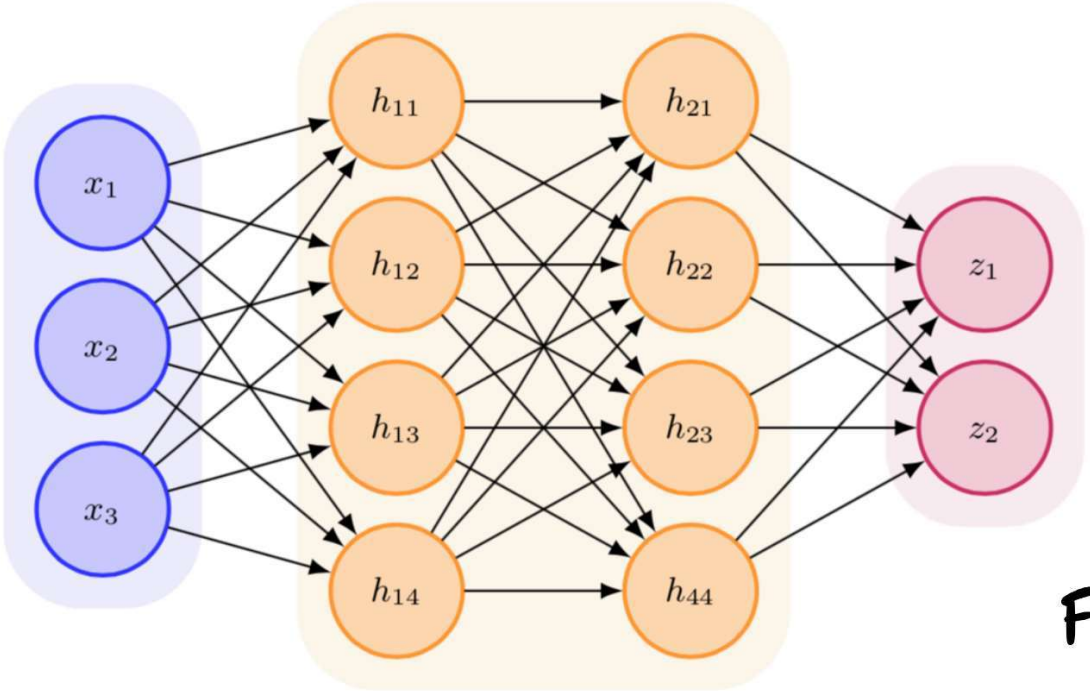
Lecture 6: Neural Networks + Backpropagation

Announcements:

- HW #2 is due tonight, uploaded to Gradescope. Please budget time for submission. Please be sure to print out Jupyter Notebooks and .py files for your submission.

- We will upload HW #3 tonight. Given the pace of the course, we decided to make it due on Friday, Feb 7, 2025 (instead of Monday, Feb 3, 2025).

Neural networks



Fully connected (FC) Neural Nets

This network has 10 neurons (not counting the input). It has (3x4) + (4x4) + (4x2) = 36 weights, and 4+4+2= 10 biases for a total of 46 learnable parameters.

Multilayer Perceptron (MLP)

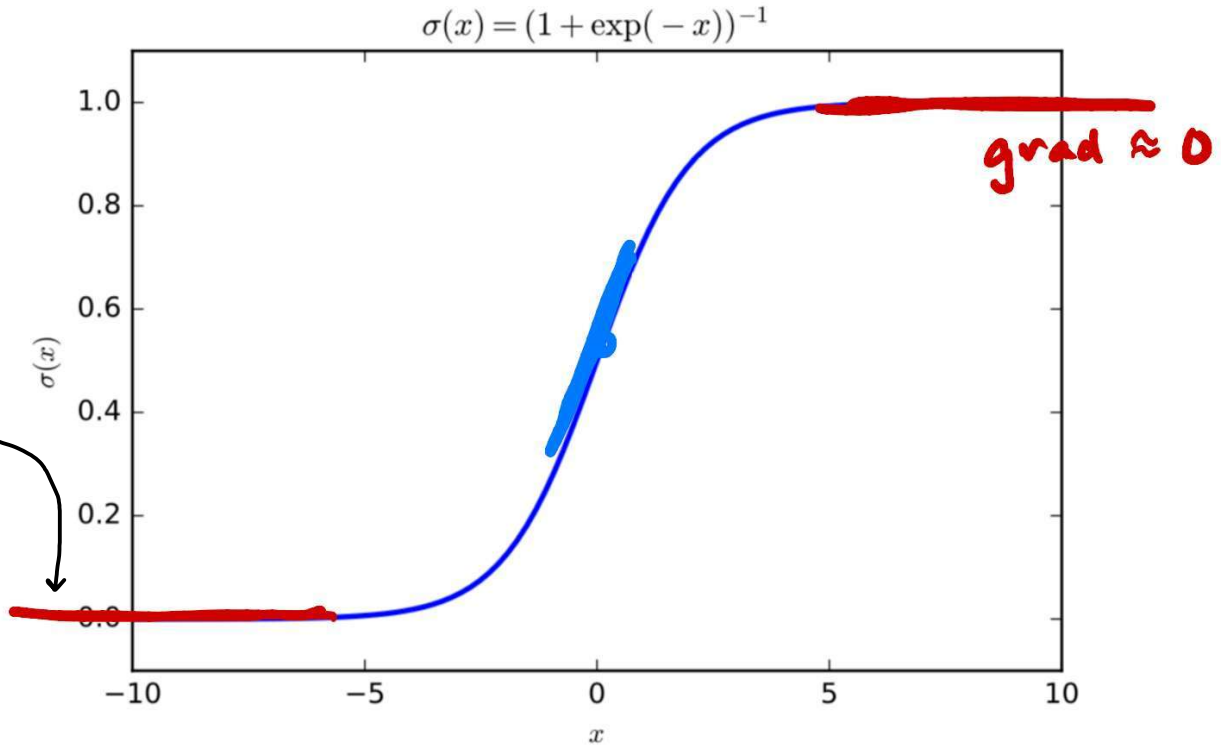
Sigmoid unit

$f(Wx+b)$

$= \sigma(Wx+b)$

Sigmoid activation, $\sigma(x) = \frac{1}{1+\exp(-x)}$

```
f = lambda x: 1.0 / (1.0 + np.exp(-x))
```



grad ≈ 0

Want to avoid functions that have regions of saturation like this

Its derivative is:

$$\frac{d\sigma(x)}{dx} = \sigma(x)(1-\sigma(x))$$

Sigmoid unit

"One recurring theme throughout neural network design is that the gradient of the cost function must be large and predictable enough to serve as a good guide for the learning algorithm." (Goodfellow et al., p. 173)

$\mathcal{L}(w)$

$\sigma(w) = \sigma(w^T x + b)$

$w \leftarrow w - \varepsilon \frac{\partial \mathcal{L}}{\partial w}$

$\frac{\partial \mathcal{L}}{\partial w} = \frac{\partial \sigma(w)}{\partial w} \frac{\partial \mathcal{L}}{\partial \sigma(w)}$

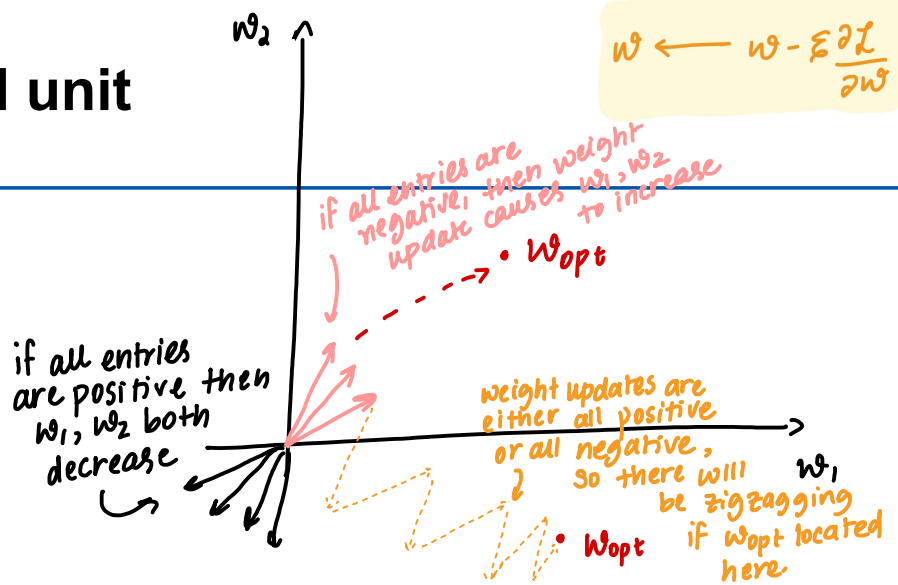
a very small #

Note: Both w_1 and w_2 will change in the same direction because we can guarantee that all entries in $\frac{\partial \mathcal{L}}{\partial w}$ are either positive or negative.

Sigmoid unit

$w \leftarrow w - \varepsilon \frac{\partial \mathcal{L}}{\partial w}$

Sigmoid activation, $\sigma(x) = \frac{1}{1+\exp(-x)}$



Pros:

- Around $x = 0$, the unit behaves linearly.
- It is differentiable everywhere.

Cons:

- At extremes, the unit *saturates* and thus has zero gradient. This results in no learning with gradient descent.
- The sigmoid unit is non-negative. SGD with the sigmoid unit zig-zags.

Note, this is not really a concern for DL, but this example is instructive for optimization.

$x \xrightarrow{(\hat{w}, \vec{b}), \sigma} h_1 \xrightarrow{(w, b), \sigma} h_2$

change of variables.

Consider $f(\mathbf{w}) = \sigma(\mathbf{w}^T \mathbf{h}_1 + b)$. Defining $z = \mathbf{w}^T \mathbf{h}_1 + b$, the derivative with respect to $\mathbf{w}$, the parameters, is:

$$\frac{\partial f(\mathbf{w})}{\partial \mathbf{w}} = \sigma(z)(1 - \sigma(z))\mathbf{h}_1 = \frac{\partial f(w)}{\partial z} \cdot \frac{\partial z}{\partial w}$$

($\sigma(z) \geq 0$, $(1-\sigma(z)) \geq 0$, $\mathbf{h}_1 \geq 0$)

since it is formed by multiplying 3 positive terms, $\frac{\partial f}{\partial w} \geq 0$

If $\mathbf{x} \geq 0$ (e.g., if the input units all had a sigmoidal output), then the gradient has all positive entries. Let's say we had some gradient, $\frac{\partial \mathcal{L}}{\partial f}$, which can be positive or negative. Then $\frac{\partial \mathcal{L}}{\partial \mathbf{w}}$ will have all positive or negative entries.

This can result in zig-zagging during gradient descent.

Note:
$0 \leq \sigma(z) \leq 1$
$0 \leq (1 - \sigma(z)) \leq 1$
$h_1 = \sigma(\hat{w}x + \vec{b})$
$\Rightarrow h_1 \geq 0$

$f(w) = \sigma(z)$
$\frac{\partial f}{\partial z} = \sigma(z)(1-\sigma(z))$
→ Result from prev. slide.

$z = w^T h_1 + b$
$\frac{\partial z}{\partial w} = h_1$

$\frac{\partial f}{\partial z}$ is a scalar, so it is either ≥ 0 or < 0

scalar

$\frac{\partial \mathcal{L}}{\partial w} = \frac{\partial f(w)}{\partial w} \cdot \frac{\partial \mathcal{L}}{\partial f(w)}$

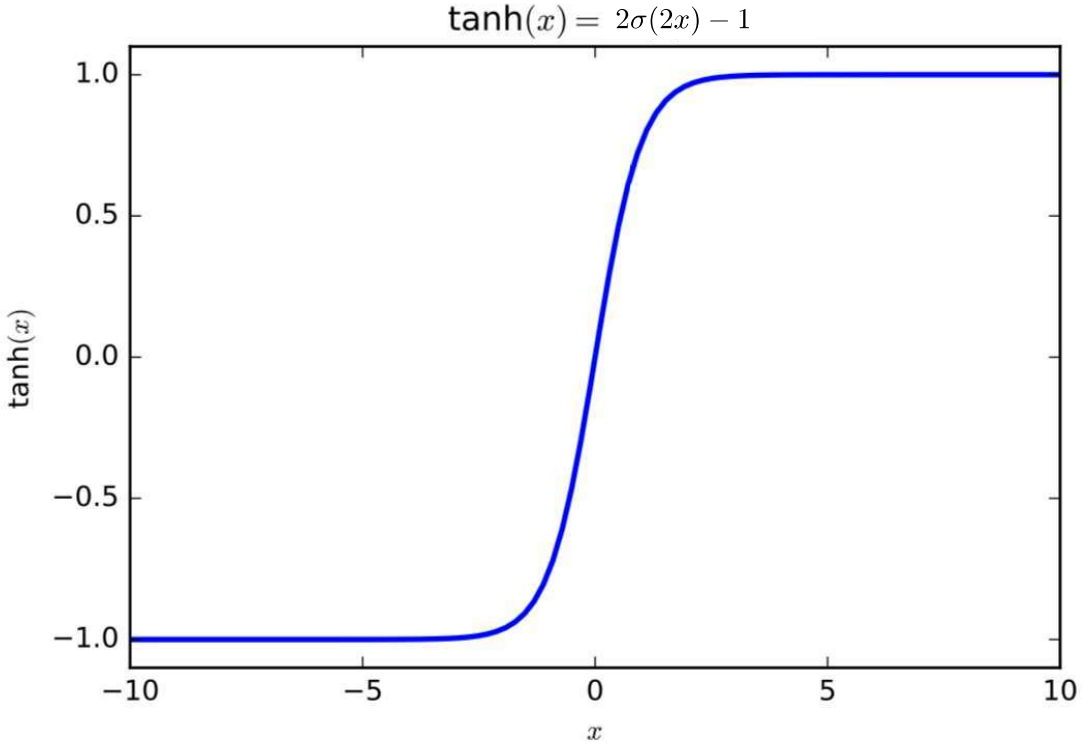
all entries in this grad. are either ≥ 0 or < 0

Hyperbolic tangent

Hyperbolic tangent, $\tanh(x) = 2\sigma(2x) - 1$

The hyperbolic tangent is a zero-centered sigmoid-looking activation.

```
f = lambda x: np.tanh(x)
```



Its derivative is:

$$\frac{d\tanh(x)}{dx} = 1 - \tanh^2(x)$$

can check this as an exercise.

Hyperbolic tangent

Hyperbolic tangent, $\tanh(x) = 2\sigma(2x) - 1$

Pros:

- Around $x = 0$, the unit behaves linearly.
- It is differentiable everywhere.
- It is zero-centered. (can be both positive and negative, so not constrained to zig zagging)

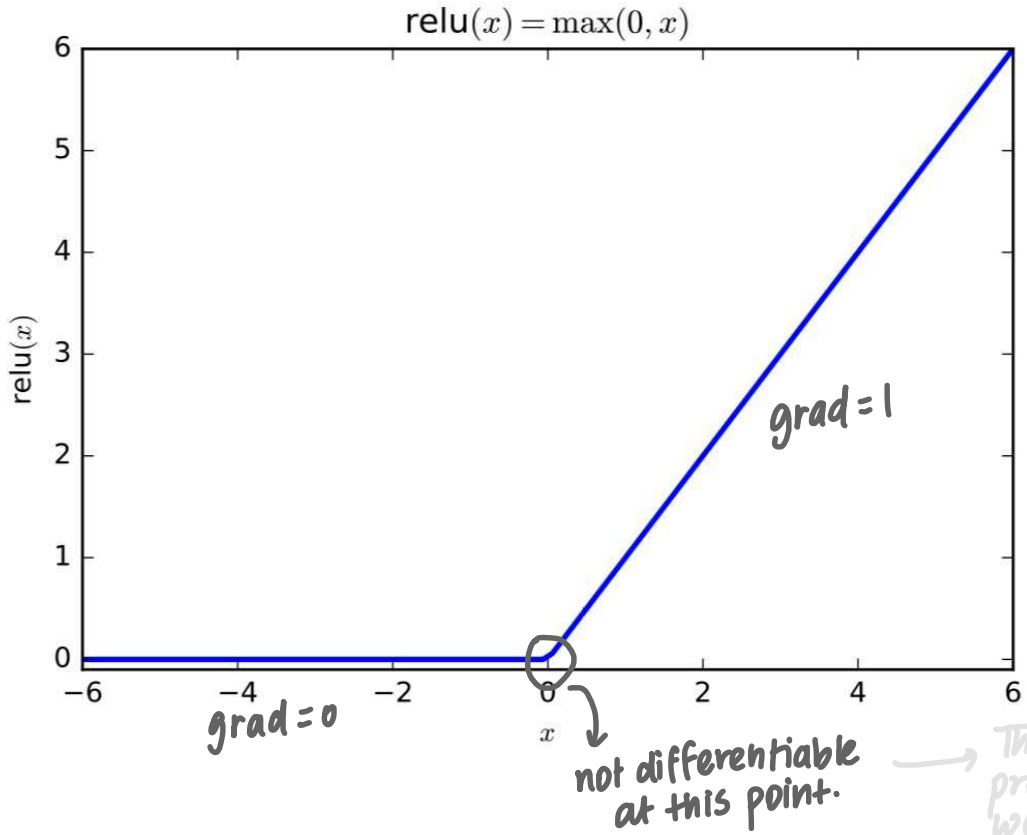
Cons:

- Like the sigmoid unit, when a unit saturates, i.e., its values grow larger or smaller, the unit saturates and no additional learning occurs.

ReLU unit

Rectified linear unit, $\mathbf{ReLU}(x) = \max(0, x)$

```
f = lambda x: x * (x > 0)
```



Its derivative is:

$$\frac{d\mathrm{ReLU}(x)}{dx} = \begin{cases} 1 & x > 0 \\ 0 & x < 0 \end{cases}$$

This function is not differentiable at $x = 0$. However, we can define its subgradient by setting the derivative to be between $[0, 1]$ at $x = 0$.

ReLU unit

Rectified linear unit, $\mathbf{ReLU}(x) = \max(0, x)$

Pros:

- In practice, learning with the ReLU unit converges faster than sigmoid and tanh. AlexNet: relu() was 6x faster to converge than tanh().
- When a unit is active, it behaves as a linear unit.
- The derivative at all points, except $x = 0$, is 0 or 1. When $x > 0$, the gradients are large, and not scaled by second order effects.
- There is no saturation if $x > 0$.

Cons:

- $\mathrm{ReLU}(x)$, like sigmoid, is non-negative. SGD with ReLU() therefore zig-zags.
- $\mathrm{ReLU}(x)$ is not differentiable at $x = 0$. However, in practice, this is not a large issue. A heuristic when evaluating $\frac{d\mathrm{ReLU}(x)}{dx}\Big|_{x=0}$ is to return the left derivative (0) or the right derivative (1); this is reasonable given digital computation is subject to numerical error.
- Learning does not happen for examples that have zero activation. This can be fixed by e.g., using a leaky ReLU or maxout unit.

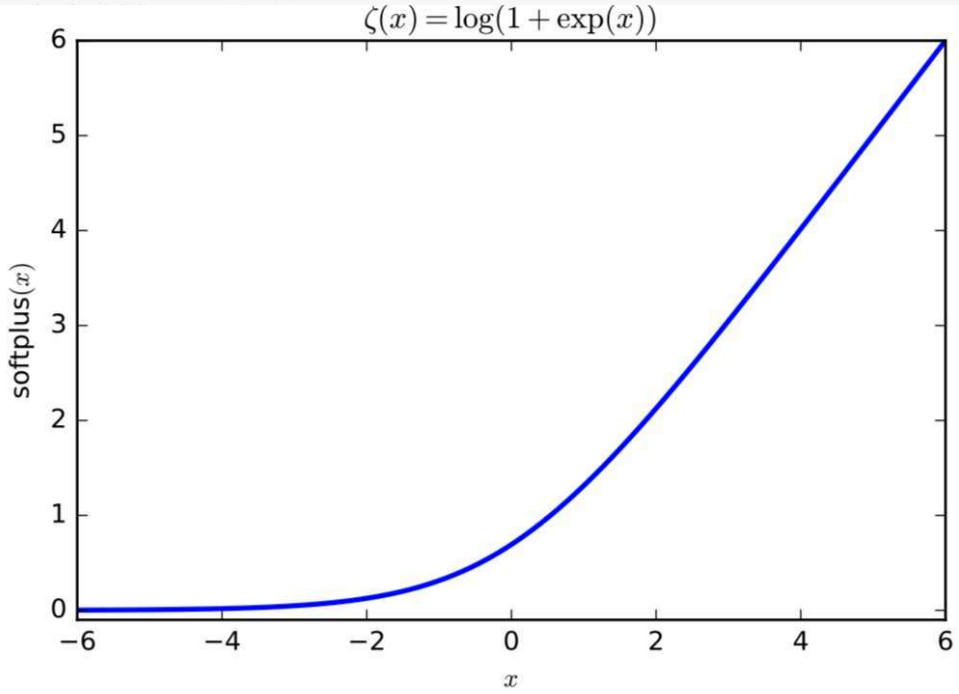
* if any input is <0, then its output is always 0 ⟹ "dead units"

Softplus unit

Softplus unit, $\zeta(x) = \log(1 + \exp(x))$

One may consider using the softplus function, $\zeta(x) = \log(1 + e^x)$, in place of $\mathrm{ReLU}(x)$. Intuitively, this ought to work well as it resembles $\mathrm{ReLU}(x)$ and is differentiable everywhere. However, empirically, it performs worse than $\mathrm{ReLU}(x)$.

```
f = lambda x: np.log(1+np.exp(x))
```



Its derivative is:

$$\frac{d\zeta(x)}{dx} = \sigma(x)$$

Softplus unit

"One might expect it to have an advantage over the [ReLU] due to being differentiable everywhere or due to saturating less completely, but empirically it does not." (Goodfellow et al., p. 191)

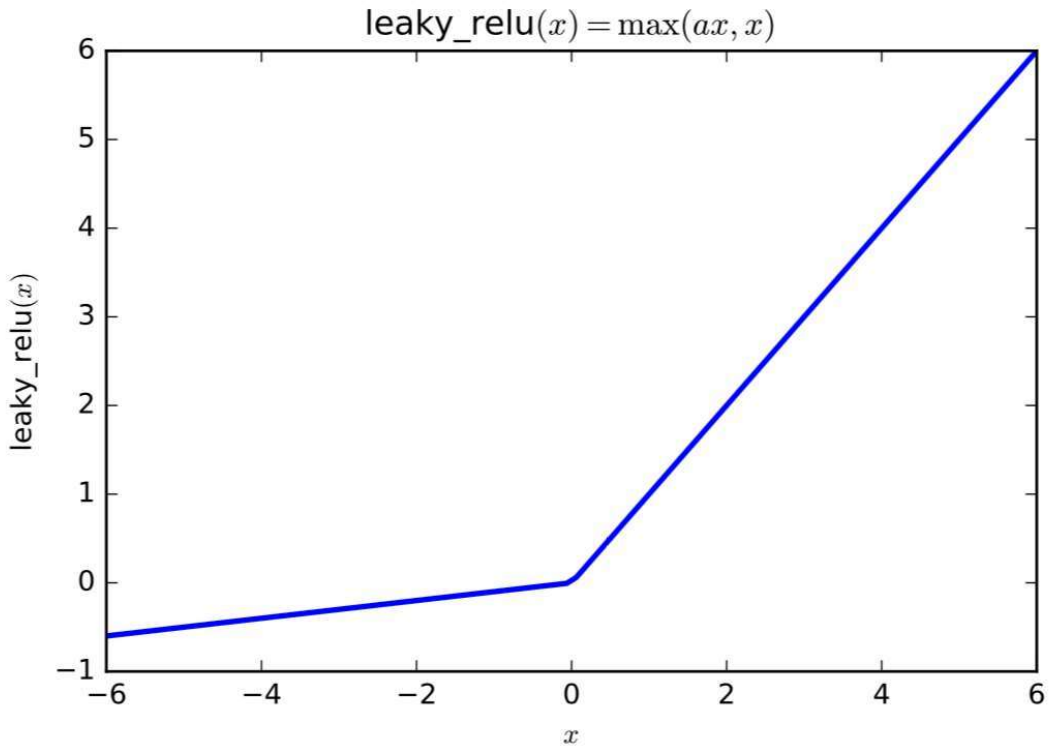
Neuron	MNIST	CIFAR10	NISTP	NORB
***With* unsupervised pre-training**				
Rectifier	**1.20%**	**49.96%**	**32.86%**	**16.46%**
Tanh	**1.16%**	**50.79%**	35.89%	17.66%
Softplus	**1.17%**	**49.52%**	**33.27%**	19.19%
***Without* unsupervised pre-training**				
Rectifier	**1.43%**	**50.86%**	**32.64%**	**16.40%**
Tanh	1.57%	52.62%	36.46%	19.29%
Softplus	1.77%	53.20%	35.48%	17.68%

Glorot et al., 2011a

Leaky ReLU / PReLU unit

Leaky rectified linear unit, $f(x) = \max(\alpha x, x)$

```
f = lambda x: x * (x>0) + 0.1*x * (x<0)
```

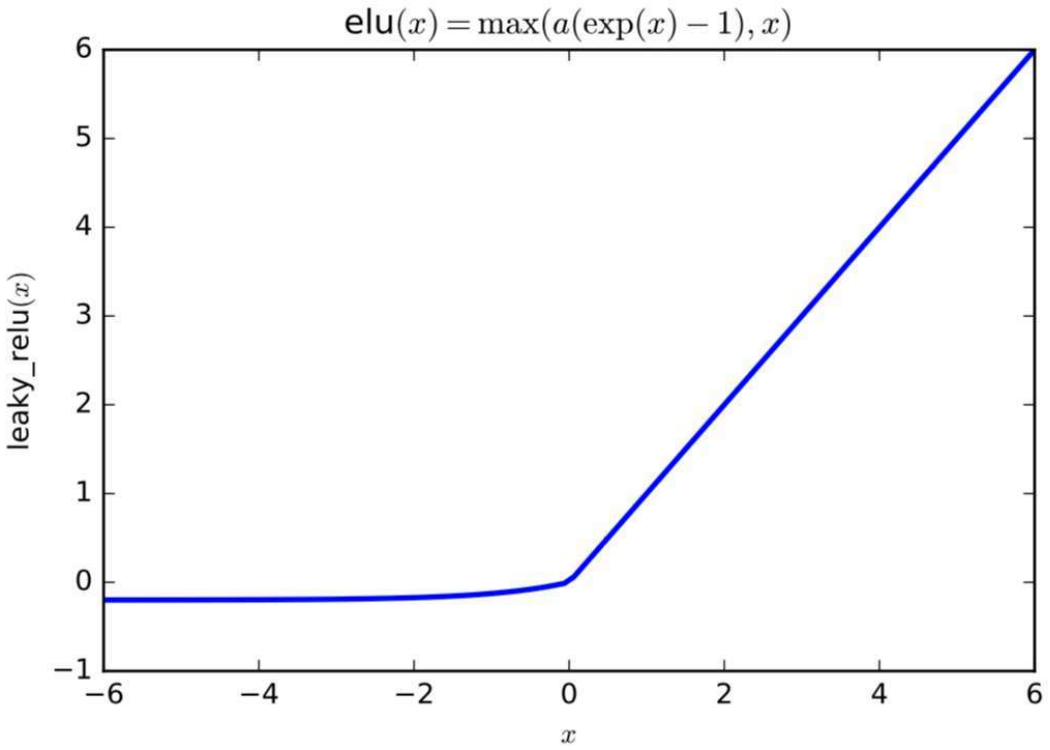


The leaky ReLU avoids the stopping of learning when $x < 0$. α may be treated as a selected hyperparameter, or it may be a parameter to be optimized in learning, in which case it is typically called the "PReLU" for parametrized rectified linear unit.

ELU unit

Exponential linear unit, $f(x) = \max(\alpha(\exp(x) - 1), x)$

```
f = lambda x: x * (x>0) + 0.2*(np.exp(x) - 1) * (x<0)
```



The exponential linear unit avoids the stopping of learning when $x < 0$. A con of this activation function is that it requires computation of the exponential, which is more expensive.

Which LU do I use?

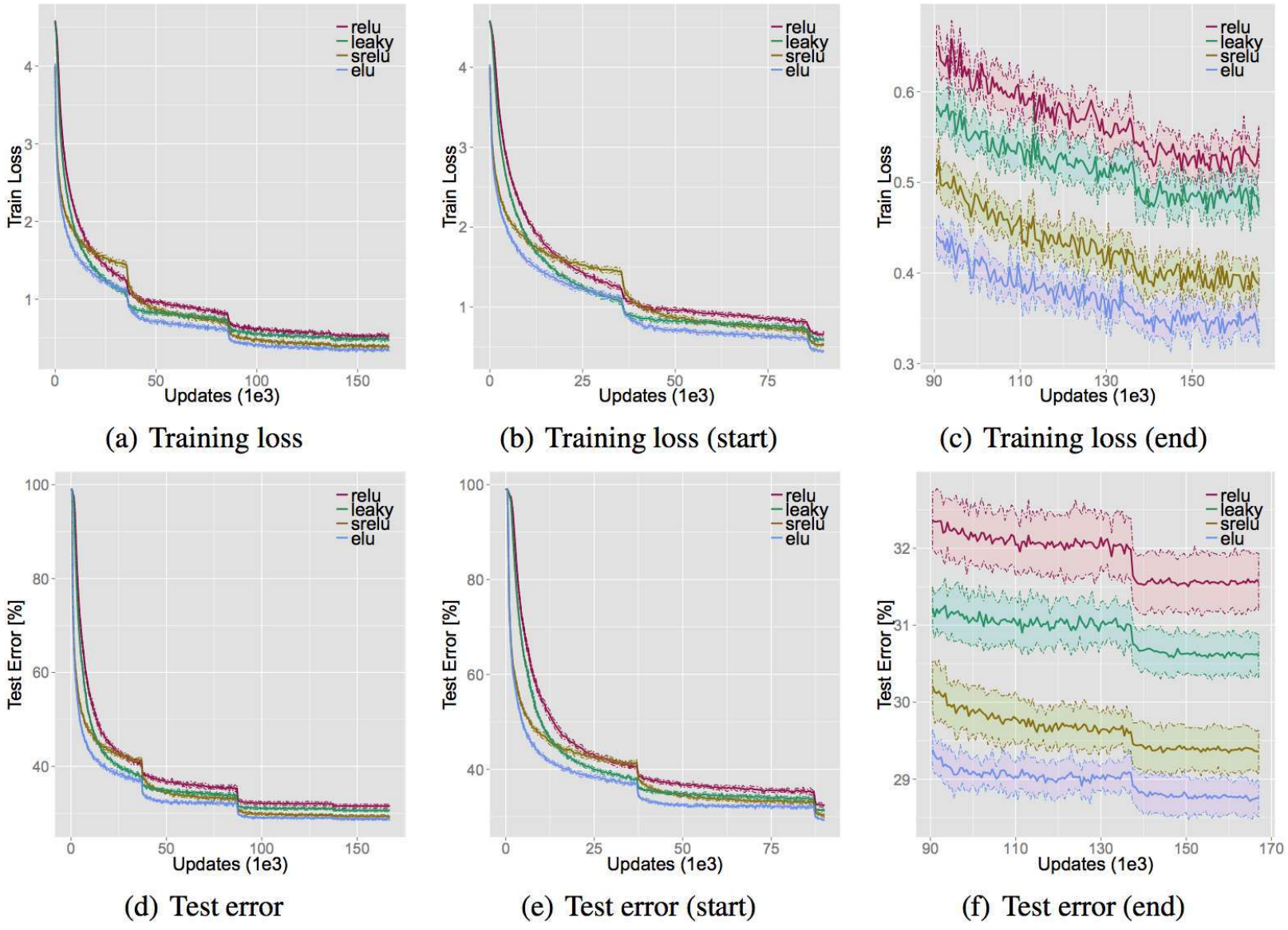
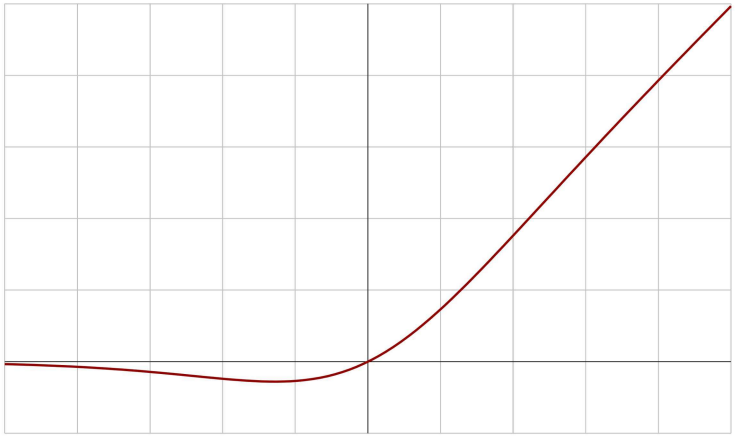


Figure 4: Comparison of ReLUs, LReLUs, and SReLUs on CIFAR-100. Panels (a-c) show the

Clevert et al., 2015

More recently…

$$\text{swish}(x) = x\,\text{sigmoid}(\beta x) = \frac{x}{1+e^{-\beta x}}.$$



Diffusion

$$\text{GELU}(x) = xP(X \le x) = x\Phi(x) = x \cdot \frac{1}{2}\left[1 + \text{erf}(x/\sqrt{2})\right]$$

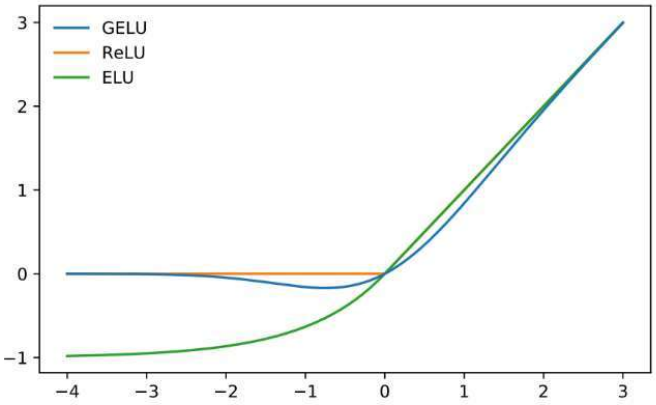


Figure 1: The GELU ($\mu = 0, \sigma = 1$), ReLU, and ELU ($\alpha = 1$).

Nonlinear activation use in ChatGPT.

What activation function do I use?

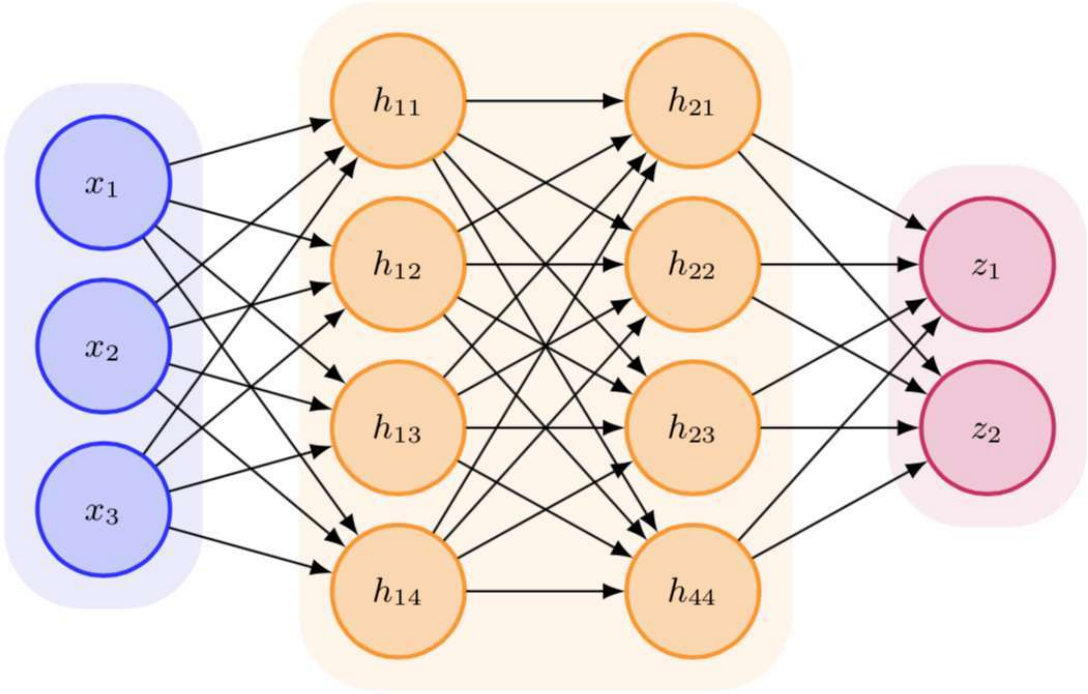
In practice...

In practice...

- The ReLU unit is very popular.
- The sigmoid unit is almost never used; tanh is preferred.
- It may be worth trying out leaky ReLU / PReLU / ELU / maxout for your application.

In practice:
Usually start off with ReLU,
will probably not use tanh.

Back to NN architecture



- Layer 1: $\mathbf{h}_1 = f(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1)$
- Layer 2: $\mathbf{h}_2 = f(\mathbf{W}_2\mathbf{h}_1 + \mathbf{b}_2)$
- $\vdots$
- Layer N: $\mathbf{z} = \mathbf{W}_N\mathbf{h}_{N-1} + \mathbf{b}_N$

What output activation do we use?

The output activation interacts with the cost function

large positive $z \longrightarrow$ class 1 ; $y^{(i)} = 1$
large negative $z \longrightarrow$ class 0 ; $y^{(i)} = 0$

What outputs and cost functions?

There are several options to process the output scores, $\mathbf{z}$, to ultimately arrive at a cost function. The choice of output units interacts with what cost function to use.

$\sigma(z) = \Pr\{x^{(i)} \text{ belonging to class 1}\}$

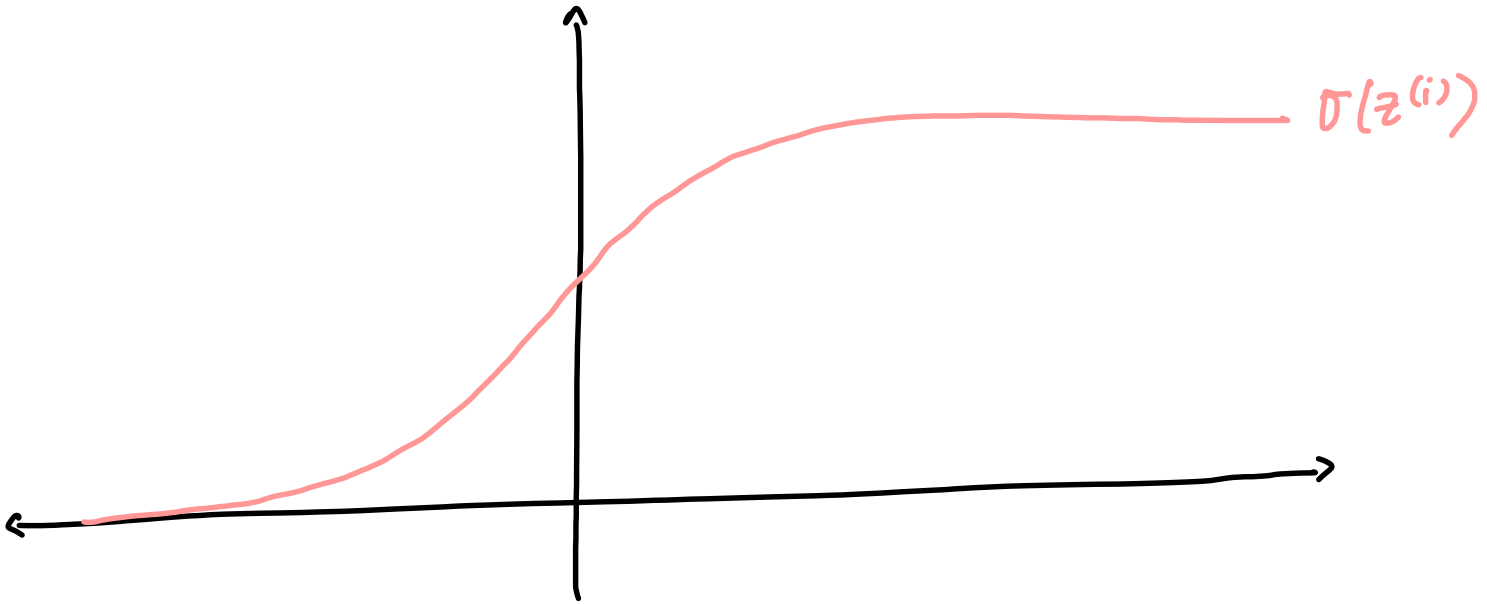
Example: Consider a neural network that produces a single score, z, for binary classification. As the output unit, we choose the sigmoid nonlinearity, so that $\hat{y} = \sigma(z)$. On a given example, $y^{(i)}$ is either 0 or 1, and $\hat{y}^{(i)} = \sigma(z^{(i)})$ can be interpreted as the algorithm's probability the output is in class 1. Is it better to use mean-square error or cross-entropy (i.e., corresponding to maximum-likelihood estimation) as the cost function? For n examples:

$$\text{MSE} = \frac{1}{2}\sum_{i=1}^{n}\left(y^{(i)} - \sigma(z^{(i)})\right)^2$$

$$\text{CE} = -\sum_{i=1}^{n}\left[y^{(i)}\log\sigma(z^{(i)}) + (1-y^{(i)})\log(1-\sigma(z^{(i)}))\right]$$

A picture for intuition

$y^{(i)} = 1$



MSE for example $x^{(i)}$: $MSE^{(i)} = (y^{(i)} - \sigma(z^{(i)}))^2$

$$\frac{\partial MSE^{(i)}}{\partial z^{(i)}} = \frac{\partial \sigma(z^{(i)})}{\partial z^{(i)}} \cdot \frac{\partial MSE^{(i)}}{\partial \sigma(z^{(i)})}$$

when $z^{(i)}$ is very negative, the gradient is approx. zero.

$(y^{(i)} - \sigma(-50))^2 \approx (1-0)^2 \approx 1$

$(y^{(i)} - \sigma(-40))^2 \approx (1-0)^2 \approx 1$

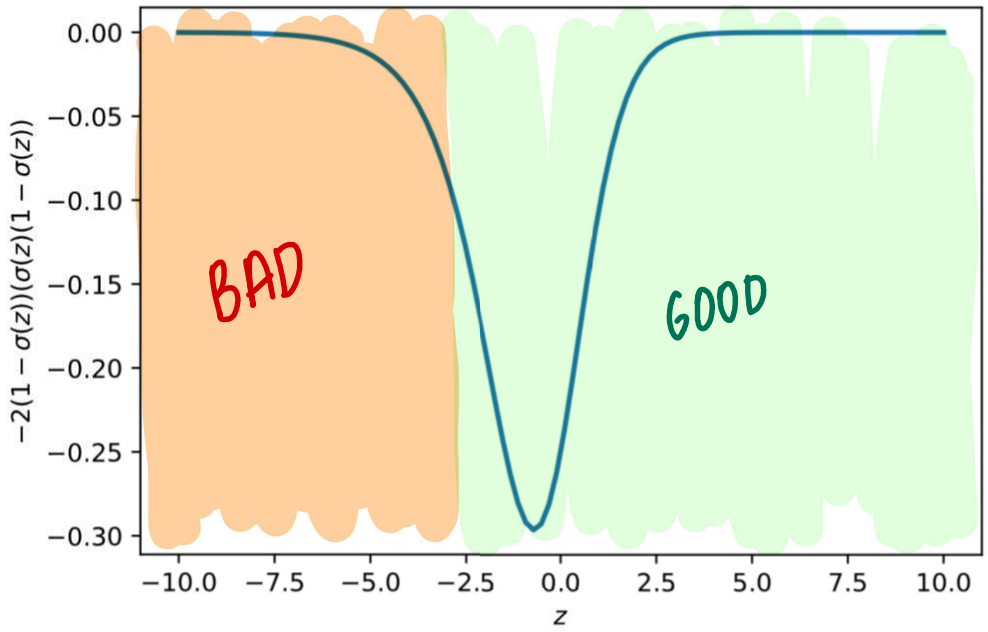
The output activation interacts with the cost function

What outputs and cost functions? (cont.)

Example (cont): Consider just one example, where $y^{(i)} = 1$. For this example,

$$\frac{\partial \text{MSE}}{\partial z^{(i)}} = -2(y^{(i)} - \sigma(z^{(i)}))(\sigma(z^{(i)})(1 - \sigma(z^{(i)})))$$

This derivative looks like the following:



When z is very negative, indicating a large MSE, the gradient saturates to zero, and no learning occurs.

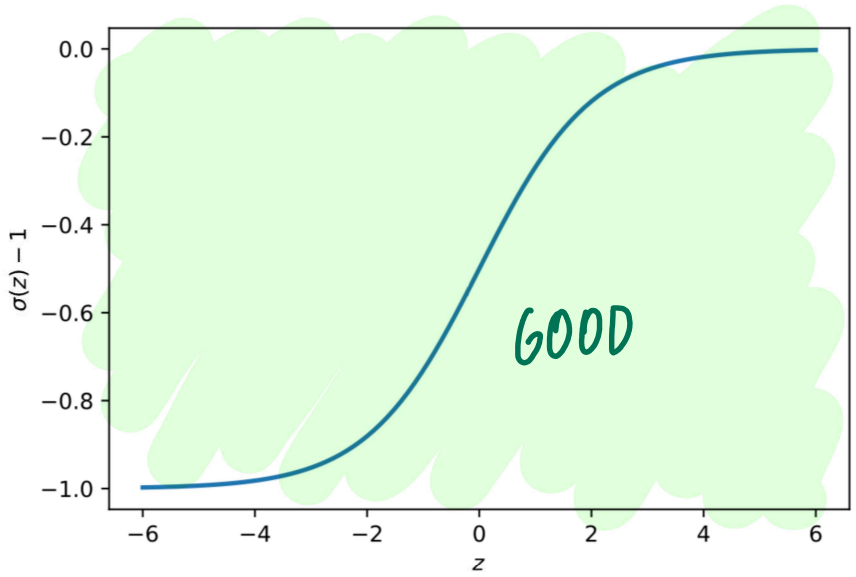
The output activation interacts with the cost function

What outputs and cost functions? (cont.)

Example (cont): Now let's consider the cross-entropy. For one example, where $y^{(i)} = 1$,

$$\frac{\partial \text{CE}}{\partial z^{(i)}} = \sigma(z^{(i)}) - 1$$

This derivative looks like the following:



Notice that when z is very negative, learning will occur, and it will only "stall" when z gets close to the right answer.

Output activations

- Softmax output: $\hat{\mathbf{y}}_i = \text{softmax}_i(\mathbf{z})$.

  The softmax is the generalization of the sigmoid output to multiple classes.

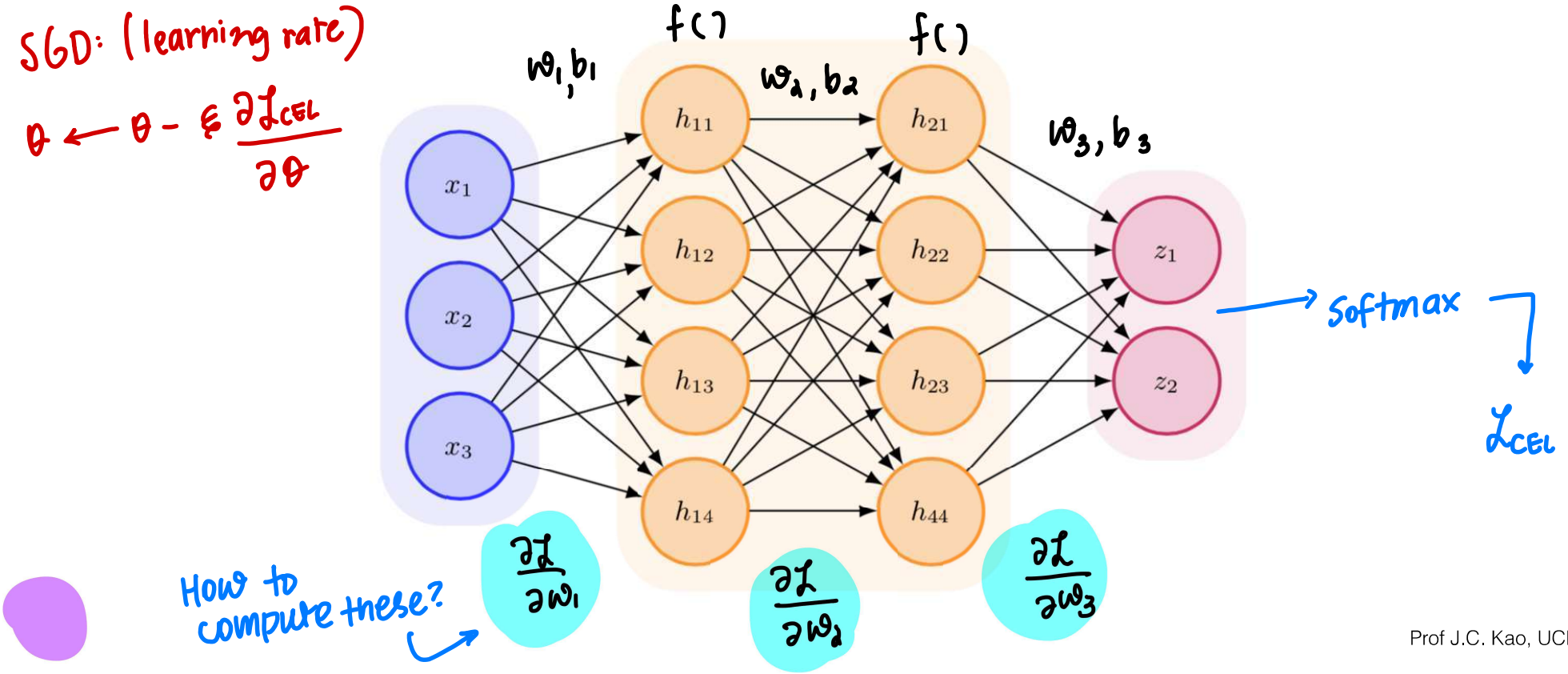
A softmax output activation is fairly common.

What next?

Now that we've defined all the elements of the neural network, the question now becomes: how do we learn its parameters?

The short answer is that we use versions of gradient descent.

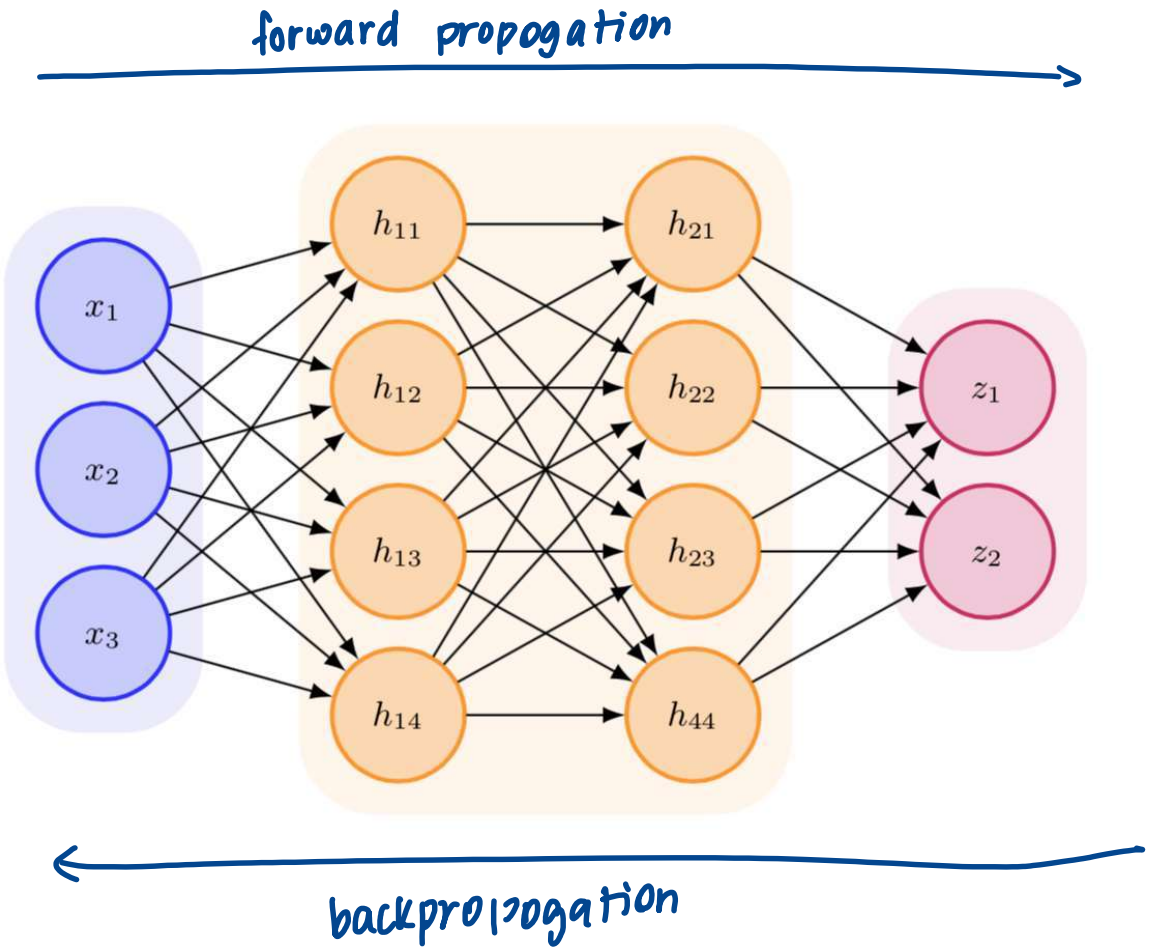
However, neural networks architectures have units that are several layers from the output. In these scenarios, how do we arrive at the gradient?



Backpropagation

operationalized way to apply chain rule for calculus.

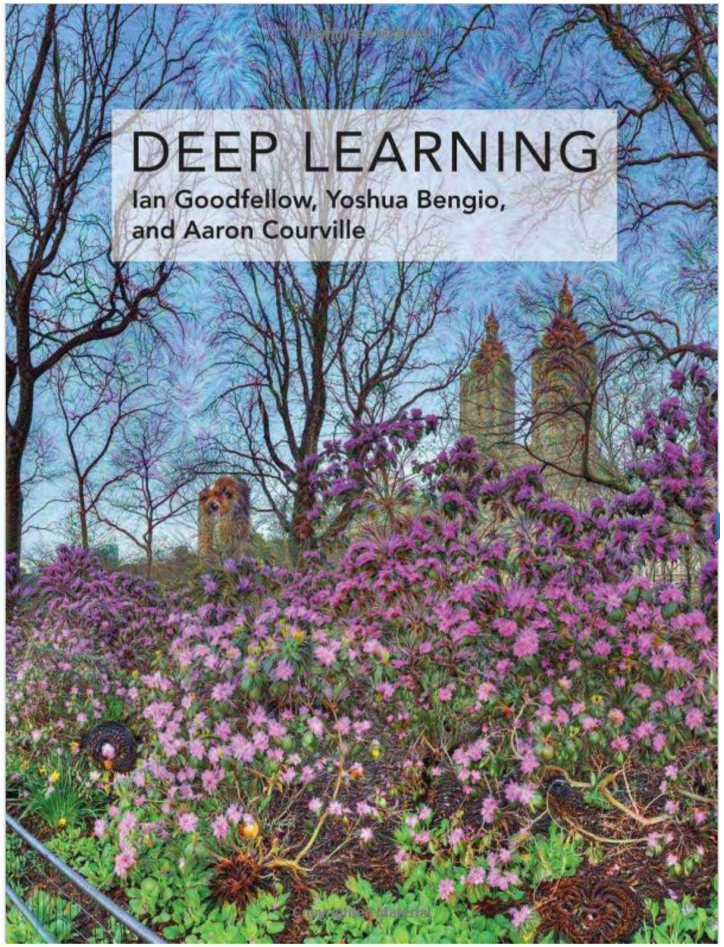
In this lecture, we'll introduce backpropagation as a technique to calculate the gradient of the loss function with respect to parameters in a neural network.



Reading

Reading:

Deep Learning, 6.5-6.6

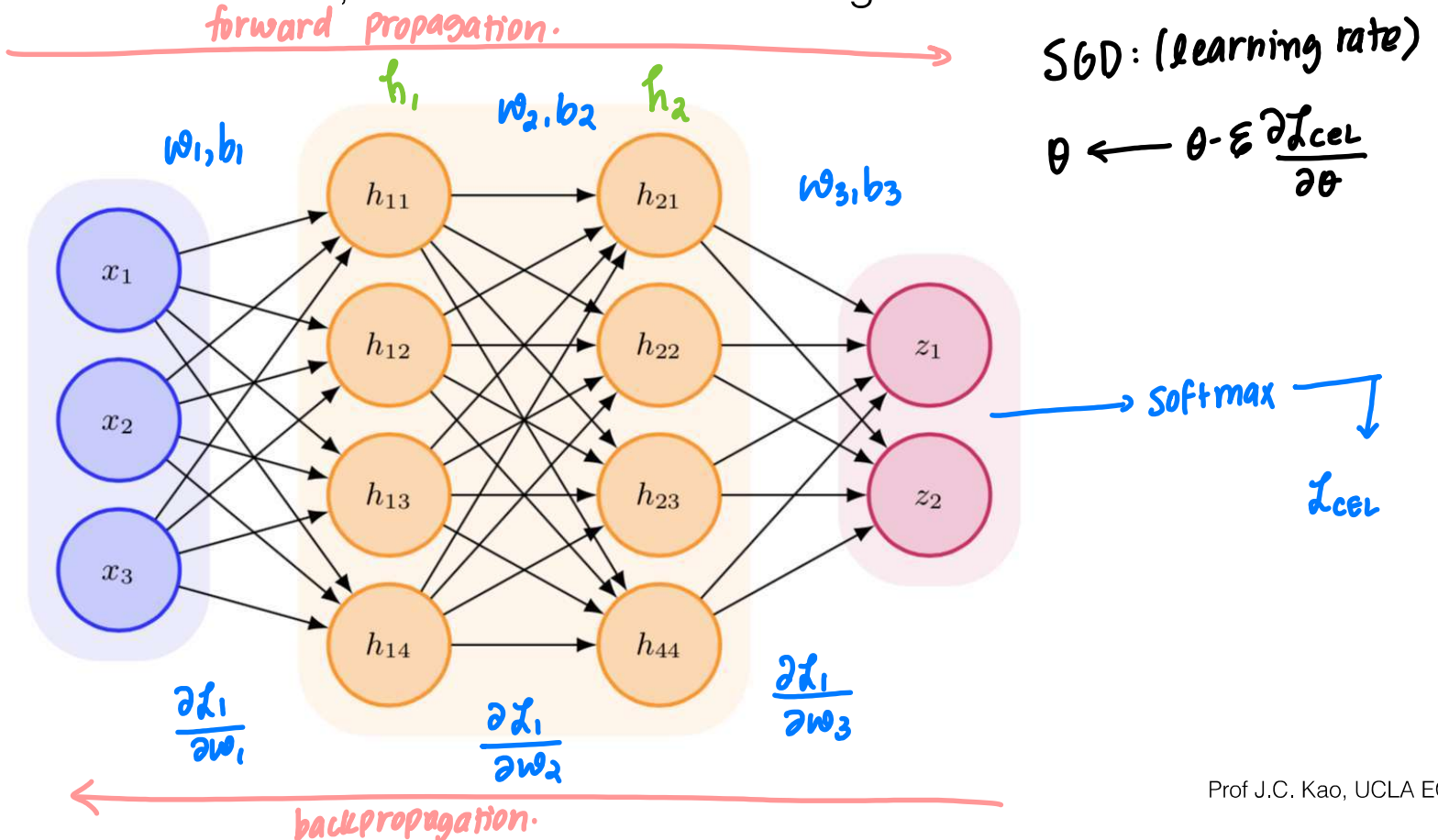


What next?

Now that we've defined all the elements of the neural network, the question now becomes: how do we learn its parameters?

The short answer is that we use versions of gradient descent.

However, neural networks architectures have units that are several layers from the output. In these scenarios, how do we arrive at the gradient?



Backpropagation

Intuitively, backpropagation is the application of the chain rule for derivatives.

Motivation for backpropagation

To do gradient descent, we need to calculate the gradient of the objective with respect to the parameters, θ. However, in a neural network, some parameters are not directly connected to the output. How do we calculate then the gradient of the objective with respect to these parameters? Backpropagation answers this question, and is a general application of the chain rule for derivatives.

Backpropagation

Nomenclature

Forward propagation:

- Forward propagation is the act of calculating the values of the hidden and output units of the neural network given an input.
- It involves taking input $\mathbf{x}$, propagating it to through each hidden unit sequentially, until you arrive at the output $\mathbf{y}$. From forward propagation, we can also calculate the cost function $J(\theta)$.
- In this manner, the forward propagated signals are the activations.
- With the input as the "start" and the output as the "end," information propagates in a forward manner.

Backpropagation (colloquially called backprop):

- As its name suggests, now information is passed backward through the network, from the cost function and outputs to the inputs.
- The signal that is backpropagated are the gradients.
- It enables the calculation of gradients at every stage going back to the input layer.

Backpropagation

Why do we need backpropagation?

- Backpropagation is computationally efficient because we will find that most of the terms used in backpropagation can be cached from the forward pass.
- Informally (at least for me) sometimes taking analytical multivariate gradients can be challenging. Backpropagation breaks down these multivariate gradients into easier steps.
- It "operationalizes" gradients, giving a simple algorithm to compute gradients in a neural network. This is critical for neural network libraries that include auto-differentiation.

A few further notes on backpropagation.

- Backpropagation is not the learning algorithm. It's the method of computing gradients.
- Backpropagation is not specific to multilayer neural networks, but a general way to compute derivatives of functions.

A simple example

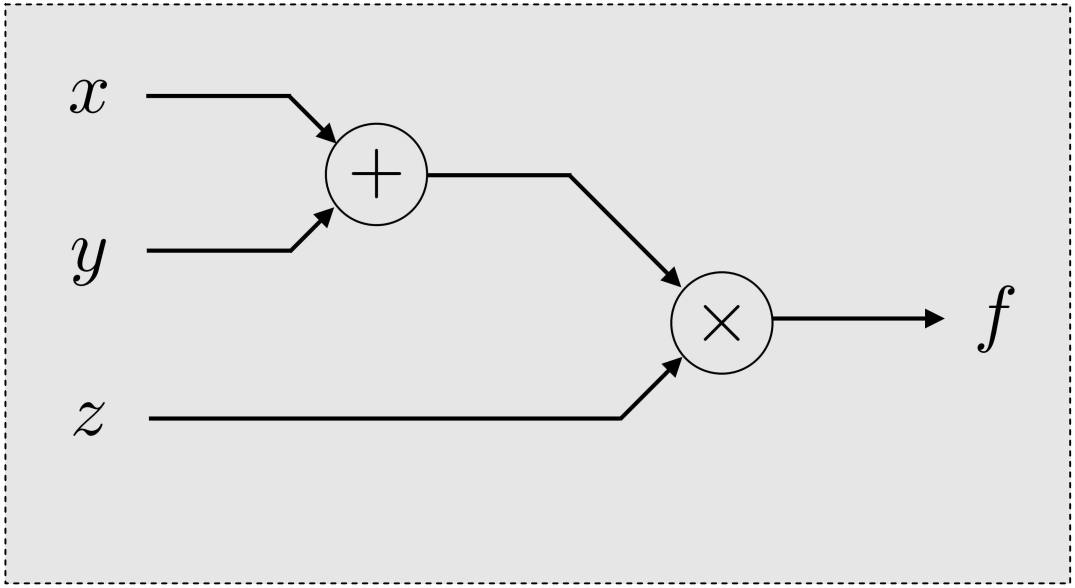
$$f(x, y, z) = (x + y)z$$

$\frac{\partial f}{\partial z} = x + y$

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

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

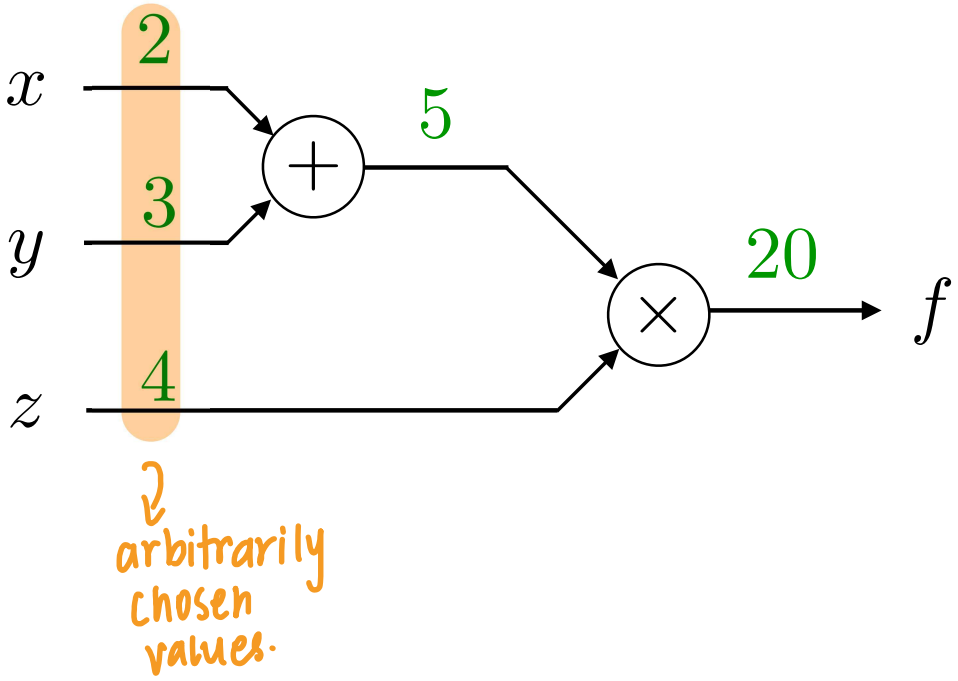
COMPUTATIONAL GRAPH



A simple example

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

Forward propagation:



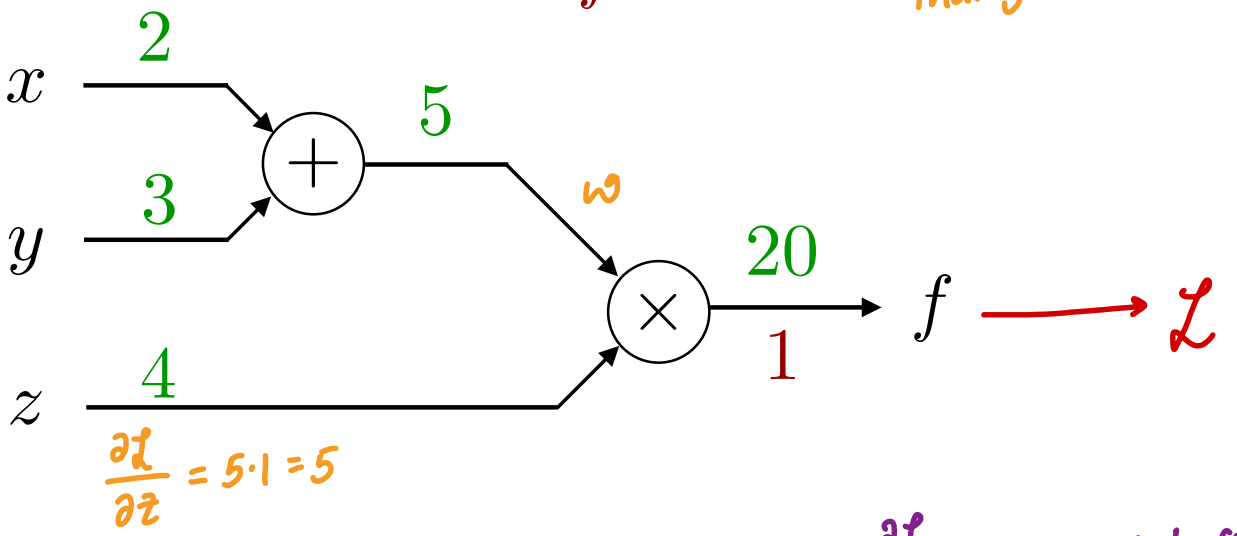
A simple example

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

Backpropagation:

$$\frac{\partial \mathcal{L}}{\partial f} = 1$$

softmax.loss_and_grad()
(now we want to backpropagate that gradient)



$\frac{\partial \mathcal{L}}{\partial z}$ ⟹ want to compute.

Note: $\frac{\partial \mathcal{L}}{\partial z} = \frac{\partial f}{\partial z} \cdot \frac{\partial \mathcal{L}}{\partial f}$

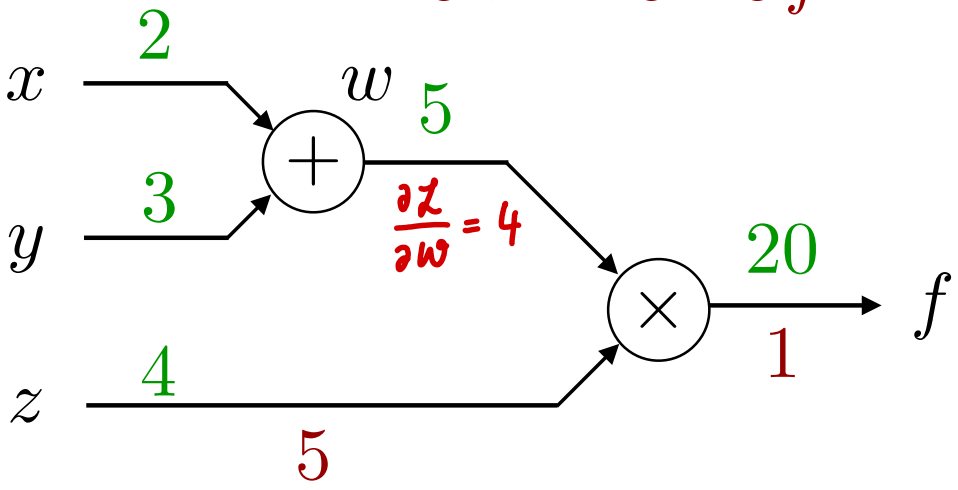
$f = w \cdot z \Longrightarrow \frac{\partial f}{\partial z} = w$

A simple example

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

Backpropagation:

$$\frac{\partial \mathcal{L}}{\partial z} = \frac{\partial f}{\partial z}\frac{\partial \mathcal{L}}{\partial f}$$



Now, want to compute

$$\frac{\partial \mathcal{L}}{\partial w} = \frac{\partial f}{\partial w}\cdot\frac{\partial \mathcal{L}}{\partial f}$$

$\frac{\partial f}{\partial w} = z$, $\frac{\partial \mathcal{L}}{\partial f} = 1$

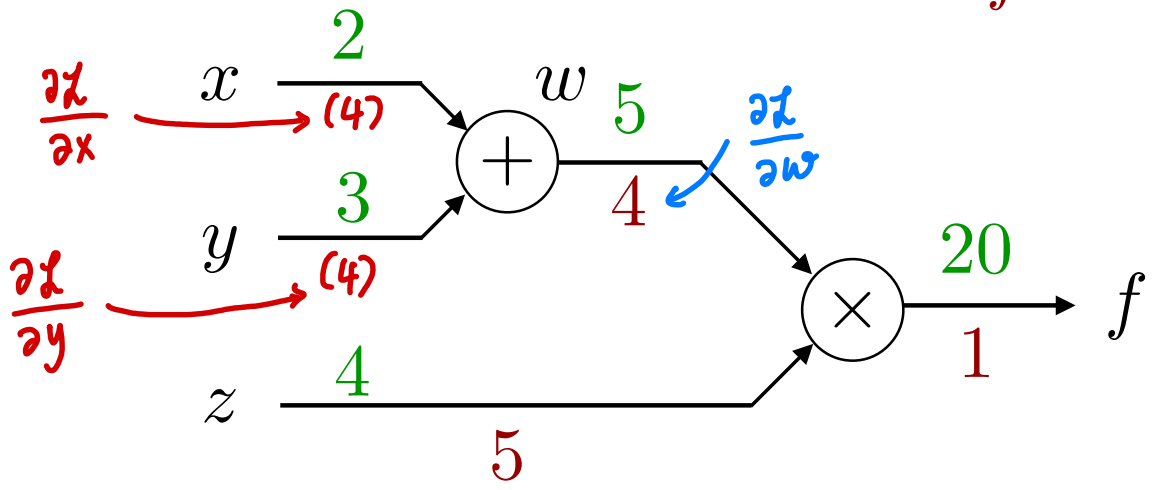
$$\frac{\partial \mathcal{L}}{\partial w} = z \cdot 1 = 4 \cdot 1 = 4$$

A simple example

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

Backpropagation:

$$\frac{\partial \mathcal{L}}{\partial w} = \frac{\partial f}{\partial w}\frac{\partial \mathcal{L}}{\partial f}$$



$$\frac{\partial \mathcal{L}}{\partial x} = \frac{\partial \mathcal{L}}{\partial w}\cdot\frac{\partial w}{\partial x} = 1\cdot 4 = 4$$

$w = x + y$

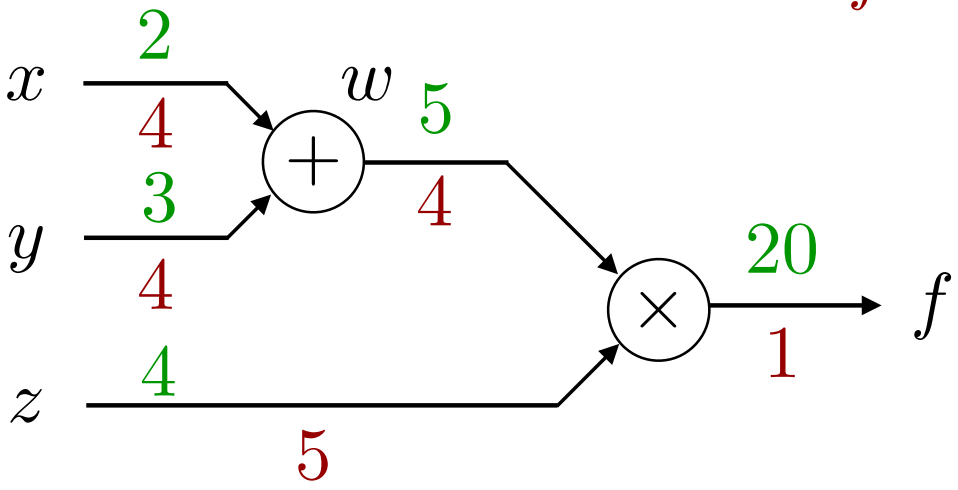
$\frac{\partial w}{\partial x} = 1$; $\frac{\partial w}{\partial y} = 1$

A simple example

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

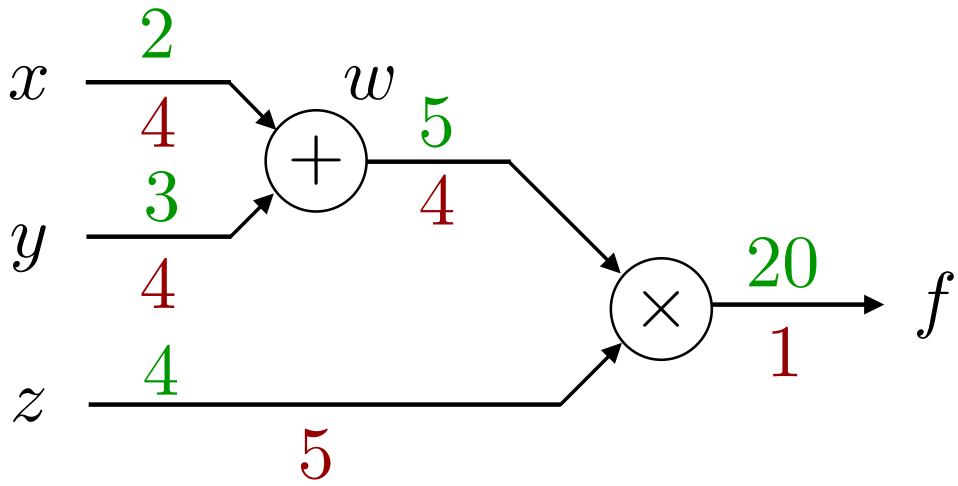
Backpropagation:

$$\frac{\partial \mathcal{L}}{\partial x} = \frac{\partial w}{\partial x}\frac{\partial f}{\partial w}\frac{\partial \mathcal{L}}{\partial f}$$



Idea: computational graphs apply to gradients

In the **forward pass**, we apply a function to the node inputs to calculate an output.
In the **backward pass**, we take the **upstream derivative** and apply a **local gradient** to calculate the backpropagated derivative.

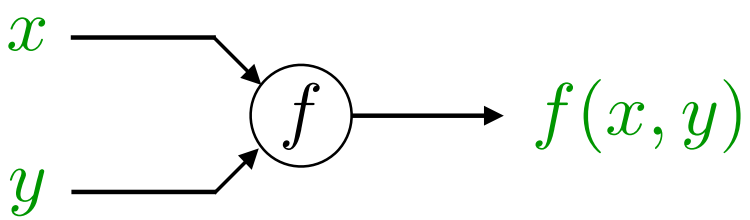


Idea: computational graphs apply to gradients

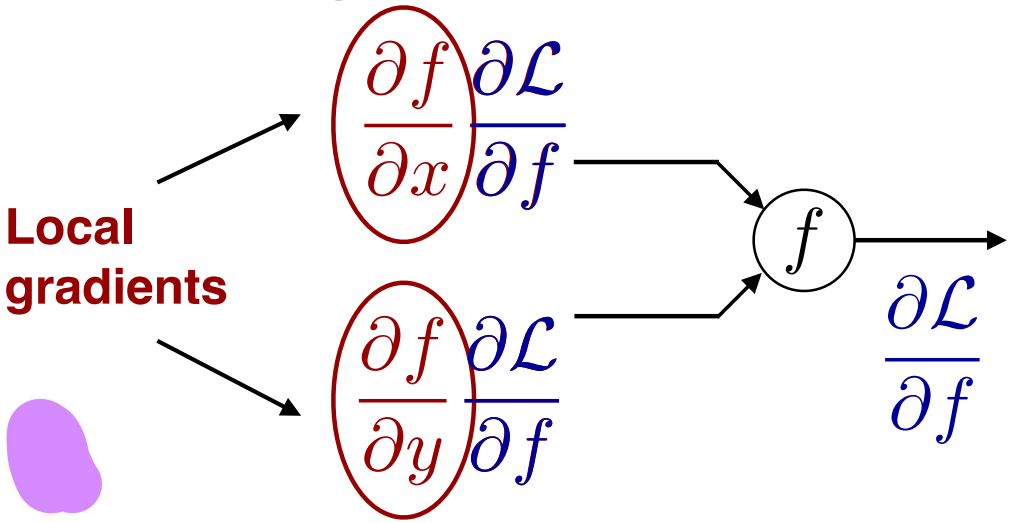
In the **forward pass**, we apply a function to the node inputs to calculate an output.
In the **backward pass**, we take the **upstream derivative** and apply a **local gradient** to calculate the backpropagated derivative.

We already know $\frac{\partial L}{\partial w}$ for Softmax from HW02.

Forward pass:



Backward pass:

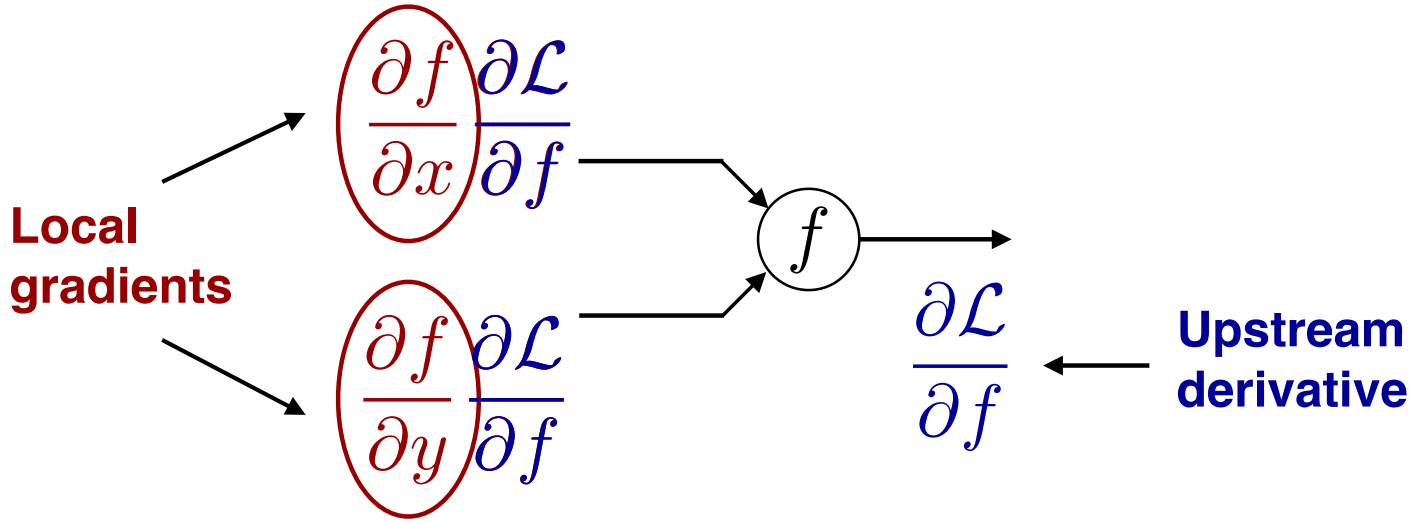


$\frac{\partial L}{\partial x} = \frac{\partial f}{\partial x} \cdot \frac{\partial L}{\partial f}$

"local gradient"

"upstream gradient" (coming from deeper layer of N.N.)

Idea: computational graphs apply to gradients



The basic intuition of backpropagation is that we break up the calculation of the gradient into **small and simple steps**. Each of these nodes in the graph is a straightforward gradient calculation, where we multiply an **input** (the "upstream derivative") with a **local gradient** (an application of the chain rule).

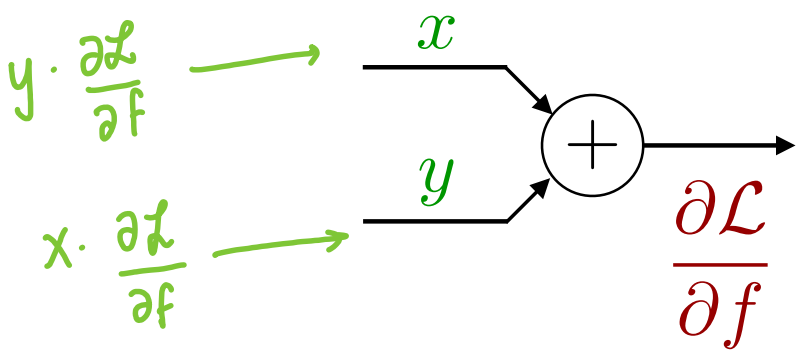
Composing all of these gradients together returns the overall gradient.

A gate view of gradients

Interpreting backpropagation as gradient “gates”:

$f = x \cdot y$

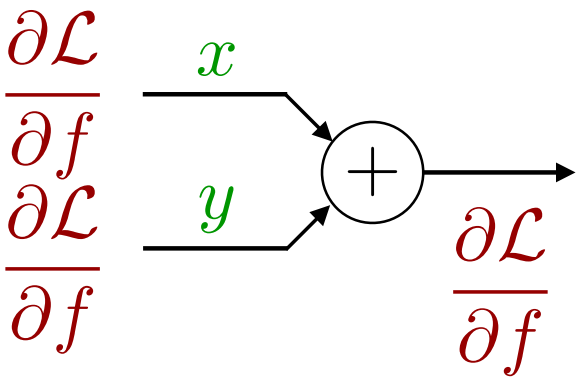
$\frac{\partial f}{\partial x} = y$ $\quad$ $\frac{\partial f}{\partial y} = x$



A gate view of gradients

Interpreting backpropagation as gradient “gates”:

Add gate: distributes the gradient



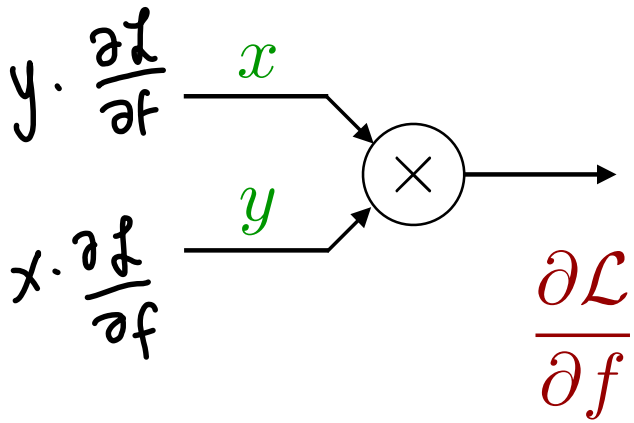
A gate view of gradients

Interpreting backpropagation as gradient “gates”:

Add gate: distributes the gradient

$f = x \cdot y$

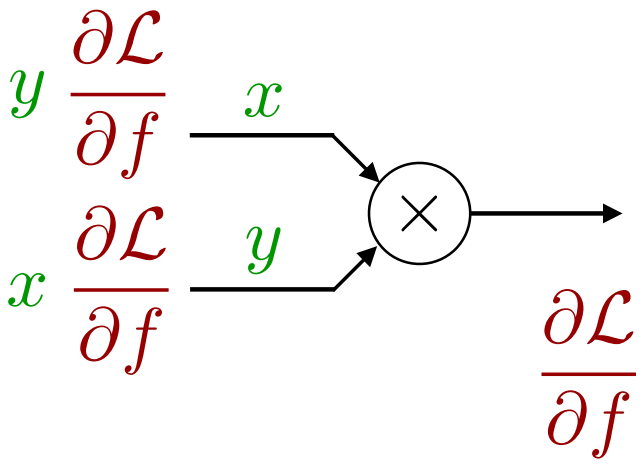
$\frac{\partial f}{\partial x} = y$ $\quad$ $\frac{\partial f}{\partial y} = x$



A gate view of gradients

Interpreting backpropagation as gradient “gates”:

Add gate: distributes the gradient
Mult gate: switches the gradient



A gate view of gradients

Interpreting backpropagation as gradient “gates”:

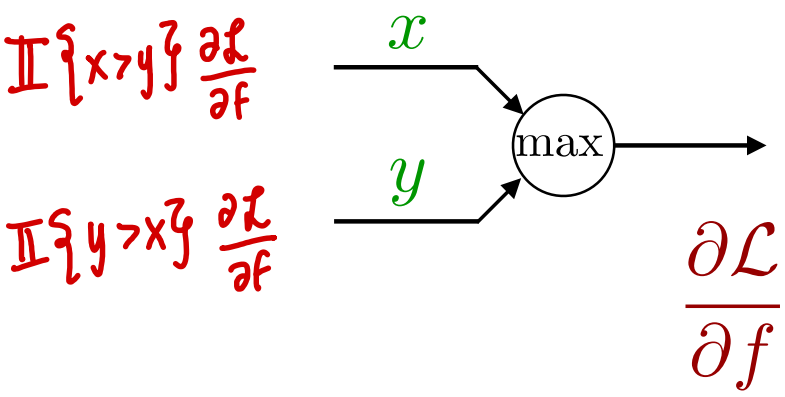
Add gate: distributes the gradient
Mult gate: switches the gradient

$\text{relu}(x) = \max(x, 0)$

$f = \max(x, y)$ ← local operation

$$\frac{\partial f}{\partial x} = \begin{cases} 1 & \text{if } x \geq y \\ 0 & \text{else.} \end{cases}$$

$$= \mathbb{1}\{x > y\}$$

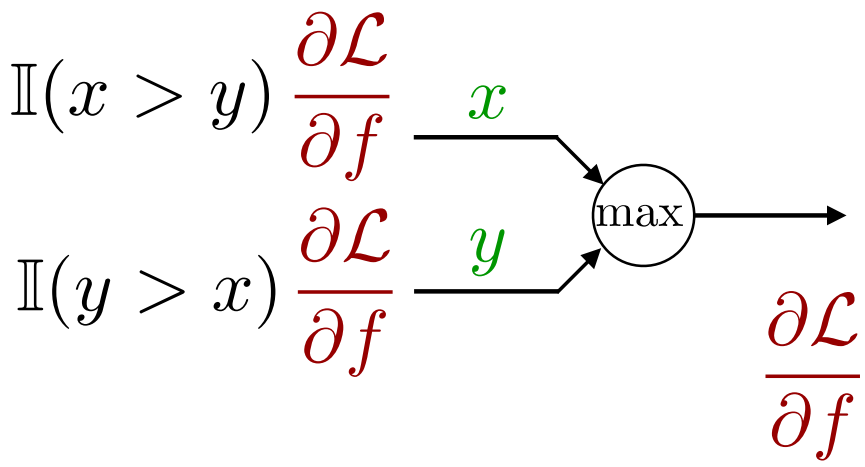


when $x = y$
↳ you can choose which wire to route it to
↳ compute gradient and distribute evenly across both wires.

A gate view of gradients

Interpreting backpropagation as gradient “gates”:

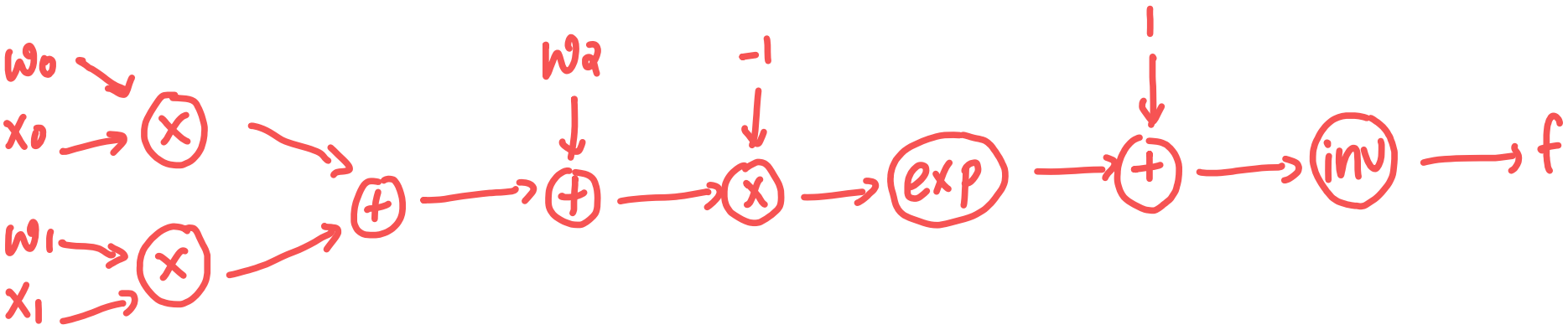
Add gate: distributes the gradient
Mult gate: switches the gradient
Max gate: routes the gradient



A more involved scalar example

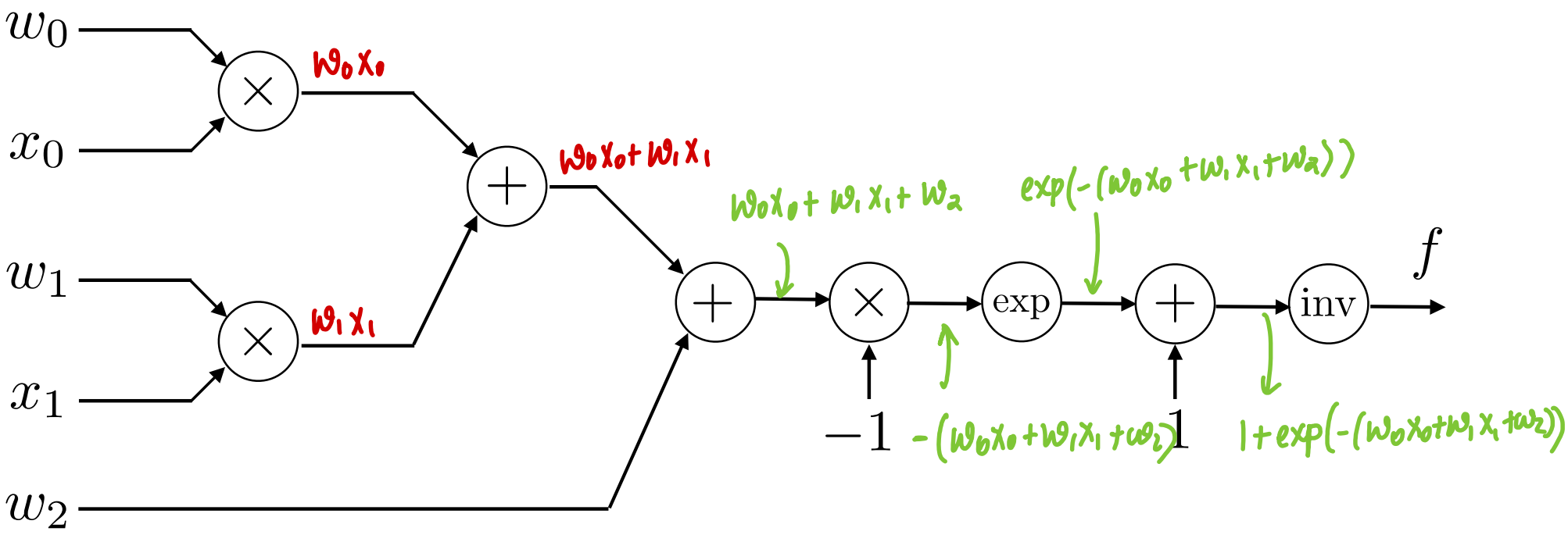
$$f(\mathbf{w}, \mathbf{x}) = \frac{1}{1 + \exp(-(w_0 x_0 + w_1 x_1 + w_2))}$$

practice



A more involved scalar example

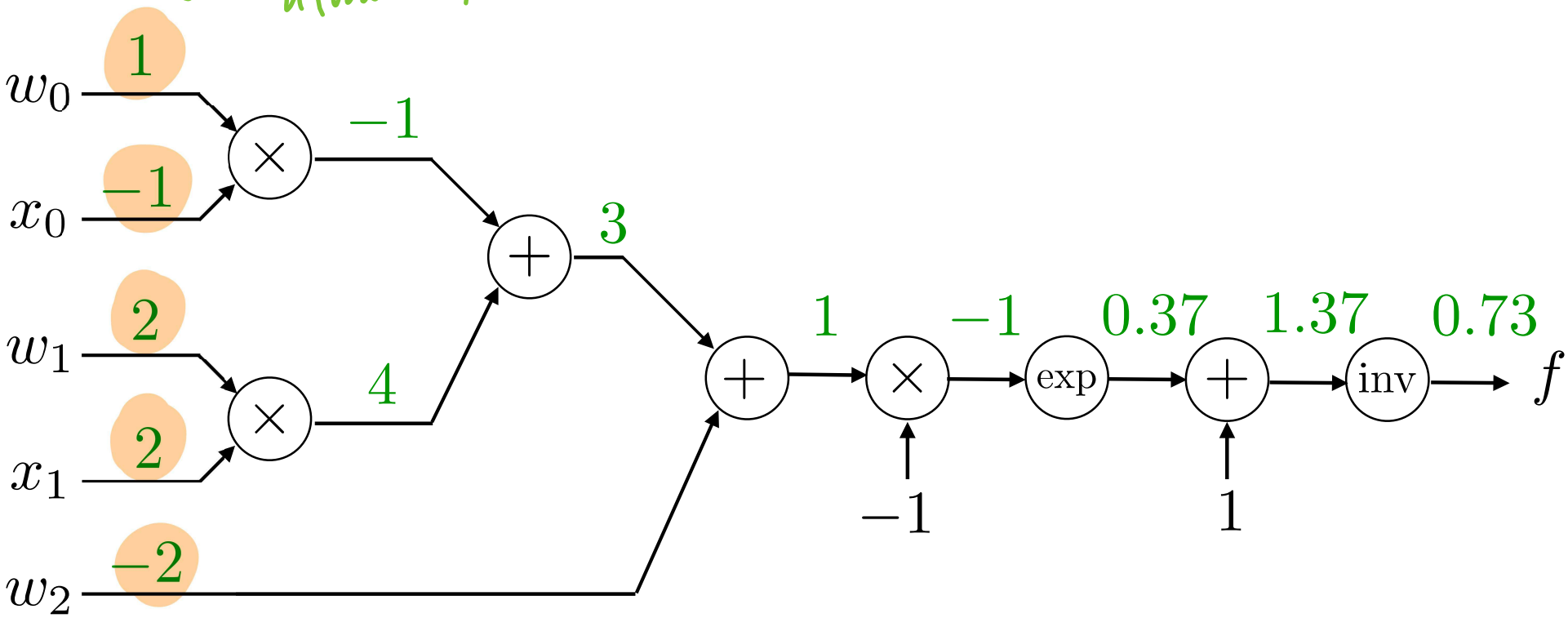
$$f(\mathbf{w}, \mathbf{x}) = \frac{1}{1 + \exp(-(w_0 x_0 + w_1 x_1 + w_2))}$$



A more involved scalar example

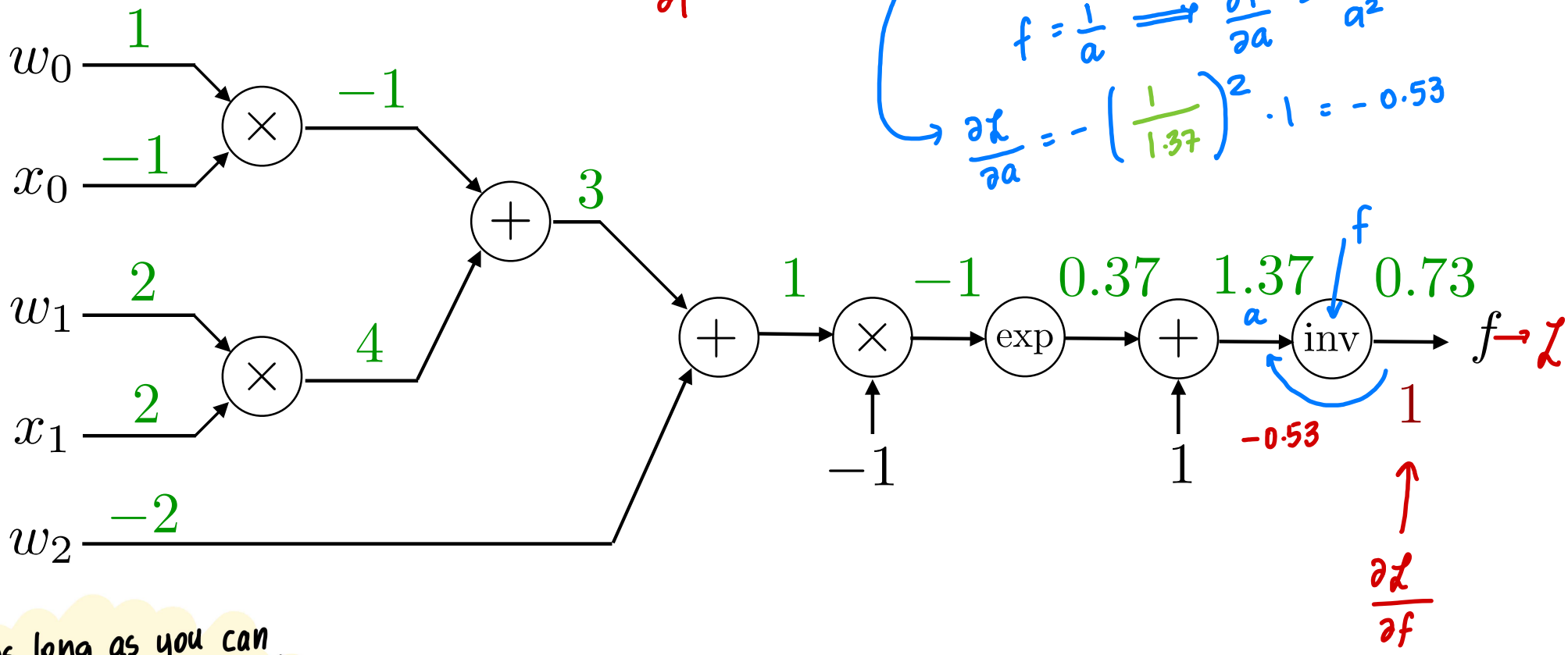
$$f(\mathbf{w}, \mathbf{x}) = \frac{1}{1 + \exp(-(w_0 x_0 + w_1 x_1 + w_2))}$$

Chose arbitrary input values and did a forward pass



A more involved scalar example

$$f(\mathbf{w}, \mathbf{x}) = \frac{1}{1 + \exp(-(w_0 x_0 + w_1 x_1 + w_2))}$$



As long as you can compute gradient of a local operation, you can back propogate.

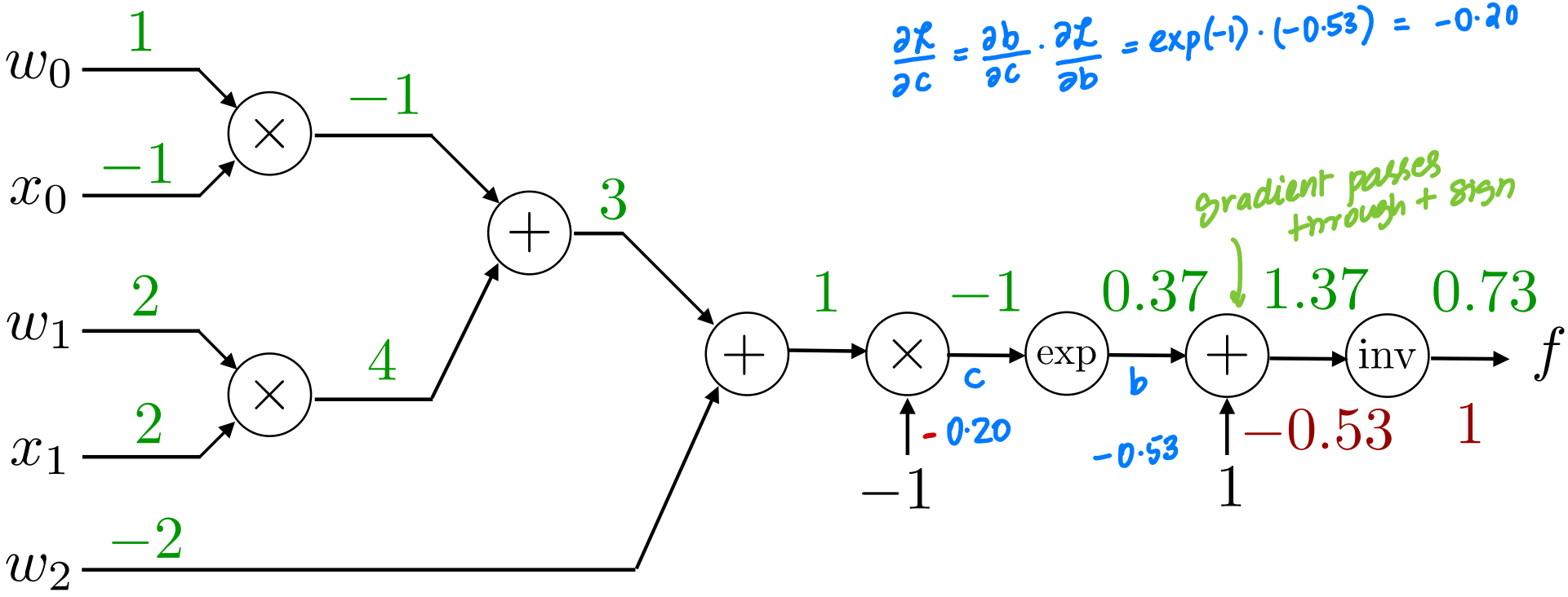
A more involved scalar example

$$f(\mathbf{w}, \mathbf{x}) = \frac{1}{1 + \exp(-(w_0 x_0 + w_1 x_1 + w_2))}$$

$b = \exp(c) = e^c$

$\frac{\partial b}{\partial c} = \exp(c)$

$\frac{\partial \mathcal{L}}{\partial c} = \frac{\partial b}{\partial c} \cdot \frac{\partial \mathcal{L}}{\partial b} = \exp(-1) \cdot (-0.53) = -0.20$

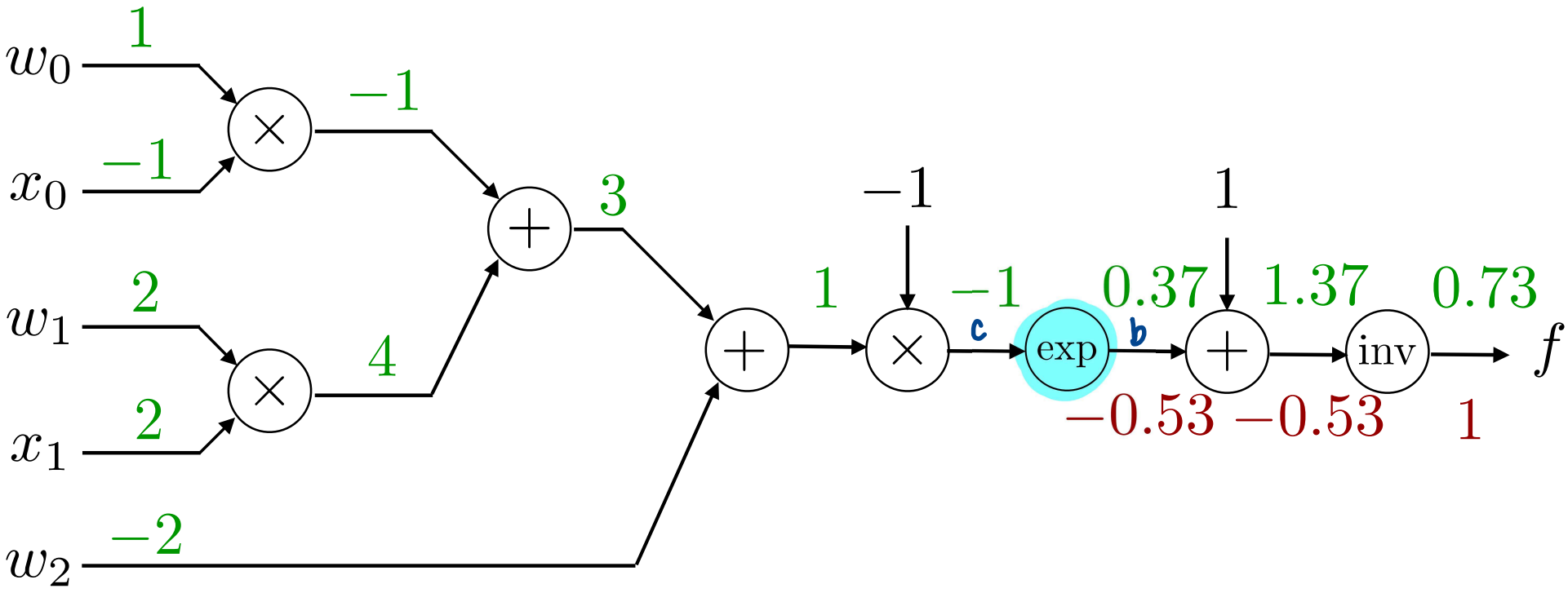


$$\frac{\partial \mathcal{L}}{\partial z} = \frac{\partial f}{\partial z} \frac{\partial \mathcal{L}}{\partial f} \qquad \frac{\partial f}{\partial z} = -\frac{1}{z^2}$$

A more involved scalar example

$$f(\mathbf{w}, \mathbf{x}) = \frac{1}{1 + \exp(-(w_0 x_0 + w_1 x_1 + w_2))}$$

$b = \exp(c) = e^c$

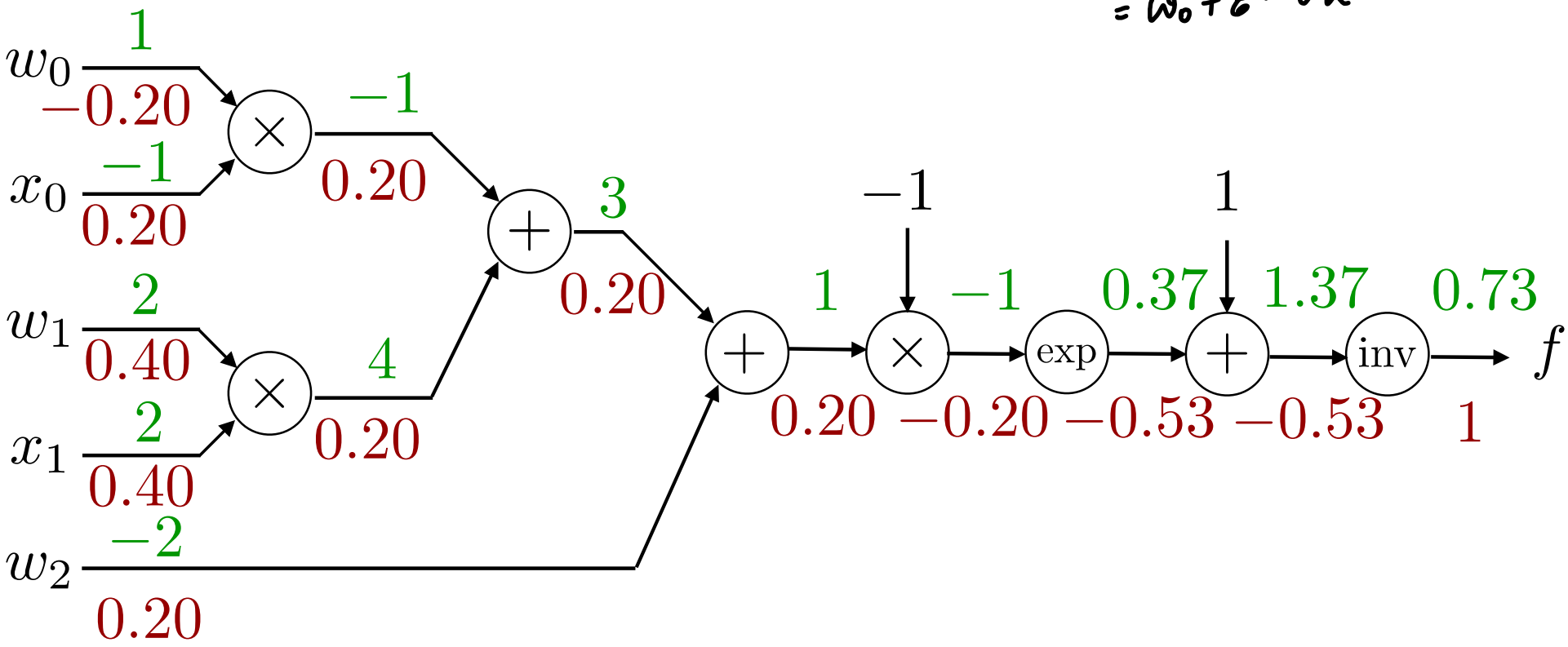


A more involved scalar example

$$f(\mathbf{w}, \mathbf{x}) = \frac{1}{1 + \exp(-(w_0 x_0 + w_1 x_1 + w_2))}$$

$w_0 \leftarrow w_0 - \varepsilon \frac{\partial L}{\partial w_0}$

$= w_0 + \varepsilon \cdot 0.2$



You can take any gradient this way

With backpropagation, as long as you can **break the computation into components where you know the local gradients, you can take the gradient of anything.**

torch.matmul(A, W)

↓

from pyTorch

forward:
    return A @ W

backward:
    return (local grad) · (upstream grad).

What happens when two gradient paths converge?

$q_1 = h(x)$
$q_2 = h(x)$

Think about this for next time.