

Introduction to NLP

CSE5321/CSEG321

Lecture 5. Neural Networks

Hwaran Lee (hwaranlee@sogang.ac.kr)

Quick comparison of Word2vec and GloVe

	Word2vec	GloVe
Name	Word 2 vector	Global vector
Objective	Learns by predicting the neighboring words using the center word	Learns by factorizing the word co-occurrence matrix
Data	Local context Sliding window of neighbor words	Global context Co-occurrence info across corpus
Optimization	Using neural nets Optimize log likelihood	Doing matrix factorization Optimize least square

4. 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!

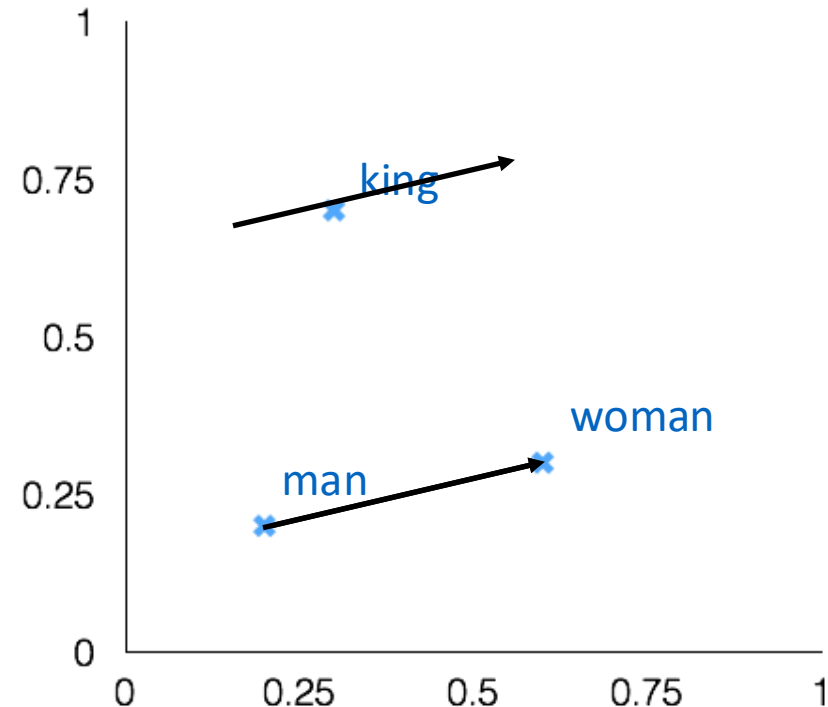
Intrinsic word vector evaluation

- 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 (!)



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 [†]	6B	57.2	65.6	68.2	57.0	32.5
SG [†]	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	<u>75.9</u>	<u>83.6</u>	<u>82.9</u>	<u>59.6</u>	<u>47.8</u>
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	88.7	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	93.2	88.3	82.9	82.2

5. Word senses and word sense ambiguity

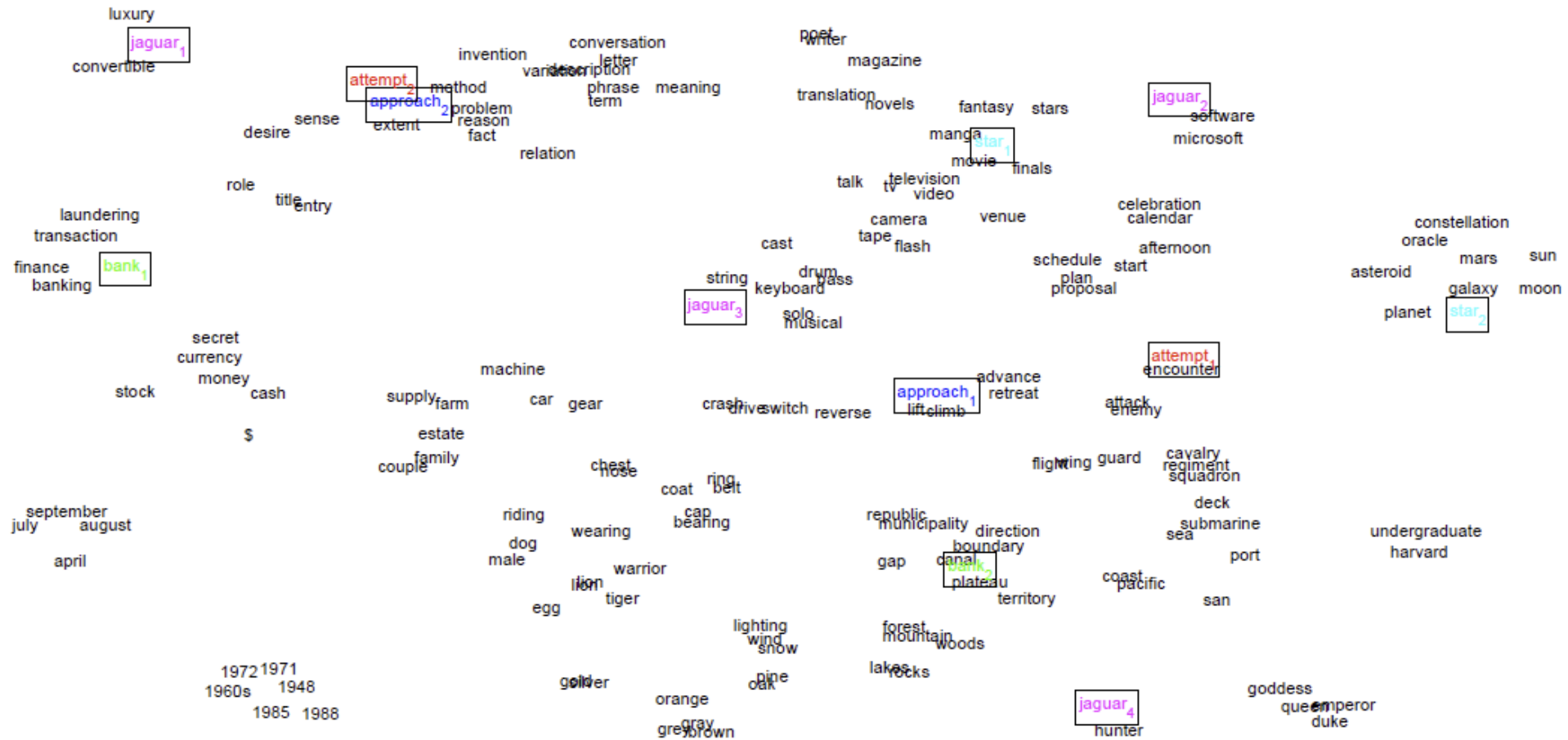
- Most words have lots of meanings!
 - Especially common words
 - Especially words that have existed for a long time
- Example: **pike**
- Does one vector capture all these meanings or do we have a mess?

pike

- A sharp point or staff
- A type of elongated fish
- A railroad line or system
- A type of road
- The future (coming down the pike)
- A type of body position (as in diving)
- To kill or pierce with a pike
- To make one's way (pike along)
- In Australian English, pike means to pull out from doing something: *I reckon he could have climbed that cliff, but he piked!*

Improving Word Representations Via Global Context And Multiple Word Prototypes (Huang et al. 2012)

- Idea: Cluster word windows around words, retrain with each word assigned to multiple different clusters bank_1 , bank_2 , etc.



Linear Algebraic Structure of Word Senses, with Applications to Polysemy (Arora, ..., Ma, ..., TACL 2018)

- Different senses of a word reside in a linear superposition (weighted sum) in standard word embeddings like word2vec
- $v_{\text{pike}} = \alpha_1 v_{\text{pike}_1} + \alpha_2 v_{\text{pike}_2} + \alpha_3 v_{\text{pike}_3}$
- Where $\alpha_1 = \frac{f_1}{f_1 + f_2 + f_3}$, etc., for frequency f
- Surprising result:
 - Because of ideas from *sparse coding* you can actually separate out the senses (providing they are relatively common)!

tie				
trousers	season	scoreline	wires	operatic
blouse	teams	goalless	cables	soprano
waistcoat	winning	equaliser	wiring	mezzo
skirt	league	clinching	electrical	contralto
sleeved	finished	scoreless	wire	baritone
pants	championship	replay	cable	coloratura

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

Classification review and notation

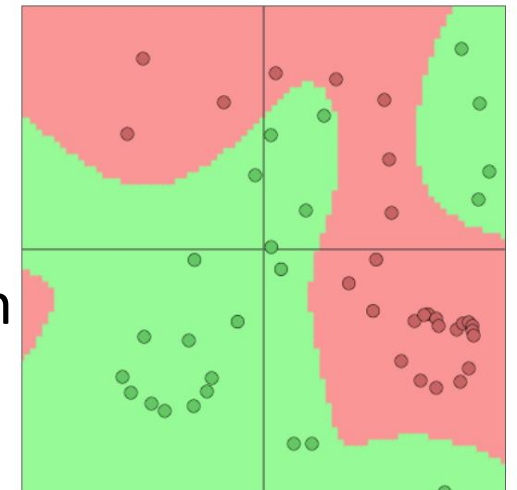
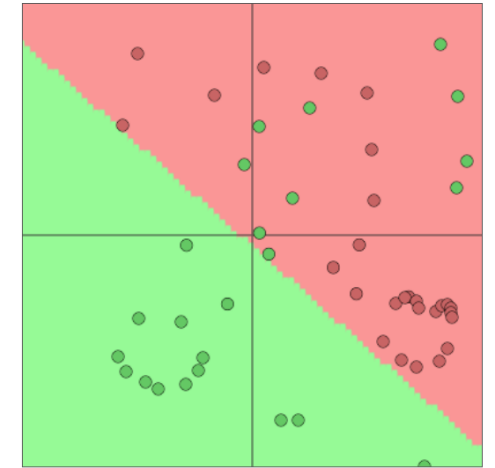
- Supervised learning: we have a **training dataset** consisting of **samples**

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

- 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 θ 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



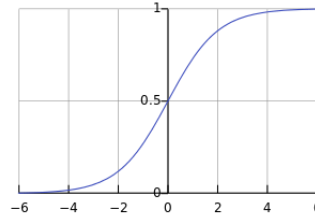
But typically, it is linear relative to the pre-final layer representation

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



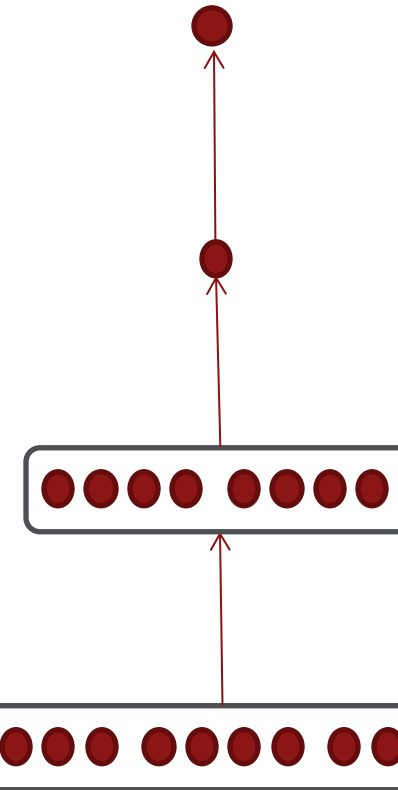
$$s = u^T h$$

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

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

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

Embedding of
1-hot words

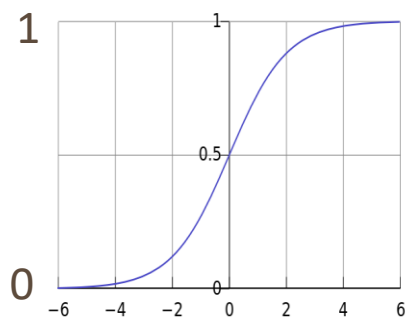


f = Some element-wise non-linear function, e.g., logistic, tanh, ReLU

Non-linearities, old and new

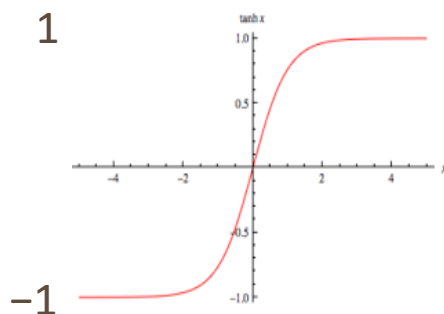
logistic (“sigmoid”)

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



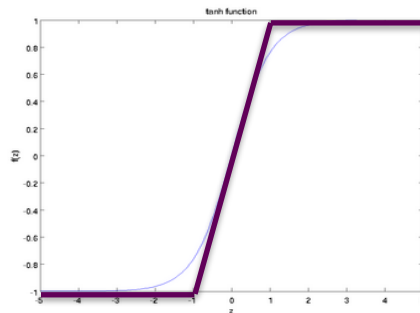
tanh

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



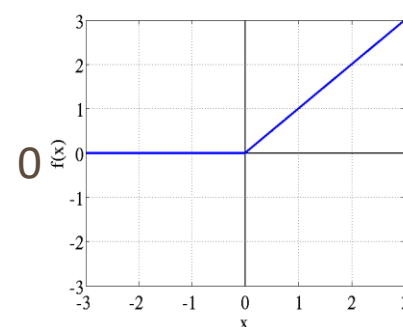
hard tanh

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

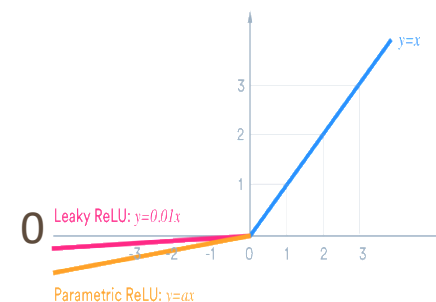


(Rectified Linear Unit)
ReLU

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



Leaky ReLU /
Parametric ReLU



tanh is just a rescaled and shifted sigmoid ($2 \times$ as steep, $[-1,1]$):

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

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

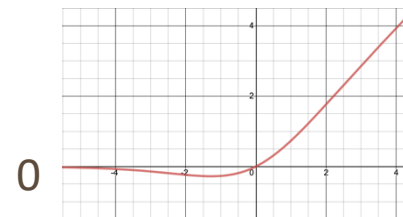
However, now, for deep networks, the first thing to try is ReLU: it trains quickly and performs well due to good gradient backflow.

ReLU has a negative “dead zone” that recent proposals mitigate

GELU is frequently used with Transformers (BERT, RoBERTa, etc.)

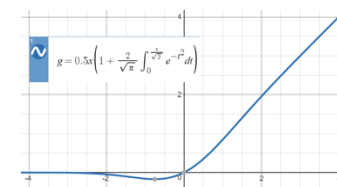
Swish [arXiv:1710.05941](https://arxiv.org/abs/1710.05941)

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



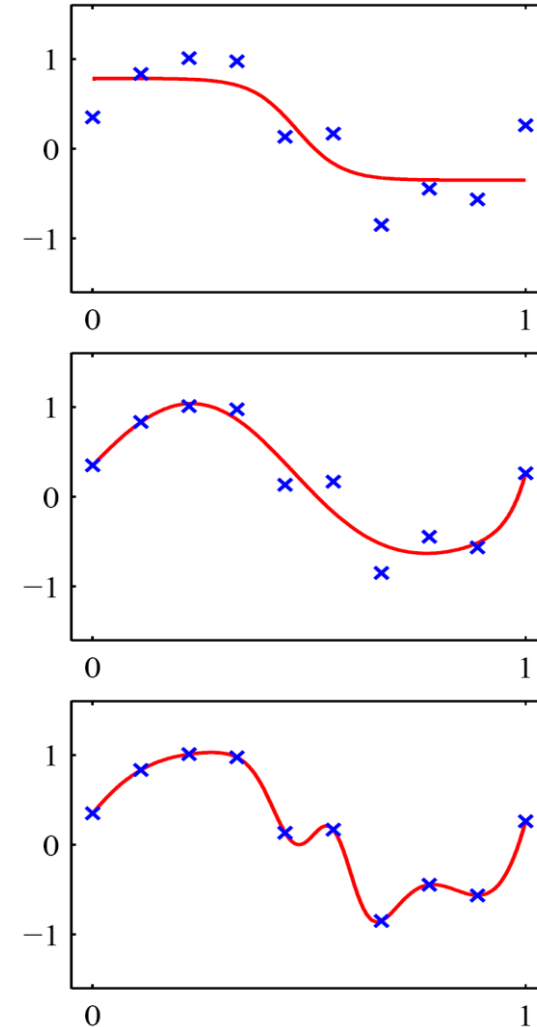
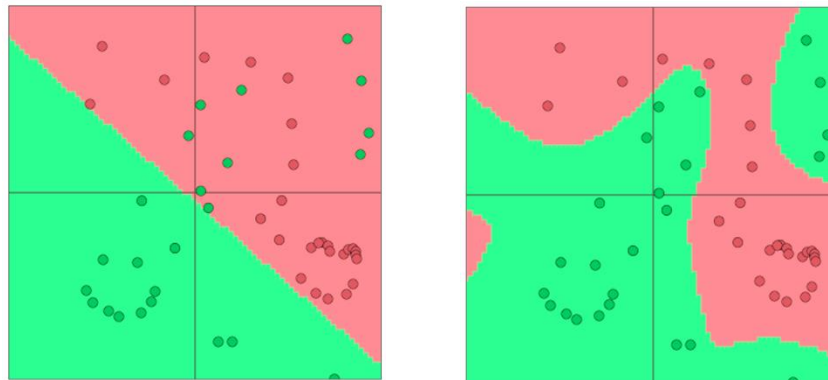
GELU [arXiv:1606.08415](https://arxiv.org/abs/1606.08415)

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



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 : $-\log p(y_i|x_i)$**

Cross entropy can be used in other ways with a more interesting p , but for now just know that you'll want to use it as the loss in PyTorch

Remember: Stochastic Gradient Descent

Update equation:

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

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

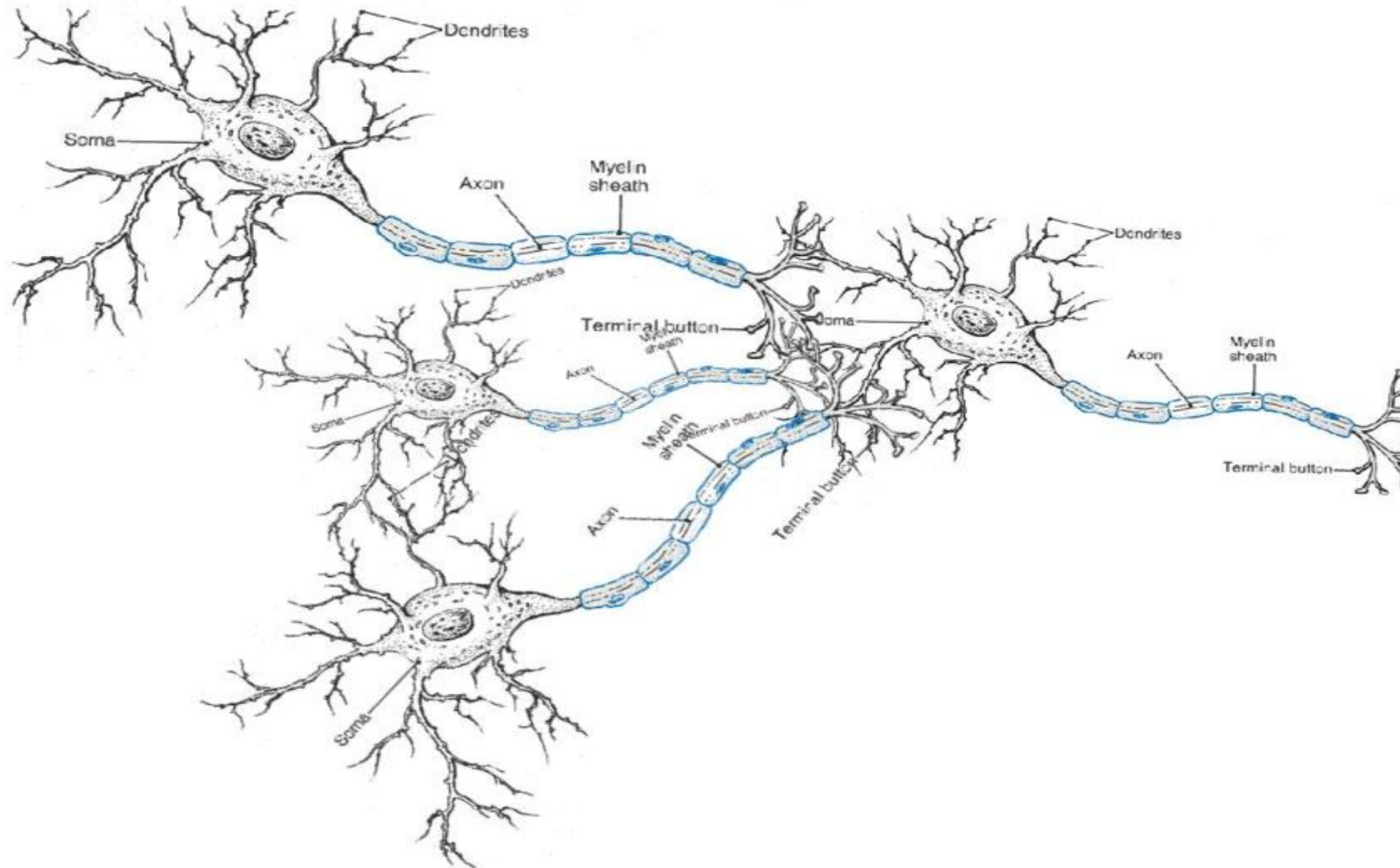
i.e., for each parameter: $\theta_j^{new} = \theta_j^{old} - \alpha \frac{\partial J(\theta)}{\partial \theta_j^{old}}$

In deep learning, 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

7. Neural computation



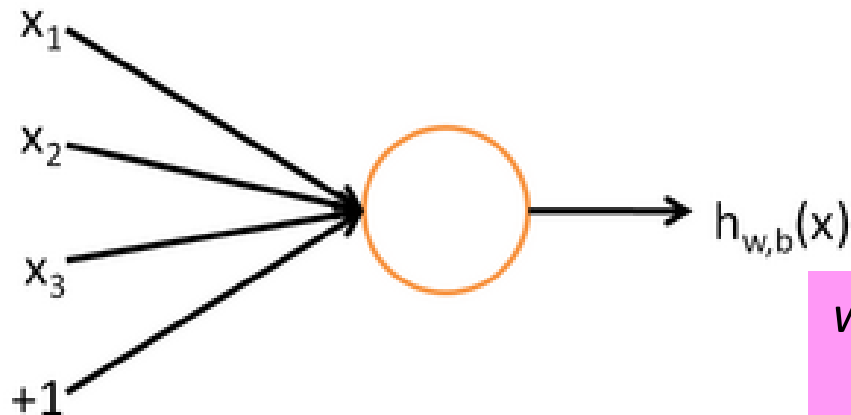
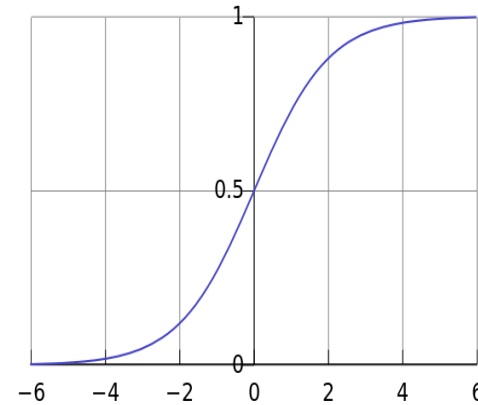
A binary logistic regression unit is a bit similar to a neuron

f = nonlinear activation function (e.g. sigmoid), w = weights, b = bias, h = hidden, x = inputs

$$h_{w,b}(x) = f(w^T x + b)$$

b : We can have an “always on” bias feature, which gives a class prior, or separate it out, as a bias term

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

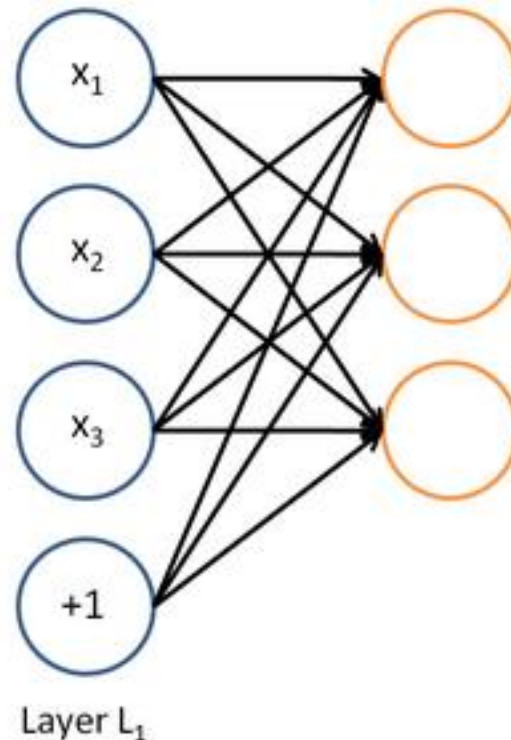


w , b are the parameters of this neuron
i.e., this logistic regression model

A neural network

= running several logistic regressions at the same time

If we feed a vector of inputs through a bunch of logistic regression functions, then we get a vector of outputs ...

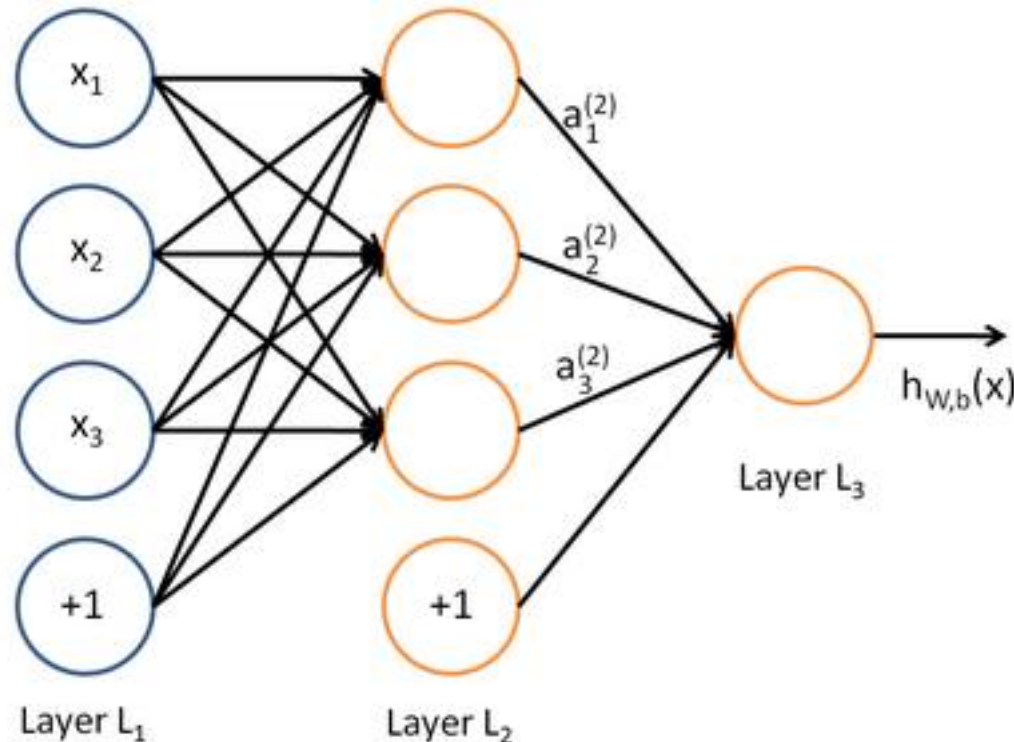


But we don't have to decide ahead of time what variables these logistic regressions are trying to predict!

A neural network

= running several logistic regressions at the same time

... which we can feed into another logistic regression function, giving composed functions

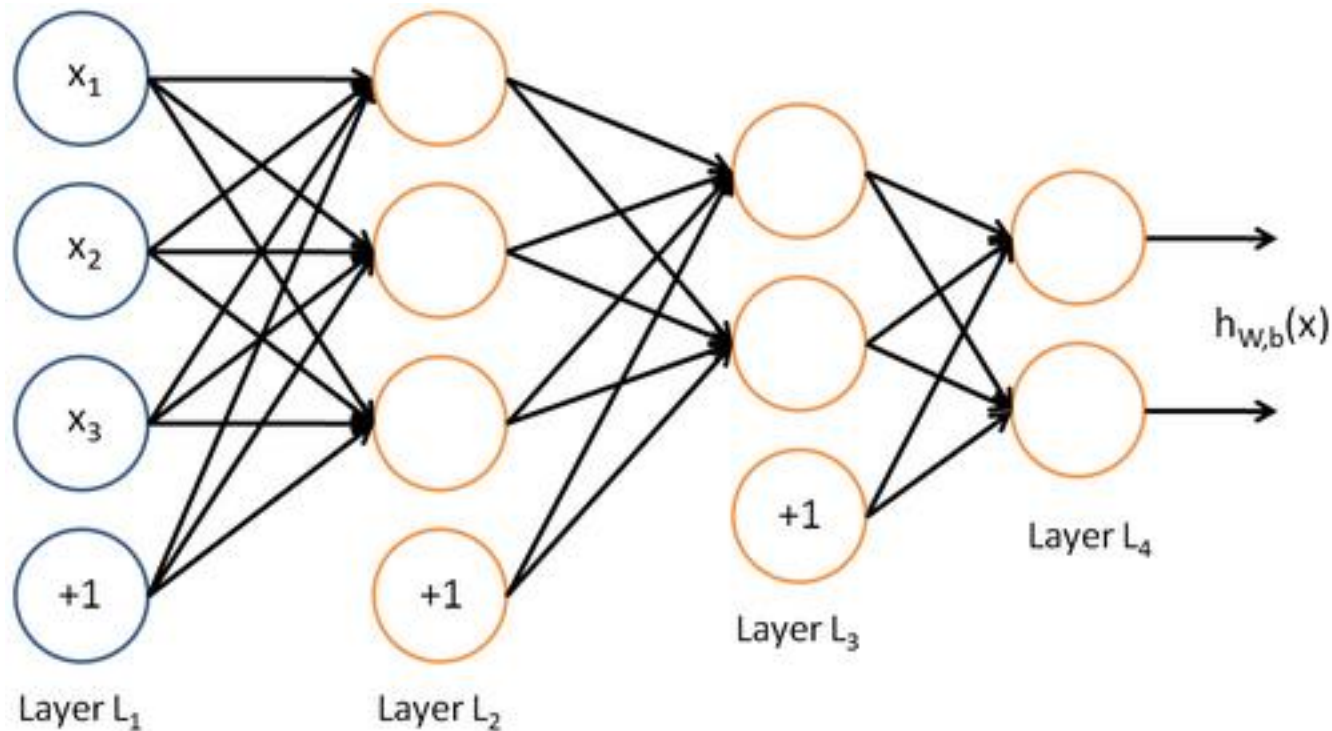


It is the loss function that will direct what the intermediate hidden variables should be, so as to do a good job at predicting the targets for the next layer, etc.

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

$$a_1 = f(W_{11}x_1 + W_{12}x_2 + W_{13}x_3 + b_1)$$

$$a_2 = f(W_{21}x_1 + W_{22}x_2 + W_{23}x_3 + b_2)$$

etc.

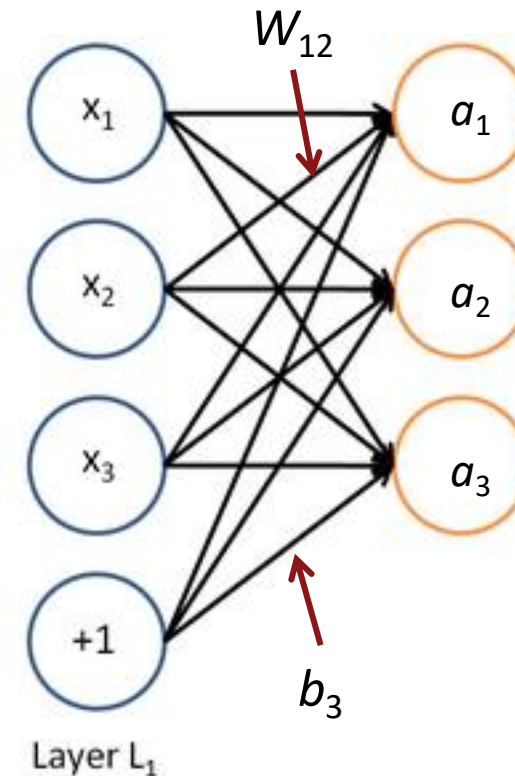
In matrix notation

$$z = Wx + b$$

$$a = f(z)$$

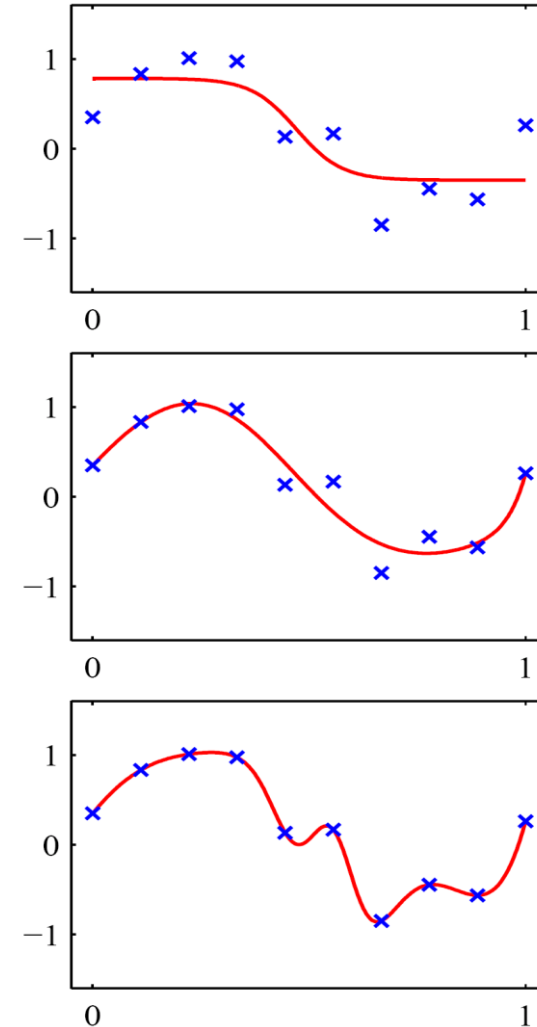
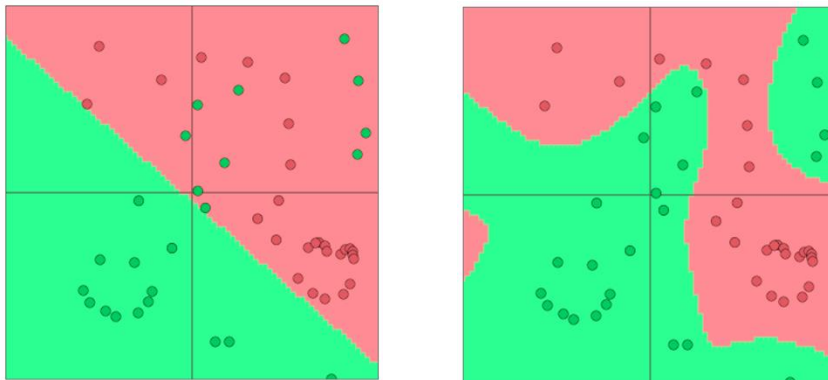
Activation f is applied element-wise:

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



Non-linearities (like f or sigmoid): 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 more complex functions!



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>

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

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

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

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

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

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

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

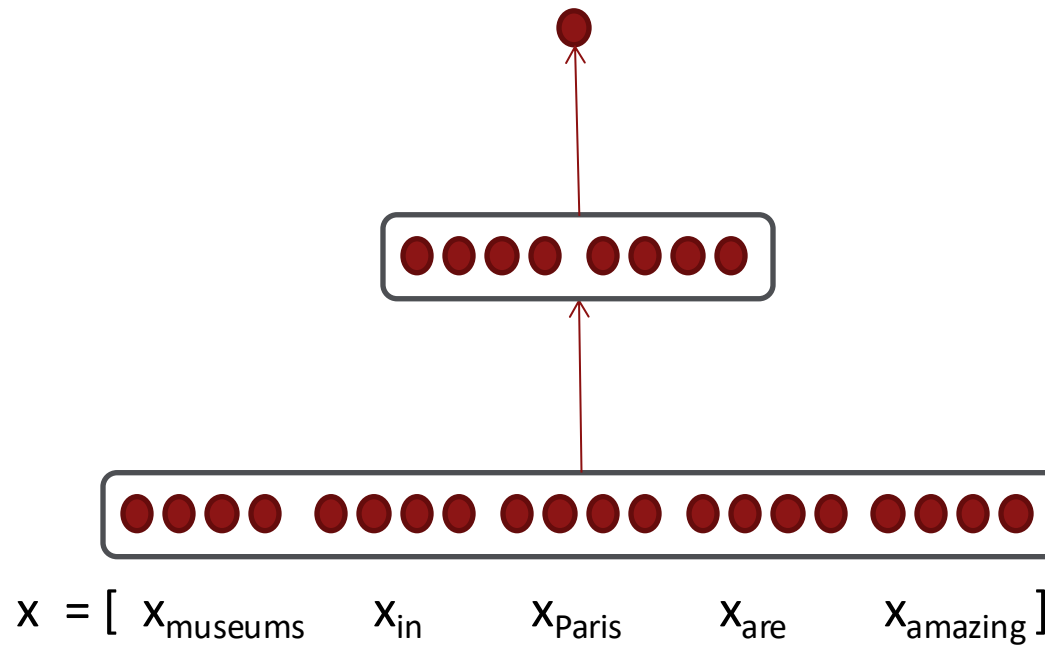
- Compute these at home for practice!
 - Check your answers with the lecture notes

Back to our Neural Net!

$$s = u^T h$$

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

x (input)



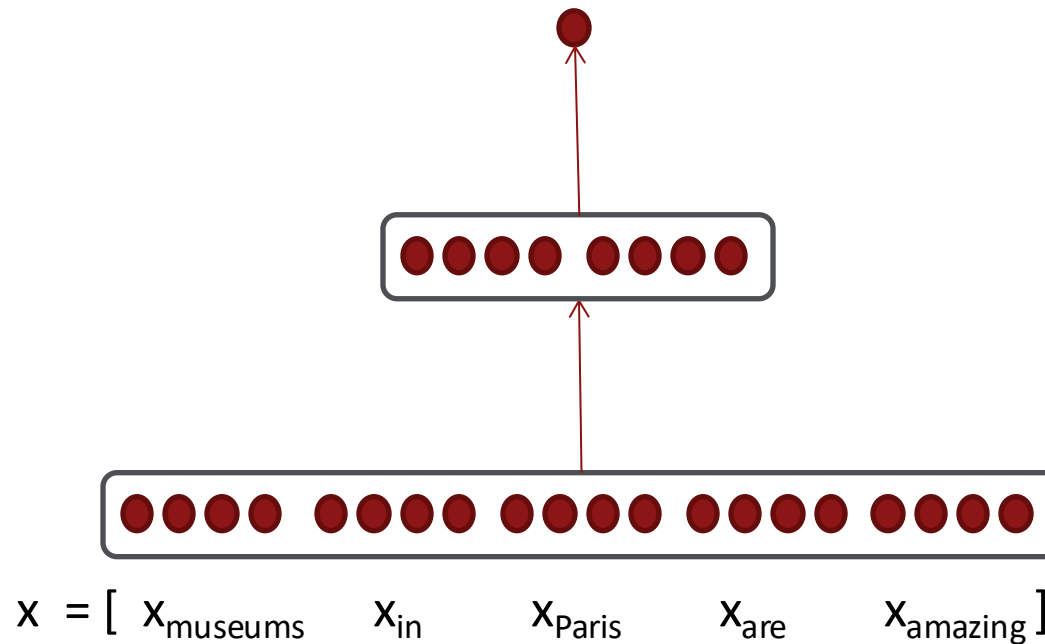
Back to our Neural Net!

- 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)



1. 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!**

2. Apply the chain rule

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

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

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

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

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

2. Apply the chain rule

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

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

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

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

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

2. 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)

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

2. 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)

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

3. 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

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

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

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

3. 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

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

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

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

3. 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

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

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

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

3. 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

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

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

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

3. Write out the Jacobians

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

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

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

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

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

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:

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

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:

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

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

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

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

δ 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

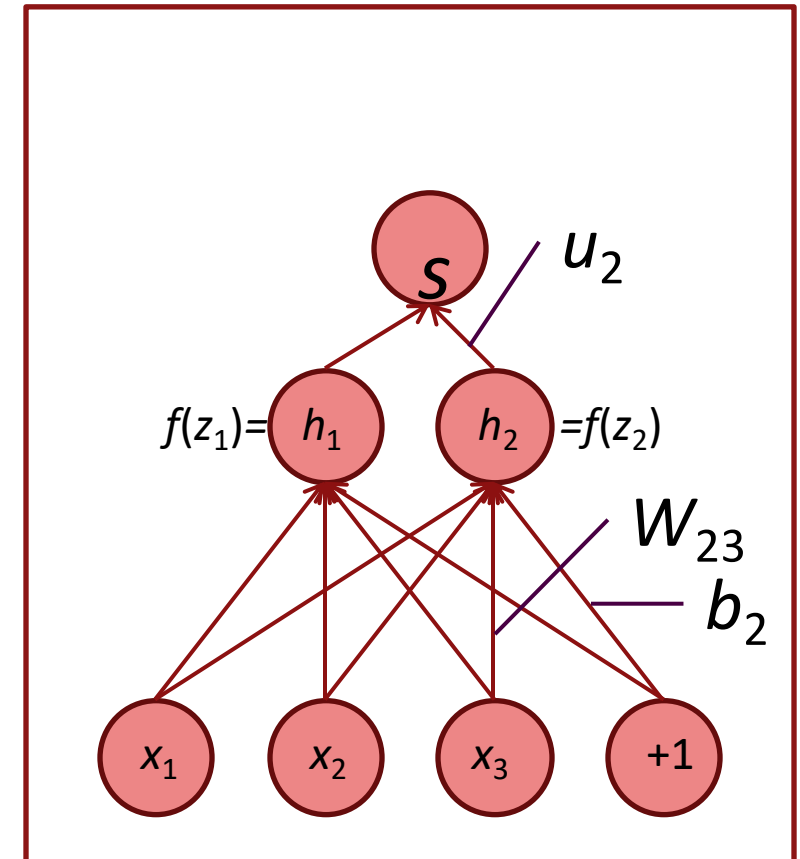
Deriving local input gradient in backprop

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

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

- 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

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



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 δ^T
2. Always follow the shape convention
 - Look at dimensions to figure out when to transpose and/or reorder terms
 - The error message δ that arrives at a hidden layer has the same dimensionality as that hidden layer

Introduction to NLP

CSE5321/CSEG321

Lecture 6. Neural Networks 2

Hwaran Lee (hwaranlee@sogang.ac.kr)

3. 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

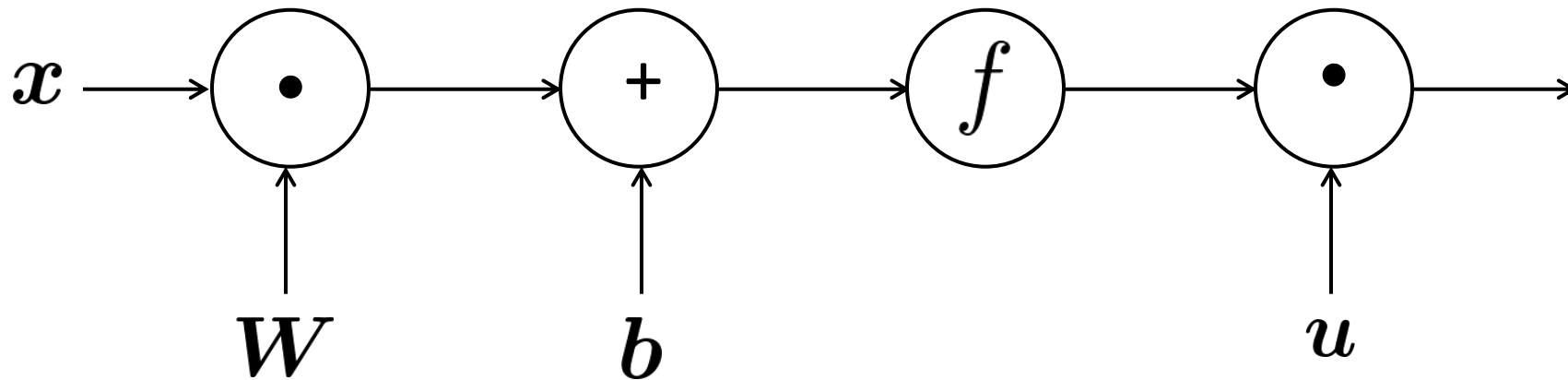
- Software represents our neural net equations as a graph
 - Source nodes: inputs
 - Interior nodes: operations

$$s = u^T h$$

$$h = f(z)$$

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

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



Computation Graphs and Backpropagation

- Software represents our neural net equations as a graph

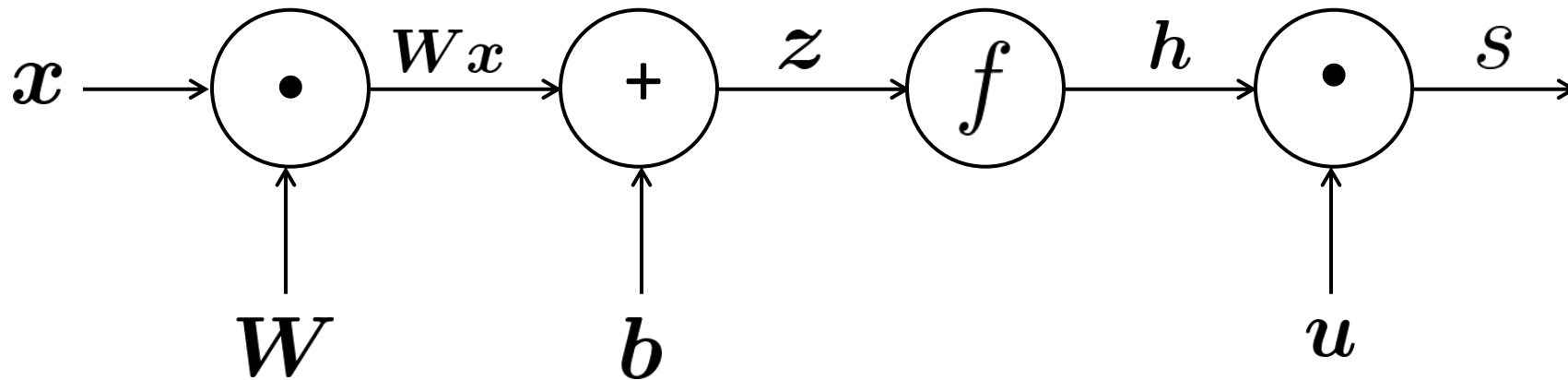
- Source nodes: inputs
- Interior nodes: operations
- Edges pass along result of the operation

$$s = u^T h$$

$$h = f(z)$$

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

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



Computation Graphs and Backpropagation

- Software represents our neural net equations as a graph

$$s = u^T h$$

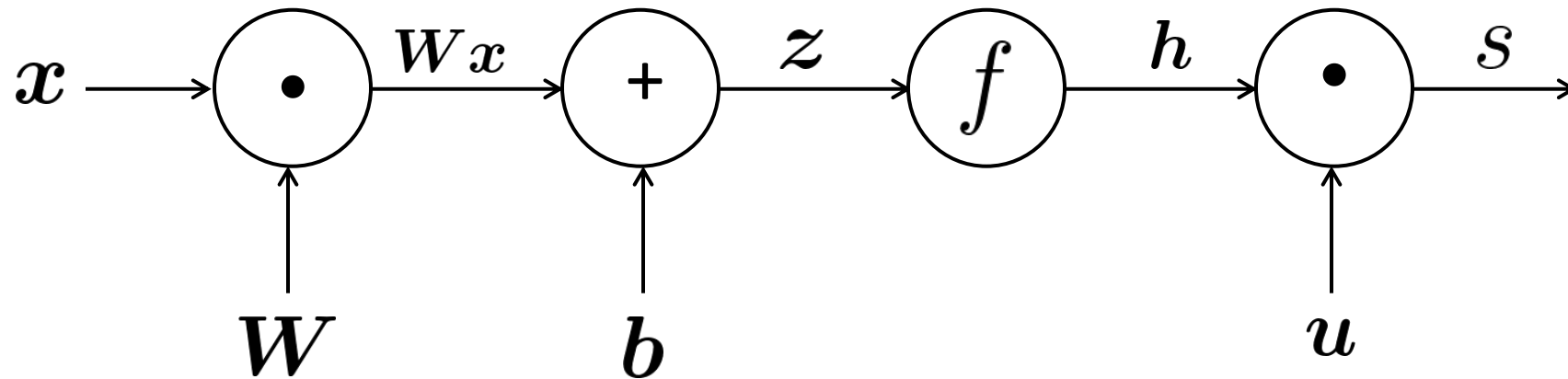
$$h = f(z)$$

$$z = c + b$$

(ut)

“Forward Propagation”

operation



Backpropagation

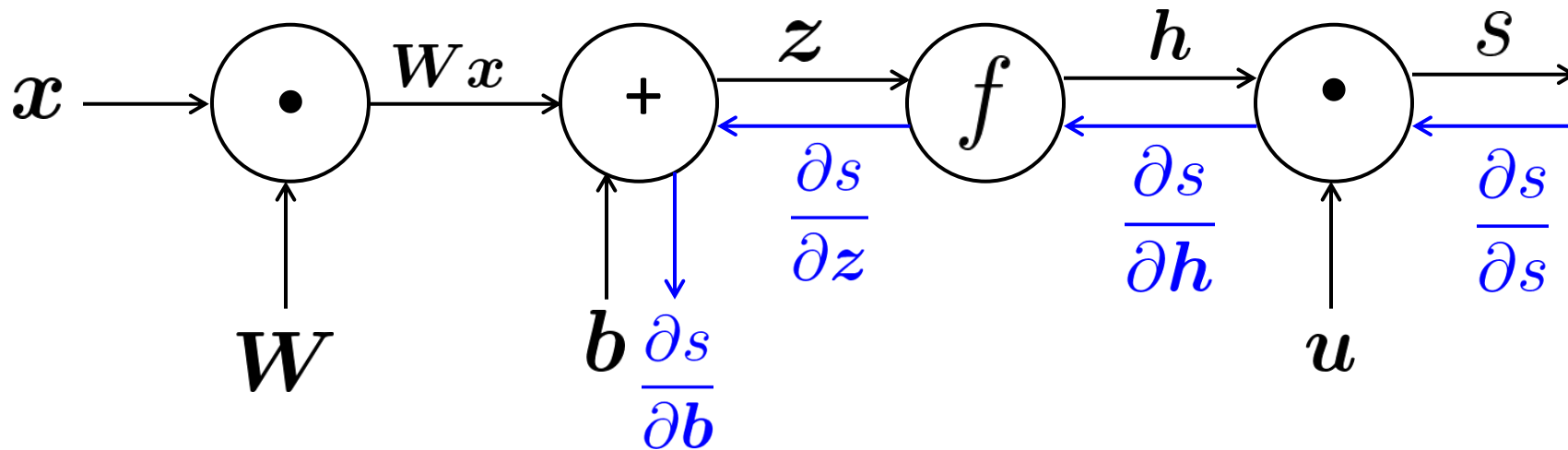
- Then go backwards along edges
 - Pass along **gradients**

$$s = u^T h$$

$$h = f(z)$$

$$z = Wx + b$$

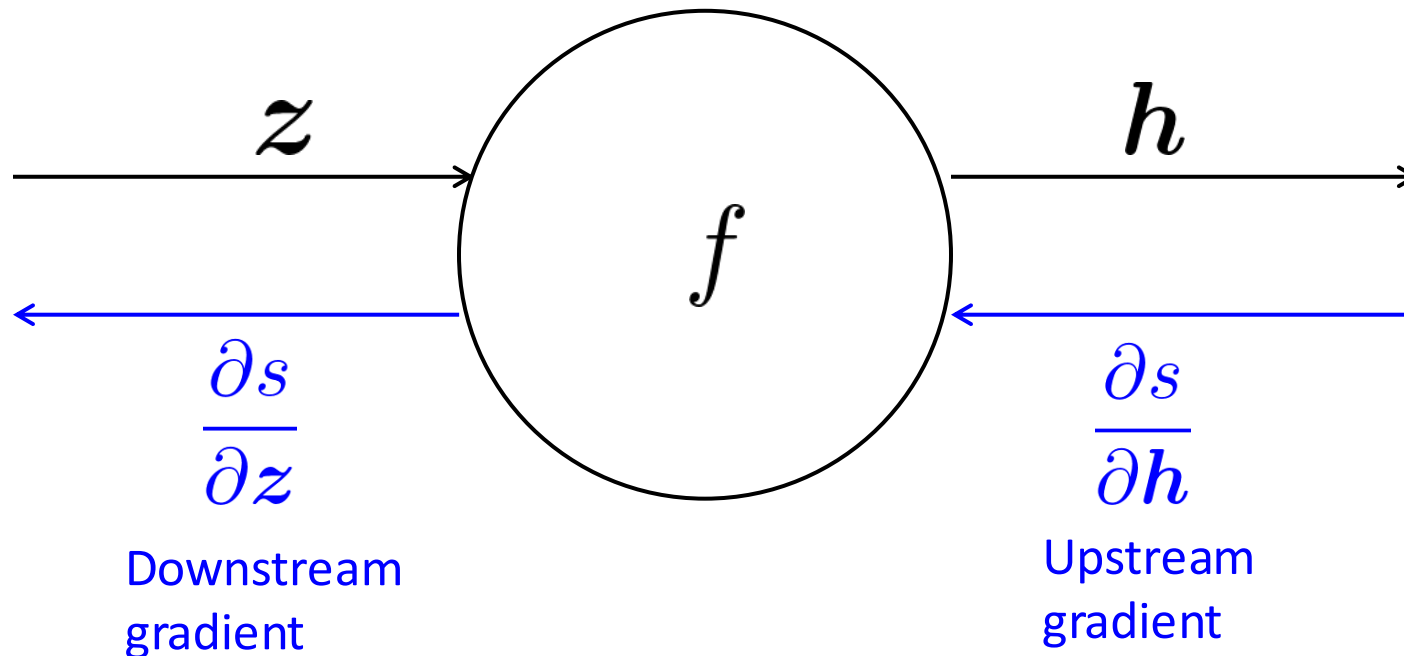
$$x \text{ (input)}$$



Backpropagation: Single Node

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

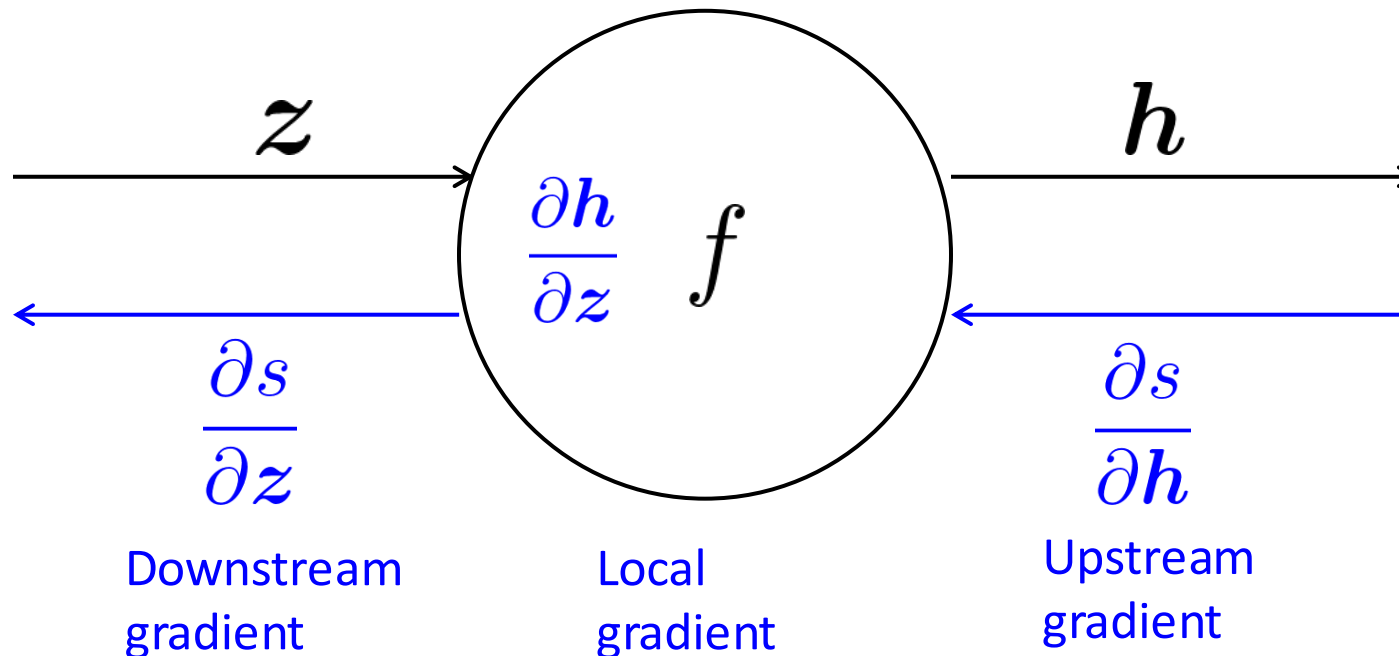
$$h = f(z)$$



Backpropagation: Single Node

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

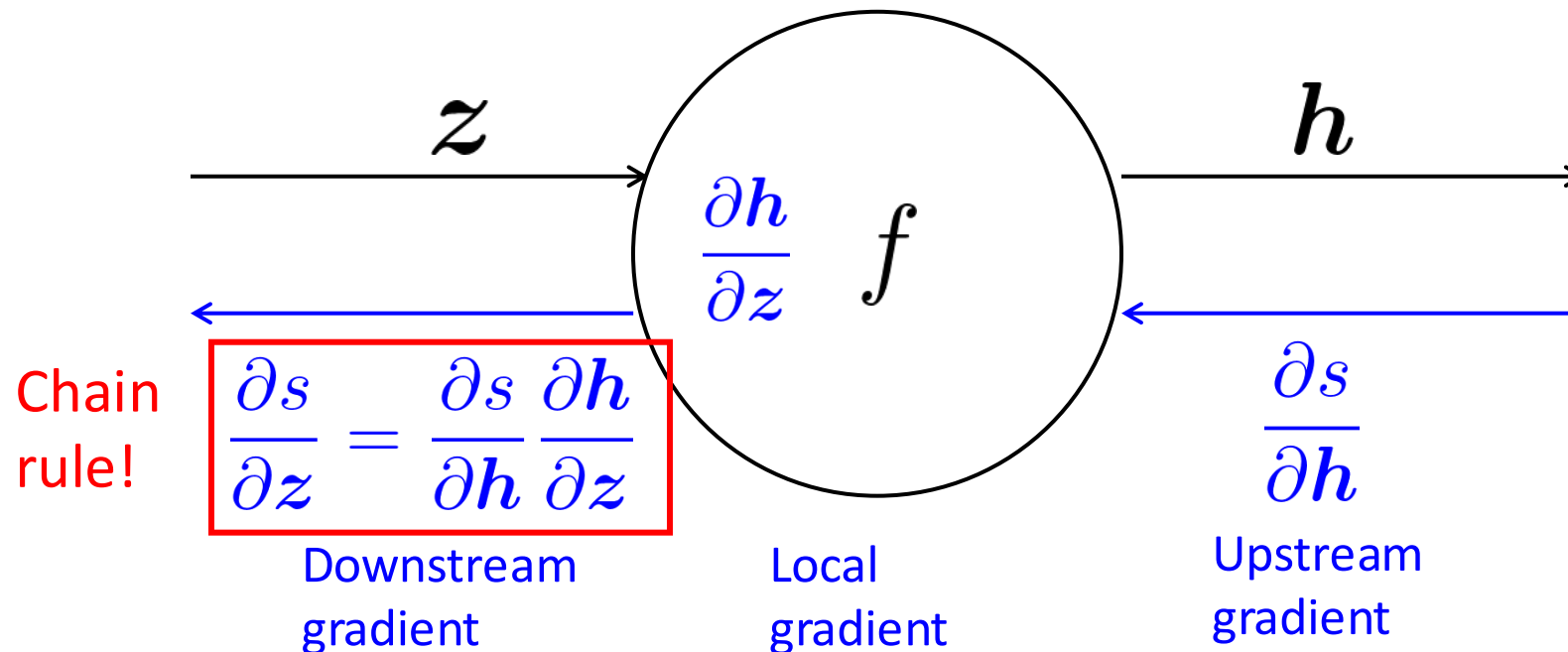
$$h = f(z)$$



Backpropagation: Single Node

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

$$h = f(z)$$

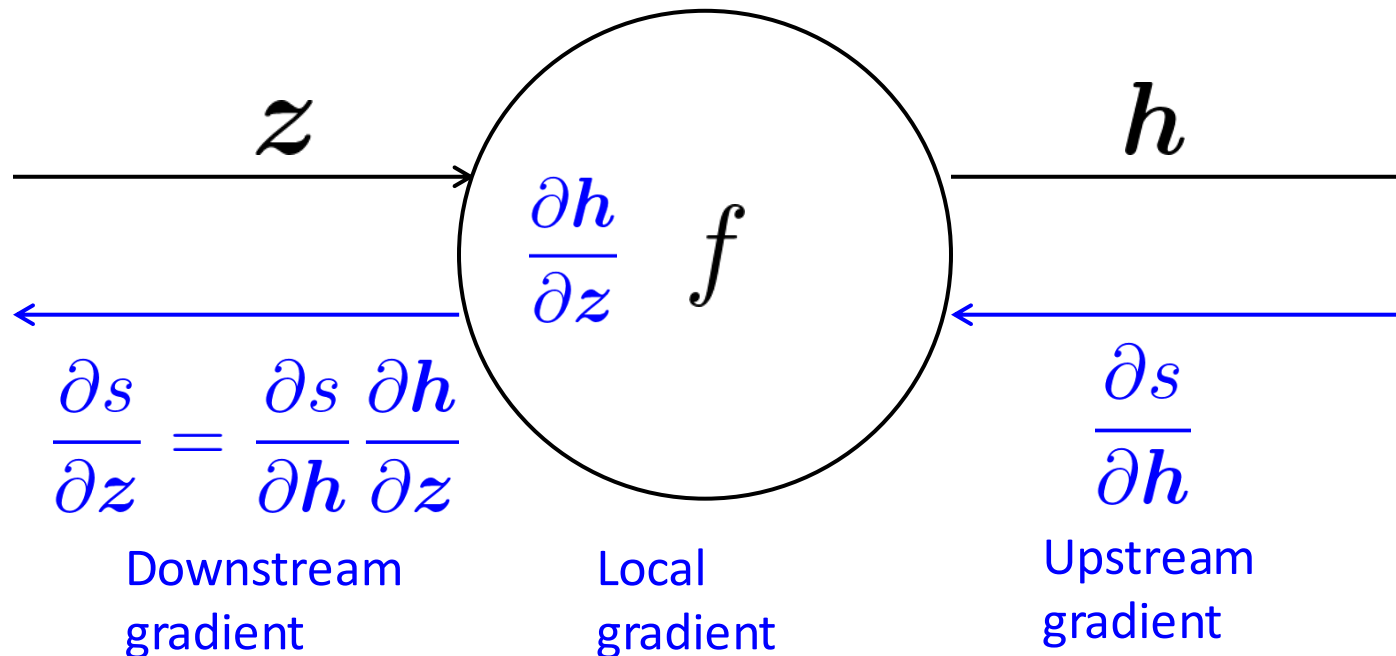


Backpropagation: Single Node

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

$$h = f(z)$$

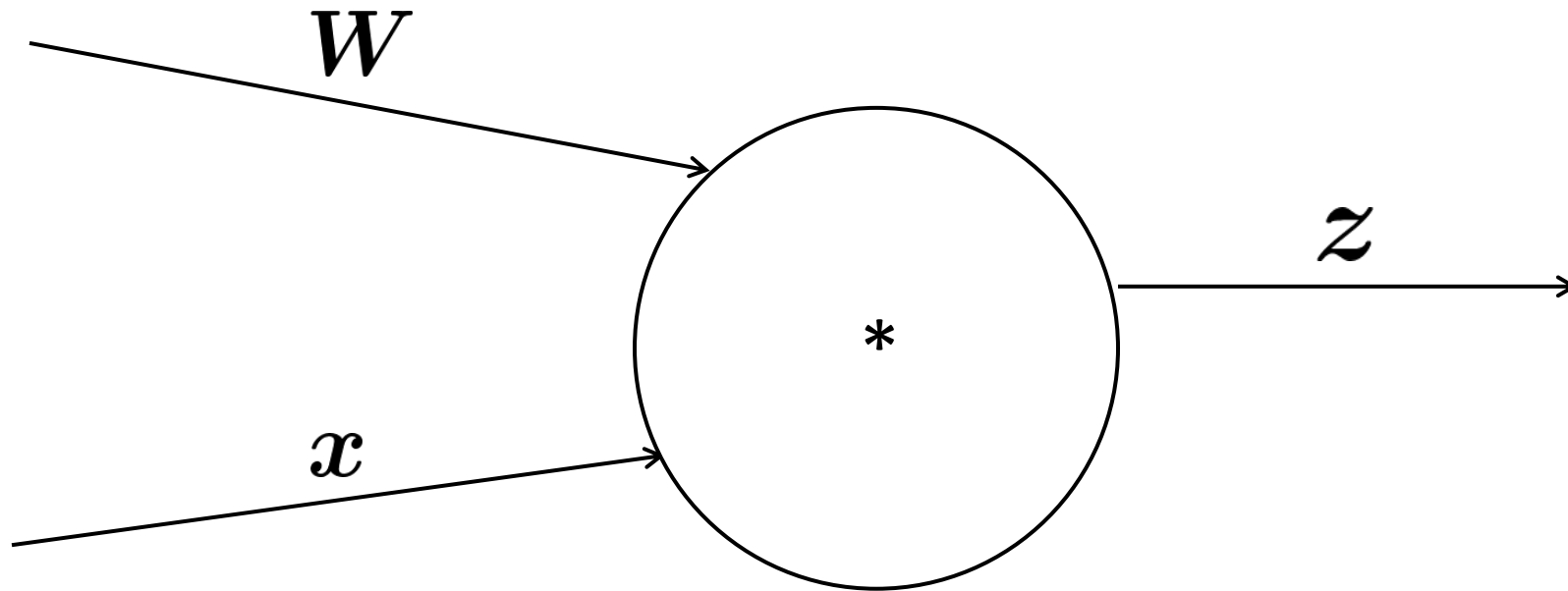
$$[\text{downstream gradient}] = [\text{upstream gradient}] \times [\text{local gradient}]$$



Backpropagation: Single Node

- What about nodes with multiple inputs?

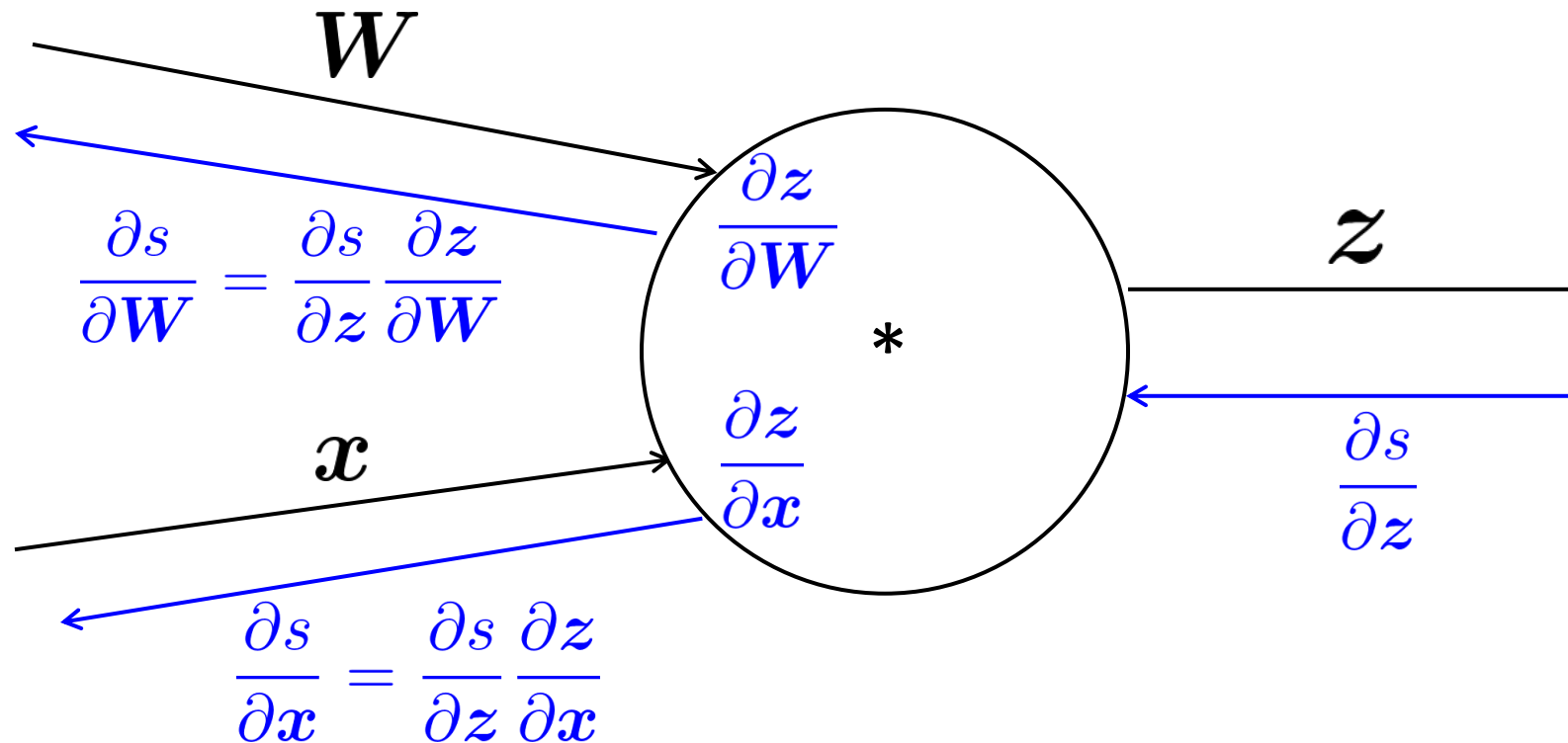
$$z = Wx$$



Backpropagation: Single Node

- Multiple inputs \rightarrow multiple local gradients

$$z = Wx$$



Downstream
gradients

Local
gradients

Upstream
gradient

An Example

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

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

An Example

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

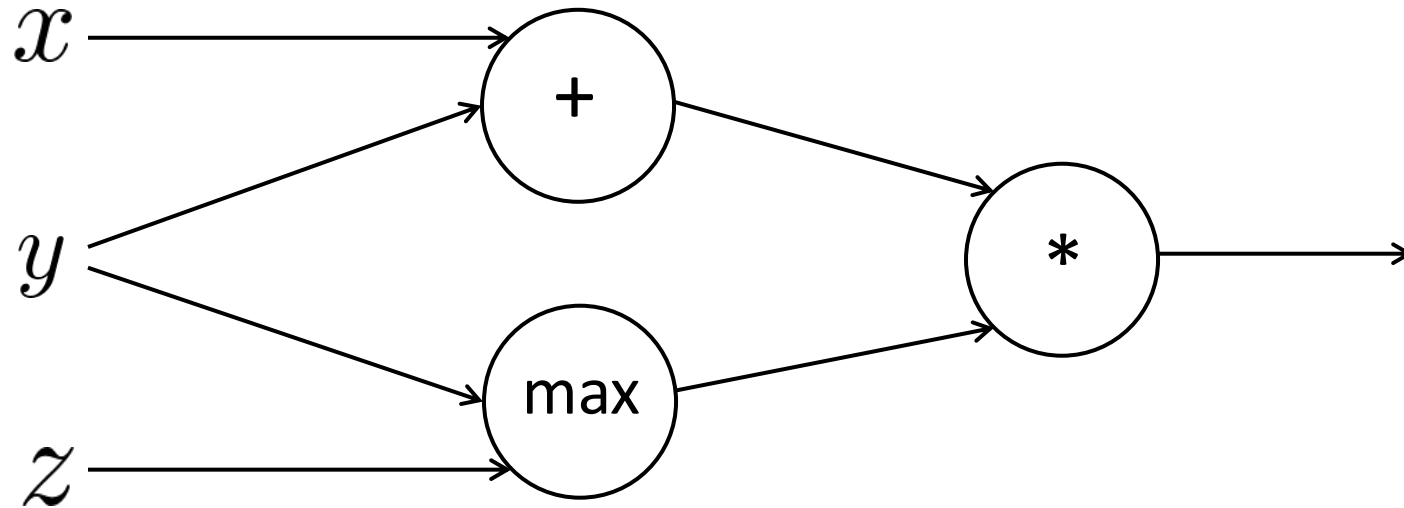
$$x = 1, y = 2, z = 0$$

Forward prop steps

$$a = x + y$$

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

$$f = ab$$



An Example

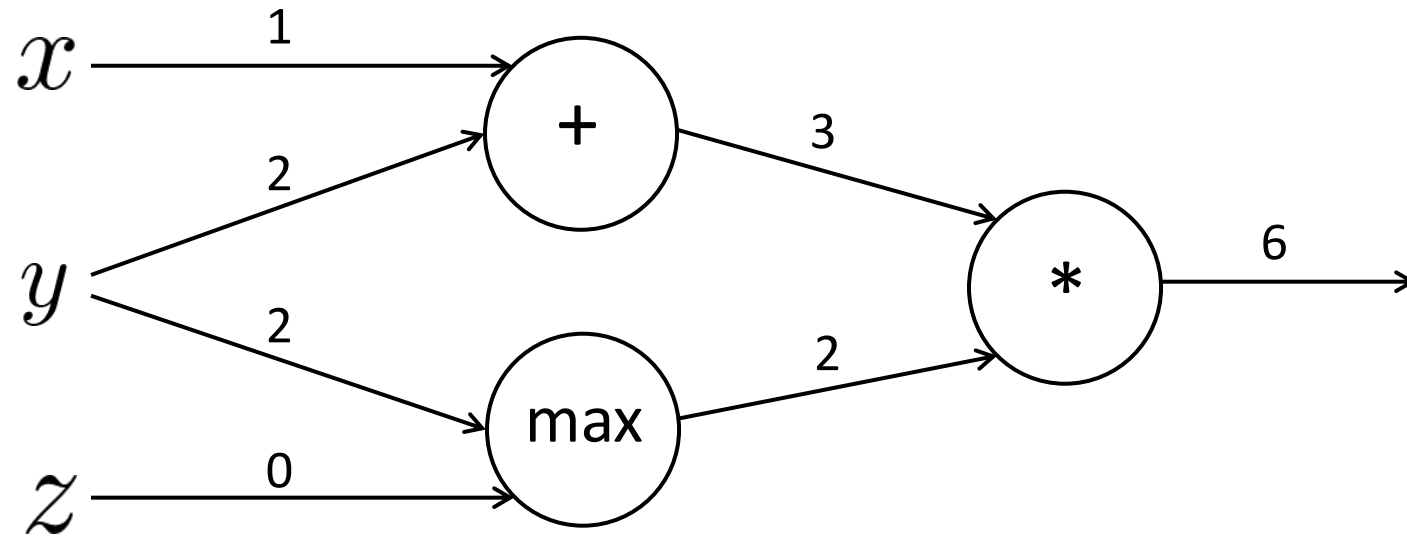
$$f(x, y, z) = (x + y) \max(y, z)$$
$$x = 1, y = 2, z = 0$$

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

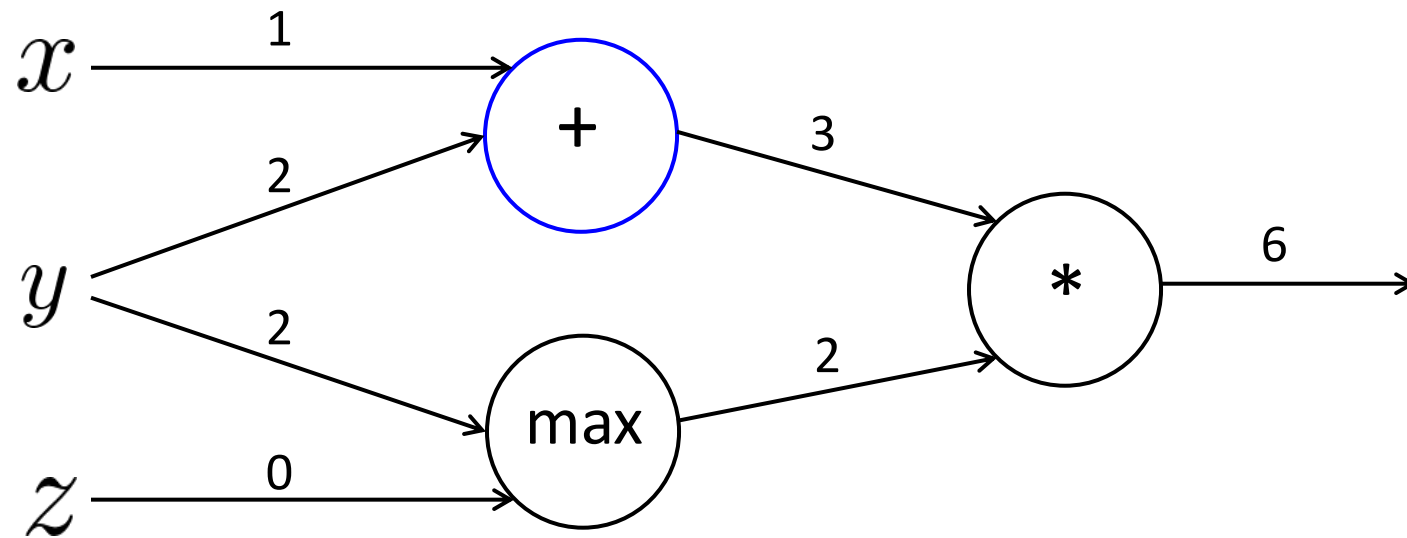
$$a = x + y$$

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

$$f = ab$$

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

$$a = x + y$$

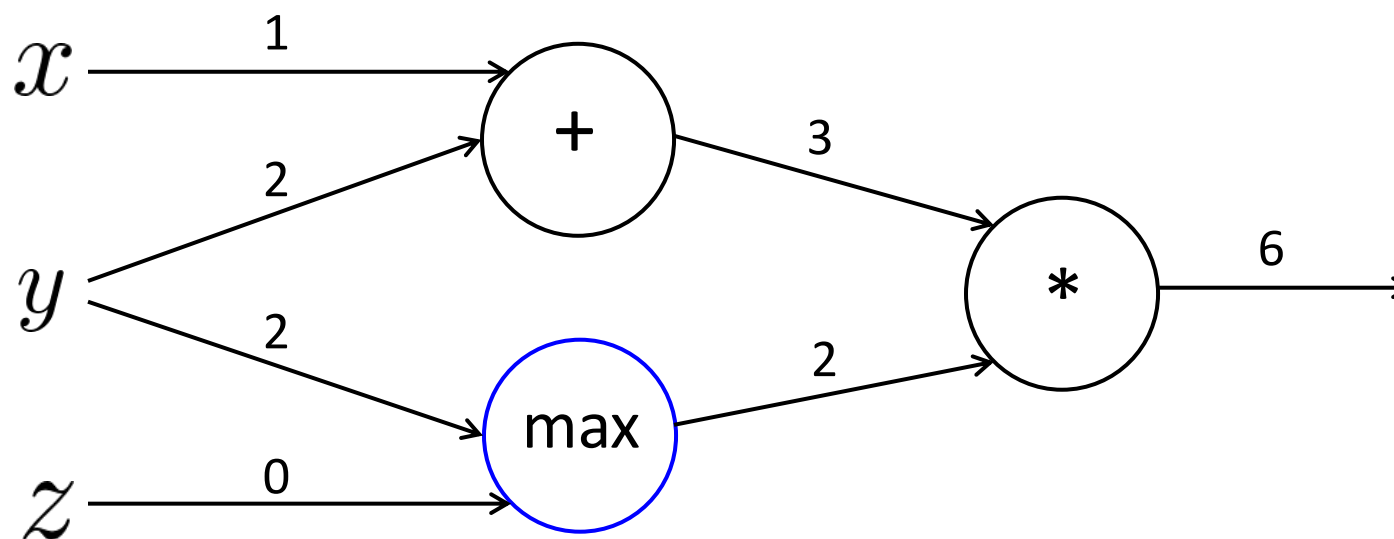
$$b = \max(y, z)$$

$$f = ab$$

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

$$a = x + y$$

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

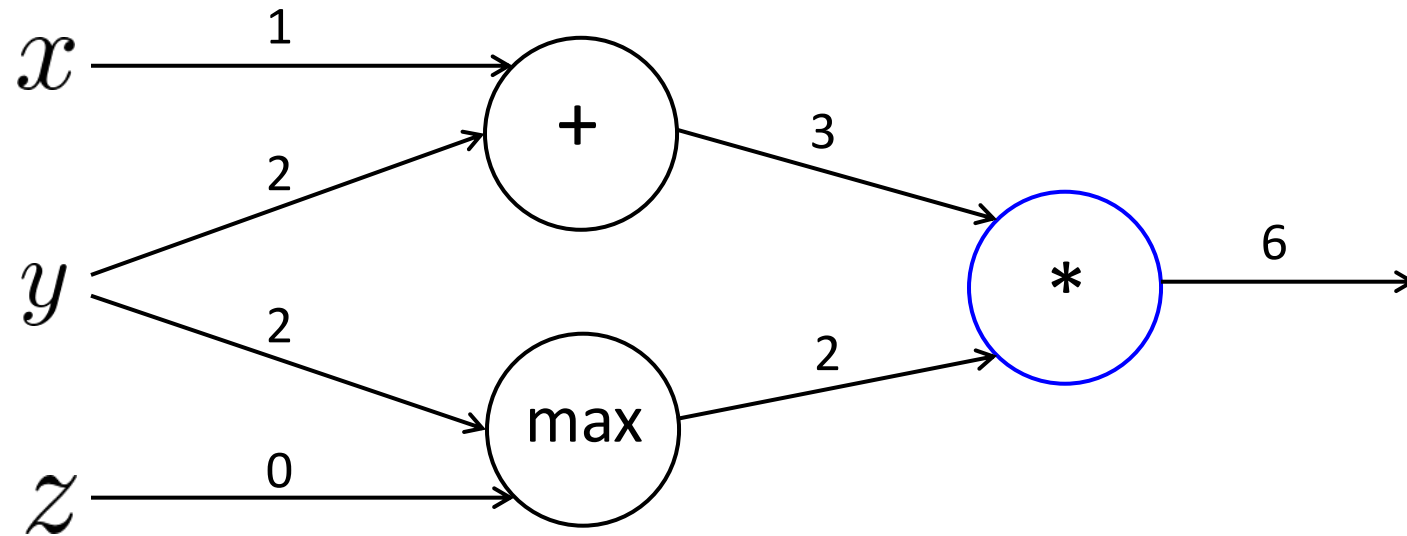
$$f = ab$$

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

$$a = x + y$$

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

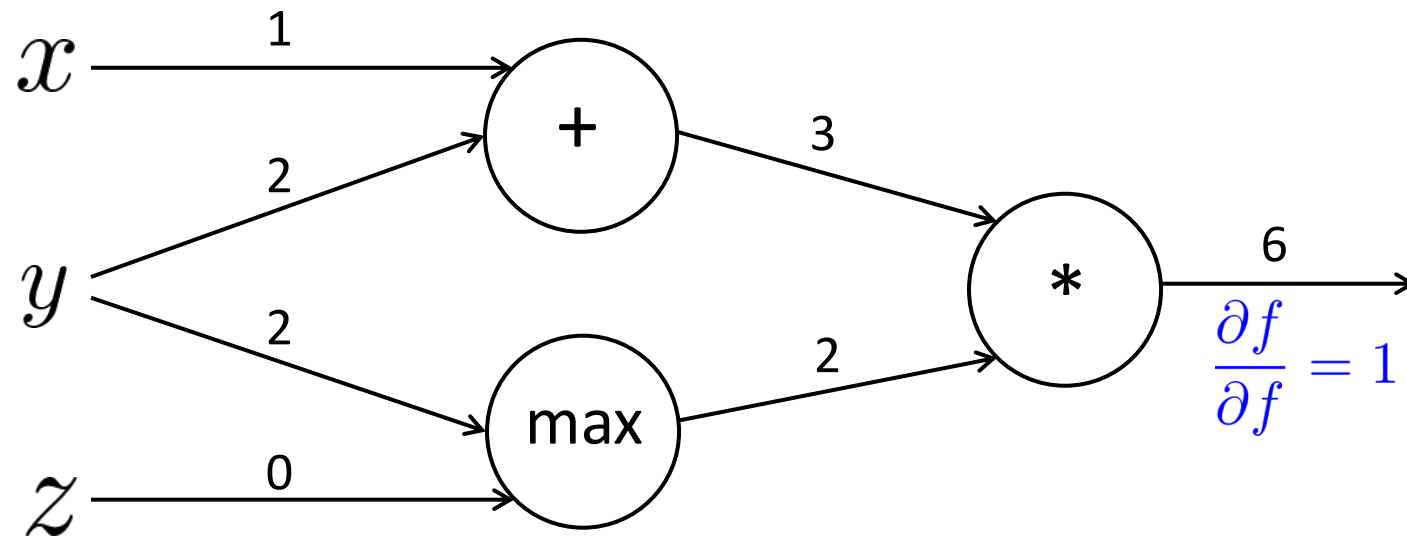
$$f = ab$$

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

$$a = x + y$$

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

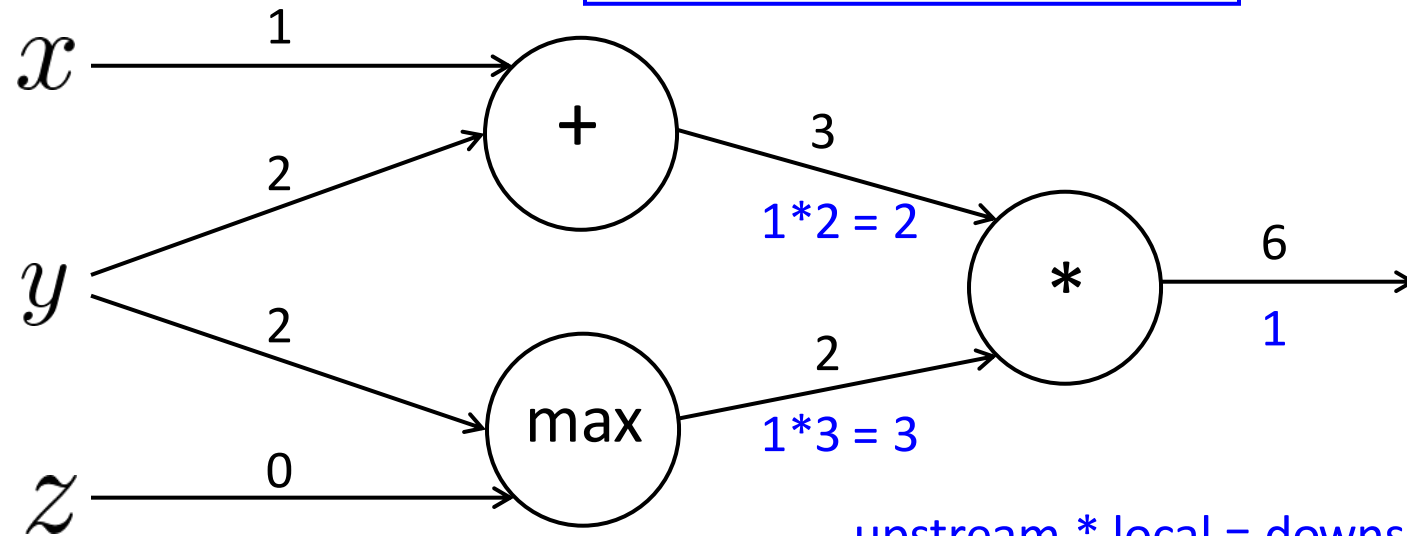
$$f = ab$$

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

$$a = x + y$$

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

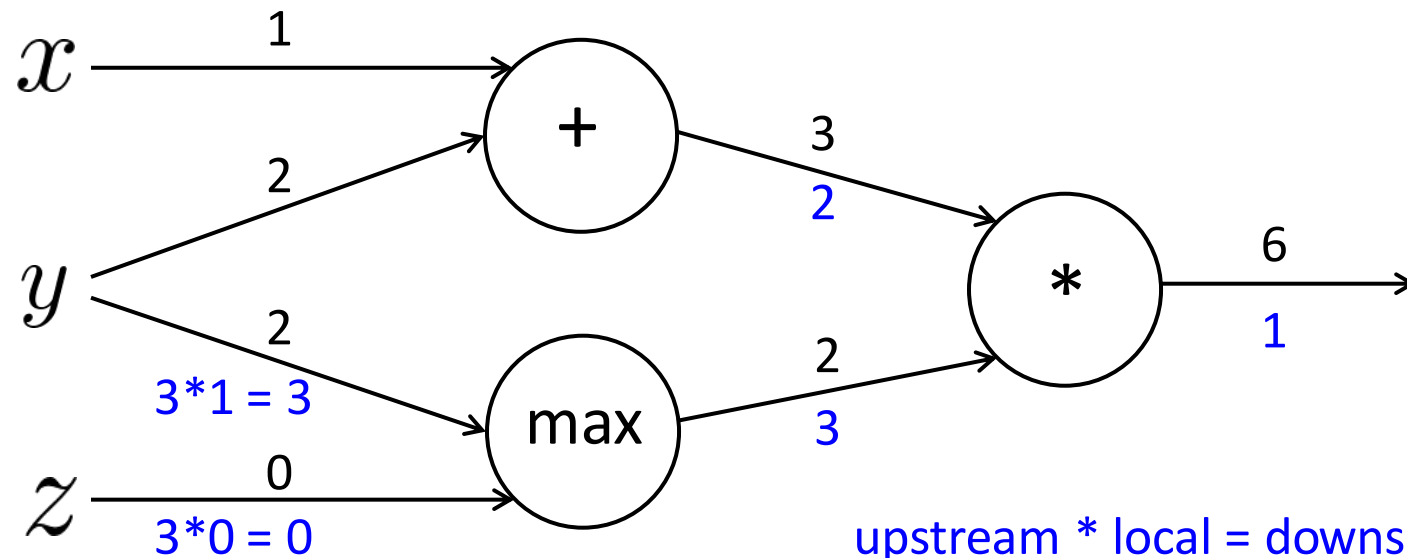
$$f = ab$$

Local gradients

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

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

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



An Example

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

Forward prop steps

$$a = x + y$$

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

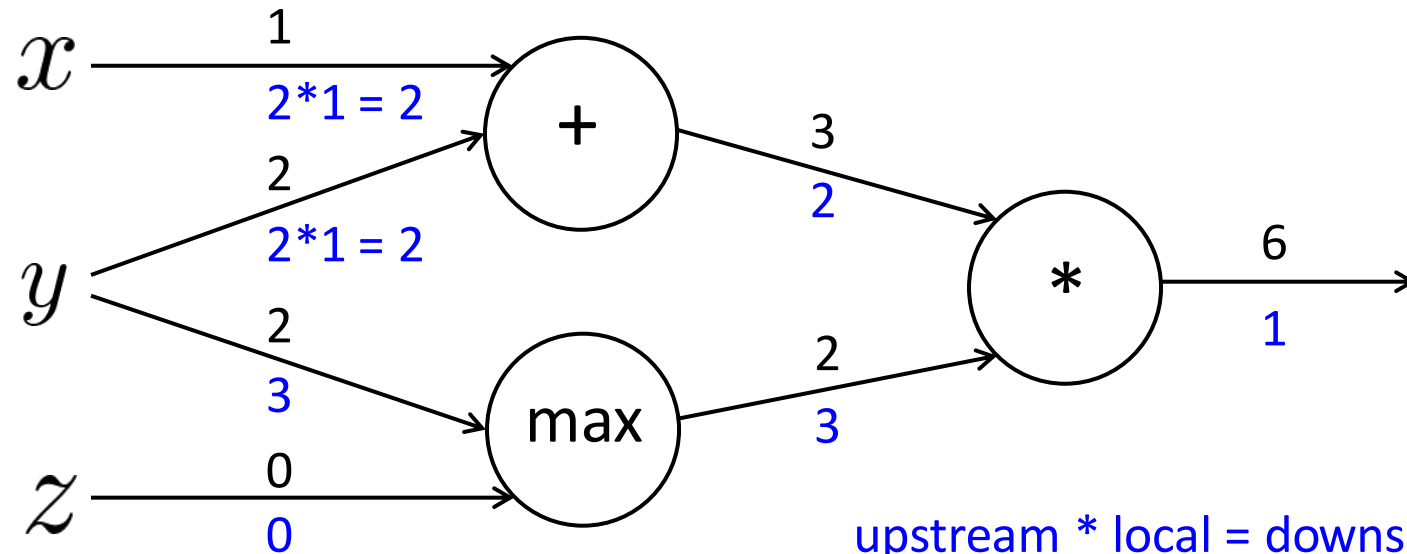
$$f = ab$$

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

$$a = x + y$$

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

$$f = ab$$

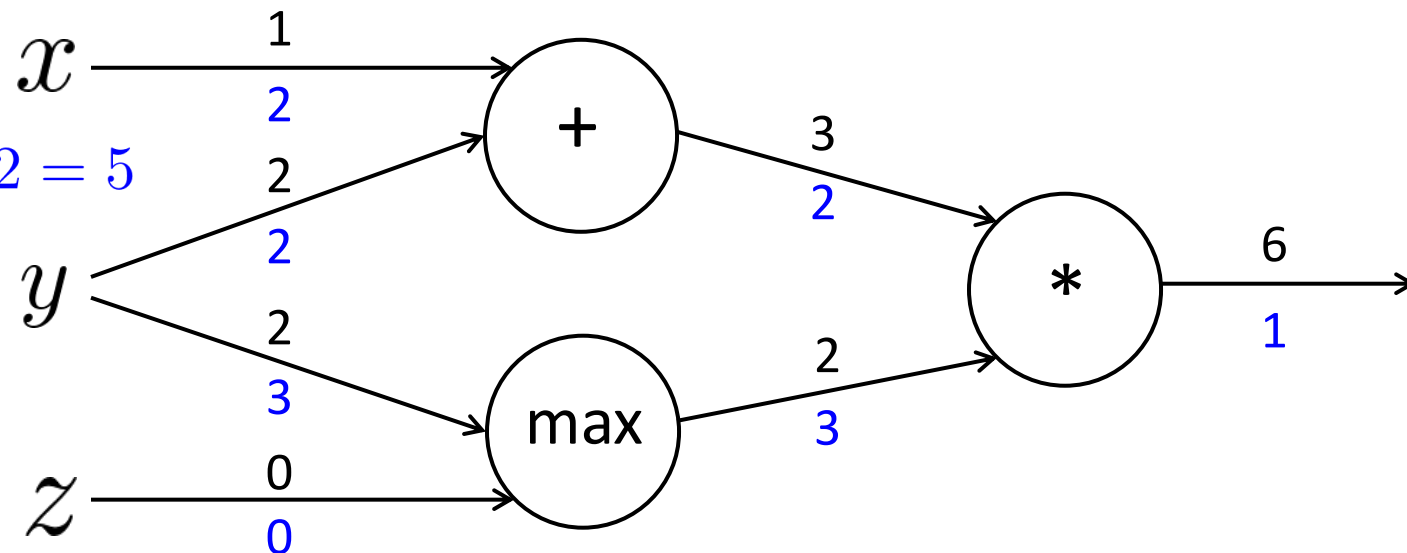
Local gradients

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

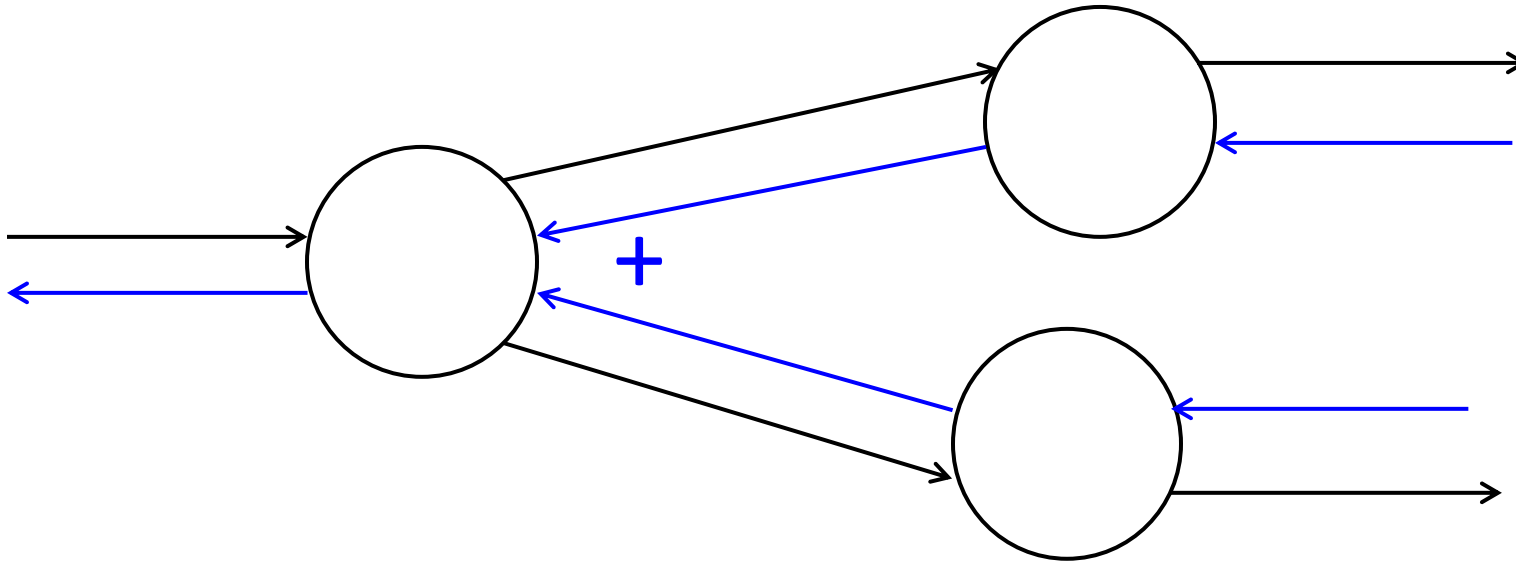
$$\frac{\partial b}{\partial y} = \mathbf{1}(y > z) = 1 \quad \frac{\partial b}{\partial z} = \mathbf{1}(z > y) = 0$$

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

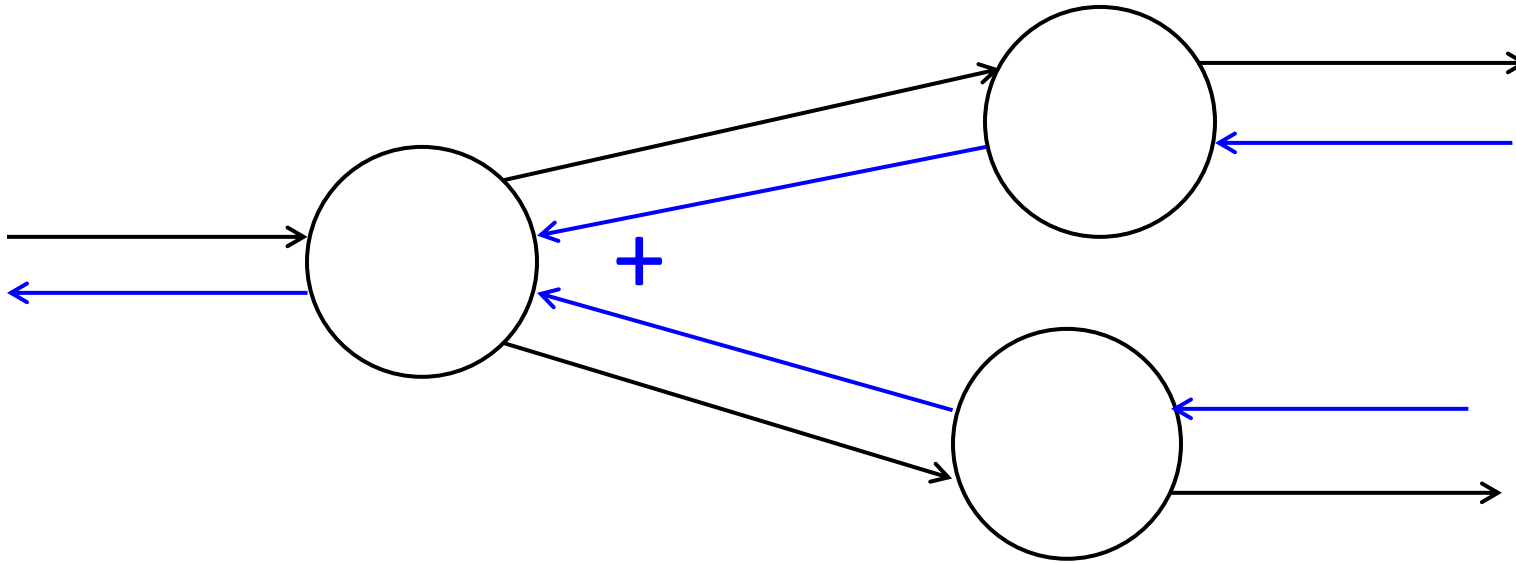
$$\frac{\partial f}{\partial x} = 2$$
$$\frac{\partial f}{\partial y} = 3 + 2 = 5$$
$$\frac{\partial f}{\partial z} = 0$$



Gradients sum at outward branches



Gradients sum at outward branches



$$a = x + y$$

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

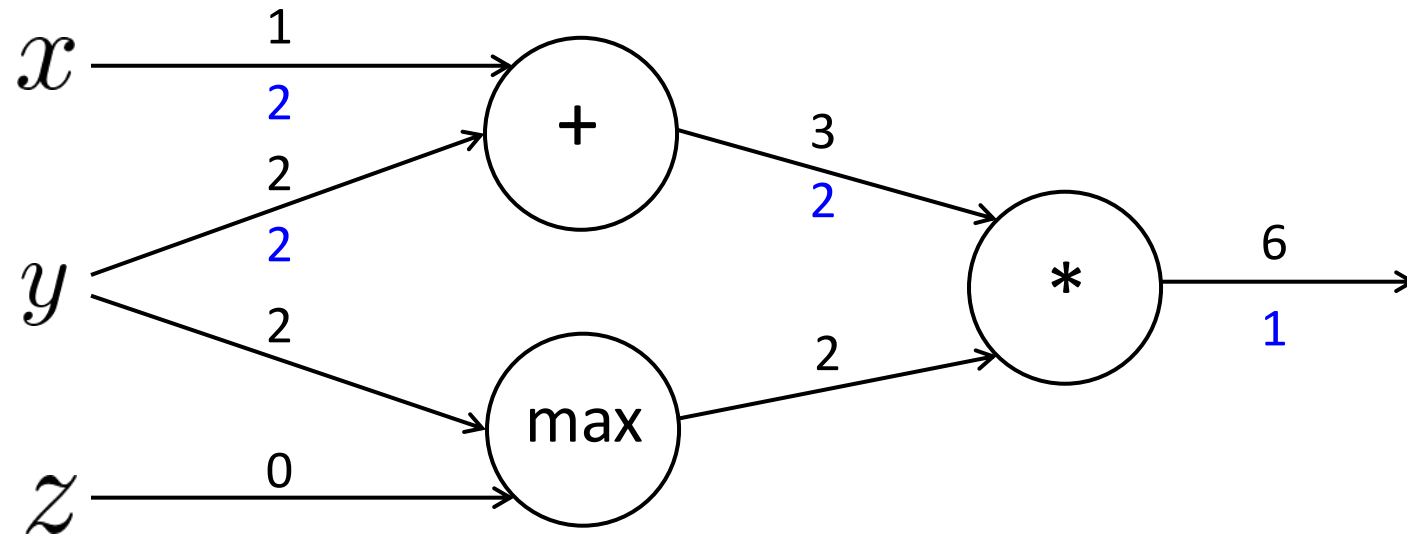
$$f = ab$$

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

Node Intuitions

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

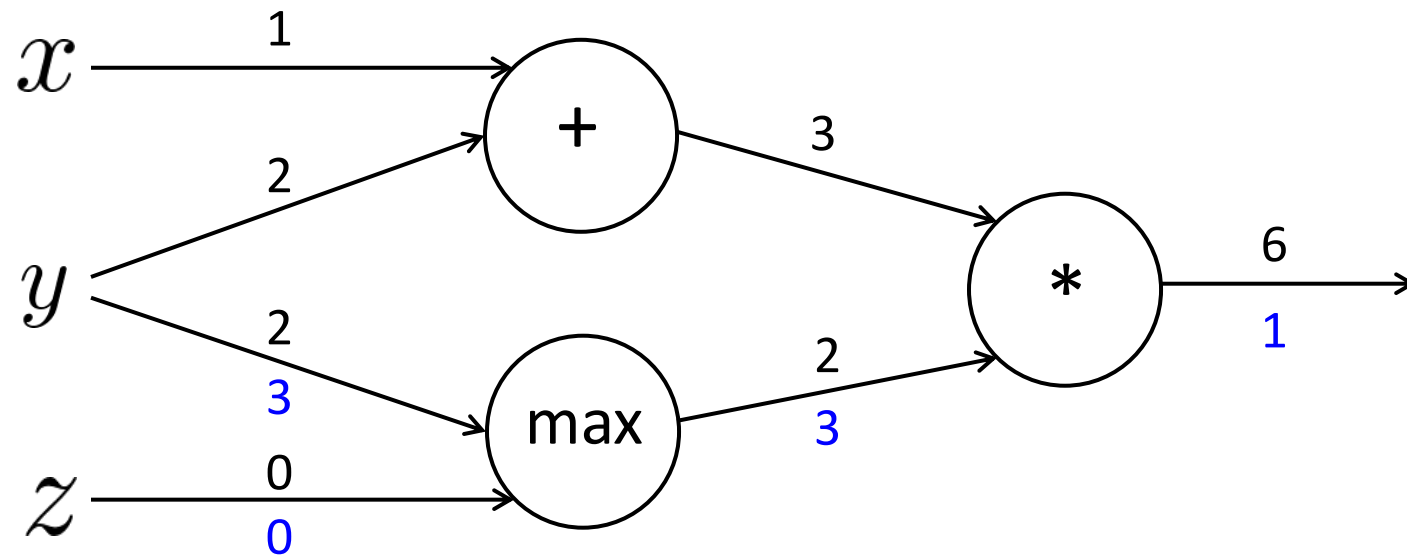
- + “distributes” the upstream gradient to each summand



Node Intuitions

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

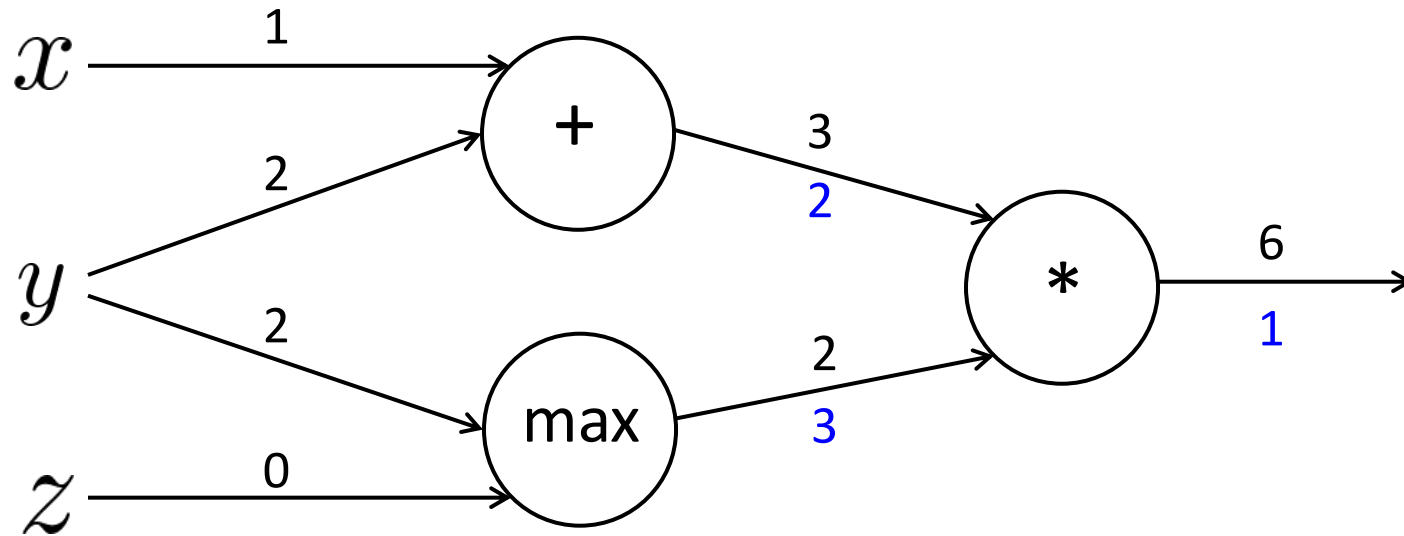
- + “distributes” the upstream gradient to each summand
- max “routes” the upstream gradient



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:

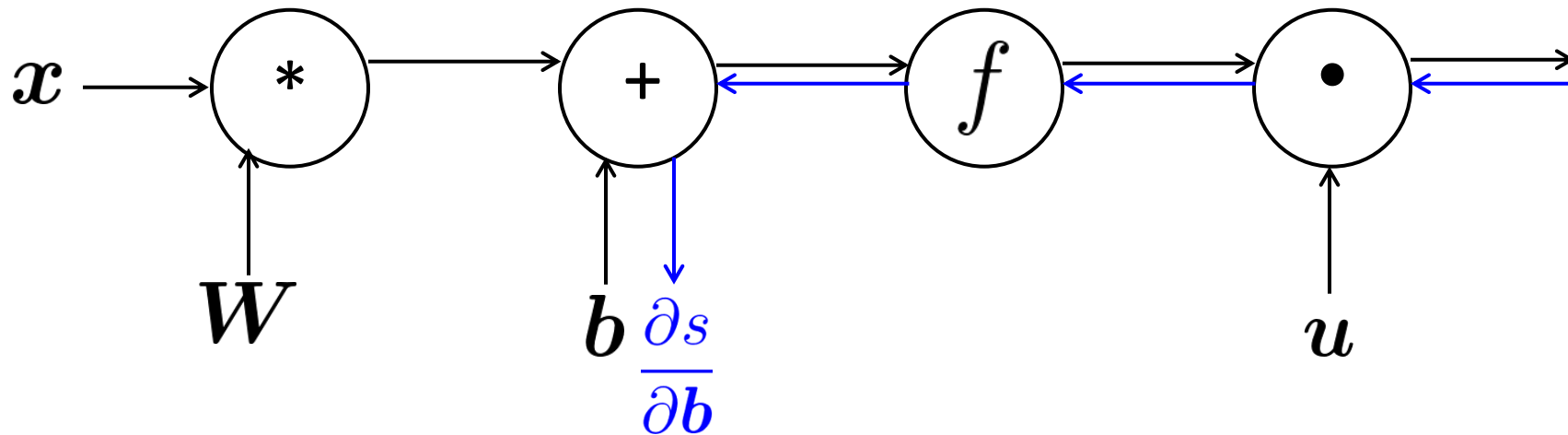
- First compute $\frac{\partial s}{\partial b}$

$$s = u^T h$$

$$h = f(z)$$

$$z = Wx + b$$

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



Efficiency: compute all gradients at once

- **Incorrect way** of doing backprop:

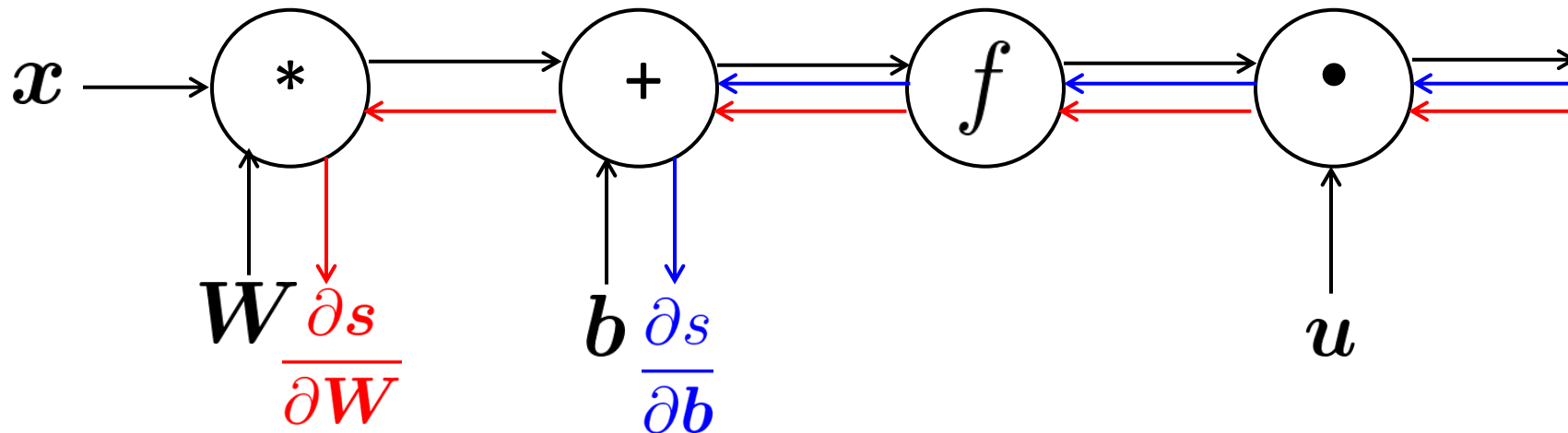
- First compute $\frac{\partial s}{\partial b}$
- Then independently compute $\frac{\partial s}{\partial W}$
- Duplicated computation!

$$s = u^T h$$

$$h = f(z)$$

$$z = Wx + b$$

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



Efficiency: compute all gradients at once

- **Correct way:**

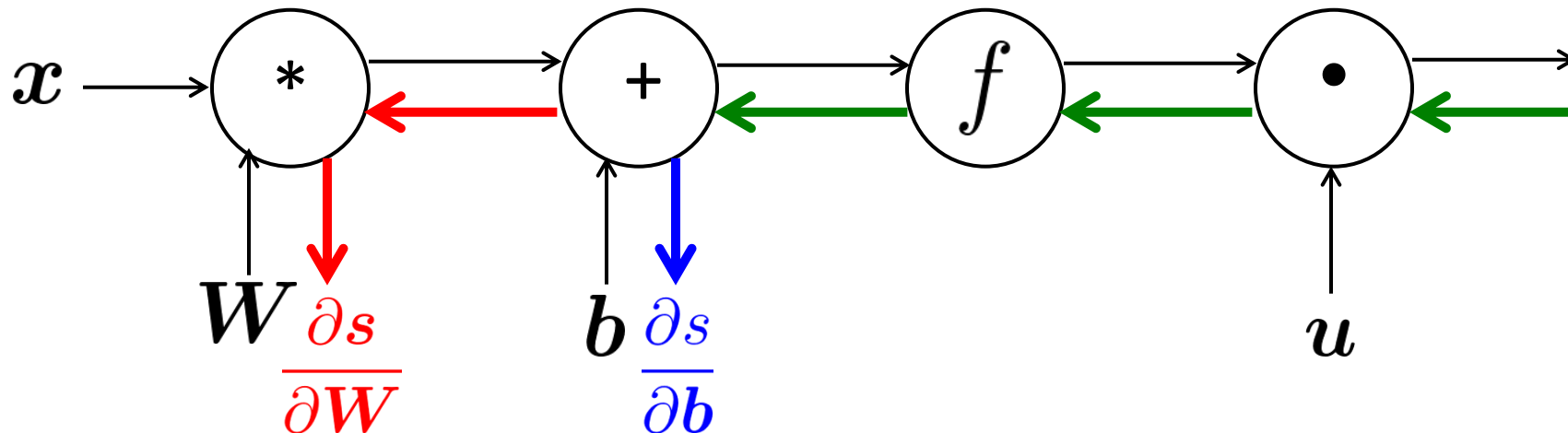
- Compute all the gradients at once
- Analogous to using δ when we computed gradients by hand

$$s = u^T h$$

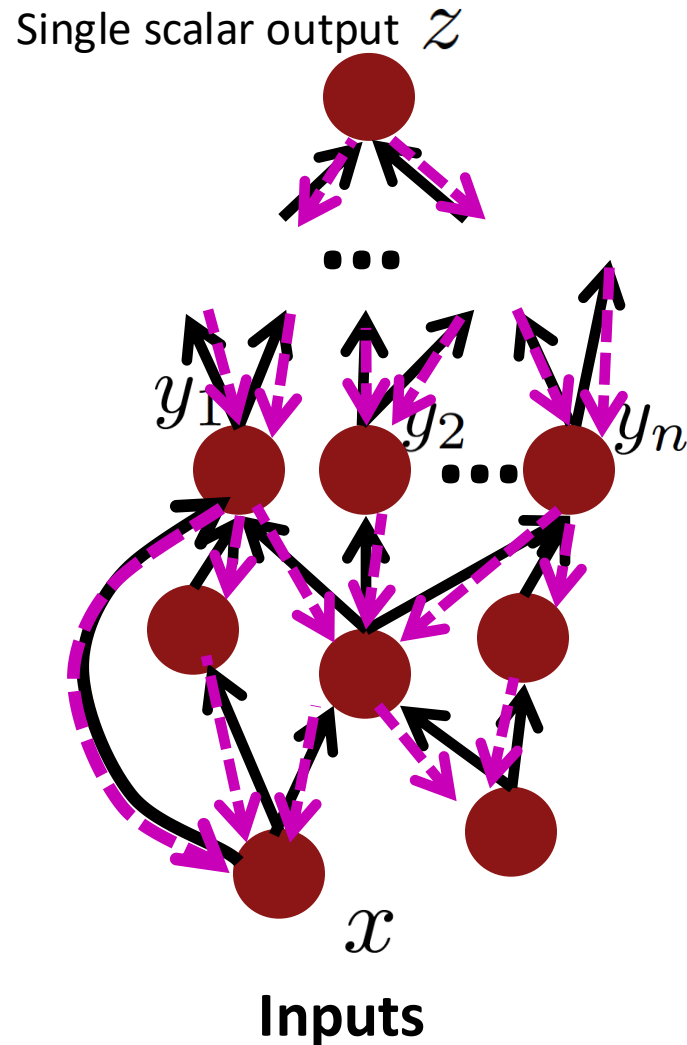
$$h = f(z)$$

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

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



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

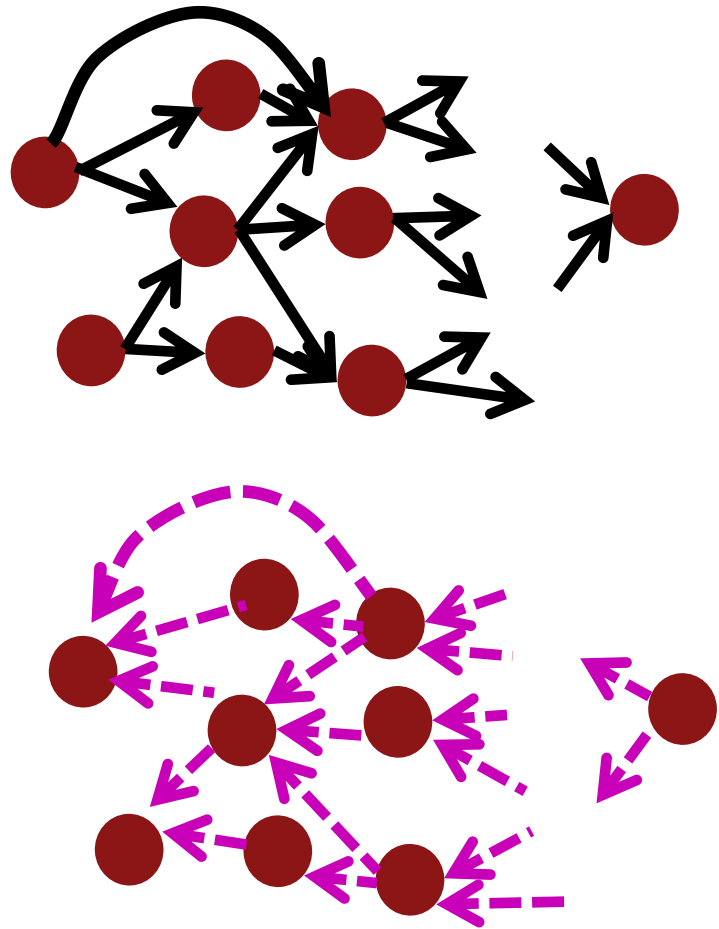
$\{y_1, y_2, \dots, y_n\} = \text{successors of } x$

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

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

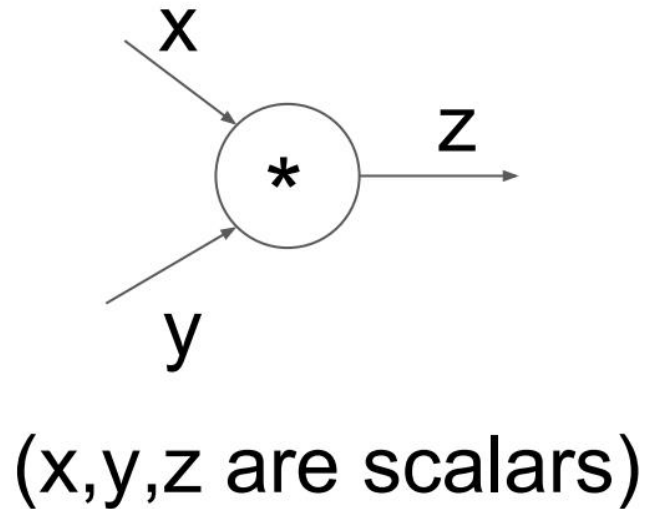


- The gradient computation can be automatically inferred from the symbolic expression of the fprop
- Each node type needs to know how to compute its output and how to compute the gradient wrt its inputs given the gradient wrt its output
- Modern DL frameworks (Tensorflow, PyTorch, etc.) do backpropagation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

Implementation: forward/backward API



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

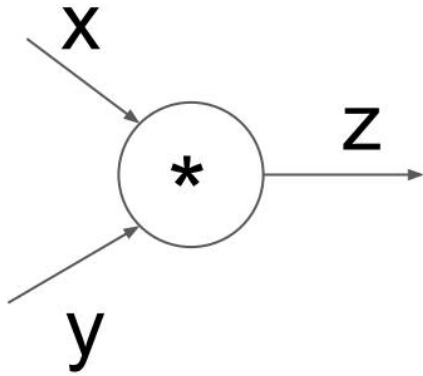
$$\frac{\partial L}{\partial z}$$

Arrow pointing to the `dz` parameter in the `backward` method.

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

Arrow pointing to the `dx` element in the `return` statement of the `backward` method.

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 ($\approx 1\text{e-}4$),
$$f'(x) \approx \frac{f(x + h) - f(x - h)}{2h}$$
- 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