Natural Language Processing with Deep Learning

CS224N/Ling284

Christopher Manning

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

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

(PERSON) (LOCATION)

- 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

WE NEED TO USE THE CONTEXT! PARIS CAN BE BOTH A PER AND A LOC.

- **Idea:** classify each word in its context window OF WORD VECTORS 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 .

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

PROB TO BE A LOCATION

PROB TO BE A PERSON.

- Resulting vector $x_{window} = \boxed{x \in R^{5d}}$
- 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

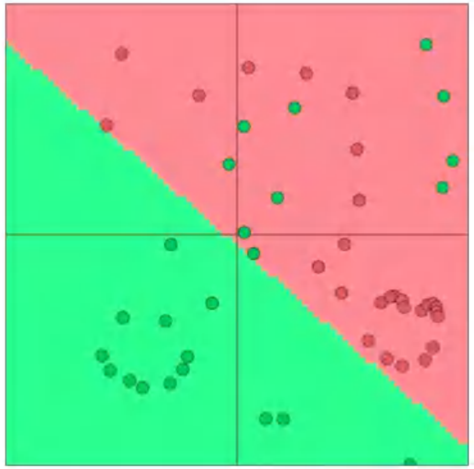
$$\{x_i, y_i\}^N_{i=1}$$

- x_i are inputs, e.g., words (indices or vectors!), sentences, documents, etc.
  - Dimension d

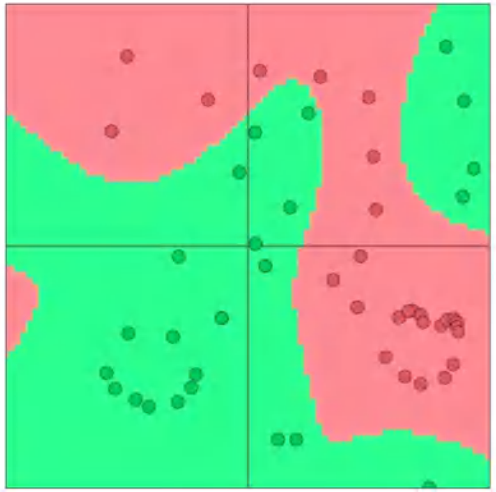
- y_i are labels (one of C classes) we try to predict, for example:
  - classes: sentiment (+/–), named entities, buy/sell decision
  - other words
  - later: multi-word sequences

Neural classification

- Typical ML/stats softmax classifier: $p(y|x) = \dfrac{\exp(W_y.x)}{\sum_{c=1}^{C} \exp(W_c.x)}$
- 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

Softmax classifier

$$p(y|x) = \frac{\exp(W_y.x)}{\sum_{c=1}^{C} \exp(W_c.x)}$$

Again, we can tease apart the prediction function into three steps:

1. For each row *y* of *W,* calculate dot product with *x*: $$W_y.x = \sum_{i=1}^{d} W_{yi} x_i = f_y$$
2. Apply softmax function to get normalized probability: $$p(y|x) = \frac{\exp(f_y)}{\sum_{c=1}^{C} \exp(f_c)} = \text{softmax}(f_y)$$
3. Choose the *y* with maximum probability

- For each training example (*x*,*y*), our objective is to maximize the probability of the correct class *y* or we can minimize the negative log probability of that class:

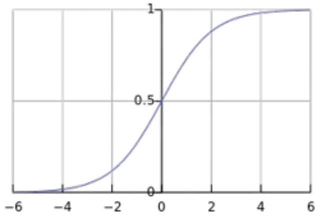
$$-\log p(y|x) = -\log\left(\frac{\exp(f_y)}{\sum_{c=1}^{C} \exp(f_c)}\right)$$

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



PROBABILITY OF A REAL NUMBER → PROB. TO BE OF A CERTAIN CLASS OR NOT

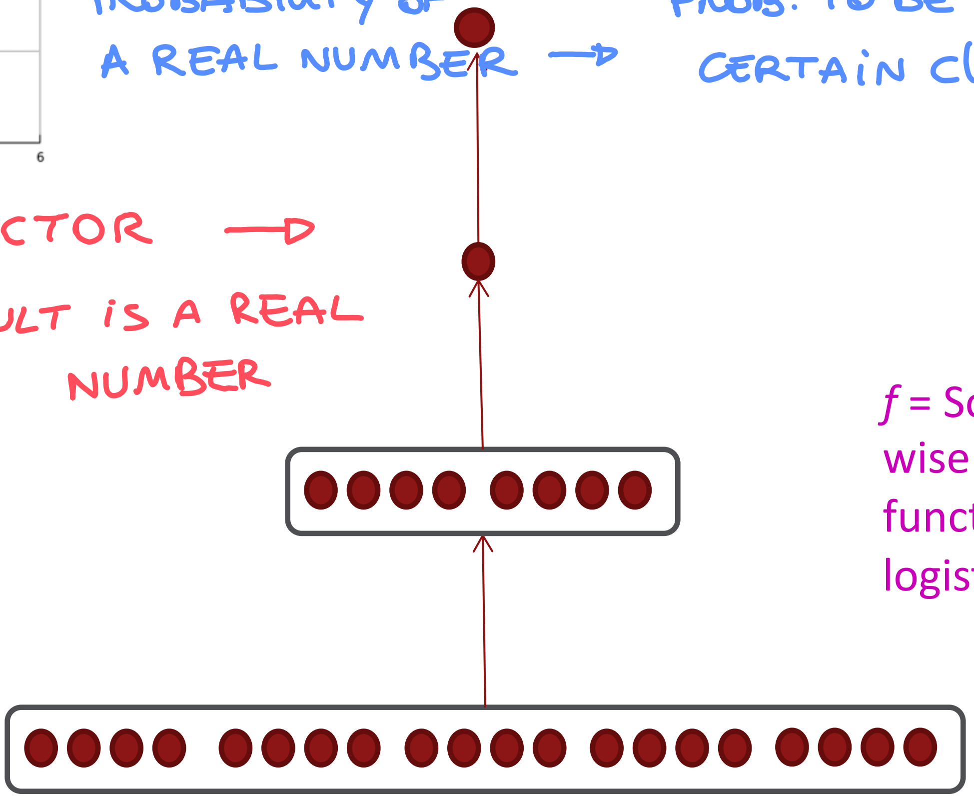
$$s = u^T h$$

HIDDEN VECTOR → RESULT IS A REAL NUMBER

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

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

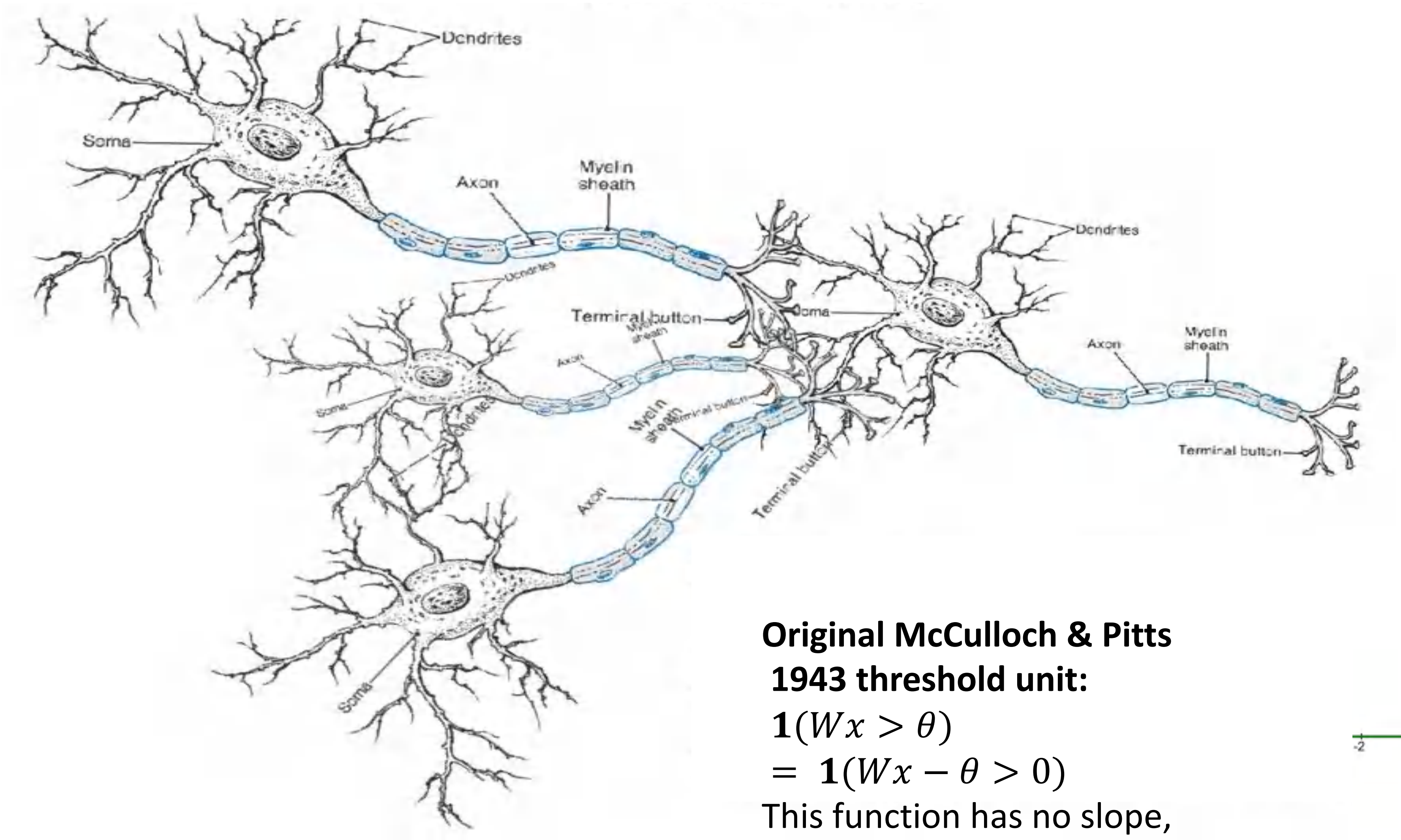
x (input) $\in \mathbb{R}^{5d}$



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

Embedding of 1-hot words

7. Neural computation

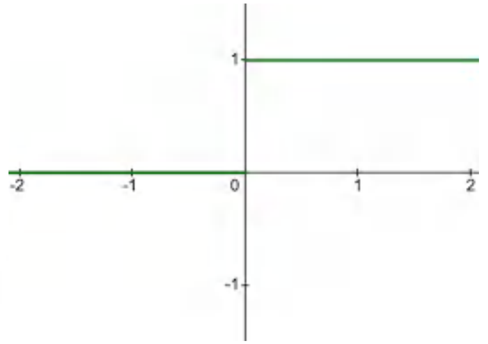


Original McCulloch & Pitts 1943 threshold unit:

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

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

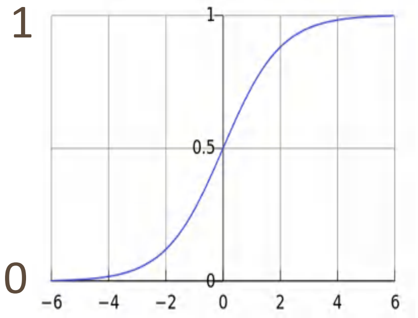
This function has no slope,
so, no gradient-based learning



Non-linearities, old and new

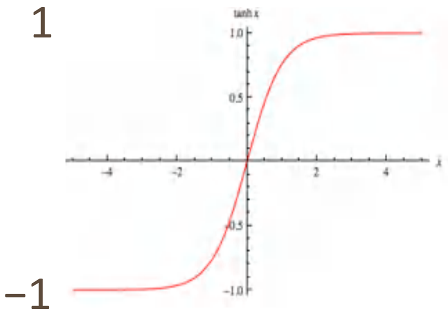
logistic ("sigmoid")

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



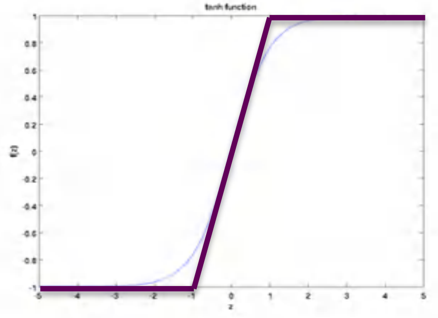
tanh

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



hard tanh

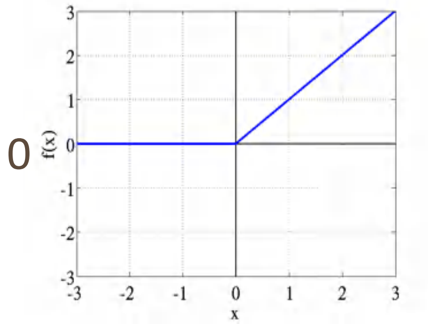
$$\mathrm{HardTanh}(x) = \begin{cases} -1 & \text{if } x < -1 \\ x & \text{if } -1 <= x <= 1 \\ 1 & \text{if } x > 1 \end{cases}$$



(Rectified Linear Unit)

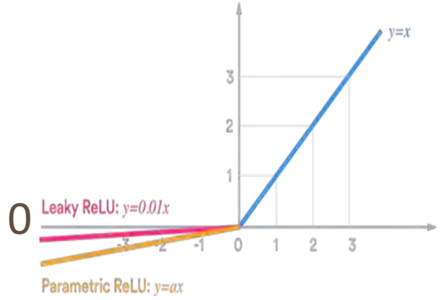
ReLU

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



Leaky ReLU /

Parametric ReLU



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

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

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

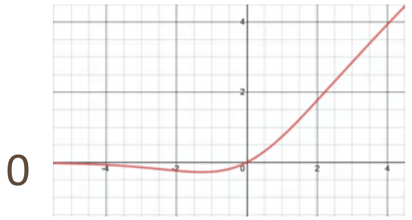
However, now, for deep networks, the first thing to try is ReLU: it trains quickly and performs well due to good gradient backflow.

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

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

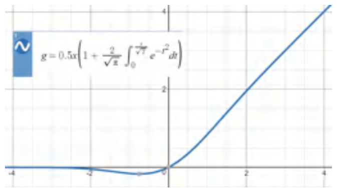
Swish arXiv:1710.05941

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



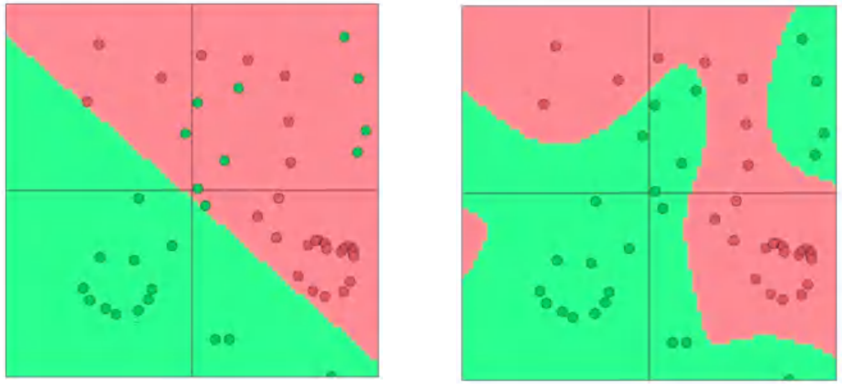
GELU arXiv:1606.08415

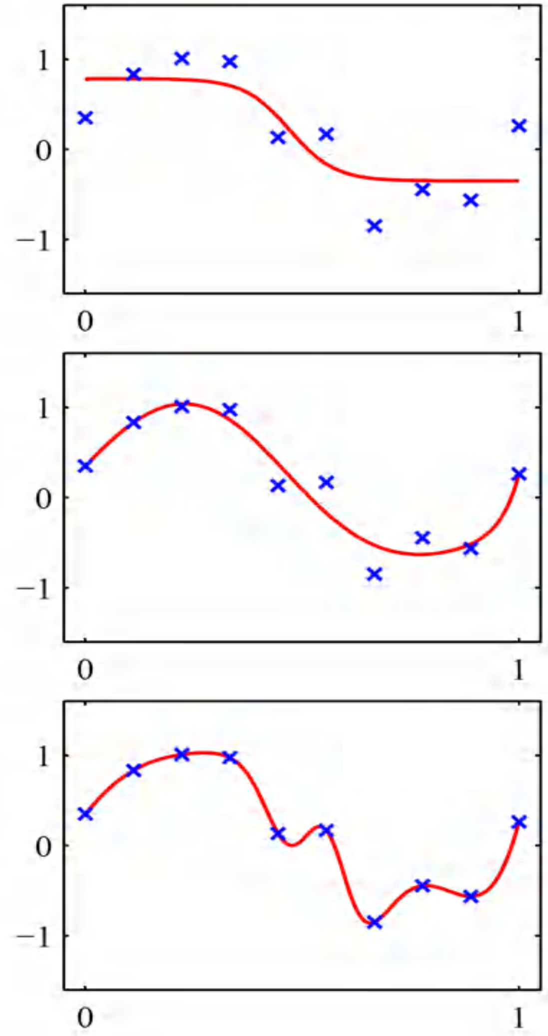
$$\begin{aligned} \mathrm{GELU}(x) &= x \cdot P(X \le x), X \sim N(0,1) \\ &\approx x \cdot \mathrm{logistic}(1.702x) \end{aligned}$$



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; l*et 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, …, 0, 1, 0, …, 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:

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

α = *step size* or *learning rate*

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

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

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

1. By hand
2. Algorithmically: the backpropagation algorithm

Lecture Plan

Lecture 4: Gradients by hand and algorithmically

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

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

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

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

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

Jacobian Matrix: Generalization of the Gradient

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

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

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

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

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

Chain Rule

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

$$z = 3y$$

$$y = x^2$$

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

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

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

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

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

Example Jacobian: Elementwise activation Function

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

$$h_i = f(z_i)$$

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

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

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

Other Jacobians

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

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

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

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

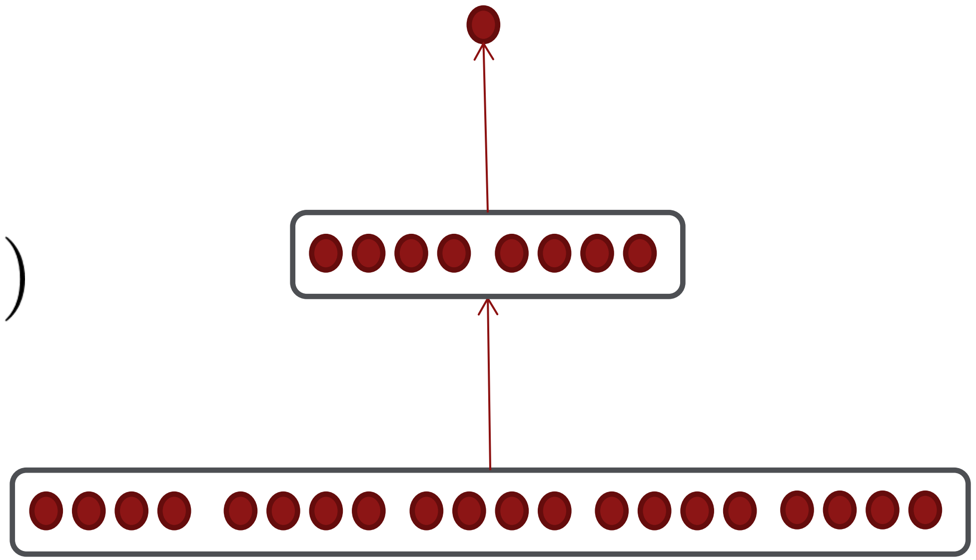
Back to our Neural Net!

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

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

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

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



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

1. Break up equations into simple pieces

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

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

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

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

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

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

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

Carefully define your variables and keep track of their dimensionality!

3. Write out the Jacobians

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

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

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

Useful Jacobians from previous slide

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

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

Re-using Computation

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

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

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

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

δ is the upstream gradient ("error signal")

Derivative with respect to Matrix: Output shape

- What does $\frac{\partial s}{\partial \boldsymbol{W}}$ look like? $\boldsymbol{W} \in \mathbb{R}^{n \times m}$
- 1 output, *nm* inputs: 1 by *nm* Jacobian?

VERY LONG ROW VECTOR

  - 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!

SAME SHAPE SO WE CAN DO THE SUBSTRACTION

  - So $\frac{\partial s}{\partial \boldsymbol{W}}$ is *n* by *m*: $\begin{bmatrix} \frac{\partial s}{\partial W_{11}} & \cdots & \frac{\partial s}{\partial W_{1m}} \\ \vdots & \ddots & \vdots \\ \frac{\partial s}{\partial W_{n1}} & \cdots & \frac{\partial s}{\partial W_{nm}} \end{bmatrix}$

Derivative with respect to Matrix

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

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

δ is upstream gradient ("error signal") at z
x is local input signal

Why the Transposes?

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

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

This is what we want to produce ultimatly

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

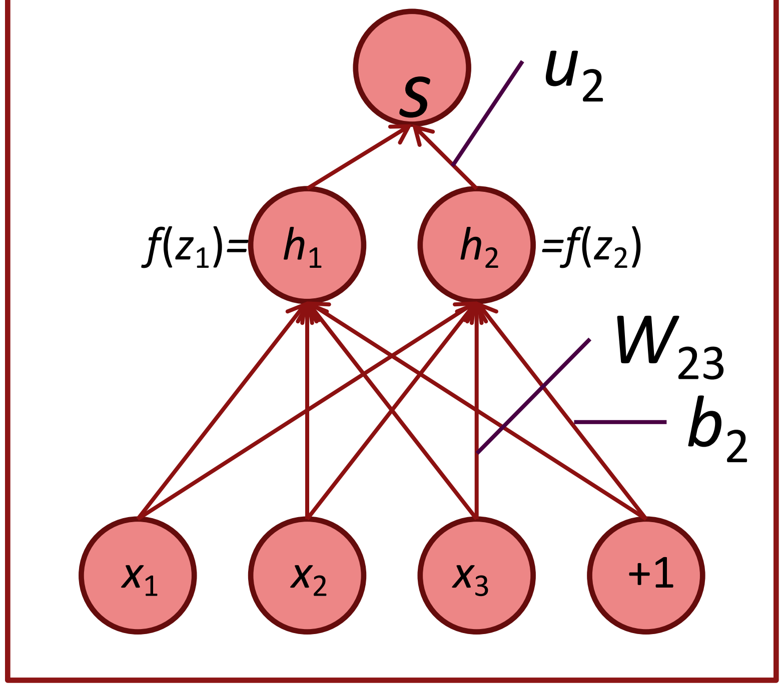
- Hacky answer: this makes the dimensions work out!
  - Useful trick for checking your work!
- Full explanation in the lecture notes
  - Each input goes to each output – you want to get outer product

Deriving local input gradient in backprop

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

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

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



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

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

What shape should derivatives be?

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

What shape should derivatives be?

Two options for working through specific problems:

1. Use Jacobian form as much as possible, reshape to follow the shape convention at the end:
   - What we just did. But at the end transpose $\frac{\partial s}{\partial \boldsymbol{b}}$ to make the derivative a column vector, resulting in $\boldsymbol{\delta}^T$

2. Always follow the shape convention
   - Look at dimensions to figure out when to transpose and/or reorder terms
   - The error message $\boldsymbol{\delta}$ that arrives at a hidden layer has the same dimensionality as that hidden layer

3. Backpropagation

We’ve almost shown you backpropagation

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

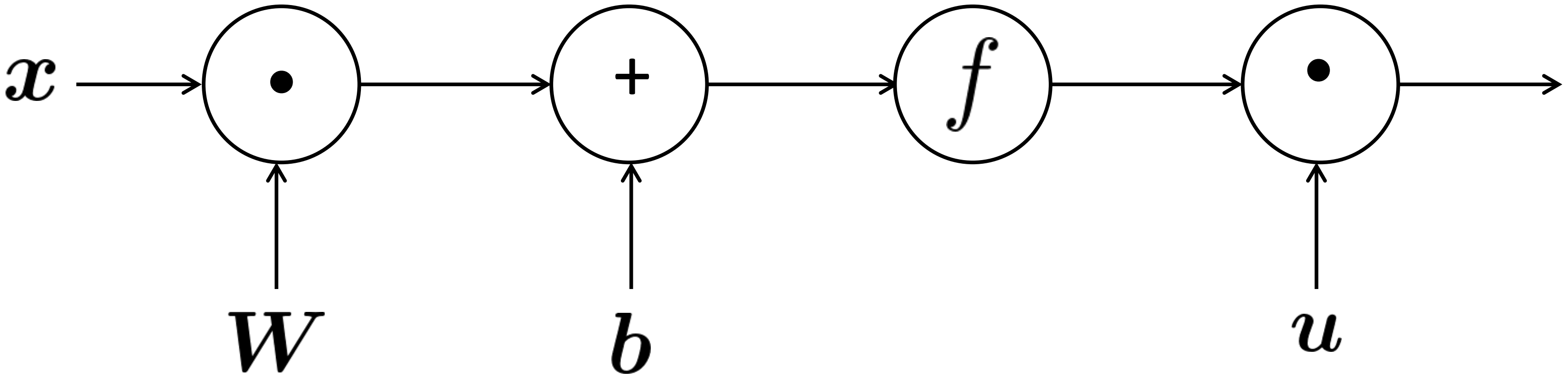
Other trick:

- 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

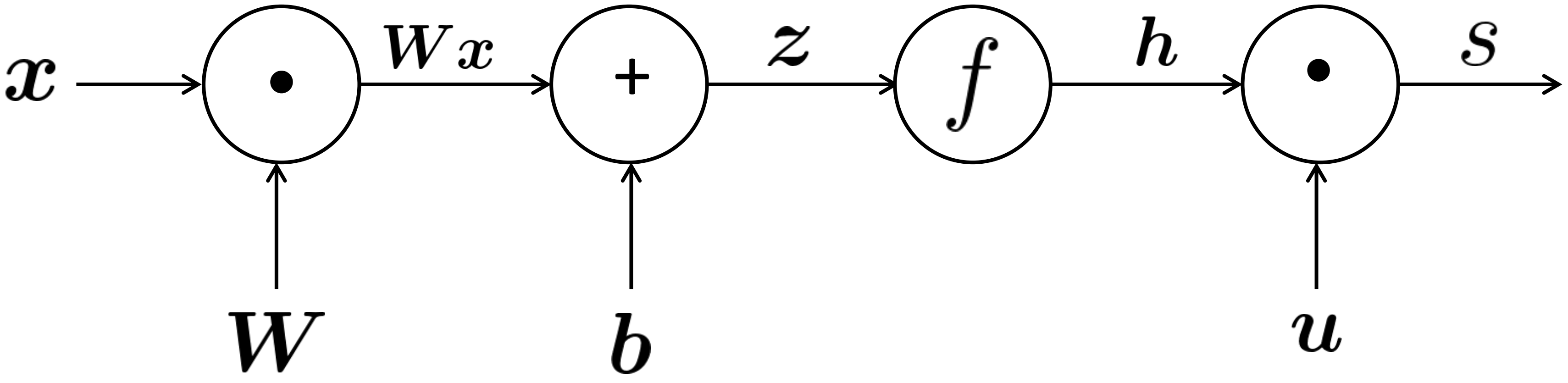
$$s = \boldsymbol{u}^T \boldsymbol{h}$$
$$\boldsymbol{h} = f(\boldsymbol{z})$$
$$\boldsymbol{z} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}$$
$$\boldsymbol{x} \quad \text{(input)}$$



Computation Graphs and Backpropagation

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

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



Computation Graphs and Backpropagation

- Software represents our neural net equations as a graph

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

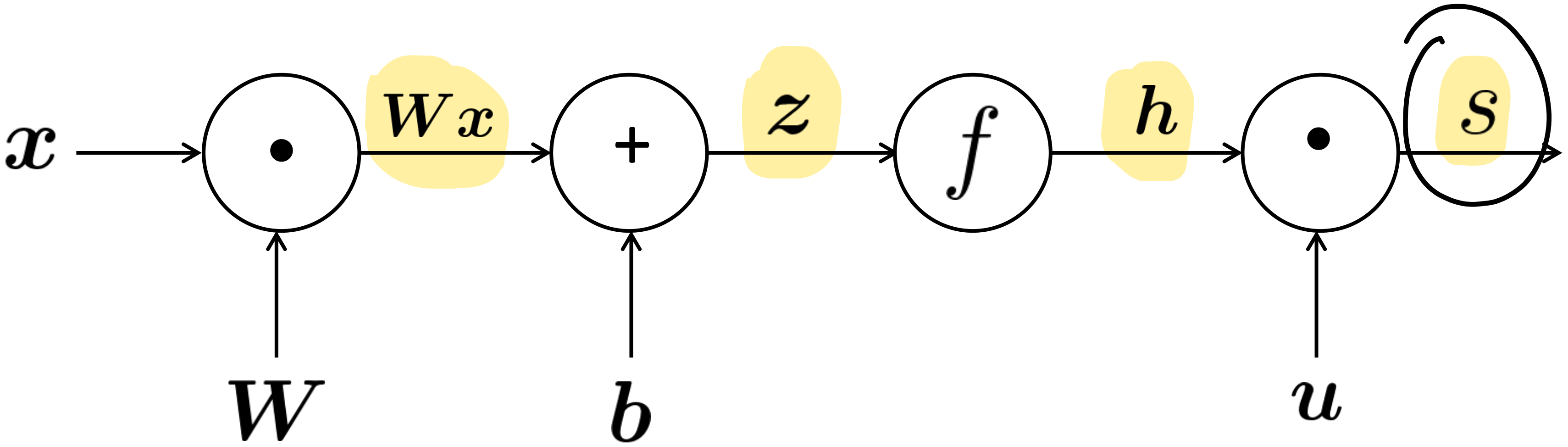
$$\boldsymbol{h} = f(\boldsymbol{z})$$

$$\ldots c + \boldsymbol{b}$$

$$\ldots \text{ut})$$

“Forward Propagation”

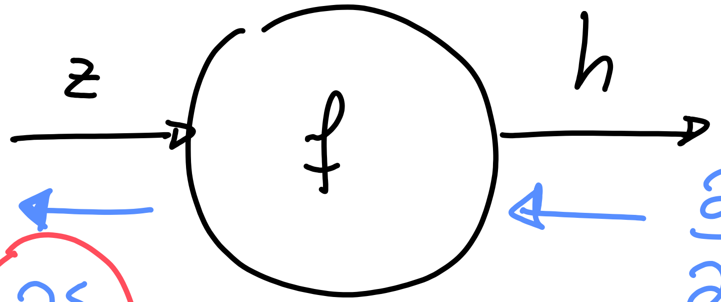
operation



computing the value of s first

WHAT WE WANT TO DO?

$h = f(z)$



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

UPSTREAM GRADIENT

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

WHAT WE WANT TO COMPUTE

DOWNSTREAM GRADIENT

- we compute the local gradient for each node.
- we use the chain rule to calculate the downstream gradient.

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

UPSTREAM GRADIENT

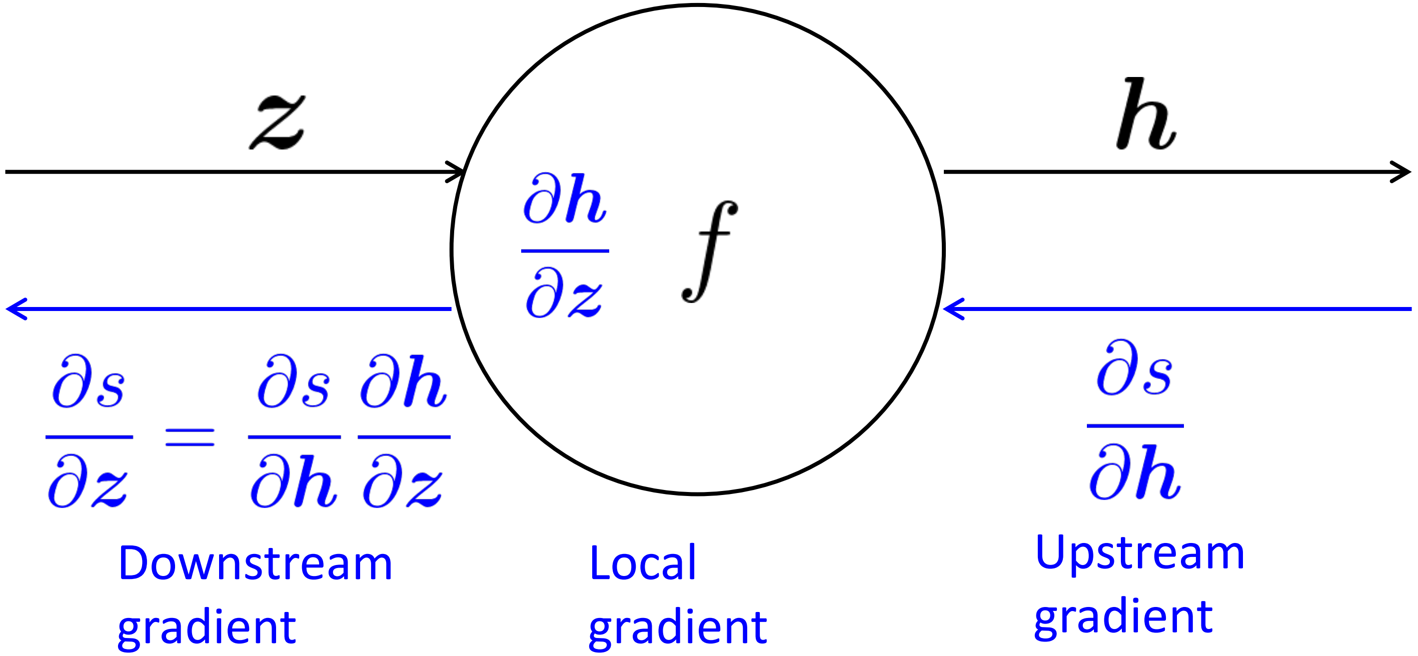
LOCAL GRADIENT

Backpropagation: Single Node

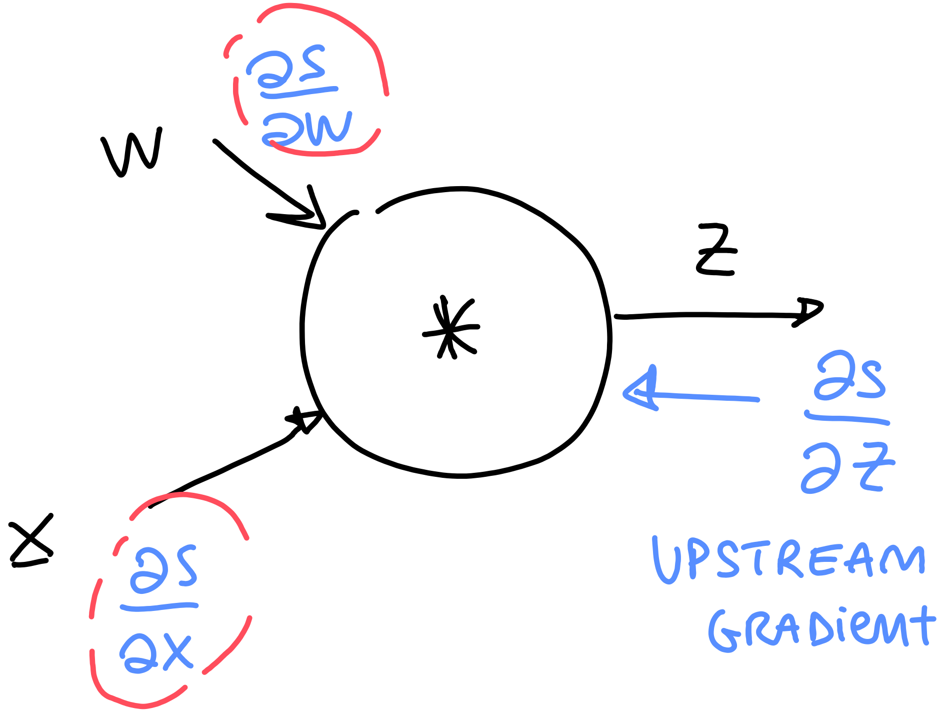
- Each node has a **local gradient**
  - The gradient of its output with respect to its input

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

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



FOR MULTIPLE INPUTS?



WHAT WE WANT TO COMPUTE

DOWNSTREAM GRADIENTS

$$z = W\underline{x}$$

I can compute the local gradients and then apply the chain rule.

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

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

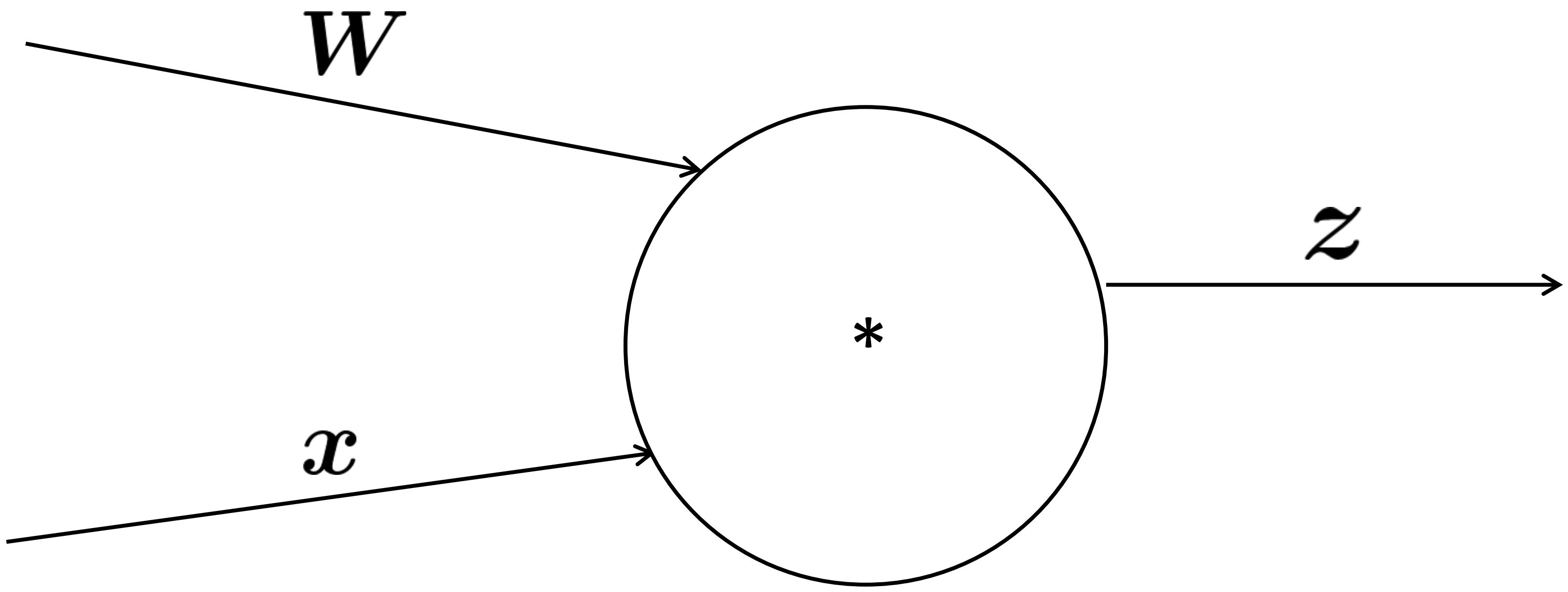
UPSTREAM GRADIENT

LOCAL GRADIENT

Backpropagation: Single Node

- What about nodes with multiple inputs?

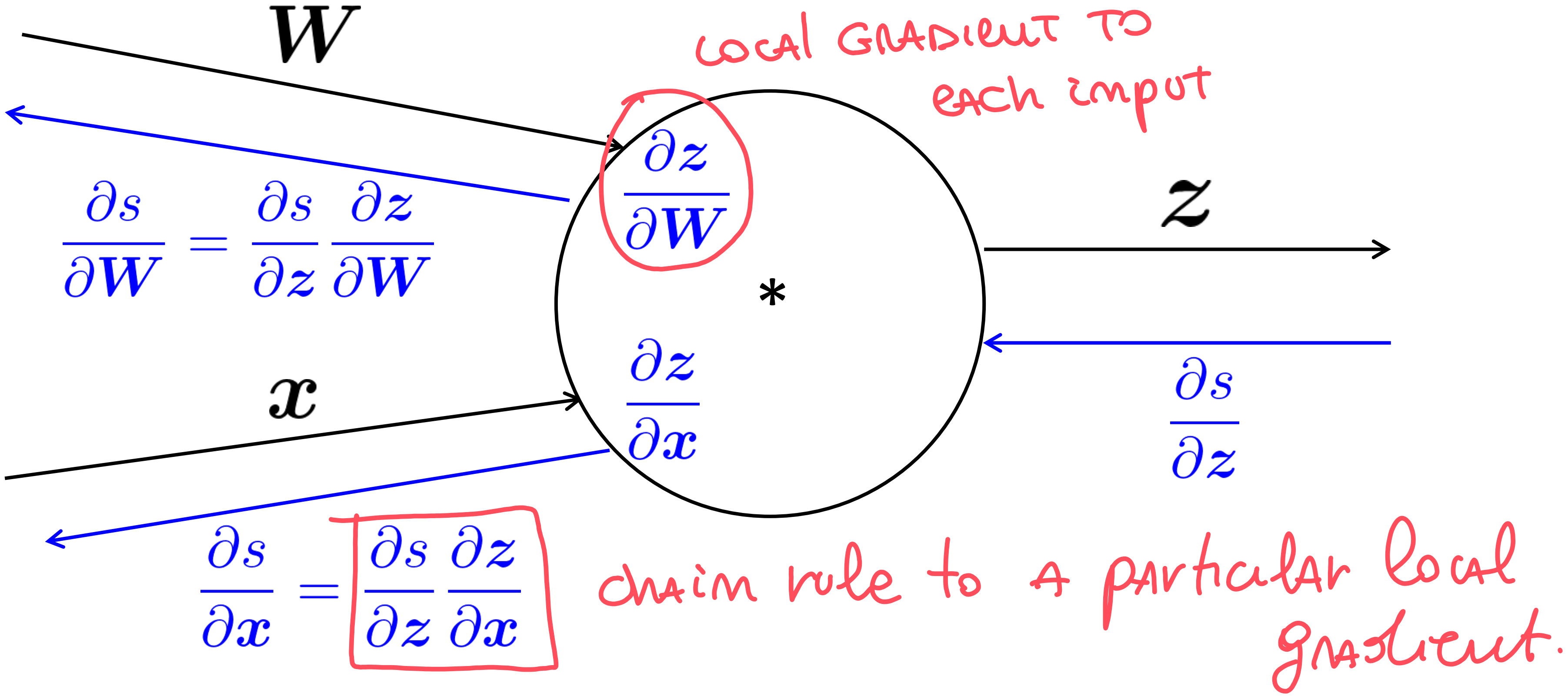
$$\boxed{z = \boldsymbol{W}\boldsymbol{x}}$$



Backpropagation: Single Node

- Multiple inputs → multiple local gradients

$$\boxed{z = Wx}$$



Downstream gradients

Local gradients

Upstream gradient

An Example

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

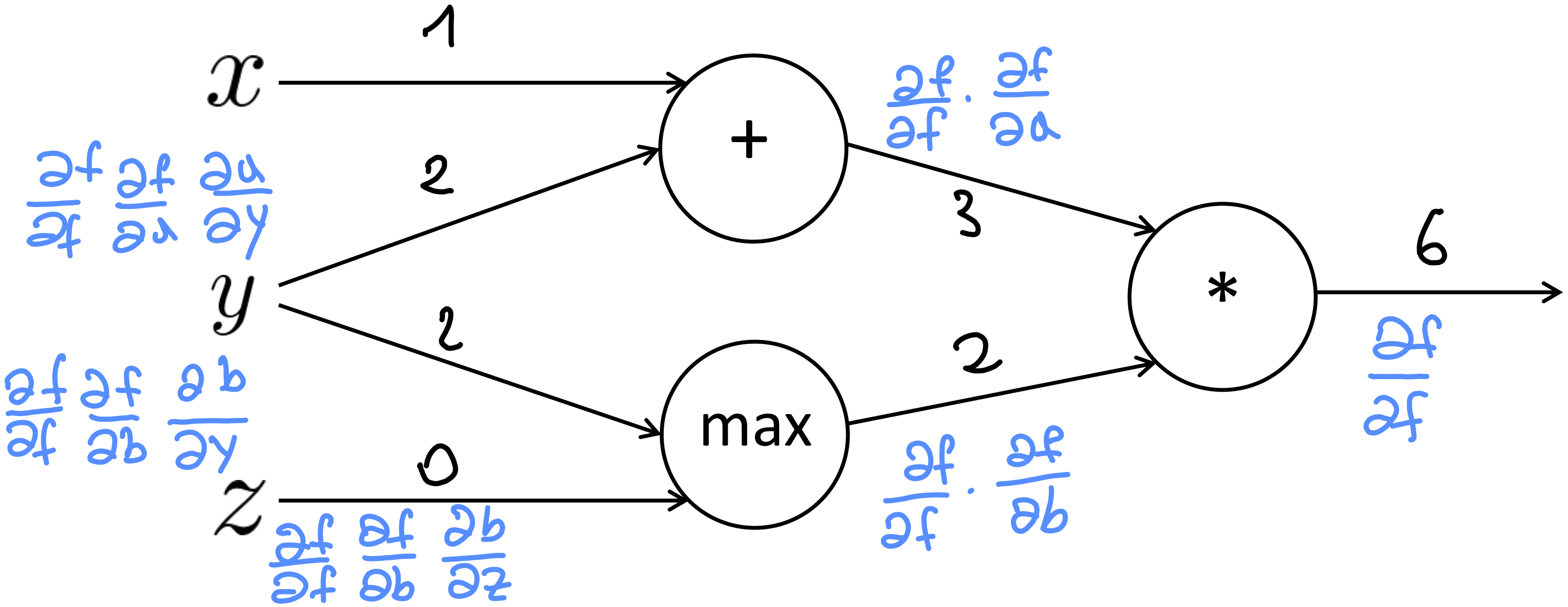
Forward prop steps

$$a = x + y$$

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

$$f = ab$$

$\frac{\partial f}{\partial f} \frac{\partial f}{\partial a} \frac{\partial a}{\partial x}$



An Example

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

Forward prop steps

$$a = x + y$$

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

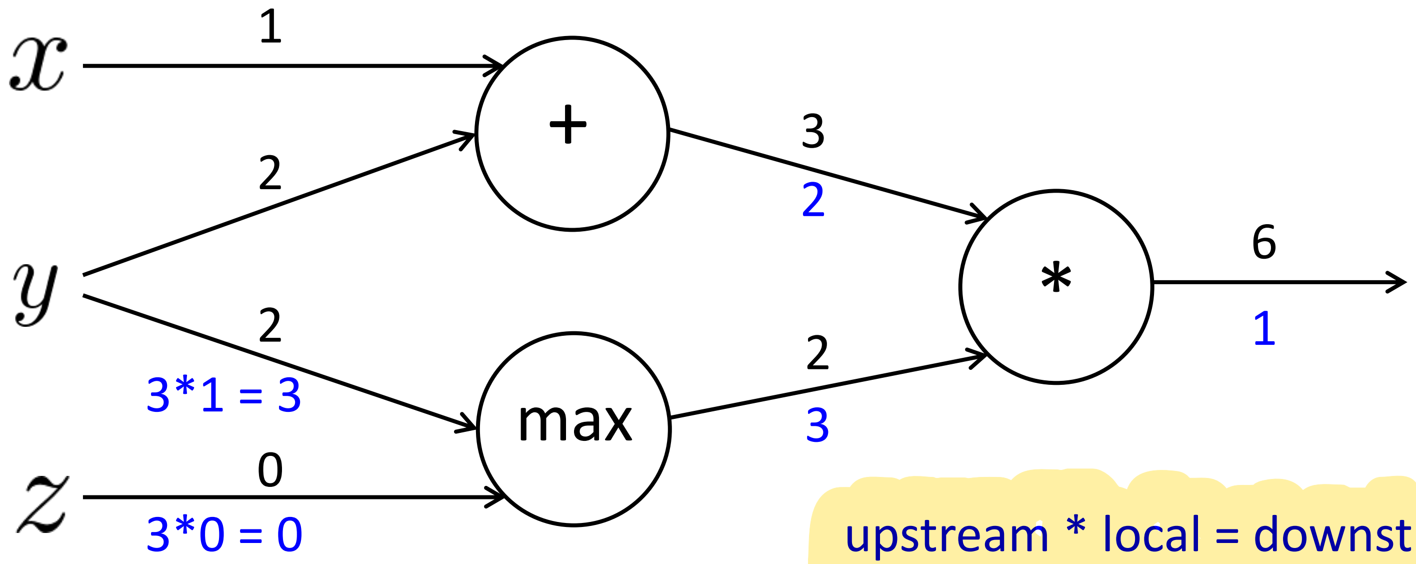
$$f = ab$$

Local gradients

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

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

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



upstream * local = downstream

An Example

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

Forward prop steps

$$a = x + y$$

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

$$f = ab$$

IF I CHANGE x in the input, I'll see the output changed by the double

Local gradients

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

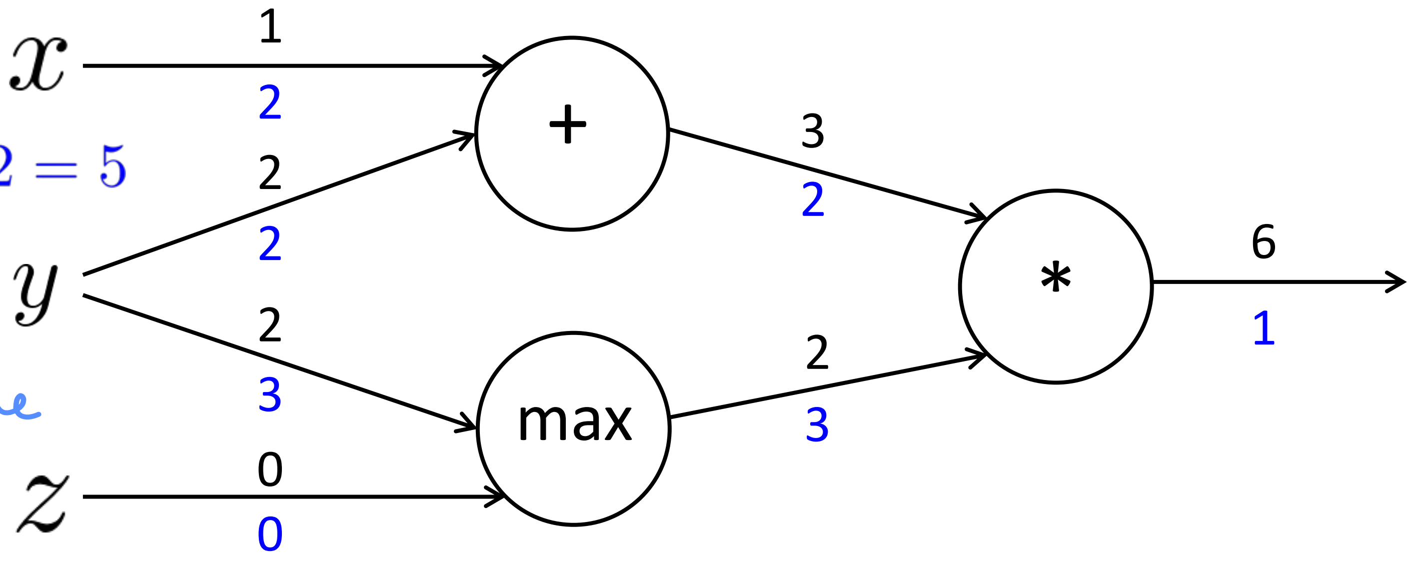
$$\frac{\partial b}{\partial y} = \mathbf{1}(y > z) = 1 \quad \frac{\partial b}{\partial z} = \mathbf{1}(z > y) = 0$$

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

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

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

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



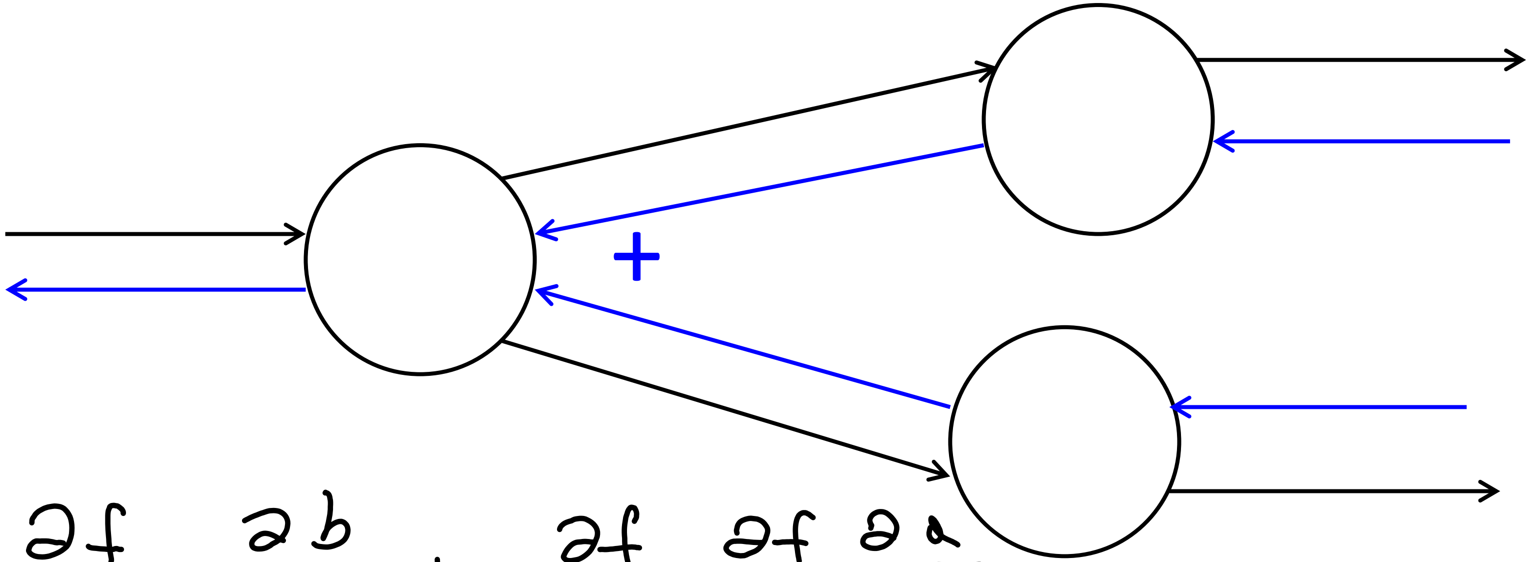
If I change z in the input, I'll see NO changes in the output.

Gradients sum at outward branches

$a = x + y$

$b = \max(y, z)$

$f = a \cdot b$



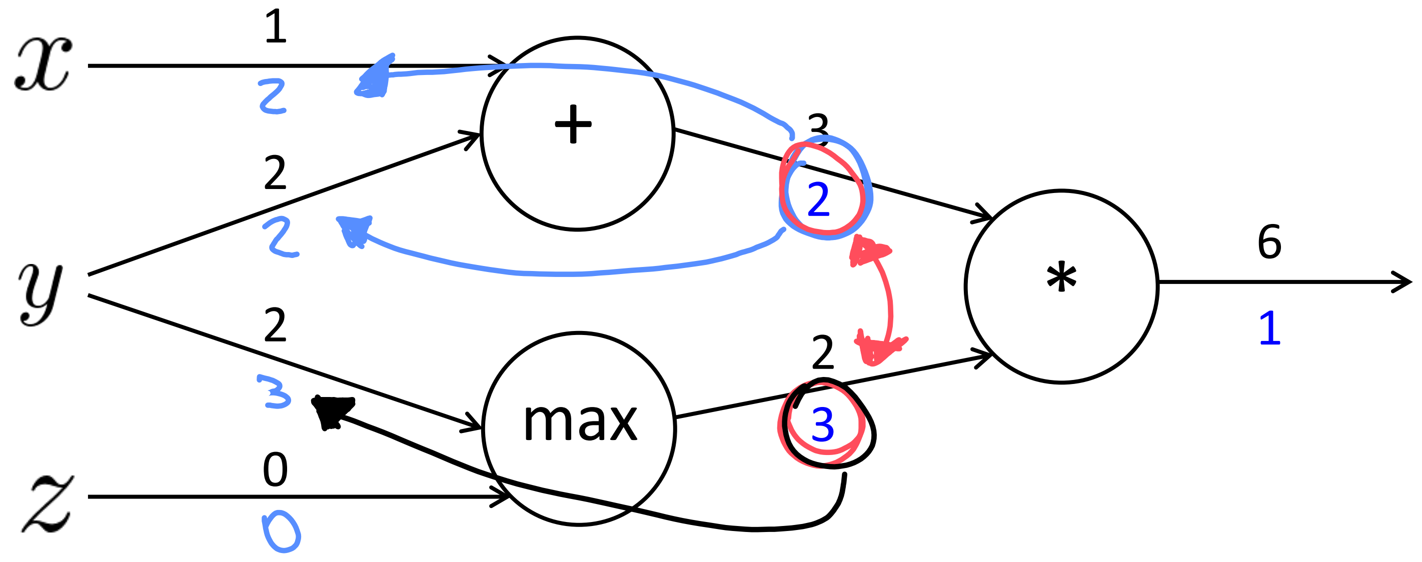
$$\frac{\partial f}{\partial y} = \frac{\partial f}{\partial f} \frac{\partial f}{\partial b} \cdot \frac{\partial b}{\partial y} + \frac{\partial f}{\partial f} \cdot \frac{\partial f}{\partial a} \frac{\partial a}{\partial y}$$

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

Node Intuitions

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

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

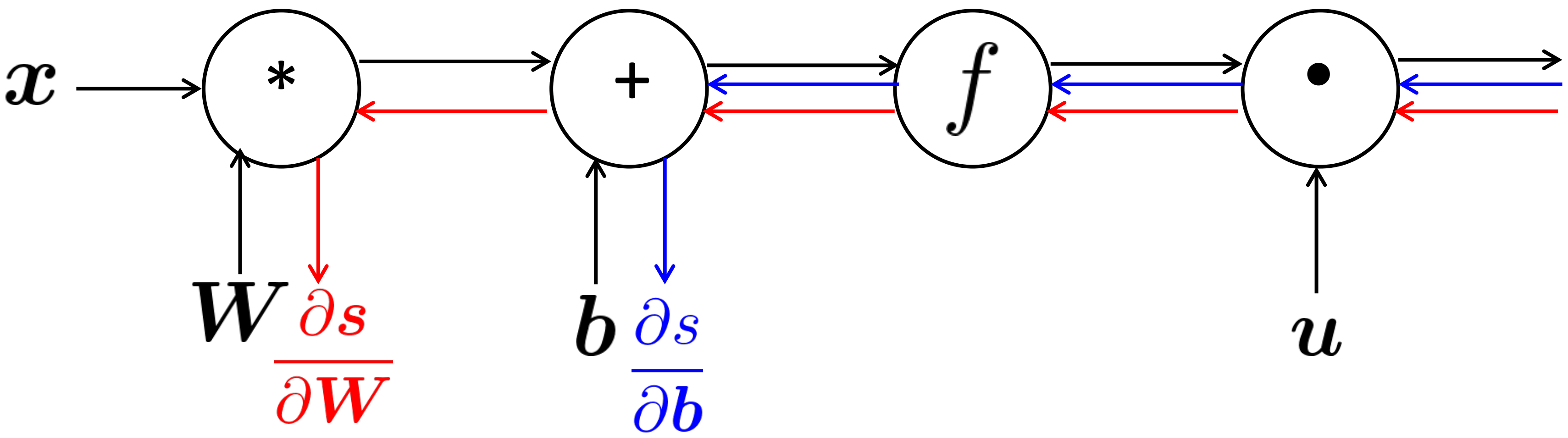


Efficiency: compute all gradients at once

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

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

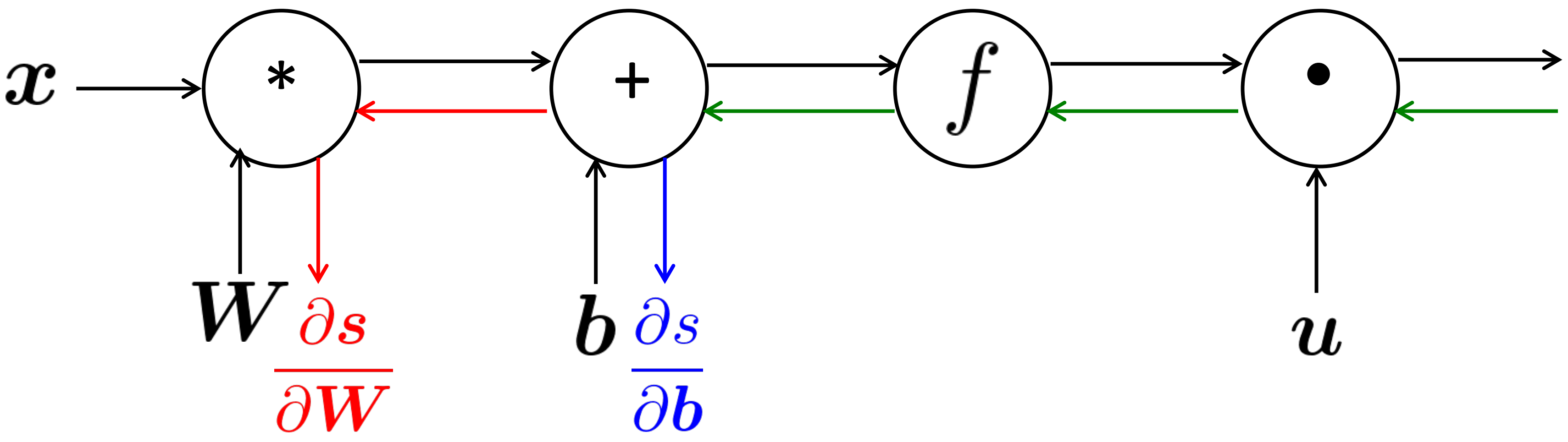
Avoid duplicate computations!!



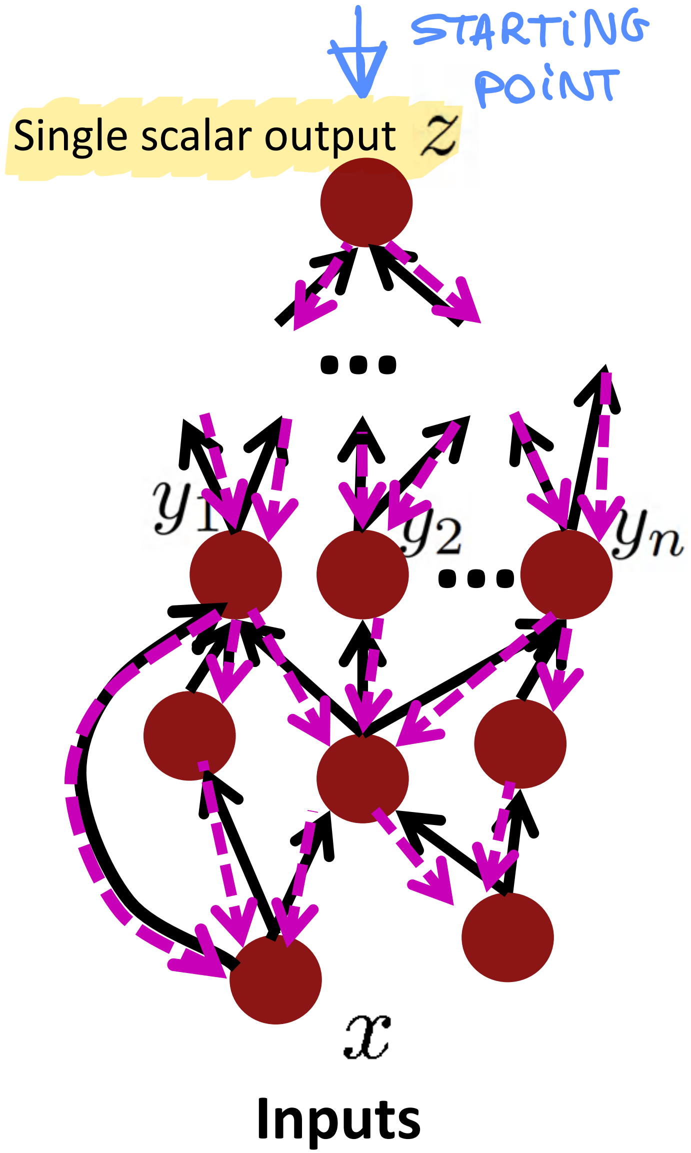
Efficiency: compute all gradients at once

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

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



Back-Prop in General Computation Graph



1. Fprop: visit nodes in topological sort order
   - Compute value of node given predecessors
2. Bprop:
   - initialize output gradient = 1
   - visit nodes in reverse order:
   Compute gradient wrt each node using gradient wrt successors

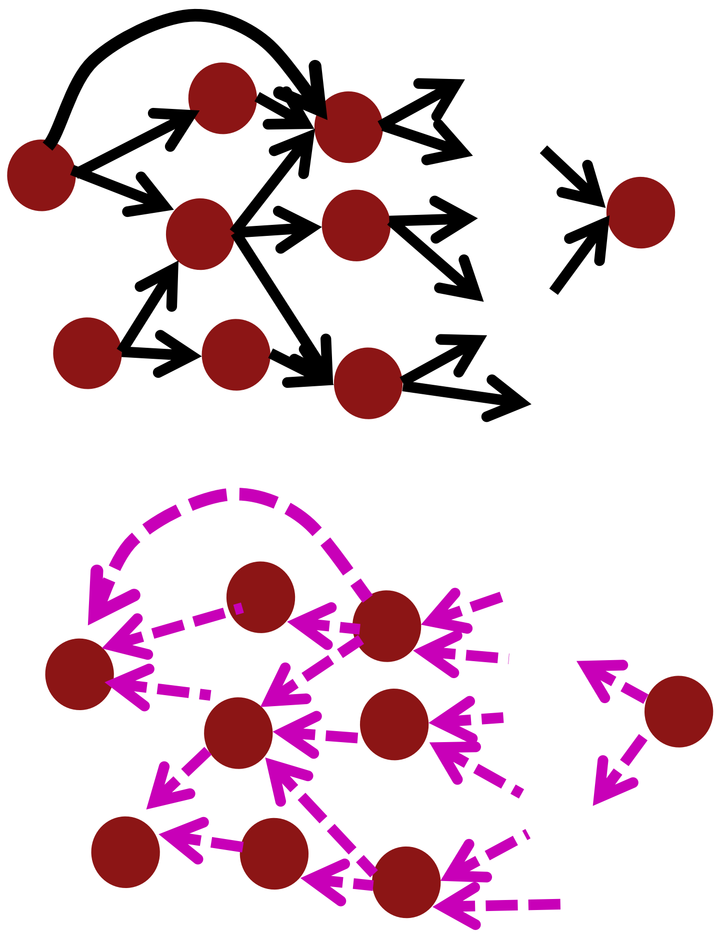
$\{y_1, y_2, \ldots y_n\}$ = successors of x

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

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

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

Automatic Differentiation

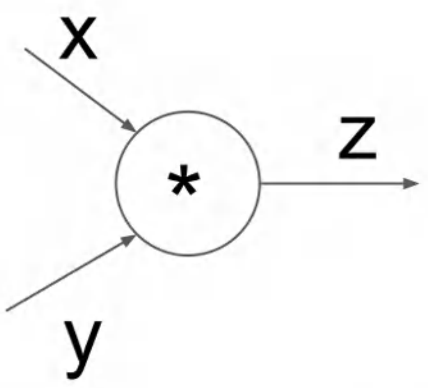


- The gradient computation can be automatically inferred from the symbolic expression of the fprop
- Each node type needs to know how to compute its output and how to compute the gradient wrt its inputs given the gradient wrt its output
- Modern DL frameworks (Tensorflow, PyTorch, etc.) do backpropagation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

Backprop Implementations

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

Implementation: forward/backward API



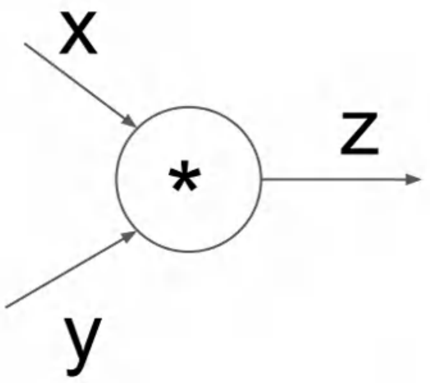
(x,y,z are scalars)

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

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

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

Implementation: forward/backward API



(x,y,z are scalars)

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

Manual Gradient checking: Numeric Gradient

- For small *h* (≈ 1e-4), $$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h}$$
- Easy to implement correctly
- But approximate and **very** slow:
  - You have to recompute *f* for **every parameter** of our model
- Useful for checking your implementation
  - In the old days, we hand-wrote everything, doing this everywhere was the key test
  - Now much less needed; you can use it to check layers are correctly implemented

Summary

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

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

Why learn all these details about gradients?

- **Modern deep learning frameworks compute gradients for you!**
  - Come to the PyTorch introduction this Friday!

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

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