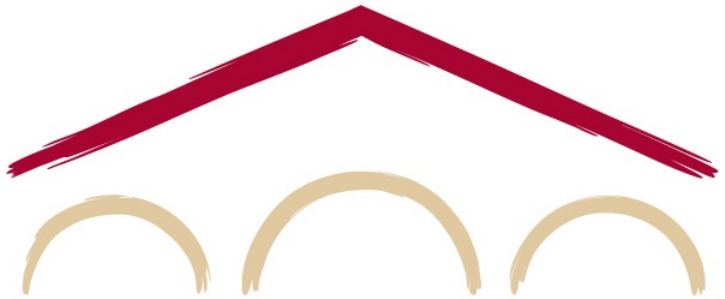
Natural Language Processing with Deep Learning CS224N/Ling284



Christopher Manning

Lecture 3: Neural net learning: Gradients by hand (matrix calculus) and algorithmically (the backpropagation algorithm)

1. Introduction

Assignment 2 makes sure you really understand the math of neural networks … then we'll let the software do it! It also teaches us about dependency parsing.

We'll go through all the math quickly today, but this is the one week of quarter to most work through the readings!!!

This will be a tough week for some! → Make sure to get help if you need it:

Visit office hours! Read tutorial materials on the syllabus!

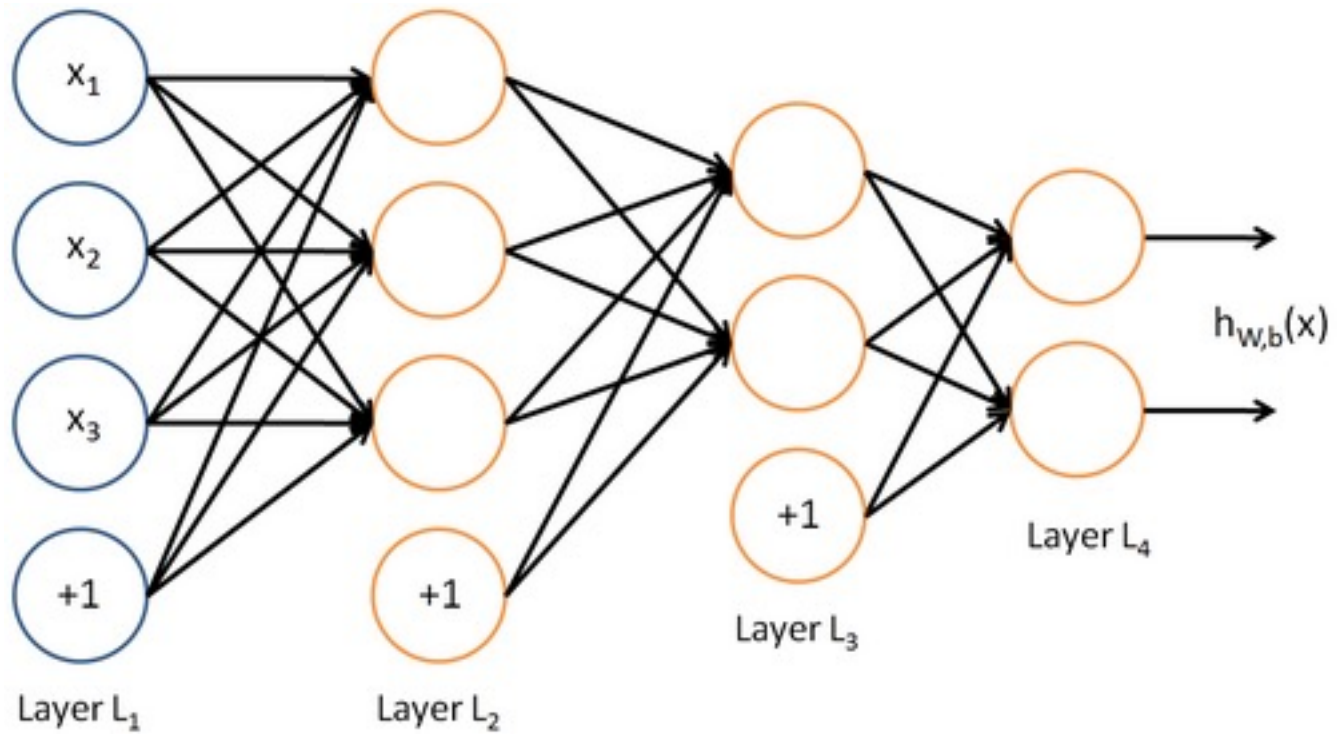
Thursday will be mainly linguistics! Some people find that tough too. 😉

PyTorch tutorial: 3:30pm Friday Apr 12 in Gates B01

A great chance to get an intro to PyTorch, a key deep learning package, used in Ass 2+!

Where we ended last time: A neural network = running several logistic regressions at the same time

Before we know it, we have a multilayer neural network….



This allows us to re-represent and compose our data multiple times and to learn a classifier that is highly non-linear in terms of the original inputs

(but, typically, is linear in terms of the pre-final layer representations)

Matrix notation for a layer

We have

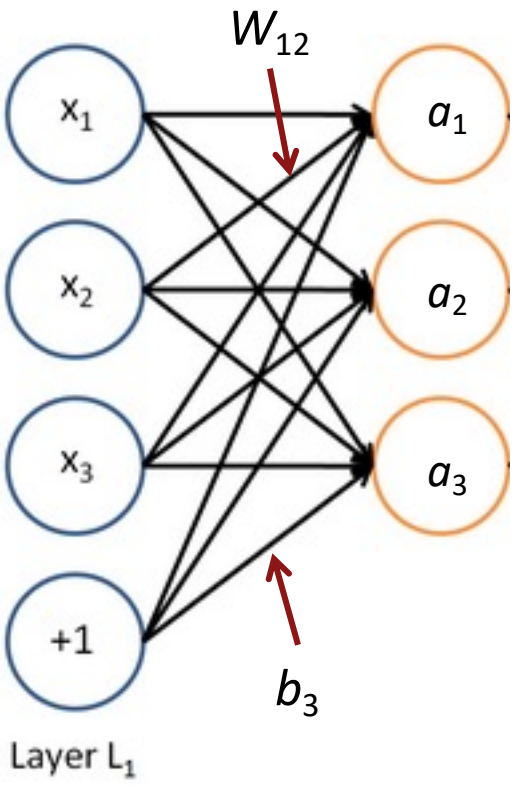
$$a_1 = f(W_{11}x_1 + W_{12}x_2 + W_{13}x_3 + b_1)$$
$$a_2 = f(W_{21}x_1 + W_{22}x_2 + W_{23}x_3 + b_2)$$
$$\text{etc.}$$

In matrix notation

$$z = Wx + b$$
$$a = f(z)$$

Activation f is applied element-wise:

$$f([z_1, z_2, z_3]) = [f(z_1), f(z_2), f(z_3)]$$

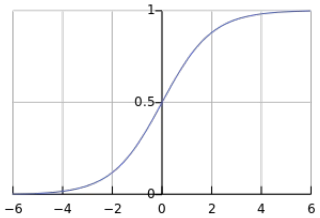


NER: Binary classification for center word being location

- We do supervised training and want high score if it's a location

$$J_t(\theta) = \sigma(s) = \frac{1}{1 + e^{-s}}$$

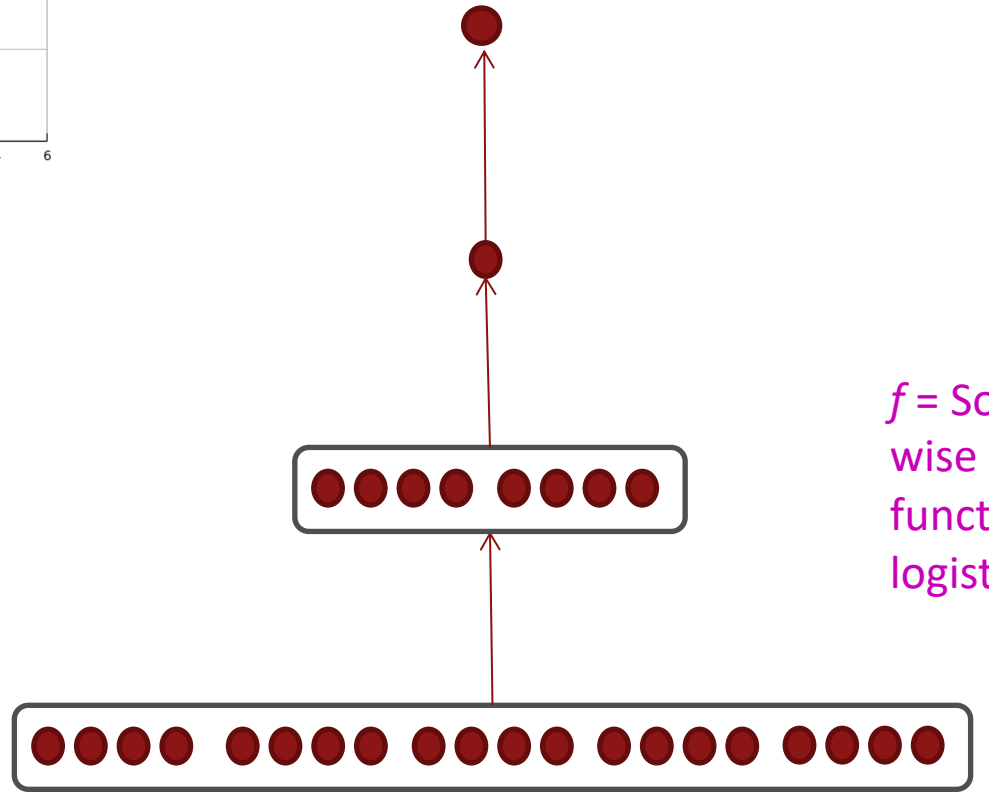
predicted model probability of class



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

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

$$\boldsymbol{x} \quad (\text{input}) \in \mathbb{R}^{5d}$$

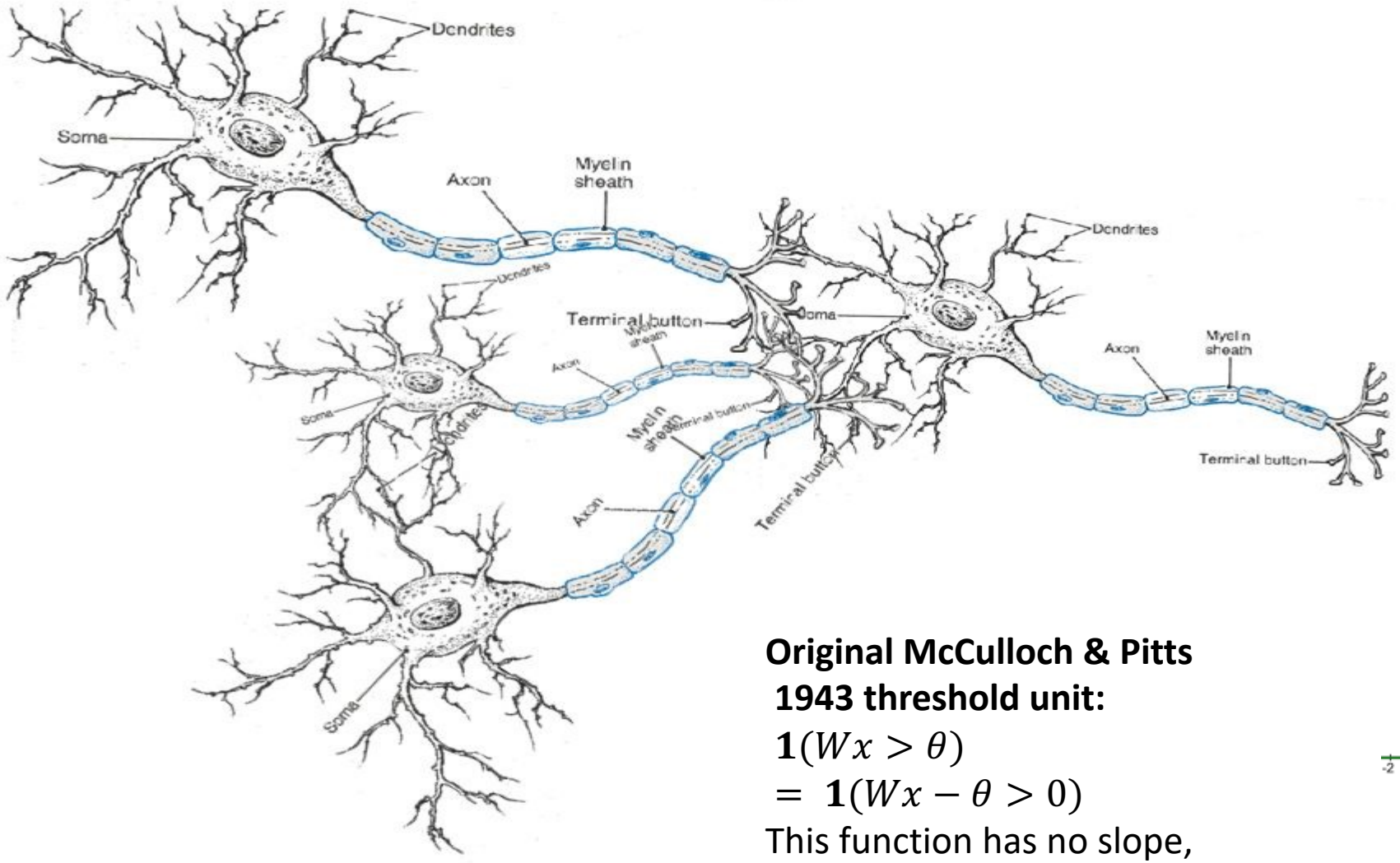


f = Some element-wise non-linear function, e.g., logistic, tanh, ReLU

$x = [\ x_{museums} \quad x_{in} \quad x_{Paris} \quad x_{are} \quad x_{amazing}\ ]$

Embedding of 1-hot words

7. Neural computation

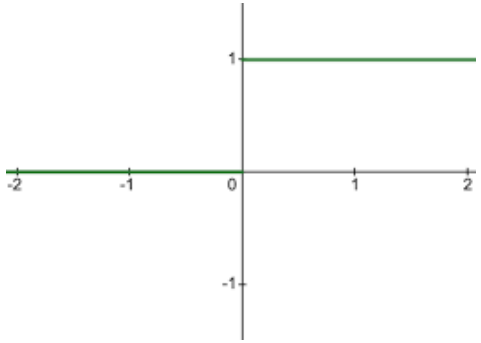


Original McCulloch & Pitts 1943 threshold unit:

$\mathbf{1}(Wx > \theta)$

$= \mathbf{1}(Wx - \theta > 0)$

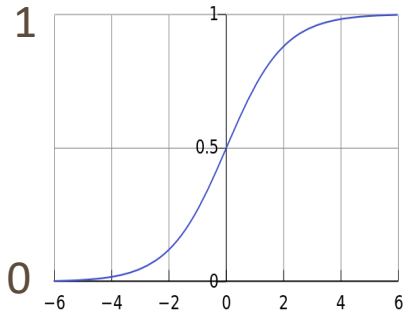
This function has no slope, so, no **gradient-based learning**



Non-linearities, old and new

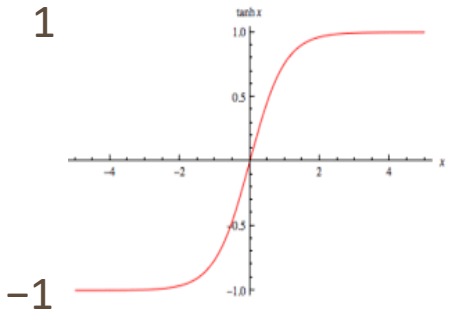
logistic ("sigmoid")

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



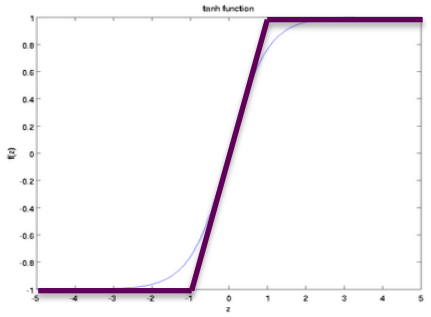
tanh

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



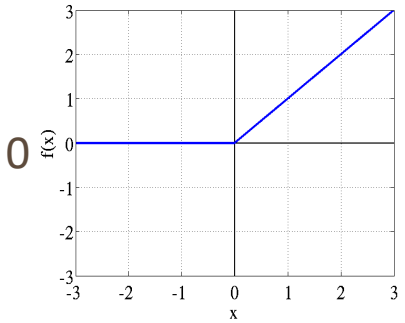
hard tanh

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

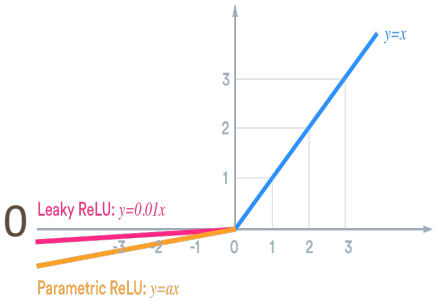


(Rectified Linear Unit) ReLU

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



Leaky ReLU / Parametric ReLU



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

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

Logistic and tanh are still used (e.g., logistic to get a probability)

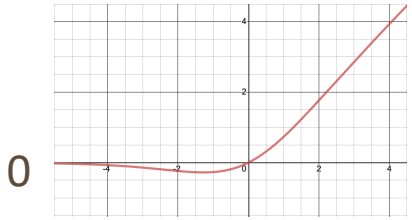
However, now, for deep networks, the first thing to try is ReLU: it trains quickly and performs well due to good gradient backflow.

ReLU has a negative "dead zone" that recent proposals mitigate

GELU/Swish often used with Transformers (BERT, RoBERTa, etc.)

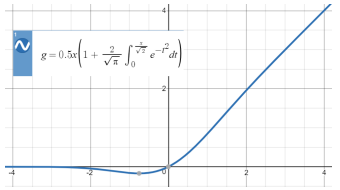
Swish arXiv:1710.05941

$$\text{swish}(x) = x \cdot \text{logistic}(x)$$



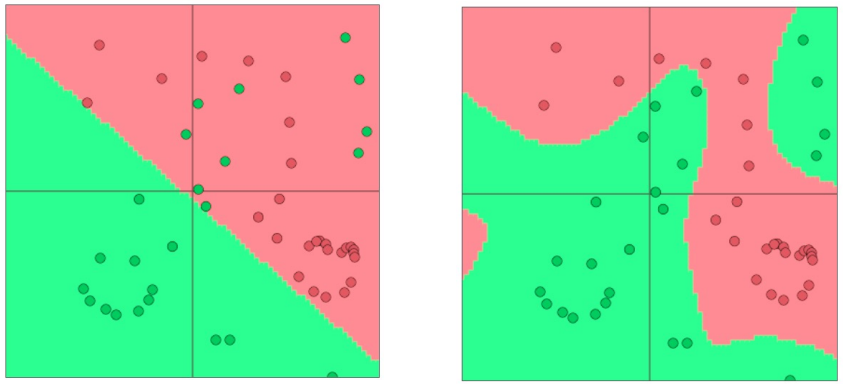
GELU arXiv:1606.08415

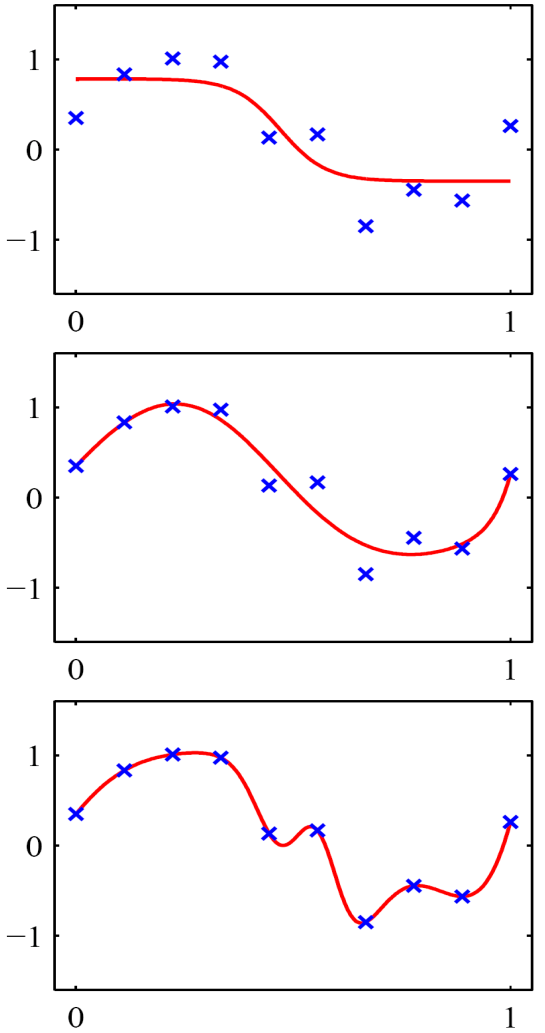
$$\text{GELU}(x) = x \cdot P(X \le x), X \sim N(0,1) \approx x \cdot \text{logistic}(1.702x)$$



Non-linearities (i.e., "*f*" on previous slide): Why they're needed

- Neural networks do function approximation, e.g., regression or classification
  - Without non-linearities, deep neural networks can't do anything more than a linear transform
  - Extra layers could just be compiled down into a single linear transform: $W_1 W_2 x = Wx$
  - But, with more layers that include non-linearities, they can approximate any complex function!





Remember: Stochastic Gradient Descent

Update equation:

$$\theta^{new} = \theta^{old} - \alpha \nabla_\theta J(\theta)$$

α = *step size* or *learning rate*

i.e., for each parameter: $\theta_j^{new} = \theta_j^{old} - \alpha \frac{\partial J(\theta)}{\partial \theta_j^{old}}$

In deep learning, θ includes the data representation (e.g., word vectors) too!

How can we compute $\nabla_\theta J(\theta)$?

1. By hand
2. Algorithmically: the backpropagation algorithm

Lecture Plan

Lecture 4: Gradients by hand and algorithmically

1. Introduction (10 mins)
2. Matrix calculus (35 mins)
3. Backpropagation (35 mins)

Key Learning: The mathematics and practical implementation of how neural networks are trained by backpropagation

Computing Gradients by Hand

- **Matrix calculus:** Fully vectorized gradients
  - "Multivariable calculus is just like single-variable calculus if you use matrices"
  - Much faster and more useful than non-vectorized gradients
  - But doing a non-vectorized gradient can be good for intuition; recall the first lecture for an example
  - **Lecture notes and matrix calculus notes cover this material in more detail**
  - **You might also review Math 51, which has an online textbook: http://web.stanford.edu/class/math51/textbook.html**

Linear Algebra, Multivariable Calculus, and Modern Applications

Math 51 course text prepared by the
Stanford University Math Department

Last modified on April 3, 2024

"... research on neural networks was held up for 20 years until somebody remembered the Chain Rule!"

T. Griffiths, Luce Professor of Information Technology, Consciousness, and Culture (Princeton)

"... linear algebra simplifies the complex processes that underlie neural network operations, [...] allowing [AI models] to efficiently process vast amounts of data, recognize patterns, and make decisions."

Medium article "The Crucial Role of Mathematics in AI Development"

G. Neural networks and the multivariable Chain Rule (optional)

The ability of computers to identify high-level patterns through experience (i.e., machine-learning based on training data) has been demonstrated quite spectacularly with AlphaGo (and its descendants such as AlphaZero). This is based on the concept of a *neural network*, which we discuss here in order to identify where an essential mathematical insight occurs involving the multivariable Chain Rule.

G.1. Mathematical model of a neural network as a composition of functions. Consider the task of a computer being given a picture with 1000 pixels (encoded as a vector in $\mathbf{R}^{1000}$) and aiming to determine whether or not it is a picture of a cat. The output of the computer's work will be a single real number between 0 and 1 that measures how likely it is that the picture is of a cat.

The overall process of feeding the picture into the computer and extracting the output from the computer should be given by evaluating a function $f_{\text{cat}} : \mathbf{R}^{1000} \to \mathbf{R}$ on a 1000-vector that encodes the input of the pixel intensities. The desired features of f_{cat} are:

- if the input is a picture of a cat, f_{cat} returns a value > 0.5,
- if the input is not a picture of a cat then f_{cat} returns a value < 0.5.

This is a "classifier" problem (cat or not cat?), and the computer needs to *construct* such a function! The process of building f_{cat} inside a computer (in a way we describe below) is an instance of "machine learning". It must be kept in mind that the function f_{cat} constructed by the computer will be an extraordinarily non-linear function, far too complicated for any human to ever discover or write down.

Two different computers working on this same "machine learning" task may build very different functions f_{cat}. All that matters is that the function does a good job at answering "cat or not cat?"; many different functions may perform equally well at this task, and we only care that the computer constructs an f_{cat} that works well in practice. The process by which f_{cat} is built inside a computer involves expressing it as a composition of many intermediate vector-valued functions (that also have to be constructed by the computer), and this all makes essential use of multivariable differential calculus, as we will explain below.

From a mathematical point of view, a *neural network* is an expression of a function $f : \mathbf{R}^n \to \mathbf{R}^m$ (often $m = 1$, as for f_{cat} above) as a composition of (typically highly non-linear) vector-valued functions:

$$\mathbf{R}^n \xrightarrow{\mathbf{f}_1} \mathbf{R}^{d_1} \xrightarrow{\mathbf{f}_2} \mathbf{R}^{d_2} \xrightarrow{\mathbf{f}_3} \cdots \xrightarrow{\mathbf{f}_{N-1}} \mathbf{R}^{d_{N-1}} \xrightarrow{\mathbf{f}_N} \mathbf{R}^m$$

in which $\mathbf{R}^n$ on the left is called the *input layer*, $\mathbf{R}^m$ on the right is called the *output layer*, and each $\mathbf{R}^{d_i}$ in between is called a *hidden layer*. Such a composition involving N functions $\mathbf{f}_i$ is called an N*-layer* neural network (so a single-layer network involves no hidden layers and so no function composition at all).

Defining $d_0 = n$ and $d_N = m$ for convenience of notation, we refer to $\mathbf{R}^{d_j}$ as the jth *layer* (with $j = 0, 1, \ldots, N$) and call d_j the number of *nodes* in that layer. Each function $\mathbf{f}_i$ describes how to get from the output of the $(i-1)$th layer (which means just the input when $i = 1$) to the next layer. If we denote a vector in $\mathbf{R}^{d_j}$ as $\mathbf{x}_j = (x_{1,j}, \ldots, x_{d_j,j})$ then (for reasons based on analogies with the behavior of neurons in a brain) each $x_{i,j}$ is called a *neuron*, hence the name "neural network" for the overall function composition process. Many people dislike the analogies with a brain and so avoid such terminology, instead referring to each $x_{i,j}$ as a *unit* and calling the overall composition process a *multi-layer perceptron*.

Gradients

- Given a function with 1 output and 1 input

$$f(x) = x^3$$

- It's gradient (slope) is its derivative

$$\frac{df}{dx} = 3x^2$$

"How much will the output change if we change the input a bit?"

At $x = 1$ it changes about 3 times as much: $1.01^3 = 1.03$

At x = 4 it changes about 48 times as much: $4.01^3 = 64.48$

Gradients

- Given a function with 1 output and *n* inputs

$$f(\boldsymbol{x}) = f(x_1, x_2, ..., x_n)$$

- Its gradient is a vector of partial derivatives with respect to each input

$$\frac{\partial f}{\partial \boldsymbol{x}} = \left[\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \cdots, \frac{\partial f}{\partial x_n}\right]$$

Jacobian Matrix: Generalization of the Gradient

- Given a function with ***m* outputs** and *n* inputs

$$\boldsymbol{f}(\boldsymbol{x}) = [f_1(x_1, x_2, ..., x_n), ..., f_m(x_1, x_2, ..., x_n)]$$

- It's Jacobian is an ***m* x *n* matrix** of partial derivatives

$$\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{x}} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix}$$

$$\left(\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{x}}\right)_{ij} = \frac{\partial f_i}{\partial x_j}$$

Chain Rule

- For composition of one-variable functions: **multiply derivatives**

$$z = 3y$$

$$y = x^2$$

$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx} = (3)(2x) = 6x$$

- For multiple variables functions: **multiply Jacobians**

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

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

$$\frac{\partial \boldsymbol{h}}{\partial \boldsymbol{x}} = \frac{\partial \boldsymbol{h}}{\partial \boldsymbol{z}}\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{x}} = \dots$$

Example Jacobian: Elementwise activation Function

$$\boldsymbol{h} = f(\boldsymbol{z}), \text{what is } \frac{\partial \boldsymbol{h}}{\partial \boldsymbol{z}}? \qquad \boldsymbol{h}, \boldsymbol{z} \in \mathbb{R}^n$$

$$h_i = f(z_i)$$

Example Jacobian: Elementwise activation Function

$$\boldsymbol{h} = f(\boldsymbol{z}), \text{what is } \frac{\partial \boldsymbol{h}}{\partial \boldsymbol{z}}? \qquad \boldsymbol{h}, \boldsymbol{z} \in \mathbb{R}^n$$

$$h_i = f(z_i)$$

Function has *n* outputs and *n* inputs → *n* by *n* Jacobian

Example Jacobian: Elementwise activation Function

$$\boldsymbol{h} = f(\boldsymbol{z}), \text{what is } \frac{\partial \boldsymbol{h}}{\partial \boldsymbol{z}}? \qquad \boldsymbol{h}, \boldsymbol{z} \in \mathbb{R}^n$$

$$h_i = f(z_i)$$

$$\left(\frac{\partial \boldsymbol{h}}{\partial \boldsymbol{z}}\right)_{ij} = \frac{\partial h_i}{\partial z_j} = \frac{\partial}{\partial z_j} f(z_i) \qquad \text{definition of Jacobian}$$

Example Jacobian: Elementwise activation Function

$$\boldsymbol{h} = f(\boldsymbol{z}), \text{what is } \frac{\partial \boldsymbol{h}}{\partial \boldsymbol{z}}? \qquad \boldsymbol{h}, \boldsymbol{z} \in \mathbb{R}^n$$

$$h_i = f(z_i)$$

$$\left(\frac{\partial \boldsymbol{h}}{\partial \boldsymbol{z}}\right)_{ij} = \frac{\partial h_i}{\partial z_j} = \frac{\partial}{\partial z_j} f(z_i) \qquad \text{definition of Jacobian}$$

$$= \begin{cases} f'(z_i) & \text{if } i = j \\ 0 & \text{if otherwise} \end{cases} \qquad \text{regular 1-variable derivative}$$

Example Jacobian: Elementwise activation Function

$$\boldsymbol{h} = f(\boldsymbol{z}), \text{what is } \frac{\partial \boldsymbol{h}}{\partial \boldsymbol{z}}? \qquad \boldsymbol{h}, \boldsymbol{z} \in \mathbb{R}^n$$

$$h_i = f(z_i)$$

$$\left(\frac{\partial \boldsymbol{h}}{\partial \boldsymbol{z}}\right)_{ij} = \frac{\partial h_i}{\partial z_j} = \frac{\partial}{\partial z_j} f(z_i) \qquad \text{definition of Jacobian}$$

$$= \begin{cases} f'(z_i) & \text{if } i = j \\ 0 & \text{if otherwise} \end{cases} \qquad \text{regular 1-variable derivative}$$

$$\frac{\partial \boldsymbol{h}}{\partial \boldsymbol{z}} = \begin{pmatrix} f'(z_1) & & 0 \\ & \ddots & \\ 0 & & f'(z_n) \end{pmatrix} = \text{diag}(\boldsymbol{f}'(\boldsymbol{z}))$$

Other Jacobians

$$\frac{\partial}{\partial \boldsymbol{x}}(\boldsymbol{W}\boldsymbol{x}+\boldsymbol{b}) = \boldsymbol{W}$$

Other Jacobians

$$\frac{\partial}{\partial \boldsymbol{x}}(\boldsymbol{W}\boldsymbol{x}+\boldsymbol{b}) = \boldsymbol{W}$$

$$\frac{\partial}{\partial \boldsymbol{b}}(\boldsymbol{W}\boldsymbol{x}+\boldsymbol{b}) = \boldsymbol{I} \ \ (\text{Identity matrix})$$

Other Jacobians

$$\frac{\partial}{\partial \boldsymbol{x}}(\boldsymbol{W}\boldsymbol{x}+\boldsymbol{b}) = \boldsymbol{W}$$

$$\frac{\partial}{\partial \boldsymbol{b}}(\boldsymbol{W}\boldsymbol{x}+\boldsymbol{b}) = \boldsymbol{I} \quad (\text{Identity matrix})$$

$$\frac{\partial}{\partial \boldsymbol{u}}(\boldsymbol{u}^T\boldsymbol{h}) = \boldsymbol{h}^T$$

Fine print: This is the correct Jacobian. Later we discuss the "shape convention"; using it the answer would be ***h***.

Other Jacobians

$$\frac{\partial}{\partial \boldsymbol{x}}(\boldsymbol{W}\boldsymbol{x}+\boldsymbol{b}) = \boldsymbol{W}$$

$$\frac{\partial}{\partial \boldsymbol{b}}(\boldsymbol{W}\boldsymbol{x}+\boldsymbol{b}) = \boldsymbol{I} \quad (\text{Identity matrix})$$

$$\frac{\partial}{\partial \boldsymbol{u}}(\boldsymbol{u}^T\boldsymbol{h}) = \boldsymbol{h}^{\boldsymbol{T}}$$

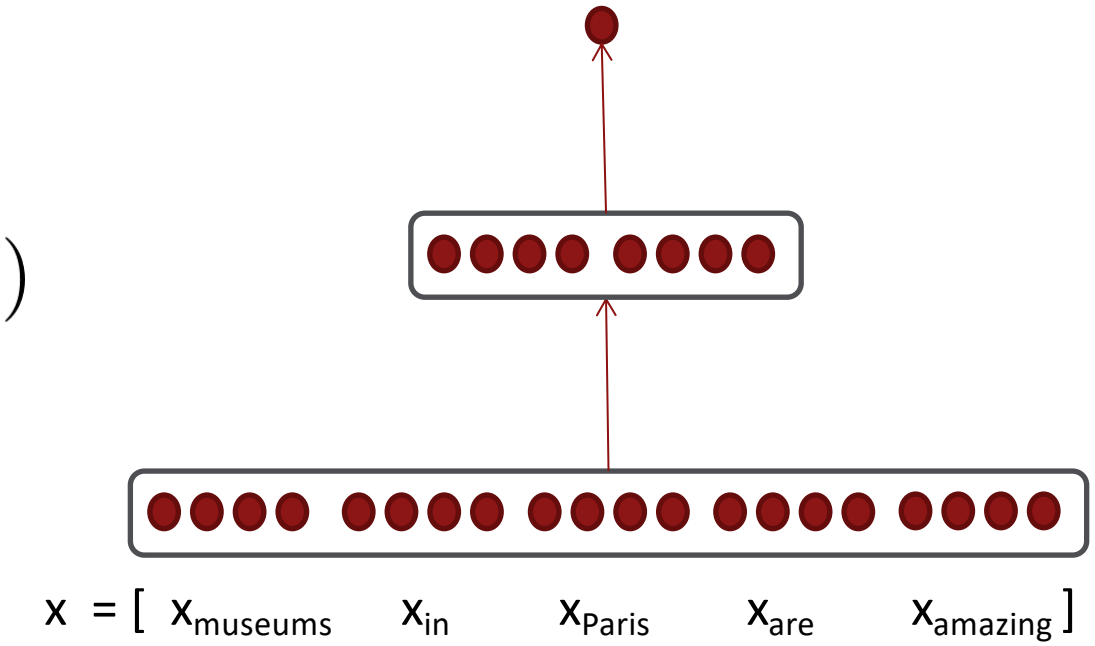
- Compute these at home for practice!
  - Check your answers with the lecture notes

Back to our Neural Net!

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

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

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



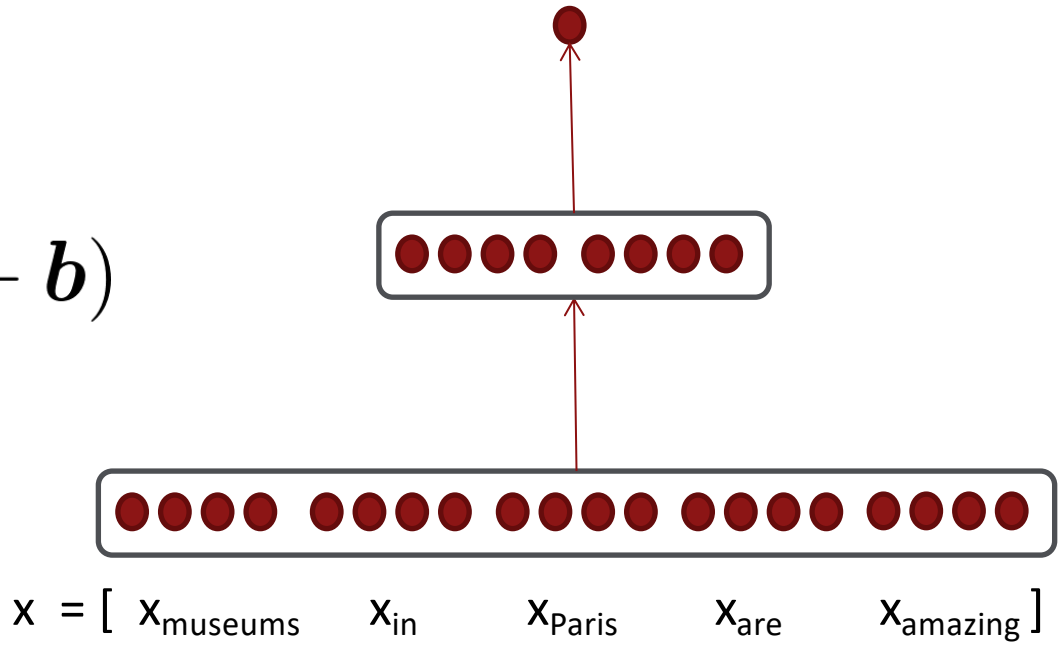
Back to our Neural Net!

- Let's find $\frac{\partial s}{\partial \boldsymbol{b}}$
  - Really, we care about the gradient of the loss J_t but we will compute the gradient of the score for simplicity (and because the logistic is in Ass 1!)

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

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

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



1. Break up equations into simple pieces

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

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

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

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

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

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

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

Carefully define your variables and keep track of their dimensionality!

2. Apply the chain rule

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

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

2. Apply the chain rule

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

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

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

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

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

2. Apply the chain rule

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

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

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

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

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

2. Apply the chain rule

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

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

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

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

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

3. Write out the Jacobians

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

$$\frac{\partial s}{\partial \boldsymbol{b}} = \frac{\partial s}{\partial \boldsymbol{h}} \quad \frac{\partial \boldsymbol{h}}{\partial \boldsymbol{z}} \quad \frac{\partial \boldsymbol{z}}{\partial \boldsymbol{b}}$$

Useful Jacobians from previous slide

$$\frac{\partial}{\partial \boldsymbol{u}}(\boldsymbol{u}^T \boldsymbol{h}) = \boldsymbol{h}^T$$
$$\frac{\partial}{\partial \boldsymbol{z}}(f(\boldsymbol{z})) = \text{diag}(f'(\boldsymbol{z}))$$
$$\frac{\partial}{\partial \boldsymbol{b}}(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}) = \boldsymbol{I}$$

3. Write out the Jacobians

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

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

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

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

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

$$\downarrow$$

$$\boldsymbol{u}^T$$

Useful Jacobians from previous slide

$$\frac{\partial}{\partial \boldsymbol{u}}(\boldsymbol{u}^T \boldsymbol{h}) = \boldsymbol{h}^T$$

$$\frac{\partial}{\partial \boldsymbol{z}}(f(\boldsymbol{z})) = \text{diag}(f'(\boldsymbol{z}))$$

$$\frac{\partial}{\partial \boldsymbol{b}}(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}) = \boldsymbol{I}$$

3. Write out the Jacobians

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

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

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

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

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

$$\boldsymbol{u}^T \operatorname{diag}(f'(\boldsymbol{z}))$$

Useful Jacobians from previous slide

$$\frac{\partial}{\partial \boldsymbol{u}}(\boldsymbol{u}^T \boldsymbol{h}) = \boldsymbol{h}^T$$

$$\frac{\partial}{\partial \boldsymbol{z}}(f(\boldsymbol{z})) = \operatorname{diag}(f'(\boldsymbol{z}))$$

$$\frac{\partial}{\partial \boldsymbol{b}}(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}) = \boldsymbol{I}$$

3. Write out the Jacobians

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

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

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

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

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

$$= \boldsymbol{u}^T \operatorname{diag}(f'(\boldsymbol{z}))\boldsymbol{I}$$

Useful Jacobians from previous slide

$$\frac{\partial}{\partial \boldsymbol{u}}(\boldsymbol{u}^T \boldsymbol{h}) = \boldsymbol{h}^T$$

$$\frac{\partial}{\partial \boldsymbol{z}}(f(\boldsymbol{z})) = \operatorname{diag}(f'(\boldsymbol{z}))$$

$$\frac{\partial}{\partial \boldsymbol{b}}(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}) = \boldsymbol{I}$$

3. Write out the Jacobians

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

$$\frac{\partial s}{\partial \boldsymbol{b}} = \frac{\partial s}{\partial \boldsymbol{h}} \frac{\partial \boldsymbol{h}}{\partial \boldsymbol{z}} \frac{\partial \boldsymbol{z}}{\partial \boldsymbol{b}}$$
$$= \boldsymbol{u}^T \operatorname{diag}(f'(\boldsymbol{z}))\boldsymbol{I}$$
$$= \boldsymbol{u}^T \circ f'(\boldsymbol{z})$$

Useful Jacobians from previous slide

$$\frac{\partial}{\partial \boldsymbol{u}}(\boldsymbol{u}^T \boldsymbol{h}) = \boldsymbol{h}^T$$
$$\frac{\partial}{\partial \boldsymbol{z}}(f(\boldsymbol{z})) = \operatorname{diag}(f'(\boldsymbol{z}))$$
$$\frac{\partial}{\partial \boldsymbol{b}}(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}) = \boldsymbol{I}$$

⊙ = Hadamard product = element-wise multiplication of 2 vectors to give vector

Re-using Computation

- Suppose we now want to compute $\frac{\partial s}{\partial \boldsymbol{W}}$
  - Using the chain rule again:

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

Re-using Computation

- Suppose we now want to compute $\frac{\partial s}{\partial \boldsymbol{W}}$
  - Using the chain rule again:

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

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

The same! Let's avoid duplicated computation …

Re-using Computation

- Suppose we now want to compute $\frac{\partial s}{\partial \boldsymbol{W}}$
  - Using the chain rule again:

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

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

$$\boldsymbol{\delta} = \frac{\partial s}{\partial \boldsymbol{h}} \frac{\partial \boldsymbol{h}}{\partial \boldsymbol{z}} = \boldsymbol{u}^T \circ f'(\boldsymbol{z})$$

δ is the upstream gradient ("error signal")

Derivative with respect to Matrix: Output shape

- What does $\frac{\partial s}{\partial \boldsymbol{W}}$ look like? $\boldsymbol{W} \in \mathbb{R}^{n \times m}$
- 1 output, *nm* inputs: 1 by *nm* Jacobian?
  - Inconvenient to then do $\theta^{new} = \theta^{old} - \alpha \nabla_\theta J(\theta)$

Derivative with respect to Matrix: Output shape

- What does $\frac{\partial s}{\partial \boldsymbol{W}}$ look like? $\boldsymbol{W} \in \mathbb{R}^{n \times m}$
- 1 output, *nm* inputs: 1 by *nm* Jacobian?
  - Inconvenient to then do $\theta^{new} = \theta^{old} - \alpha \nabla_\theta J(\theta)$

- Instead, we leave pure math and use the **shape convention**: the shape of the gradient is the shape of the parameters!
  - So $\frac{\partial s}{\partial \boldsymbol{W}}$ is *n* by *m*:

$$\begin{bmatrix} \frac{\partial s}{\partial W_{11}} & \cdots & \frac{\partial s}{\partial W_{1m}} \\ \vdots & \ddots & \vdots \\ \frac{\partial s}{\partial W_{n1}} & \cdots & \frac{\partial s}{\partial W_{nm}} \end{bmatrix}$$

Derivative with respect to Matrix

- What is $\frac{\partial s}{\partial \boldsymbol{W}} = \boldsymbol{\delta} \frac{\partial \boldsymbol{z}}{\partial \boldsymbol{W}}$
  - $\boldsymbol{\delta}$ is going to be in our answer
  - The other term should be $\boldsymbol{x}$ because $\boldsymbol{z} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}$

- Answer is: $\frac{\partial s}{\partial \boldsymbol{W}} = \boldsymbol{\delta}^T \boldsymbol{x}^T$

δ is upstream gradient ("error signal") at z

x is local input signal

Why the Transposes?

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

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

$$= \begin{bmatrix} \delta_1 \\ \vdots \\ \delta_n \end{bmatrix} [x_1, ..., x_m] = \begin{bmatrix} \delta_1 x_1 & \dots & \delta_1 x_m \\ \vdots & \ddots & \vdots \\ \delta_n x_1 & \dots & \delta_n x_m \end{bmatrix}$$

- Hacky answer: this makes the dimensions work out!
  - Useful trick for checking your work!
- Full explanation in the lecture notes
  - Each input goes to each output – you want to get outer product

Deriving local input gradient in backprop

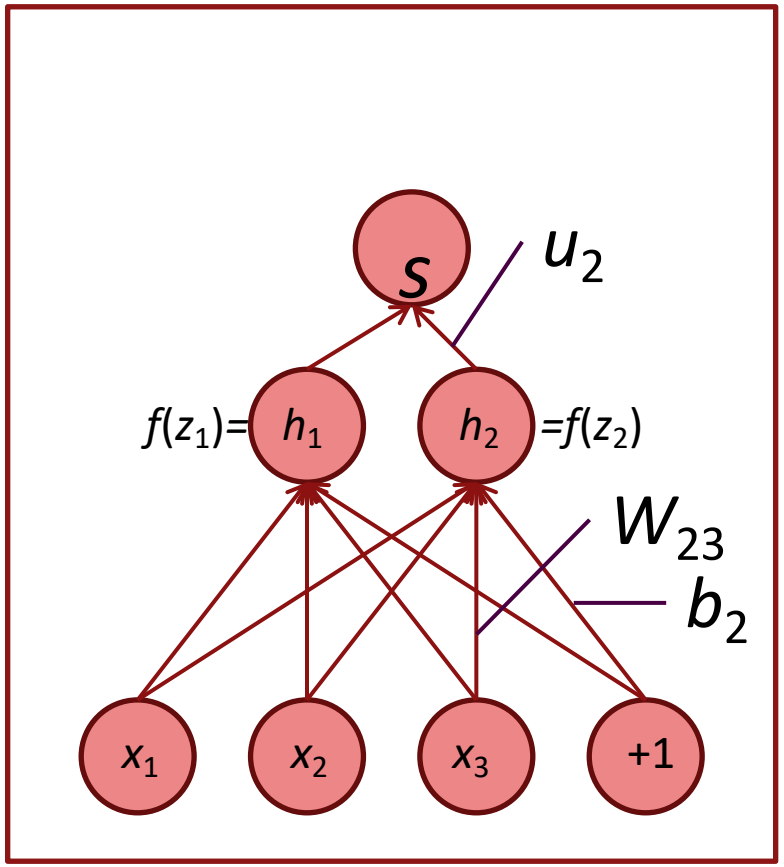
- For $\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{W}}$ in our equation:

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

- 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

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

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



What shape should derivatives be?

- Similarly, $\frac{\partial s}{\partial \boldsymbol{b}} = \boldsymbol{h}^T \circ f'(\boldsymbol{z})$ is a row vector
  - But shape convention says our gradient should be a column vector because ***b*** is a column vector …
- Disagreement between Jacobian form (which makes the chain rule easy) and the shape convention (which makes implementing SGD easy)
  - We expect answers in the assignment to follow the **shape convention**
  - But Jacobian form is useful for computing the answers

What shape should derivatives be?

Two options for working through specific problems:

1. Use Jacobian form as much as possible, reshape to follow the shape convention at the end:
   - What we just did. But at the end transpose $\frac{\partial s}{\partial \boldsymbol{b}}$ to make the derivative a column vector, resulting in $\boldsymbol{\delta}^T$
2. Always follow the shape convention
   - Look at dimensions to figure out when to transpose and/or reorder terms
   - The error message $\boldsymbol{\delta}$ that arrives at a hidden layer has the same dimensionality as that hidden layer

3. Backpropagation

We've almost shown you backpropagation

- It's taking derivatives and using the (generalized, multivariate, or matrix) chain rule

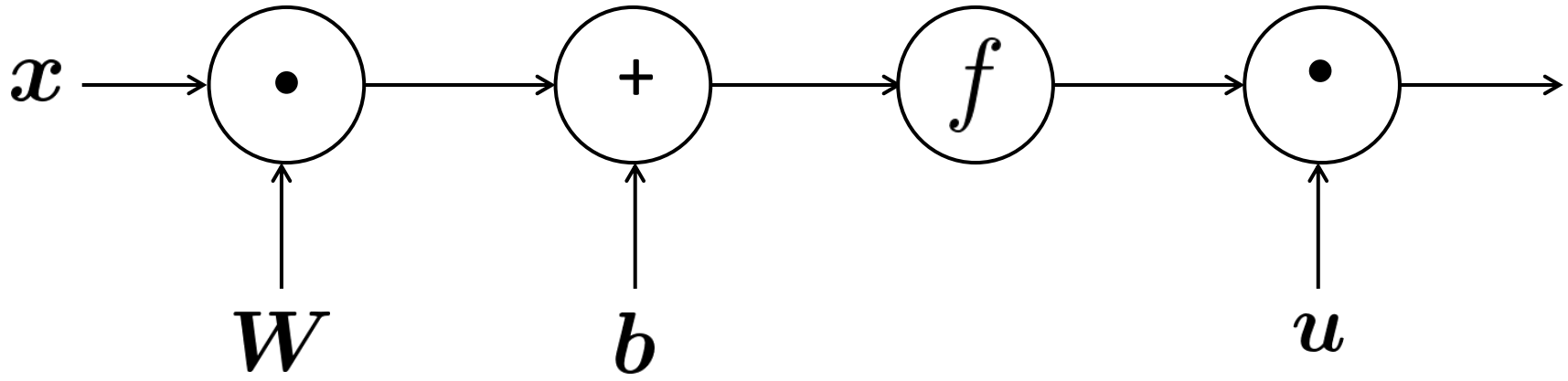
One more concept:

- We **re-use** derivatives computed for higher layers in computing derivatives for lower layers to minimize computation

Computation Graphs and Backpropagation

- Software represents our neural net equations as a graph
  - Source nodes: inputs
  - Interior nodes: operations

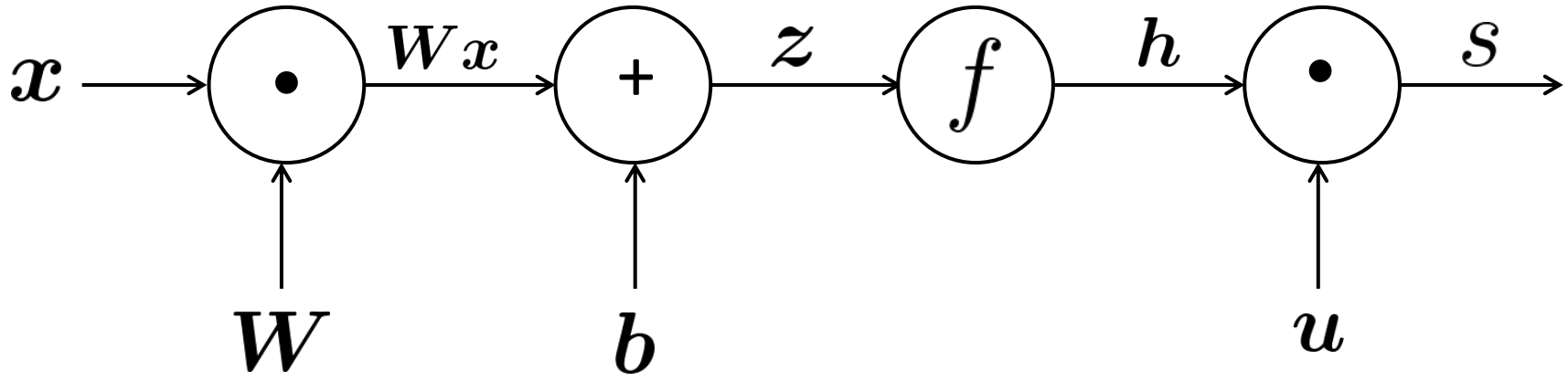
$$s = \boldsymbol{u}^T \boldsymbol{h}$$
$$\boldsymbol{h} = f(\boldsymbol{z})$$
$$\boldsymbol{z} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}$$
$$\boldsymbol{x} \quad (\text{input})$$



Computation Graphs and Backpropagation

- Software represents 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

- Software represents our neural net equations as a graph

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

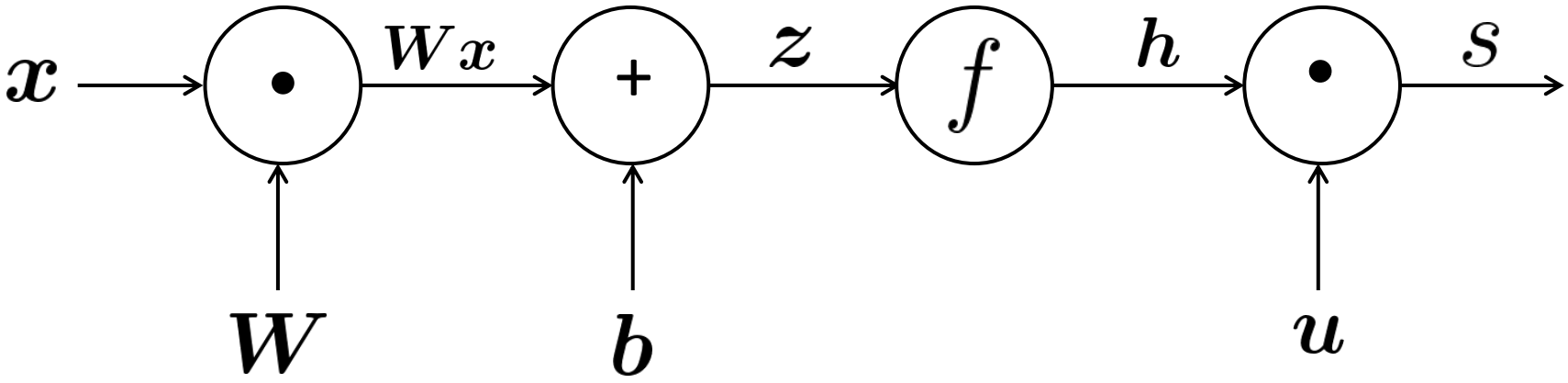
$$\boldsymbol{h} = f(\boldsymbol{z})$$

$$\boldsymbol{z} + \boldsymbol{b}$$

$$\text{ut})$$

"Forward Propagation"

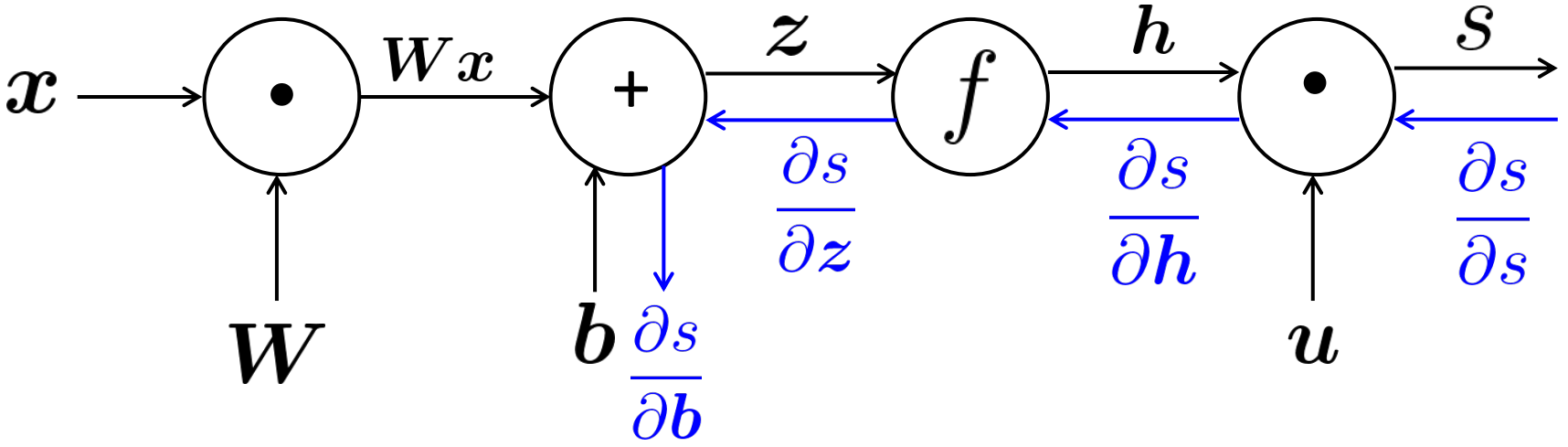
operation



Backpropagation

- Then go backwards along edges
  - Pass along **gradients**

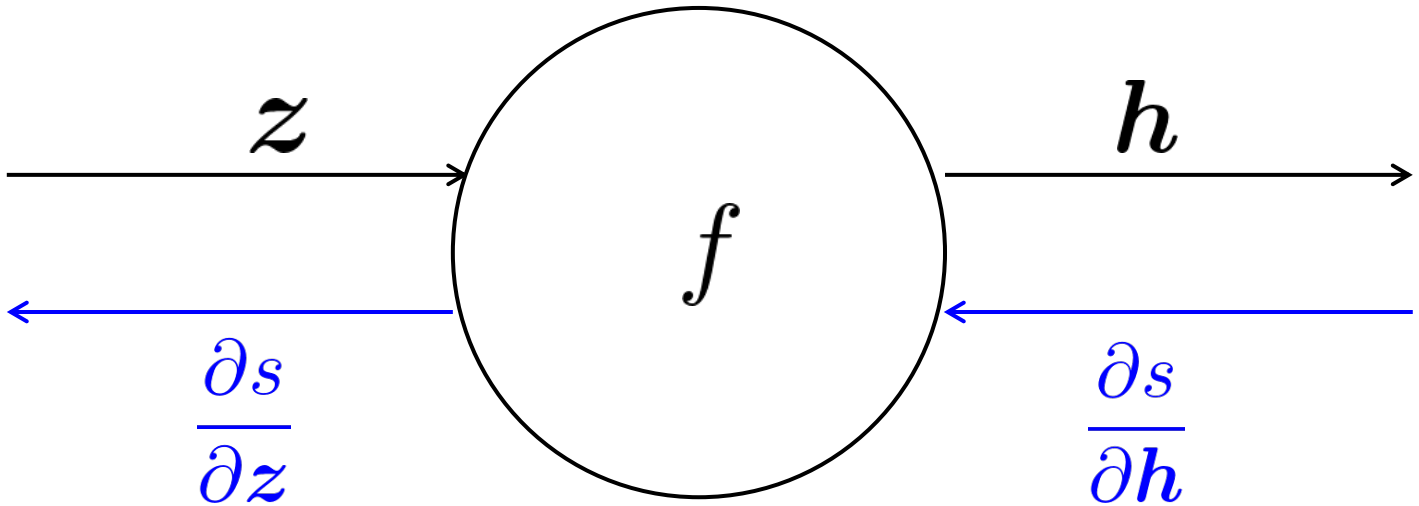
$$s = \boldsymbol{u}^T \boldsymbol{h}$$
$$\boldsymbol{h} = f(\boldsymbol{z})$$
$$\boldsymbol{z} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}$$
$$\boldsymbol{x} \quad \text{(input)}$$



Backpropagation: Single Node

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

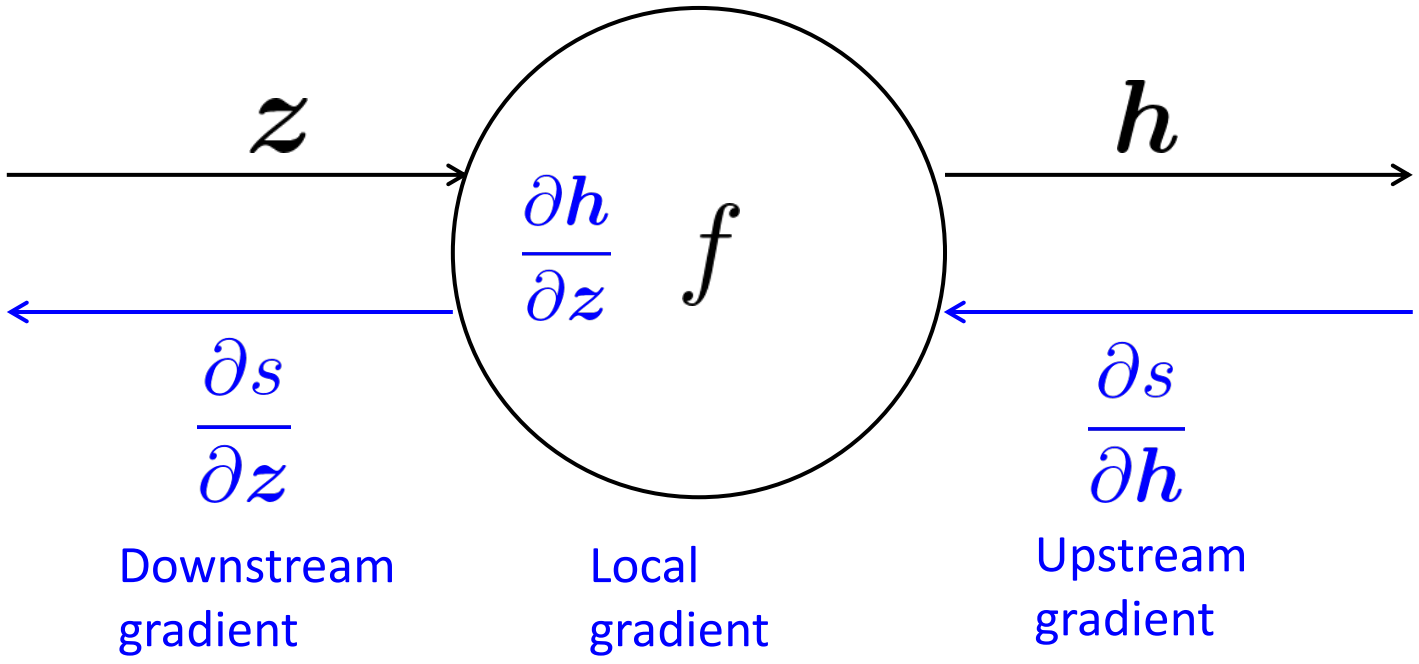
$$\boldsymbol{h} = f(\boldsymbol{z})$$



Backpropagation: Single Node

- Each node has a **local gradient**
  - The gradient of its output with respect to its input

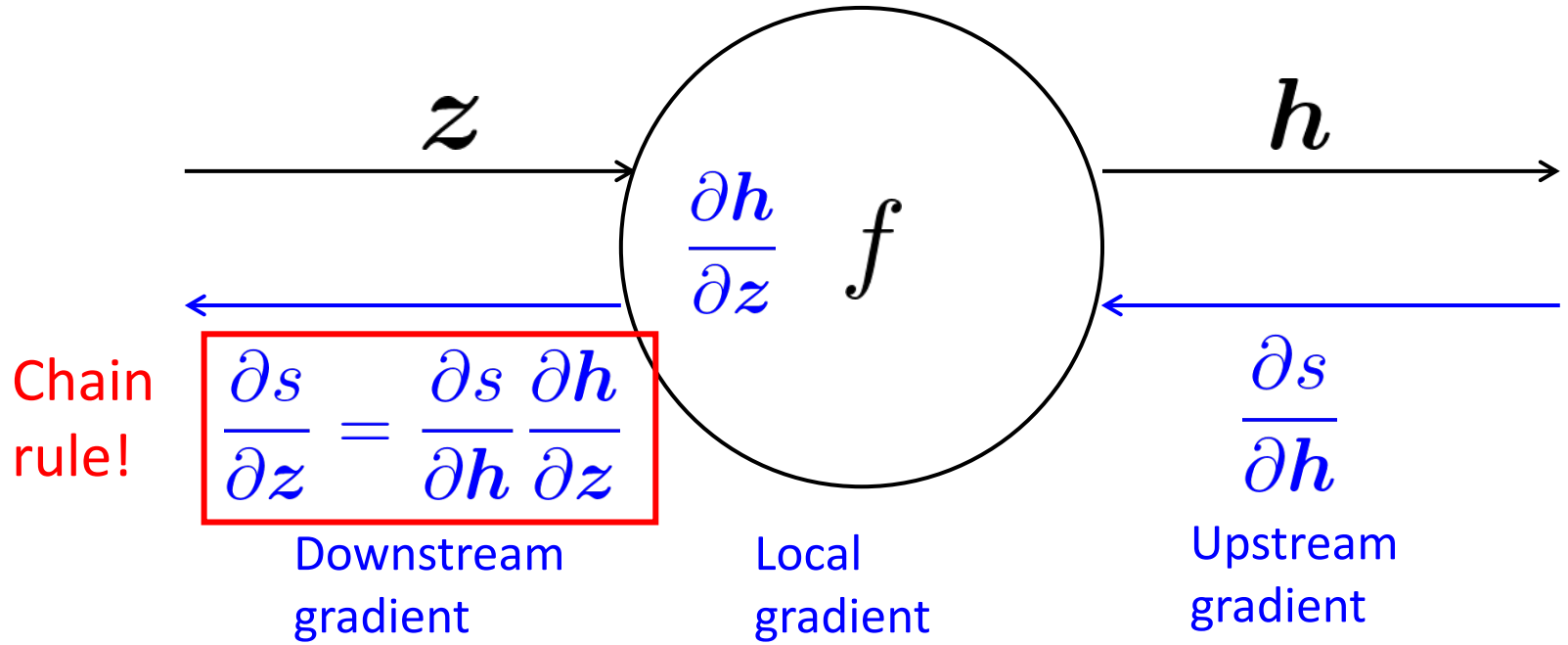
$$\boldsymbol{h} = f(\boldsymbol{z})$$



Backpropagation: Single Node

- Each node has a **local gradient**
  - The gradient of its output with respect to its input

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

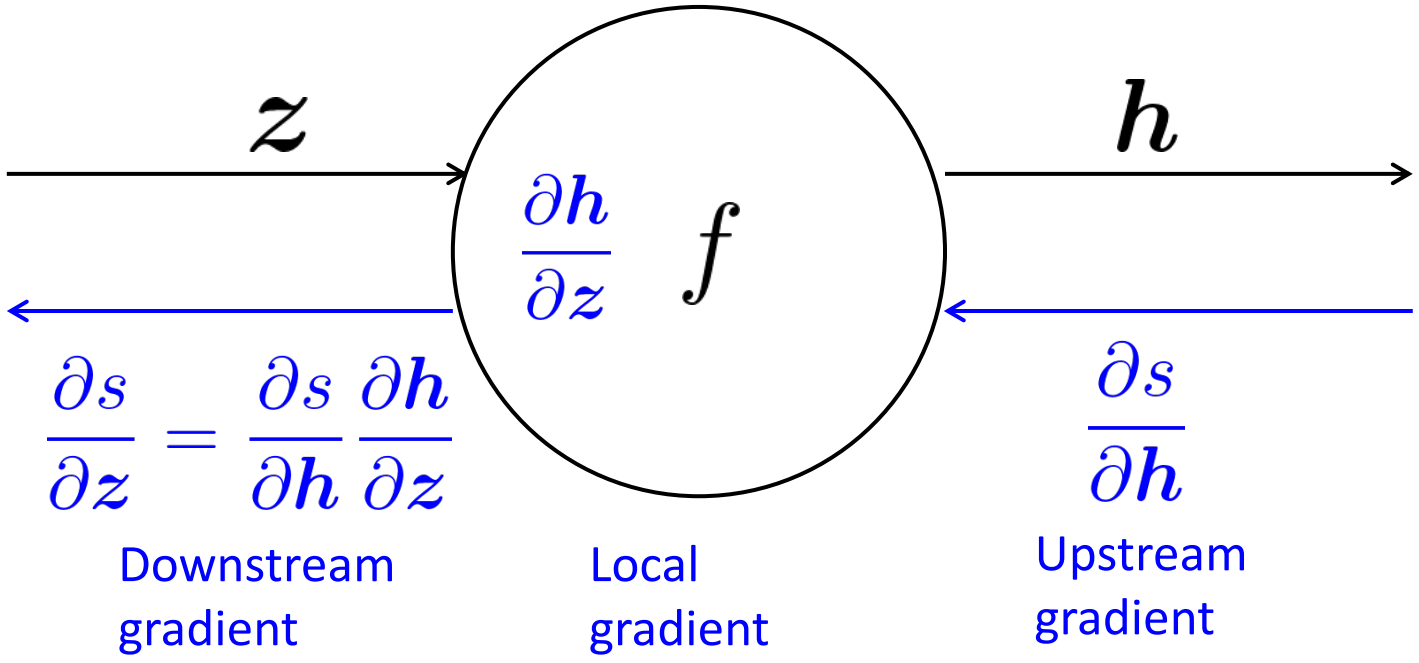


Backpropagation: Single Node

- Each node has a **local gradient**
  - The gradient of its output with respect to its input

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

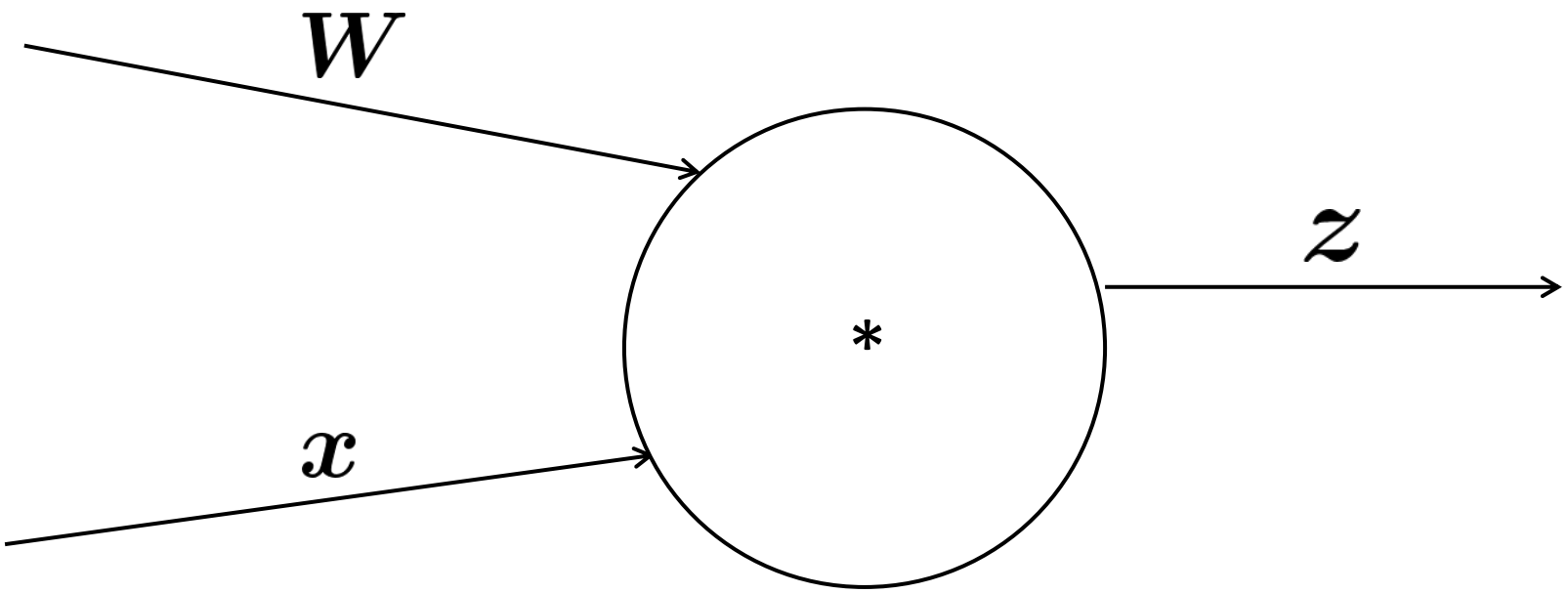
- [downstream gradient] = [upstream gradient] x [local gradient]



Backpropagation: Single Node

- What about nodes with multiple inputs?

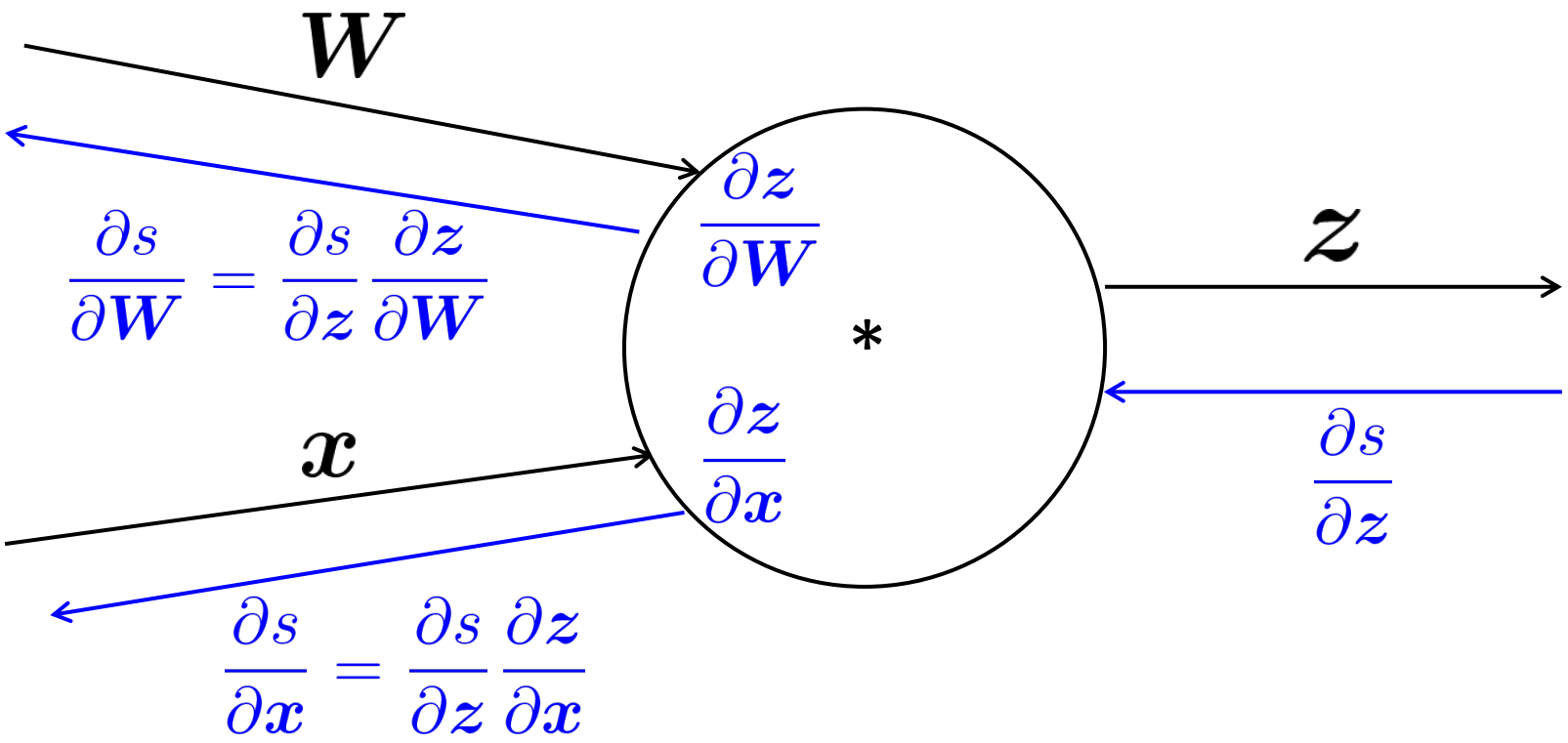
$$z = Wx$$



Backpropagation: Single Node

- Multiple inputs → multiple local gradients

$$z = Wx$$



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

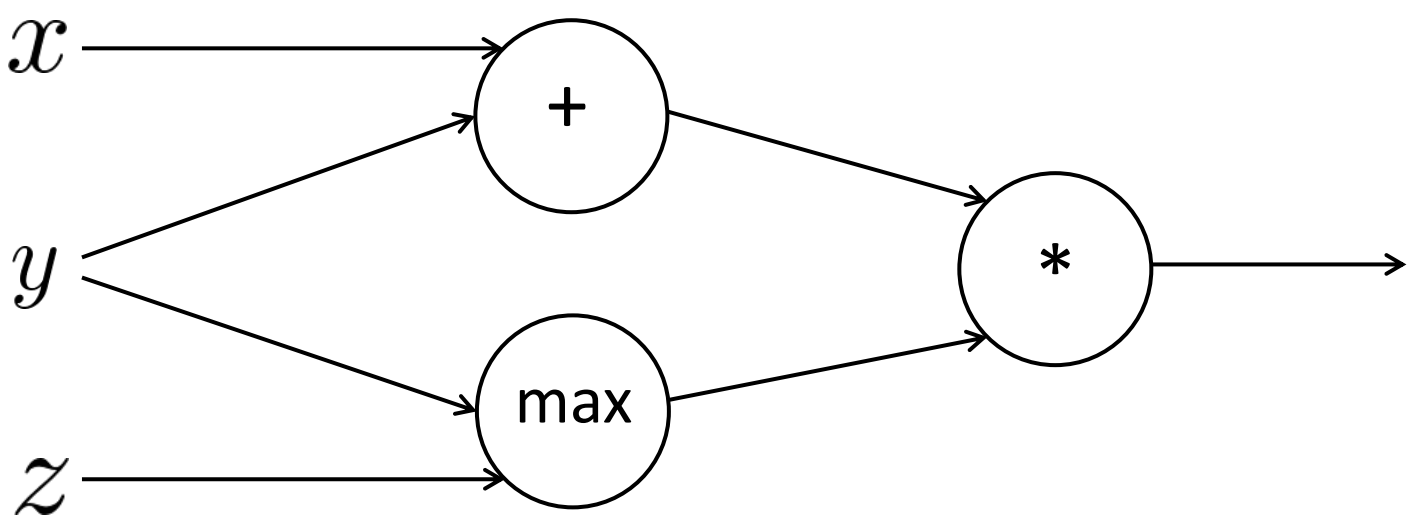
$$f(x, y, z) = (x + y) \max(y, z)$$
$$x = 1, y = 2, z = 0$$

Forward prop steps

$$a = x + y$$

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

$$f = ab$$



An Example

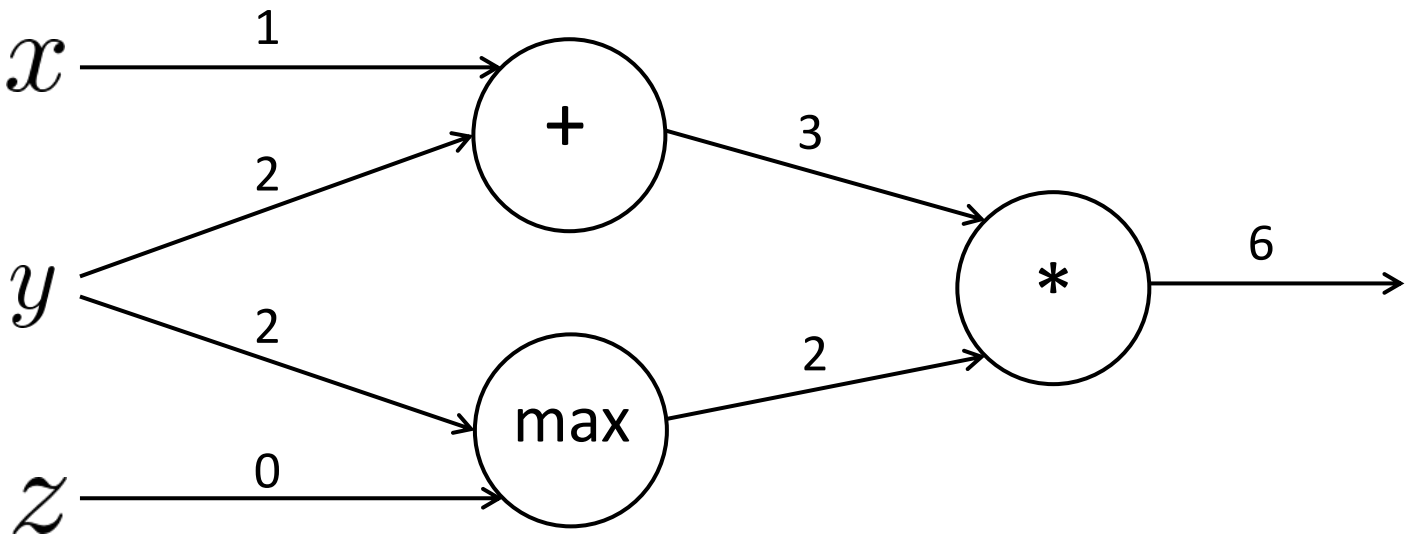
$$f(x, y, z) = (x + y)\max(y, z)$$
$$x = 1, y = 2, z = 0$$

Forward prop steps

$$a = x + y$$

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

$$f = ab$$



An Example

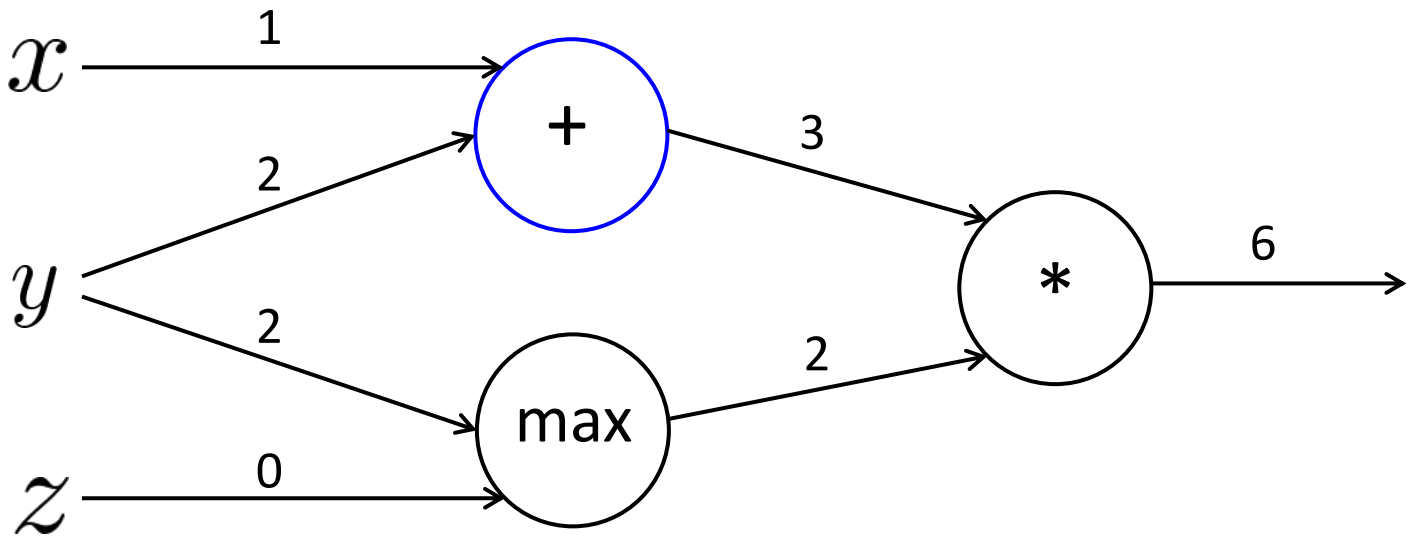
$$f(x, y, z) = (x + y) \max(y, z)$$
$$x = 1, y = 2, z = 0$$

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

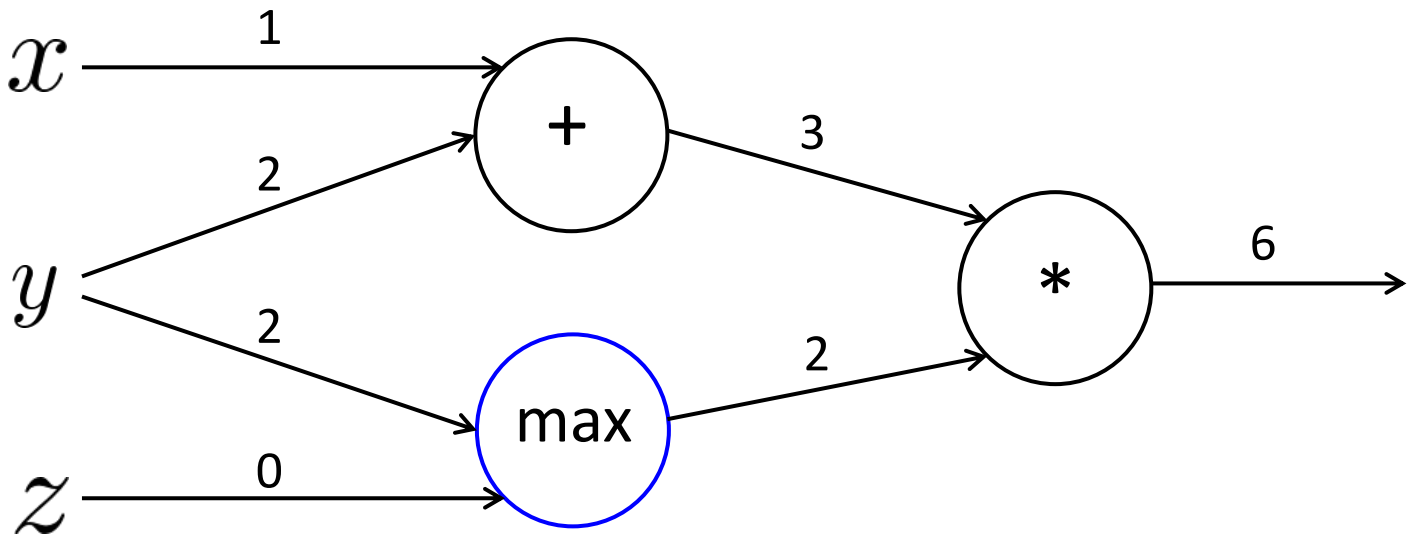
$$f(x, y, z) = (x + y) \max(y, z)$$
$$x = 1, y = 2, z = 0$$

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

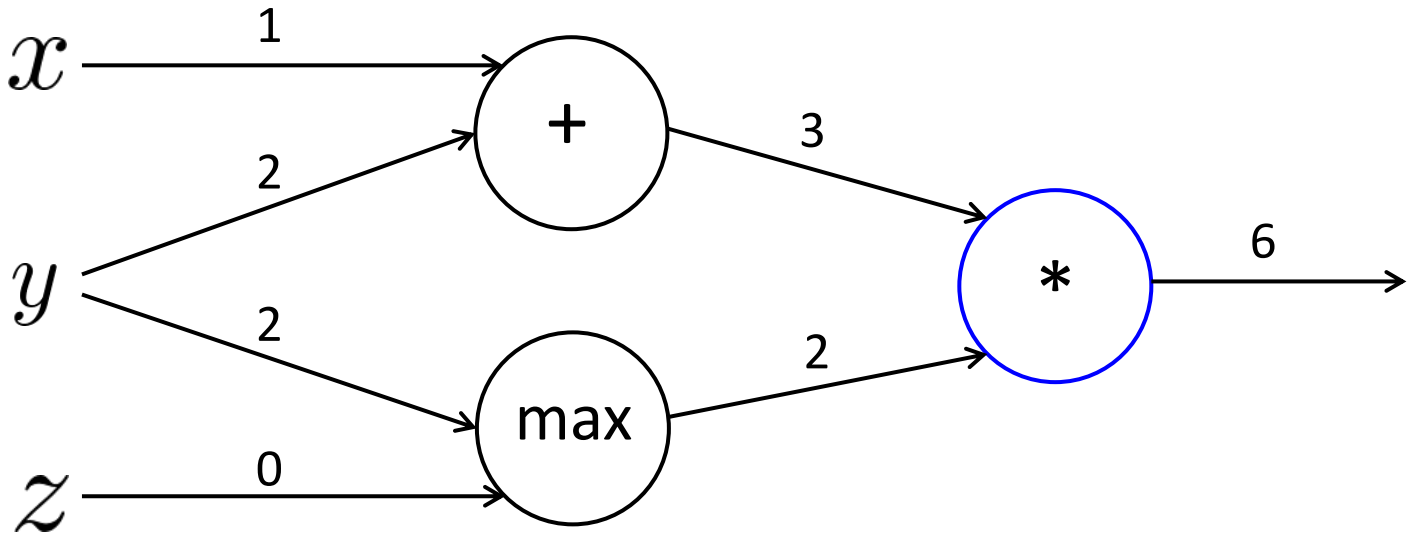
$$f(x, y, z) = (x + y) \max(y, z)$$
$$x = 1, y = 2, z = 0$$

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

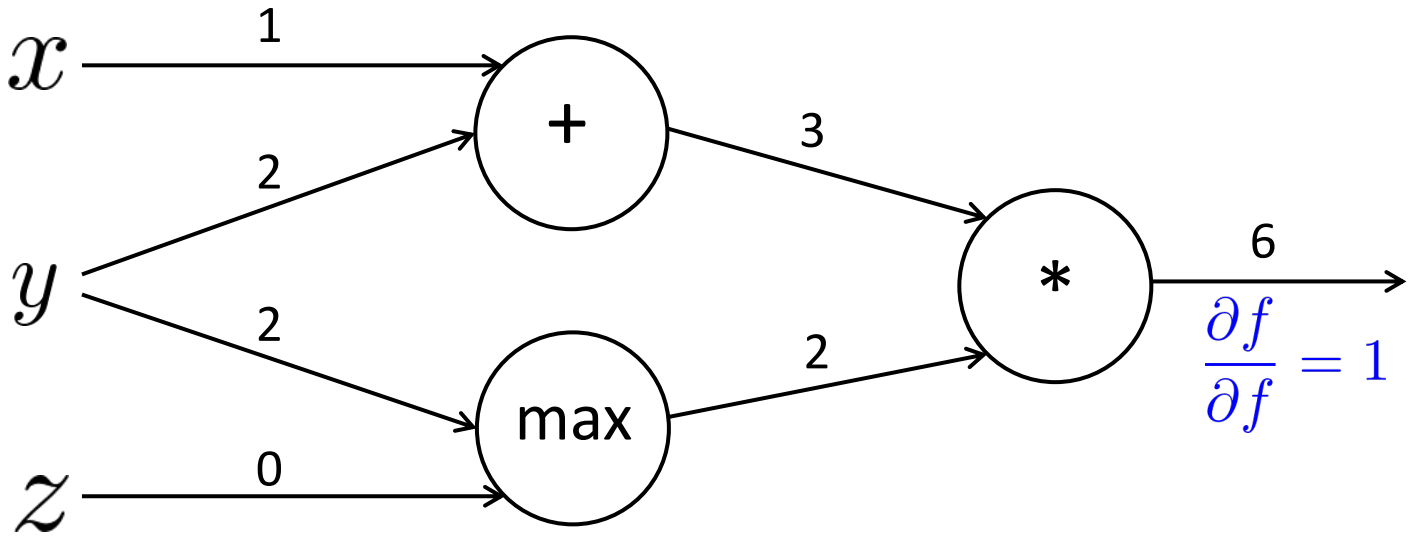
$$f(x, y, z) = (x + y) \max(y, z)$$
$$x = 1, y = 2, z = 0$$

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

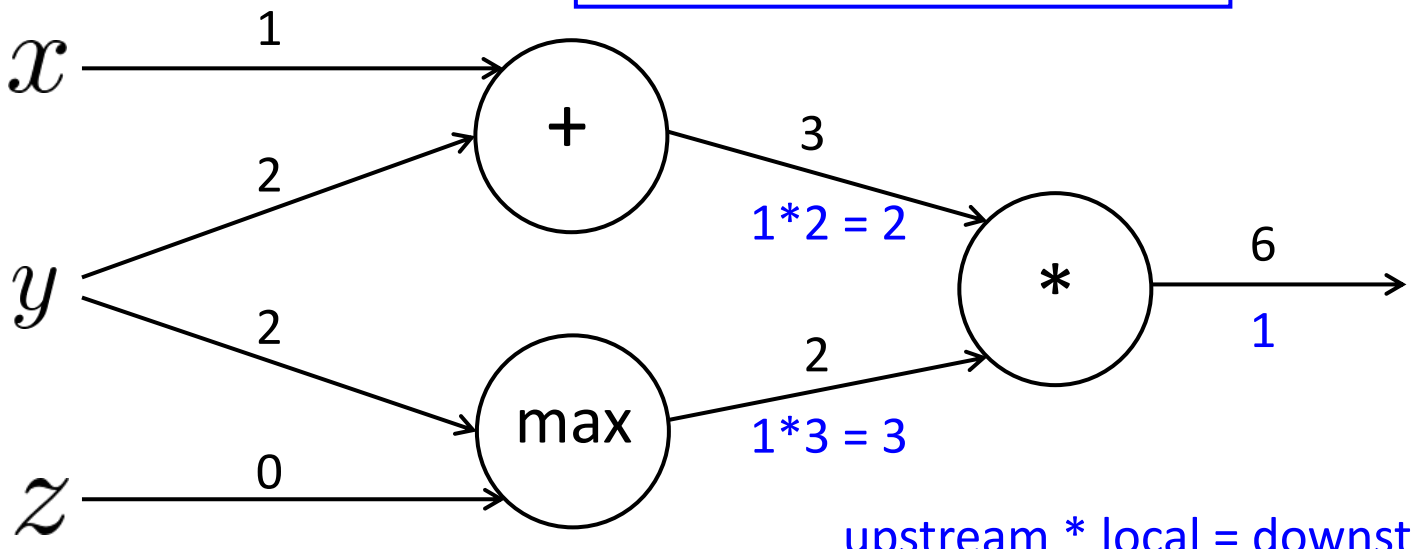
$$f(x, y, z) = (x + y) \max(y, z)$$
$$x = 1, y = 2, z = 0$$

Forward prop steps

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

Local gradients

$$\frac{\partial a}{\partial x} = 1 \quad \frac{\partial a}{\partial y} = 1$$
$$\frac{\partial b}{\partial y} = \mathbf{1}(y > z) = 1 \quad \frac{\partial b}{\partial z} = \mathbf{1}(z > y) = 0$$
$$\frac{\partial f}{\partial a} = b = 2 \quad \frac{\partial f}{\partial b} = a = 3$$



upstream * local = downstream

An Example

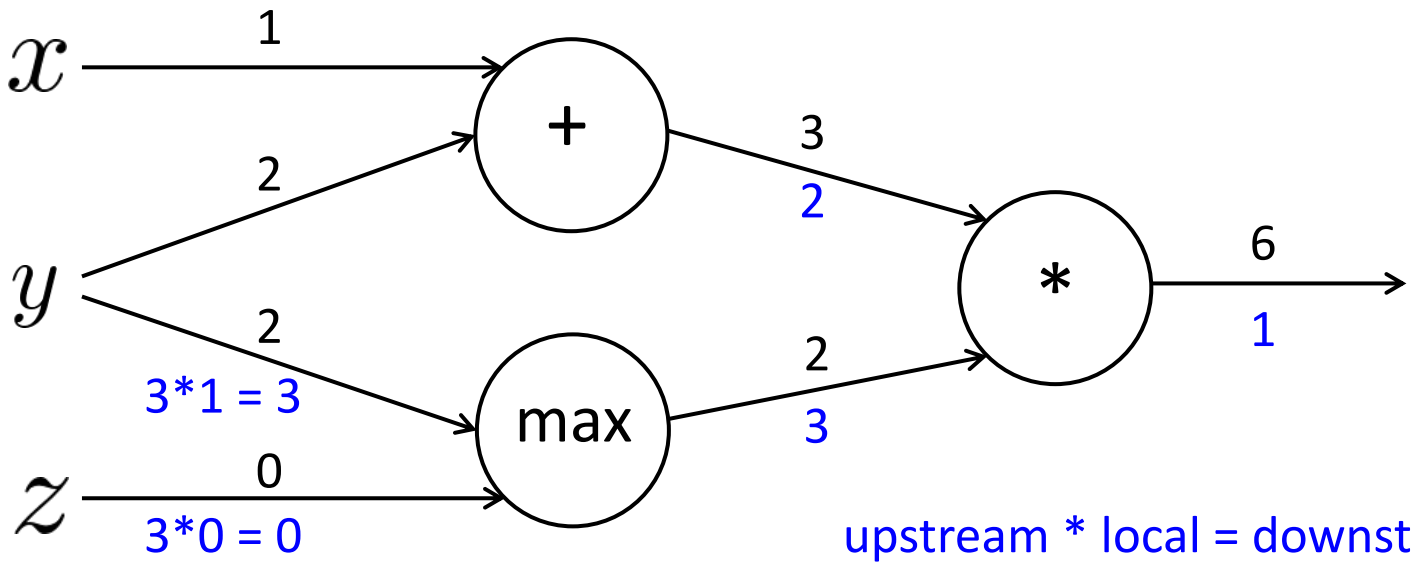
$$f(x, y, z) = (x + y) \max(y, z)$$
$$x = 1, y = 2, z = 0$$

Forward prop steps

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

Local gradients

$$\frac{\partial a}{\partial x} = 1 \quad \frac{\partial a}{\partial y} = 1$$
$$\frac{\partial b}{\partial y} = \mathbf{1}(y > z) = 1 \quad \frac{\partial b}{\partial z} = \mathbf{1}(z > y) = 0$$
$$\frac{\partial f}{\partial a} = b = 2 \quad \frac{\partial f}{\partial b} = a = 3$$



upstream * local = downstream

An Example

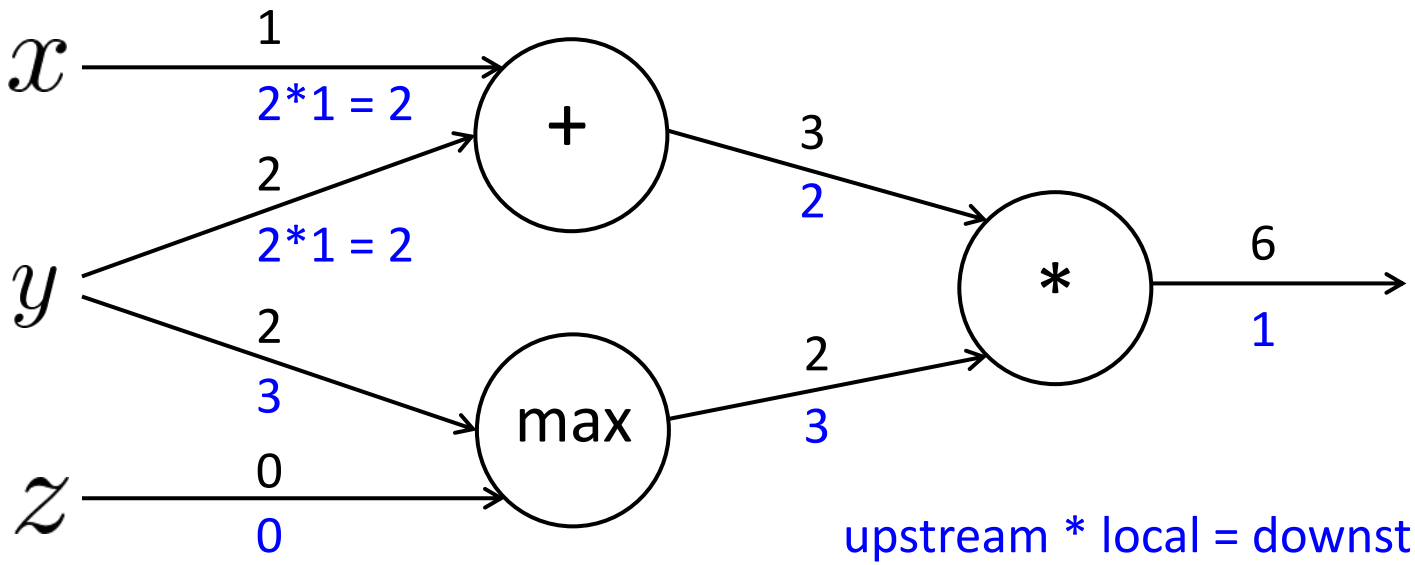
$$f(x, y, z) = (x + y) \max(y, z)$$
$$x = 1, y = 2, z = 0$$

Forward prop steps

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

Local gradients

$$\frac{\partial a}{\partial x} = 1 \quad \frac{\partial a}{\partial y} = 1$$
$$\frac{\partial b}{\partial y} = \mathbf{1}(y > z) = 1 \quad \frac{\partial b}{\partial z} = \mathbf{1}(z > y) = 0$$
$$\frac{\partial f}{\partial a} = b = 2 \quad \frac{\partial f}{\partial b} = a = 3$$



upstream * local = downstream

An Example

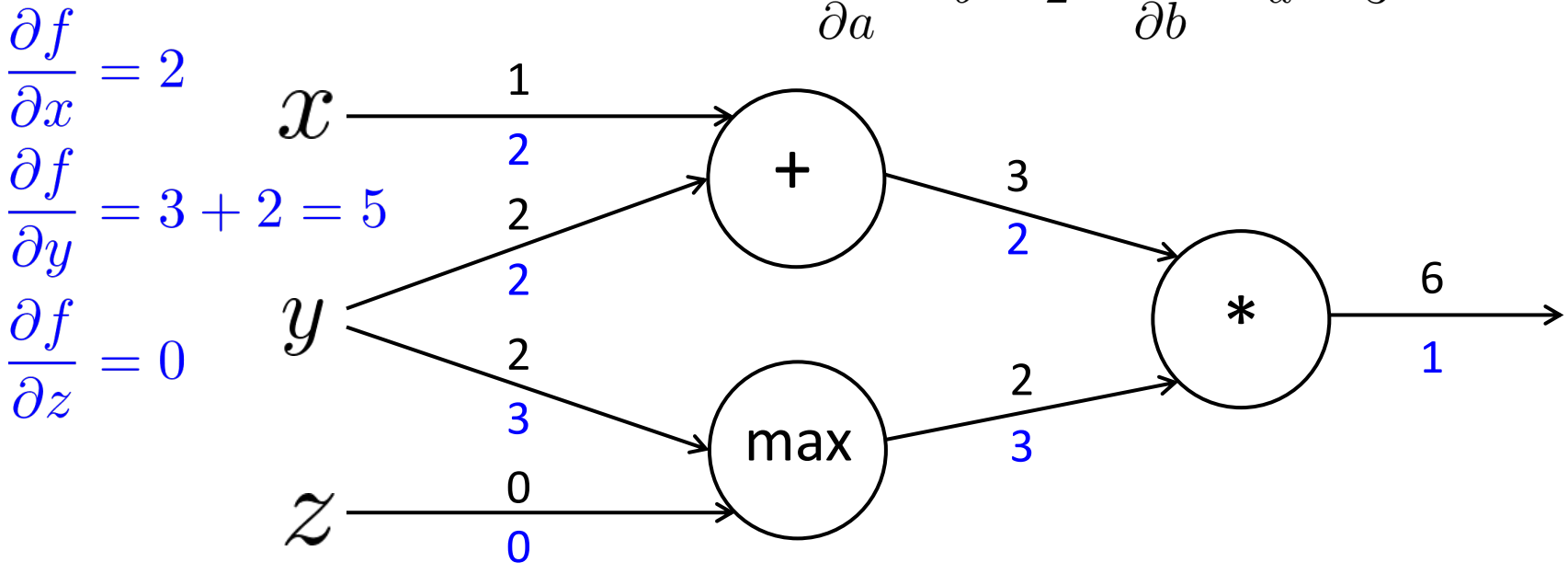
$$f(x, y, z) = (x + y) \max(y, z)$$
$$x = 1, y = 2, z = 0$$

Forward prop steps

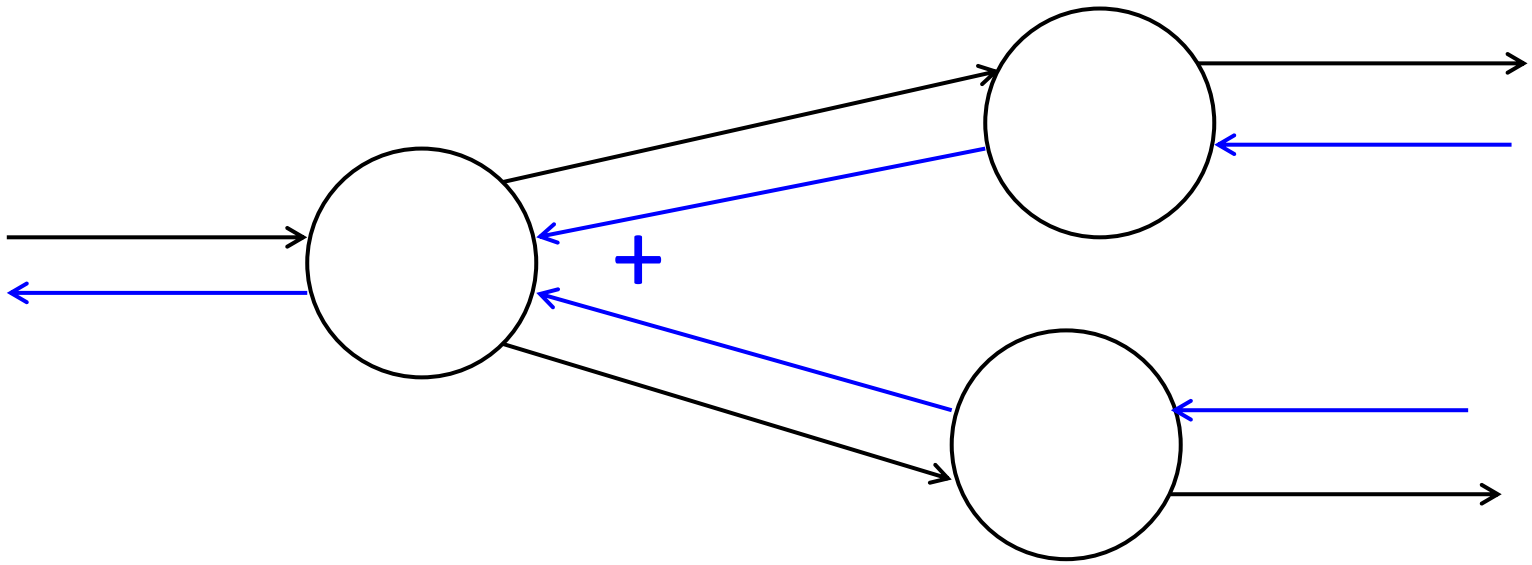
$$a = x + y$$
$$b = \max(y, z)$$
$$f = ab$$

Local gradients

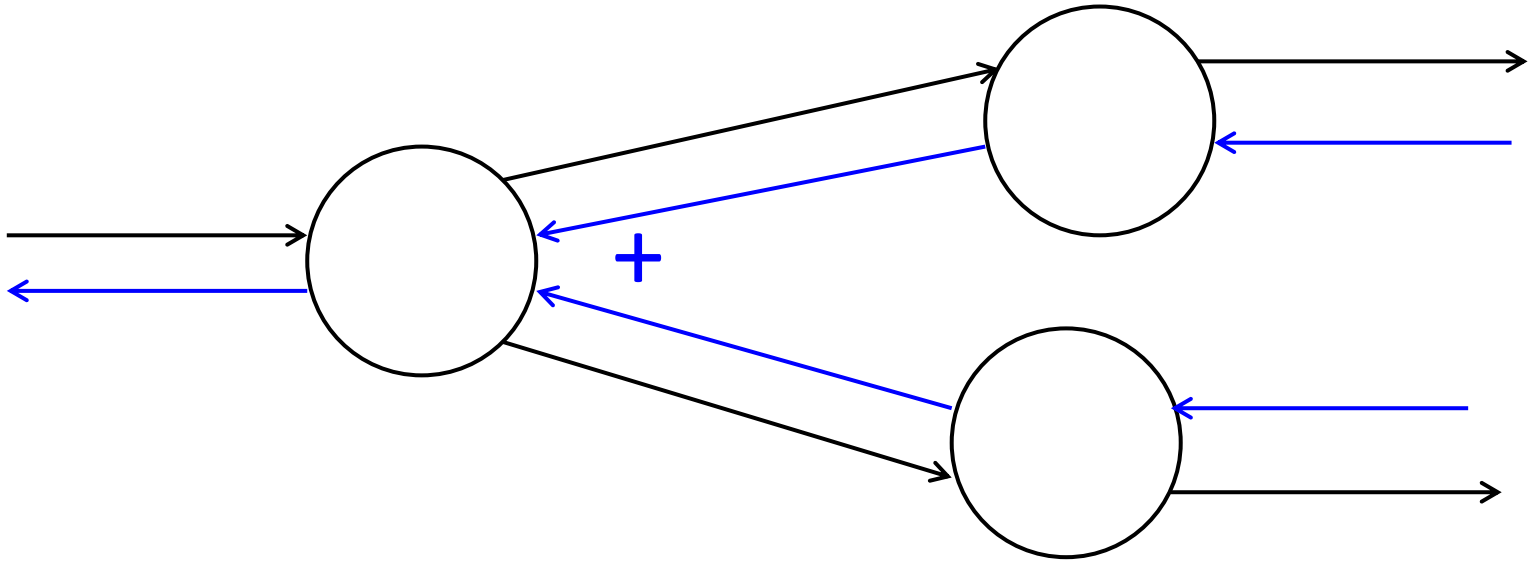
$$\frac{\partial a}{\partial x} = 1 \quad \frac{\partial a}{\partial y} = 1$$
$$\frac{\partial b}{\partial y} = \mathbf{1}(y > z) = 1 \quad \frac{\partial b}{\partial z} = \mathbf{1}(z > y) = 0$$
$$\frac{\partial f}{\partial a} = b = 2 \quad \frac{\partial f}{\partial b} = a = 3$$



Gradients sum at outward branches



Gradients sum at outward branches



$$a = x + y$$

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

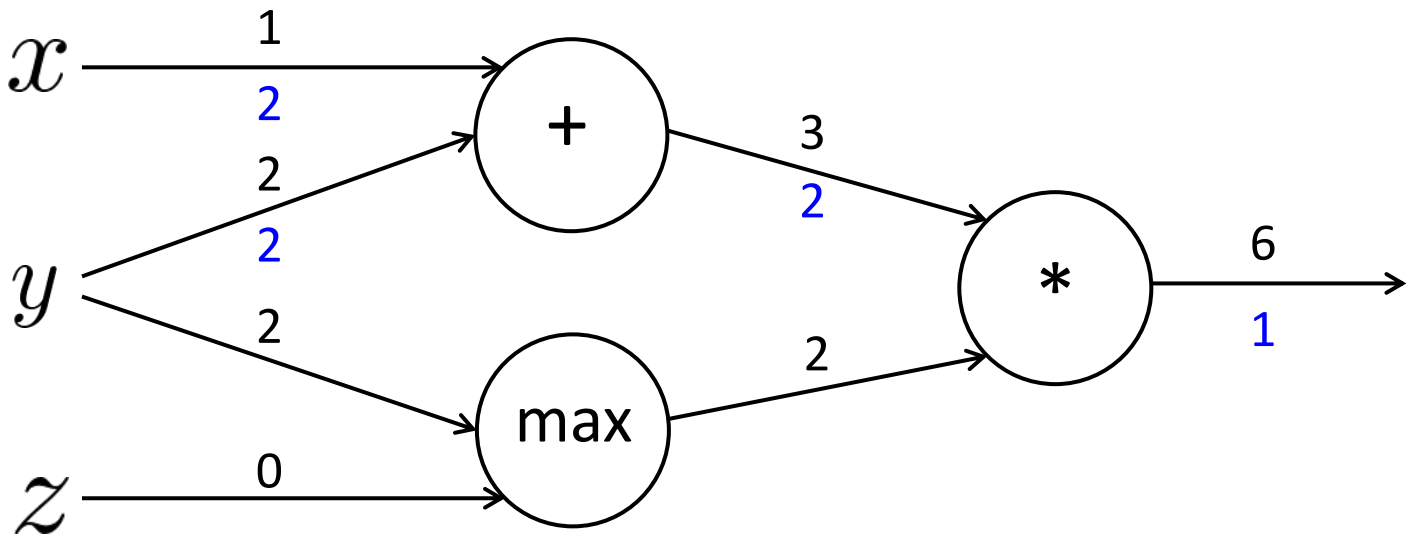
$$f = ab$$

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

Node Intuitions

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

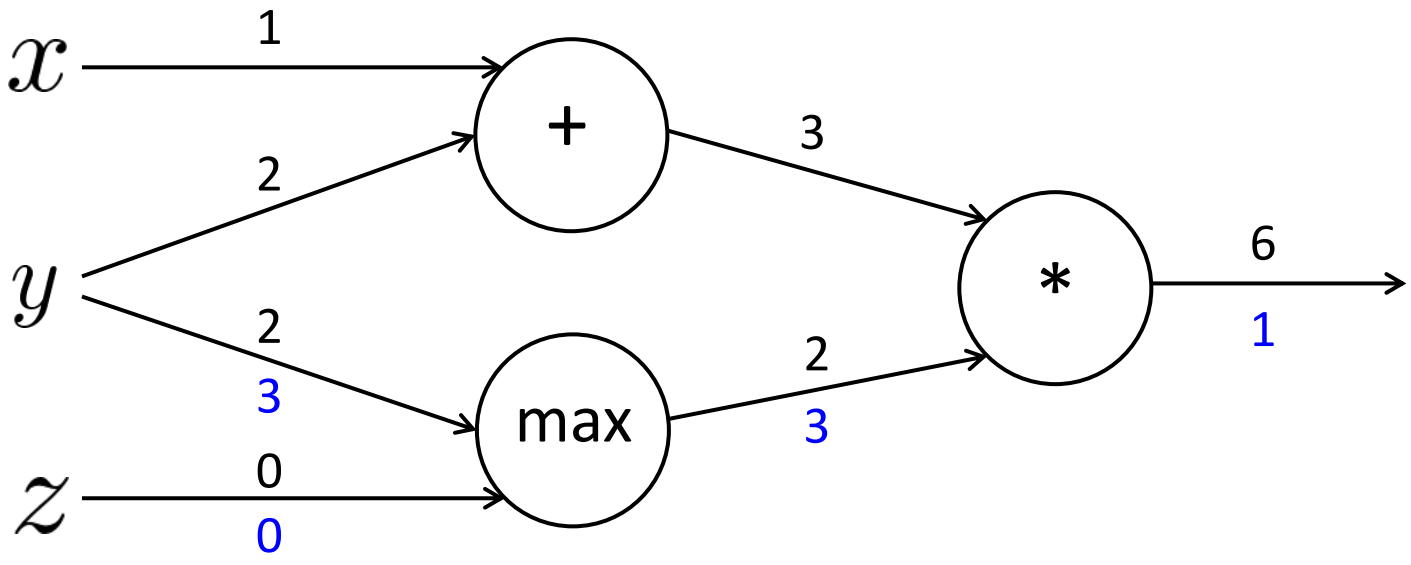
- + "distributes" the upstream gradient to each summand



Node Intuitions

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

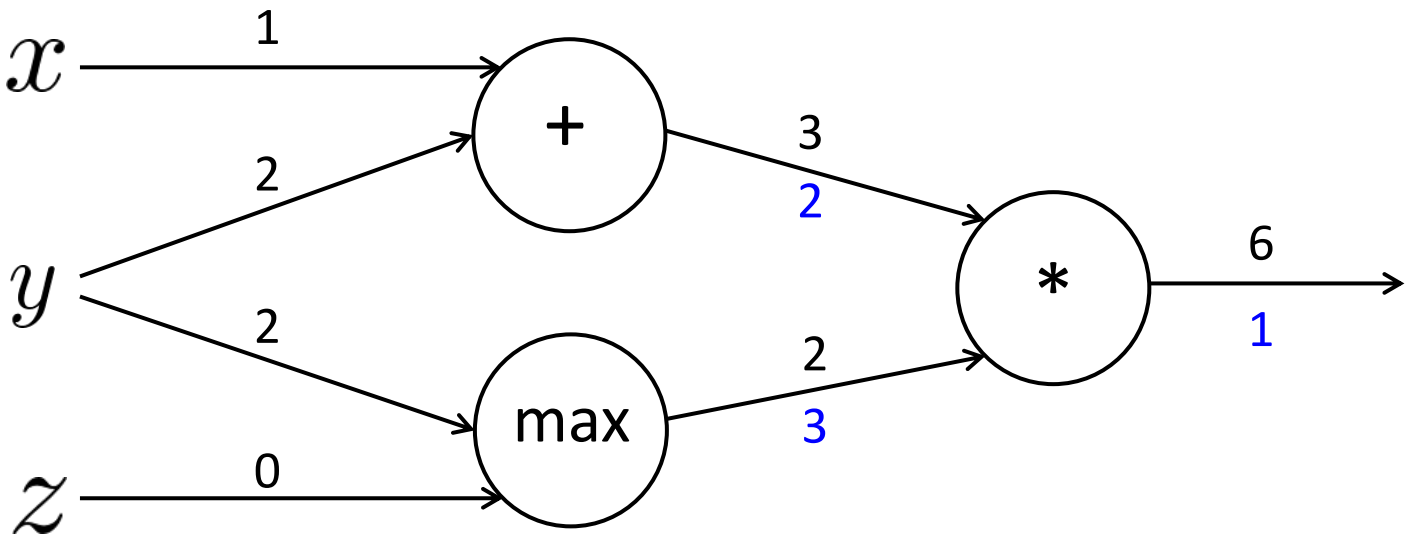
- + "distributes" the upstream gradient to each summand
- max "routes" the upstream gradient



Node Intuitions

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

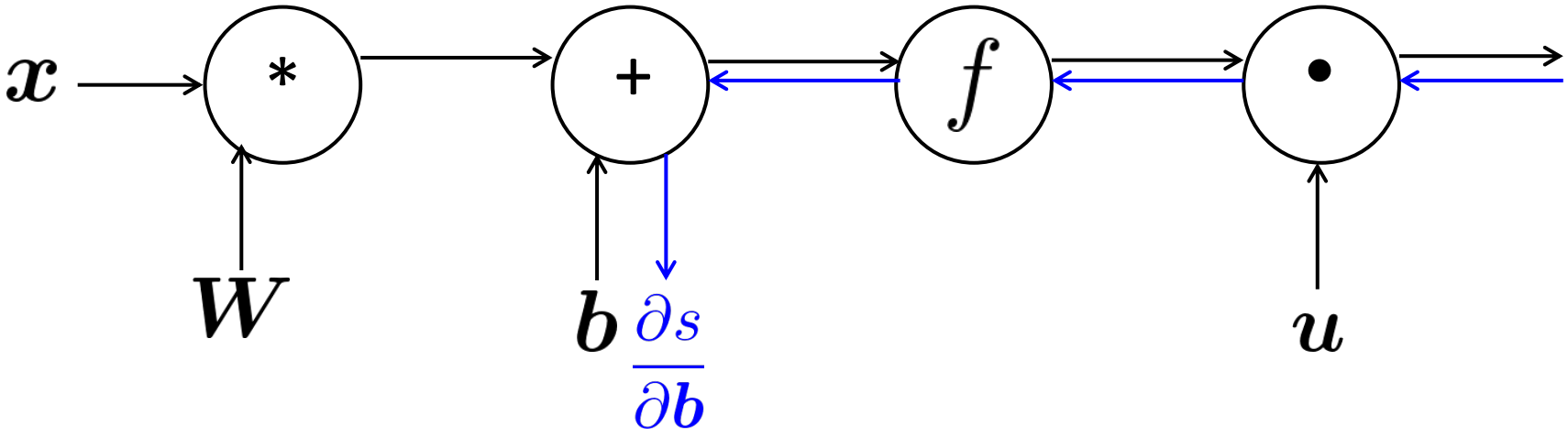
- + "distributes" the upstream gradient
- max "routes" the upstream gradient
- * "switches" the upstream gradient



Efficiency: compute all gradients at once

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

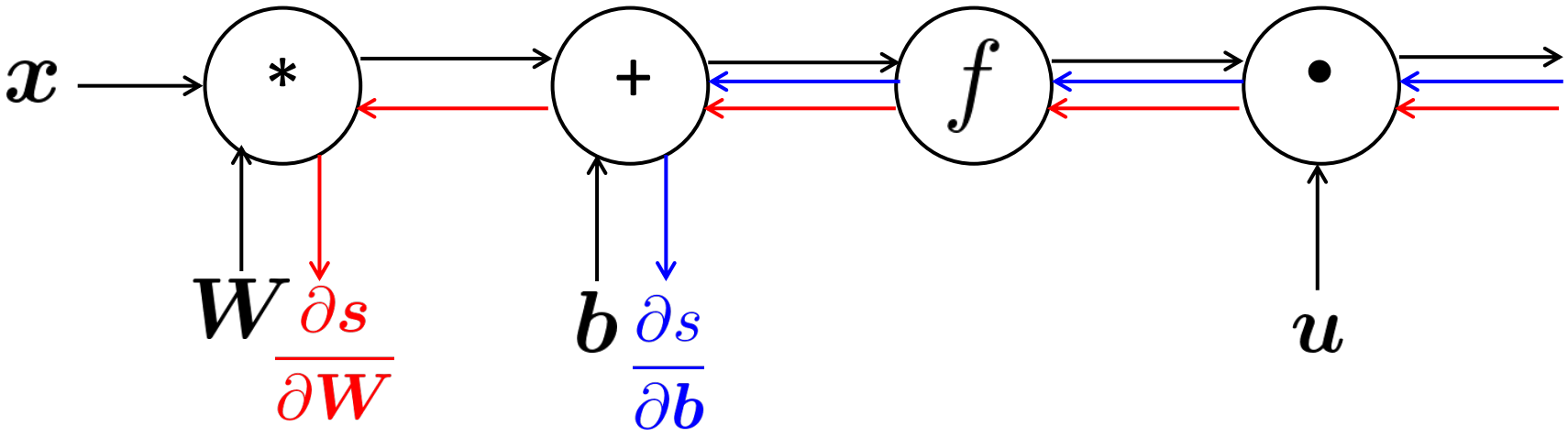
$$s = \boldsymbol{u}^T \boldsymbol{h}$$
$$\boldsymbol{h} = f(\boldsymbol{z})$$
$$\boldsymbol{z} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}$$
$$\boldsymbol{x} \quad \text{(input)}$$



Efficiency: compute all gradients at once

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

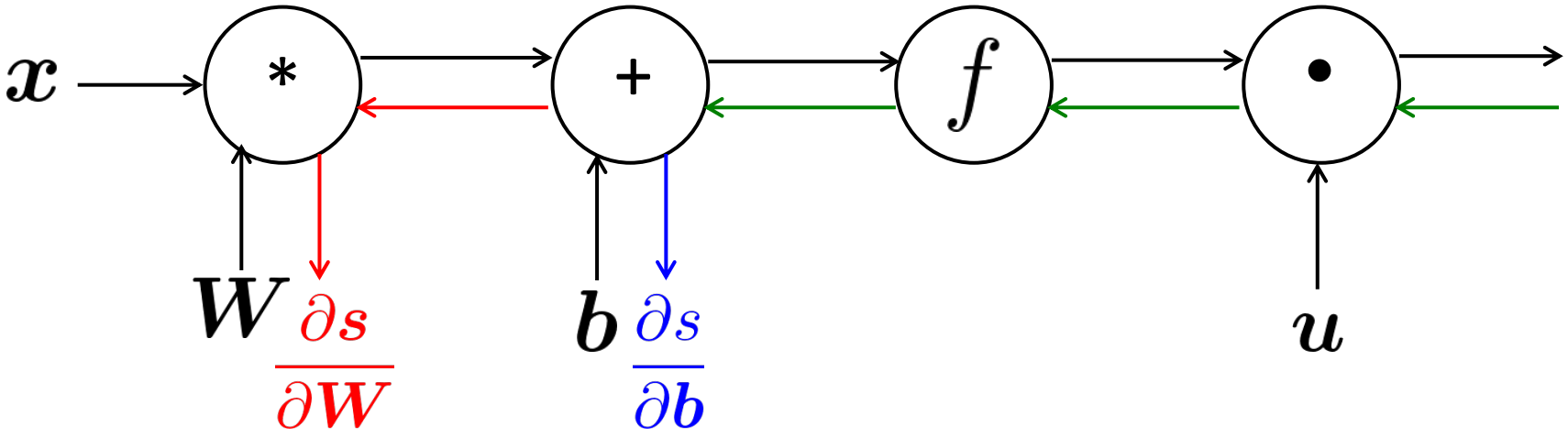
$$s = \boldsymbol{u}^T \boldsymbol{h}$$
$$\boldsymbol{h} = f(\boldsymbol{z})$$
$$\boldsymbol{z} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}$$
$$\boldsymbol{x} \quad \text{(input)}$$



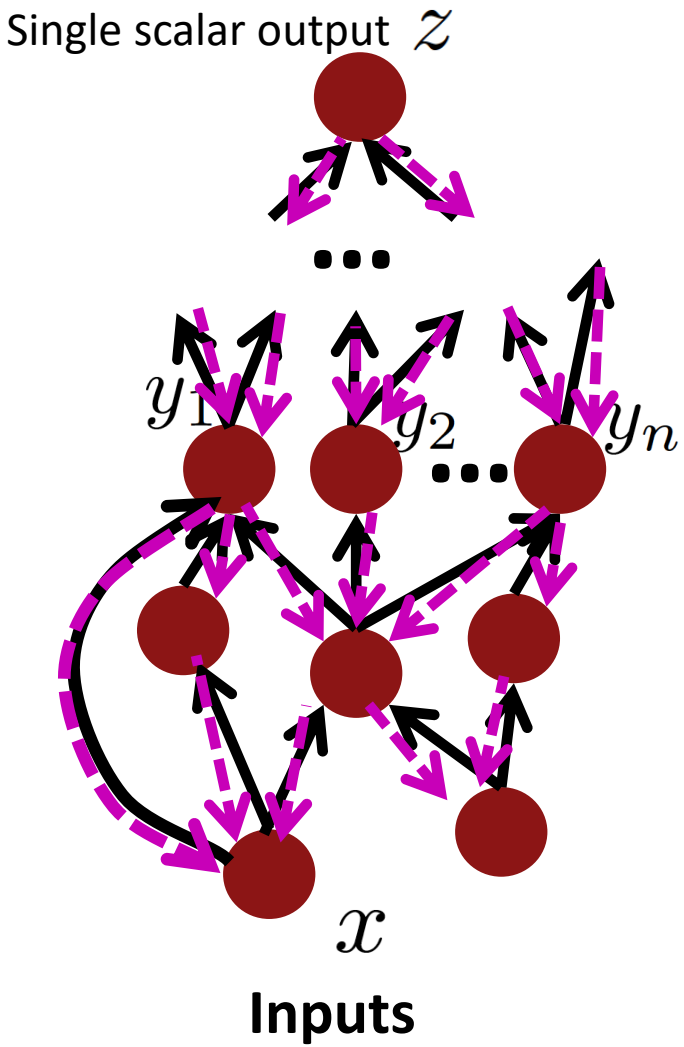
Efficiency: compute all gradients at once

- 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



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

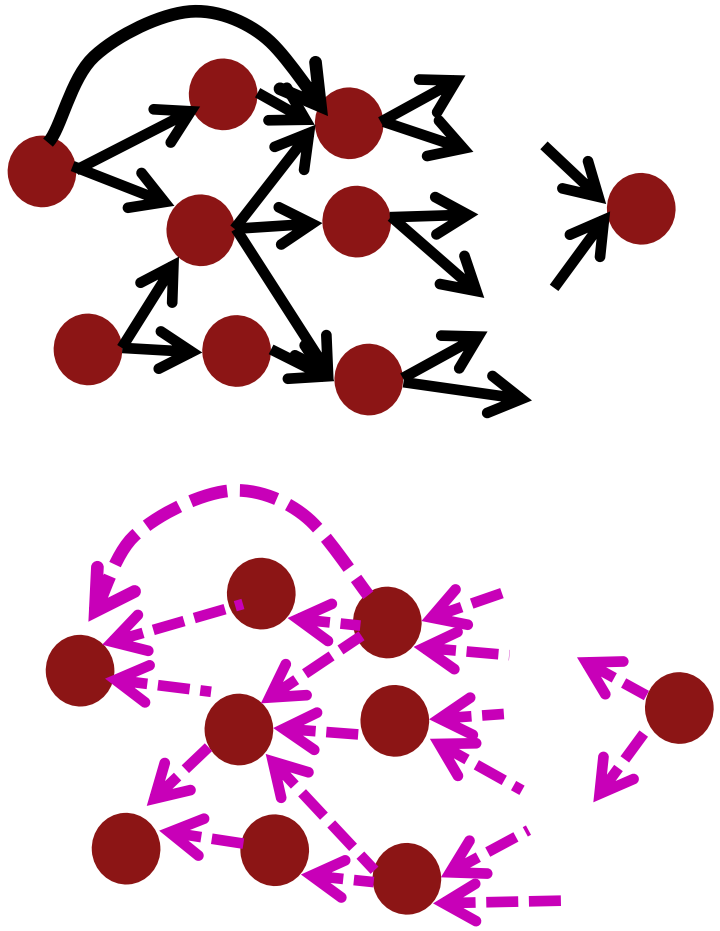
$\{y_1, y_2, \ldots y_n\}$ = successors of x

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

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

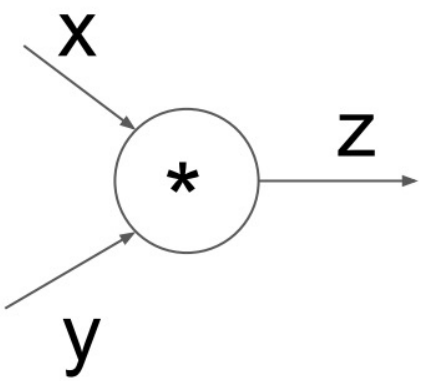


- The gradient computation can be automatically inferred from the symbolic expression of the fprop
- Each node type needs to know how to compute its output and how to compute the gradient wrt its inputs given the gradient wrt its output
- Modern DL frameworks (Tensorflow, PyTorch, etc.) do backpropagation 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



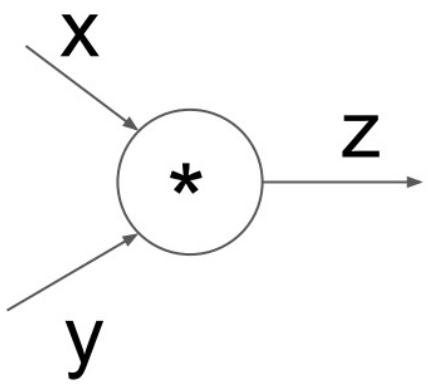
(x,y,z are scalars)

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

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

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

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]
```

Manual Gradient checking: Numeric Gradient

- For small *h* (≈ 1e-4),

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h}$$

- Easy to implement correctly
- But approximate and **very** slow:
  - You have to recompute *f* for **every parameter** of our model
- Useful for checking your implementation
  - In the old days, we hand-wrote everything, doing this everywhere was the key test
  - Now much less needed; you can use it to check layers are correctly implemented

Summary

We've mastered the core technology of neural nets! 🎉🎉🎉

- **Backpropagation:** recursively (and hence efficiently) 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!**
  - Come to the PyTorch introduction this Friday!

- 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 out of the box
  - 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