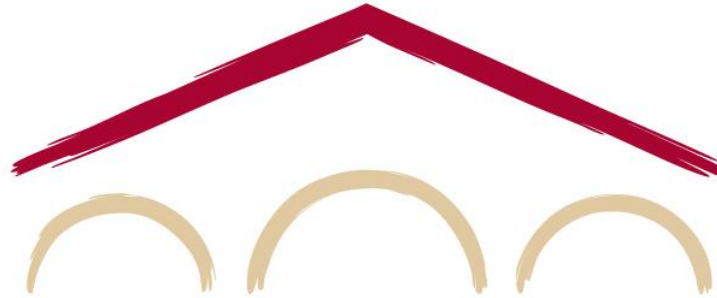


# Natural Language Processing with Deep Learning

## CS224N/Ling284



Diyi Yang

Lecture 3: Neural Network Foundations

# Lecture Plan

## Lecture 2: Neural Network Foundations

1. Course logistics (3 mins) + Word2vec evaluation (7 mins)
2. Introducing neural networks (10 mins)
3. Matrix calculus (25 mins)
4. Backpropagation (35 mins)

Key Goal: the mathematics and practical implementation of how neural networks are trained by backpropagation

# 1. Course Logistics

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

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!

**PyTorch tutorial:** 1:30-2:20pm this Friday NVIDIA Auditorium

A great chance to get an intro to PyTorch, a key deep learning package!

**Poster session:** March 16, 12:15-3:15pm, AOERC. In-person presence required for in-person students; other exceptions see Ed forum post (due week 3)

# Recap from Lecture 2: How to evaluate word vectors?

- Related to general evaluation in NLP: Intrinsic vs. extrinsic
- Intrinsic:
  - Evaluation on a specific/intermediate subtask
  - Fast to compute
  - Helps to understand that system
  - Not clear if it's helpful unless correlation to real task is established
- Extrinsic:
  - Evaluation on a real task
  - Can take a long time to compute accuracy
  - Unclear if the subsystem is the problem or its interaction or other subsystems
  - If replacing exactly one subsystem with another improves accuracy → Winning!

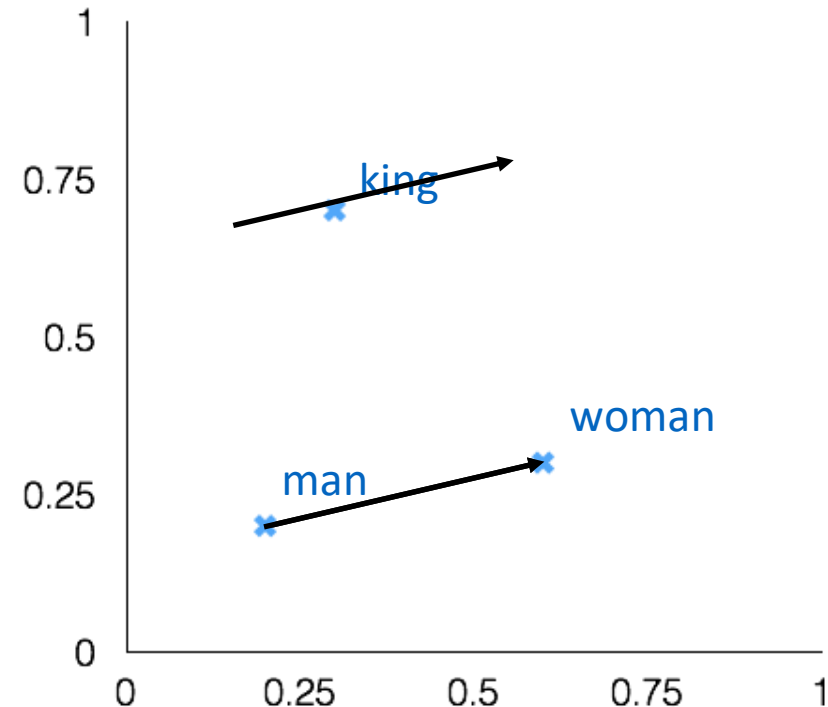
# Intrinsic word vector evaluation

- Word Vector Analogies

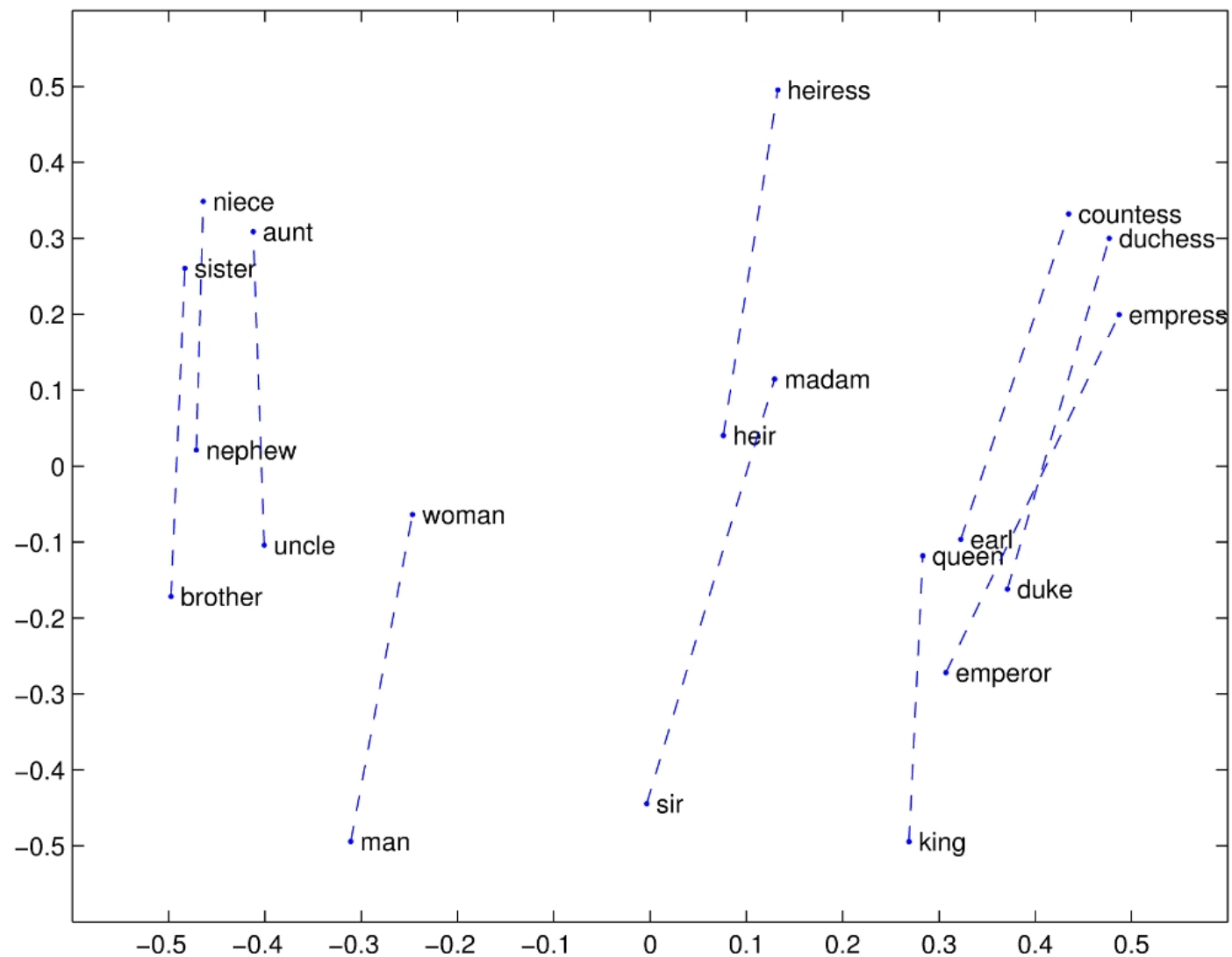
a:b :: c:?  
man:woman :: king:?

$$d = \arg \max_i \frac{(x_b - x_a + x_c)^T x_i}{\|x_b - x_a + x_c\|}$$

- Evaluate word vectors by how well their cosine distance after addition captures intuitive semantic and syntactic analogy questions
- Discarding the input words from the search (!)
- Problem: What if the information is there but not linear?



# GloVe Visualization



# Meaning similarity: Another intrinsic word vector evaluation

- Word vector distances and their correlation with human judgments
- Example dataset: WordSim353

<http://www.cs.technion.ac.il/~gabr/resources/data/wordsim353/>

Word 1	Word 2	Human (mean)
tiger	cat	7.35
tiger	tiger	10
book	paper	7.46
computer	internet	7.58
plane	car	5.77
professor	doctor	6.62
stock	phone	1.62
stock	CD	1.31
stock	jaguar	0.92

# Correlation evaluation

- Word vector distances and their correlation with human judgments

Model	Size	WS353	MC	RG	SCWS	RW
SVD	6B	35.3	35.1	42.5	38.3	25.6
SVD-S	6B	56.5	71.5	71.0	53.6	34.7
SVD-L	6B	65.7	<u>72.7</u>	75.1	56.5	37.0
CBOW <sup>†</sup>	6B	57.2	65.6	68.2	57.0	32.5
SG <sup>†</sup>	6B	62.8	65.2	69.7	<u>58.1</u>	37.2
GloVe	6B	<u>65.8</u>	<u>72.7</u>	<u>77.8</u>	53.9	<u>38.1</u>
SVD-L	42B	74.0	76.4	74.1	58.3	39.9
GloVe	42B	<b><u>75.9</u></b>	<b><u>83.6</u></b>	<b><u>82.9</u></b>	<b><u>59.6</u></b>	<b><u>47.8</u></b>
CBOW*	100B	68.4	79.6	75.4	59.4	45.5



# Extrinsic word vector evaluation

- One example where good word vectors should help directly: **named entity recognition**: identifying references to a person, organization or location:  
**Chris Manning** lives in **Palo Alto**.

Model	Dev	Test	ACE	MUC7
Discrete	91.0	85.4	77.4	73.4
SVD	90.8	85.7	77.3	73.7
SVD-S	91.0	85.5	77.6	74.3
SVD-L	90.5	84.8	73.6	71.5
HPCA	92.6	<b>88.7</b>	81.7	80.7
HSMN	90.5	85.7	78.7	74.7
CW	92.2	87.4	81.7	80.2
CBOW	93.1	88.2	82.2	81.1
GloVe	<b>93.2</b>	88.3	<b>82.9</b>	<b>82.2</b>

## 2. Deep Learning Classification: Named Entity Recognition (NER)

- The task: **find** and **classify** names in text, by labeling word tokens, for example:

Last night , Paris Hilton wowed in a sequin gown .

PER PER

Samuel Quinn was arrested in the Hilton Hotel in Paris in April 1989 .

PER PER LOC LOC LOC DATE DATE

- Possible uses:
  - Tracking mentions of particular entities in documents
  - For question answering, answers are usually named entities
  - Relating sentiment analysis to the entity under discussion
- Often followed by Entity Linking/Canonicalization into a Knowledge Base such as Wikidata

# Simple NER: Window classification using binary logistic classifier

- **Idea:** classify each word in its context window of neighboring words
- Train logistic classifier on hand-labeled data to classify center word {yes/no} for each class based on a concatenation of word vectors in a window
  - Really, we usually use multi-class softmax, but we're trying to keep it simple 😊
- **Example:** Classify “Paris” as +/- location in context of sentence with window length 2:

the museums in Paris are amazing to see .

$$X_{\text{window}} = [x_{\text{museums}} \quad x_{\text{in}} \quad x_{\text{Paris}} \quad x_{\text{are}} \quad x_{\text{amazing}}]^T$$

- Resulting vector  $x_{\text{window}} = \boxed{x \in \mathbb{R}^{5d}}$
- To classify all words: run classifier for each class on the vector centered on each word in the sentence

# Classification review and notation

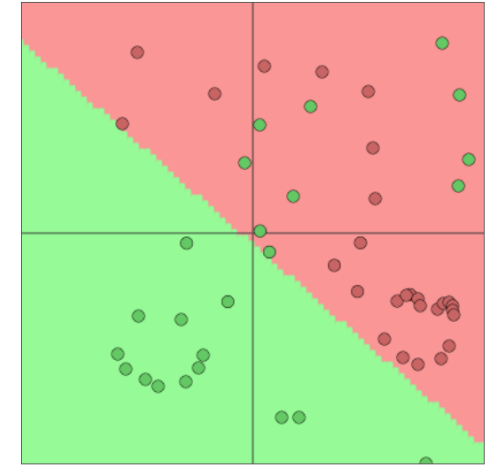
- Supervised learning: we have a **training dataset** consisting of **samples**

$$\{x_i, y_i\}_{i=1}^N$$

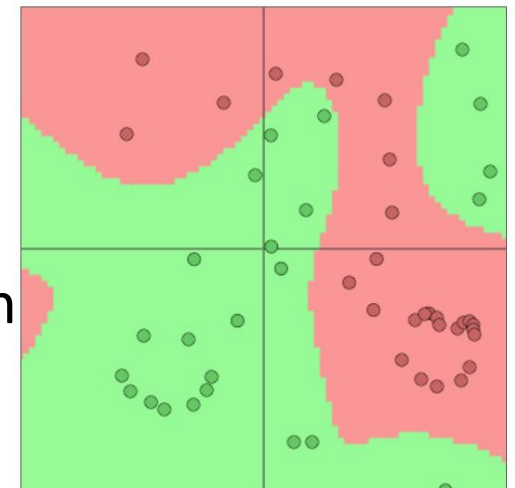
- $x_i$  are **inputs**, e.g., words (indices or vectors!), sentences, documents, etc.
  - Dimension  $d$
- $y_i$  are **labels** (one of  $C$  classes) we try to predict, for example:
  - classes: sentiment (+/−), named entities, buy/sell decision
  - other words
  - later: multi-word sequences

# Neural classification

- **Typical** ML/stats softmax classifier:  $p(y|x) = \frac{\exp(W_y \cdot x)}{\sum_{c=1}^C \exp(W_c \cdot x)}$
- Learned parameters  $\theta$  are just elements of  $W$  (not input representation  $x$ , which has sparse symbolic features)
- Classifier gives linear decision boundary, which can be limiting



- A **neural network classifier** differs in that:
  - We learn **both**  $W$  and **(distributed!)** representations for words
  - The word vectors  $x$  re-represent one-hot vectors, moving them around in an intermediate layer vector space, for easy classification with a (linear) softmax classifier
    - Conceptually, we have an embedding layer:  $x = Le$
  - We use deep networks—more layers—that let us re-represent and compose our data multiple times, giving a non-linear classifier



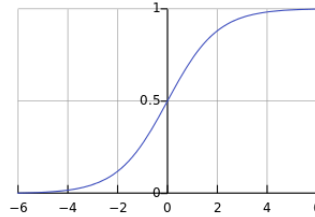
But typically, it is linear relative to the pre-final layer representation

# 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



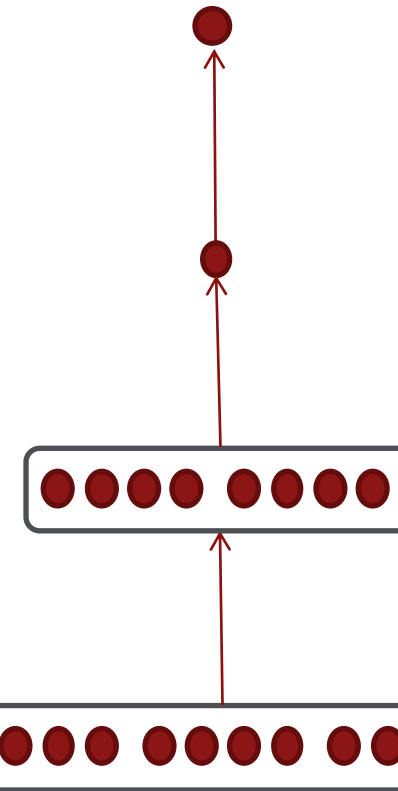
$$s = u^T h$$

$$h = f(Wx + b)$$

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

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

Embedding of  
1-hot words

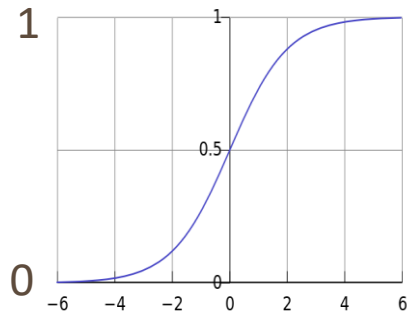


$f$  = Some element-wise non-linear function, e.g., logistic, tanh, ReLU

# Non-linearities, old and new

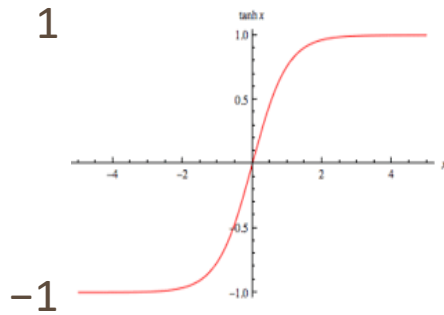
logistic (“sigmoid”)

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



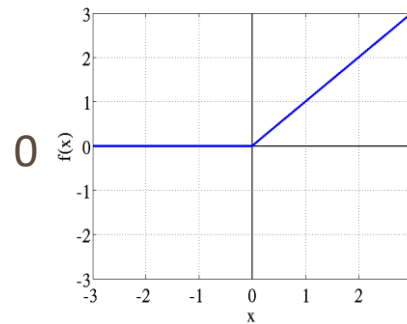
tanh

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

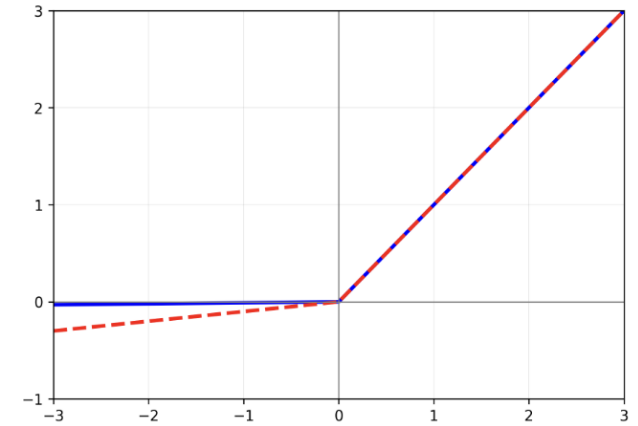


ReLU (Rectified Linear Unit)

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



Leaky ReLU / Parametric ReLU



tanh is just a rescaled and shifted sigmoid ( $2 \times$  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, often, 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

# Non-linearities, old and new

**GELU** (Gaussian error linear unit);  
frequently used with Transformers

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

**GLU (gated linear unit)** uses a gate/switch

$$\text{GLU}(x) = (xV + v) \otimes \sigma(xW + b)$$

**SwiGLU** (Swish-gated linear unit)

$$\text{SwiGLU}(x) = (xV + c) \otimes \text{Swish}_\beta(xW + b)$$

Frequently used, e.g., in Llama3, Qwen3

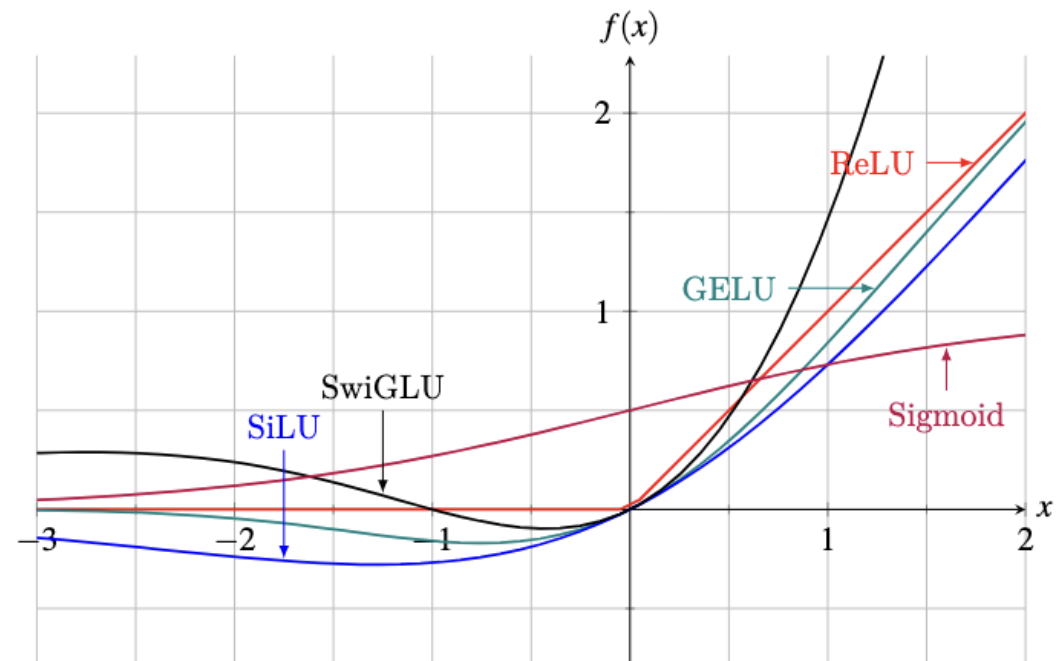
**SiLU**

Sigmoid linear unit

$$\text{SiLU}(x) = x \cdot \sigma(x)$$

**Swish**

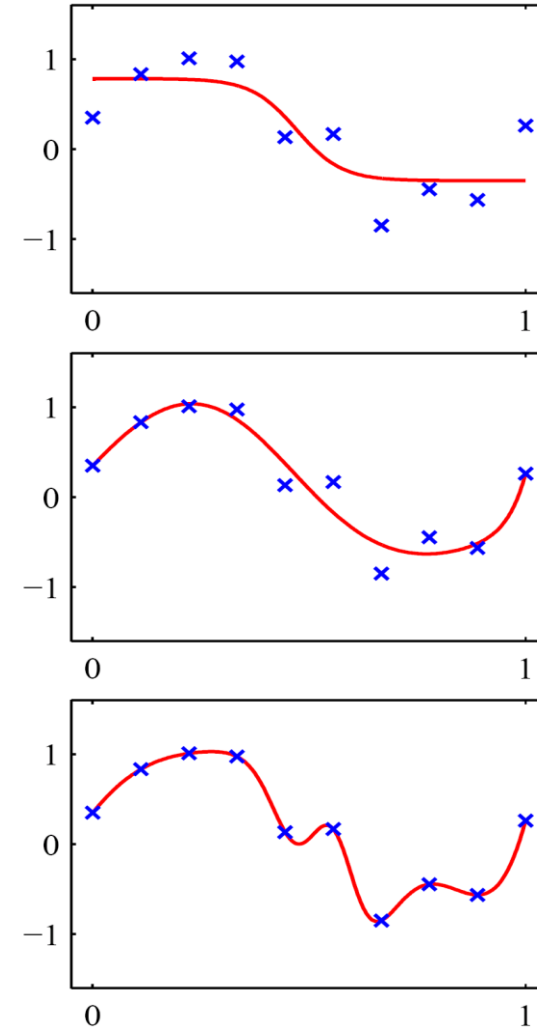
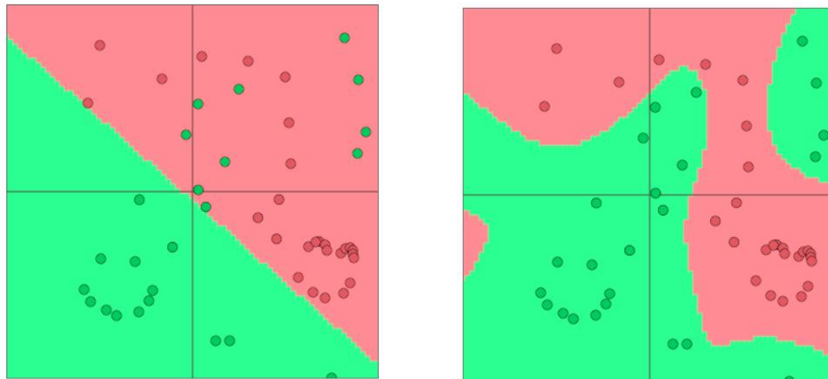
$$\text{Swish}(x) = x \cdot \sigma(\beta x)$$





# 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!



# Training with “cross entropy loss” – you use this in PyTorch!

- Until now, our objective was stated as to **maximize the probability of the correct class  $y$**  or equivalently we can **minimize the negative log probability of that class**
- Now restated in terms of **cross entropy**, a concept from **information theory**
- Let the true probability distribution be  $p$ ; let our computed model probability be  $q$

- The cross entropy is:

$$H(p, q) = - \sum_{c=1}^C p(c) \log q(c)$$

- Assuming a ground truth (or true or gold or target) probability distribution that is 1 at the right class and 0 everywhere else,  $p = [0, \dots, 0, 1, 0, \dots, 0]$ , then:
- **Because of one-hot  $p$ , the only term left is the negative log probability of the true class  $y_i$ :  $-\log p(y_i|x_i)$**

# Remember: Stochastic Gradient Descent

Update equation:

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

$\alpha = \text{step size or learning rate}$

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

In deep learning,  $\theta$  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

### 3. 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 second 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>

# 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

$$f(\mathbf{x}) = f(x_1, x_2, \dots, x_n)$$

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

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

# Jacobian Matrix: Generalization of the Gradient

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

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

- It's Jacobian is an  **$m \times n$  matrix** of partial derivatives

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

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

# 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**

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

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

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



## Example Jacobian: Elementwise activation Function

$$\mathbf{h} = f(\mathbf{z}), \text{ what is } \frac{\partial \mathbf{h}}{\partial \mathbf{z}}? \quad \mathbf{h}, \mathbf{z} \in \mathbb{R}^n$$
$$h_i = f(z_i)$$

## Example Jacobian: Elementwise activation Function

$$\mathbf{h} = f(\mathbf{z}), \text{ what is } \frac{\partial \mathbf{h}}{\partial \mathbf{z}}? \quad \mathbf{h}, \mathbf{z} \in \mathbb{R}^n$$
$$h_i = f(z_i)$$

Function has  $n$  outputs and  $n$  inputs  $\rightarrow n$  by  $n$  Jacobian

## Example Jacobian: Elementwise activation Function

$$\mathbf{h} = f(\mathbf{z}), \text{ what is } \frac{\partial \mathbf{h}}{\partial \mathbf{z}}?$$

$$\mathbf{h}, \mathbf{z} \in \mathbb{R}^n$$

$$h_i = f(z_i)$$

$$\left( \frac{\partial \mathbf{h}}{\partial \mathbf{z}} \right)_{ij} = \frac{\partial h_i}{\partial z_j} = \frac{\partial}{\partial z_j} f(z_i)$$

definition of Jacobian

## Example Jacobian: Elementwise activation Function

$$\mathbf{h} = f(\mathbf{z}), \text{ what is } \frac{\partial \mathbf{h}}{\partial \mathbf{z}}?$$

$$h_i = f(z_i)$$

$$\mathbf{h}, \mathbf{z} \in \mathbb{R}^n$$

$$\left( \frac{\partial \mathbf{h}}{\partial \mathbf{z}} \right)_{ij} = \frac{\partial h_i}{\partial z_j} = \frac{\partial}{\partial z_j} f(z_i)$$

$$= \begin{cases} f'(z_i) & \text{if } i = j \\ 0 & \text{if otherwise} \end{cases}$$

definition of Jacobian

regular 1-variable derivative

## Example Jacobian: Elementwise activation Function

$$\mathbf{h} = f(\mathbf{z}), \text{ what is } \frac{\partial \mathbf{h}}{\partial \mathbf{z}}?$$

$$\mathbf{h}, \mathbf{z} \in \mathbb{R}^n$$

$$h_i = f(z_i)$$

$$\left( \frac{\partial \mathbf{h}}{\partial \mathbf{z}} \right)_{ij} = \frac{\partial h_i}{\partial z_j} = \frac{\partial}{\partial z_j} f(z_i)$$

definition of Jacobian

$$= \begin{cases} f'(z_i) & \text{if } i = j \\ 0 & \text{if otherwise} \end{cases}$$

regular 1-variable derivative

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

## Other Jacobians

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

## Other Jacobians

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

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

## Other Jacobians

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

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

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

Fine print: This is the correct Jacobian.  
Later we discuss the “shape convention”;  
using it the answer would be  $\mathbf{h}$ .



## Other Jacobians

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

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

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

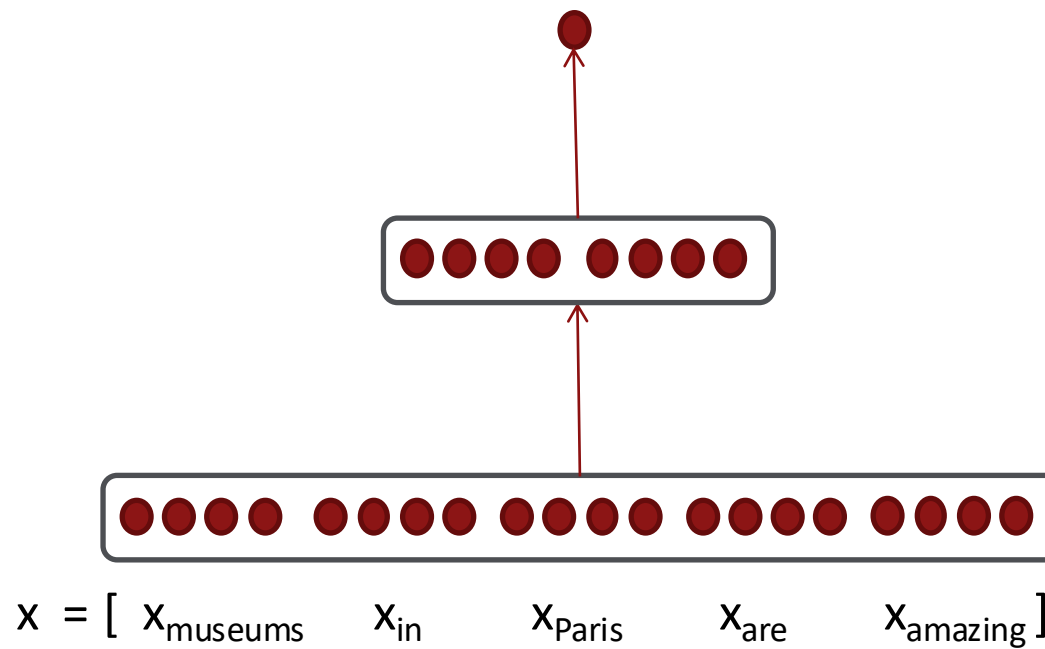
- Compute these at home for practice!
  - Check your answers with the lecture notes

# Back to our Neural Net!

$$s = u^T h$$

$$h = f(Wx + b)$$

$x$  (input)



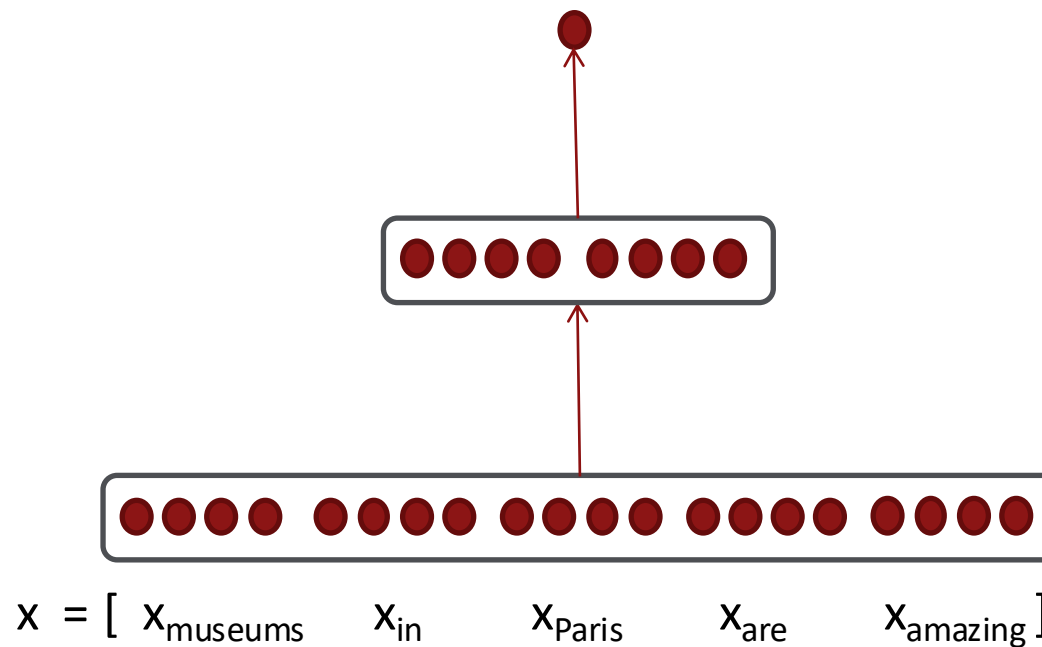
# Back to our Neural Net!

- Let's find  $\frac{\partial s}{\partial b}$ 
  - Really, we care about the gradient of the loss  $J$  but we will compute the gradient of the score for simplicity

$$s = u^T h$$

$$h = f(Wx + b)$$

$x$  (input)



## a. Break up equations into simple pieces

$$s = u^T h$$

$$s = u^T h$$

$$h = f(Wx + b) \quad \rightarrow \quad h = f(z)$$

$$z = Wx + b$$

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

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

Carefully define your variables and **keep track of their dimensionality!**

## b. Apply the chain rule

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

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

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

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

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

## b. Apply the chain rule

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

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

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

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

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

## b. Apply the chain rule

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

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

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

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

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

## b. Apply the chain rule

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

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

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

$\mathbf{x}$  (input)

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



## c. Write out the Jacobians

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

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

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

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

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

Useful Jacobians from previous slide

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

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

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

## c. Write out the Jacobians

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

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

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

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

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

$\downarrow$   
 $\mathbf{u}^T$

Useful Jacobians from previous slide

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

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

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

## c. Write out the Jacobians

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

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

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

$\mathbf{x}$  (input)

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

$\downarrow \qquad \qquad \downarrow$

$$\mathbf{u}^T \text{diag}(f'(\mathbf{z}))$$

Useful Jacobians from previous slide

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

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

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

## c. Write out the Jacobians

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

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

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

$\mathbf{x}$  (input)

$$\begin{aligned} \frac{\partial s}{\partial \mathbf{b}} &= \frac{\partial s}{\partial \mathbf{h}} \quad \frac{\partial \mathbf{h}}{\partial \mathbf{z}} \quad \frac{\partial \mathbf{z}}{\partial \mathbf{b}} \\ &\quad \downarrow \quad \downarrow \quad \downarrow \\ &= \mathbf{u}^T \text{diag}(f'(\mathbf{z})) \mathbf{I} \end{aligned}$$

Useful Jacobians from previous slide

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

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

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

## c. Write out the Jacobians

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

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

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

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

$$\begin{aligned} \frac{\partial s}{\partial \mathbf{b}} &= \frac{\partial s}{\partial \mathbf{h}} \quad \frac{\partial \mathbf{h}}{\partial \mathbf{z}} \quad \frac{\partial \mathbf{z}}{\partial \mathbf{b}} \\ &\quad \downarrow \quad \quad \downarrow \quad \quad \downarrow \\ &= \mathbf{u}^T \text{diag}(f'(\mathbf{z})) \mathbf{I} \\ &= \mathbf{u}^T \odot f'(\mathbf{z}) \end{aligned}$$

Useful Jacobians from previous slide

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

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

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

$\odot$  = Hadamard product =  
element-wise multiplication  
of 2 vectors to give vector

## Re-using Computation

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

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

# Re-using Computation

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

$$\frac{\partial s}{\partial \mathbf{W}} = \frac{\partial s}{\partial \mathbf{h}} \frac{\partial \mathbf{h}}{\partial \mathbf{z}} \frac{\partial \mathbf{z}}{\partial \mathbf{W}}$$
$$\frac{\partial s}{\partial \mathbf{b}} = \frac{\partial s}{\partial \mathbf{h}} \frac{\partial \mathbf{h}}{\partial \mathbf{z}} \frac{\partial \mathbf{z}}{\partial \mathbf{b}}$$

The same! Let's avoid duplicated computation ...

# Re-using Computation

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

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

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

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

$\delta$  is the upstream gradient (“error signal”)



## Derivative with respect to Matrix: Output shape

- What does  $\frac{\partial s}{\partial \mathbf{W}}$  look like?  $\mathbf{W} \in \mathbb{R}^{n \times m}$
- 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  $\frac{\partial s}{\partial \mathbf{W}}$  look like?  $\mathbf{W} \in \mathbb{R}^{n \times m}$
- 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  $\frac{\partial s}{\partial \mathbf{W}}$  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  $\frac{\partial s}{\partial \mathbf{W}} = \delta \frac{\partial z}{\partial \mathbf{W}}$ 
  - $\delta$  is going to be in our answer
  - The other term should be  $\mathbf{x}$  because  $\mathbf{z} = \mathbf{W}\mathbf{x} + \mathbf{b}$

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

$\delta$  is upstream gradient (“error signal”) at  $z$   
 $\mathbf{x}$  is local input signal

## Why the Transposes?

$$\begin{aligned} \frac{\partial s}{\partial \mathbf{W}} &= \boldsymbol{\delta}^T \mathbf{x}^T \\ [n \times m] \quad [n \times 1][1 \times m] \\ &= \begin{bmatrix} \delta_1 \\ \vdots \\ \delta_n \end{bmatrix} [x_1, \dots, x_m] = \begin{bmatrix} \delta_1 x_1 & \dots & \delta_1 x_m \\ \vdots & \ddots & \vdots \\ \delta_n x_1 & \dots & \delta_n x_m \end{bmatrix} \end{aligned}$$

- 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

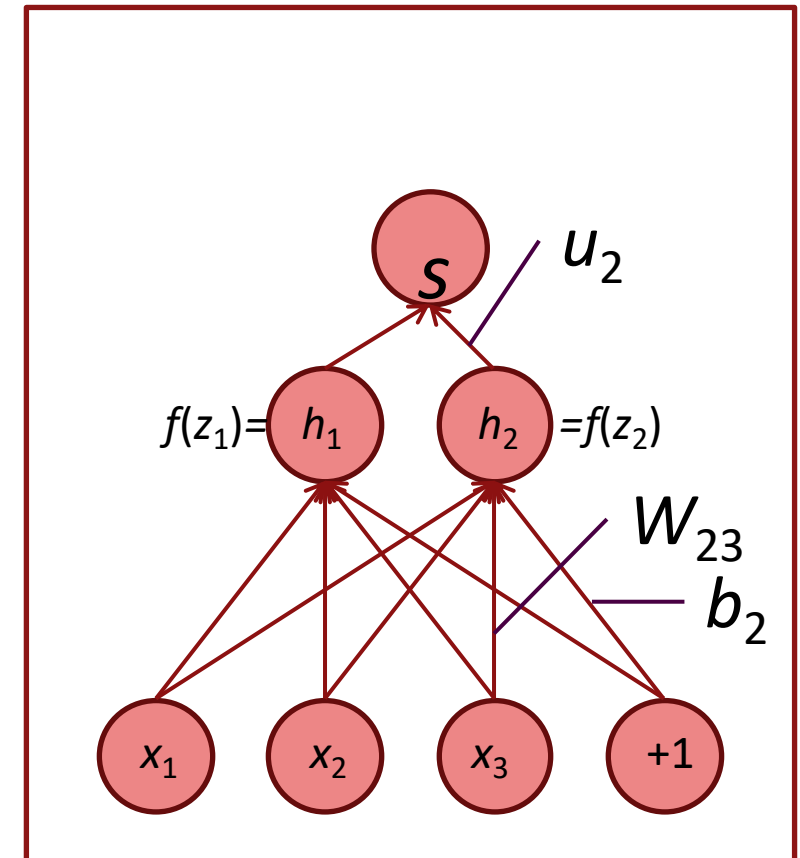
# Deriving local input gradient in backprop

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

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

- 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  $z_1$

$$\begin{aligned} \frac{\partial z_i}{\partial W_{ij}} &= \frac{\partial}{\partial W_{ij}} W_{i \cdot} x + b_i \\ &= \frac{\partial}{\partial W_{ij}} \sum_{k=1}^d W_{ik} x_k = x_j \end{aligned}$$



# What shape should derivatives be?

- Similarly,  $\frac{\partial s}{\partial \mathbf{b}} = \mathbf{h}^T \circ f'(z)$  is a row vector
  - But shape convention says our gradient should be a column vector because  $\mathbf{b}$  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  $\frac{\partial s}{\partial \mathbf{b}}$  to make the derivative a column vector, resulting in  $\delta^T$

2. Always follow the shape convention

- Look at dimensions to figure out when to transpose and/or reorder terms
- The error message  $\delta$  that arrives at a hidden layer has the same dimensionality as that hidden layer

Use Jacobian  
to **compute**;  
Use shape  
convention to  
**format**.

## 4. Backpropagation

We've almost shown you backpropagation

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

Other trick:

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



# Computation Graphs and Backpropagation

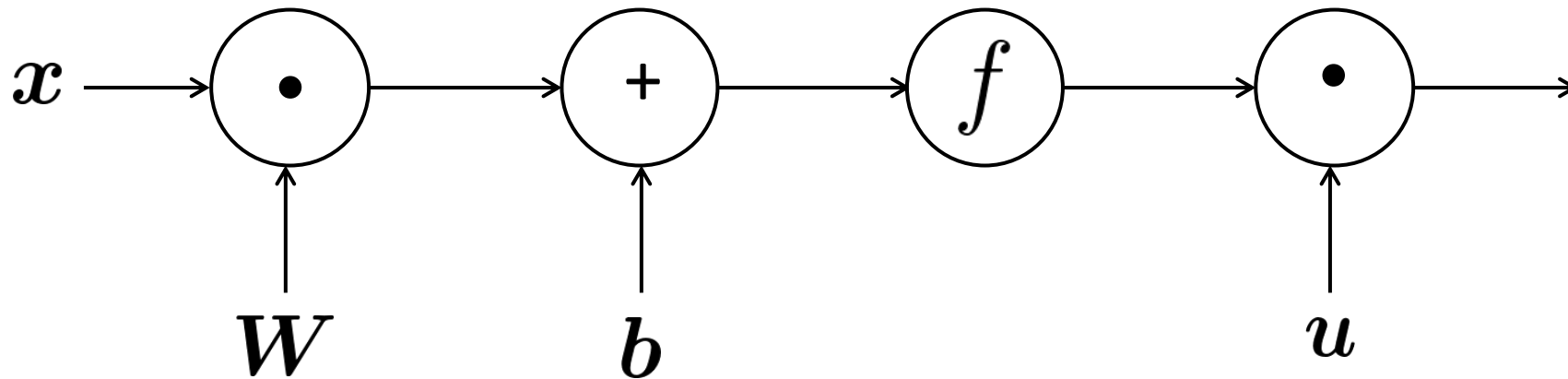
- Software represents our neural net equations as a graph
  - Source nodes: inputs
  - Interior nodes: operations

$$s = u^T h$$

$$h = f(z)$$

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

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



# Computation Graphs and Backpropagation

- Software represents our neural net equations as a graph

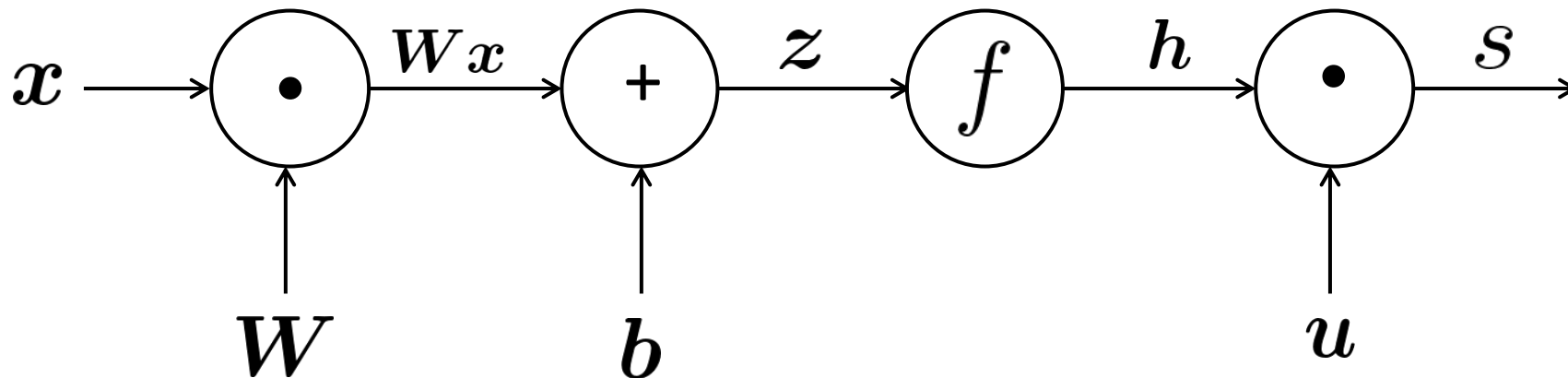
- Source nodes: inputs
- Interior nodes: operations
- Edges pass along result of the operation

$$s = u^T h$$

$$h = f(z)$$

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

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



# Computation Graphs and Backpropagation

- Software represents our neural net equations as a graph

$$s = u^T h$$

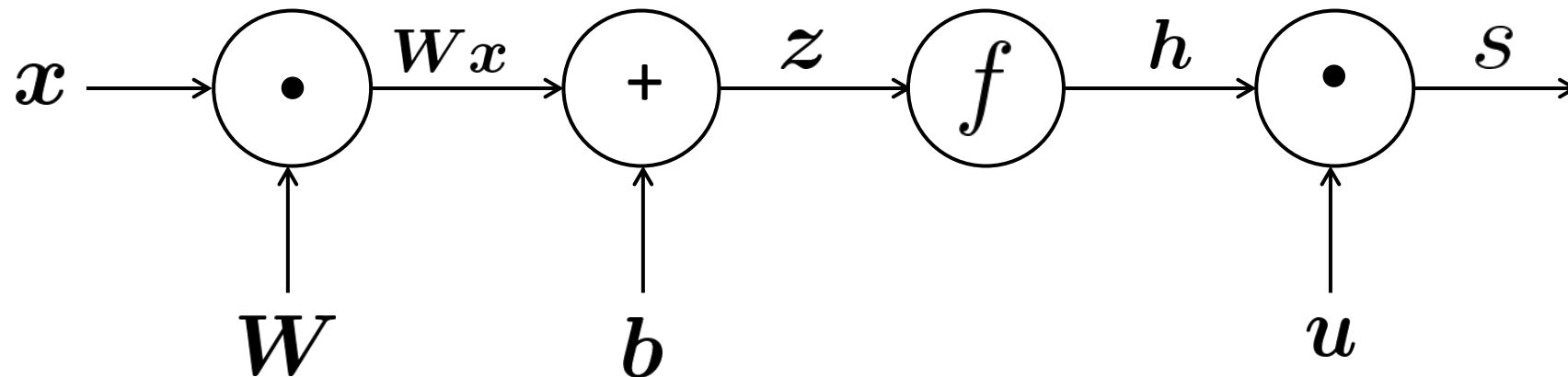
$$h = f(z)$$

$$z = x + b$$

(input)

“Forward Propagation”

operation



# Backpropagation

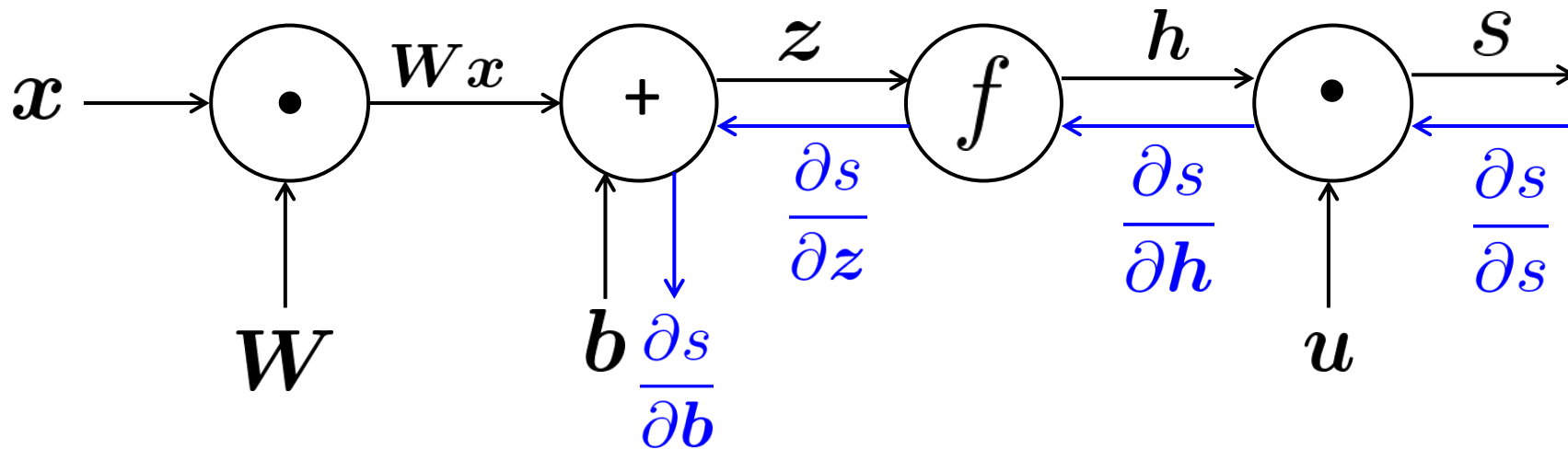
- Then go backwards along edges
  - Pass along **gradients**

$$s = u^T h$$

$$h = f(z)$$

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

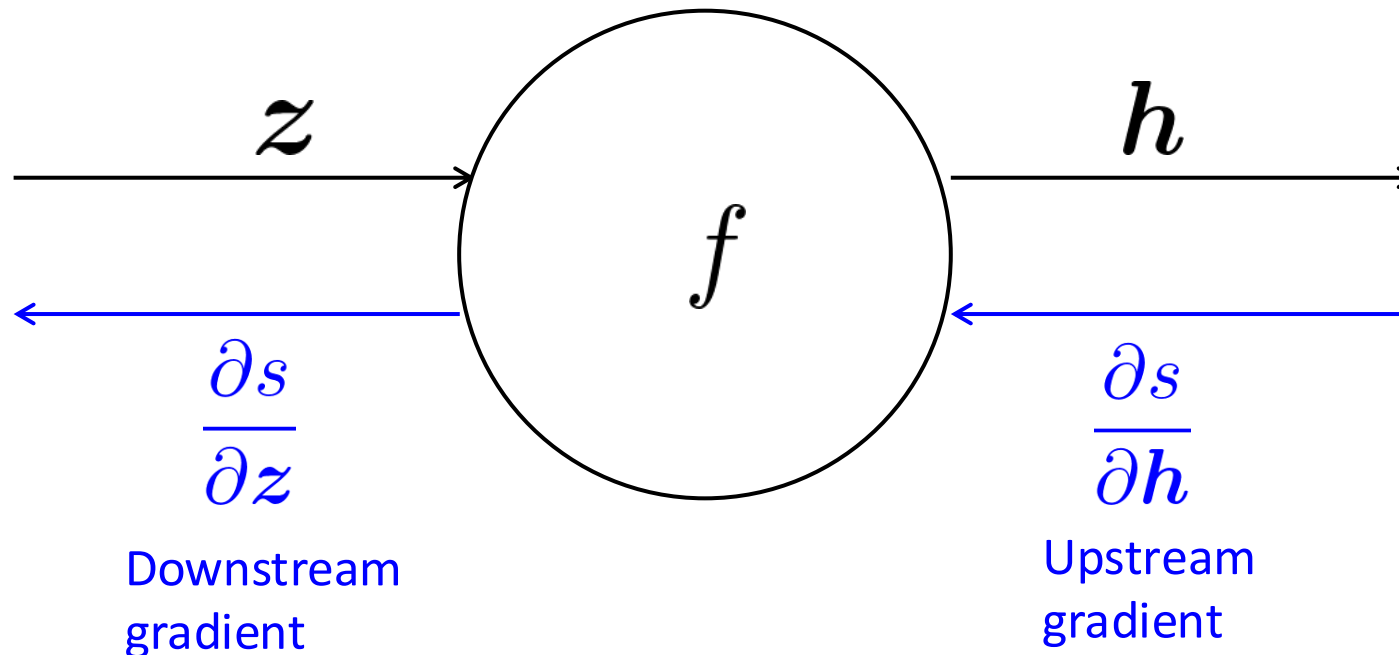
$$x \text{ (input)}$$



# Backpropagation: Single Node

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

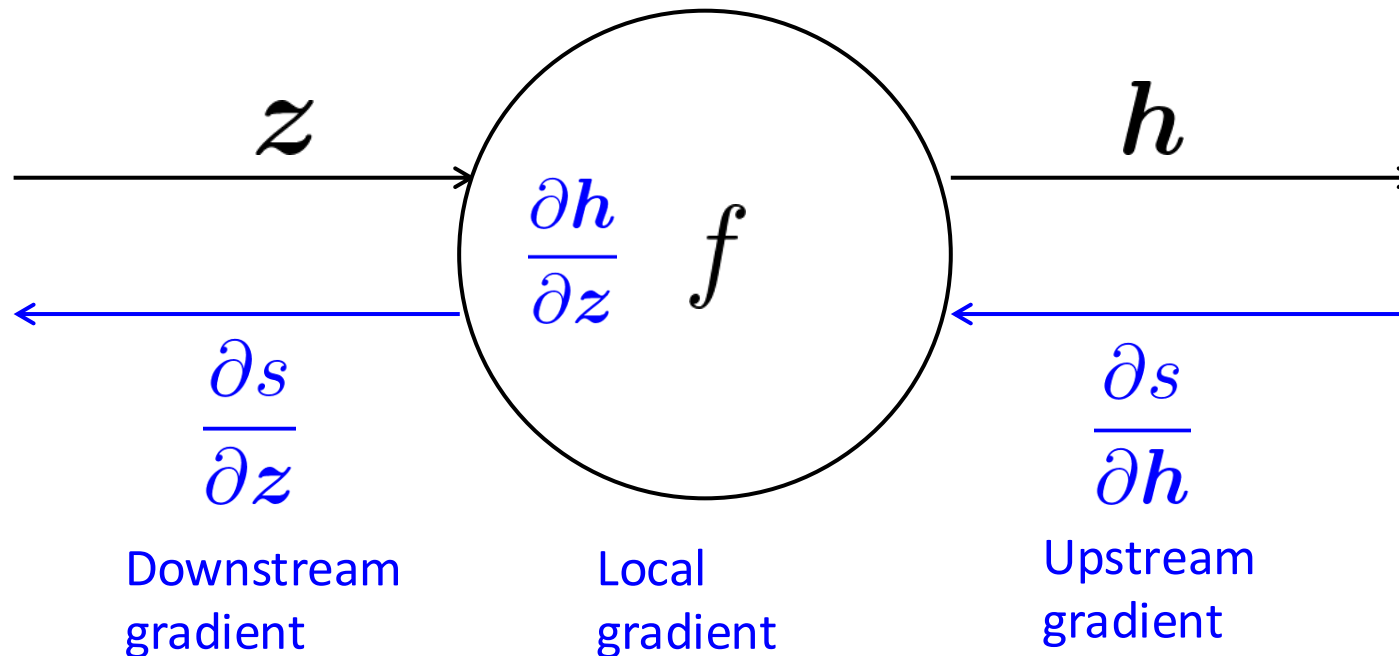
$$h = f(z)$$



# Backpropagation: Single Node

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

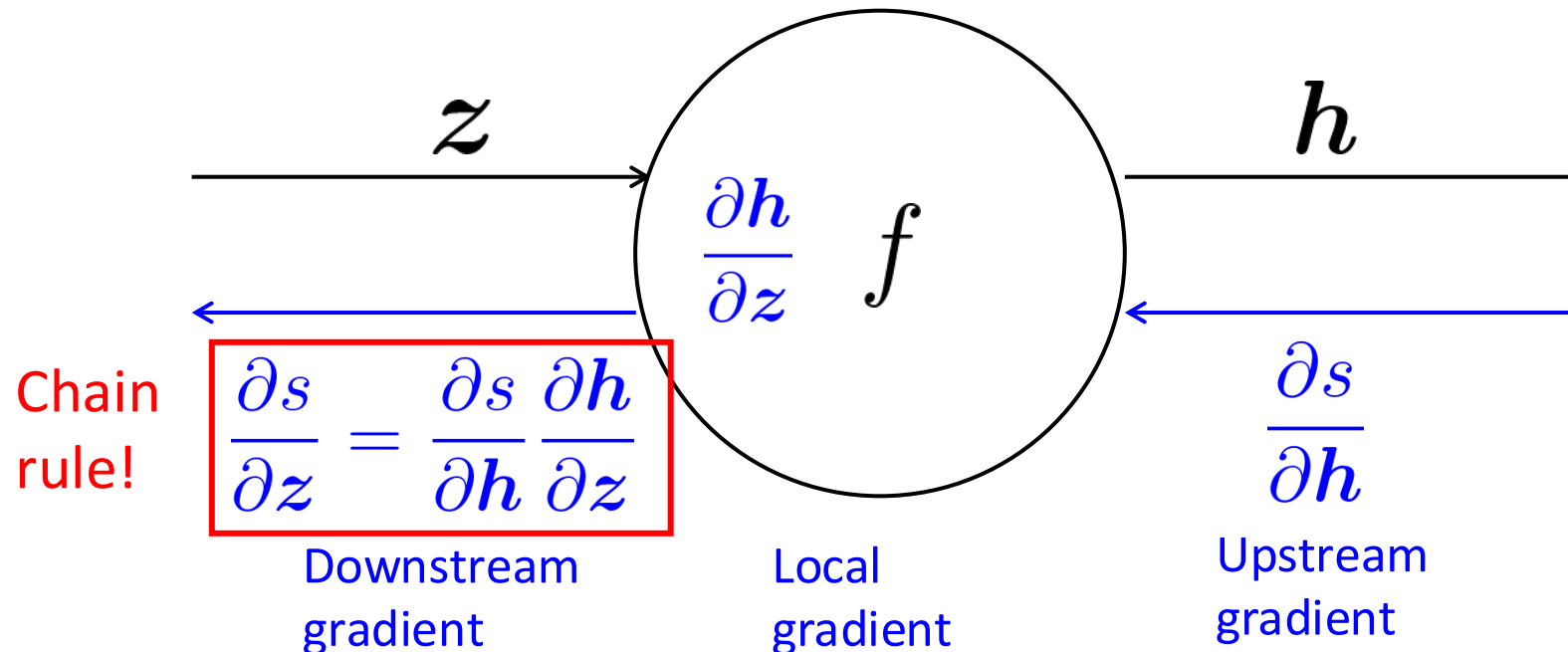
$$h = f(z)$$



# Backpropagation: Single Node

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

$$h = f(z)$$

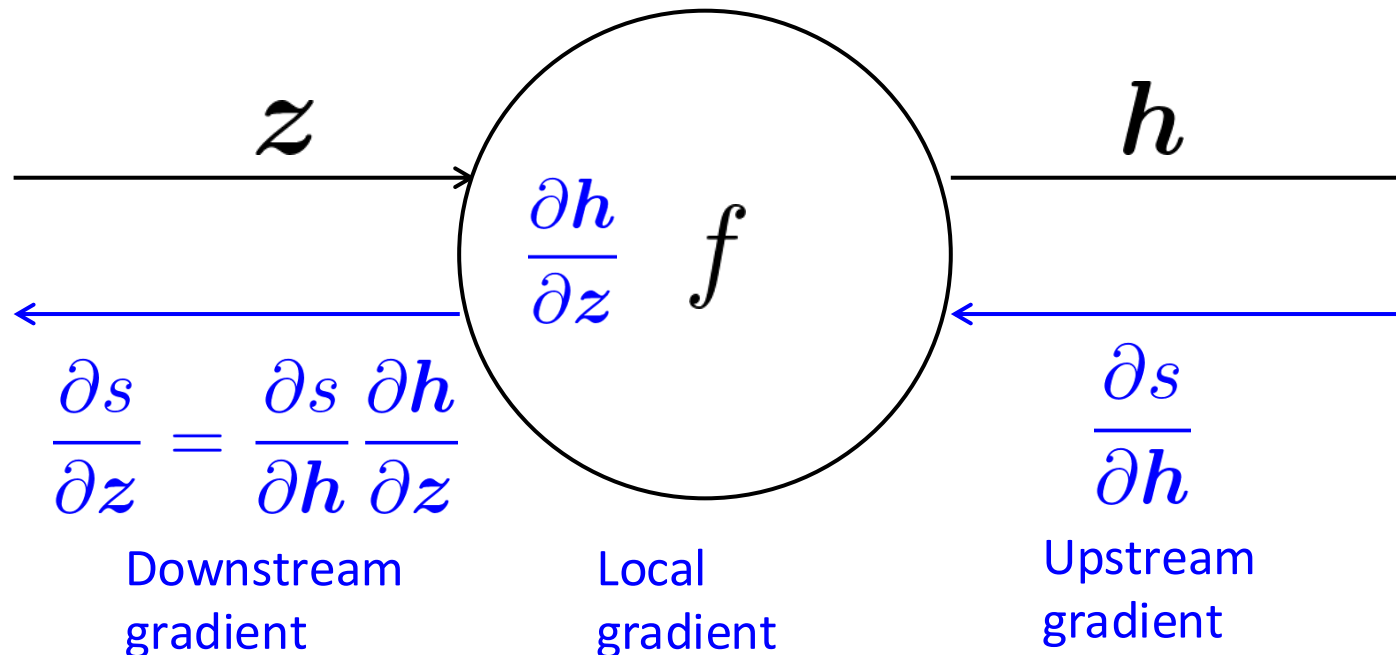


# Backpropagation: Single Node

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

$$h = f(z)$$

$$[\text{downstream gradient}] = [\text{upstream gradient}] \times [\text{local gradient}]$$

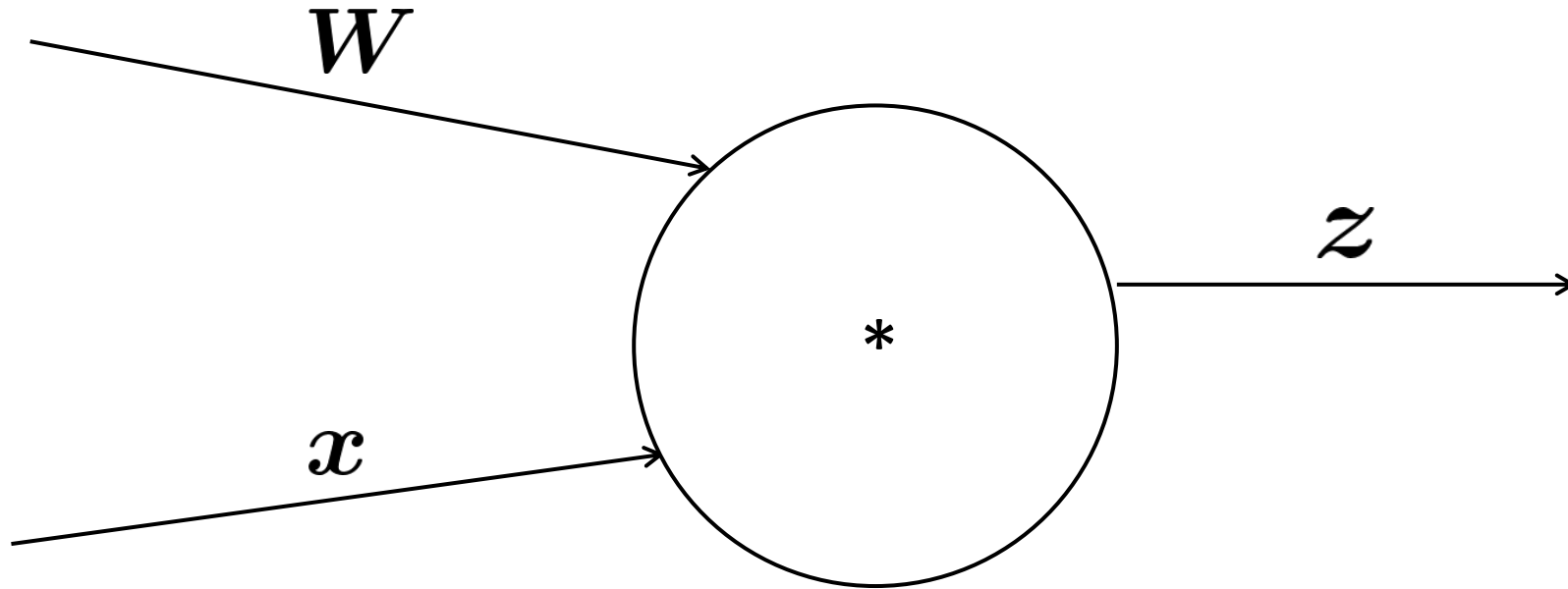




# Backpropagation: Single Node

- What about nodes with multiple inputs?

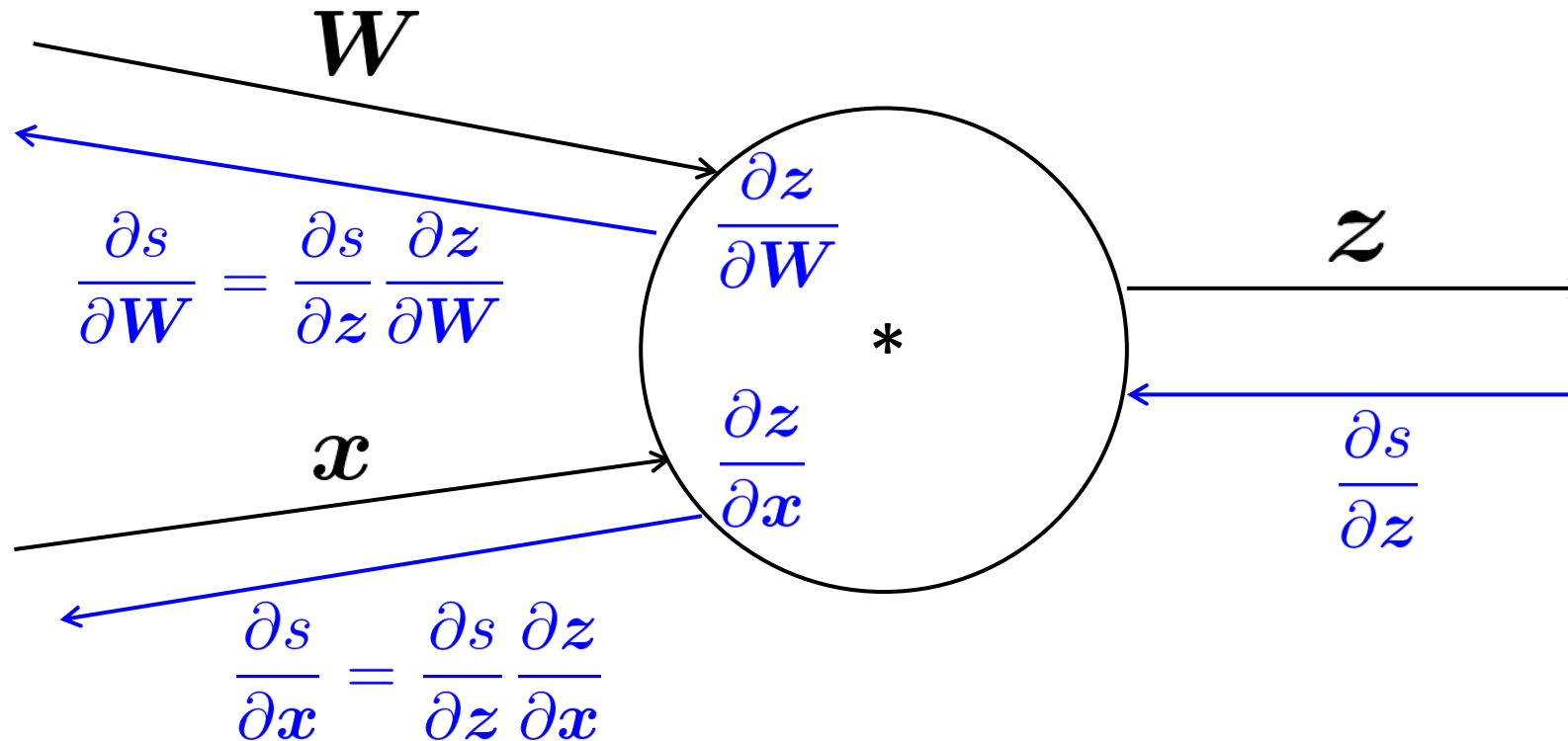
$$z = Wx$$



# Backpropagation: Single Node

- Multiple inputs  $\rightarrow$  multiple local gradients

$$z = Wx$$

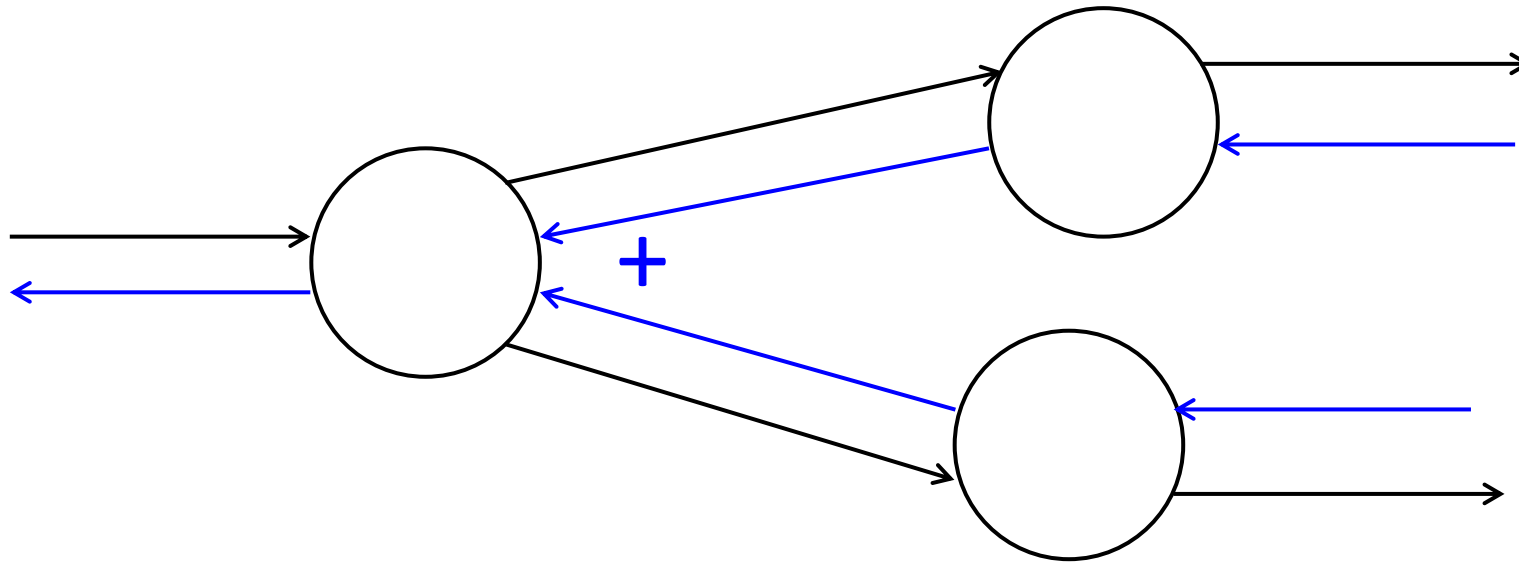


Downstream  
gradients

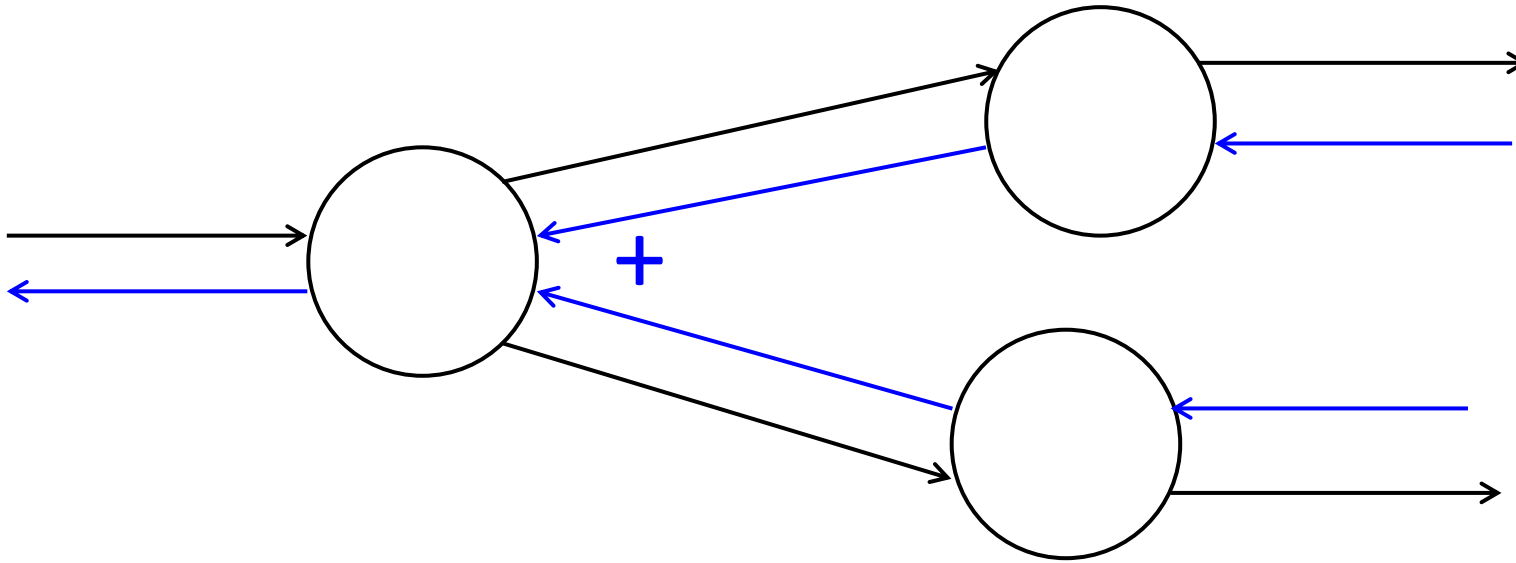
Local  
gradients

Upstream  
gradient

## Gradients sum at outward branches



## Gradients sum at outward branches



$$a = x + y$$

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

$$f = ab$$

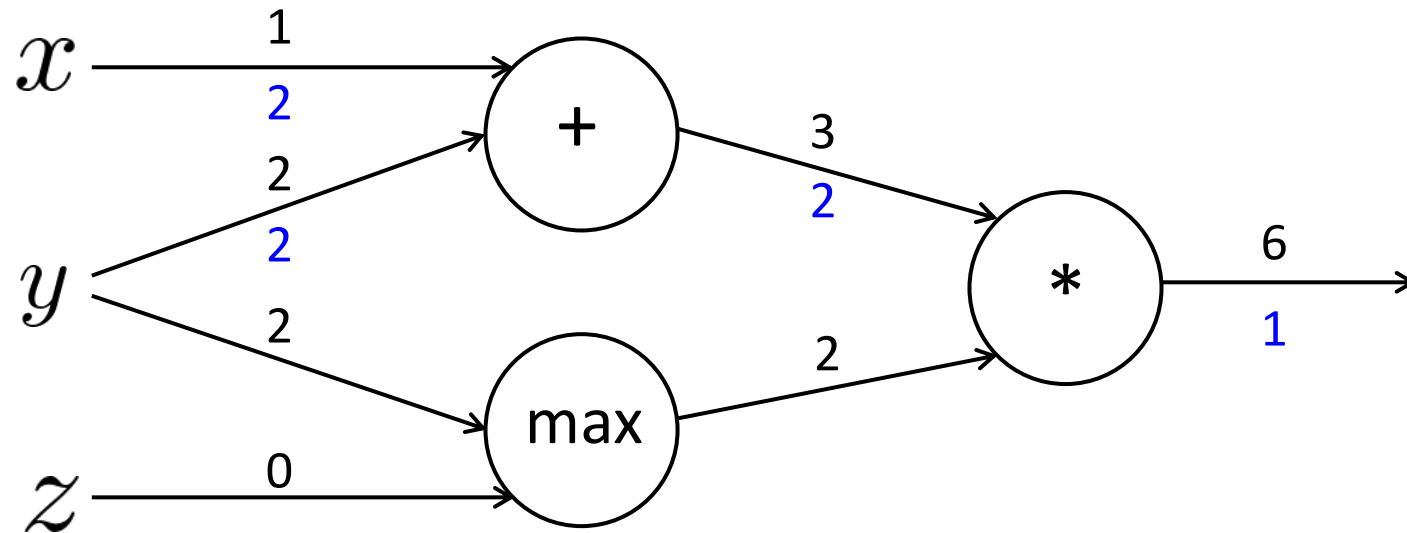
$$\frac{\partial f}{\partial y} = \frac{\partial f}{\partial a} \frac{\partial a}{\partial y} + \frac{\partial f}{\partial b} \frac{\partial b}{\partial y}$$

## Node Intuitions

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

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

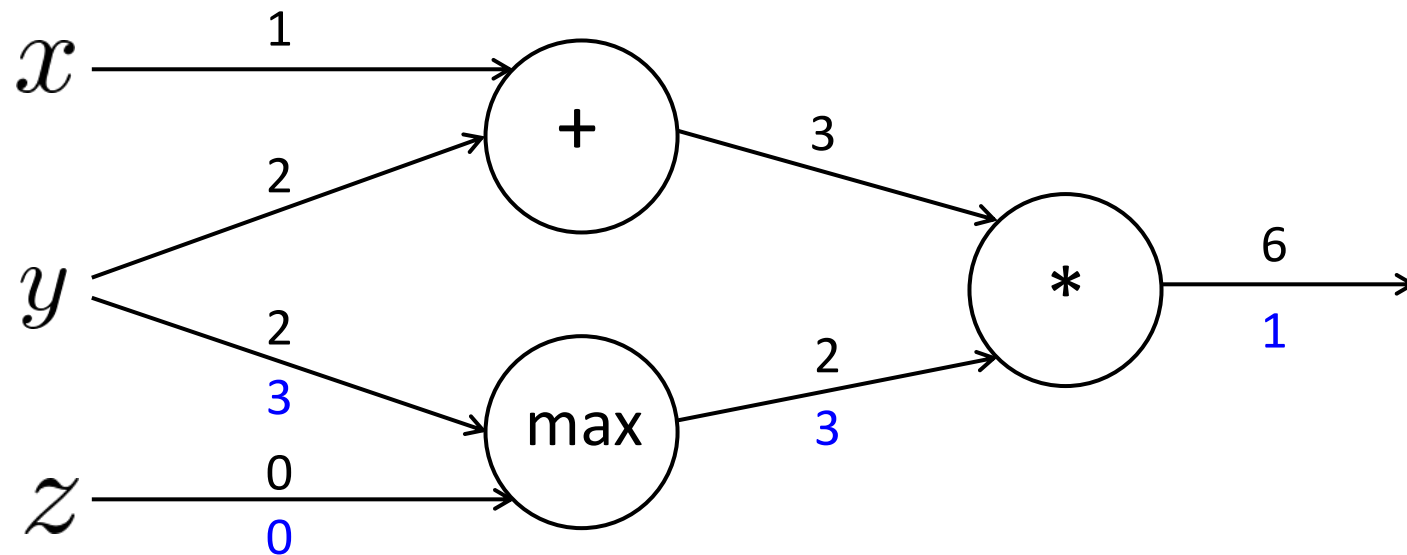
- + “distributes” the upstream gradient to each summand



## Node Intuitions

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

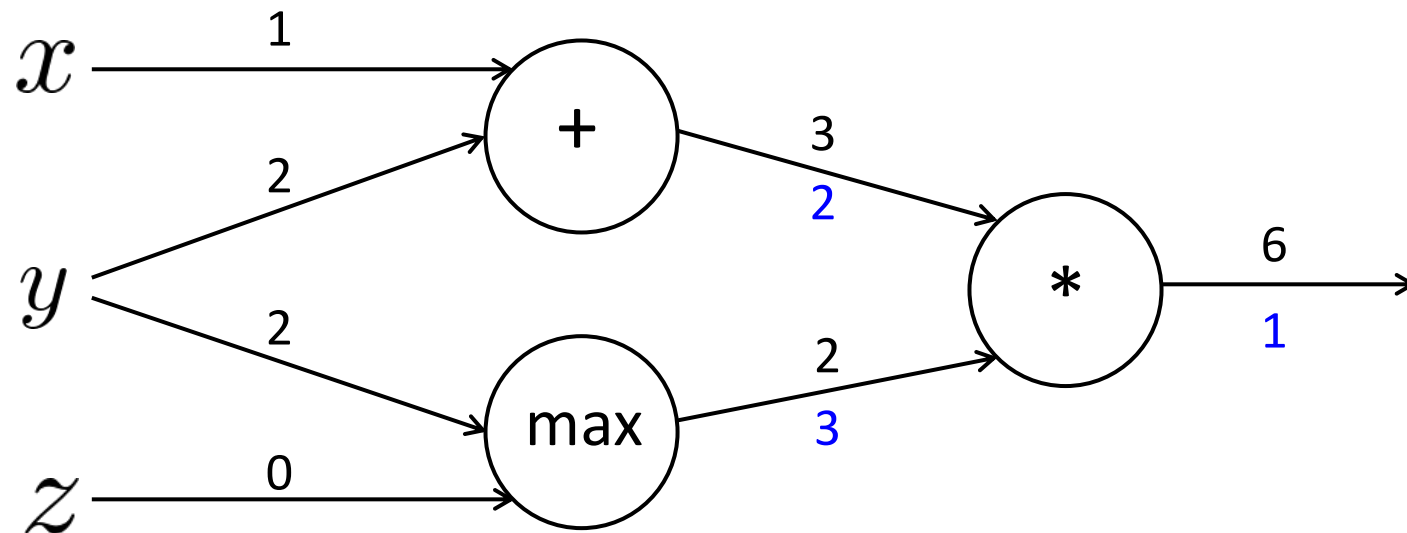
- + “distributes” the upstream gradient to each summand
- max “routes” the upstream gradient



## 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:

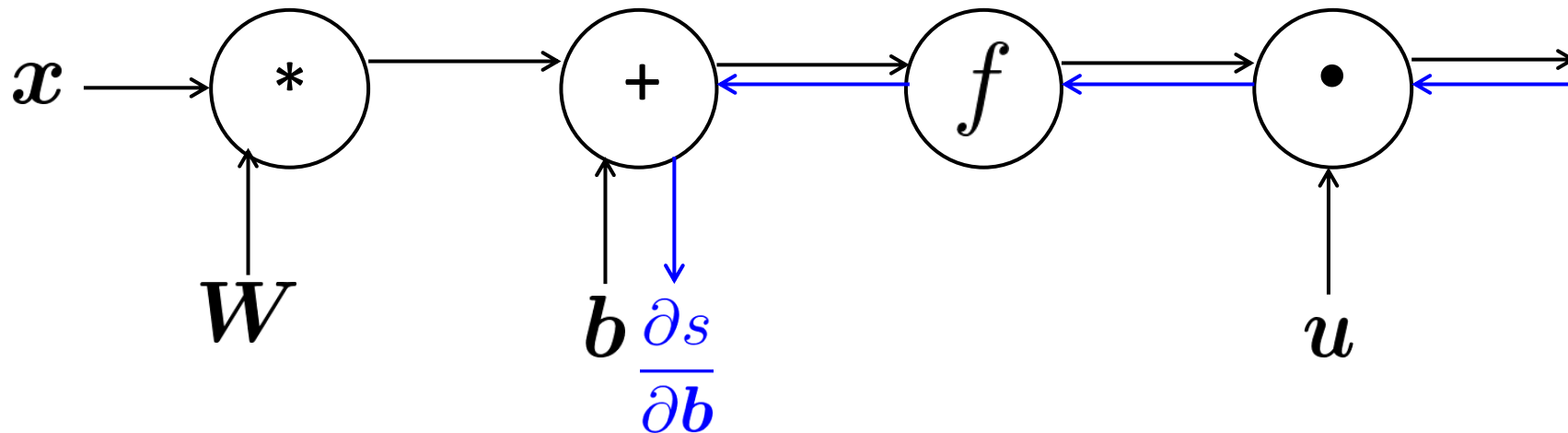
- First compute  $\frac{\partial s}{\partial b}$

$$s = u^T h$$

$$h = f(z)$$

$$z = Wx + b$$

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





# Efficiency: compute all gradients at once

- **Incorrect way** of doing backprop:

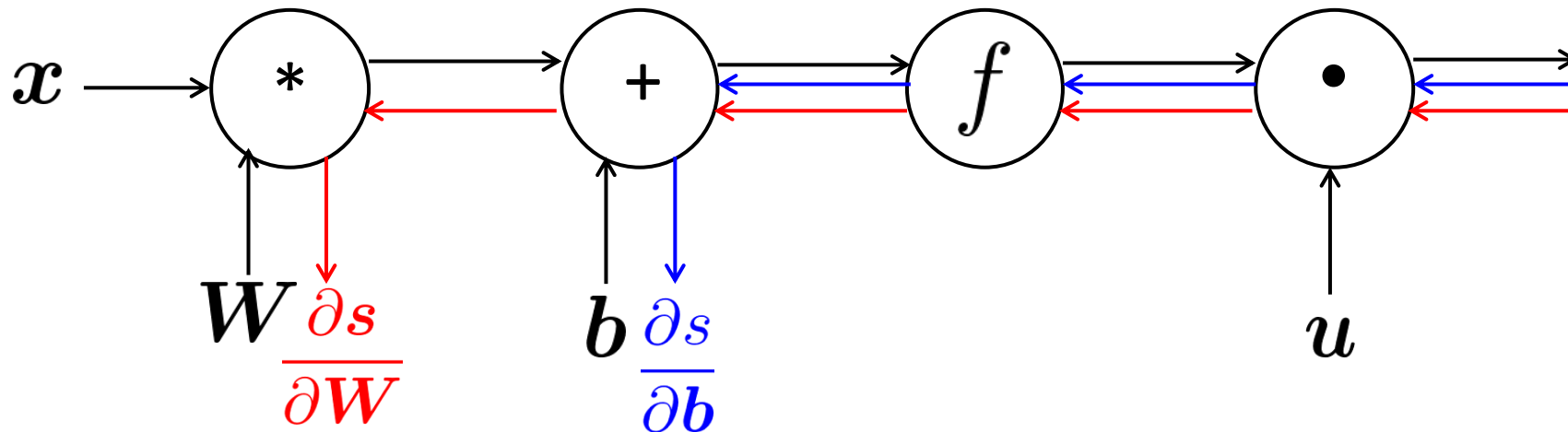
- First compute  $\frac{\partial s}{\partial b}$
- Then independently compute  $\frac{\partial s}{\partial W}$
- Duplicated computation!

$$s = u^T h$$

$$h = f(z)$$

$$z = Wx + b$$

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



# Efficiency: compute all gradients at once

- **Correct way:**

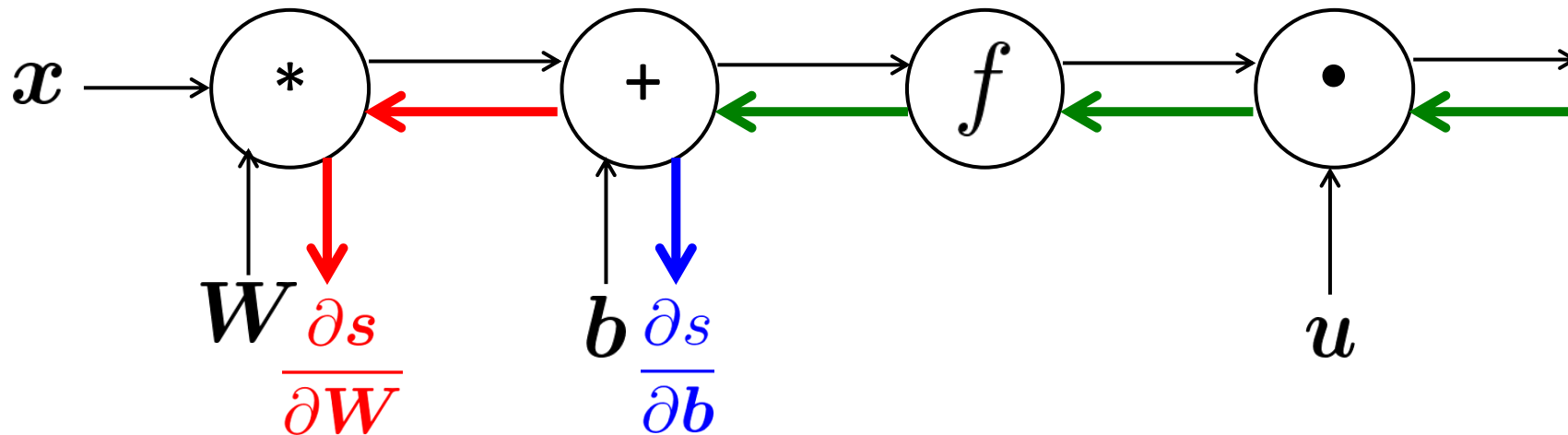
- Compute all the gradients at once
- Analogous to using  $\delta$  when we computed gradients by hand

$$s = u^T h$$

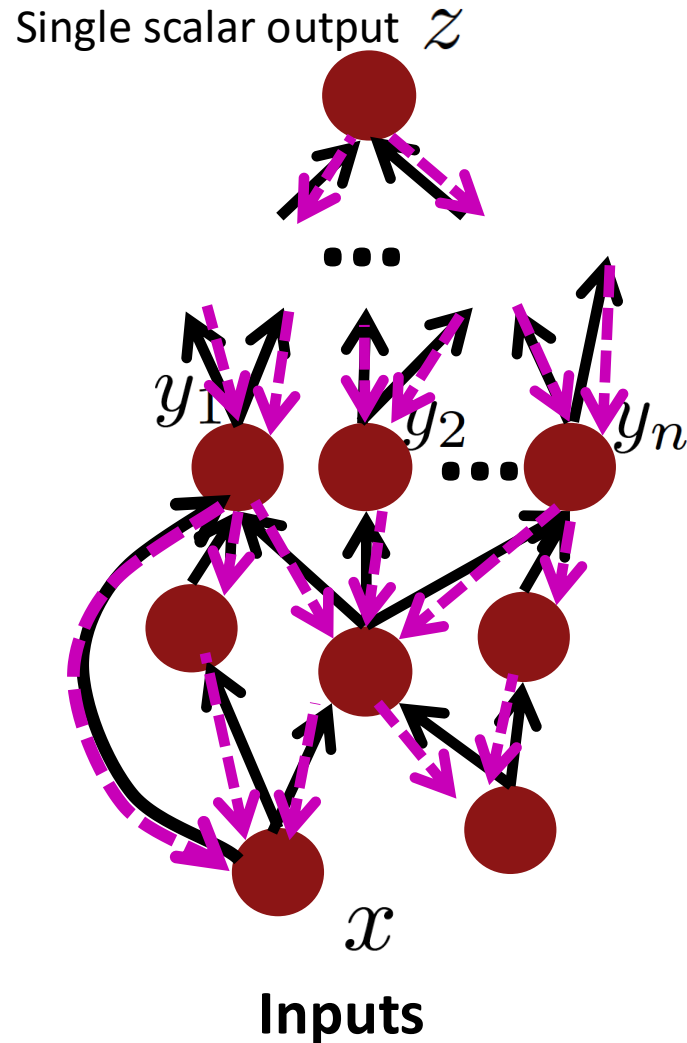
$$h = f(z)$$

$$z = Wx + b$$

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



# 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

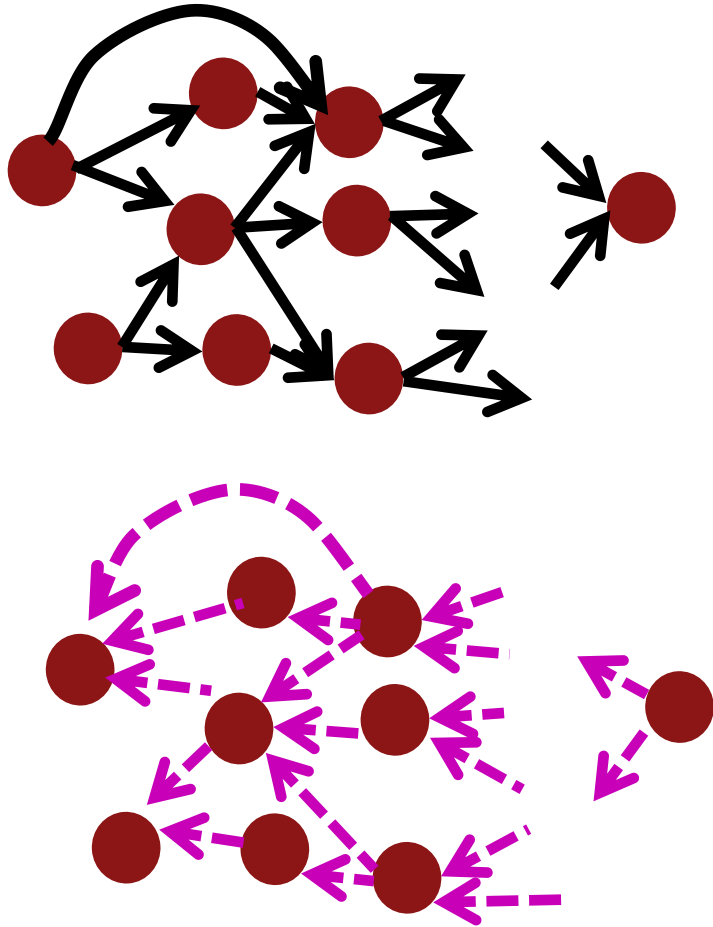
$\{y_1, y_2, \dots, y_n\} = \text{successors of } x$

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

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
```

# Manual Gradient checking: Numeric Gradient

- For small  $h$  ( $\approx 1\text{e-}4$ ),
$$f'(x) \approx \frac{f(x + h) - f(x - h)}{2h}$$
- 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
  - $[\text{downstream gradient}] = [\text{upstream gradient}] \times [\text{local gradient}]$
- **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