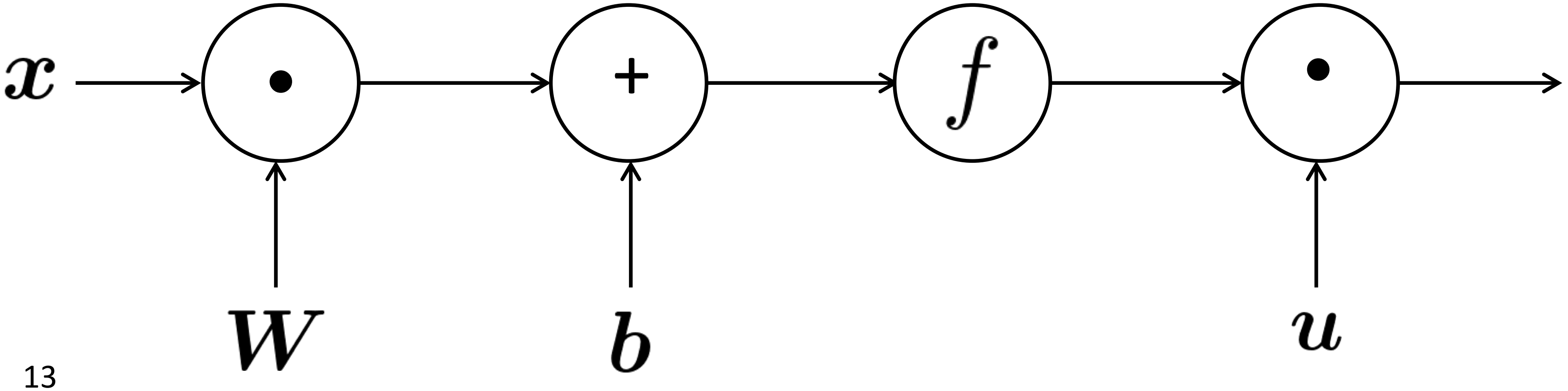
- We represent our neural net equations as a graph
  - Source nodes: inputs
  - Interior nodes: operations

$$s = \boldsymbol{u}^T \boldsymbol{h}$$

$$\boldsymbol{h} = f(\boldsymbol{z})$$

$$\boldsymbol{z} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}$$

$$\boldsymbol{x} \quad \text{(input)}$$



Computation Graphs and Backpropagation

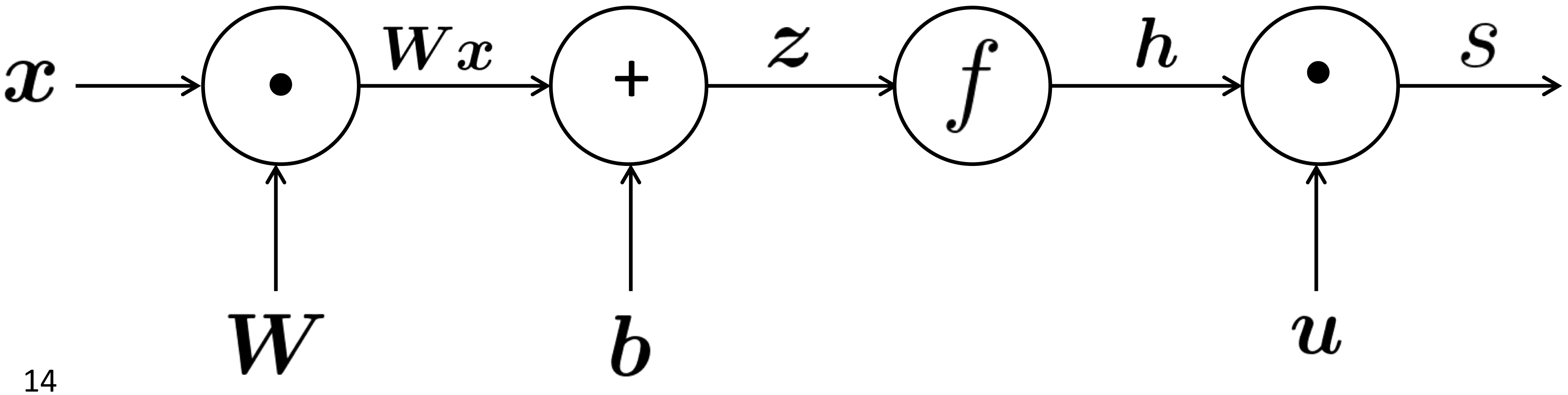
- We represent our neural net equations as a graph
  - Source nodes: inputs
  - Interior nodes: operations
  - Edges pass along result of the operation

$$s = \boldsymbol{u}^T \boldsymbol{h}$$

$$\boldsymbol{h} = f(\boldsymbol{z})$$

$$\boldsymbol{z} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}$$

$$\boldsymbol{x} \quad \text{(input)}$$



Computation Graphs and Backpropagation

- Representing our neural net equations as a graph

$$s = \boldsymbol{u}^T \boldsymbol{h}$$

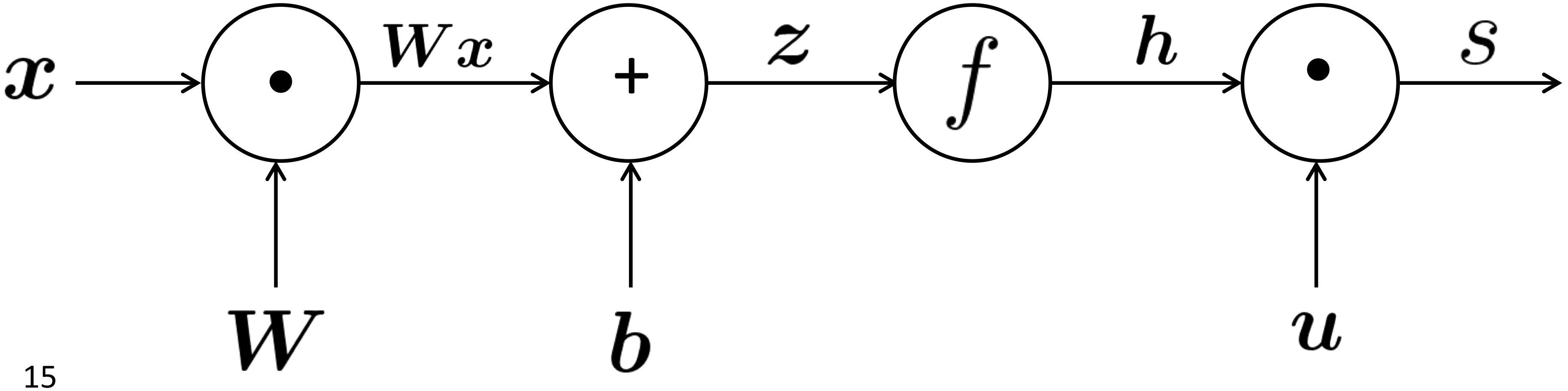
$$\boldsymbol{h} = f(\boldsymbol{z})$$

$+ \boldsymbol{b}$

ut)

“Forward Propagation”

operation



Backpropagation

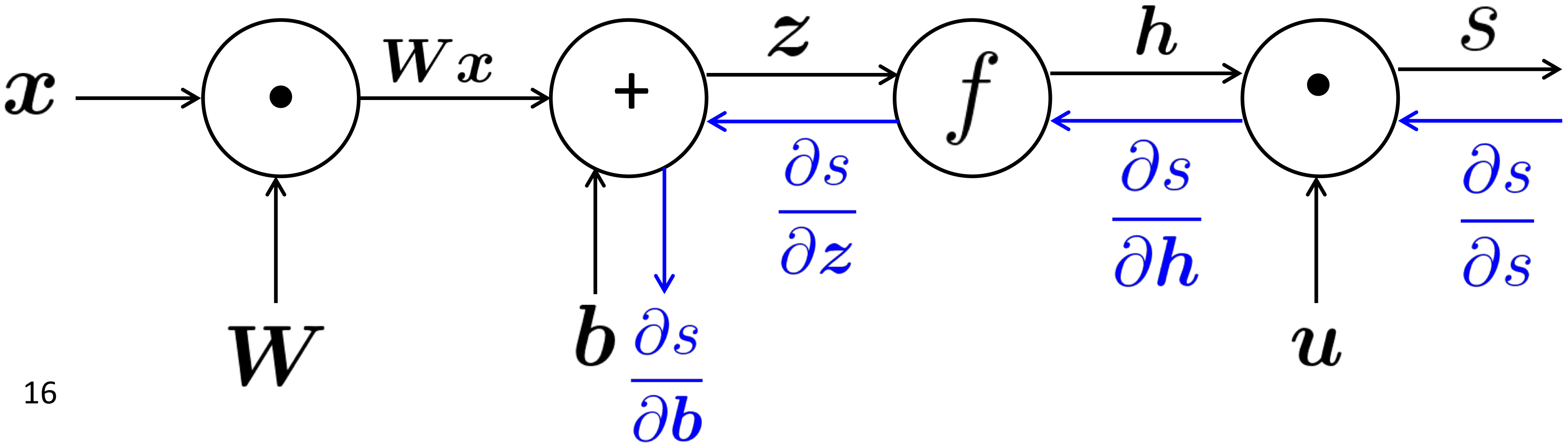
- Go backwards along edges
  - Pass along **gradients**

$$s = \boldsymbol{u}^T \boldsymbol{h}$$

$$\boldsymbol{h} = f(\boldsymbol{z})$$

$$\boldsymbol{z} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}$$

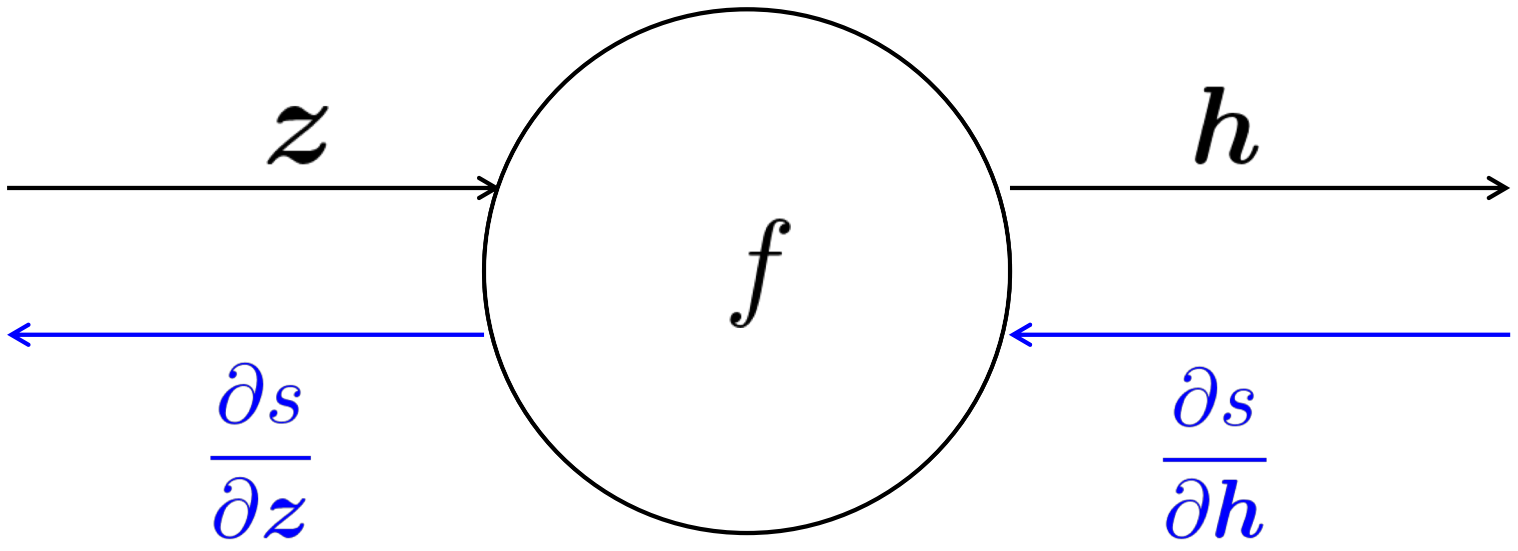
$$\boldsymbol{x} \quad \text{(input)}$$



Backpropagation: Single Node

- Node receives an "upstream gradient"
- Goal is to pass on the correct "downstream gradient"

$$\boxed{\boldsymbol{h} = f(\boldsymbol{z})}$$



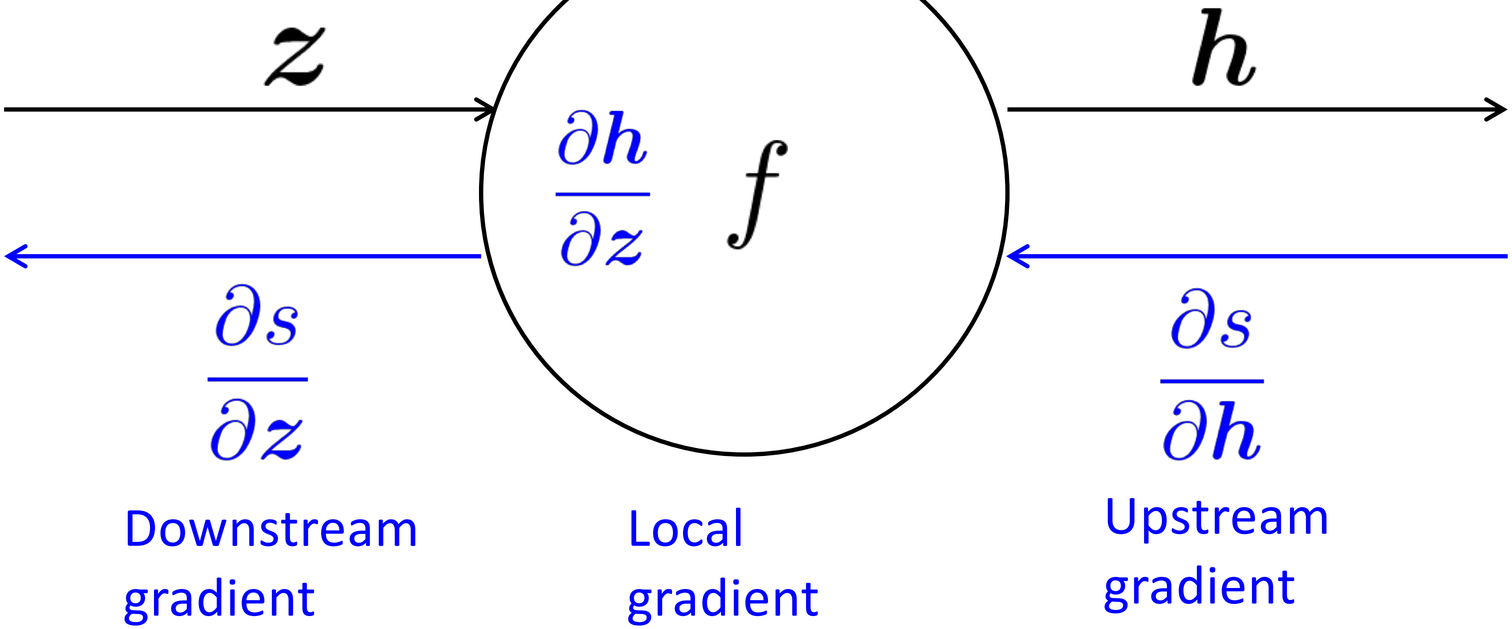
Downstream gradient

Upstream gradient

Backpropagation: Single Node

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

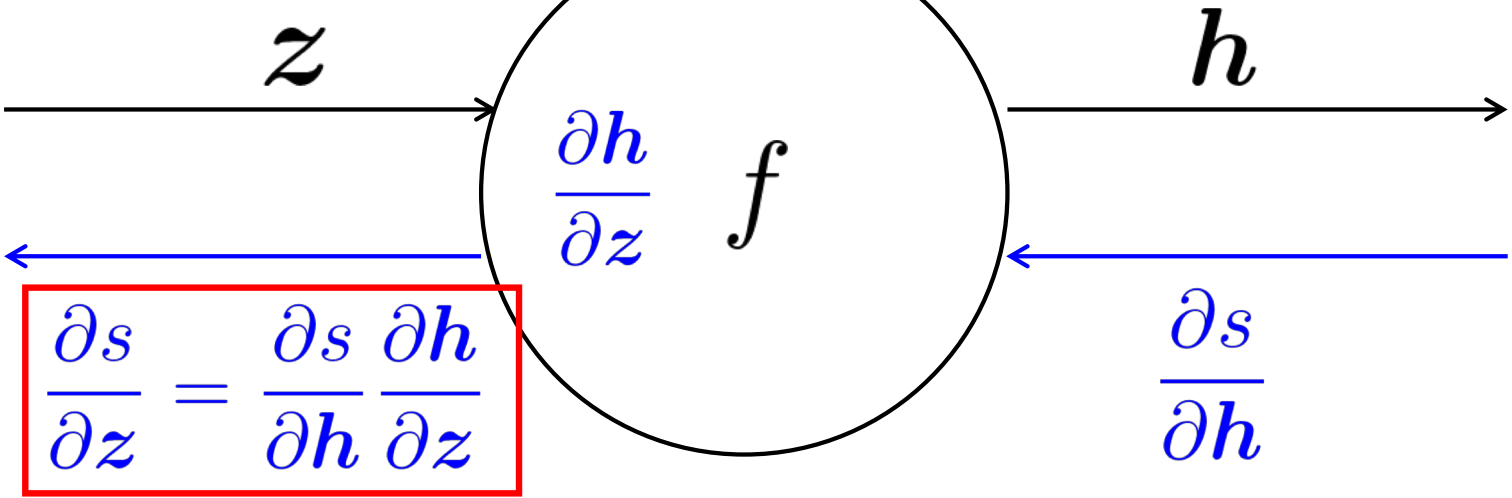
$$\boxed{\boldsymbol{h} = f(\boldsymbol{z})}$$



Backpropagation: Single Node

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

$$\boxed{\boldsymbol{h} = f(\boldsymbol{z})}$$



Chain rule!

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

Downstream gradient

Local gradient

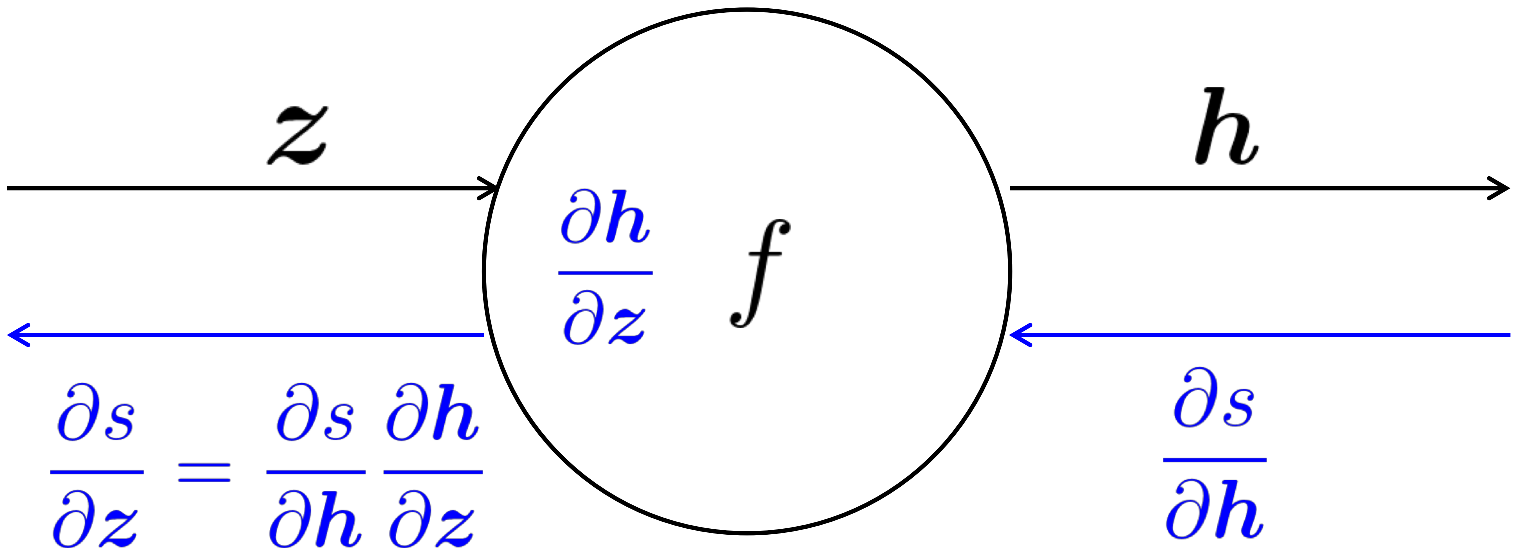
Upstream gradient

Backpropagation: Single Node

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

$$\boxed{\boldsymbol{h} = f(\boldsymbol{z})}$$

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



$\boldsymbol{z}$ $\quad f \quad$ $\boldsymbol{h}$

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

Downstream gradient

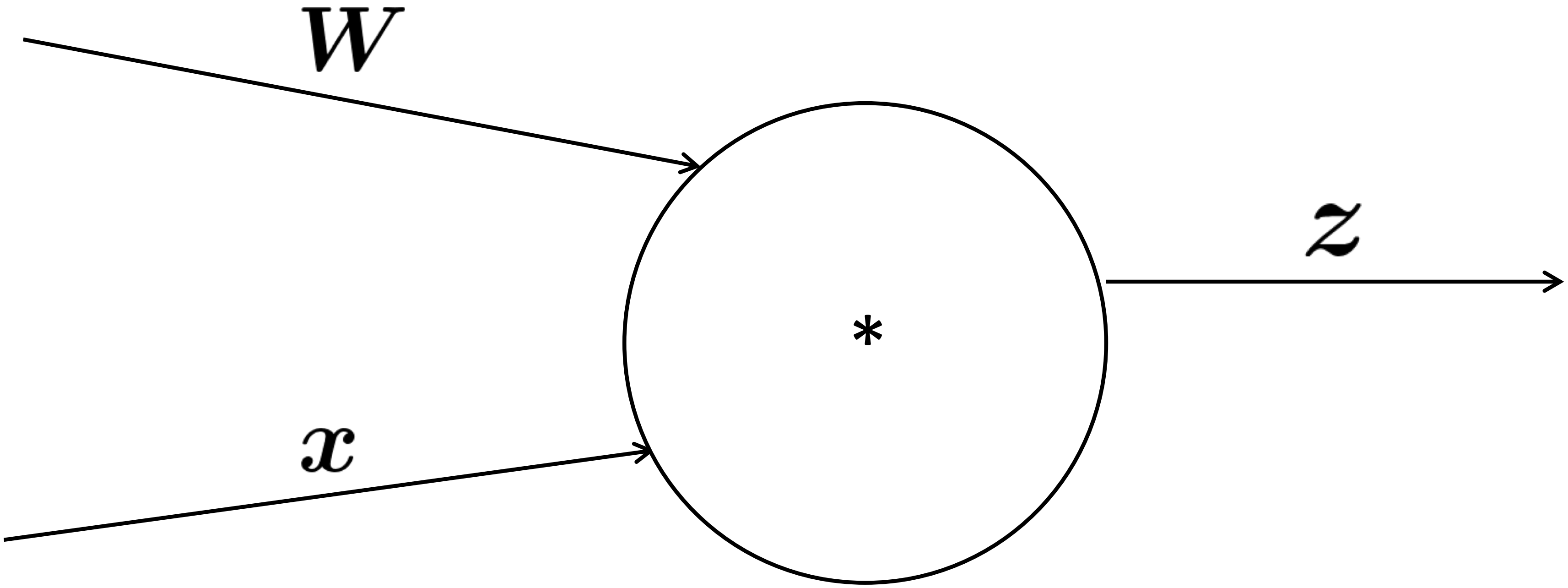
Local gradient

Upstream gradient

Backpropagation: Single Node

- What about nodes with multiple inputs?

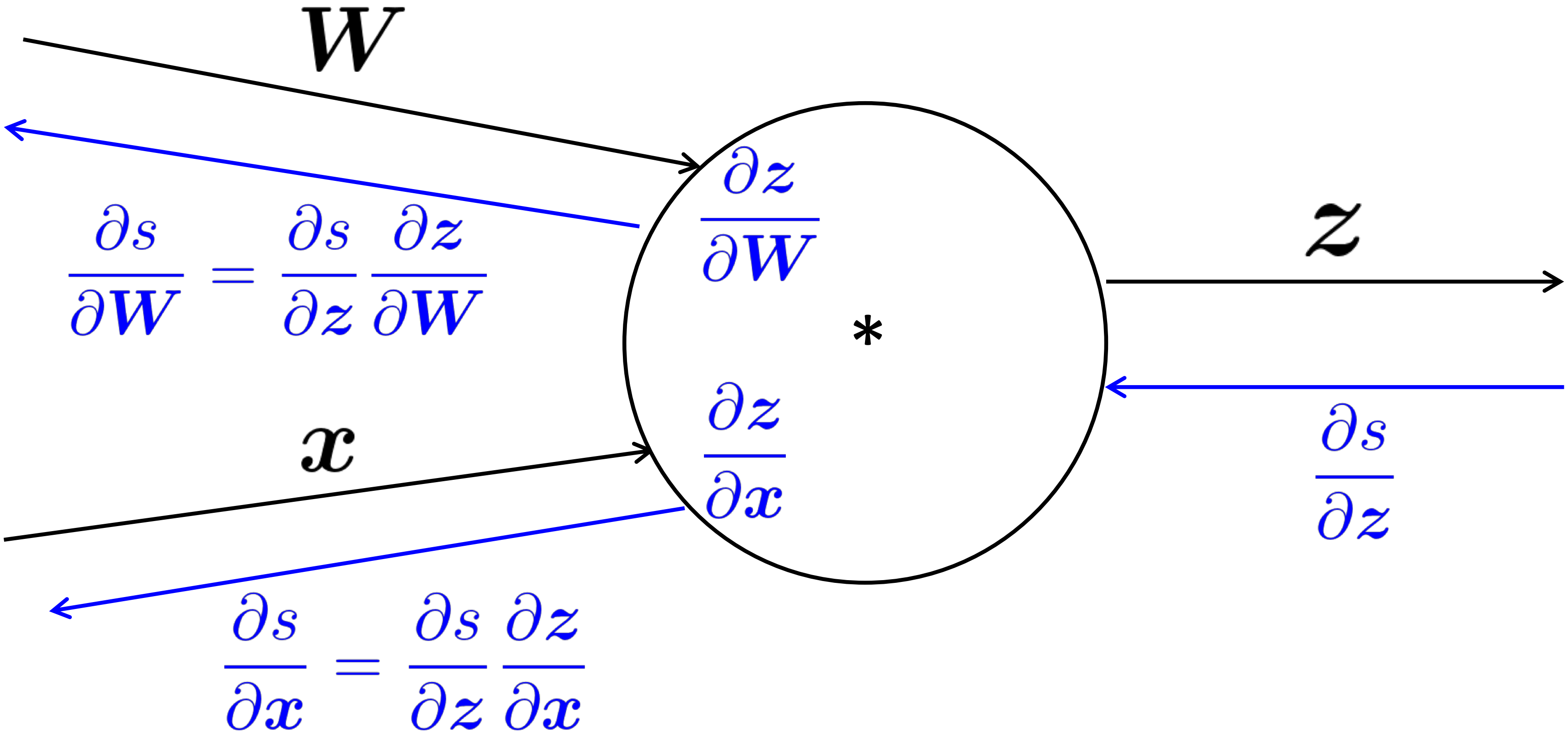
$$\boxed{z = \boldsymbol{W}\boldsymbol{x}}$$



Backpropagation: Single Node

- Multiple inputs → multiple local gradients

$$\boxed{\boldsymbol{z} = \boldsymbol{W}\boldsymbol{x}}$$



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

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

Downstream gradients

Local gradients

Upstream gradient

An Example

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

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

An Example

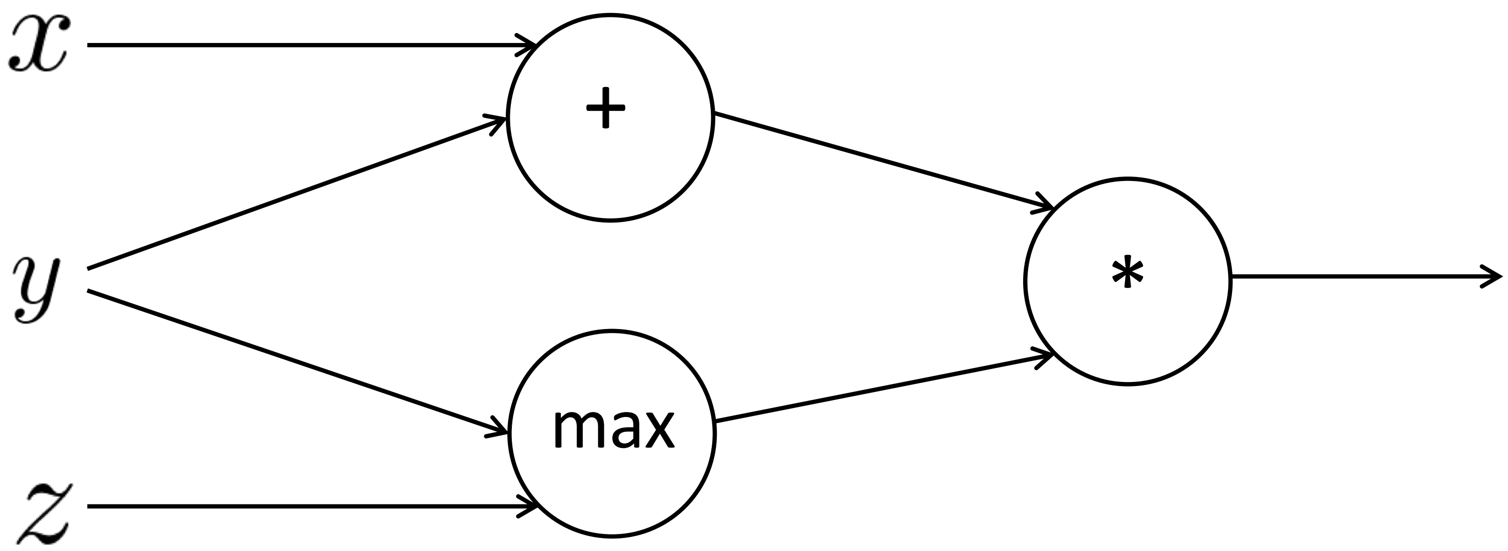
$$f(x, y, z) = (x + y)\max(y, z)$$
$$x = 1, y = 2, z = 0$$

Forward prop steps

$$a = x + y$$

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

$$f = ab$$



An Example

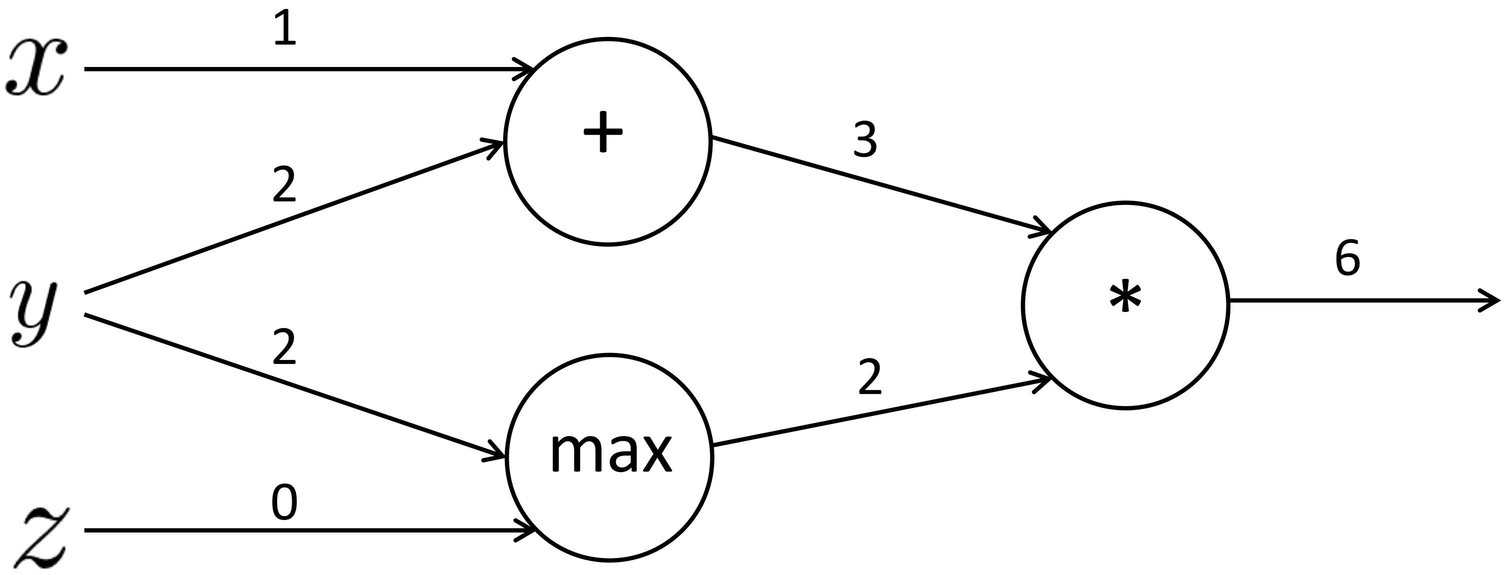
$$f(x,y,z) = (x+y)\max(y,z)$$
$$x=1, y=2, z=0$$

Forward prop steps

$$a = x + y$$

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

$$f = ab$$



An Example

$$f(x, y, z) = (x + y)\max(y, z)$$
$$x = 1, y = 2, z = 0$$

Forward prop steps

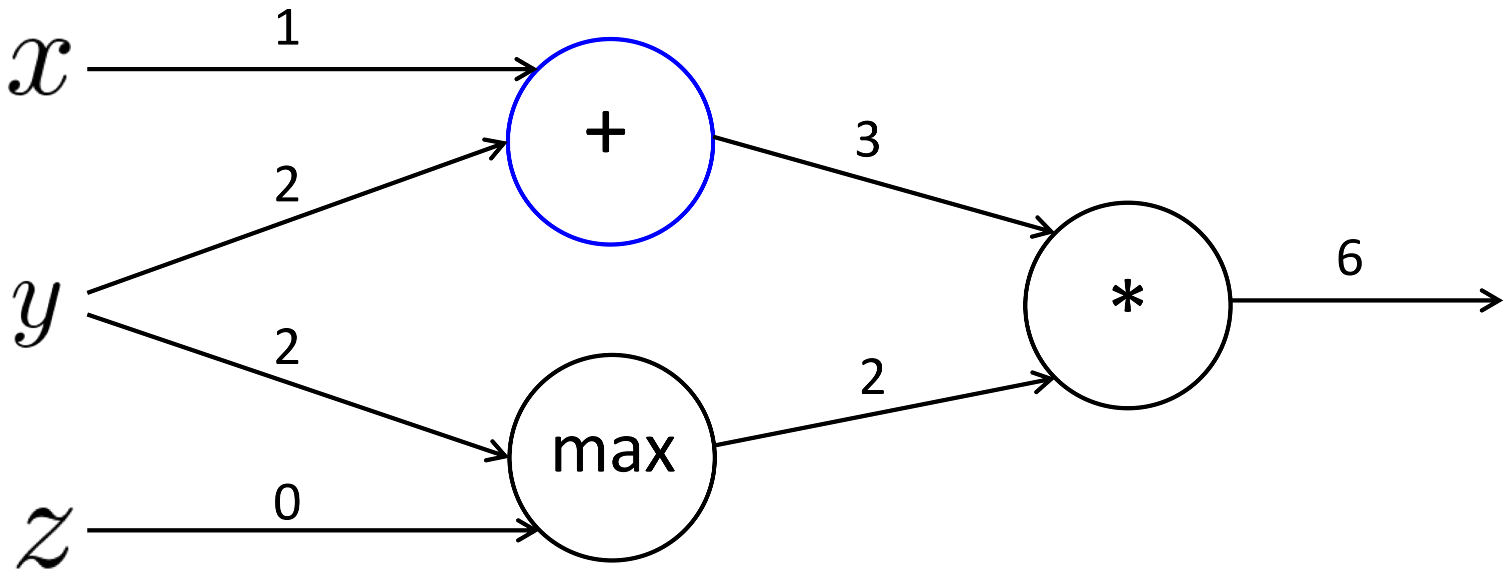
$$a = x + y$$

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

$$f = ab$$

Local gradients

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



An Example

$$f(x, y, z) = (x + y)\max(y, z)$$
$$x = 1, y = 2, z = 0$$

Forward prop steps

$$a = x + y$$

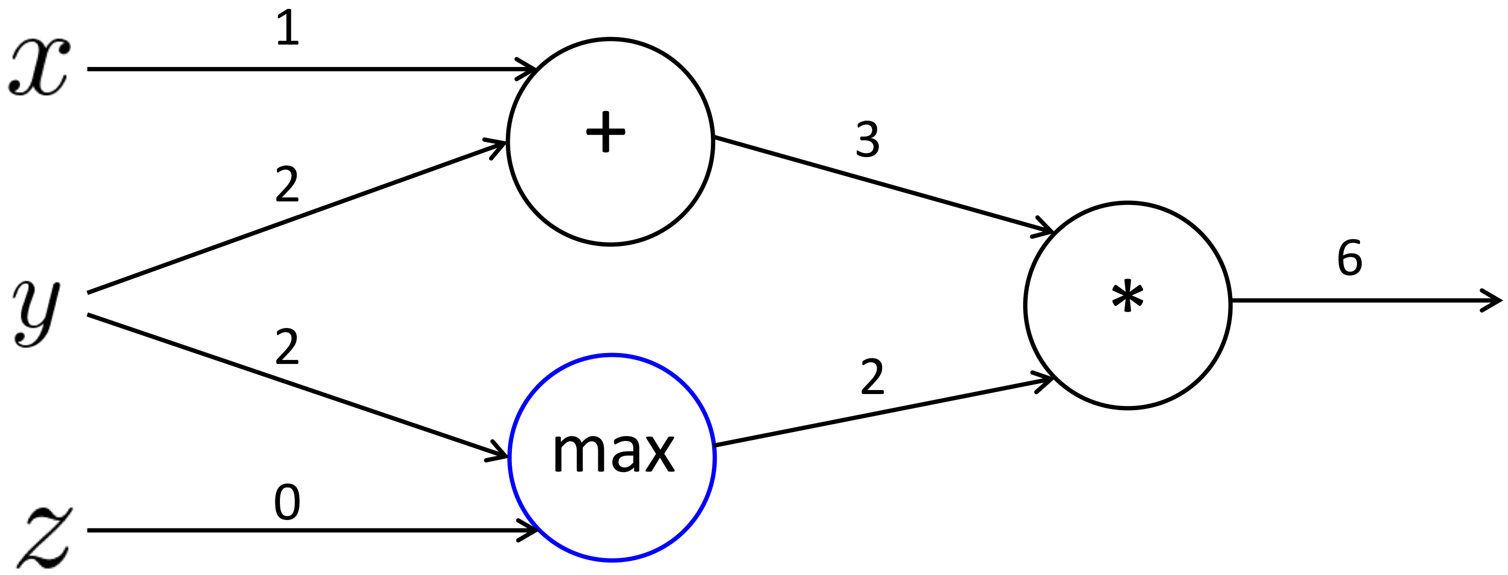
$$b = \max(y, z)$$

$$f = ab$$

Local gradients

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

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



An Example

$$f(x,y,z) = (x+y)\max(y,z)$$
$$x = 1, y = 2, z = 0$$

Forward prop steps

$$a = x + y$$

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

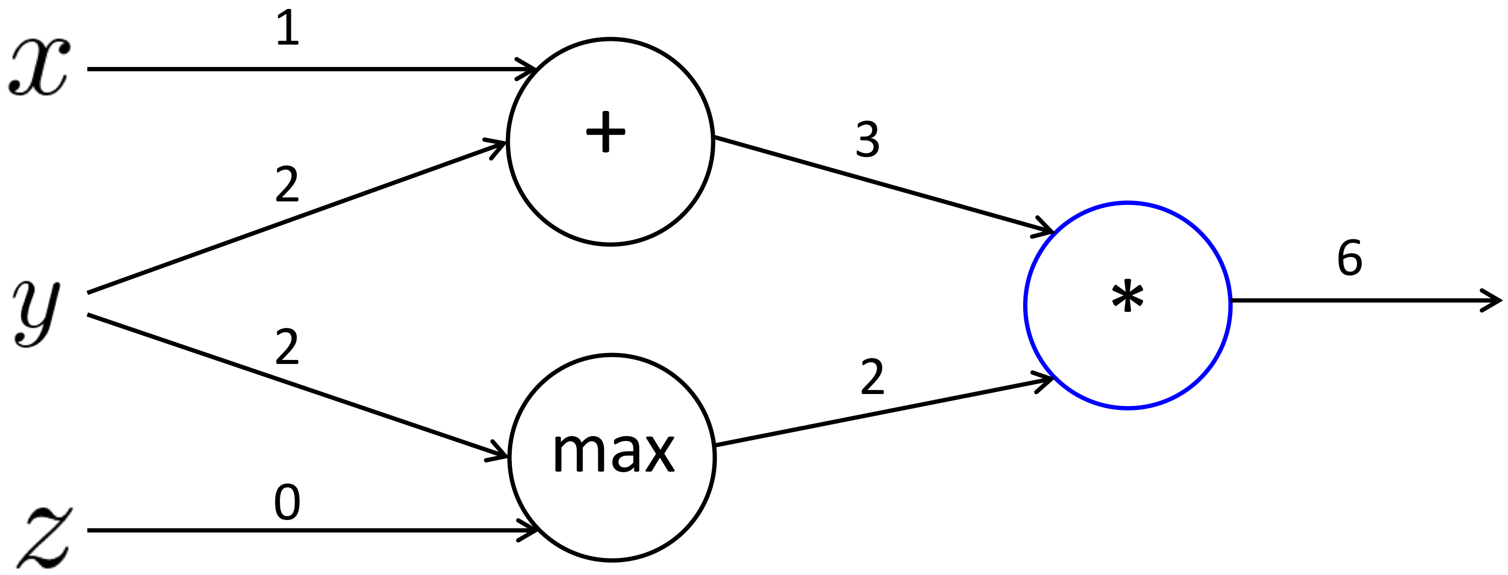
$$f = ab$$

Local gradients

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

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

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



An Example

$$\boxed{\begin{array}{l} f(x, y, z) = (x + y)\max(y, z) \\ x = 1, y = 2, z = 0 \end{array}}$$

Forward prop steps

$$a = x + y$$

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

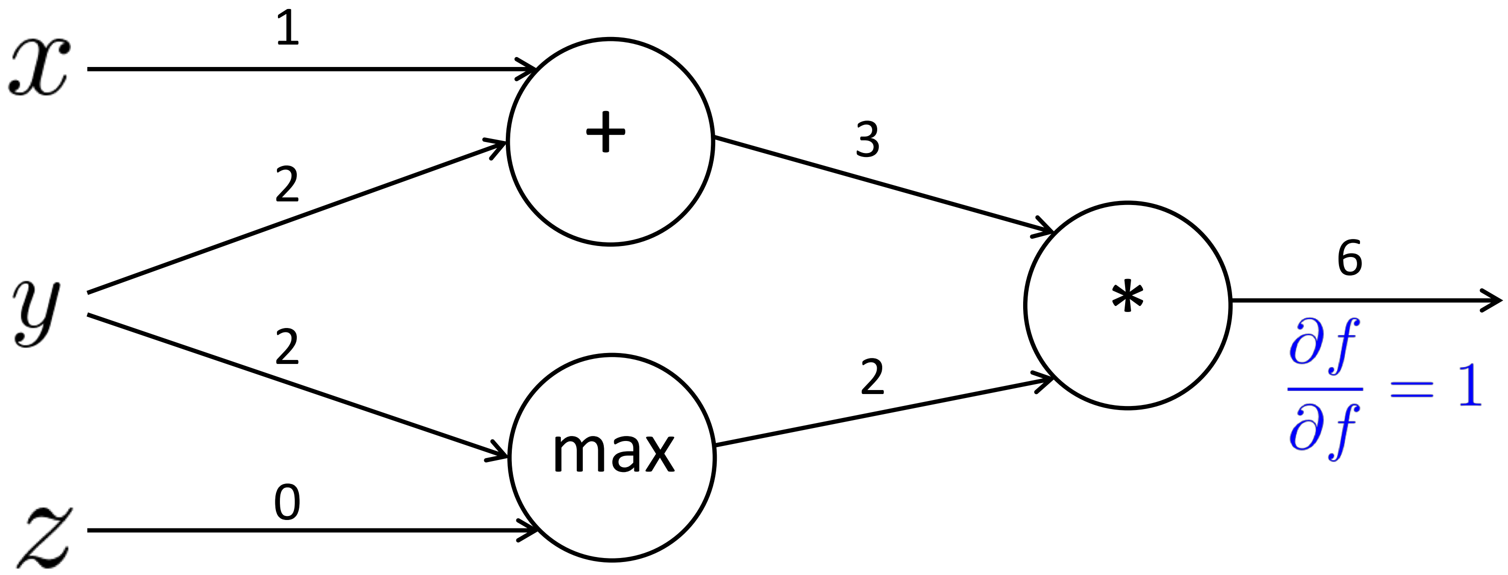
$$f = ab$$

Local gradients

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

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

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



An Example

$$\boxed{\begin{array}{l} f(x,y,z) = (x+y)\max(y,z) \\ x=1, y=2, z=0 \end{array}}$$

Forward prop steps

$$a = x + y$$

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

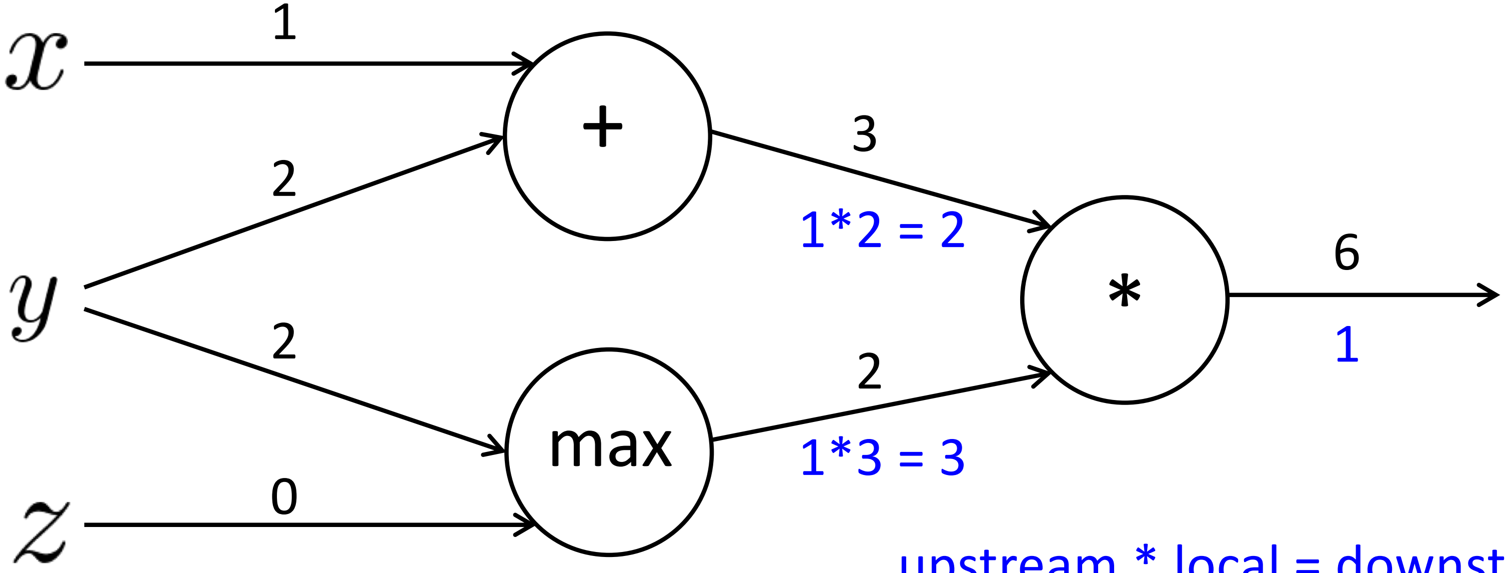
$$f = ab$$

Local gradients

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

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

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



upstream * local = downstream

An Example

$$f(x, y, z) = (x + y)\max(y, z)$$
$$x = 1, y = 2, z = 0$$

Forward prop steps

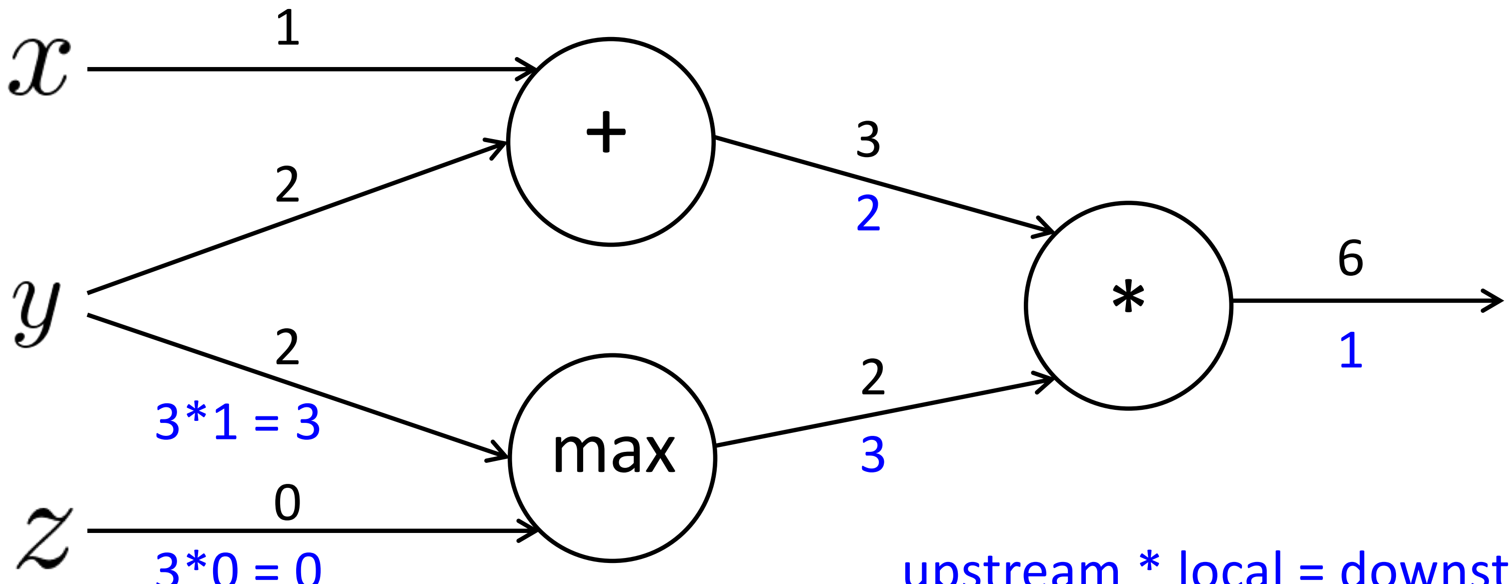
$$a = x + y$$
$$b = \max(y, z)$$
$$f = ab$$

Local gradients

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

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

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



upstream * local = downstream

An Example

$$\boxed{\begin{aligned} &f(x,y,z) = (x+y)\max(y,z) \\ &x = 1, y = 2, z = 0 \end{aligned}}$$

Forward prop steps

$$a = x + y$$

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

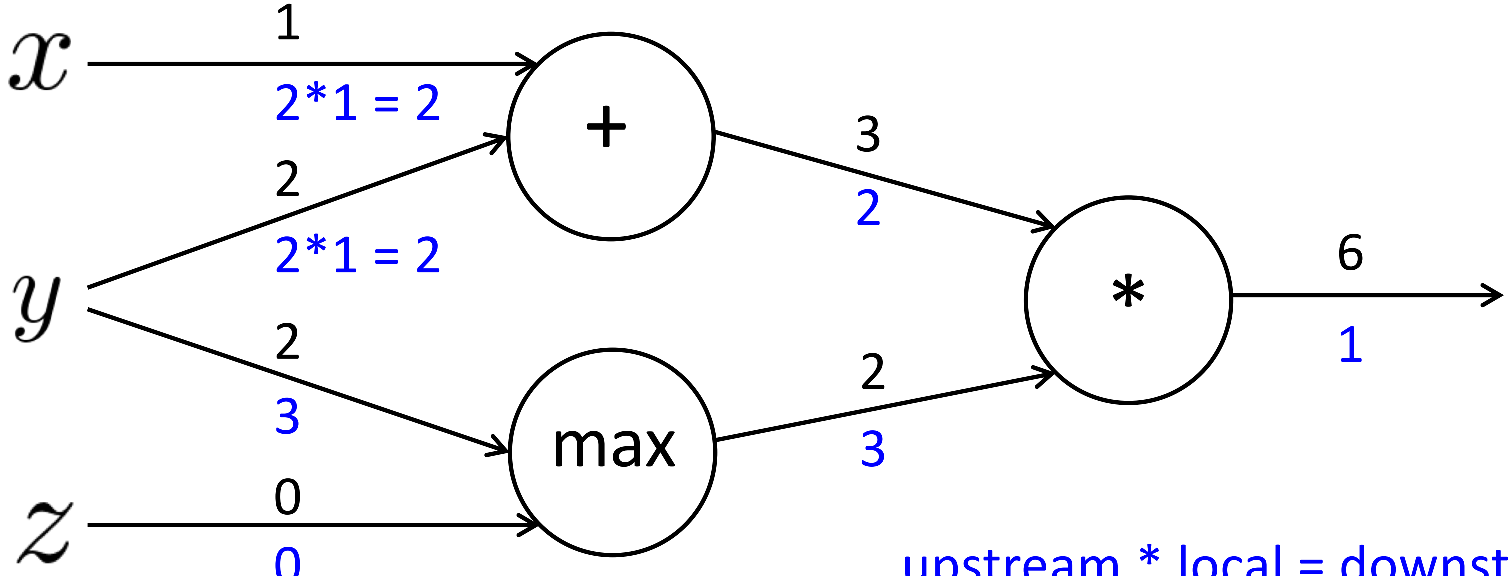
$$f = ab$$

Local gradients

$$\boxed{\frac{\partial a}{\partial x} = 1 \qquad \frac{\partial a}{\partial y} = 1}$$

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

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



upstream * local = downstream

An Example

$$\boxed{\begin{array}{l} f(x, y, z) = (x + y)\max(y, z) \\ x = 1, y = 2, z = 0 \end{array}}$$

Forward prop steps

$$a = x + y$$

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

$$f = ab$$

Local gradients

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

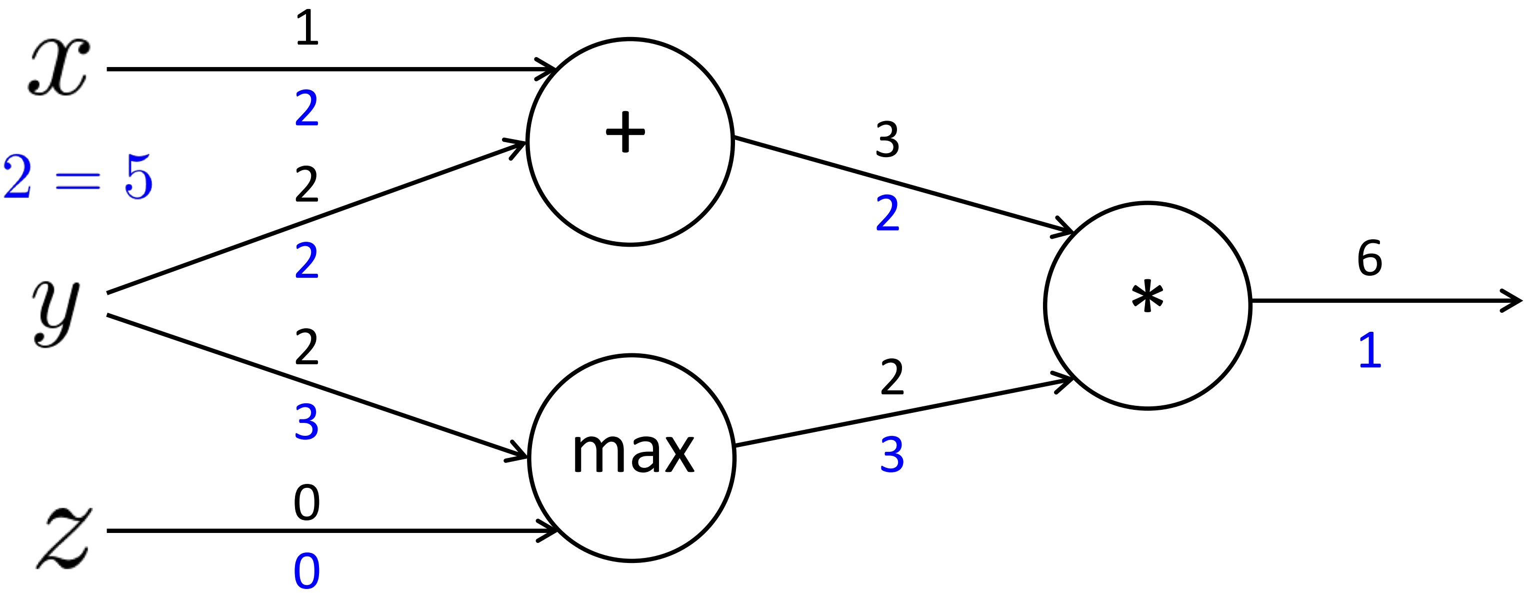
$$\frac{\partial b}{\partial y} = \mathbf{1}(y > z) = 1 \quad \frac{\partial b}{\partial z} = \mathbf{1}(z > y) = 0$$

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

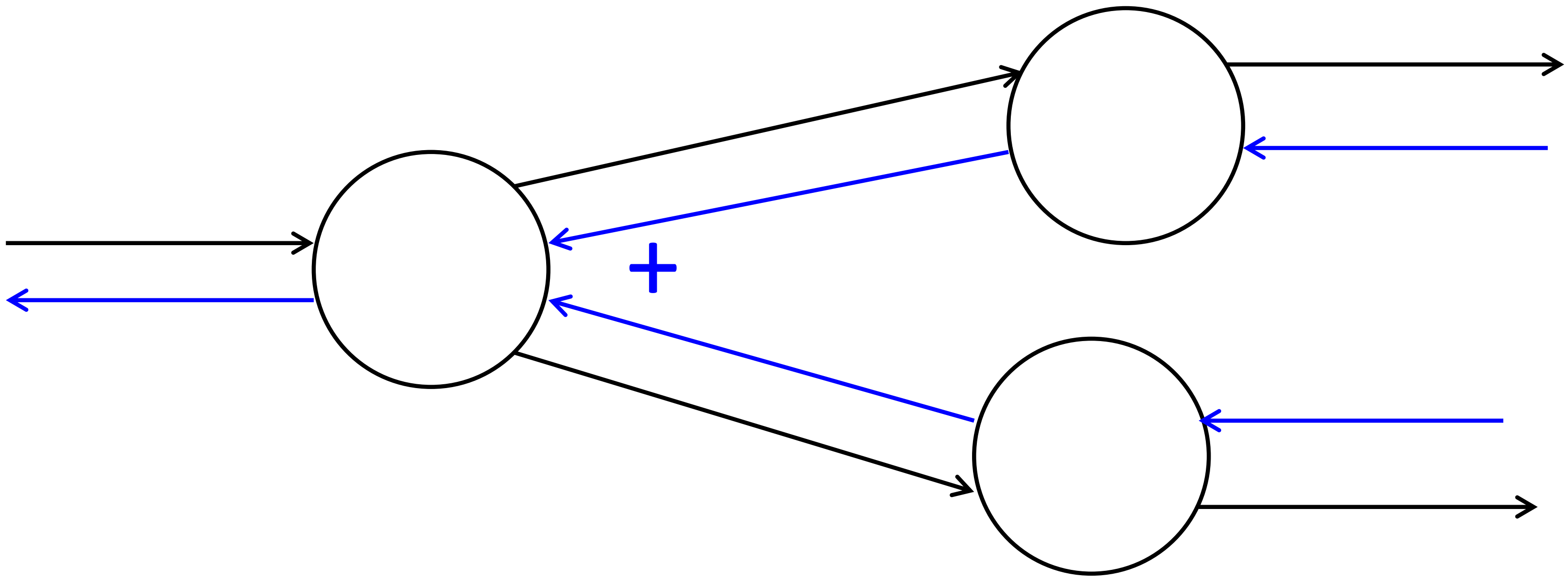
$$\frac{\partial f}{\partial x} = 2$$

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

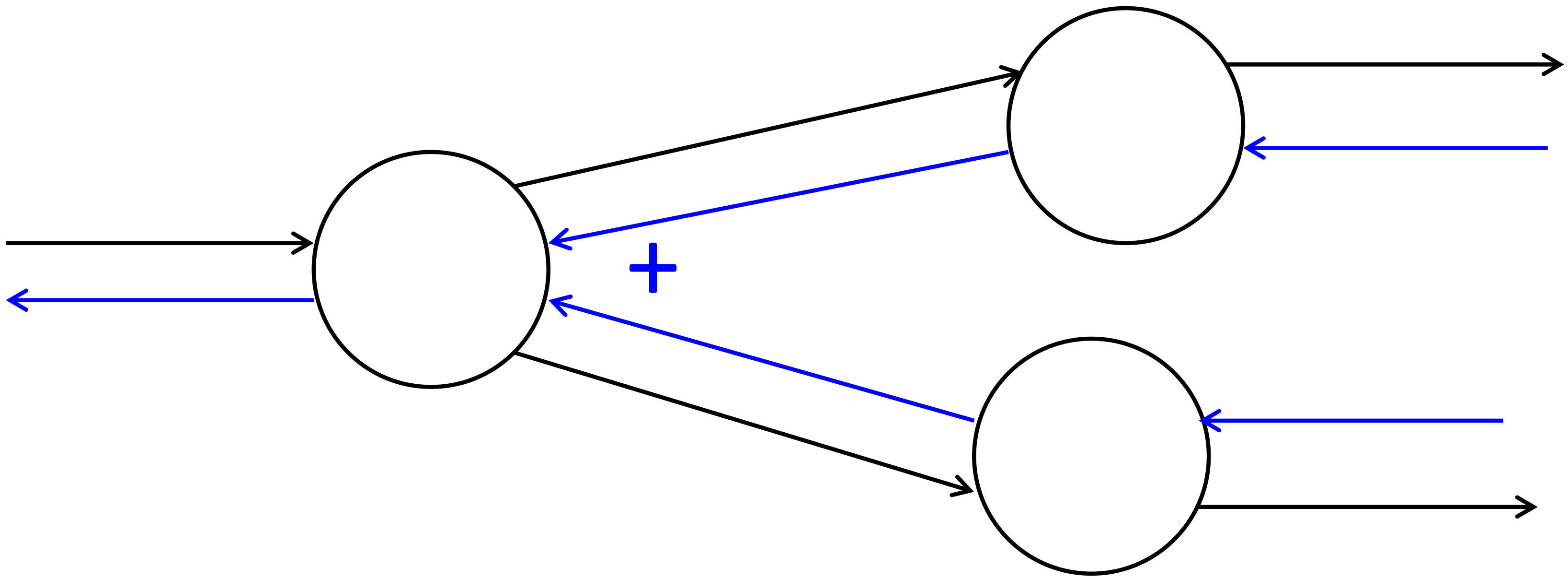
$$\frac{\partial f}{\partial z} = 0$$



Gradients sum at outward branches



Gradients sum at outward branches



$$a = x + y$$

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

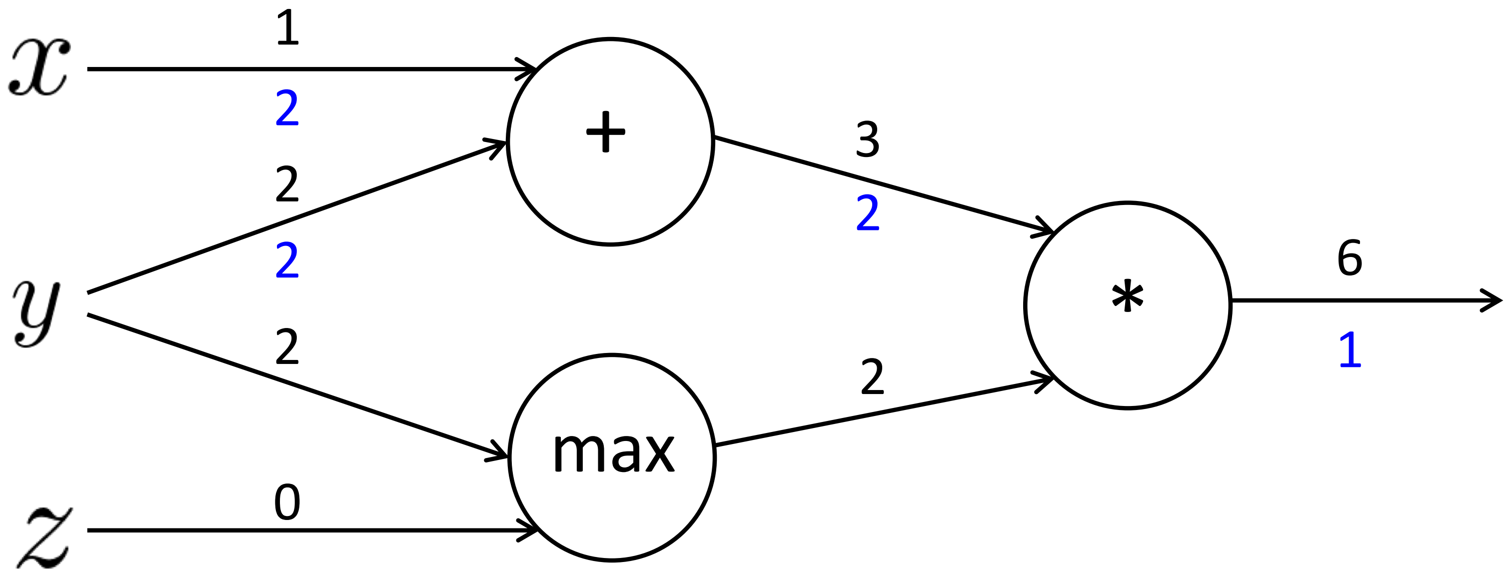
$$f = ab$$

$$\frac{\partial f}{\partial y} = \frac{\partial f}{\partial a}\frac{\partial a}{\partial y} + \frac{\partial f}{\partial b}\frac{\partial b}{\partial y}$$

Node Intuitions

$$f(x, y, z) = (x + y)\max(y, z)$$
$$x = 1, y = 2, z = 0$$

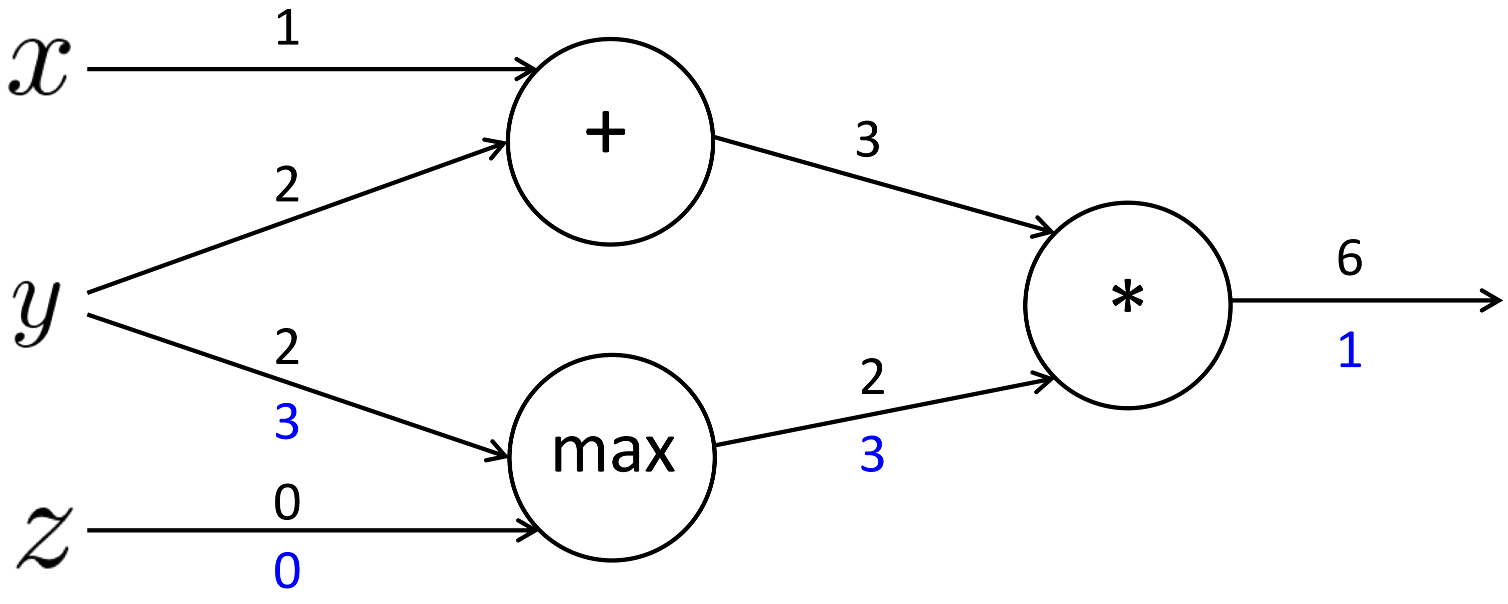
- + “distributes” the upstream gradient



Node Intuitions

$$f(x, y, z) = (x + y)\max(y, z)$$
$$x = 1, y = 2, z = 0$$

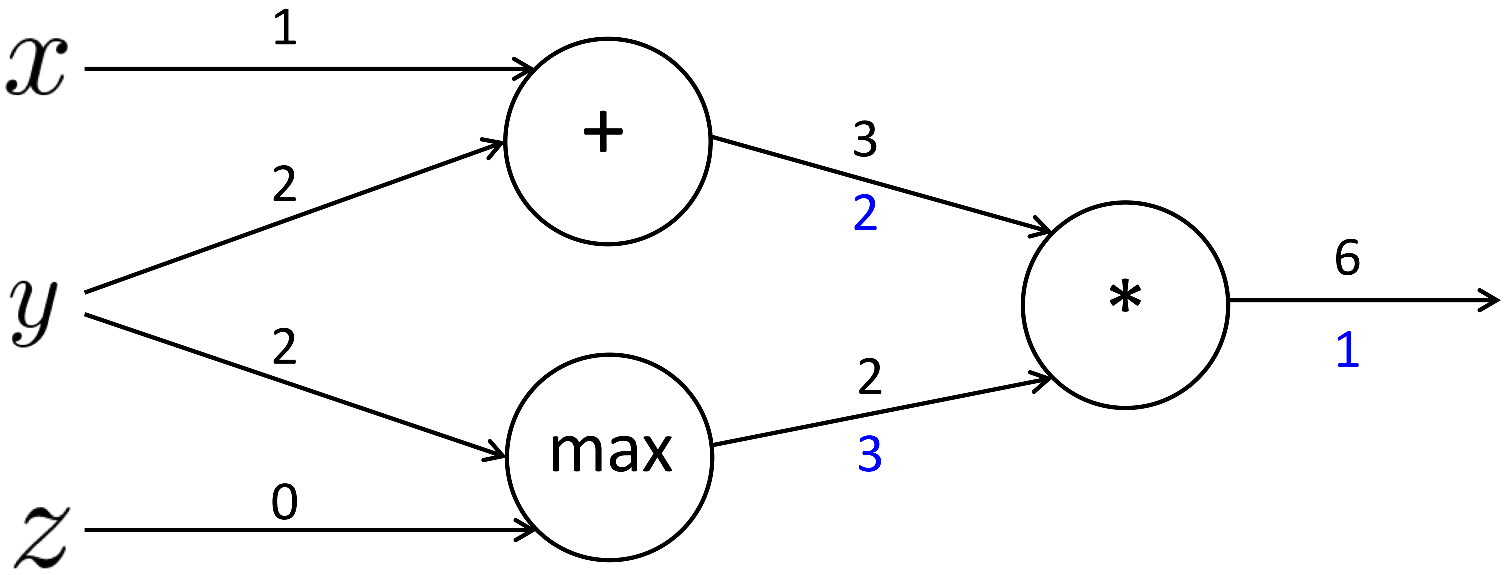
- \+ “distributes” the upstream gradient to each summand
- max “routes” the upstream gradient



Node Intuitions

$$f(x, y, z) = (x + y) \max(y, z)$$
$$x = 1, y = 2, z = 0$$

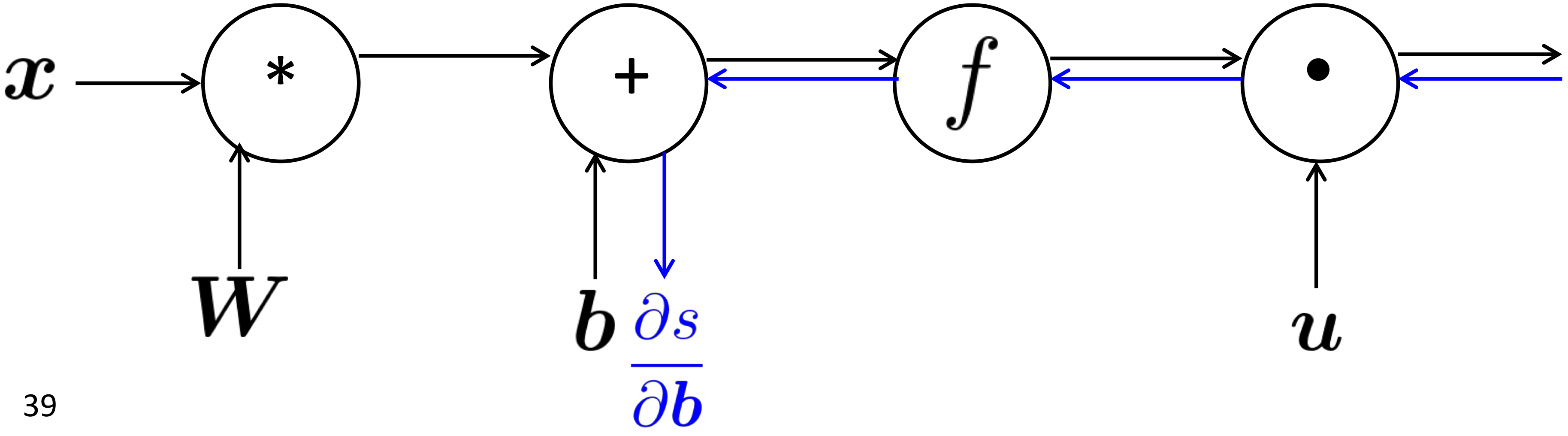
- + “distributes” the upstream gradient
- max “routes” the upstream gradient
- * “switches” the upstream gradient



Efficiency: compute all gradients at once

- Incorrect way of doing backprop:
    - First compute $\frac{\partial s}{\partial \boldsymbol{b}}$

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



Efficiency: compute all gradients at once

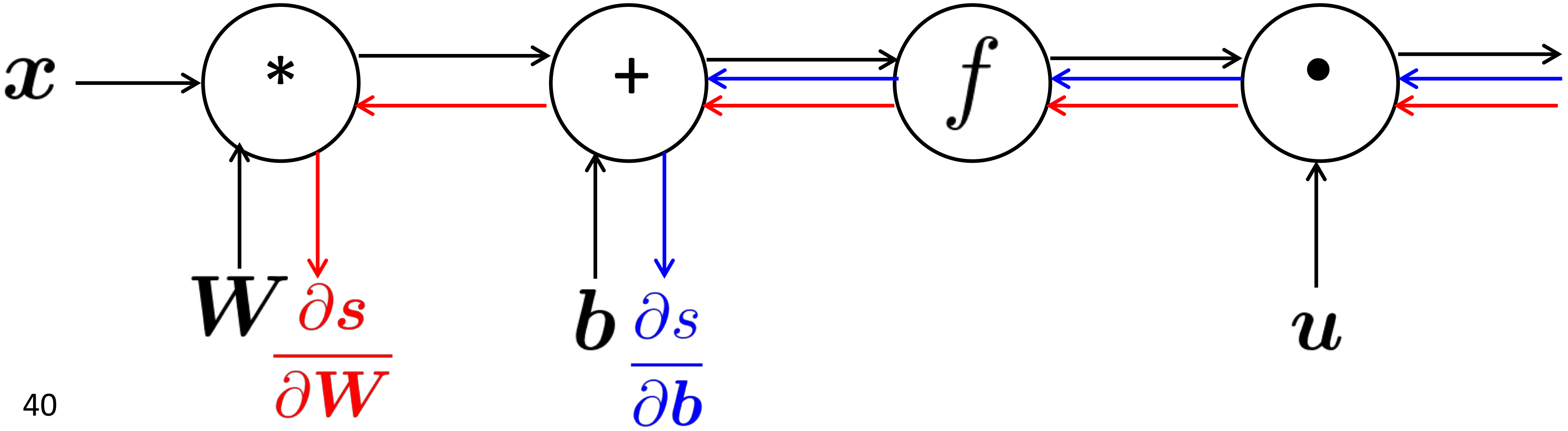
- Incorrect way of doing backprop:
  - First compute $\frac{\partial s}{\partial \boldsymbol{b}}$
  - Then independently compute $\frac{\partial s}{\partial \boldsymbol{W}}$
  - Duplicated computation!

$$s = \boldsymbol{u}^T \boldsymbol{h}$$

$$\boldsymbol{h} = f(\boldsymbol{z})$$

$$\boldsymbol{z} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}$$

$$\boldsymbol{x} \quad \text{(input)}$$



Efficiency: compute all gradients at once

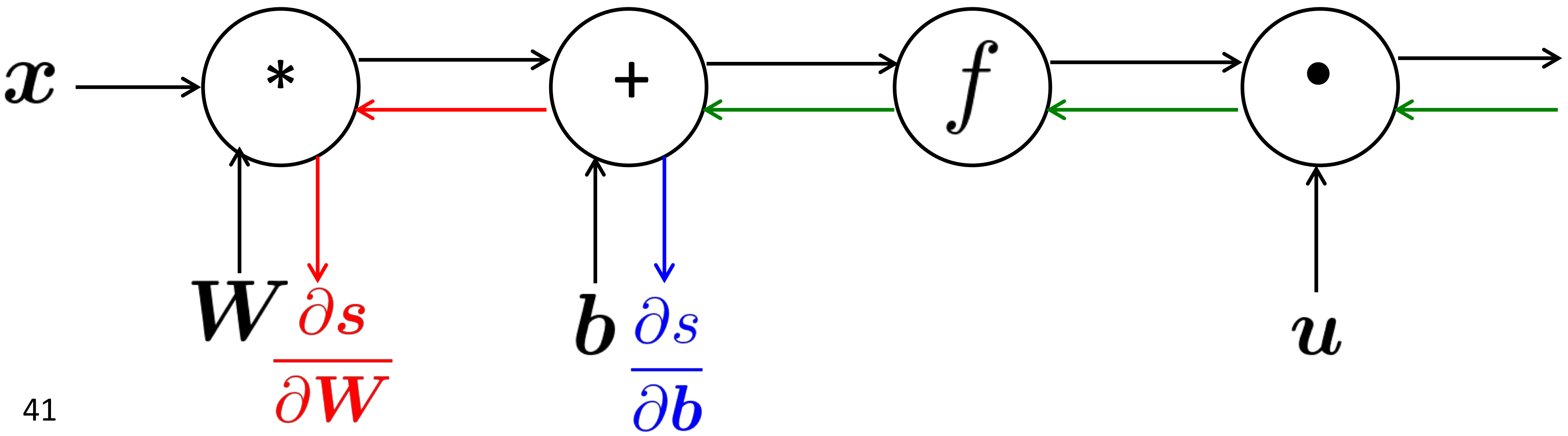
- Correct way:
  - Compute all the gradients at once
  - Analogous to using $\boldsymbol{\delta}$ when we computed gradients by hand

$$s = \boldsymbol{u}^T \boldsymbol{h}$$

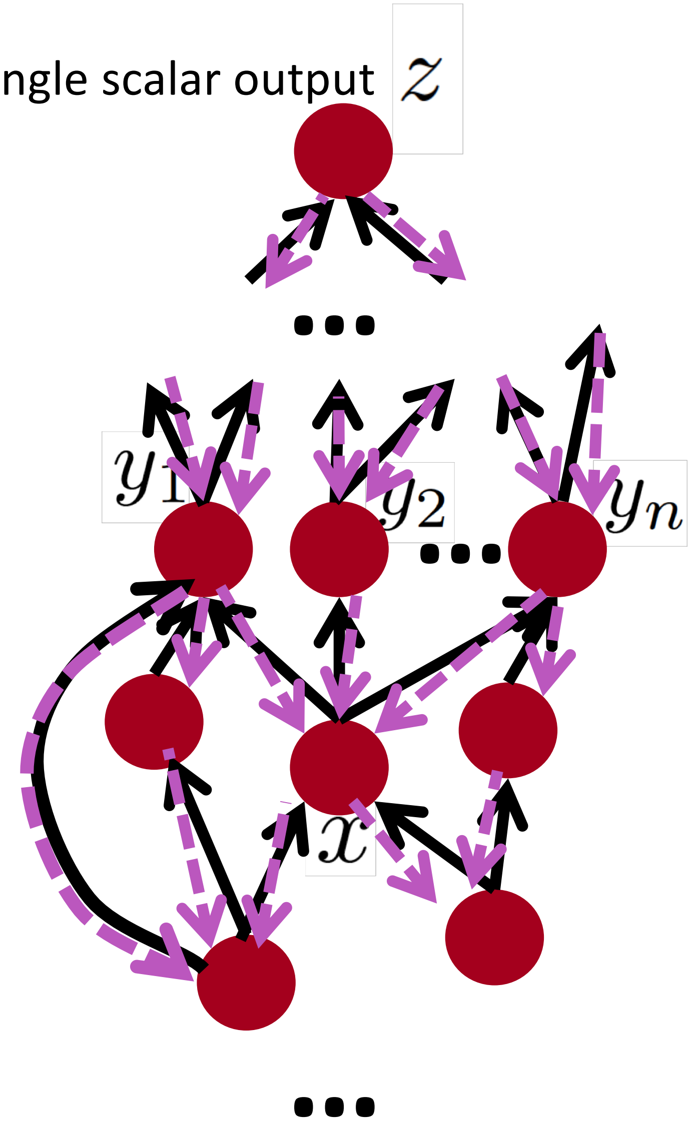
$$\boldsymbol{h} = f(\boldsymbol{z})$$

$$\boldsymbol{z} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}$$

$$\boldsymbol{x} \quad \text{(input)}$$



Back-Prop in General Computation Graph



Single scalar output

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

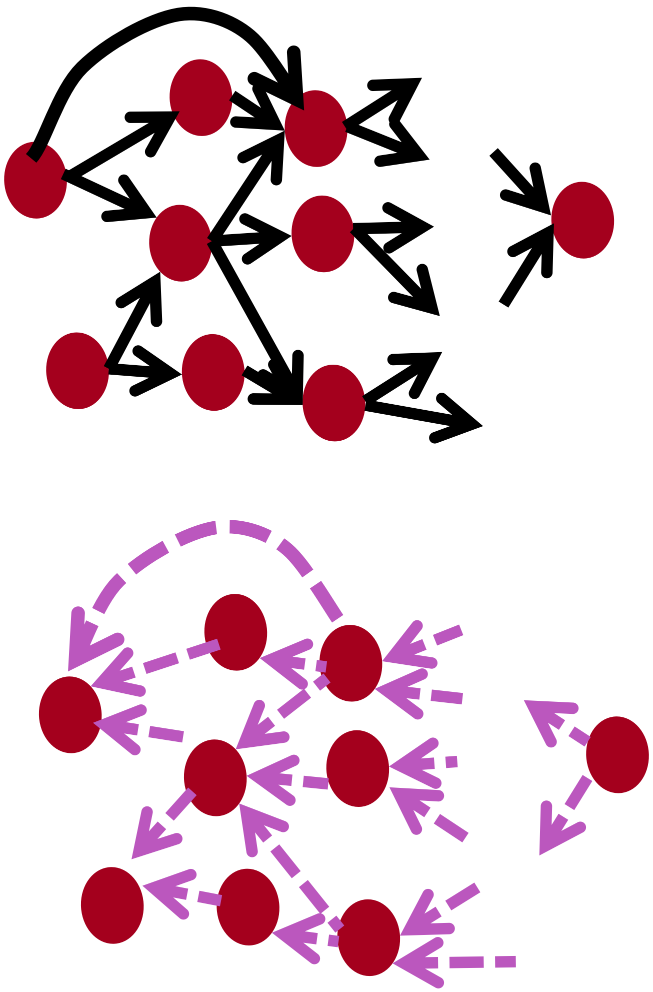
$\{y_1,\, y_2,\, \ldots\, y_n\}$ = successors of x

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

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

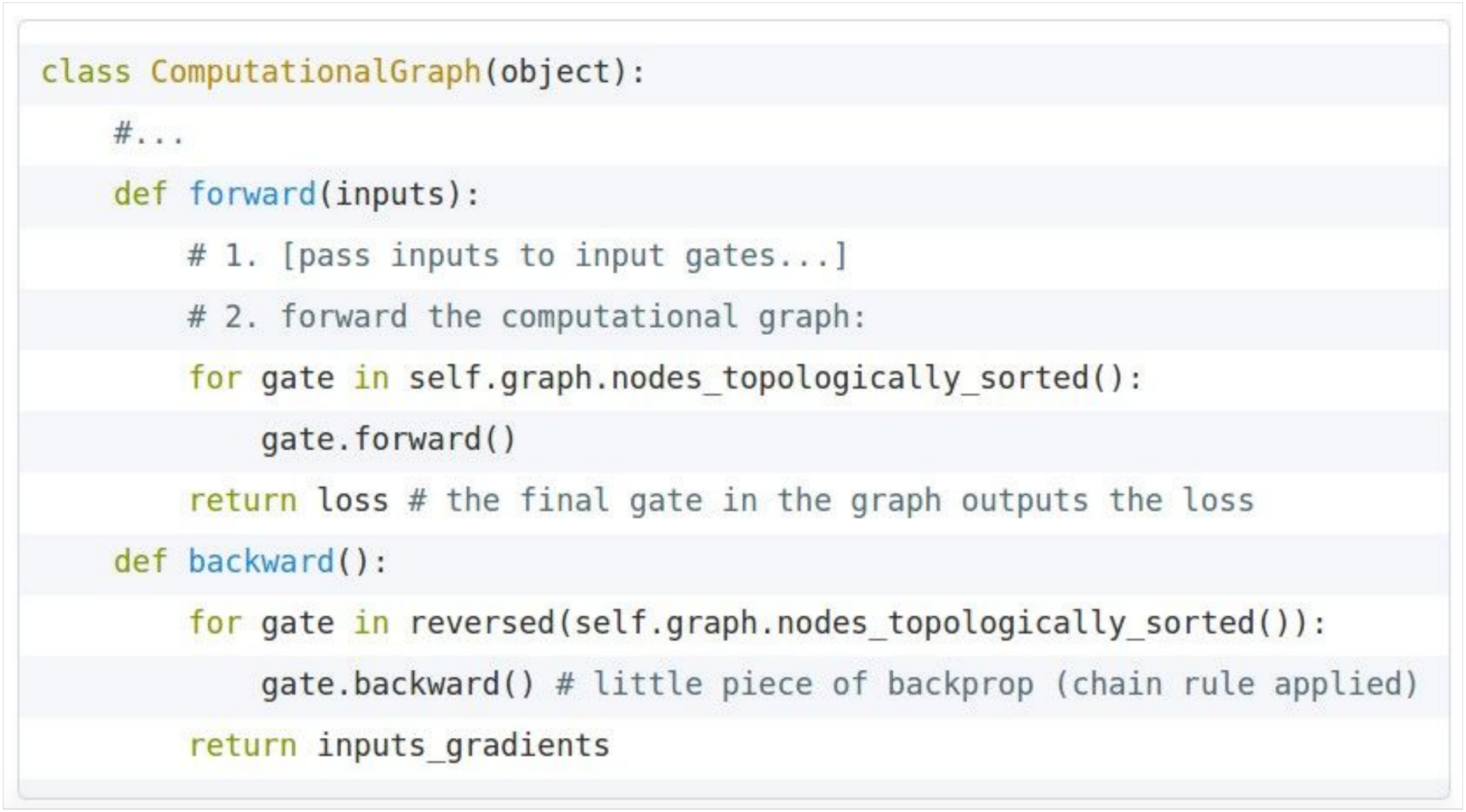
In general our nets have regular layer-structure and so we can use matrices and Jacobians…

Automatic Differentiation



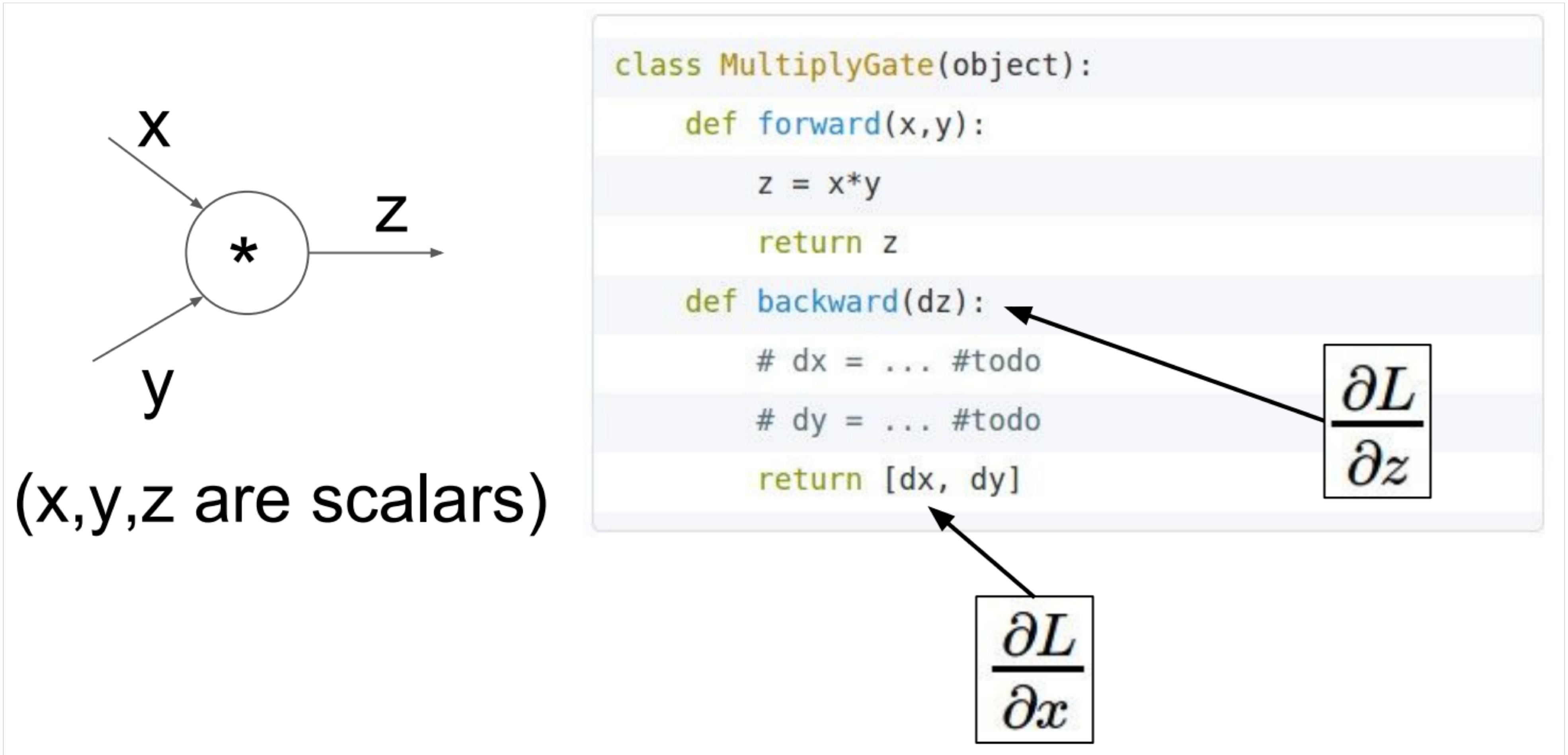
- The gradient computation can be automatically inferred from the symbolic expression of the fprop
- Each node type needs to know how to compute its output and how to compute the gradient wrt its inputs given the gradient wrt its output
- Modern DL frameworks (Tensorflow, PyTorch, etc.) do backpropagation for you but mainly leave layer/node writer to hand-calculate the local derivative

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



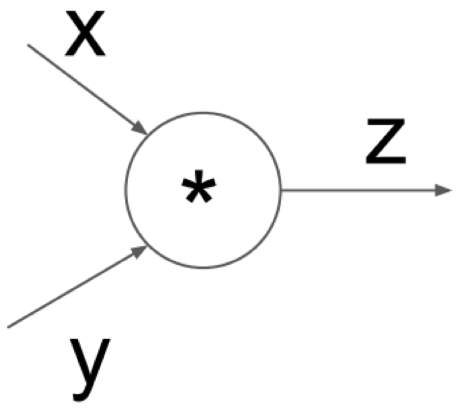
(x,y,z are scalars)

```
class MultiplyGate(object):
    def forward(x,y):
        z = x*y
        return z
    def backward(dz):
        # dx = ... #todo
        # dy = ... #todo
        return [dx, dy]
```

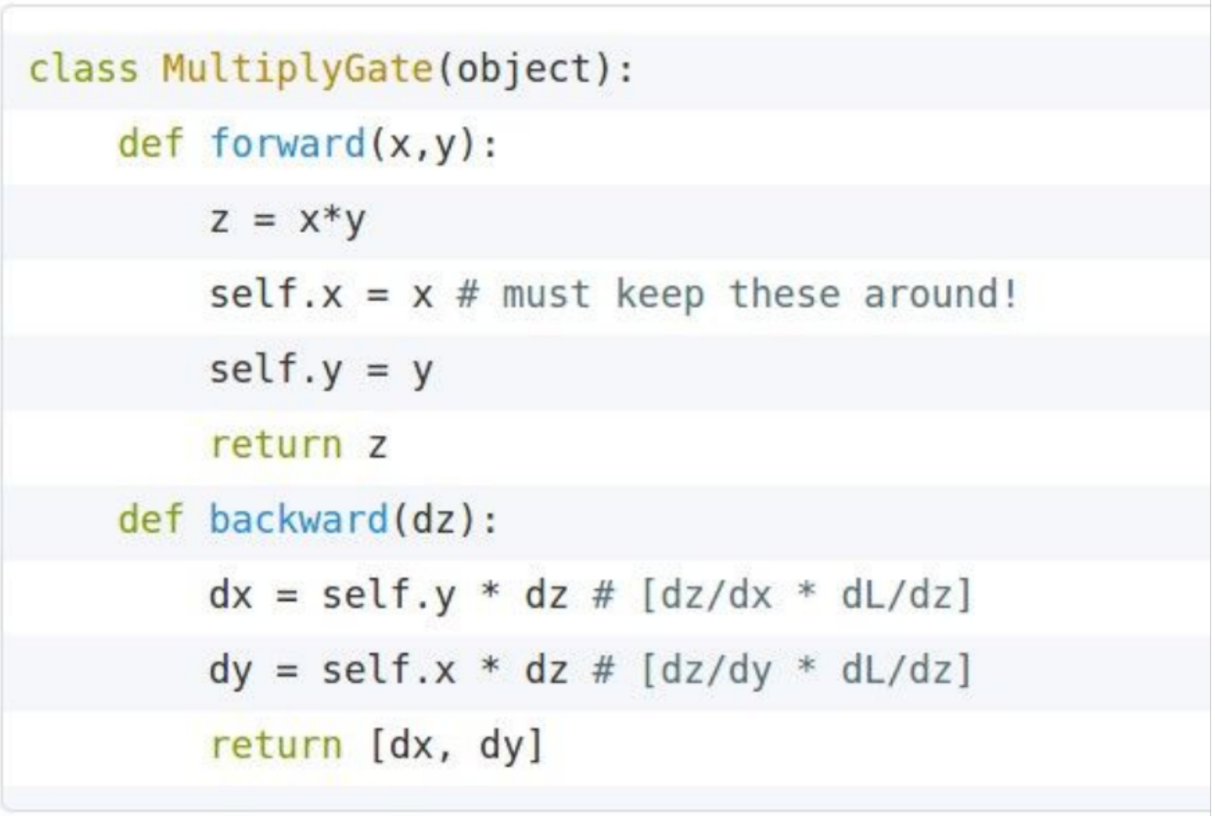
$$\boxed{\frac{\partial L}{\partial z}}$$

$$\boxed{\frac{\partial L}{\partial x}}$$

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]
```

Gradient checking: Numeric Gradient

- For small *h* (≈ 1e-4), $f'(x) \approx \dfrac{f(x+h) - f(x-h)}{2h}$
- Easy to implement correctly
- But approximate and **very** slow:
  - Have to recompute *f* for **every parameter** of our model
- Useful for checking your implementation
  - In the old days when we hand-wrote everything, it was key to do this everywhere.
  - Now much less needed, when throwing together layers

Summary

- We’ve mastered the core technology of neural nets!!!

- Backpropagation: recursively apply the chain rule along computation graph
  - [downstream gradient] = [upstream gradient] x [local gradient]
- Forward pass: compute results of operations and save intermediate values
- Backward pass: apply chain rule to compute gradients

Why learn all these details about gradients?

- Modern deep learning frameworks compute gradients for you

- But why take a class on compilers or systems when they are implemented for you?
  - Understanding what is going on under the hood is useful!

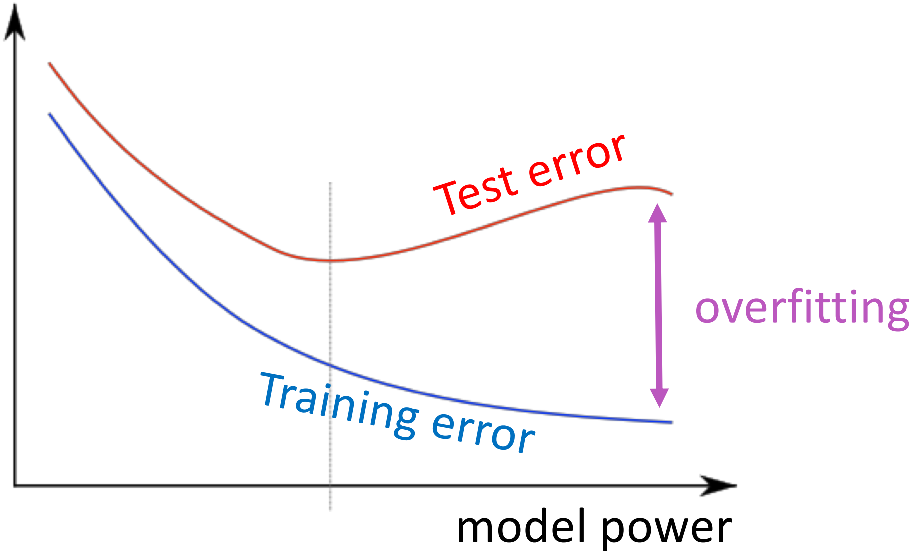
- Backpropagation doesn’t always work perfectly.
  - Understanding why is crucial for debugging and improving models
  - See Karpathy article (in syllabus):
    - https://medium.com/@karpathy/yes-you-should-understand-backprop-e2f06eab496b
  - Example in future lecture: exploding and vanishing gradients

3. We have models with many params! Regularization!

- Really a full loss function in practice includes **regularization** over all parameters θ, e.g., L2 regularization:

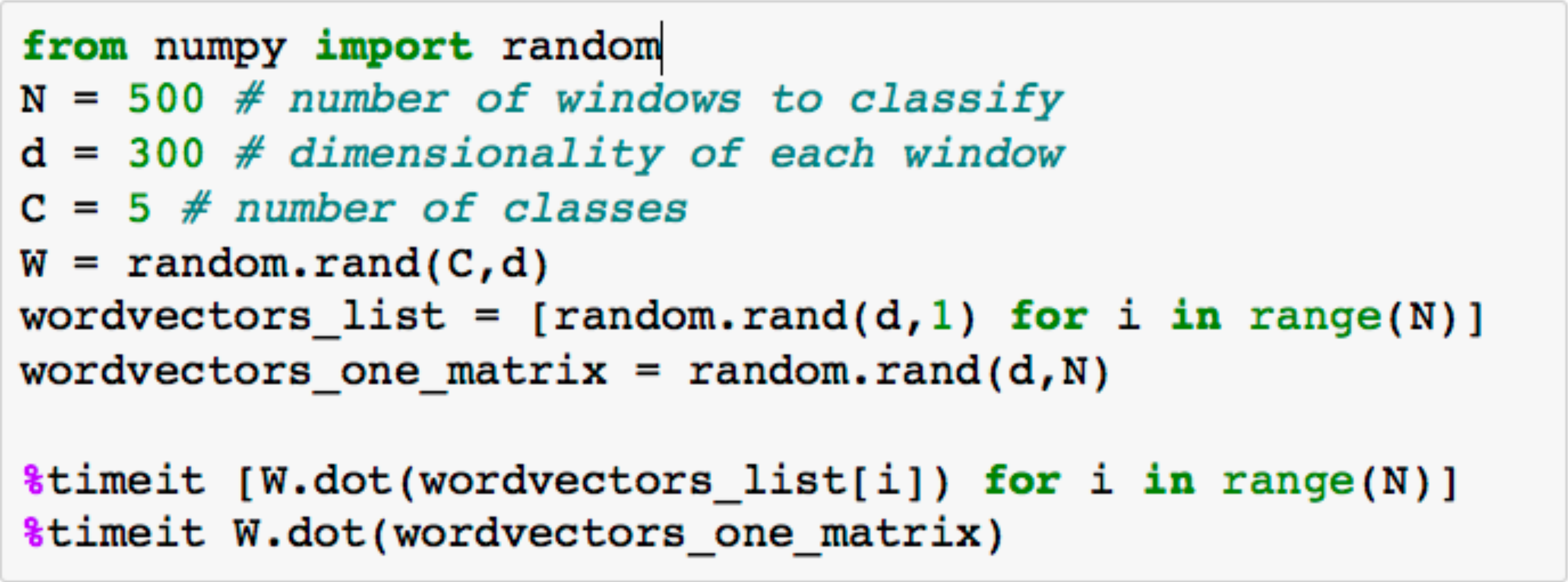
$$J(\theta) = \frac{1}{N}\sum_{i=1}^{N} -\log\left(\frac{e^{f_{y_i}}}{\sum_{c=1}^{C} e^{f_c}}\right) \boxed{+ \lambda \sum_{k} \theta_k^2}$$

- Regularization (largely) prevents overfitting when we have a lot of features (or later a very powerful/deep model, ++)



“Vectorization”

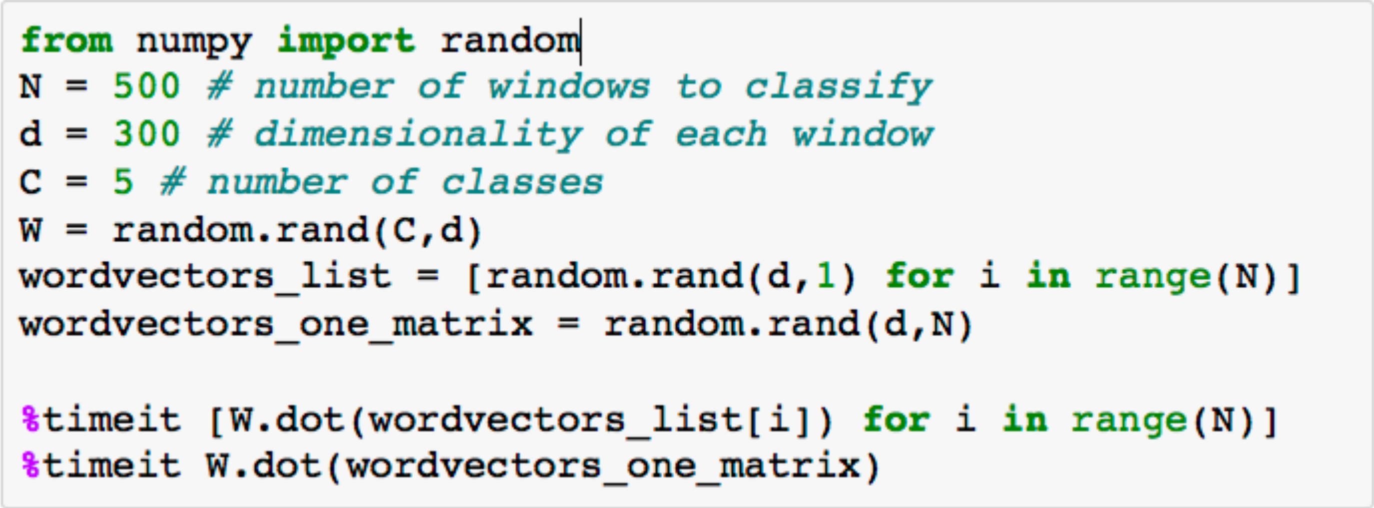
- E.g., looping over word vectors versus concatenating them all into one large matrix and then multiplying the softmax weights with that matrix



```
from numpy import random
N = 500 # number of windows to classify
d = 300 # dimensionality of each window
C = 5 # number of classes
W = random.rand(C,d)
wordvectors_list = [random.rand(d,1) for i in range(N)]
wordvectors_one_matrix = random.rand(d,N)

%timeit [W.dot(wordvectors_list[i]) for i in range(N)]
%timeit W.dot(wordvectors_one_matrix)
```

- 1000 loops, best of 3: **639 µs** per loop
  10000 loops, best of 3: **53.8 µs** per loop

“Vectorization”



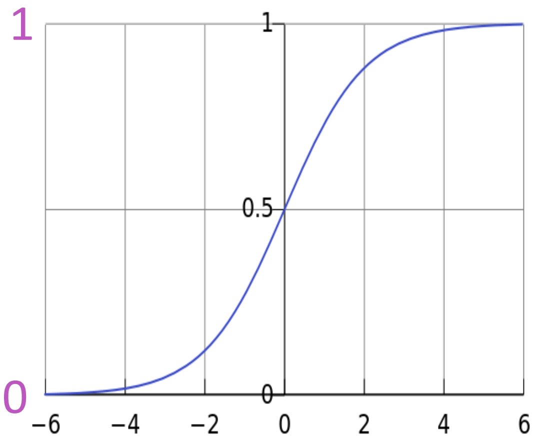
```
from numpy import random
N = 500 # number of windows to classify
d = 300 # dimensionality of each window
C = 5 # number of classes
W = random.rand(C,d)
wordvectors_list = [random.rand(d,1) for i in range(N)]
wordvectors_one_matrix = random.rand(d,N)

%timeit [W.dot(wordvectors_list[i]) for i in range(N)]
%timeit W.dot(wordvectors_one_matrix)
```

- The (10x) faster method is using a C x N matrix
- Always try to use vectors and matrices rather than for loops!
- You should speed-test your code a lot too!!
- tl;dr: Matrices are awesome!!!

Non-linearities: The starting points

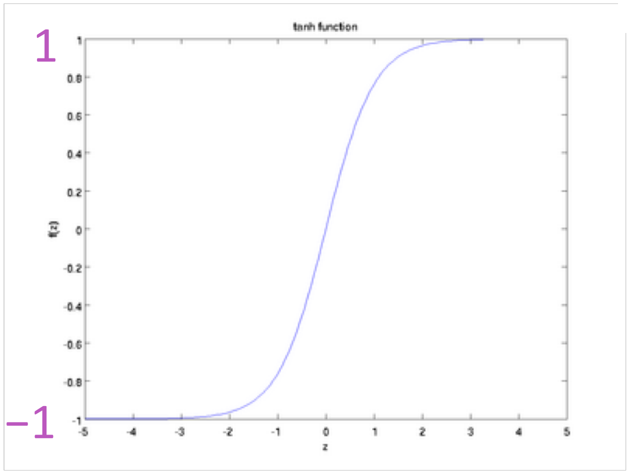
logistic (“sigmoid”)

$$f(z) = \frac{1}{1 + \exp(-z)}.$$



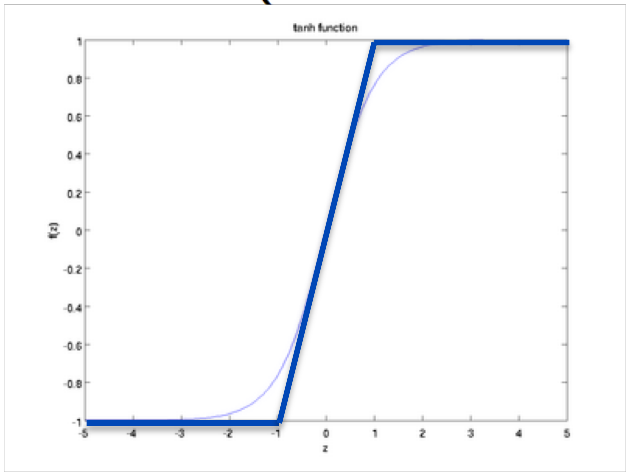
tanh

$$f(z) = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}},$$



hard tanh

$$\mathrm{HardTanh}(x) = \begin{cases} -1 & \text{if } x < -1 \\ x & \text{if } -1 <= x <= 1 \\ 1 & \text{if } x > 1 \end{cases}$$



tanh is just a rescaled and shifted sigmoid (2 × as steep, [−1,1]):

$$\tanh(z) = 2\mathrm{logistic}(2z) - 1$$

Both logistic and tanh are still used in particular uses, but are no longer the defaults for making deep networks

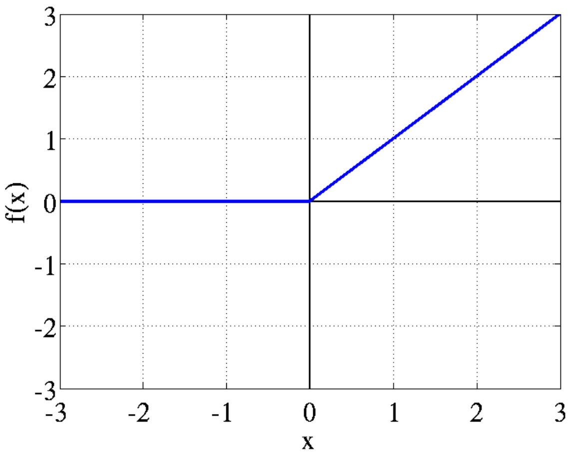
Non-linearities: The new world order

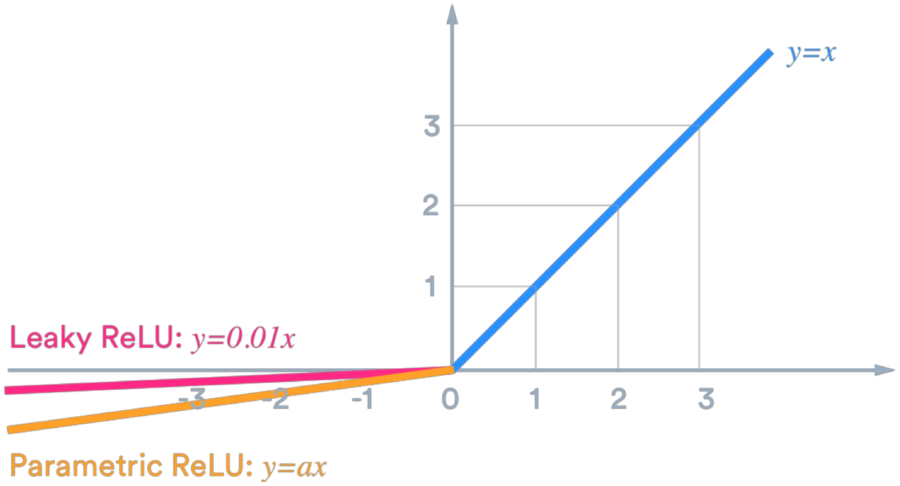
ReLU (rectified linear unit) hard tanh

Leaky ReLU

Parametric ReLU

$$\text{rect}(z) = \max(z, 0)$$

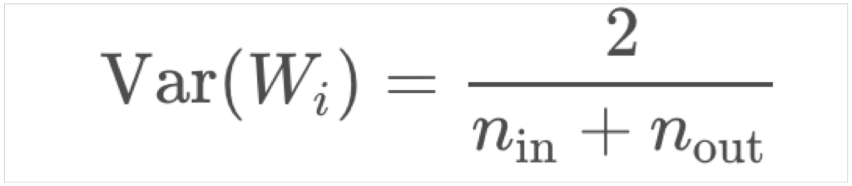




- For building a feed-forward deep network, the first thing you should try is ReLU — it trains quickly and performs well due to good gradient backflow

Parameter Initialization

- You normally must initialize weights to small random values
  - To avoid symmetries that prevent learning/specialization
- Initialize hidden layer biases to 0 and output (or reconstruction) biases to optimal value if weights were 0 (e.g., mean target or inverse sigmoid of mean target)
- Initialize **all other weights** ~ Uniform(–*r*, *r*), with *r* chosen so numbers get neither too big or too small
- Xavier initialization has variance inversely proportional to fan-in n_{in} (previous layer size) and fan-out n_{out} (next layer size):



$$\mathrm{Var}(W_i) = \frac{2}{n_{\mathrm{in}} + n_{\mathrm{out}}}$$

Optimizers

- Usually, plain SGD will work just fine
    - However, getting good results will often require hand-tuning the learning rate (next slide)
- For more complex nets and situations, or just to avoid worry, you often do better with one of a family of more sophisticated “adaptive” optimizers that scale the parameter adjustment by an accumulated gradient.
    - These models give per-parameter learning rates
        - Adagrad
        - RMSprop
        - Adam ← A fairly good, safe place to begin in many cases
        - SparseAdam
        - …

Learning Rates

- You can just use a constant learning rate. Start around *lr* = 0.001?
  - It must be order of magnitude right – try powers of 10
    - Too big: model may diverge or not converge
    - Too small: your model may not have trained by the deadline
- Better results can generally be obtained by allowing learning rates to decrease as you train
  - By hand: halve the learning rate every *k* epochs
    - An epoch = a pass through the data (shuffled or sampled)
  - By a formula: $lr = \ lr_0 e^{-kt}$, for epoch t
  - There are fancier methods like cyclic learning rates (q.v.)
- Fancier optimizers still use a learning rate but it may be an initial rate that the optimizer shrinks – so may be able to start high