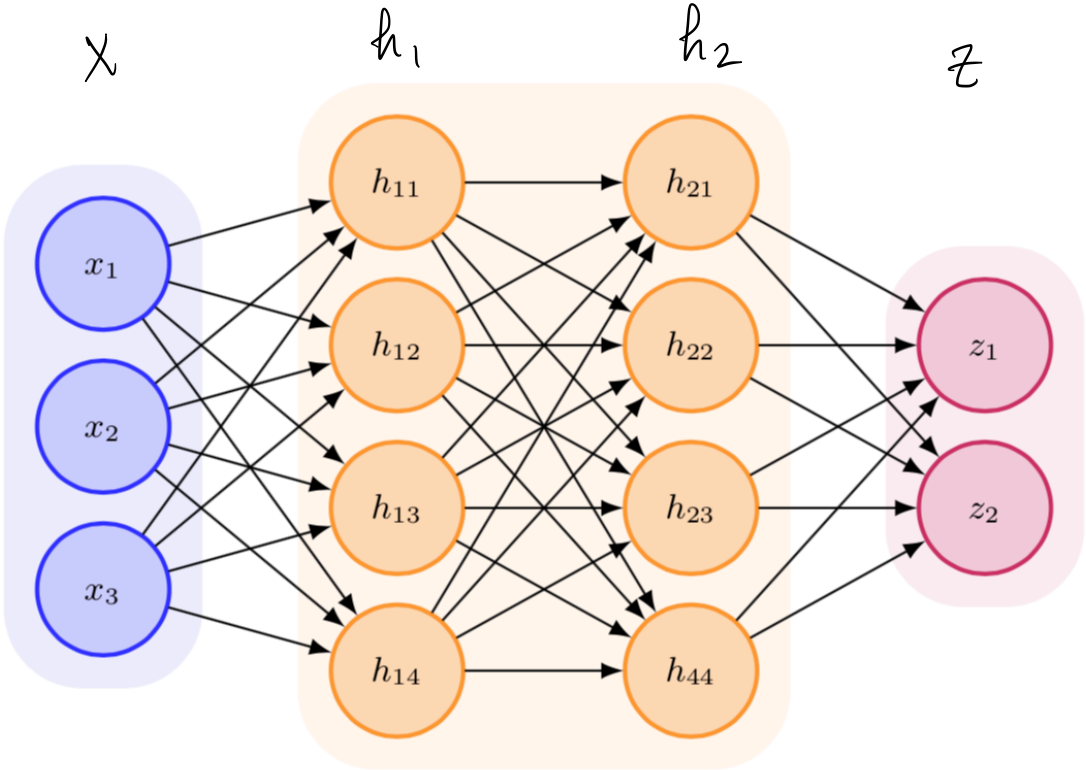
Announcements, 2018-01-29

- HW #2 due Wednesday, 31 Jan 2018, at 11:59pm.
  - You may NOT use any late days on this.
  - Be sure you submit the .py files that implement the knn, svm, and softmax classes! (This is in addition to the Jupyter notebooks.)
- HW #3 will be distributed on Wednesday, 31 Jan 2018, and will be due a week later on Wednesday, 7 Feb 2018.

Recap from 2018-01-24

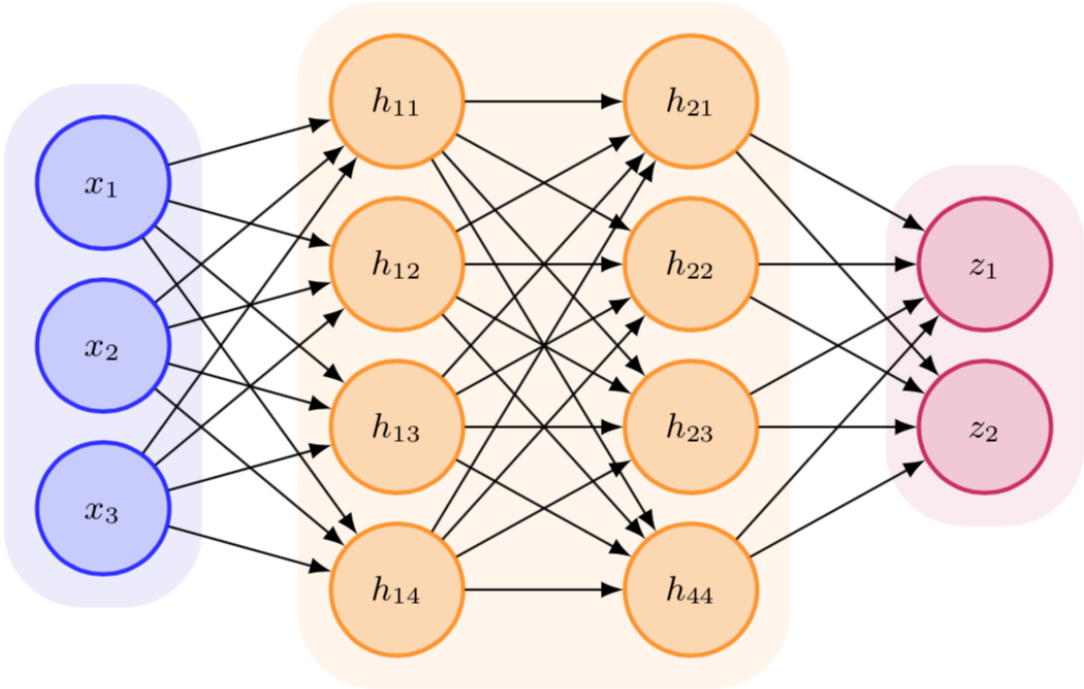


First layer: $\mathbf{h}_1 = f(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1)$

Second layer: $\mathbf{h}_2 = f(\mathbf{W}_2\mathbf{h}_1 + \mathbf{b}_2)$

Third (output layer): $\mathbf{z} = f(\mathbf{W}_3\mathbf{h}_2 + \mathbf{b}_3)$

Recap from 2018-01-24




```
# Define the activation function
f = lambda x: x * (x > 0)
# Forward pass of a 3-layer network
h1 = f(np.dot(W1, x) + b1)
h2 = f(np.dot(W2, h1) + b2)
z = f(np.dot(W3, h2) + b3)
```

Recap from 2018-01-24

Neural networks

The above figure suggests the following equation for a neural network.

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

$f(x) = x$

Any composition of linear functions can be reduced to a single linear function. Here, $\mathbf{z} = \mathbf{W}\mathbf{x} + \mathbf{b}$, where

$$\mathbf{W} = \mathbf{W}_N \cdots \mathbf{W}_2\mathbf{W}_1$$

and

$$\mathbf{b} = \mathbf{b}_N + \mathbf{W}_N\mathbf{b}_{N-1} + \cdots + \mathbf{W}_N \cdots \mathbf{W}_3\mathbf{b}_2 + \mathbf{W}_N \cdots \mathbf{W}_2\mathbf{b}_1$$

Recap from 2018-01-24

Introducing nonlinearity

To increase the network capacity, we can make it nonlinear. We do this by introducing a nonlinearity, $f(\cdot)$, at the output of each artificial neuron.

- 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: $\mathbf{z} = \mathbf{W}_N\mathbf{h}_{N-1} + \mathbf{b}_N$

$f(x) = \max(0, x)$

A few notes:

- These equations describe a *feedforward neural network*, also called a *multilayer perceptron*.
- $f(\cdot)$ is typically called an *activation function* and is applied elementwise on its input.
- The activation function does not typically act on the output layer, $\mathbf{z}$, as these are meant to be interpreted as scores. Instead, separate "output activations" are used to process $\mathbf{z}$. While these output activations may be the same as the activation function, they are typically different. For example, it may comprise a softmax or SVM classifier.

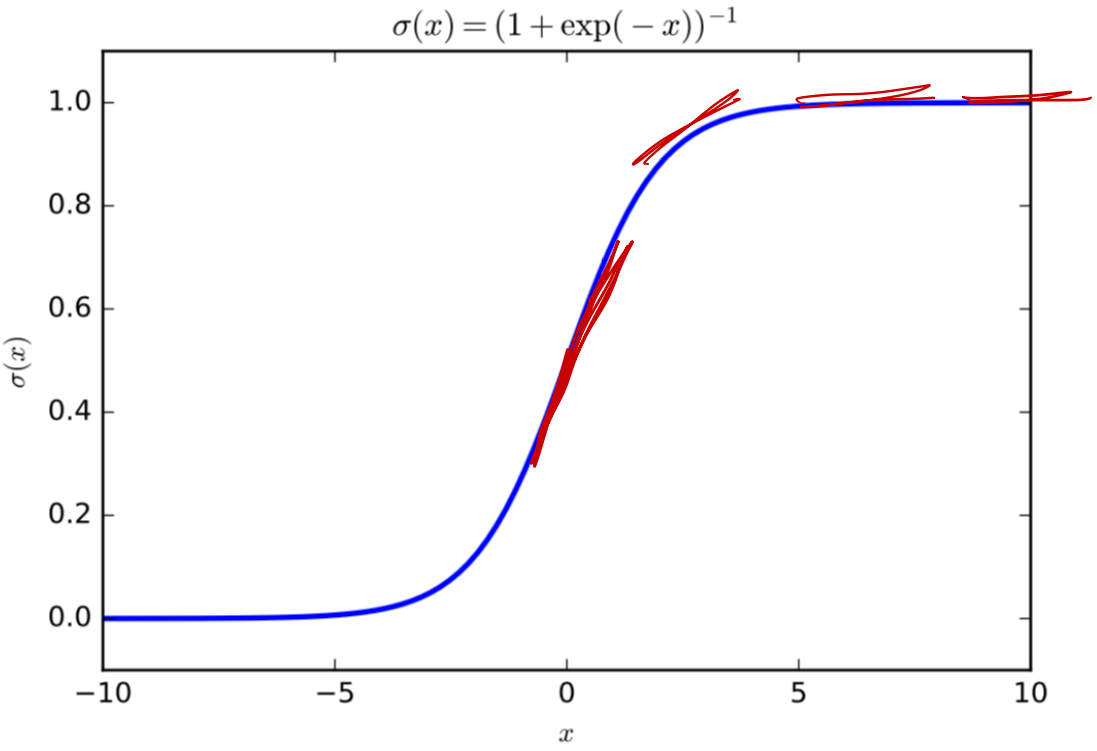
Recap from 2018-01-24

There are a variety of activation functions. We'll discuss some more commonly encountered ones.

Recap from 2018-01-24

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

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



Its derivative is:

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

$0.5\,(1-0.5) = 0.25$

Recap from 2018-01-24

"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)

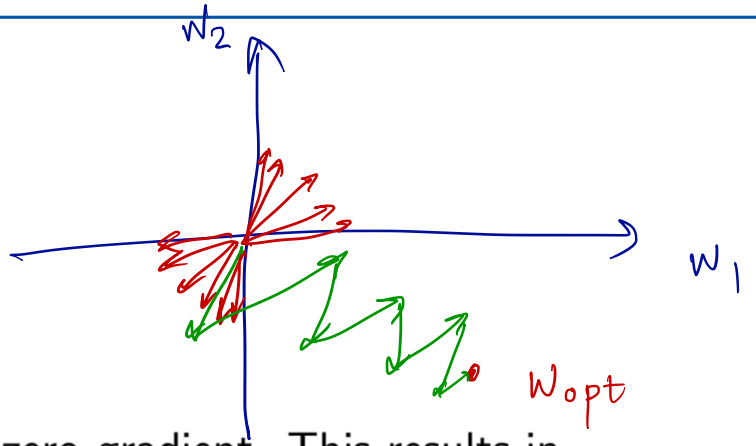
$L(w)$ when $|w|$ is large

$$\frac{\partial L}{\partial \sigma(w)} \frac{\partial \sigma(w)}{\partial w} = \frac{\partial L}{\partial \sigma(w)} \cdot \text{small number}$$

$$0.25 \frac{\partial L}{\partial \sigma(w)}$$

Sigmoid unit

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 not zero-centered; rather its outputs are centered at 0.5.

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

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

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.

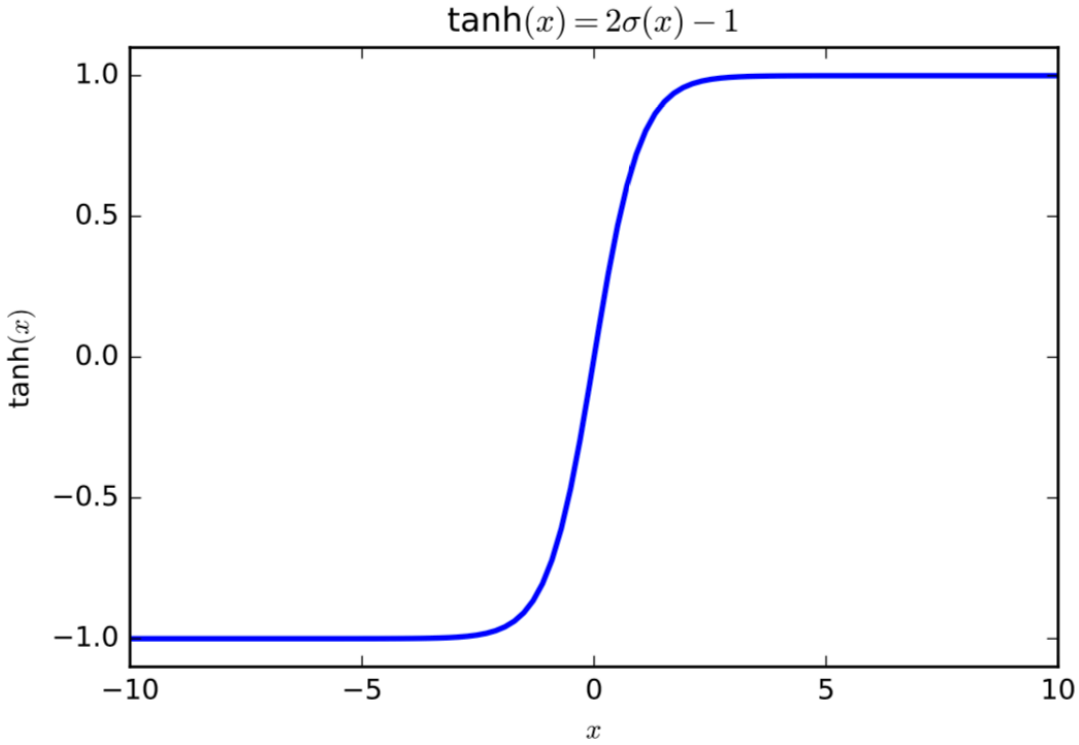
This can result in zig-zagging during gradient descent.

Hyperbolic tangent

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

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

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



Its derivative is:

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

Hyperbolic tangent

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

Pros:

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

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.

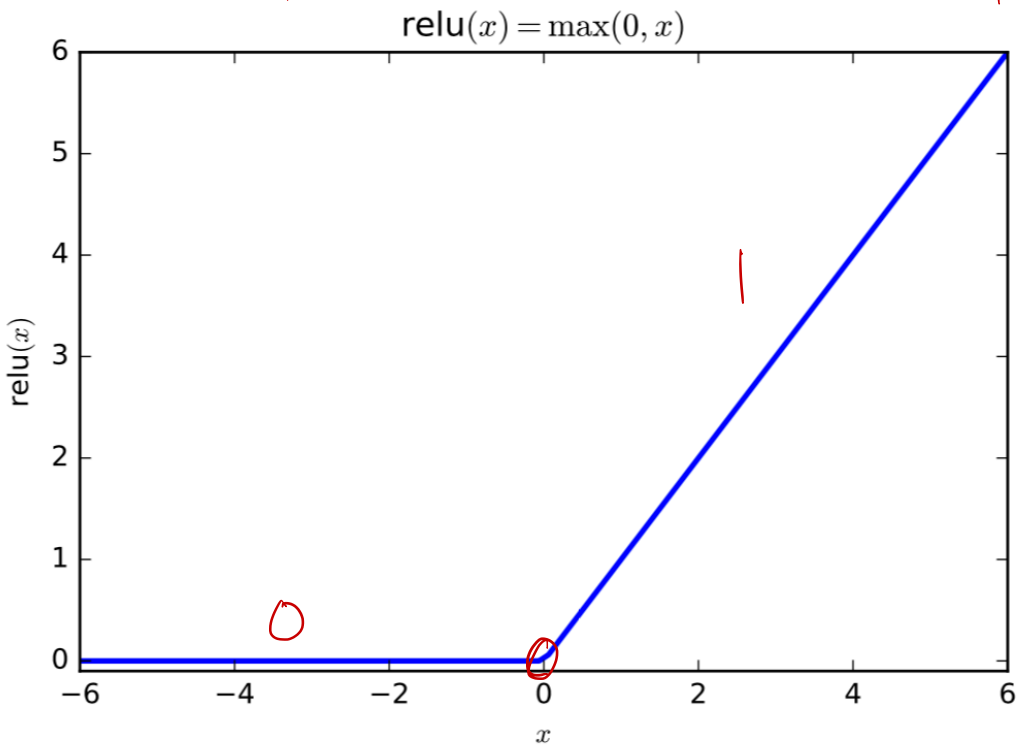
ReLU unit

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

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

np.maximum(0, x)

np.max(0, x)



$\frac{\partial L}{\partial \text{relu}(w)} \frac{\partial \text{relu}(w)}{\partial w}$

Its derivative is:

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

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.
- 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 not zero-centered.
- $\mathrm{ReLU}(x)$ 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.

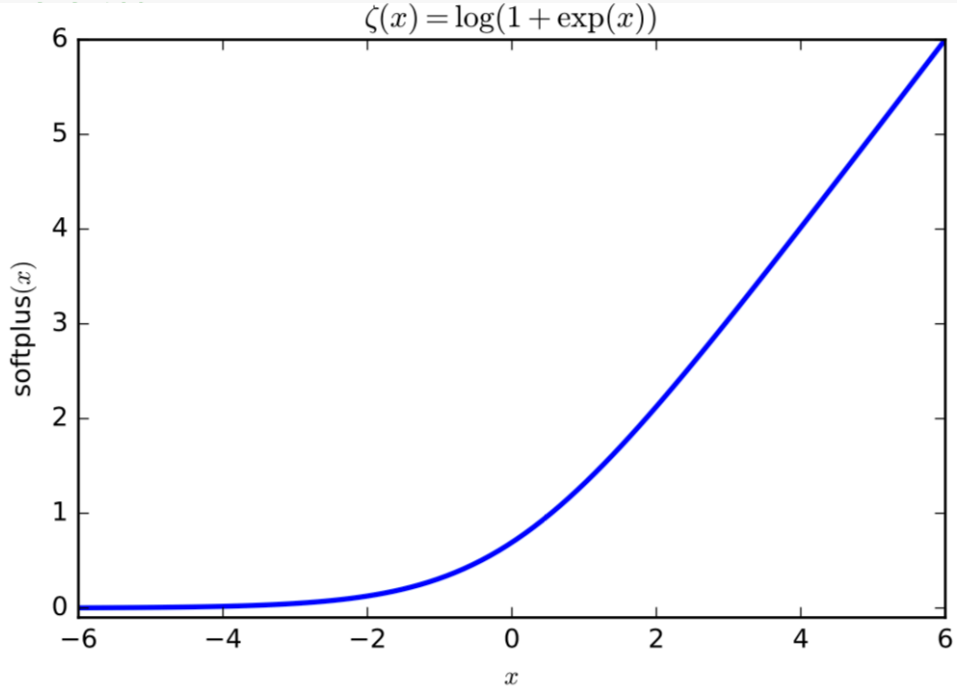
Softplus unit

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

x is large

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%

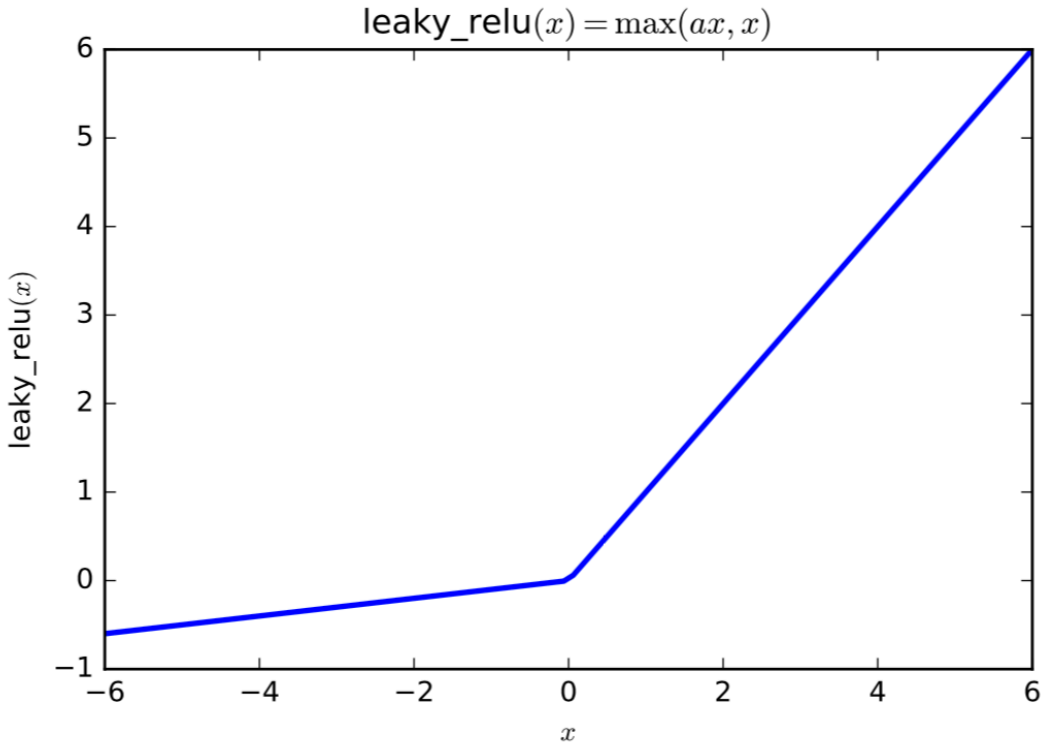
Glorot et al., 2011a

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)
```

$\alpha = 0.01$

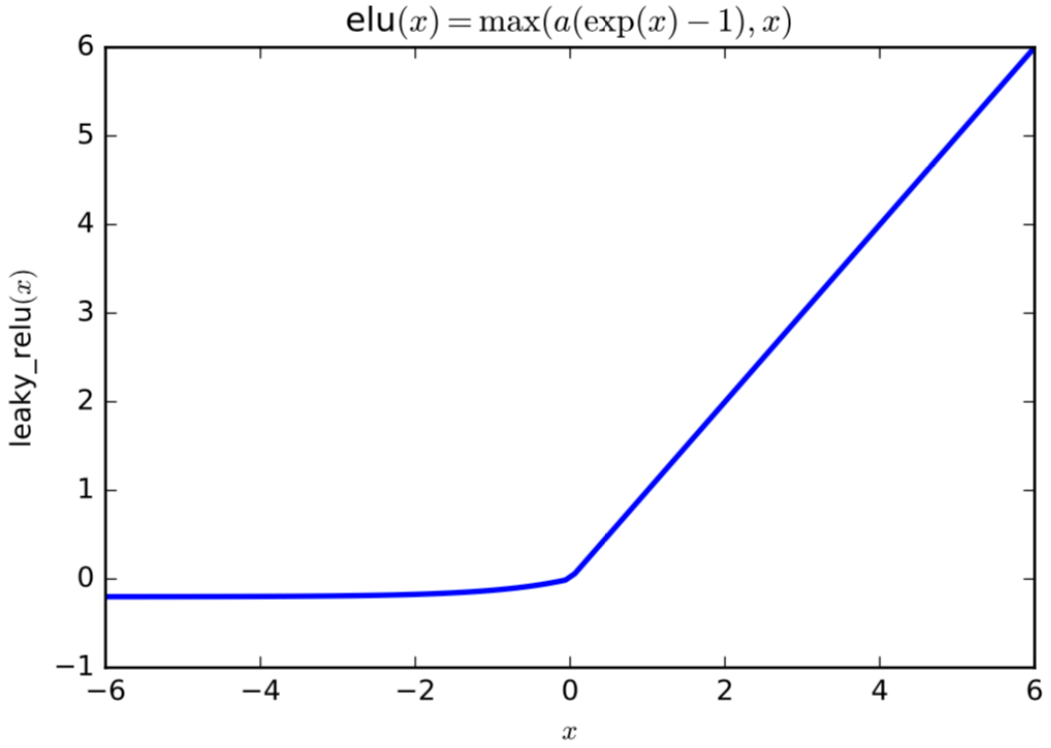


The leaky ReLU avoids the stopping of learning when $x < 0$. α may be treated as a selected hyperparameter, or it may be a parameter to be optimized in learning, in which case it is typically called the "PReLU" for parametrized rectified linear unit.

ELU unit

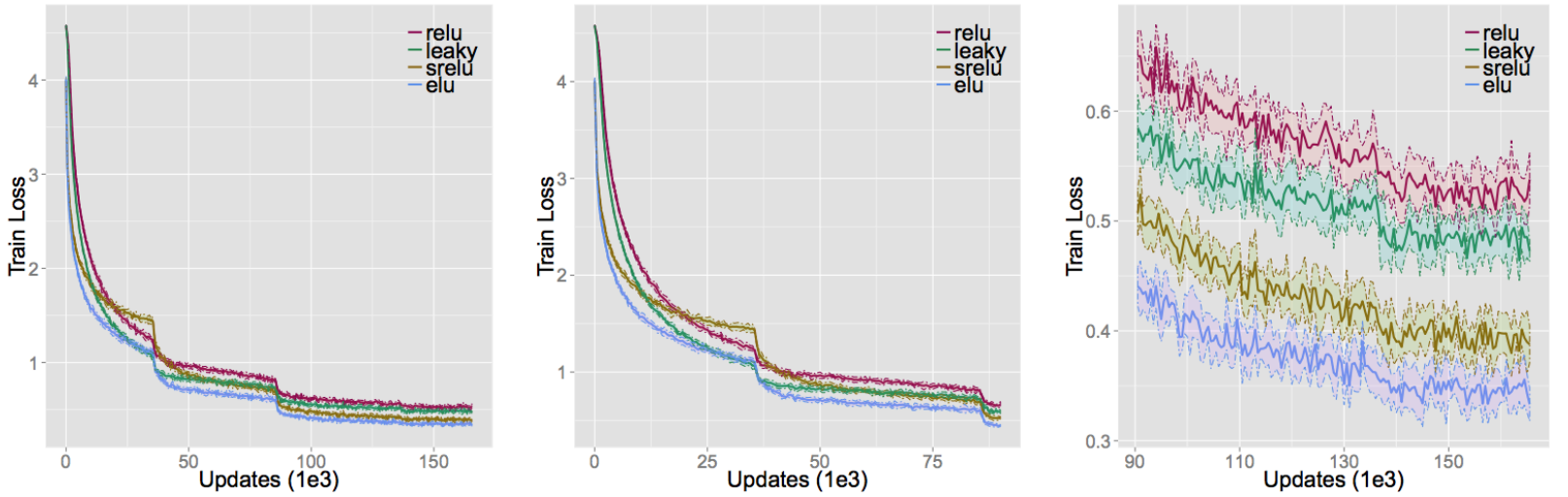
Exponential linear unit, $f(x) = \max(\alpha(\exp(x) - 1), x)$

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

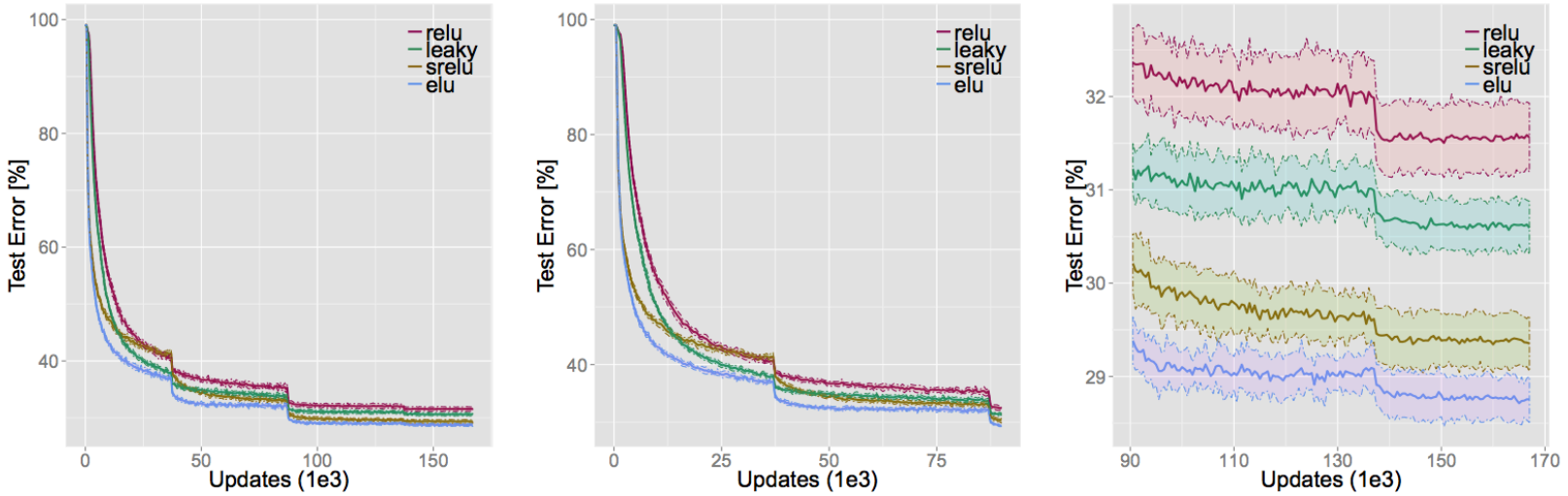


The exponential linear unit avoids the stopping of learning when $x < 0$. A con of this activation function is that it requires computation of the exponential, which is more expensive.

Which LU do I use?



(a) Training loss (b) Training loss (start) (c) Training loss (end)



(d) Test error (e) Test error (start) (f) Test error (end)

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

Clevert et al., 2015

Maxout unit

A generalization of the ReLU and PReLU units is the maxout unit, where:

$$\text{maxout}(\mathbf{x}) = \max(\mathbf{w}_1^T \mathbf{x} + b_1, \mathbf{w}_2^T \mathbf{x} + b_2)$$

This can be generalized to more than two components. If $\mathbf{w}_1 = \mathbf{0}$ and $b_1 = 0$, this is the rectified linear unit.

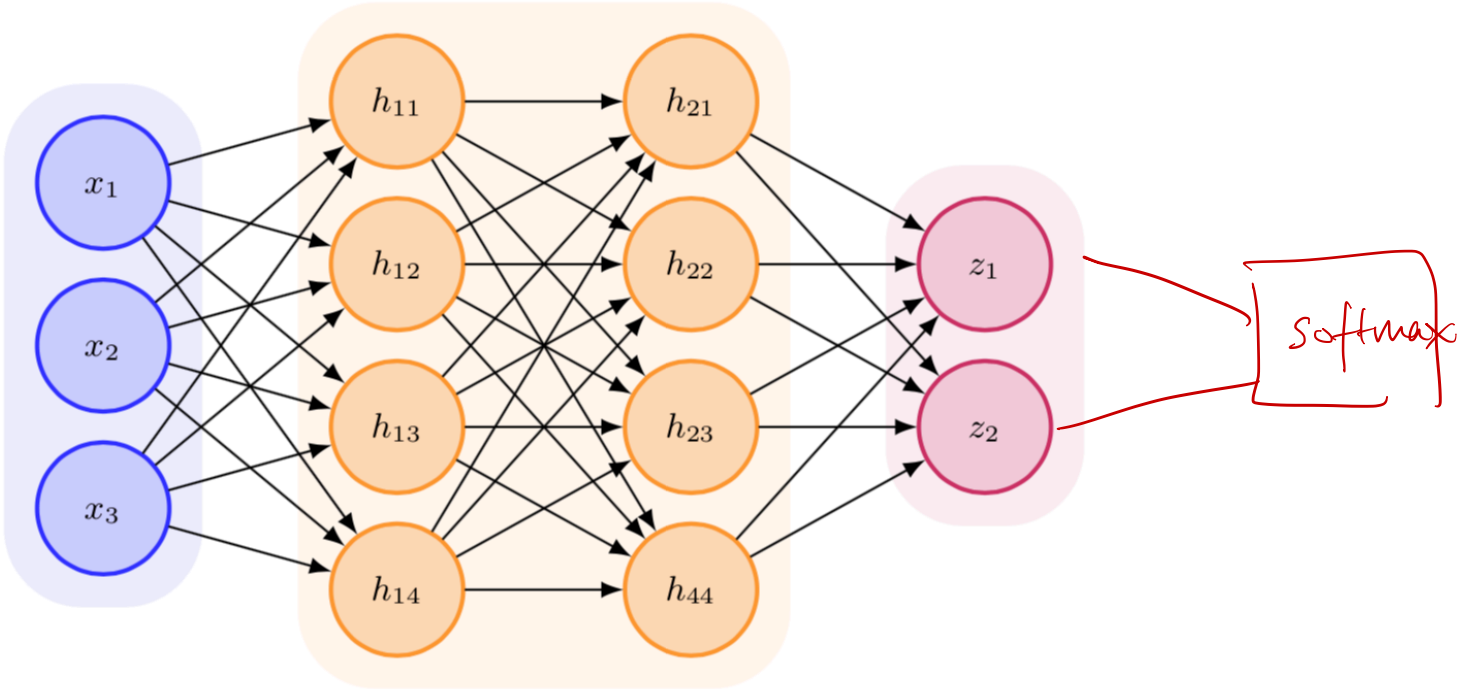
What activation function do I use?

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.
- This is an active area of research.

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 $z \rightarrow$ class 1 ; $y^{(i)} = 1$

negative $z \rightarrow$ class 2 ; $y^{(i)} = 0$

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.

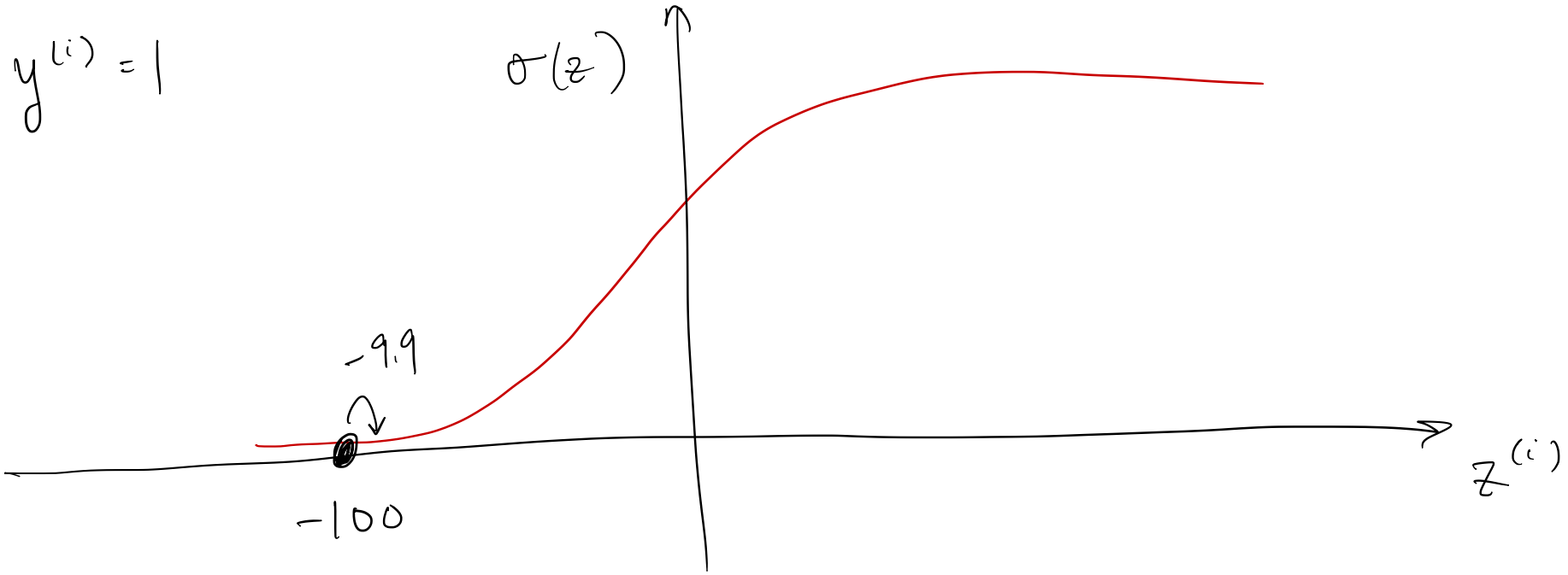
$\sigma(z) \rightarrow \Pr\left(x^{(i)} \text{ is in class } 1\right)$

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



$$MSE_i = \left(y^{(i)} - \sigma\left(z^{(i)}\right)\right)^2$$

$$= 1^2 = 1$$

The output activation interacts with the cost function

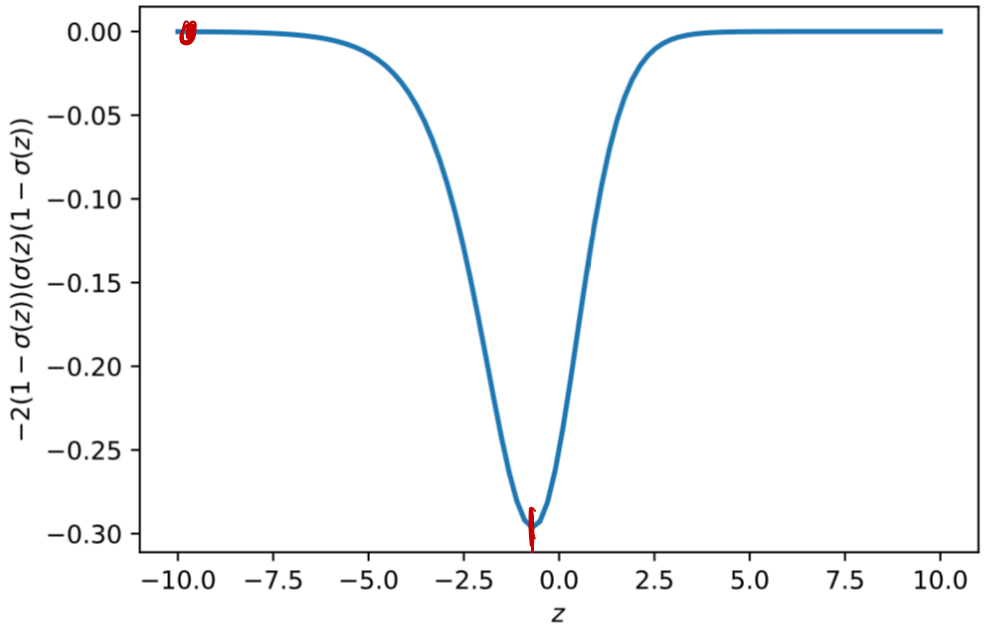
What outputs and cost functions? (cont.)

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:

$$z_i \leftarrow z_i - \epsilon \frac{\partial \text{MSE}}{\partial z_i}$$



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

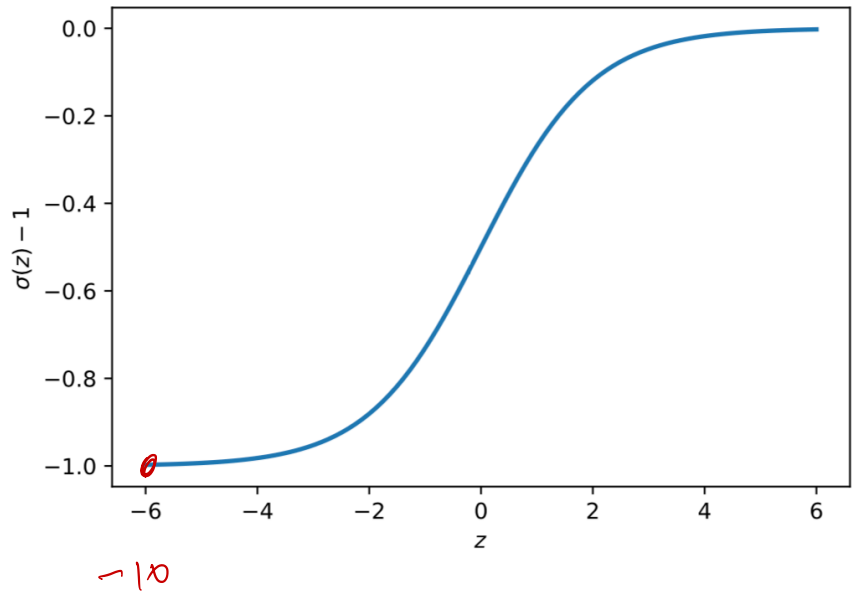
The output activation interacts with the cost function

What outputs and cost functions? (cont.)

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

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

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

- Linear output units: $\hat{\mathbf{y}} = \mathbf{z}$.

  These output units typically specify the conditional mean of a Gaussian distribution, i.e.,

  $$p(\mathbf{y}|\mathbf{z}) = \mathcal{N}(\mathbf{z}, \mathbf{I})$$

  and in this case, MLE estimation is equivalent to minimizing squared error.

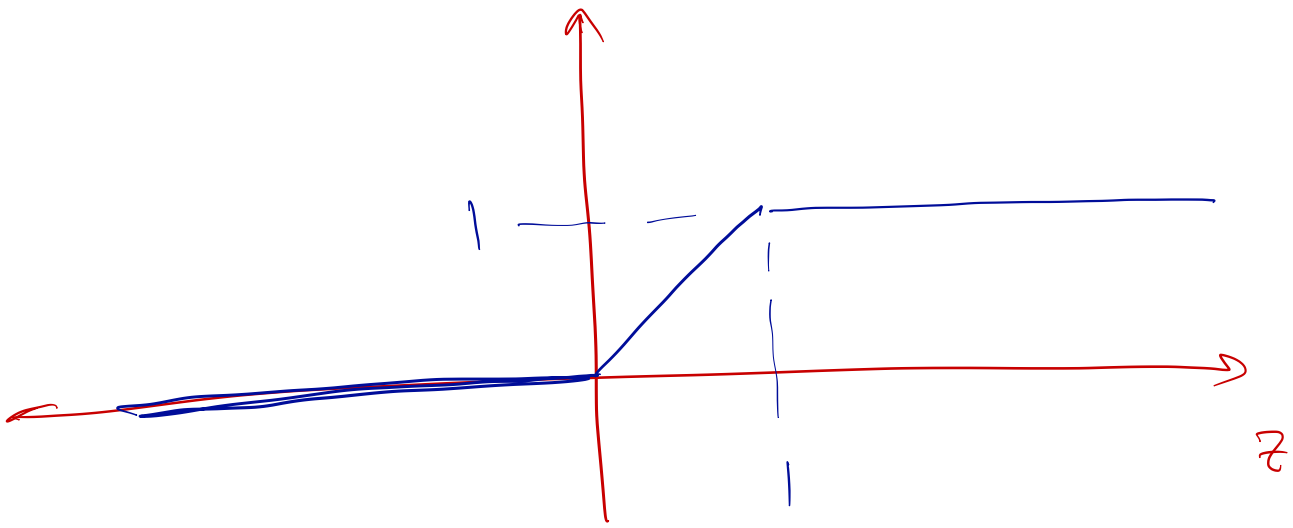
Output activations

- Sigmoid outputs: $\hat{\mathbf{y}} = \sigma(\mathbf{z})$.

  These outputs are typically used in binary classification to approximate a Bernoulli distribution.

  Question: Why not use the following output?

$$\Pr(y = 1|\mathbf{x}) = \max(0, \min(1, \mathbf{z}))$$



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.

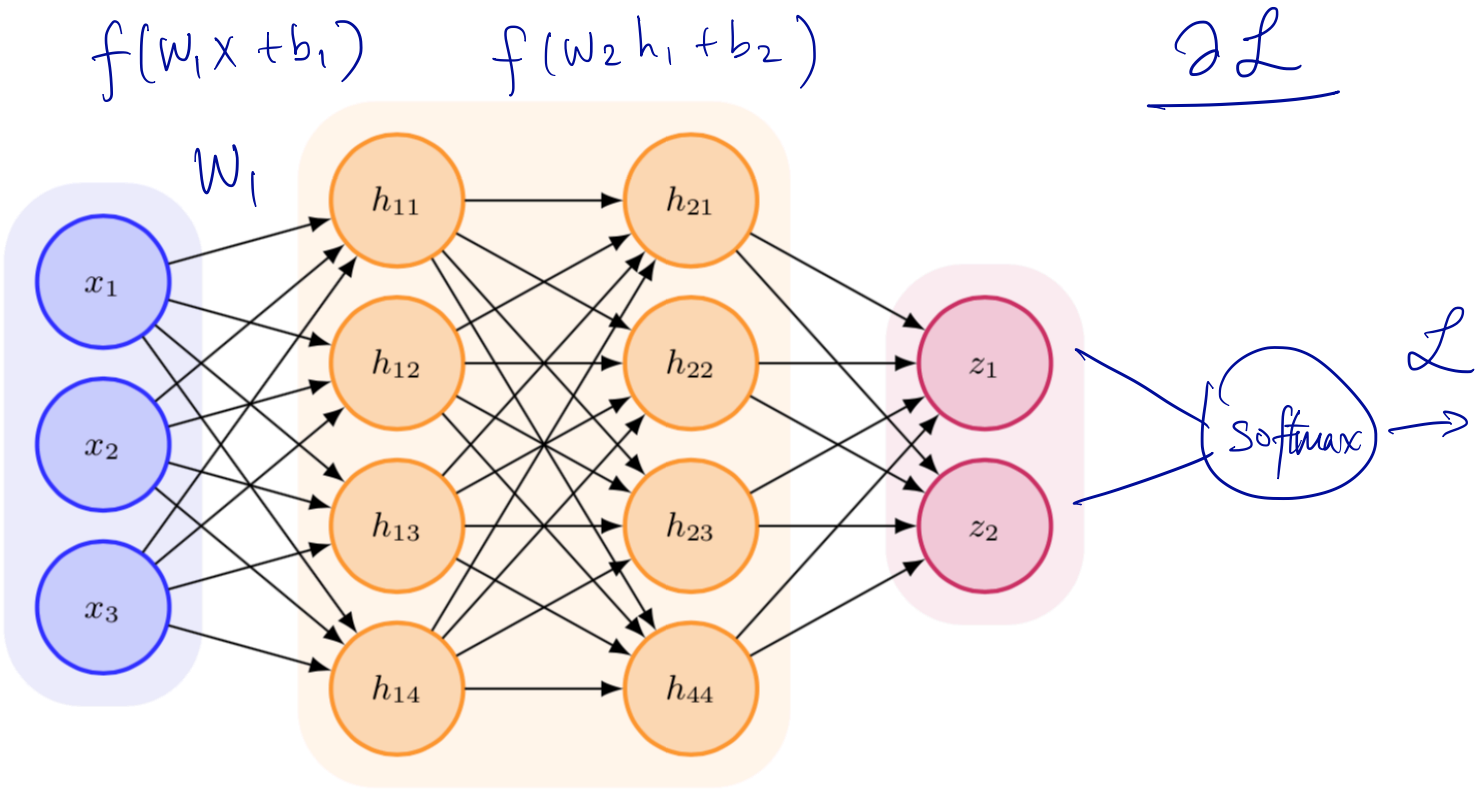
A softmax output activation is fairly common.

What next?

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

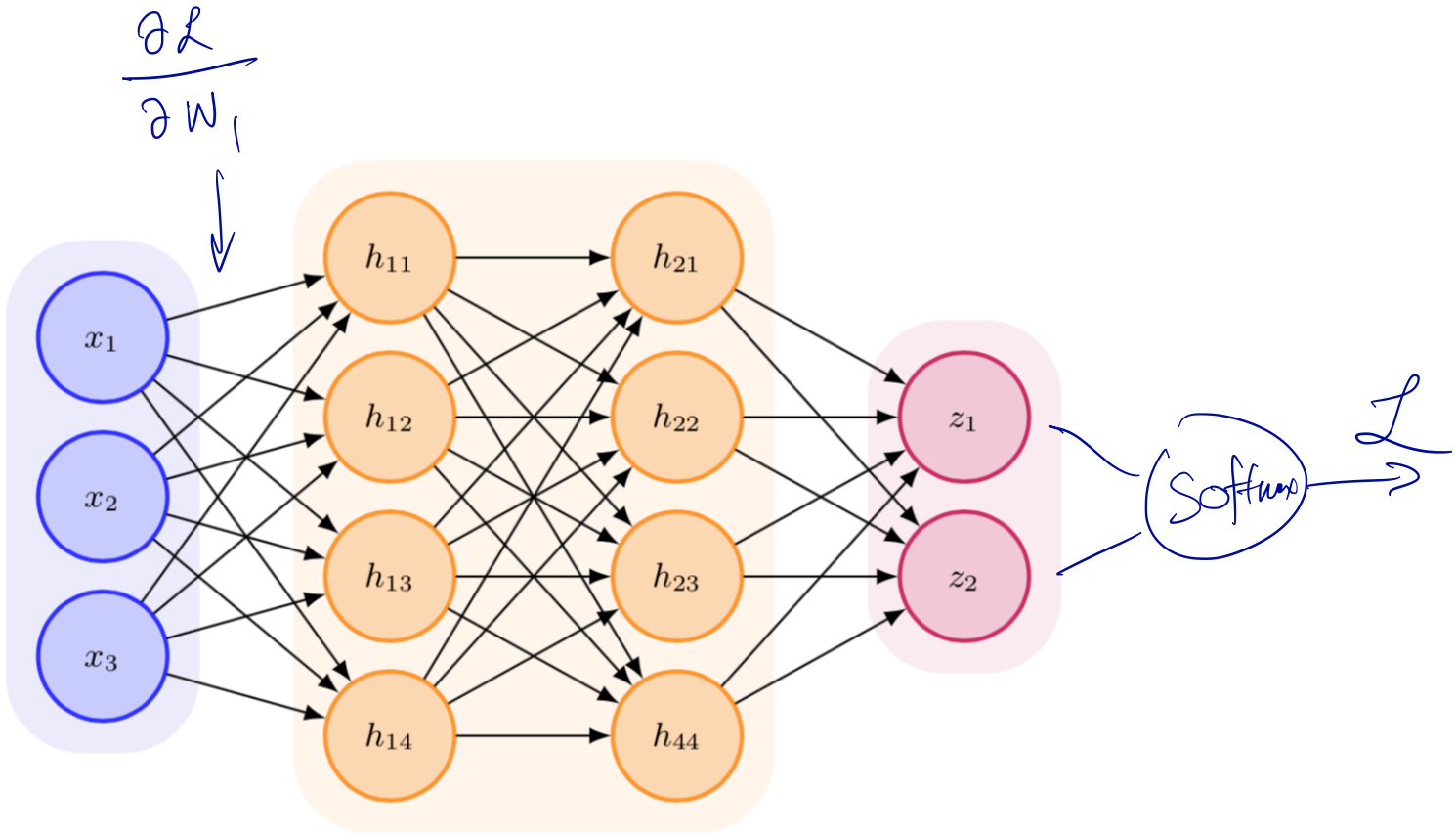
The short answer is that we use versions of gradient descent.

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



Lecture 5: Backpropagation

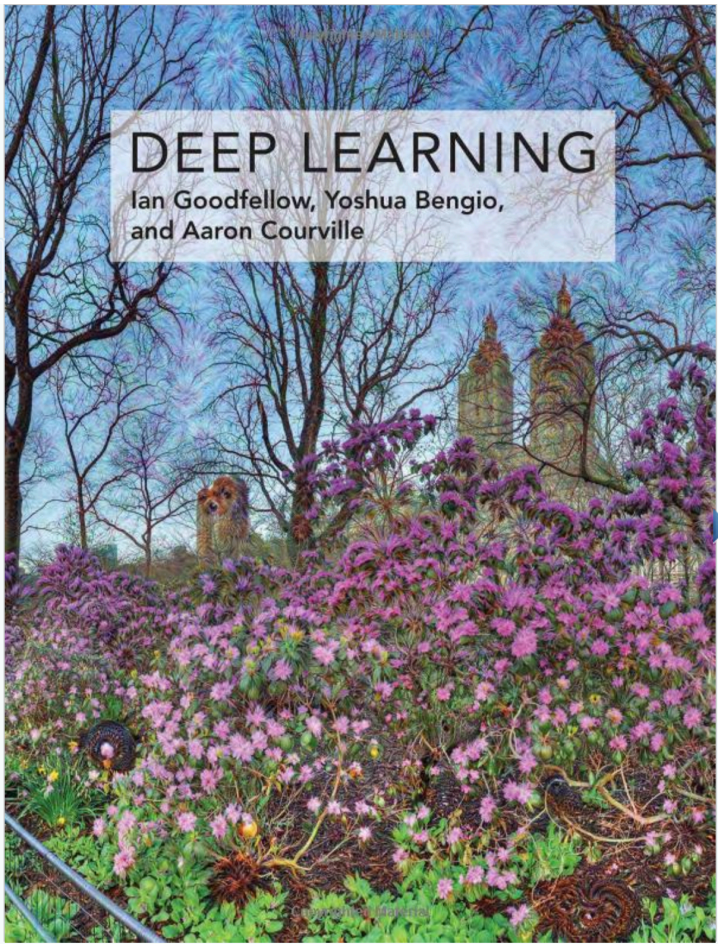
In this lecture, we'll introduce backpropagation as a technique to calculate the gradient of the loss function with respect to parameters in a neural network.



Announcements, 2018-01-29

Reading:

Deep Learning, 6.5-6.6



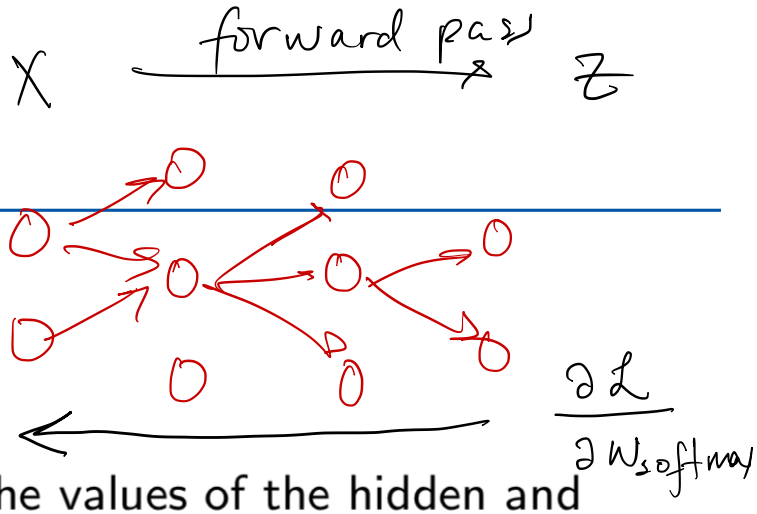
Backpropagation

In the most intuitive sense, backpropagation is an 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 pass

- 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?

It is possible in many cases to calculate an analytical gradient. So why use backpropagation?

- Evaluating analytical gradients may be computationally expensive.
- Backpropagation is generalizable and is often inexpensive.

A few further notes on backpropagation.

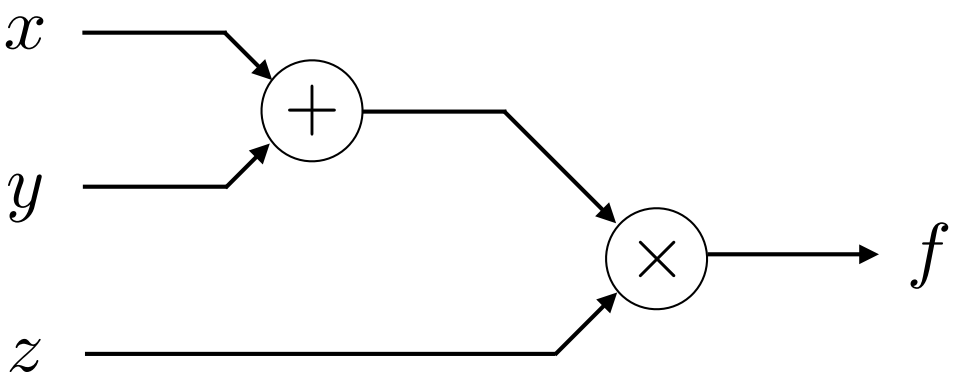
- Backpropagation is not the learning algorithm. It's the method of computing gradients.
- Backpropagation is not specific to multilayer neural networks, but a general way to compute derivatives of functions.

A simple example

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

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

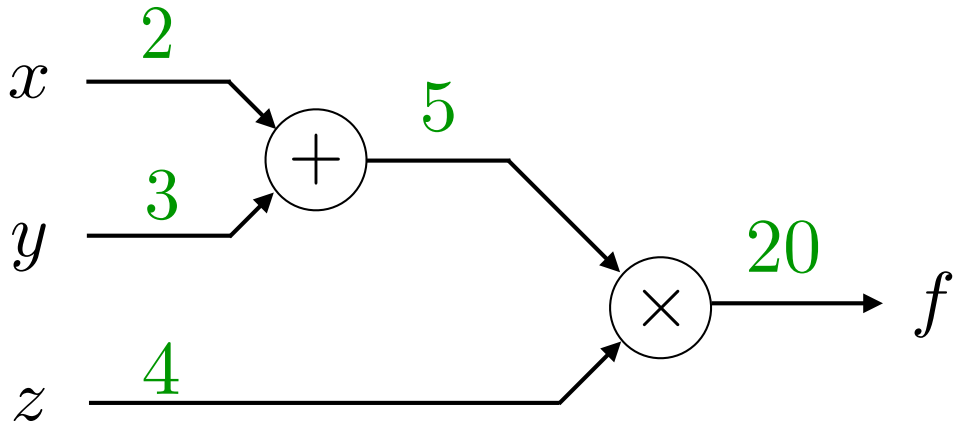
$$\frac{\partial f}{\partial x} = z \qquad \frac{\partial f}{\partial y} = z$$



A simple example

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

Forward propagation:



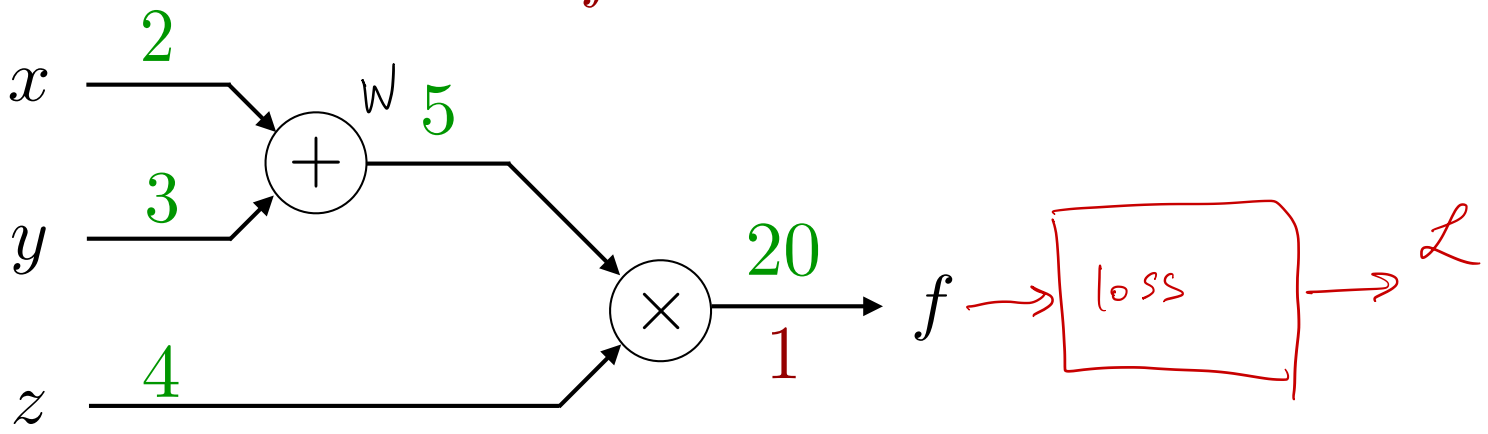
A simple example

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

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

Backpropagation:

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



$$f = wz$$

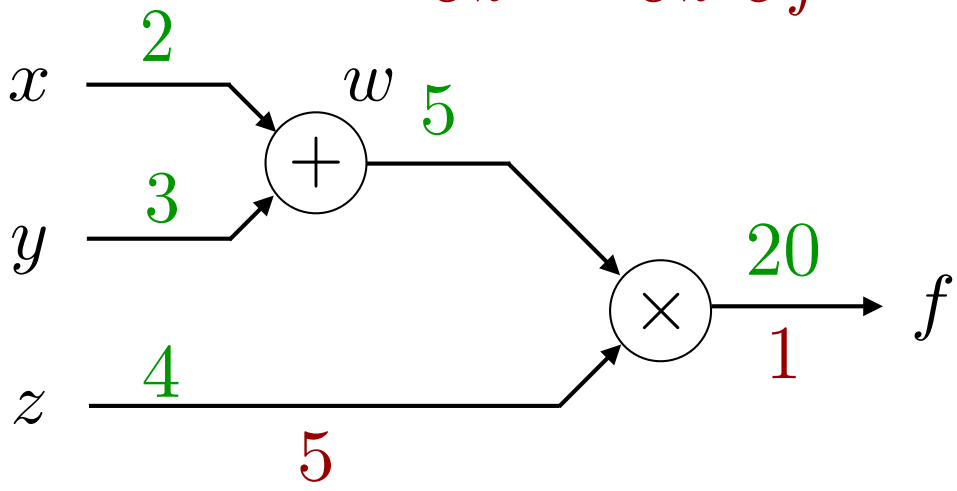
$$\frac{\partial f}{\partial z} = w$$

A simple example

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

Backpropagation:

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



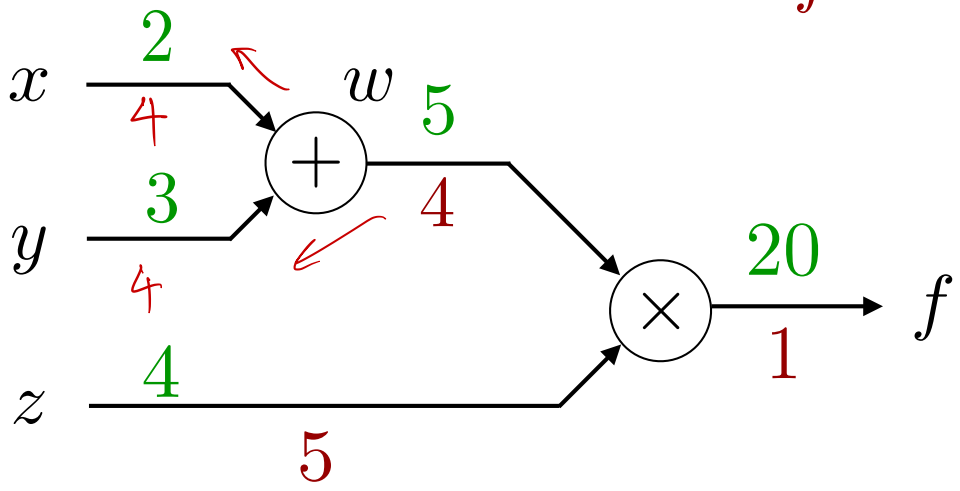
A simple example

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

Backpropagation:

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

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

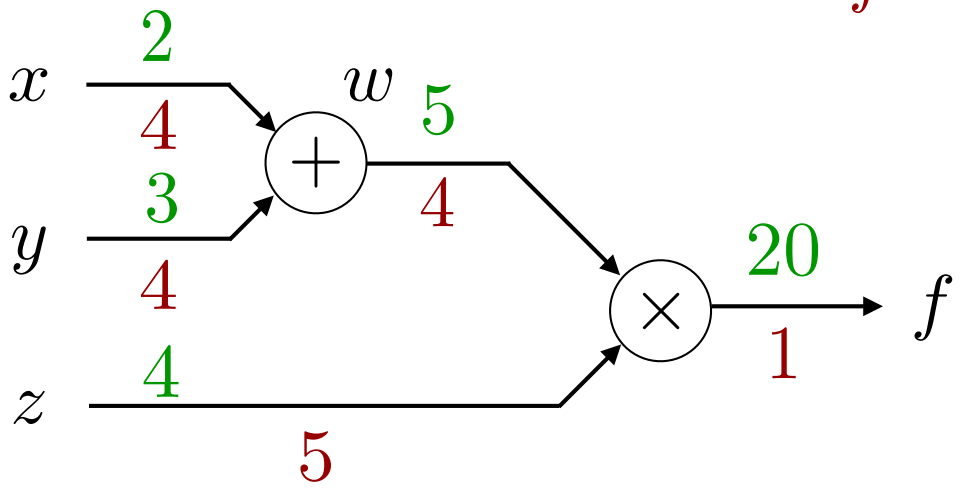


A simple example

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

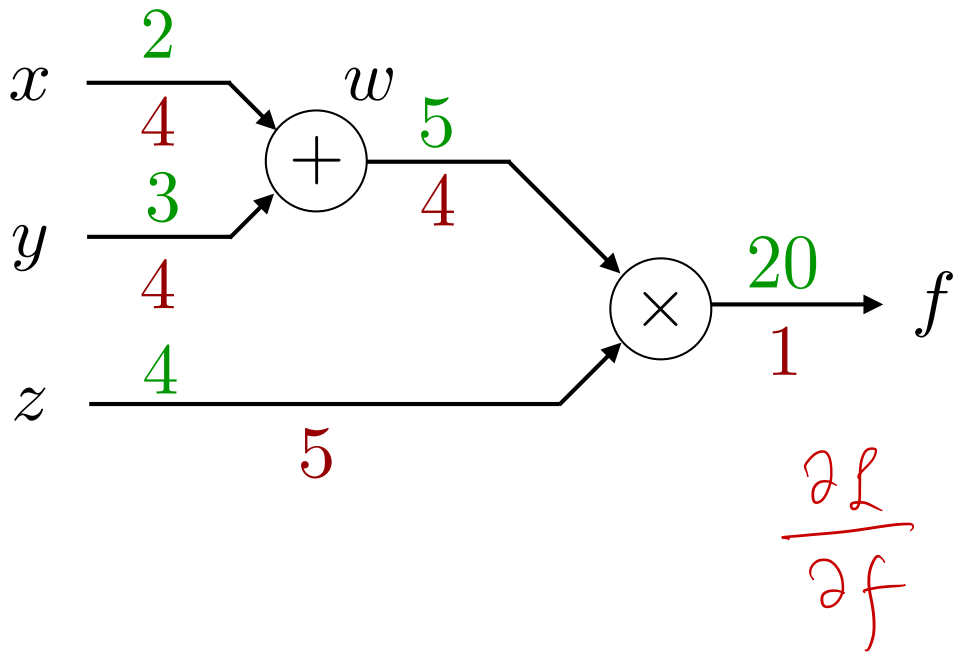
Backpropagation:

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



Idea: computational graphs apply to gradients

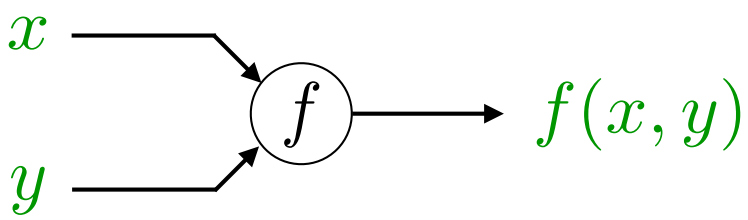
In the **forward pass**, we apply a function to the node inputs to calculate an output.
In the **backward pass**, we take the **upstream derivative** and apply a **local gradient** to calculate the backpropagated derivative.



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.

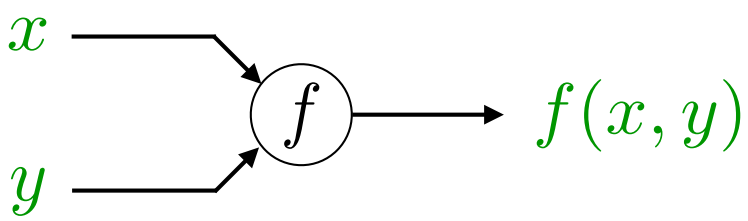
Forward pass:



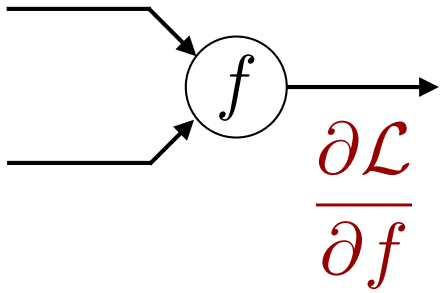
Idea: computational graphs apply to gradients

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.

Forward pass:



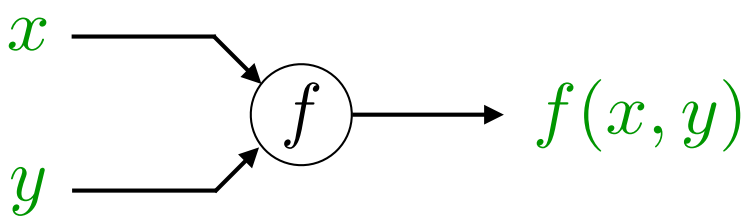
Backward pass:



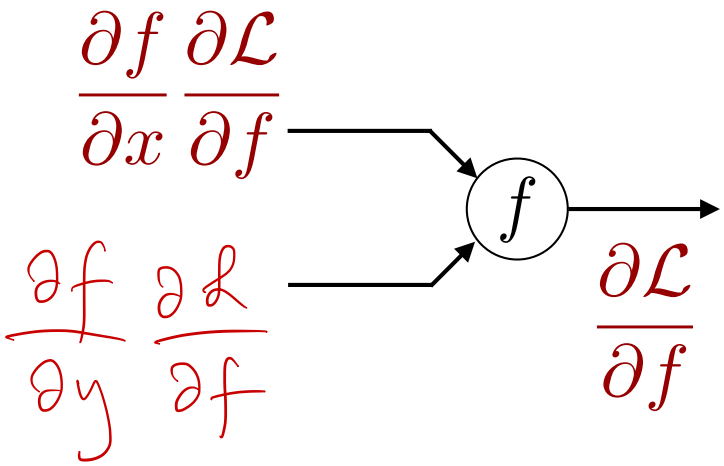
Idea: computational graphs apply to gradients

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.

Forward pass:



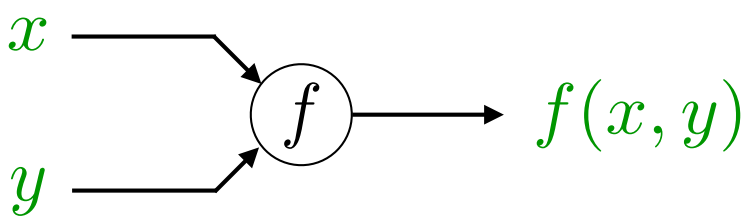
Backward pass:



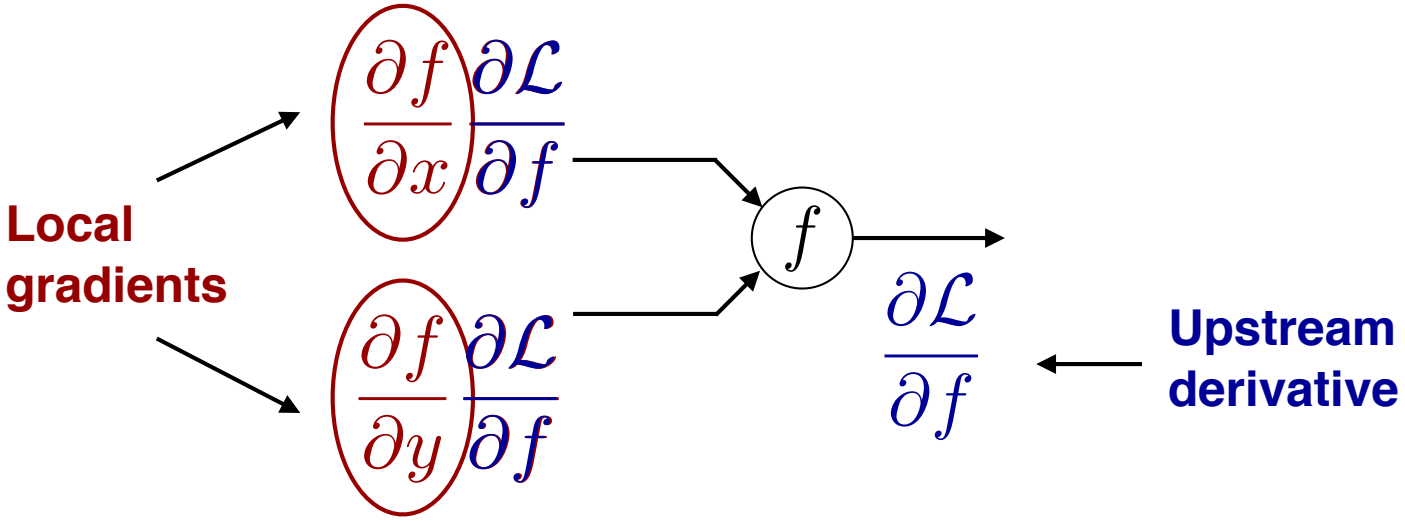
Idea: computational graphs apply to gradients

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.

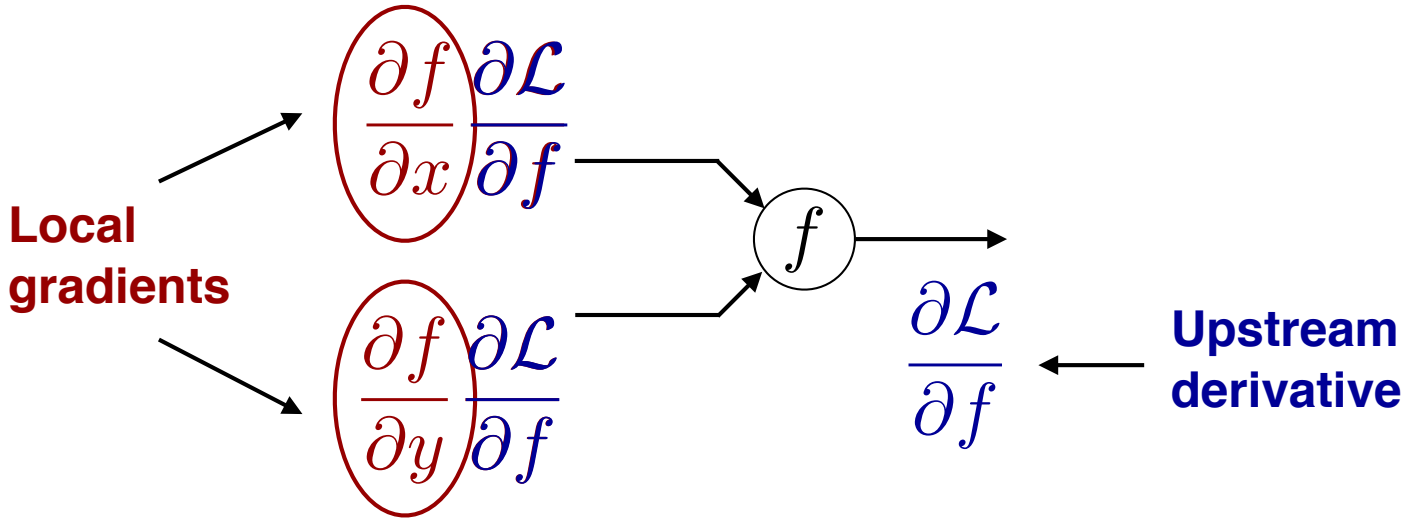
Forward pass:



Backward pass:



Idea: computational graphs apply to gradients



The basic intuition of backpropagation is that we breakup 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).

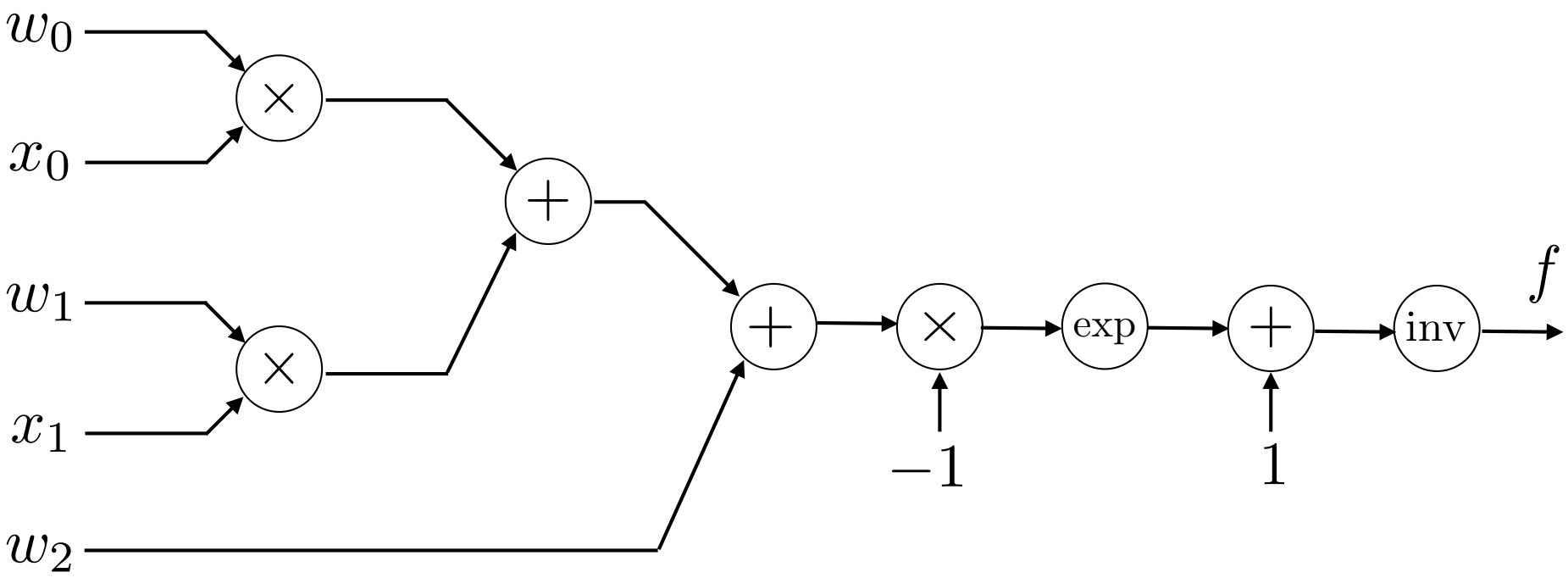
Composing all of these gradients together returns the overall gradient.

A more involved scalar example

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

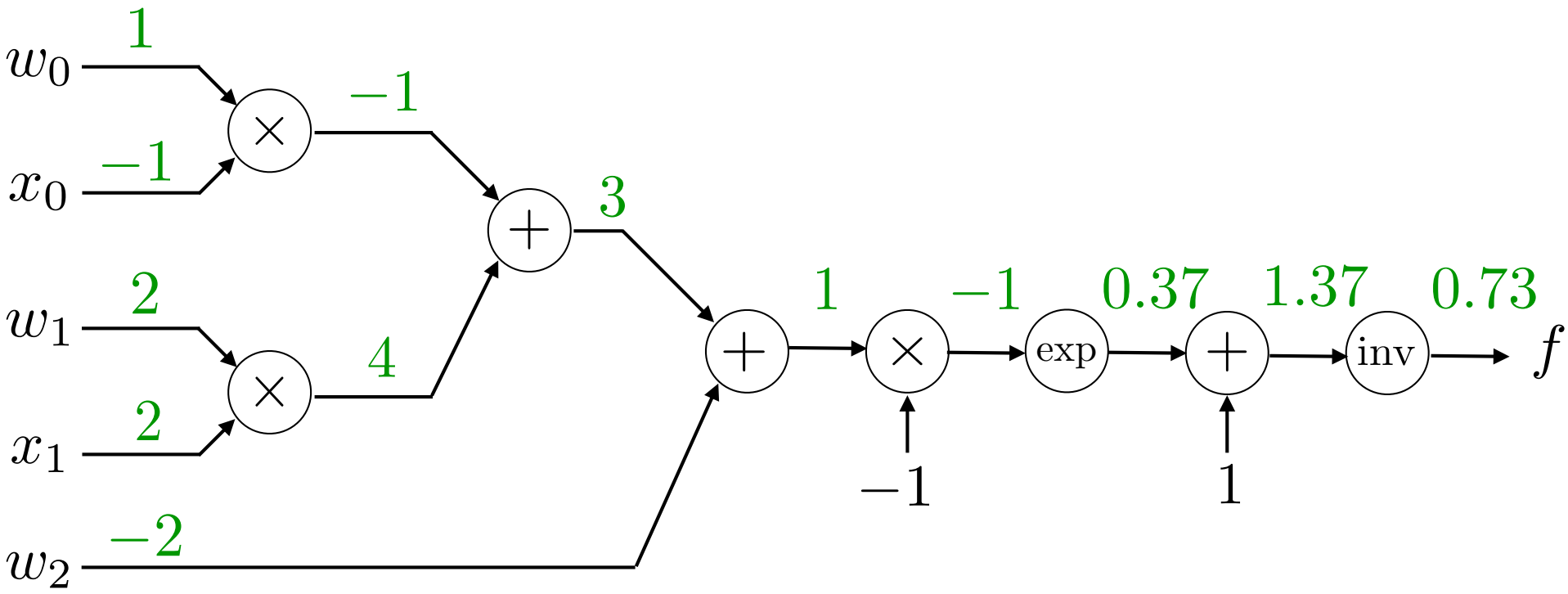
A more involved scalar example

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



A more involved scalar example

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



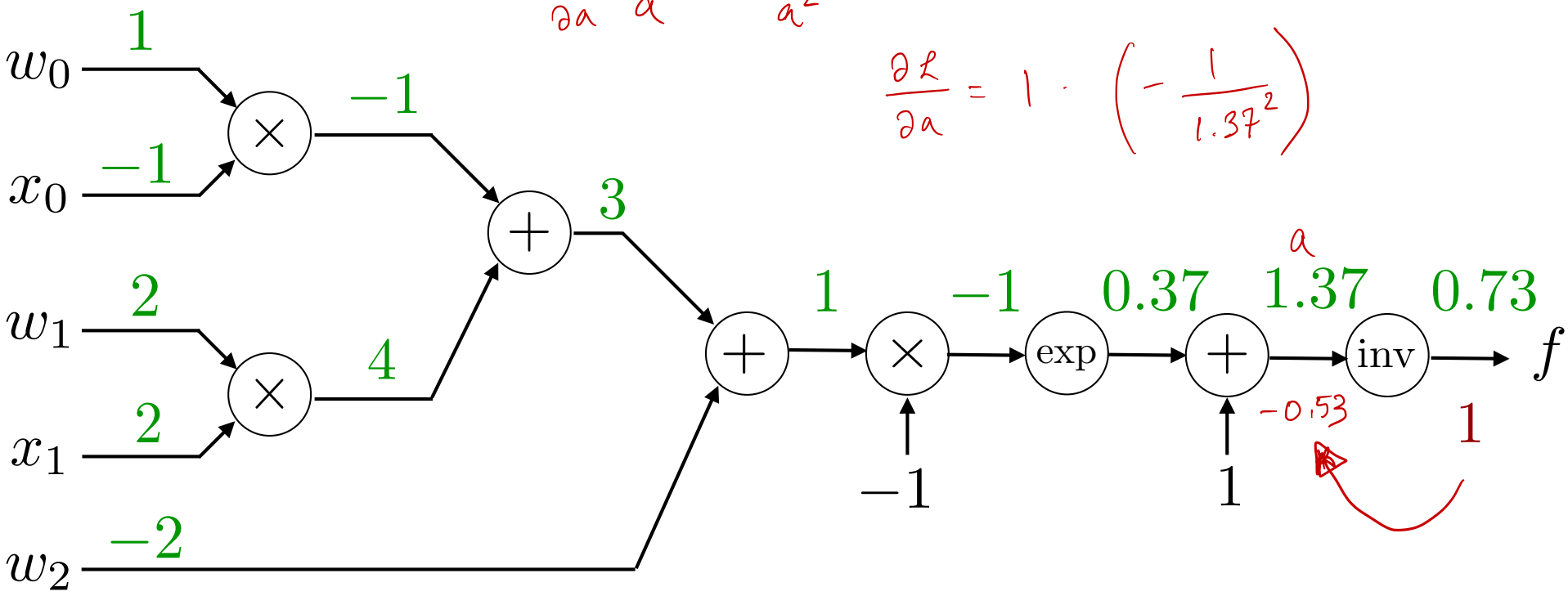
A more involved scalar example

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

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

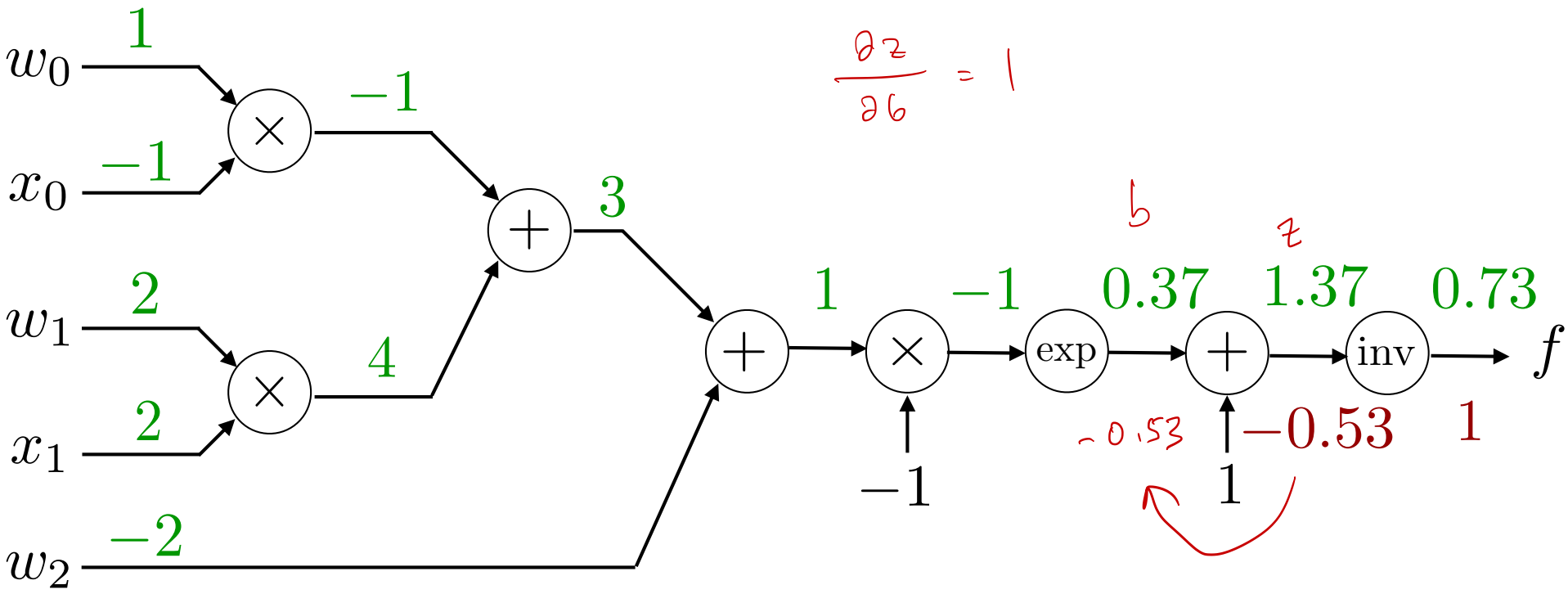
$\frac{\partial}{\partial a} \frac{1}{a} = -\frac{1}{a^2}$

$\frac{\partial f}{\partial a} = 1 \cdot \left(-\frac{1}{1.37^2}\right)$



A more involved scalar example

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

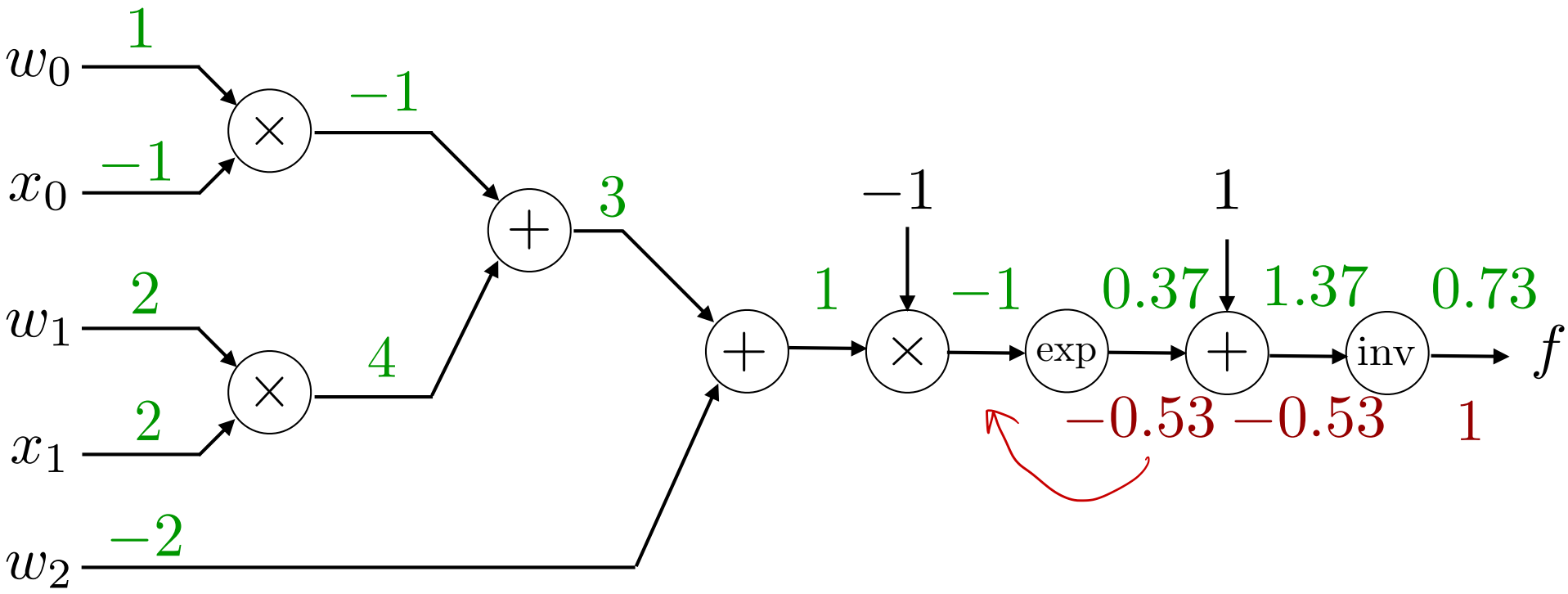


$$\frac{\partial \mathcal{L}}{\partial z} = \frac{\partial f}{\partial z}\frac{\partial \mathcal{L}}{\partial f} \qquad \frac{\partial f}{\partial z} = -\frac{1}{z^2}$$

A more involved scalar example

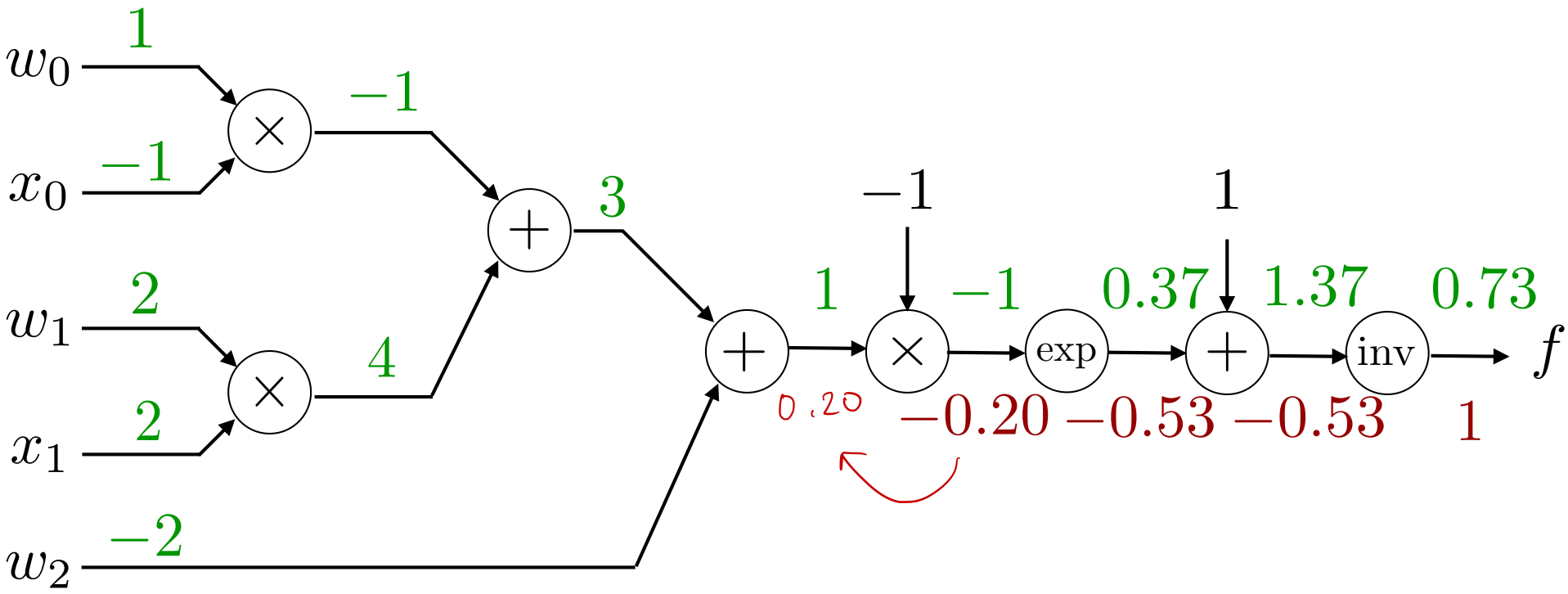
$$f(\mathbf{w},\mathbf{x}) = \frac{1}{1+\exp(-(w_0x_0+w_1x_1+w_2))}$$

$-0.53 \cdot \exp(-1)$



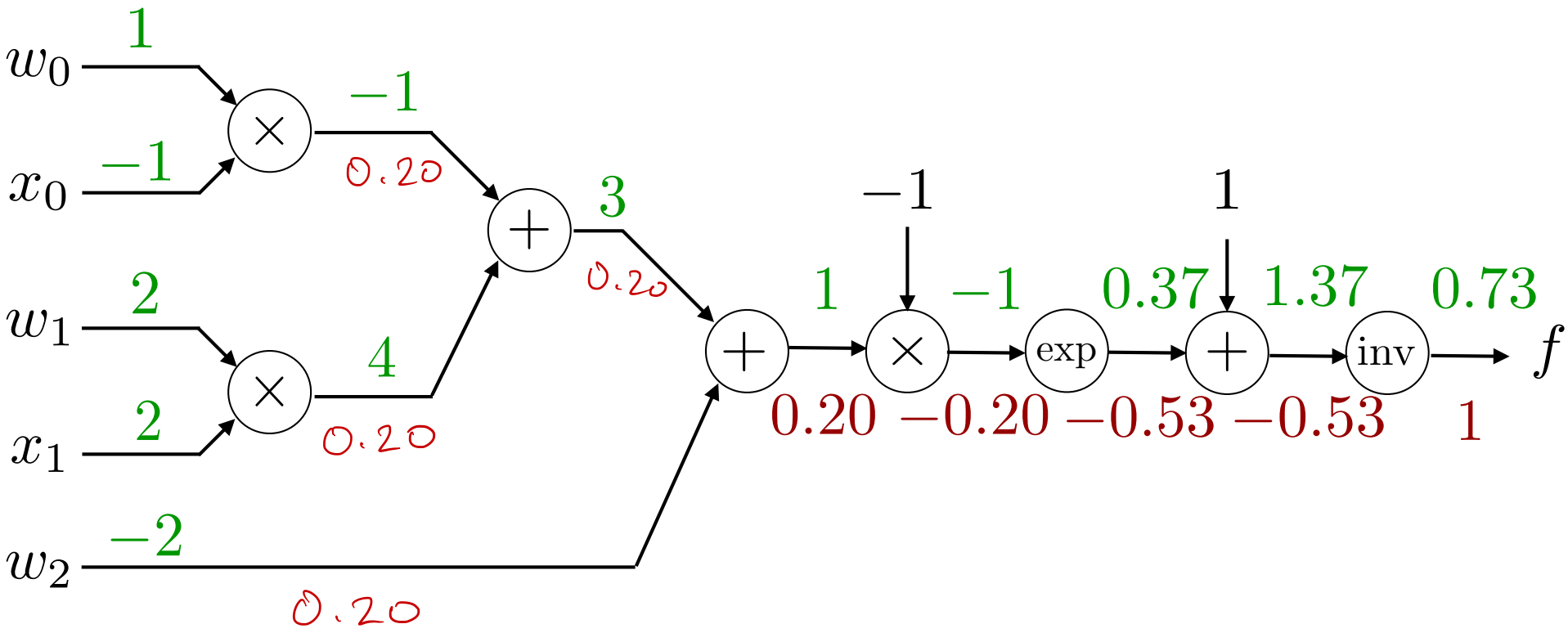
A more involved scalar example

$$f(\mathbf{w},\mathbf{x}) = \frac{1}{1+\exp(-(w_0x_0 + w_1x_1 + w_2))}$$



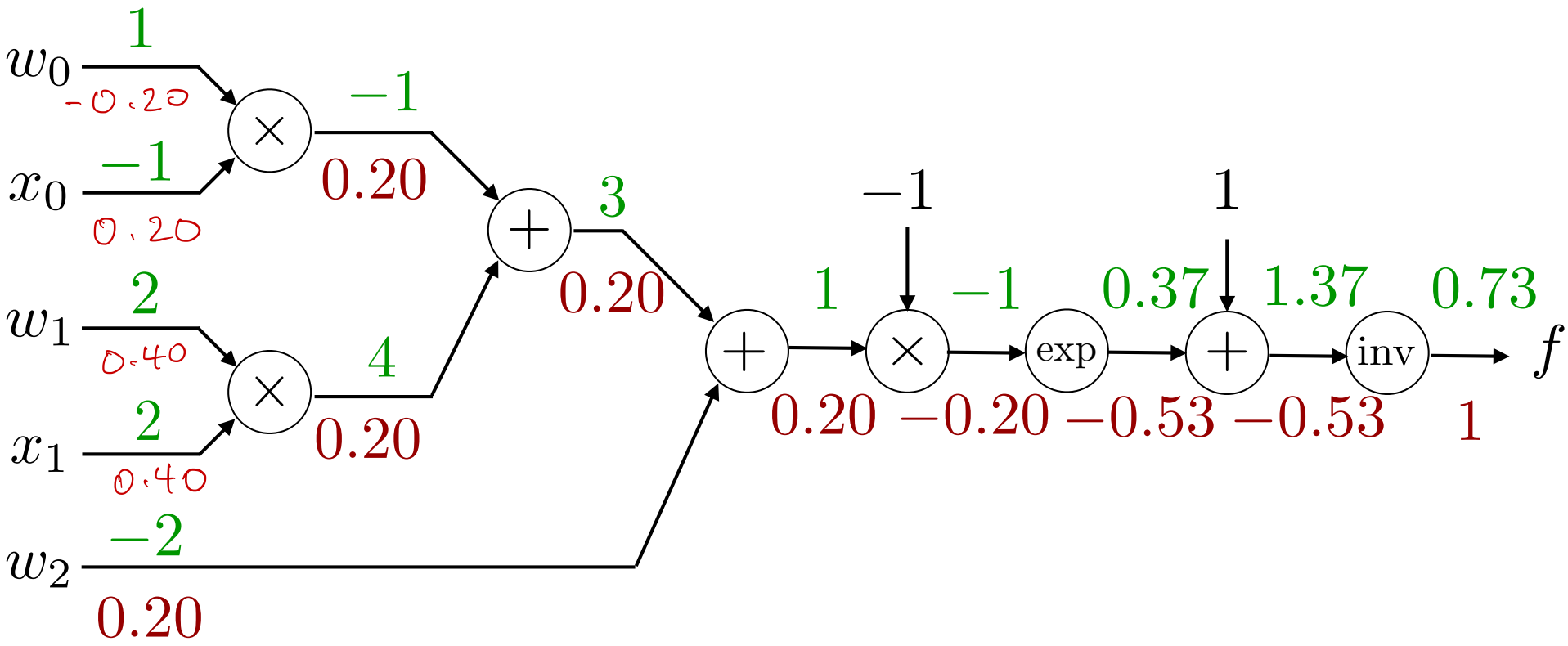
A more involved scalar example

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



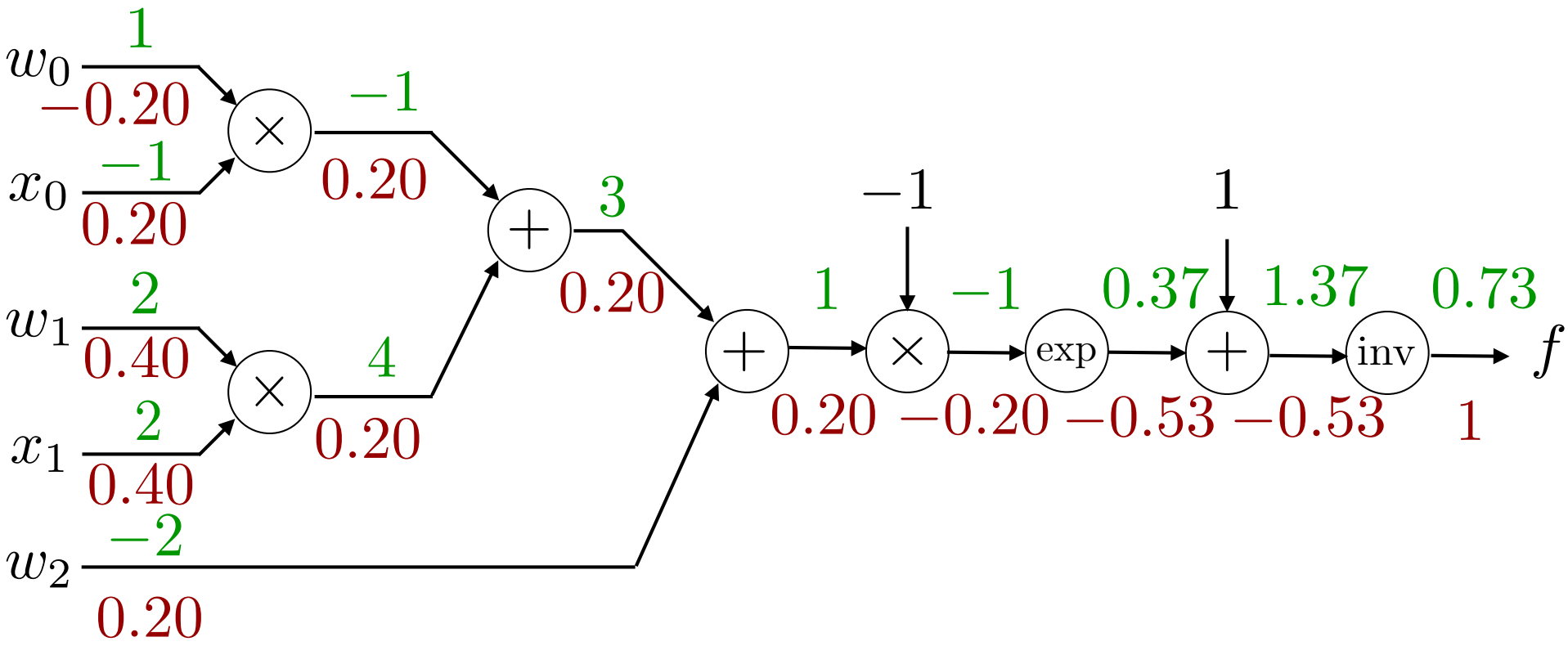
A more involved scalar example

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



A more involved scalar example

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

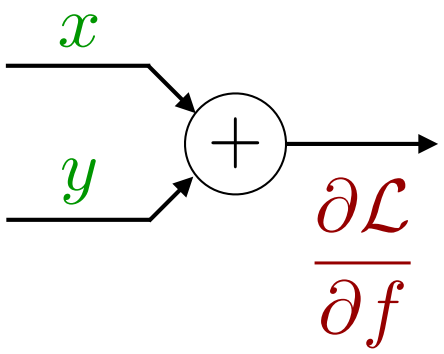


You can take any gradient this way

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.**

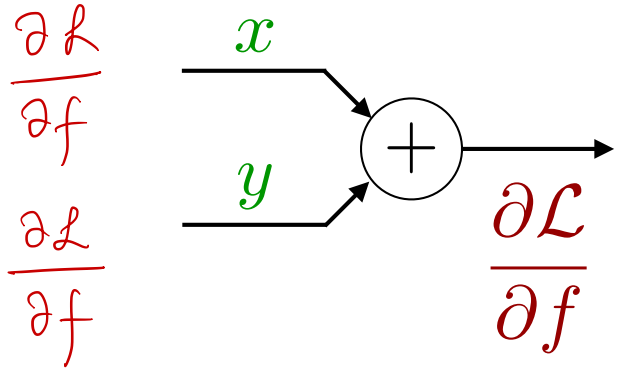
A gate view of gradients

Interpreting backpropagation as gradient “gates”:



A gate view of gradients

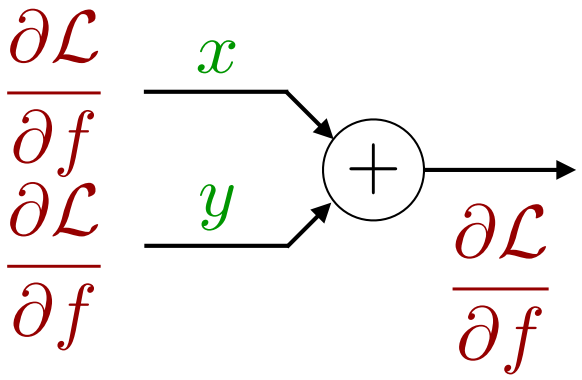
Interpreting backpropagation as gradient “gates”:



A gate view of gradients

Interpreting backpropagation as gradient “gates”:

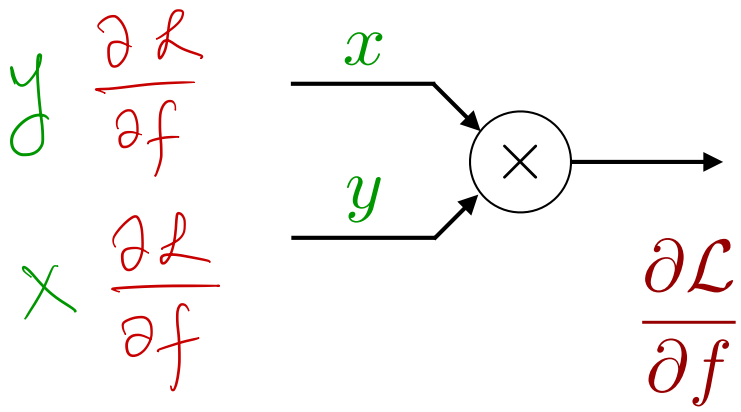
Add gate: distributes the gradient



A gate view of gradients

Interpreting backpropagation as gradient “gates”:

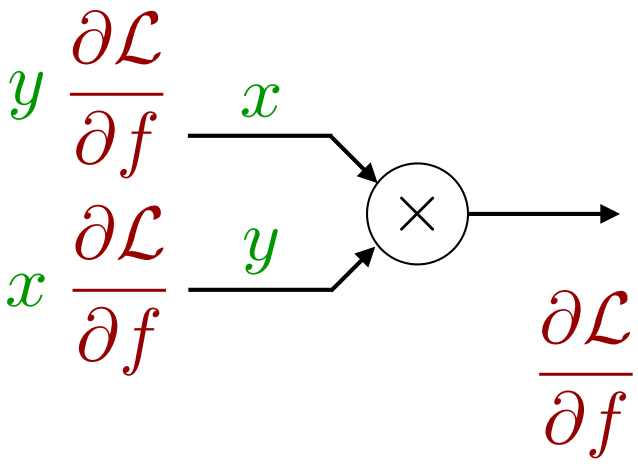
Add gate: distributes the gradient



A gate view of gradients

Interpreting backpropagation as gradient “gates”:

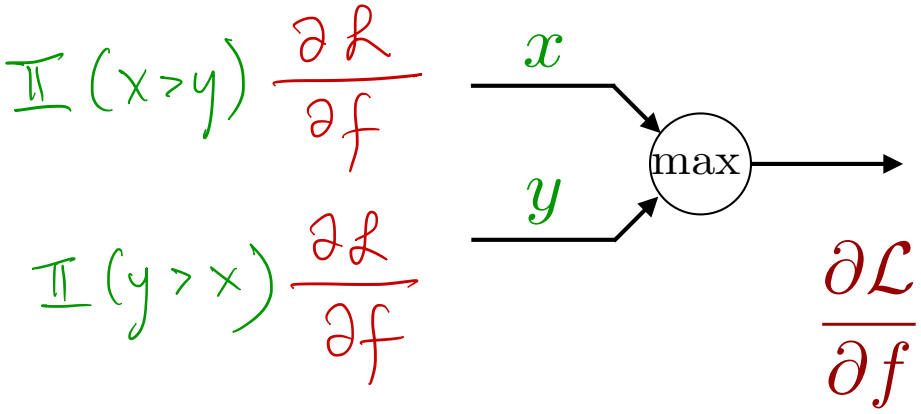
Add gate: distributes the gradient
Mult gate: switches the gradient



A gate view of gradients

Interpreting backpropagation as gradient “gates”:

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



A gate view of gradients

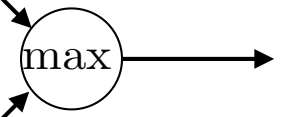
Interpreting backpropagation as gradient “gates”:

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

$x > y$

$$\frac{\partial f}{\partial f} = \mathbb{I}(x > y)\frac{\partial \mathcal{L}}{\partial f} \quad x$$

$$0 = \mathbb{I}(y > x)\frac{\partial \mathcal{L}}{\partial f} \quad y$$



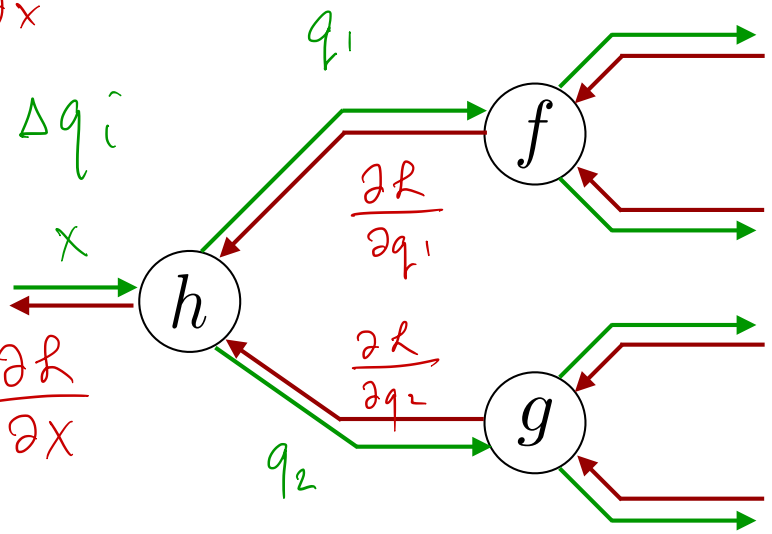
$$\frac{\partial \mathcal{L}}{\partial f}$$

What happens when two gradient paths converge?

$$\frac{\partial \ell}{\partial x} = \sum_{i=1}^{N} \frac{\partial \ell}{\partial q_i} \frac{\partial q_i}{\partial x}$$

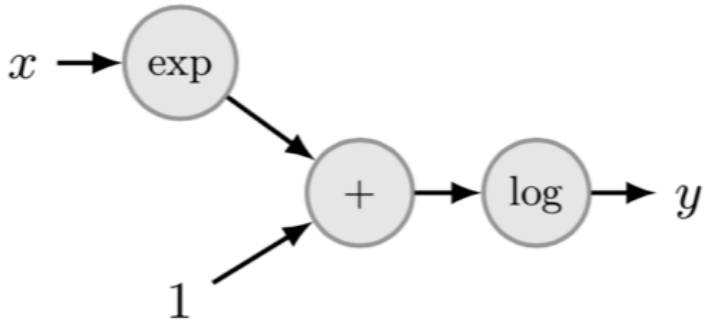
$$\Delta q_i = \frac{\partial q_i}{\partial x} \Delta x$$

$$\Delta \ell = \frac{\partial \ell}{\partial q_i} \Delta q_i$$



One last example

- $y = \text{softplus}(x)$. $= \log(1 + \exp(x))$

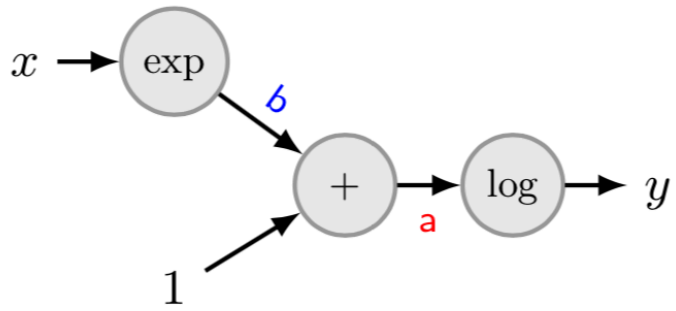


Backpropagation example: softplus

- $y = \text{softplus}(x) = \log(1 + \exp(x))$. We can analytically calculate the derivative.

$$\frac{dy}{dx} = \frac{\exp(x)}{1 + \exp(x)}$$

Let's now calculate it via backpropagation.



Here, we have that $b = \exp(x)$ and that $a = 1 + \exp(x)$. Applying the chain rule, we have that:

$$\begin{aligned}
\frac{dy}{da} &= \frac{d}{da}\log a = \frac{1}{a} \\
\frac{dy}{db} &= \frac{da}{db}\frac{dy}{da} = \frac{dy}{da} \\
\frac{dy}{dx} &= \frac{db}{dx}\frac{dy}{db} = \exp(x)\frac{1}{a}
\end{aligned}$$

Multivariate backpropagation

To do multivariate backpropagation, we need a multivariate chain rule.

Derivative of a scalar w.r.t. a vector

$$\nabla_{\mathbf{x}} y = \begin{bmatrix} \frac{\partial y}{\partial x_1} \\ \frac{\partial y}{\partial x_2} \\ \vdots \\ \frac{\partial y}{\partial x_n} \end{bmatrix}$$

In other words, the gradient is:

- A *vector* that is the same size as $\mathbf{x}$, i.e., if $\mathbf{x} \in \mathbb{R}^n$ then $\nabla_{\mathbf{x}} y \in \mathbb{R}^n$.
- Each dimension of $\nabla_{\mathbf{x}} y$ tells us how small changes in $\mathbf{x}$ in that dimension affect y. i.e., changing the ith dimension of $\mathbf{x}$ by a small amount, Δx_i, will change y by

  $$\frac{\partial y}{\partial x_i} \Delta x_i$$

  We may also denote this as:

  $$\left(\nabla_{\mathbf{x}} y\right)_i \Delta x_i$$

Derivative of a scalar w.r.t. a matrix

The derivative of a scalar, y, with respect to a matrix, $\mathbf{A} \in \mathbb{R}^{m \times n}$, is given by:

$$\nabla_{\mathbf{A}} y = \begin{bmatrix} \frac{\partial y}{\partial a_{11}} & \frac{\partial y}{\partial a_{12}} & \cdots & \frac{\partial y}{\partial a_{1n}} \\ \frac{\partial y}{\partial a_{21}} & \frac{\partial y}{\partial a_{22}} & \cdots & \frac{\partial y}{\partial a_{2n}} \\ \cdots & \cdots & \ddots & \vdots \\ \frac{\partial y}{\partial a_{m1}} & \frac{\partial y}{\partial a_{m2}} & \cdots & \frac{\partial y}{\partial a_{mn}} \end{bmatrix}$$

Like the gradient, the i, jth element of $\nabla_{\mathbf{A}} y$ tells us how small changes in a_{ij} affect y.

Note:

- If you search for the derivative of a scalar with respect to a matrix, you may find people give a transposed definition to the one above.
- Both are valid, but you must be consistent with your notation and use the correct rules. Our notation is called "denominator layout" notation; the other layout is called "numerator layout" notation.
- In the denominator layout, the dimensions of $\nabla_{\mathbf{A}} y$ and $\mathbf{A}$ are the same. The same holds for the gradient, i.e., the dimensions of $\nabla_{\mathbf{x}} y$ and $\mathbf{x}$ are the same. In the numerator layout notation, the dimensions are transposed.
- More on this later, but in "denominator layout," the chain rule goes right to left as opposed to left to right.

Derivative of a vector w.r.t. a vector

Let $\mathbf{y} \in \mathbb{R}^n$ be a function of $\mathbf{x} \in \mathbb{R}^m$. What dimensionality should the derivative of $\mathbf{y}$ with respect to $\mathbf{x}$ be?

- e.g., to see how $\Delta\mathbf{x}$ modifies y_i, we would calculate:

$$\Delta y_i = \nabla_{\mathbf{x}} y_i \cdot \Delta\mathbf{x}$$

- This suggests that the derivative ought to be an $n \times m$ matrix, denoted $\mathbf{J}$, of the form:

$$\mathbf{J} = \begin{bmatrix} (\nabla_{\mathbf{x}} y_1)^T \\ (\nabla_{\mathbf{x}} y_2)^T \\ \vdots \\ (\nabla_{\mathbf{x}} y_n)^T \end{bmatrix} \begin{bmatrix} \Delta x \end{bmatrix}$$

$$= \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \frac{\partial y_1}{\partial x_2} & \cdots & \frac{\partial y_1}{\partial x_m} \\ \frac{\partial y_2}{\partial x_1} & \frac{\partial y_2}{\partial x_2} & \cdots & \frac{\partial y_2}{\partial x_m} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial y_n}{\partial x_1} & \frac{\partial y_n}{\partial x_2} & \cdots & \frac{\partial y_n}{\partial x_m} \end{bmatrix}$$

The matrix would tell us how a small change in $\Delta\mathbf{x}$ results in a small change in $\Delta\mathbf{y}$ according to the formula:

$$\Delta\mathbf{y} \approx \mathbf{J}\Delta\mathbf{x}$$

Derivative of a vector w.r.t. a vector (cont.)

The matrix $\mathbf{J}$ is called the Jacobian matrix.

A word on notation:

- In the denominator layout definition, the denominator vector changes along columns.

$$\nabla_x f(x) = \begin{bmatrix} \end{bmatrix} \qquad \nabla_{\mathbf{x}}\mathbf{y} = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \frac{\partial y_2}{\partial x_1} & \cdots & \frac{\partial y_n}{\partial x_1} \\ \frac{\partial y_1}{\partial x_2} & \frac{\partial y_2}{\partial x_2} & \cdots & \frac{\partial y_n}{\partial x_2} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial y_1}{\partial x_m} & \frac{\partial y_2}{\partial x_m} & \cdots & \frac{\partial y_n}{\partial x_m} \end{bmatrix} \in \mathbb{R}^{m \times n}$$

$$\triangleq \frac{\partial \mathbf{y}}{\partial \mathbf{x}}$$

- Hence the notation we use for the Jacobian would be:

$$\mathbf{J} = (\nabla_{\mathbf{x}}\mathbf{y})^T = \left(\frac{\partial \mathbf{y}}{\partial \mathbf{x}}\right)^T$$

Derivative of a vector w.r.t. a vector

Example: derivative of a vector with respect to a vector

The following derivative will appear later on in the class. Let $\mathbf{W} \in \mathbb{R}^{h \times n}$ and $\mathbf{x} \in \mathbb{R}^n$. We would like to calculate the derivative of $f(\mathbf{x}) = \mathbf{W}\mathbf{x}$ with respect to $\mathbf{x}$.

$$\nabla_{\mathbf{x}} \mathbf{W}\mathbf{x} = \nabla_{\mathbf{x}} \begin{bmatrix} w_{11}x_1 + w_{12}x_2 + \cdots + w_{1n}x_n \\ w_{21}x_1 + w_{22}x_2 + \cdots + w_{2n}x_n \\ \vdots \\ w_{h1}x_1 + w_{h2}x_2 + \cdots + w_{hn}x_n \end{bmatrix}$$

$$= \begin{bmatrix} w_{11} & w_{21} & \cdots & w_{h1} \\ w_{12} & w_{22} & \cdots & w_{h2} \\ \vdots & \vdots & \ddots & \vdots \\ w_{1n} & w_{2n} & \cdots & w_{hn} \end{bmatrix}$$

$$= \mathbf{W}^T$$

Hessian: the generalization of the second derivative

Hessian

The Hessian matrix of a function $f(\mathbf{x})$ is a square matrix of second-order partial derivatives of $f(\mathbf{x})$. It is composed of elements:

$$\mathbf{H} = \begin{bmatrix} \frac{\partial f}{\partial x_1^2} & \frac{\partial f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial f}{\partial x_1 \partial x_n} \\ \frac{\partial f}{\partial x_2 x_1} & \frac{\partial f}{\partial x_2^2} & \cdots & \frac{\partial f}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f}{\partial x_m x_1} & \frac{\partial f}{\partial x_m \partial x_2} & \cdots & \frac{\partial f}{\partial x_m^2} \end{bmatrix}$$

We can denote this matrix as $\nabla_{\mathbf{x}} (\nabla_{\mathbf{x}} f(\mathbf{x}))$. We often denote this simply as $\nabla_{\mathbf{x}}^2 f(\mathbf{x})$.

The scalar chain rule

The scalar chain rule states that if $y = f(x)$ and $z = g(y)$, then

$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}$$

Intuitively: the chain rule tells us that a small change in x will cause a small change in y that will in turn cause a small change in z, i.e., for apropriately small Δx,

$$\begin{aligned} \Delta y &\approx \frac{dy}{dx}\Delta x \\ \Delta z &\approx \frac{dz}{dy}\Delta y \\ &= \frac{dz}{dy}\frac{dy}{dx}\Delta x \end{aligned}$$

Vector chain rule

Chain rule for vector valued functions

In the "denominator" layout, the chain rule runs from right to left. We won't derive this, but we will check the dimensionality and intuition.

Let $\mathbf{x} \in \mathbb{R}^m$, $\mathbf{y} \in \mathbb{R}^n$, and $\mathbf{z} \in \mathbb{R}^p$. Further, let $\mathbf{y} = f(\mathbf{x})$ for $f : \mathbb{R}^m \to \mathbb{R}^n$ and $\mathbf{z} = g(\mathbf{y})$ for $g : \mathbb{R}^n \to \mathbb{R}^p$. Then,

$$\nabla_{\mathbf{x}} \mathbf{z} \quad = \quad \nabla_{\mathbf{x}} \mathbf{y} \, \nabla_{\mathbf{y}} \mathbf{z}$$

$y = f(x)$

$z = g(y)$

Equivalently:

$$\frac{\partial \mathbf{z}}{\partial \mathbf{x}} \quad = \quad \frac{\partial \mathbf{y}}{\partial \mathbf{x}} \frac{\partial \mathbf{z}}{\partial \mathbf{y}}$$

- Note that $\nabla_{\mathbf{x}} \mathbf{z}$ should have dimensionality $\mathbb{R}^{m \times p}$.
- As $\nabla_{\mathbf{x}} \mathbf{y} \in \mathbb{R}^{m \times n}$ and $\nabla_{\mathbf{y}} \mathbf{z} \in \mathbb{R}^{n \times p}$, the operations are dimension consistent.

Chain rule for vector valued functions (cont.)

- Intuitively, a small change $\Delta\mathbf{x}$ affects $\Delta\mathbf{z}$ through the Jacobian $(\nabla_{\mathbf{x}}\mathbf{z})^T$

$$\Delta\mathbf{z} \approx (\nabla_{\mathbf{x}}\mathbf{z})^T \Delta\mathbf{x} \tag{1}$$

- The chain rule is intuitive, since:

$$\begin{aligned} \Delta\mathbf{y} &\approx (\nabla_{\mathbf{x}}\mathbf{y})^T \Delta\mathbf{x} \\ \Delta\mathbf{z} &\approx (\nabla_{\mathbf{y}}\mathbf{z})^T \Delta\mathbf{y} \end{aligned}$$

Composing these, we have that:

$$\Delta\mathbf{z} \approx (\nabla_{\mathbf{y}}\mathbf{z})^T (\nabla_{\mathbf{x}}\mathbf{y})^T \Delta\mathbf{x} \tag{2}$$

- Combining equations (1) and (2), we arrive at:

$$(\nabla_{\mathbf{x}}\mathbf{z})^T = (\nabla_{\mathbf{y}}\mathbf{z})^T (\nabla_{\mathbf{x}}\mathbf{y})^T$$

which, after transposing both sides, reduces to the (right to left) chain rule:

$$\nabla_{\mathbf{x}}\mathbf{z} = \nabla_{\mathbf{x}}\mathbf{y}\, \nabla_{\mathbf{y}}\mathbf{z}$$

Multivariate chain rule example

Consider the squared loss function: $\varepsilon = \|\mathbf{x} - \mathbf{h}\|^2$, where $\mathbf{h}$ is some target we are trying to approximate via $\mathbf{x}$. We wish to calculate $\nabla_{\mathbf{x}}\varepsilon$.

We will set $\mathbf{y} = \mathbf{x} - \mathbf{h}$ and use the chain rule. Note that $\varepsilon = \mathbf{y}^T\mathbf{y}$.

- First, we find $\nabla_{\mathbf{y}}\varepsilon$.

$$\begin{aligned} \nabla_{\mathbf{y}}\varepsilon &= \nabla_{\mathbf{y}}\left(\sum_i y_i^2\right) \\ &= 2\mathbf{y} \end{aligned}$$

- Next, we find $\nabla_{\mathbf{x}}\mathbf{y}$.

$$\begin{aligned} \nabla_{\mathbf{x}}\mathbf{y} &= \begin{bmatrix} \frac{\partial(x_1-h_1)}{\partial x_1} & \frac{\partial(x_2-h_2)}{\partial x_1} & \cdots & \frac{\partial(x_n-h_n)}{\partial x_1} \\ \frac{\partial(x_1-h_1)}{\partial x_2} & \frac{\partial(x_2-h_2)}{\partial x_2} & \cdots & \frac{\partial(x_n-h_n)}{\partial x_2} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial(x_1-h_1)}{\partial x_n} & \frac{\partial(x_2-h_2)}{\partial x_n} & \cdots & \frac{\partial(x_n-h_n)}{\partial x_n} \end{bmatrix} \\ &= \mathbf{I} \end{aligned}$$

Vector chain rule example

Multivariate chain rule example (cont.)

- Using the chain rule,

$$\begin{aligned} \nabla_{\mathbf{x}} \varepsilon &= (\nabla_{\mathbf{x}} \mathbf{y}) (\nabla_{\mathbf{y}} \varepsilon) \\ &= 2(\mathbf{x} - \mathbf{h}) \end{aligned}$$

A few notes:

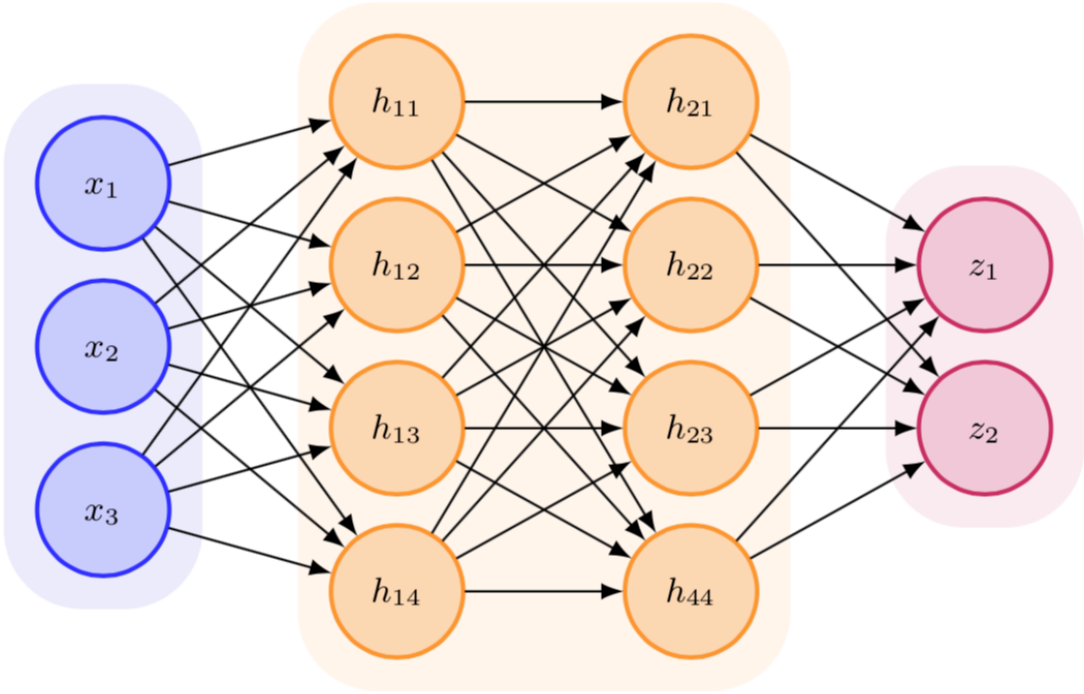
- Formally, $\nabla_{\mathbf{x}} \mathbf{y}$ is an $n \times n$ matrix. However, knowing that $\mathbf{y} = \mathbf{x} - \mathbf{h}$, we can intuit that its affect should be multiplication by 1 (or formally, the identity matrix). This intuition is important for deep learning and dealing with derivatives of tensors.
- Often times, we don't want to represent and store the entire tensor derivative, which could be enormous. Many entries in this tensor derivative will be 0, much like many of the entries in $\nabla_{\mathbf{x}} \mathbf{y}$ are zero.

Derivatives with respect to tensors

Derivatives of tensors

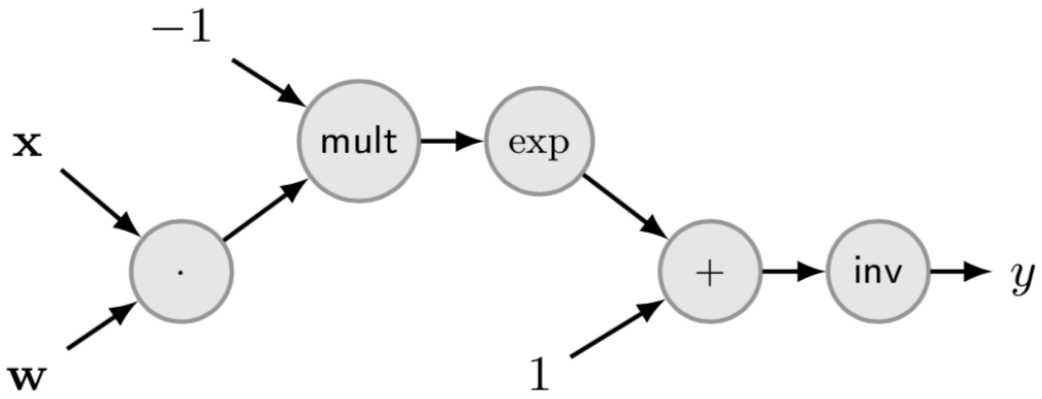
Occasionally in this class, we may need to take a derivative that is more than 2-dimensional. For example, we may want to take the derivative of a vector with respect to a matrix. This would be a 3-dimensional tensor. However, we typically can find a shortcut to perform operations without having to store these high-dimensional tensor derivatives. We'll discuss this more when we get to it in backpropagation.

Back to backpropagation

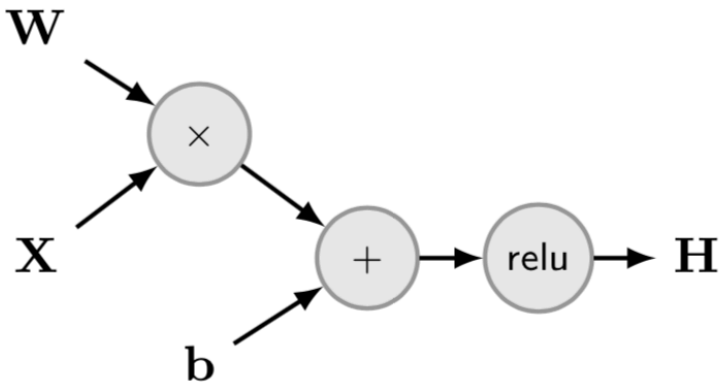


Computational graphs (cont)

- $y = \sigma(\mathbf{w}^T\mathbf{x}) = \frac{1}{1+\exp(-\mathbf{w}^T\mathbf{x})}$. (Note, we let $\cdot$ denote dot product.)

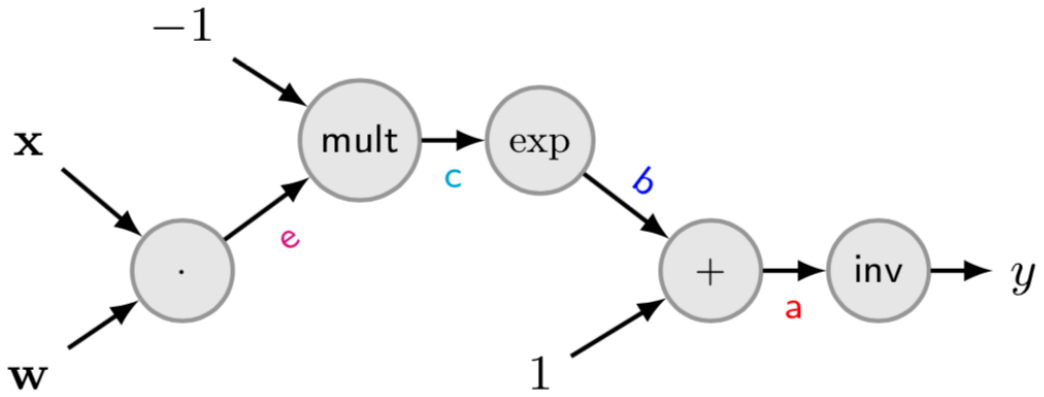


- $\mathbf{H} = \mathrm{ReLU}(\mathbf{WX} + \mathbf{b})$. (Note: we let $\times$ denote matrix multiplication with appropriate order.)



Backpropagation example: sigmoid

- $y = \sigma(\mathbf{w}^T\mathbf{x}) = \frac{1}{1+\exp(-\mathbf{w}^T\mathbf{x})}$.



Apply the chain rule:

$$\begin{aligned}
\frac{dy}{da} &= -\frac{1}{a^2} \\
\frac{dy}{db} &= \frac{dy}{da} \\
\frac{dy}{dc} &= \exp(c)\frac{dy}{db} \\
\frac{dy}{de} &= -\frac{dy}{dc}
\end{aligned}$$

Backpropagation example: sigmoid (cont.)

Finally, we have that:

$$\begin{aligned}\nabla_{\mathbf{x}} y &= \nabla_{\mathbf{x}} e \frac{dy}{de} \\ \nabla_{\mathbf{w}} y &= \nabla_{\mathbf{w}} e \frac{dy}{de}\end{aligned}$$

Putting it all together, we have that:

$$\begin{aligned}\nabla_{\mathbf{w}} y &= \mathbf{x}\frac{dy}{de} \\ &= -\mathbf{x}\frac{dy}{dc} \\ &= -\exp(-\mathbf{w}^T\mathbf{x})\mathbf{x}\frac{dy}{db} \\ &= -\exp(-\mathbf{w}^T\mathbf{x})\mathbf{x}\frac{dy}{da} \\ &= -\frac{\exp(-\mathbf{w}^T\mathbf{x})}{(1+\exp(-\mathbf{w}^T\mathbf{x}))^2}\mathbf{x}\end{aligned}$$

A similar computation could be done to compute $\nabla_{\mathbf{x}} y$.

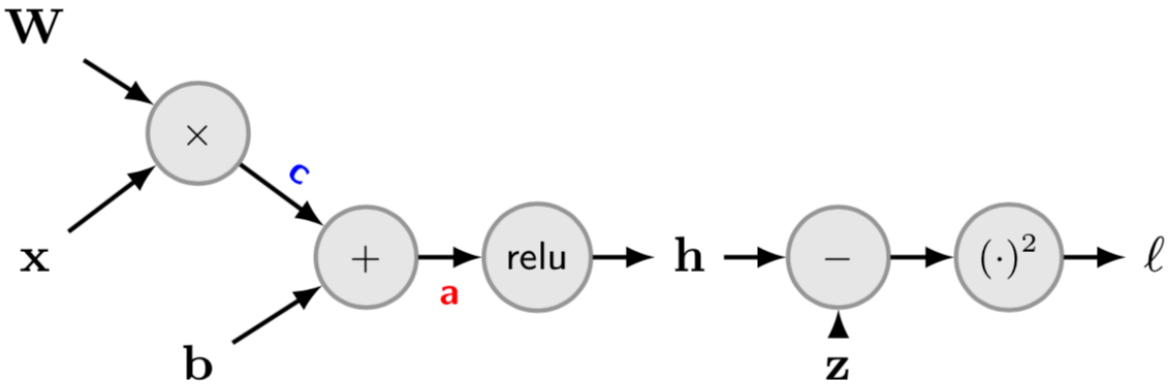
Backpropagation: a few intermediate notes

A few things to note thus far:

- So far, all multiplications have been scalar-scalar or vector-scalar, where order does not matter. It may be the case that we have matrix-vector or matrix-matrix multiplies, in which case the order should be correct. While one could do the rigorous math to determine the order, in general, one quick "trick" to get the order correct with less cognitive effort is to look at the dimensionality.
- Prior to backpropagation, we would have performed a forward pass to calculate intermediate values. These intermediate values were denoted a, b, c, e, etc. in the prior plots. We can cache these and would not have to recalculate their values in doing backpropagation.

Backpropagation: neural network layer

Here, $\mathbf{h} = \mathrm{ReLU}(\mathbf{W}\mathbf{x} + \mathbf{b})$, with $\mathbf{h} \in \mathbb{R}^h$, $\mathbf{x} \in \mathbb{R}^m$, $\mathbf{W} \in \mathbb{R}^{h \times m}$, and $\mathbf{b} \in \mathbb{R}^h$. While many output cost functions might be used, let's consider a simple squared-loss output $\ell = (\mathbf{h} - \mathbf{z})^2$ where $\mathbf{z}$ is some target value.



A few things to note:

- Note that $\nabla_{\mathbf{h}} \ell = 2(\mathbf{h} - \mathbf{z})$. We will start backpropagation at $\mathbf{h}$ rather than at ℓ.
- In the following backpropagation, we'll have need for elementwise multiplication. This is formally called a Hadamard product, and we will denote it via $\odot$. Concretely, the ith entry of $\mathbf{x} \odot \mathbf{y}$ is given by $x_i y_i$.

Backpropagation: neural network layer (cont.)

Applying the chain rule, we have:

$$\begin{aligned}
\frac{\partial \ell}{\partial \mathbf{a}} &= \mathbb{I}(\mathbf{a} > 0) \odot \frac{\partial \ell}{\partial \mathbf{h}} \\
\frac{\partial \ell}{\partial \mathbf{c}} &= \frac{\partial \ell}{\partial \mathbf{a}} \\
\frac{\partial \ell}{\partial \mathbf{x}} &= \frac{\partial \mathbf{c}}{\partial \mathbf{x}} \frac{\partial \ell}{\partial \mathbf{c}} \\
&= \mathbf{W}^T \frac{\partial \ell}{\partial \mathbf{c}}
\end{aligned}$$

A few notes:

- For $\frac{\partial \mathbf{c}}{\partial \mathbf{x}}$, see example in the Tools notes.
- Why was the chain rule written right to left instead of left to right? This turns out to be a result of our convention of how we defined derivatives (using the "denominator layout notation") in the linear algebra notes.
- Though maybe not the most satisfying answer, you can always check the order of operations is correct by considering non-square matrices, where the dimensionality must be correct.

Backpropagation: neural network layer (cont.)

What about $\frac{\partial \ell}{\partial \mathbf{W}}$? In doing backpropagation, we have to calculate $\frac{\partial \mathbf{c}}{\partial \mathbf{W}}$ which is a 3-dimensional tensor.

However, we also intuit the following (informally):

- $\frac{\partial \ell}{\partial \mathbf{W}}$ is a matrix that is $h \times m$.
- The derivative of $\mathbf{W}\mathbf{x}$ with respect to $\mathbf{W}$ "ought to look like" $\mathbf{x}$.
- Since $\frac{\partial \ell}{\partial \mathbf{c}} \in \mathbb{R}^h$, and $\mathbf{x} \in \mathbb{R}^m$ then, intuitively,

$$\frac{\partial \ell}{\partial \mathbf{W}} = \frac{\partial \ell}{\partial \mathbf{c}} \mathbf{x}^T$$

This intuition turns out to be correct, and it is common to use this intuition. However, the first time around we should do this rigorously and then see the general pattern of why this intuition works.

Backpropagation for a neural network layer

Calculating a tensor derivative

$\frac{\partial \mathbf{c}}{\partial \mathbf{W}}$ is an $h \times m \times h$ tensor.

- Letting c_k denote the ith element of $\mathbf{c}$, we see that each $\frac{\partial c_i}{\partial \mathbf{W}}$ is an $h \times m$ matrix, and that there are h of these for each element c_i from $i = 1, \ldots, h$.
- The *matrix*, $\frac{\partial c_i}{\partial \mathbf{W}}$, can be calculated as follows:

$$\frac{\partial c_i}{\partial \mathbf{W}} = \frac{\partial}{\partial \mathbf{W}} \sum_{j=1}^{h} w_{ij} x_j$$

and thus, the (i, j)th element of this matrix is:

$$\left(\frac{\partial c_i}{\partial \mathbf{W}} \right)_{i,j} = x_j$$

It is worth noting that

$$\left(\frac{\partial c_k}{\partial \mathbf{W}} \right)_{i,j} = x_j \qquad \text{for } k \neq i$$

Calculating a tensor derivative (cont.)

Hence, $\frac{\partial c_i}{\partial \mathbf{W}}$ is a matrix where the ith row is $\mathbf{x}^T$ and all other rows are the zeros (we denote the zero vector by $\mathbf{0}$). i.e.,

$$\frac{\partial c_1}{\partial \mathbf{W}} = \begin{bmatrix} -\mathbf{x}^T- \\ -\mathbf{0}^T- \\ \vdots \\ -\mathbf{0}^T- \end{bmatrix} \qquad \frac{\partial c_2}{\partial \mathbf{W}} = \begin{bmatrix} -\mathbf{0}^T- \\ -\mathbf{x}^T- \\ \vdots \\ -\mathbf{0}^T- \end{bmatrix} \qquad \text{etc...}$$

Now applying the chain rule,

$$\frac{\partial \ell}{\partial \mathbf{W}} = \frac{\partial \mathbf{c}}{\partial \mathbf{W}} \frac{\partial \ell}{\partial \mathbf{c}}$$

is a tensor product between an $(h \times m \times h)$ tensor and an $(h \times 1)$ vector, whose resulting dimensionality is $(h \times m \times 1)$ or equivalently, an $(h \times m)$ matrix.

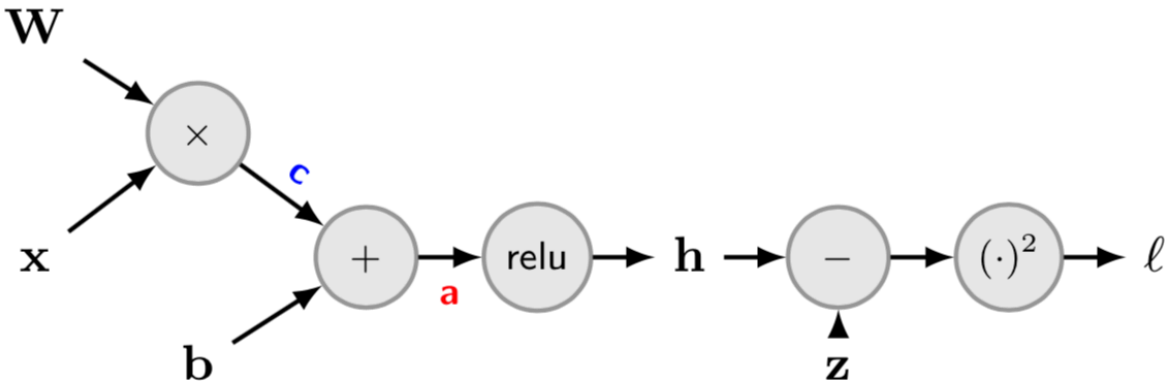
Calculating a tensor derivative (cont.)

We carry out this tensor-vector multiply in the standard way.

$$
\begin{aligned}
\frac{\partial \ell}{\partial \mathbf{W}} &= \frac{\partial \mathbf{c}}{\partial \mathbf{W}} \frac{\partial \ell}{\partial \mathbf{c}} \\
&= \sum_{i=1}^{h} \frac{\partial c_i}{\partial \mathbf{W}} \left(\frac{\partial \ell}{\partial \mathbf{c}} \right)_i \\
&= \left(\frac{\partial \ell}{\partial \mathbf{c}} \right)_1 \begin{bmatrix} -\mathbf{x}^T- \\ -\mathbf{0}^T- \\ \vdots \\ -\mathbf{0}^T- \end{bmatrix} + \left(\frac{\partial \ell}{\partial \mathbf{c}} \right)_2 \begin{bmatrix} -\mathbf{0}^T- \\ -\mathbf{x}^T- \\ \vdots \\ -\mathbf{0}^T- \end{bmatrix} + \cdots + \left(\frac{\partial \ell}{\partial \mathbf{c}} \right)_h \begin{bmatrix} -\mathbf{0}^T- \\ -\mathbf{0}^T- \\ \vdots \\ -\mathbf{x}^T- \end{bmatrix} \\
&= \begin{bmatrix} \left(\frac{\partial \ell}{\partial \mathbf{c}} \right)_1 \mathbf{x}^T \\ \left(\frac{\partial \ell}{\partial \mathbf{c}} \right)_2 \mathbf{x}^T \\ \vdots \\ \left(\frac{\partial \ell}{\partial \mathbf{c}} \right)_h \mathbf{x}^T \end{bmatrix} \\
&= \frac{\partial \ell}{\partial \mathbf{c}} \mathbf{x}^T
\end{aligned}
$$

Backpropagation: neural network layer

Here, $\mathbf{h} = \mathrm{ReLU}(\mathbf{W}\mathbf{x} + \mathbf{b})$, with $\mathbf{h} \in \mathbb{R}^h$, $\mathbf{x} \in \mathbb{R}^m$, $\mathbf{W} \in \mathbb{R}^{h \times m}$, and $\mathbf{b} \in \mathbb{R}^h$. While many output cost functions might be used, let's consider a simple squared-loss output $\ell = (\mathbf{h} - \mathbf{z})^2$ where $\mathbf{z}$ is some target value.



A few things to note:

- Note that $\nabla_{\mathbf{h}} \ell = 2(\mathbf{h} - \mathbf{z})$. We will start backpropagation at $\mathbf{h}$ rather than at ℓ.
- In the following backpropagation, we'll have need for elementwise multiplication. This is formally called a Hadamard product, and we will denote it via $\odot$. Concretely, the ith entry of $\mathbf{x} \odot \mathbf{y}$ is given by $x_i y_i$.

Backpropagation for a neural network layer

A few notes on tensor derivatives

- In general, the simpler rule can be inferred via pattern intuition / looking at the dimensionality of the matrices, and these tensor derivatives need not be explicitly derived. (See intuition three slides prior.)
- Indeed, actually calculating these tensor derivatives, storing them, and then doing e.g., a tensor-vector multiply, is usually not a good idea for both memory and computation. In this example, storing all these zeros and performing the multiplications is unnecessary.
- If we know the end result is simply an outer product of two vectors, we need not even calculate an additional derivative in this step of backpropagation, or store an extra value (assuming the inputs were previously cached).