

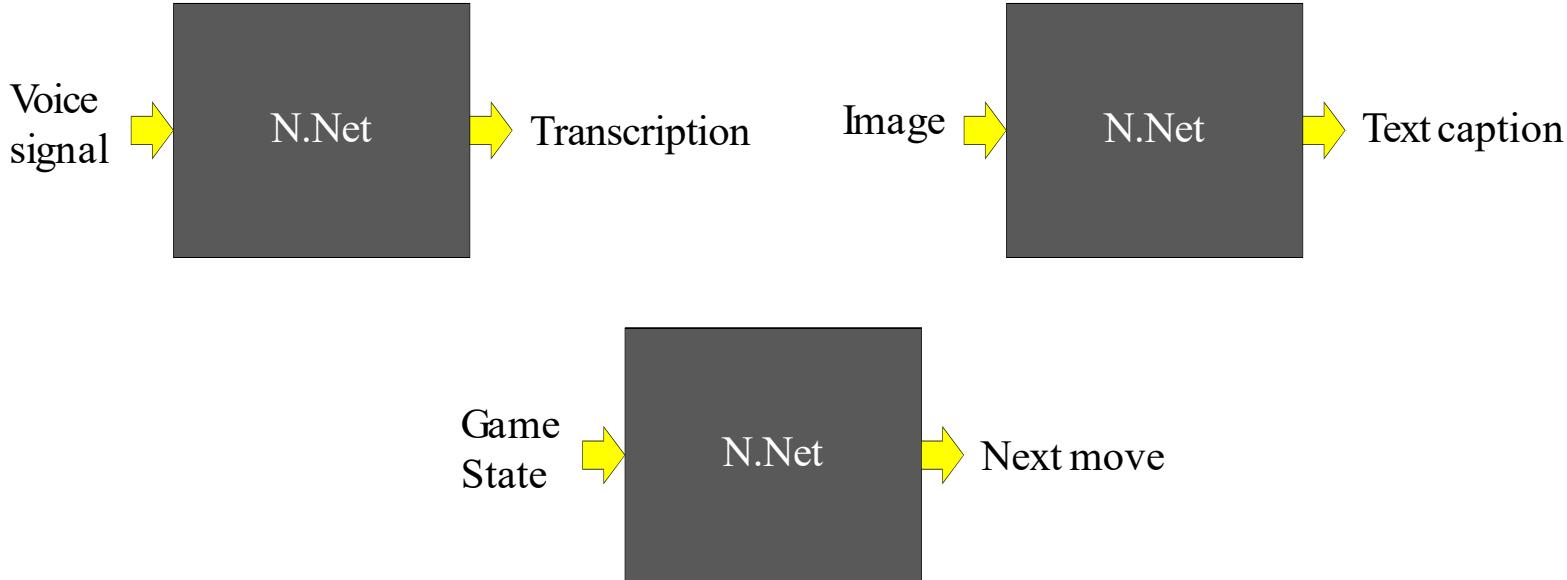
Multi-Layer Networks & Back-Propagation

M. Soleymani
Deep Learning

Sharif University of Technology
Spring 2024

Most slides have been adapted from Bhiksha Raj, 11-785
and some from Fei Fei Li et. al, cs231n, Stanford

These boxes are functions



- Take an input
- Produce an output
- Can be modeled by a neural network!
- How do we compose the network that performs the requisite function?

Problem setup

- Given: the architecture of the network
- Training data: A set of input-output pairs

$$(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \dots, (x^{(N)}, y^{(N)})$$

- We want to find the function f
 - We consider a neural network as a parametric function $f(x; W)$
- We need a **loss function** to show how penalizes the obtained output $f(x; W)$ when the desired output is y

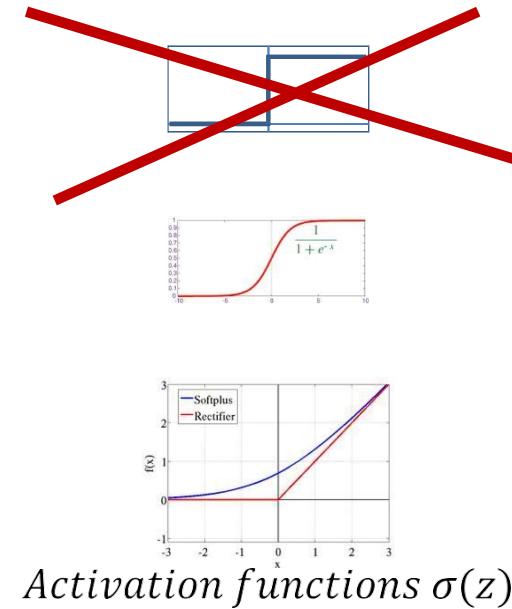
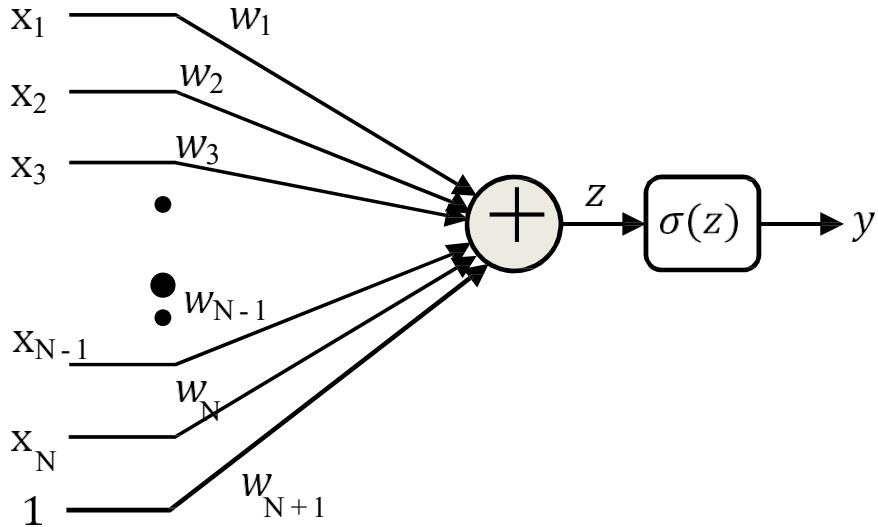
$$E(W) = \frac{1}{N} \sum_{n=1}^N loss(f(x^{(n)}; W), y^{(n)})$$

- Minimize the cost function

Training an MLP

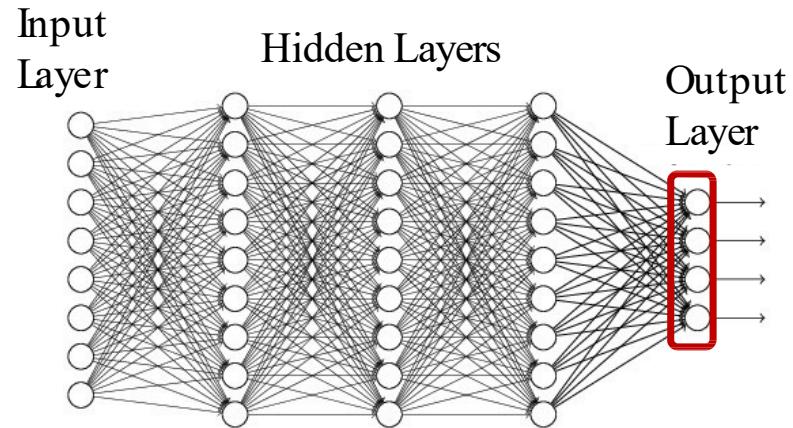
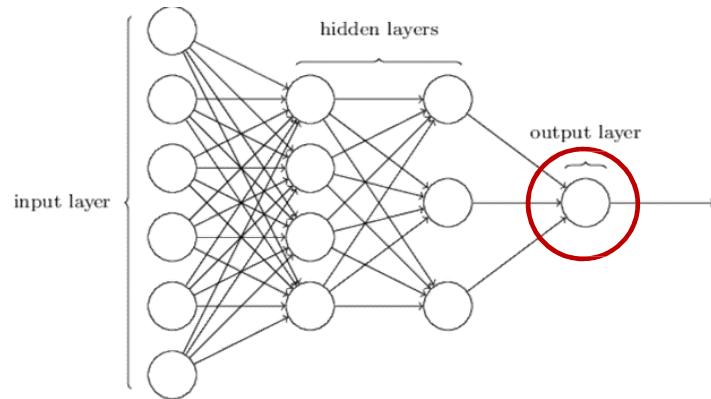
- We define differentiable loss or divergence between the output of the network and the desired output for the training instances
 - And a total error, which is the average divergence over all training instances
- we use continuous activation functions to enables us to estimate network parameters
- We optimize network parameters to minimize this error

Activation function



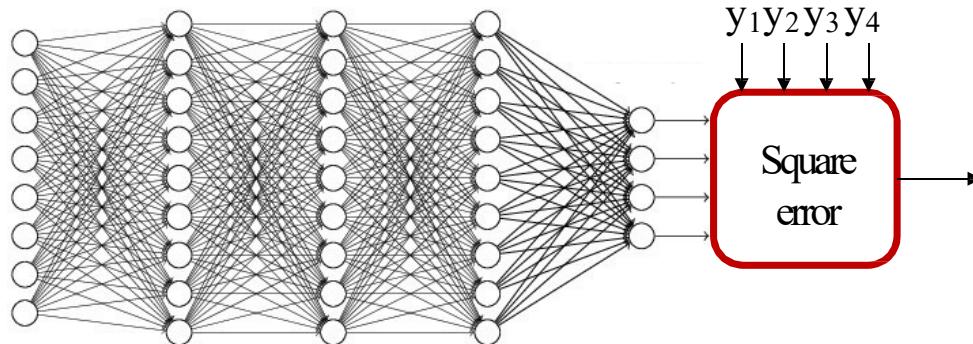
- Makes the neuron differentiable, with non-zero derivatives over much of the input space
 - Small changes in weight can result in non-negligible changes in output
 - This enables us to estimate the parameters using gradient descent techniques..

Representing the output



- If the desired output is real-valued, no special tricks are necessary
 - Scalar Output : single output neuron
 - o = scalar (real value)
 - Vector Output : as many output neurons as the dimension of the desired output
 - $\mathbf{o} = [o_1, o_2, \dots, o_K]^T$ (vector of real values)

Examples of loss functions

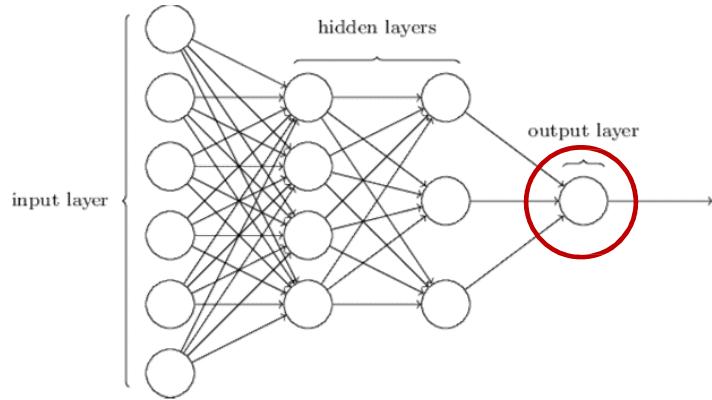


- For real-valued output vectors, the (scaled) L_2 divergence is popular

$$loss(\mathbf{y}, \mathbf{o}) = \frac{1}{2} \|\mathbf{y} - \mathbf{o}\|^2 = \frac{1}{2} \sum_k (y_k - o_k)^2$$

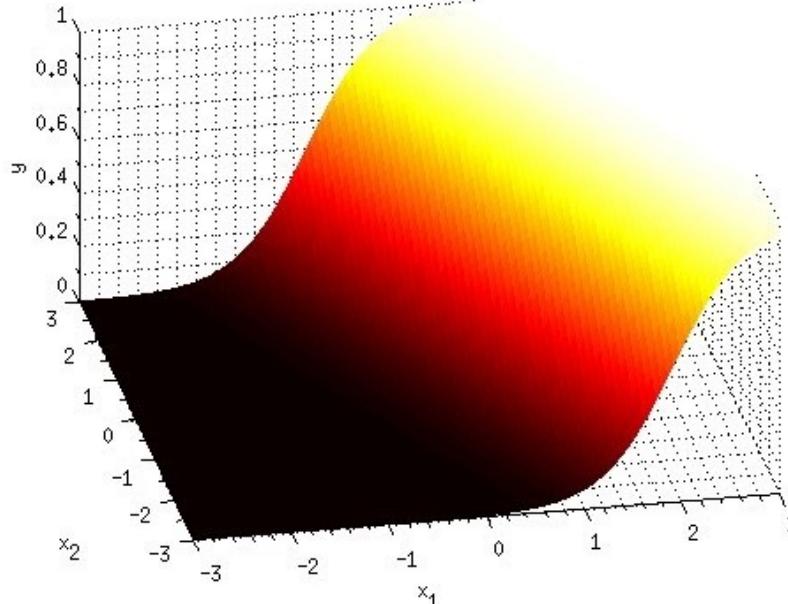
- Squared Euclidean distance between true and desired output
- Note: this is differentiable

Representing the output



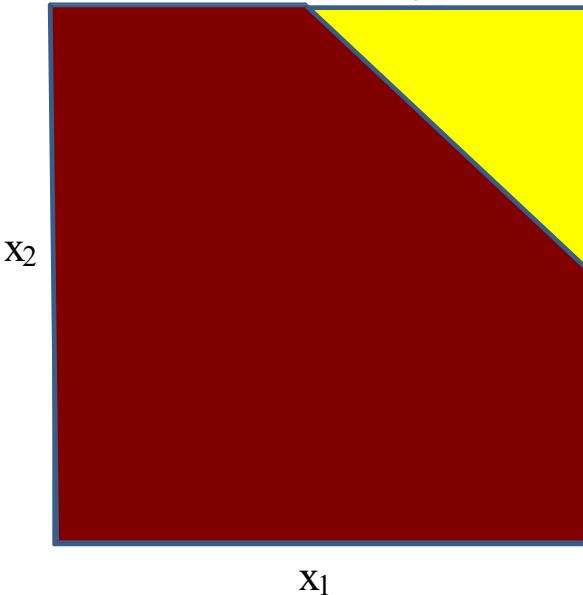
- If the desired output is binary (is this a cat or not), use a simple 1/0 representation of the desired output
 - 1 = YES it's a cat
 - 0 = NO it's not a cat.

Logistic regression



When X is a 2-D variable

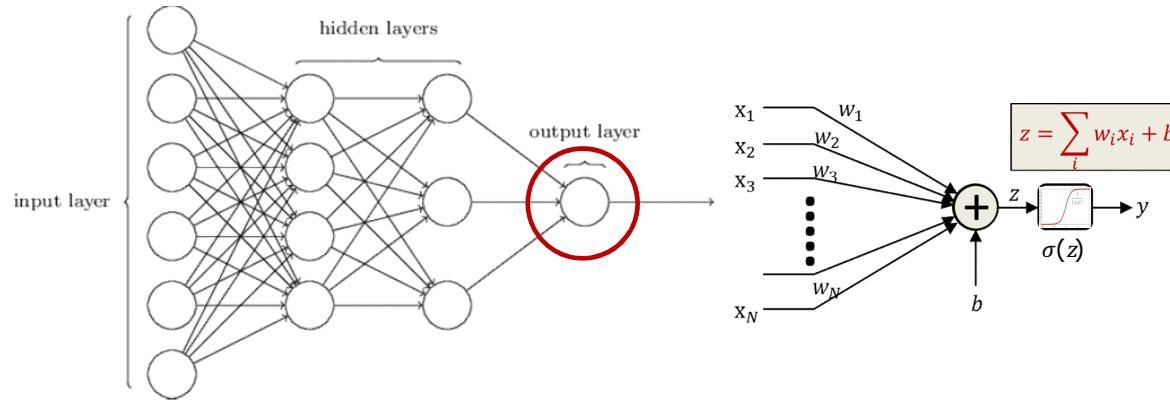
Decision: $P(Y=1|x) \geq 0.5?$



$$P(Y = 1|X) = \frac{1}{1 + \exp(-\sum_i w_i x_i - b)}$$

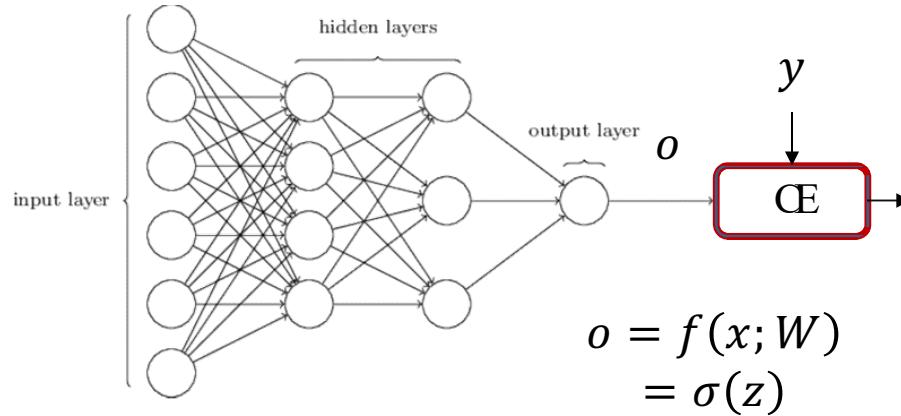
- This is the perceptron with a sigmoid activation
 - It actually computes the probability that the input belongs to class 1

Representing the output



- If the desired output is binary (is this a cat or not), use a simple 1/0 representation of the desired output
- Output activation: Typically a sigmoid
 - Viewed as the probability $P(Y = 1|\mathbf{x})$ of class value 1
 - Indicating the fact that for actual data, in general a feature vector may occur for both classes, but with different probabilities
 - Is differentiable

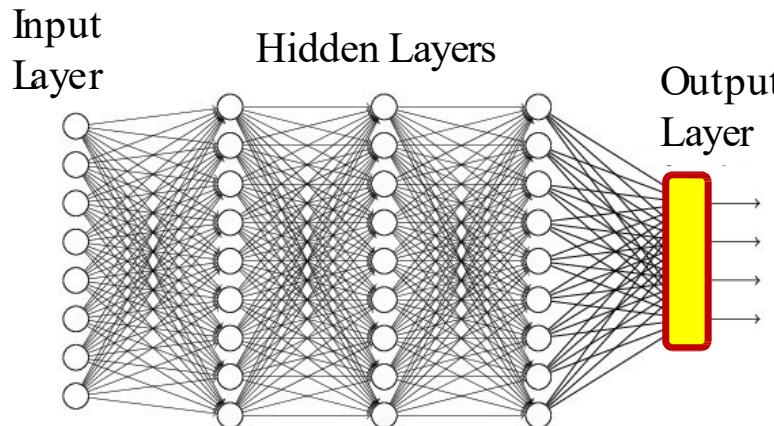
For binary classifier: Logistic regression



- For binary classifier with scalar output $o \in (0,1)$, y is 0/1:

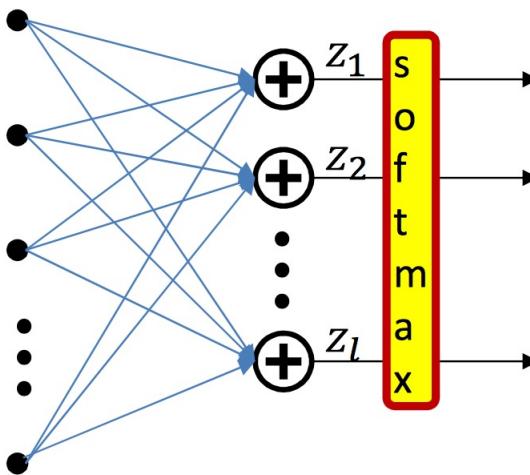
$$\text{loss}(y, o) = -y \log o - (1 - y) \log(1 - o)$$

Multi-class networks



- For a multi-class classifier with K classes, the one-hot representation for the desired output y
 - The neural network's output too must ideally be binary ($K-1$ zeros and a single 1 in the right place)
- More realistically, the network's output will be a probability vector
 - K probability values that sum to 1.

Vector activation example: Softmax



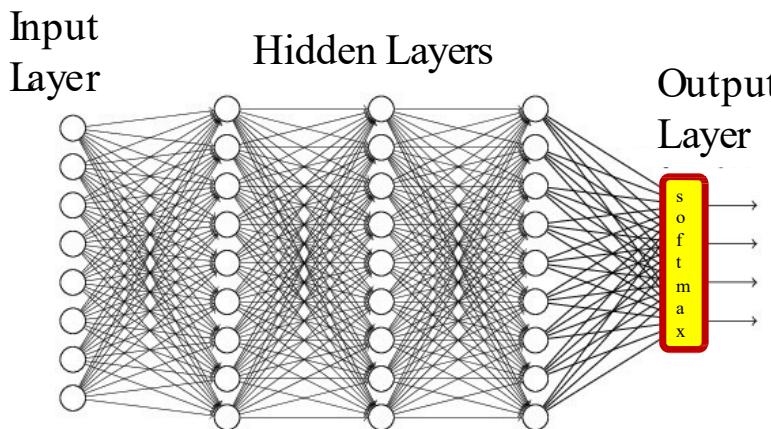
- Example: Softmax **vector** activation

$$z_i = \sum_j w_{ji} x_j + b_i$$

Parameters are
weights w_{ji}
and bias b_i

$$o_i = \frac{\exp(z_i)}{\sum_j \exp(z_j)}$$

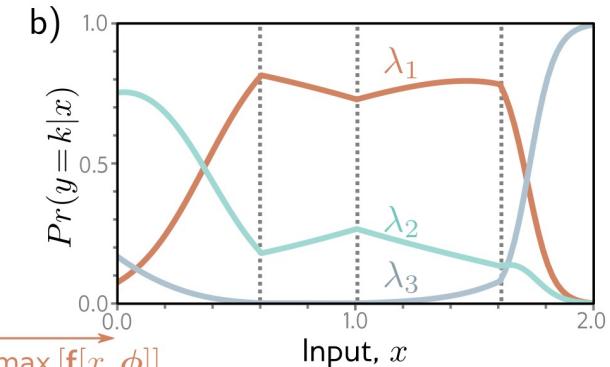
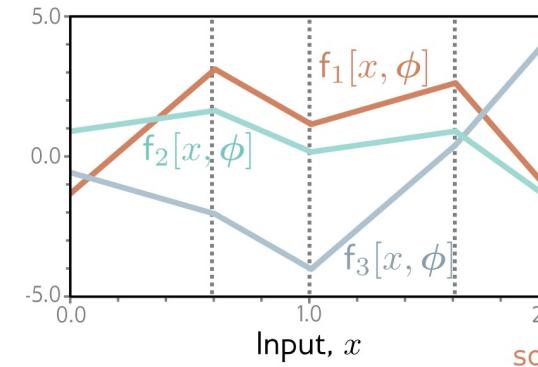
Multi-class classification: Output



- Softmax vector activation is often used at the output of multi-class classifier nets

$$z_i = \sum_j w_{ji}^{(l)} a_j^{(n-1)}$$

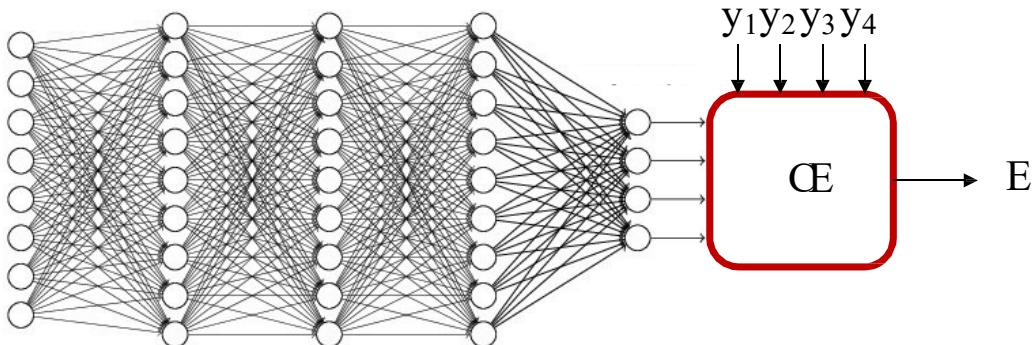
$$o_i = \frac{\exp(z_i)}{\sum_j \exp(z_j)}$$



[Understanding Deep Learning, 2023]

- This can be viewed as the probability $o_i = P(\text{class} = i | x)$

For multi-class classification



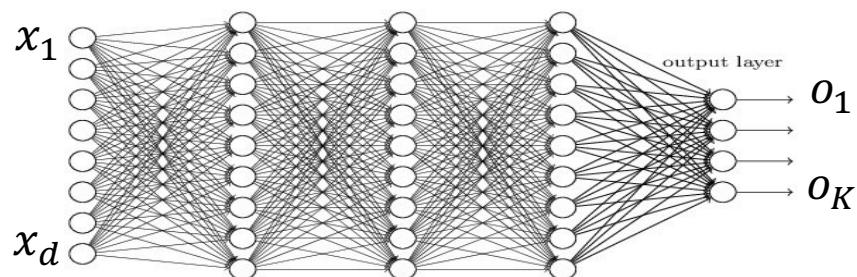
- Desired output y is one-hot vector $[0 \ 0 \dots 1 \ \dots 0 \ 0 \ 0]^T$ with the 1 in the c -th position (for class c)
- Actual output will be probability distribution $[o_1, o_2, \dots, o_l]^T$
- The cross-entropy between the desired one-hot output and actual class c

$$loss(y, o) = - \sum_i y_i \log o_i = - \log o_c$$

Choosing cost function: Examples

- Regression problem

- SSE



$$E = \sum_{n=1}^N E_n$$

$$E_n = \frac{1}{2} \left\| \mathbf{o}^{(n)} - \mathbf{y}^{(n)} \right\|^2 = \sum_{k=1}^K \left(o_k^{(n)} - y_k^{(n)} \right)^2 \quad \text{Multi-dimensional output}$$

- Classification problem

- Cross-entropy
 - Binary classification

$$\text{loss}_n = -y^{(n)} \log o^{(n)} - (1 - y^{(n)}) \log(1 - o^{(n)}) \quad o = \sigma(z) = \frac{1}{1 + e^{-z}}$$

Output layer uses sigmoid activation function

$$\text{loss}_n = -\log o_{y^{(n)}}$$

Output is found by a softmax layer $o_i = \frac{e^{z_i}}{\sum_{j=1}^K e^{z_j}}$

Likelihood Objective

- Likelihood

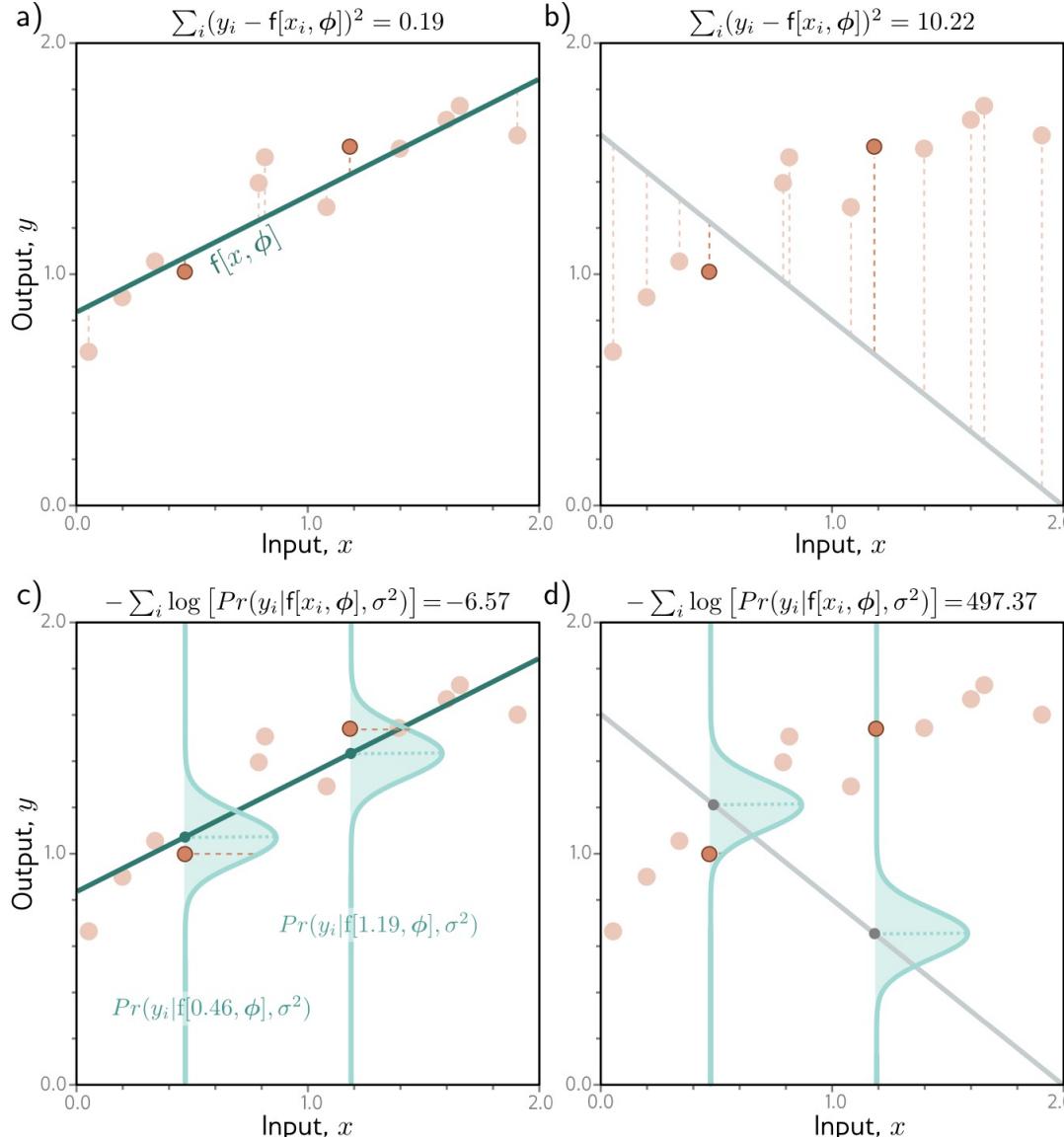
$$p(y^{(1)}, y^{(2)}, \dots, y^{(N)} | x^{(1)}, x^{(2)}, \dots, x^{(N)}) = \prod_{n=1}^N p(y^{(n)} | x^{(n)}) = \prod_{n=1}^N p(y^{(n)} | f(x^{(n)}; W))$$

- Log-likelihood

$$\log \prod_{n=1}^N p(y^{(n)} | f(x^{(n)}; W)) = \sum_{n=1}^N \log p(y^{(n)} | f(x^{(n)}; W))$$

- $-\log p(y^{(n)} | f(x^{(n)}; W))$ corresponds to the loss function

Probabilistic Viewpoint of Sum of Squares Error (SSE)



$$p(y|f(x; \phi)) = N(y| f(x; \phi), \sigma^2) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{1}{2\sigma^2}(y-f(x;\phi))^2}$$

$$\log p(y|f(x; \phi)) = -\log \sqrt{2\pi} \sigma - \frac{1}{2\sigma^2} (y - f(x; \phi))^2$$

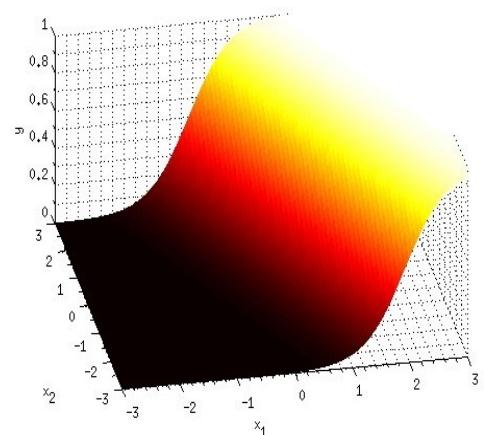
$$\begin{aligned} & \underset{\phi}{\operatorname{argmax}} \sum_{n=1}^N \log p(y^{(n)} | f(x^{(n)}; \phi)) \\ &= \underset{\phi}{\operatorname{argmin}} \sum_{n=1}^N (y^{(n)} - f(x^{(n)}; \phi))^2 \end{aligned}$$

Probabilistic Viewpoint of Cost Functions

- Binary classification:

$$Bern(y|\theta) = \theta^y(1-\theta)^{1-y} \quad \theta = P(Y=1)$$

$$\begin{aligned}-\log p(y|f(x; W)) &= -\log Bern(y|f(x; W)) \\&= -\log f(x; W)^{y(n)}(1-f(x; W))^{1-y(n)} \\&= -y(n)\log f(x; W) - (1-y(n))\log(1-f(x; W))\end{aligned}$$



- Multi-class classification:

$$Multi(y|\theta) = \prod_{k=1}^K \theta_k^{y_k} \quad \theta_k = P(y_k=1)$$

$$-\log p(y|f(x; W)) = -\log Multi(y|f(x; W)) = -\sum_{k=1}^K y_k^{(n)} \log f_k(x; W)$$

Cross Entropy Loss

$$D_{KL}[q||p] = \int q(z) \log \frac{q(z)}{p(z)} dz = \int q(z) \log q(z) dz - \int q(z) \log p(z) dz$$

- For multi-class classification problem:
 - $q(y|x)$ is a one-hot vector shows the target class of x
 - $p_\theta(y|x)$ shows the output of the parametric model

$$D_{KL}[q||p] = \sum_z q(z) \log \frac{q(z)}{p(z)}$$

$$\operatorname{argmin}_\theta D_{KL}[q||p_\theta] = - \sum_{k=1}^K y_k \log p_\theta(y = k|x)$$

Problem setup

- Given: the architecture of the network
- Training data: A set of input-output pairs
$$(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \dots, (x^{(N)}, y^{(N)})$$
- We need a **loss function** to show how penalizes the obtained output $o = f(x; W)$ when the desired output is y

$$E(W) = \sum_{n=1}^N loss(o^{(n)}, y^{(n)}) = \frac{1}{N} \sum_{n=1}^N loss(f(x^{(n)}; W), y^{(n)})$$

- Minimize E w.r.t. W that contains $\{w_{i,j}^{[k]}, b_j^{[k]}\}$

Recap: Iterative optimization of cost function

- Cost function: $E(\mathbf{w})$
- Optimization problem: $\hat{\mathbf{w}} = \underset{\mathbf{w}}{\operatorname{argmin}} E(\mathbf{w})$
- Steps:
 - Start from \mathbf{w}^0
 - Repeat
 - Update \mathbf{w}^t to \mathbf{w}^{t+1} in order to reduce E
 - $t \leftarrow t + 1$
 - until we hopefully end up at a minimum

Recap: How to compute the slope?

- In 1-dimension, the derivative of a function:

$$\frac{df(x)}{dx} = \lim_{h \rightarrow 0} \frac{f(x + h) - f(x)}{h}$$

- the slope of the error surface can be calculated by taking the derivative of the error function at that point
- In multiple dimensions, the **gradient** is the vector of (partial derivatives) along each dimension
- The direction of steepest descent is the **negative gradient**

Recap: Gradient descent (or steepest descent)

- In each step, takes steps proportional to the negative of the gradient vector of the function at the current point \mathbf{w}^t :

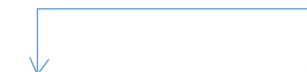
$$\mathbf{w}^{t+1} = \mathbf{w}^t - \gamma_t \nabla J(\mathbf{w}^t)$$

- $J(\mathbf{w})$ decreases fastest if one goes from \mathbf{w}^t in the direction of $-\nabla J(\mathbf{w}^t)$
- Assumption: $J(\mathbf{w})$ is defined and differentiable in a neighborhood of a point \mathbf{w}^t

Learning rate: The amount of time he travels before taking another measurement is the learning rate of the algorithm.

Recap: Gradient descent

- Minimize $J(\mathbf{w})$

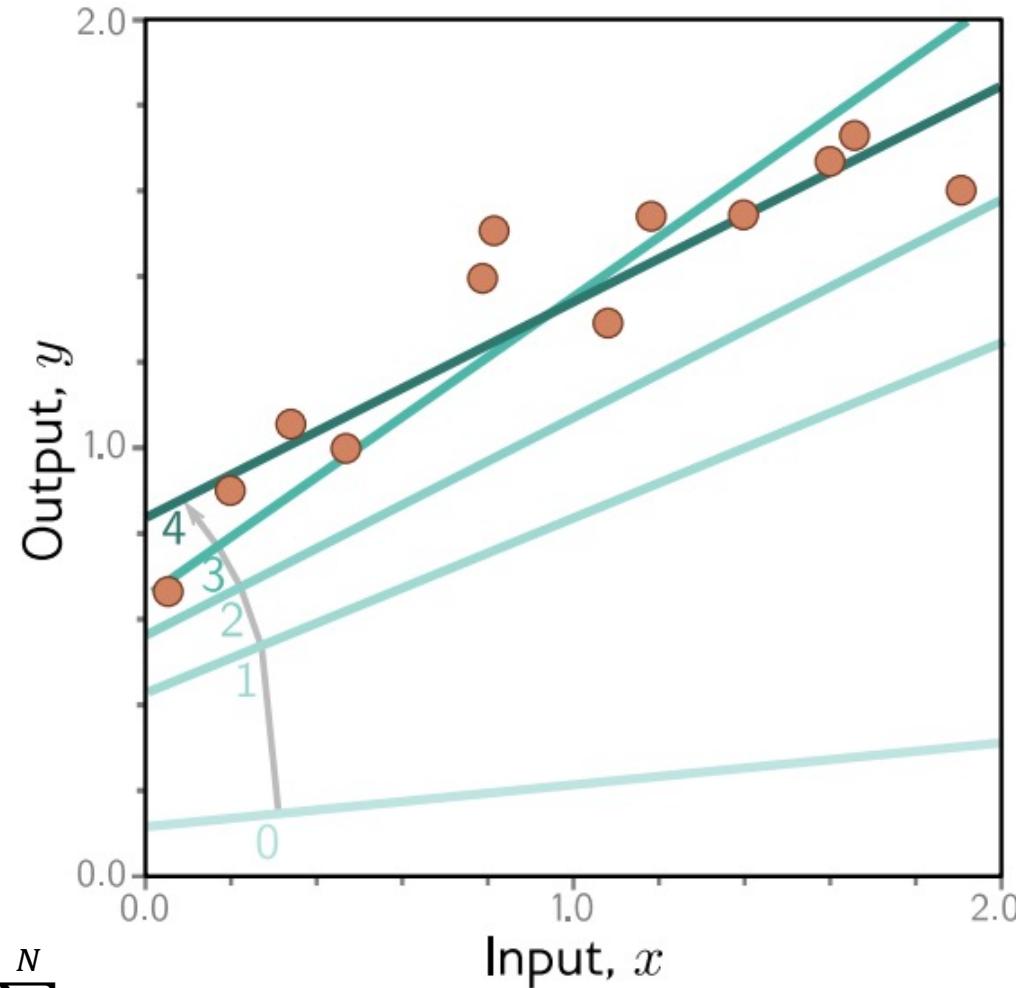
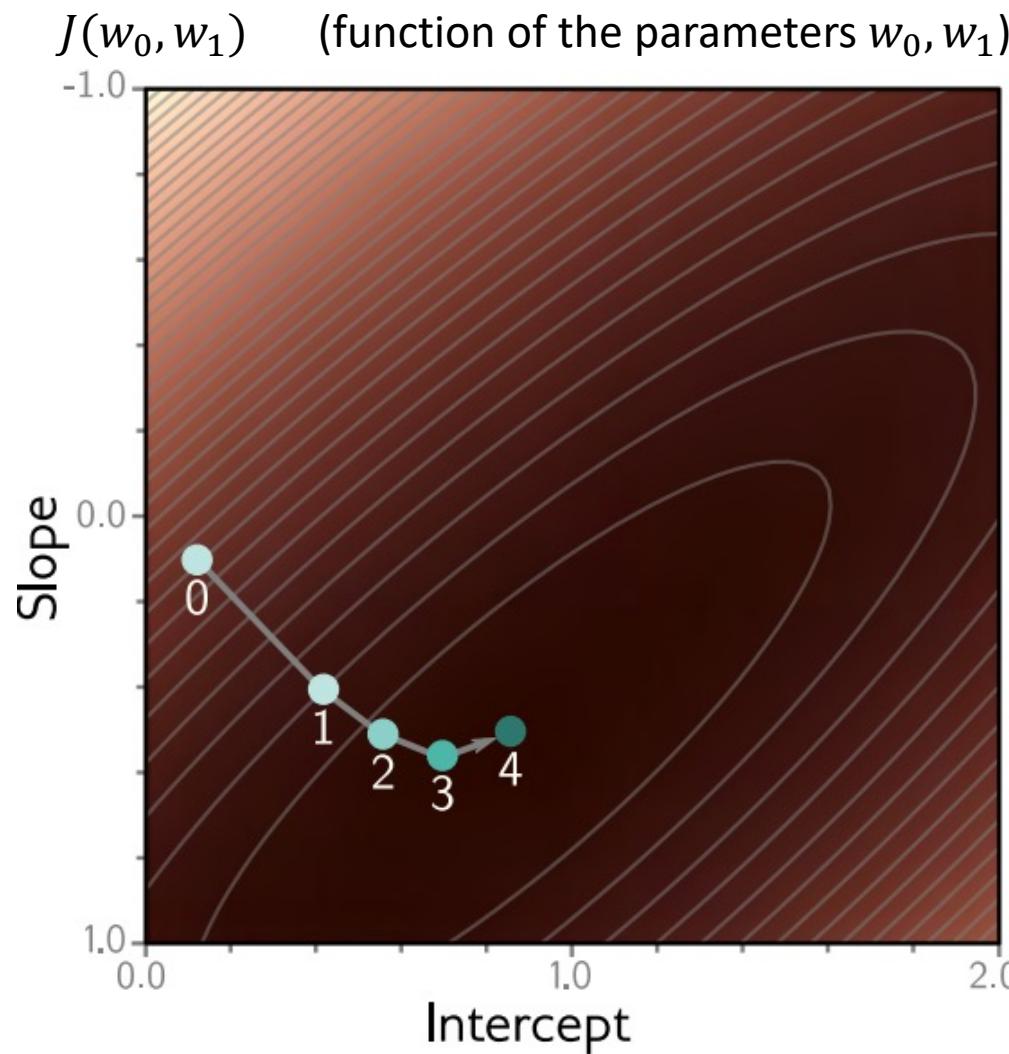
 Step size
(Learning rate parameter)

$$\mathbf{w}^{t+1} = \mathbf{w}^t - \eta \nabla_{\mathbf{w}} J(\mathbf{w}^t)$$

$$\nabla_{\mathbf{w}} J(\mathbf{w}) = \left[\frac{\partial J(\mathbf{w})}{\partial w_0}, \frac{\partial J(\mathbf{w})}{\partial w_2}, \dots, \frac{\partial J(\mathbf{w})}{\partial w_d} \right]^T$$

- If η is small enough, then $J(\mathbf{w}^{t+1}) \leq J(\mathbf{w}^t)$.
- η can be allowed to change at every iteration as η_t .

Recap: SSE optimization by gradient descent



$$\mathbf{w}^{t+1} = \mathbf{w}^t - \eta \sum_{i=1}^N (\mathbf{w}^{tT} \mathbf{x}^{(i)} - y^{(i)}) \mathbf{x}^{(i)}$$

How to adjust weights for multi layer networks?

- We need multiple layers of adaptive, non-linear hidden units. But how can we train such nets?
 - We need an efficient way of adapting all the weights, not just the last layer.
 - Learning the weights going into hidden units is equivalent to learning features.
 - This is difficult because nobody is telling us directly what the hidden units should do.

Training multi-layer networks

- Back-propagation
 - Training algorithm that is used to adjust weights in multi-layer networks (based on the training data)
 - The back-propagation algorithm is based on gradient descent
 - Use chain rule and dynamic programming to efficiently compute gradients

Training Neural Nets through Gradient Descent

Total training error:

$$E = \sum_{n=1}^N loss(\mathbf{o}^{(n)}, \mathbf{y}^{(n)})$$

- Gradient descent algorithm
- Initialize all weights and biases $\{w_{ij}^{[k]}\}$
 - Assuming the bias is also represented as a weight
- Do :
 - For every layer k for all i, j update:
 - $w_{i,j}^{[k]} = w_{i,j}^{[k]} - \eta \frac{dE}{dw_{i,j}^{[k]}}$
 - Until E has converged

The derivative

Total training error:

$$E = \sum_{n=1}^N loss(\mathbf{o}^{(n)}, \mathbf{y}^{(n)})$$

- Computing the derivative

Total derivative:

$$\frac{dE}{dw_{i,j}^{[k]}} = \sum_{n=1}^N \frac{loss(\mathbf{o}^{(n)}, \mathbf{y}^{(n)})}{dw_{i,j}^{[k]}}$$

Training by gradient descent

- Initialize all weights $\{w_{ij}^{[k]}\}$
- Do :
 - For all i, j, k , initialize $\frac{dE}{dw_{i,j}^{[k]}} = 0$
 - For all $n = 1: N$
 - For every layer k for all i, j :
 - Compute $\frac{d \text{loss}(o^{(n)}, y^{(n)})}{dw_{i,j}^{[k]}}$
 - $\frac{dE}{dw_{i,j}^{[k]}} += \frac{d \text{loss}(o^{(n)}, y^{(n)})}{dw_{i,j}^{[k]}}$
 - For every layer k for all i, j :

$$w_{i,j}^{[k]} = w_{i,j}^{[k]} - \frac{\eta}{N} \frac{dE}{dw_{i,j}^{[k]}}$$

The derivative

Total training error:

$$E = \sum_{n=1}^N loss(\mathbf{o}^{(n)}, \mathbf{y}^{(n)})$$

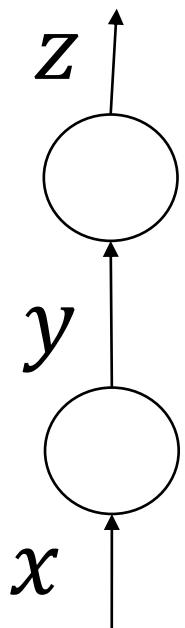
Total derivative:

$$\frac{dE}{dw_{i,j}^{[k]}} = \sum_{n=1}^N \frac{d loss(\mathbf{o}^{(n)}, \mathbf{y}^{(n)})}{dw_{i,j}^{[k]}}$$

- So we must first figure out how to compute the derivative of divergences of individual training inputs

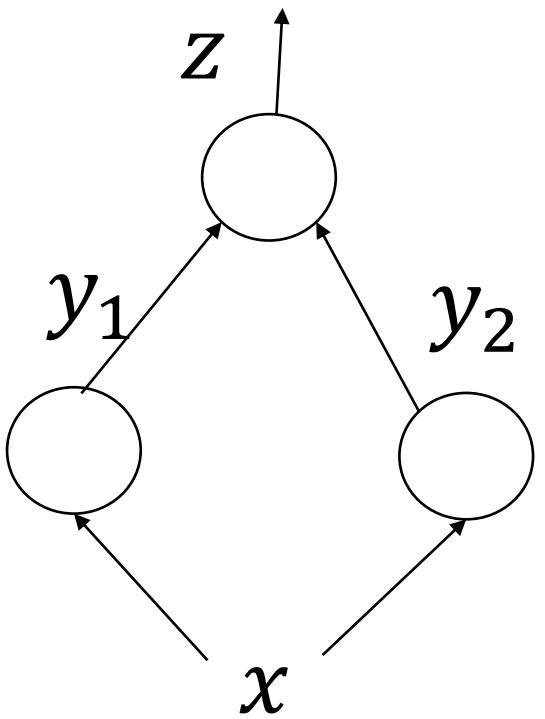
Simple chain rule

- $z = f(g(x))$
- $y = g(x)$



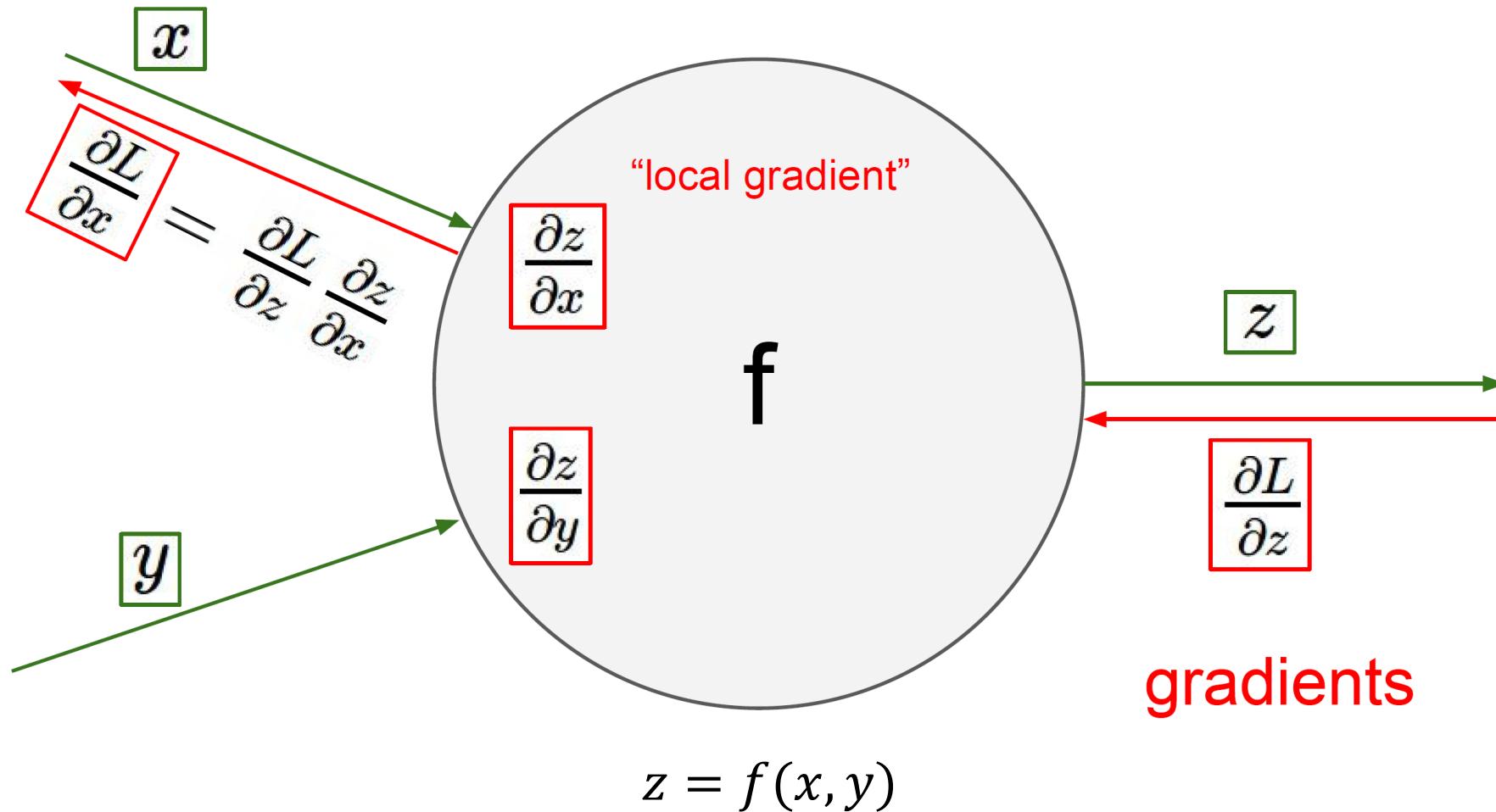
$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y} \frac{\partial y}{\partial x}$$

Multiple paths chain rule

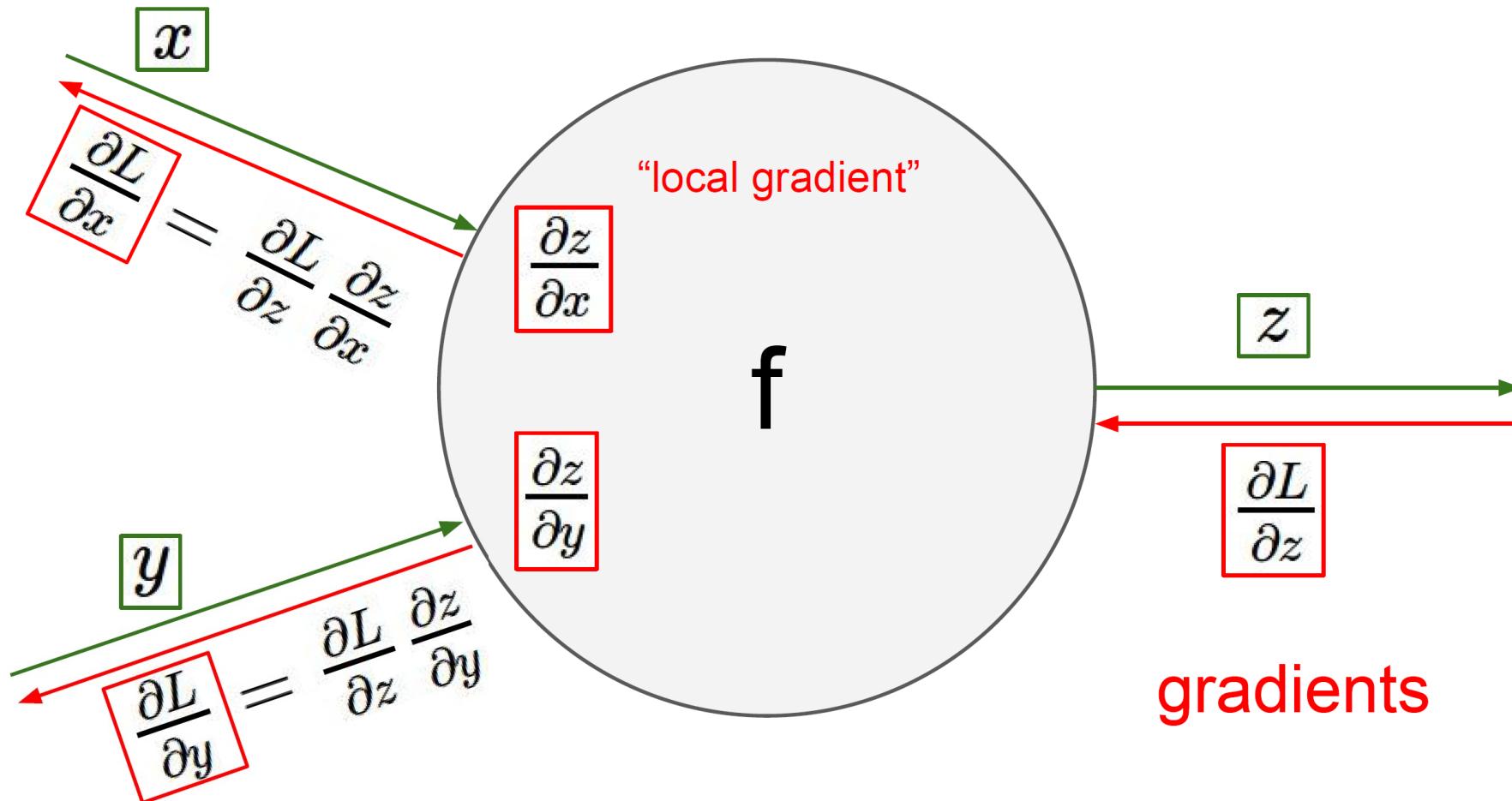


$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y_1} \frac{\partial y_1}{\partial x} + \frac{\partial z}{\partial y_2} \frac{\partial y_2}{\partial x}$$

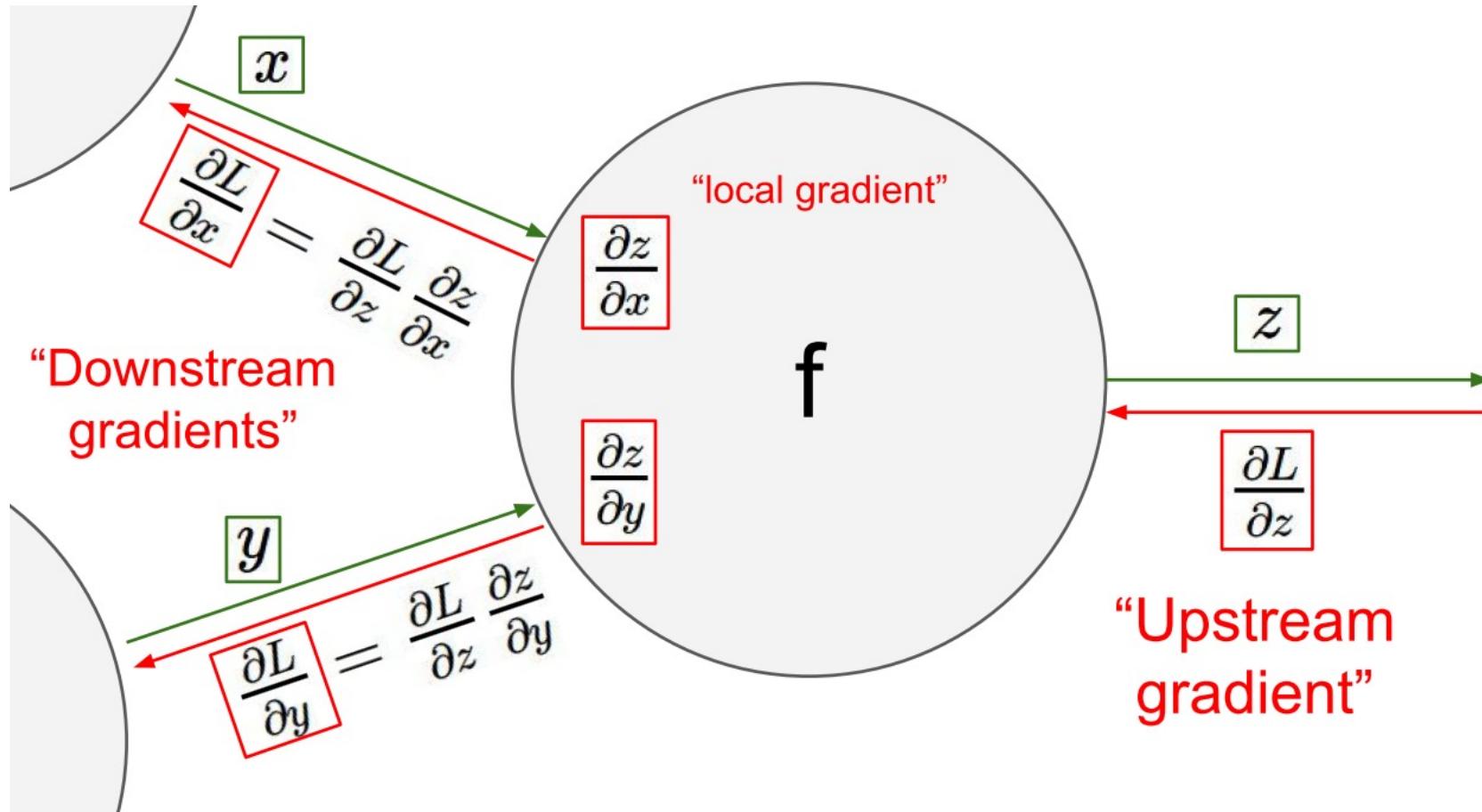
How to propagate the gradients backward



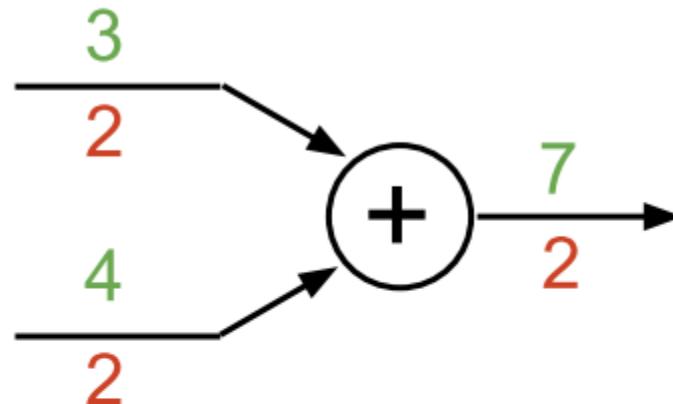
How to propagate the gradients backward



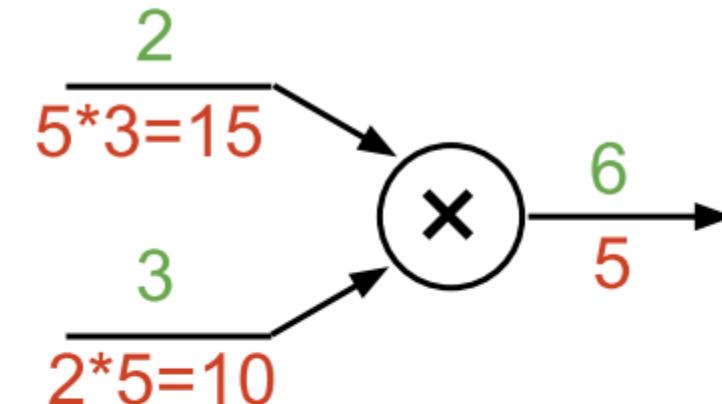
Downstream gradients from upstream ones



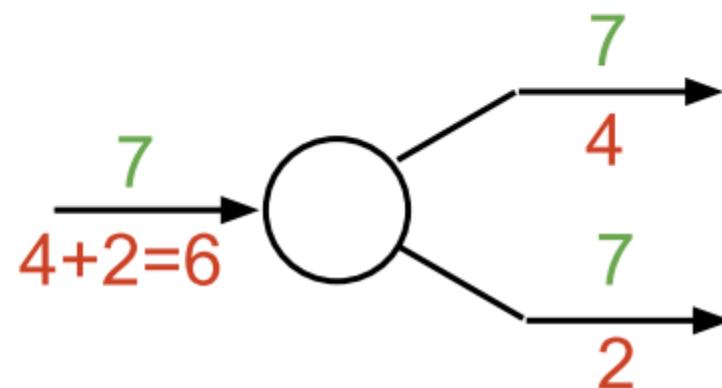
add gate: gradient distributor



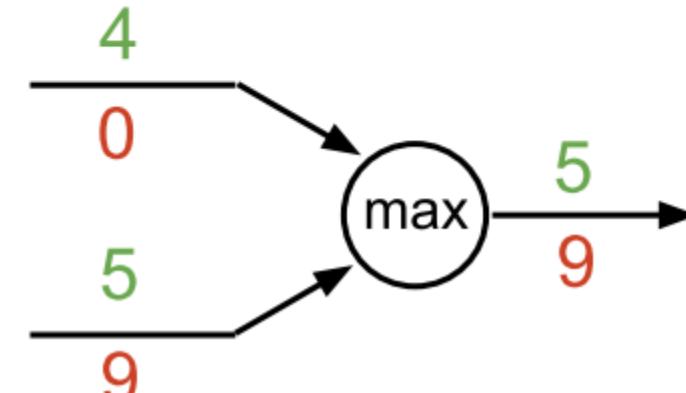
mul gate: “swap multiplier”



copy gate: gradient adder



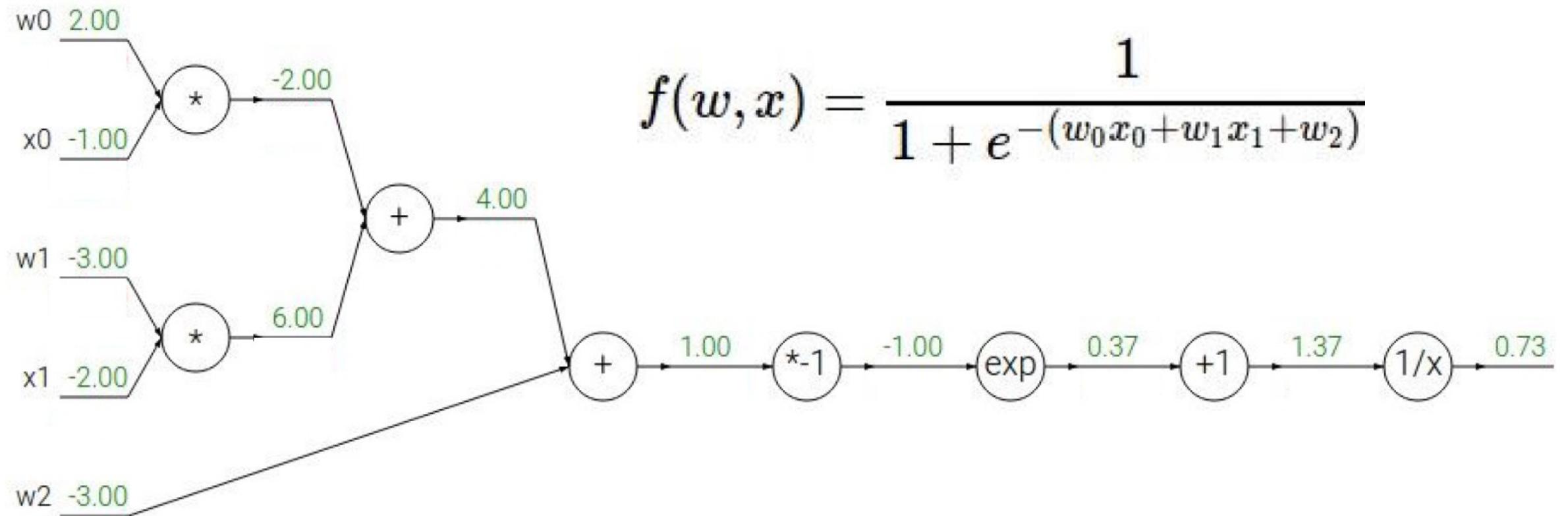
max gate: gradient router



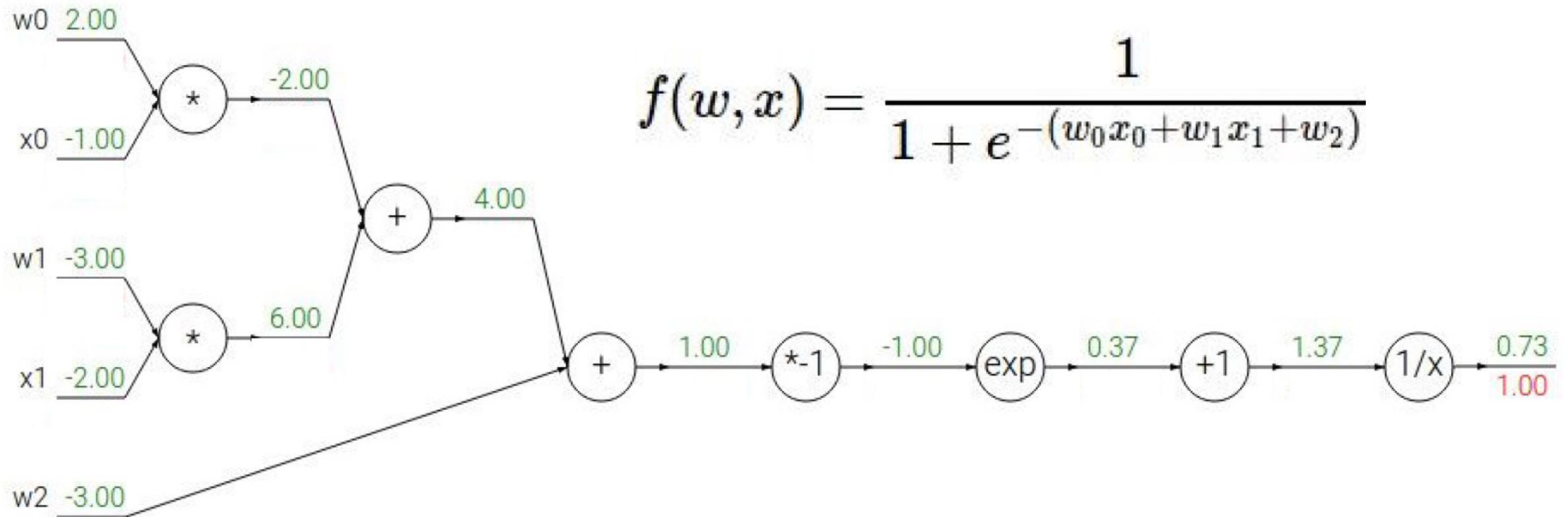
Returning to our problem

- How to compute $\frac{d \text{ loss}(\mathbf{o}, \mathbf{y})}{d w_{i,j}^{[k]}}$

Another example



Another example



$$f(x) = e^x$$

→

$$\frac{df}{dx} = e^x$$

$$f_a(x) = ax$$

→

$$\frac{df}{dx} = a$$

$$f(x) = \frac{1}{x}$$

→

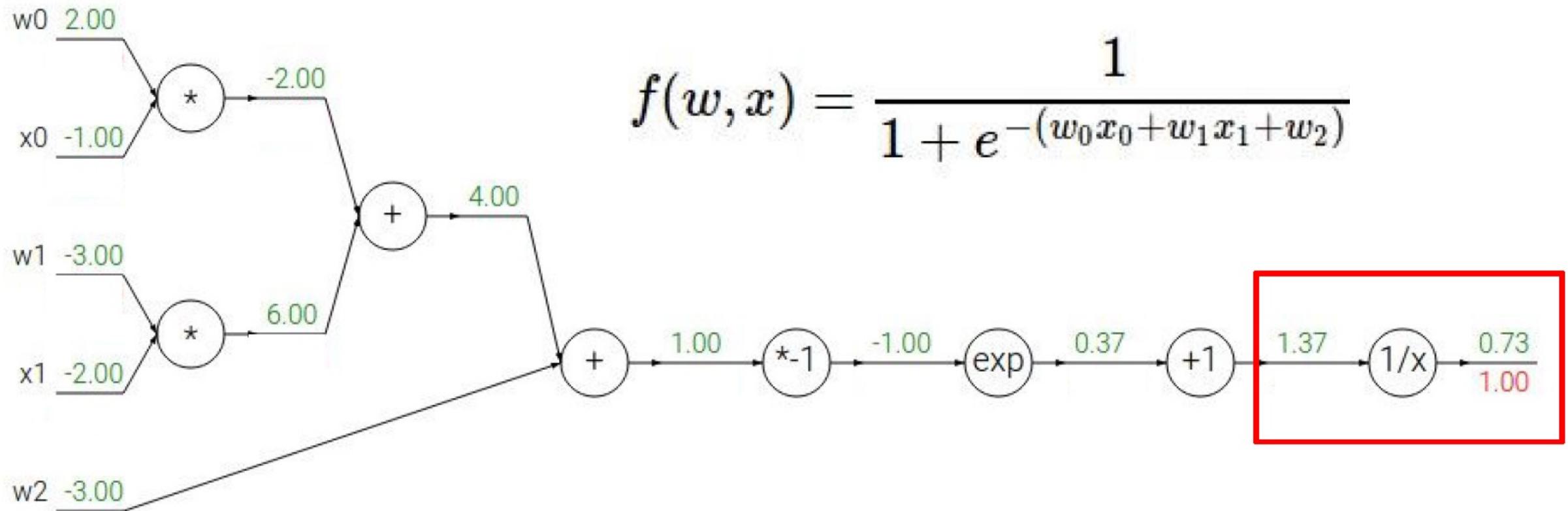
$$\frac{df}{dx} = -1/x^2$$

$$f_c(x) = c + x$$

→

$$\frac{df}{dx} = 1$$

Another example



$$f(x) = e^x$$

→

$$\frac{df}{dx} = e^x$$

$$f_a(x) = ax$$

→

$$\frac{df}{dx} = a$$

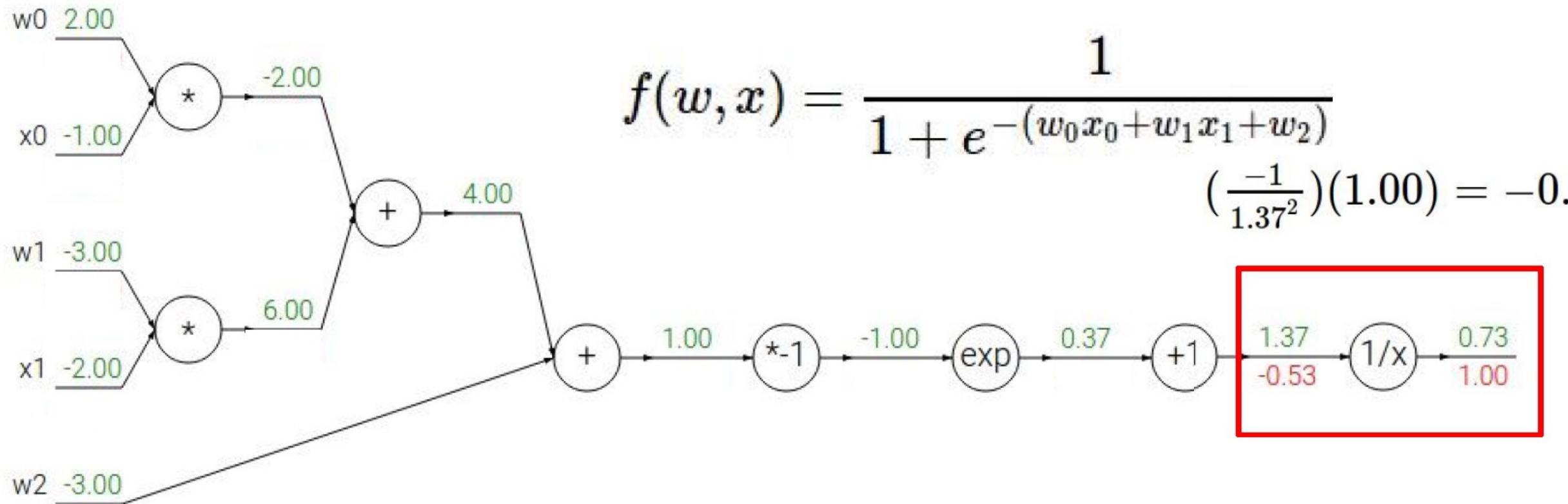
$$f(x) = \frac{1}{x}$$

$$f_c(x) = c + x$$

$$\frac{df}{dx} = -1/x^2$$

$$\frac{df}{dx} = 1$$

Another example



$$f(x) = e^x$$

\rightarrow

$$\frac{df}{dx} = e^x$$

$$f_a(x) = ax$$

\rightarrow

$$\frac{df}{dx} = a$$

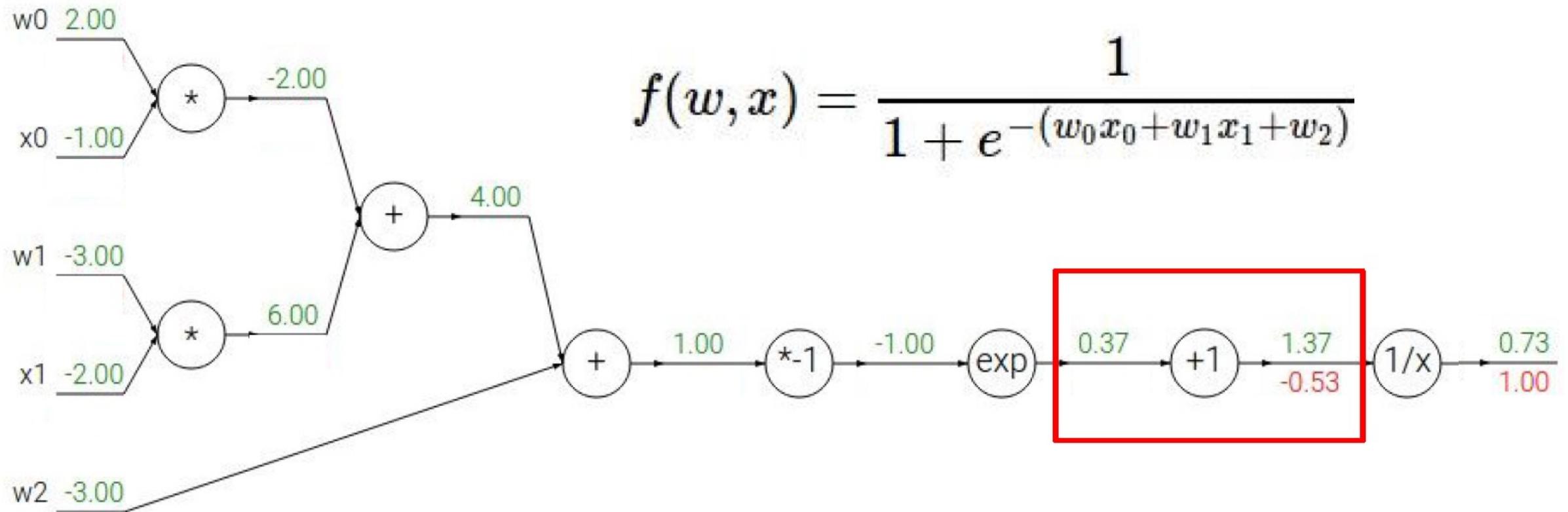
$$f(x) = \frac{1}{x}$$

$$f_c(x) = c + x$$

$$\frac{df}{dx} = -1/x^2$$

$$\frac{df}{dx} = 1$$

Another example



$$f(x) = e^x$$

→

$$\frac{df}{dx} = e^x$$

$$f_a(x) = ax$$

→

$$\frac{df}{dx} = a$$

$$f(x) = \frac{1}{x}$$

→

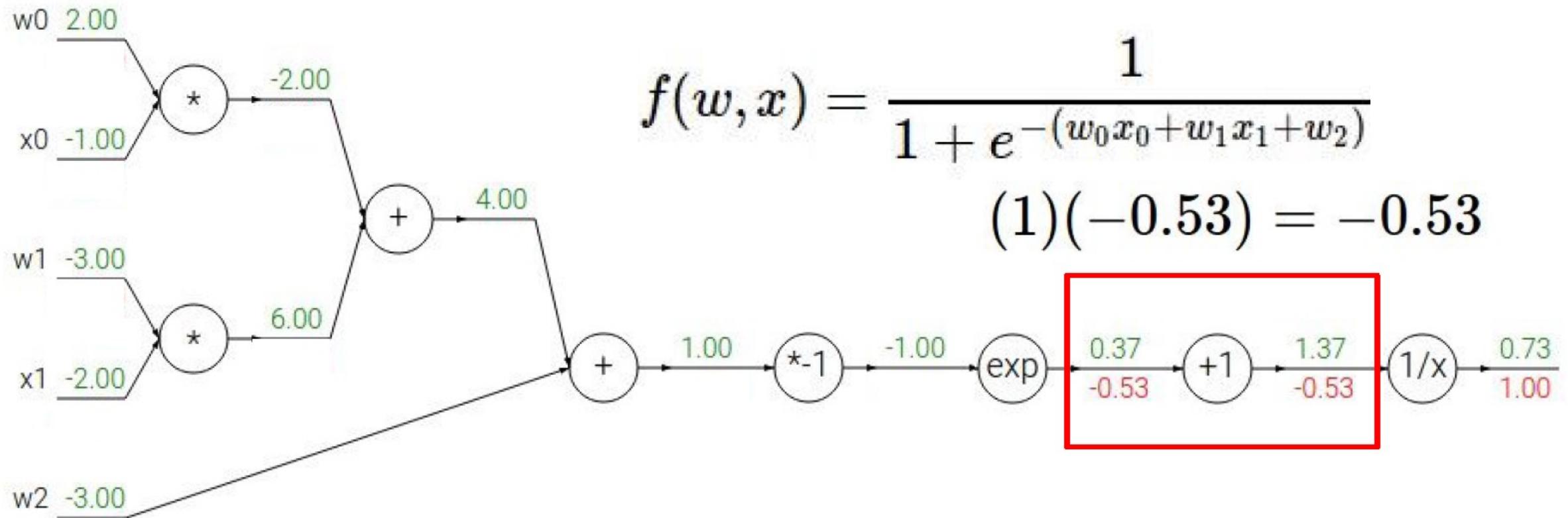
$$\frac{df}{dx} = -1/x^2$$

$$f_c(x) = c + x$$

→

$$\frac{df}{dx} = 1$$

Another example



$$f(x) = e^x$$

\rightarrow

$$\frac{df}{dx} = e^x$$

$$f_a(x) = ax$$

\rightarrow

$$\frac{df}{dx} = a$$

$$f(x) = \frac{1}{x}$$

\rightarrow

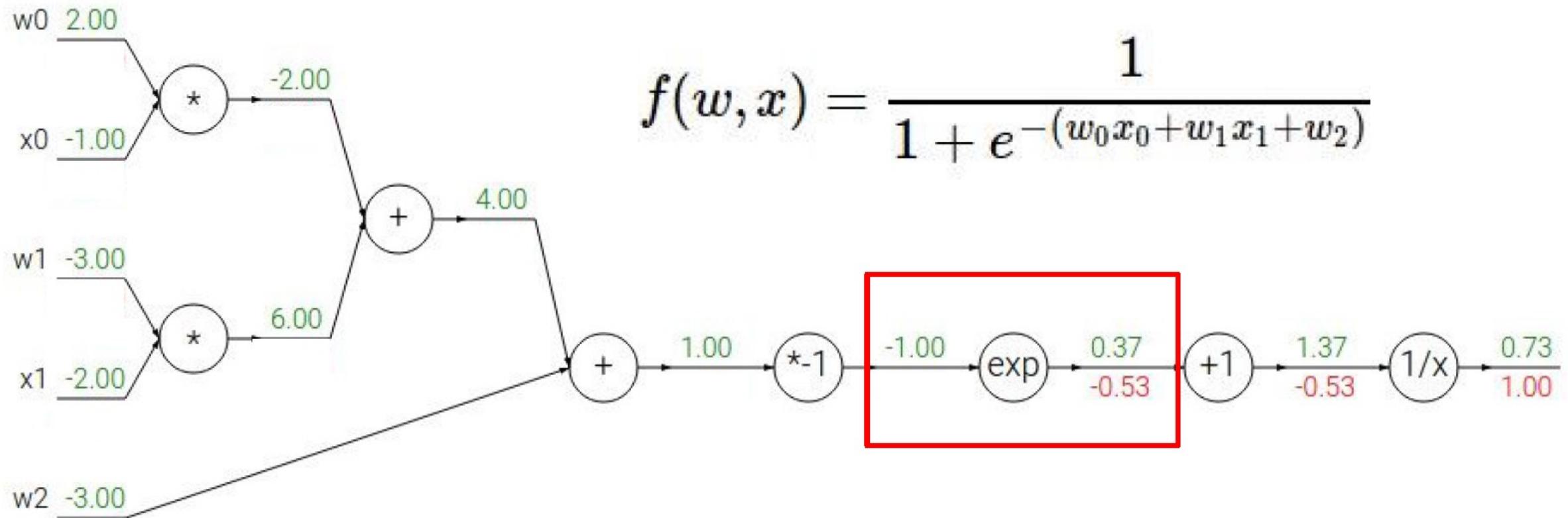
$$\frac{df}{dx} = -1/x^2$$

$$f_c(x) = c + x$$

\rightarrow

$$\frac{df}{dx} = 1$$

Another example



$$f(x) = e^x$$

→

$$\frac{df}{dx} = e^x$$

$$f_a(x) = ax$$

→

$$\frac{df}{dx} = a$$

$$f(x) = \frac{1}{x}$$

→

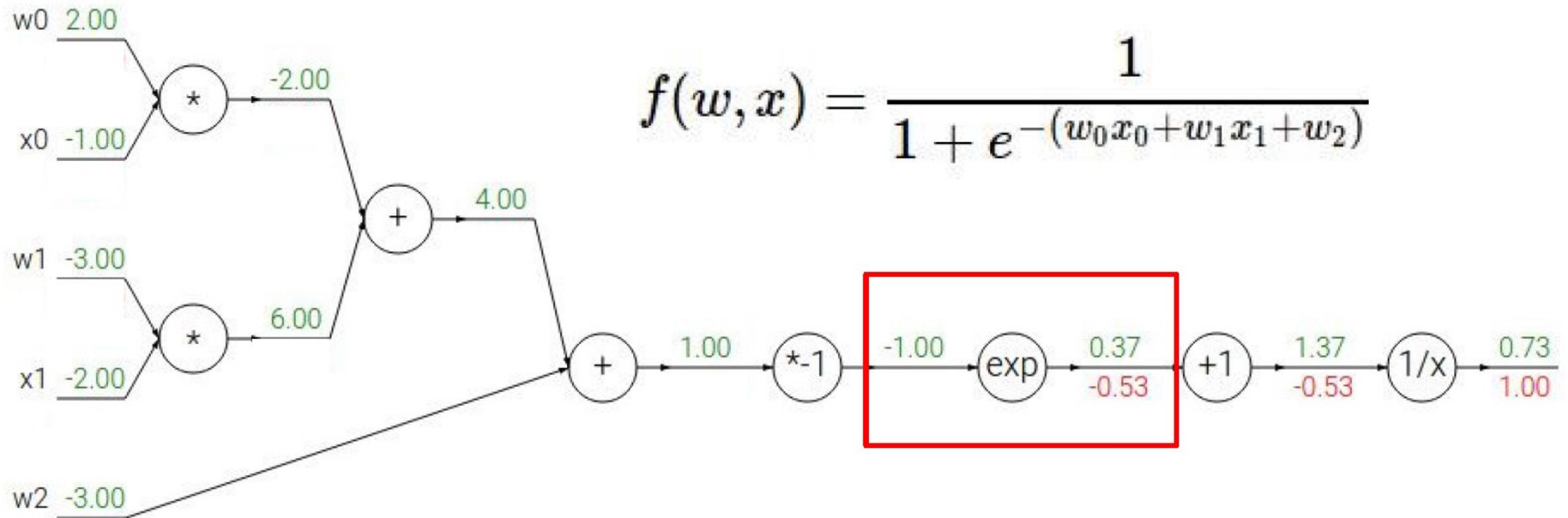
$$\frac{df}{dx} = -1/x^2$$

$$f_c(x) = c + x$$

→

$$\frac{df}{dx} = 1$$

Another example



$$f(x) = e^x$$

\rightarrow

$$\frac{df}{dx} = e^x$$

$$f_a(x) = ax$$

\rightarrow

$$\frac{df}{dx} = a$$

$$f(x) = \frac{1}{x}$$

\rightarrow

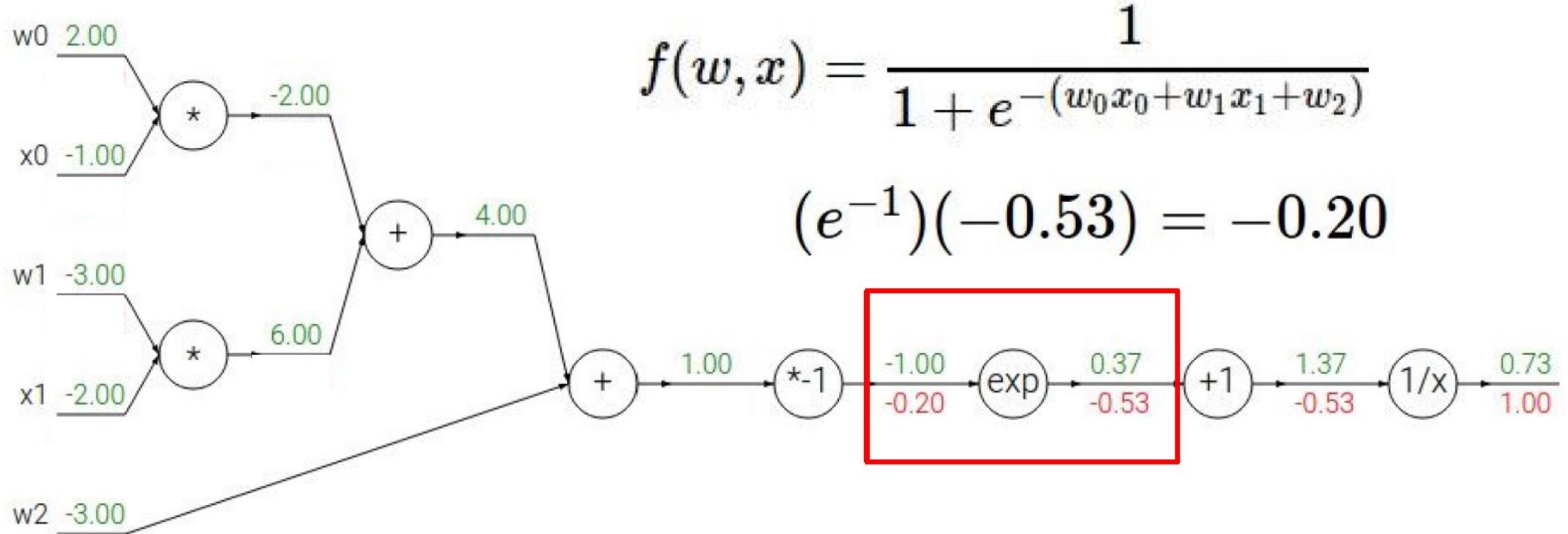
$$\frac{df}{dx} = -1/x^2$$

$$f_c(x) = c + x$$

\rightarrow

$$\frac{df}{dx} = 1$$

Another example



$$f(x) = e^x$$

→

$$\frac{df}{dx} = e^x$$

$$f_a(x) = ax$$

→

$$\frac{df}{dx} = a$$

$$f(x) = \frac{1}{x}$$

→

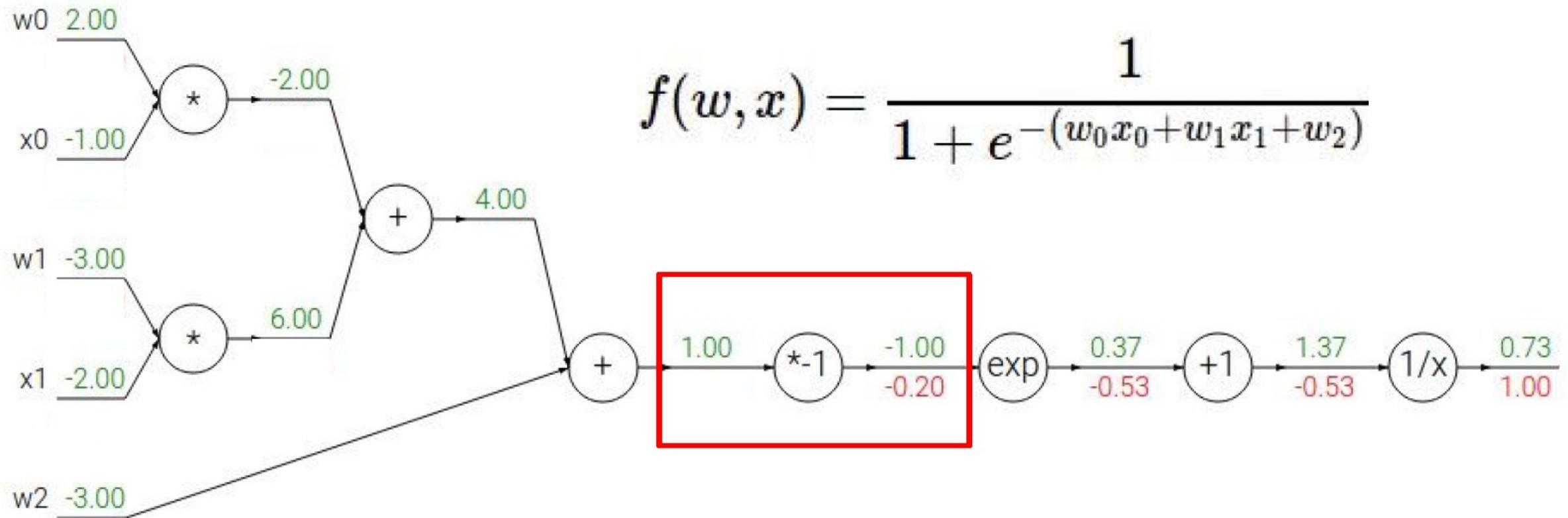
$$\frac{df}{dx} = -1/x^2$$

$$f_c(x) = c + x$$

→

$$\frac{df}{dx} = 1$$

Another example



$$f(x) = e^x$$

→

$$\frac{df}{dx} = e^x$$

$$f_a(x) = ax$$

→

$$\frac{df}{dx} = a$$

$$f(x) = \frac{1}{x}$$

→

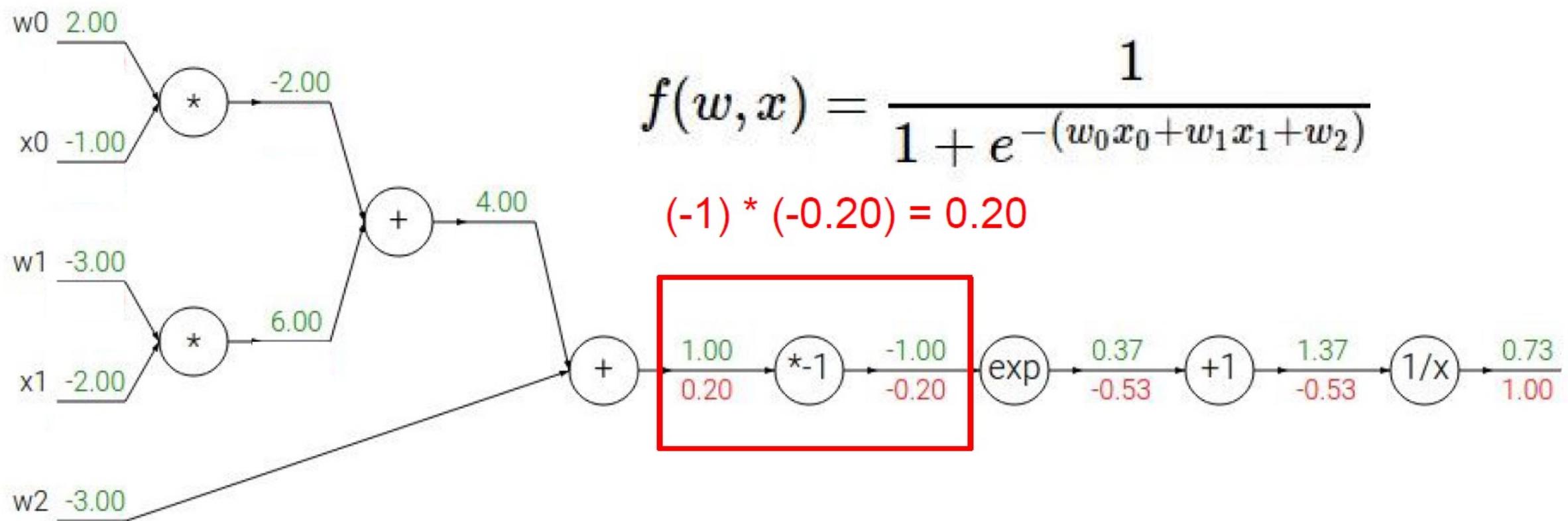
$$\frac{df}{dx} = -1/x^2$$

$$f_c(x) = c + x$$

→

$$\frac{df}{dx} = 1$$

Another example



$$f(x) = e^x$$

→

$$\frac{df}{dx} = e^x$$

$$f_a(x) = ax$$

→

$$\frac{df}{dx} = a$$

$$f(x) = \frac{1}{x}$$

→

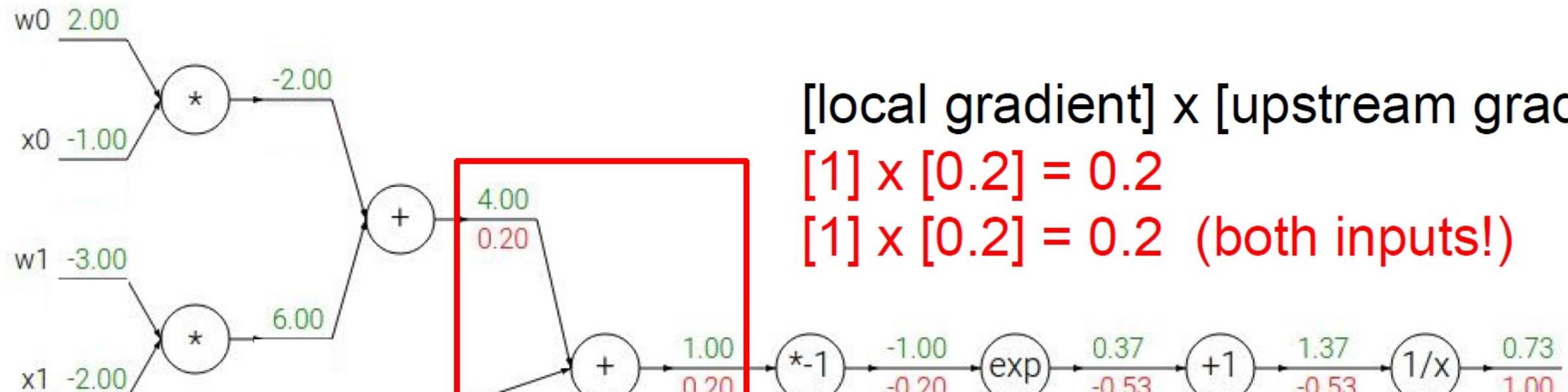
$$\frac{df}{dx} = -1/x^2$$

$$f_c(x) = c + x$$

→

$$\frac{df}{dx} = 1$$

Another example



$$w_2 \begin{matrix} -3.00 \\ 0.20 \end{matrix}$$

$$f(x) = e^x$$

\rightarrow

$$\frac{df}{dx} = e^x$$

$$f_a(x) = ax$$

\rightarrow

$$\frac{df}{dx} = a$$

$$f(x) = \frac{1}{x}$$

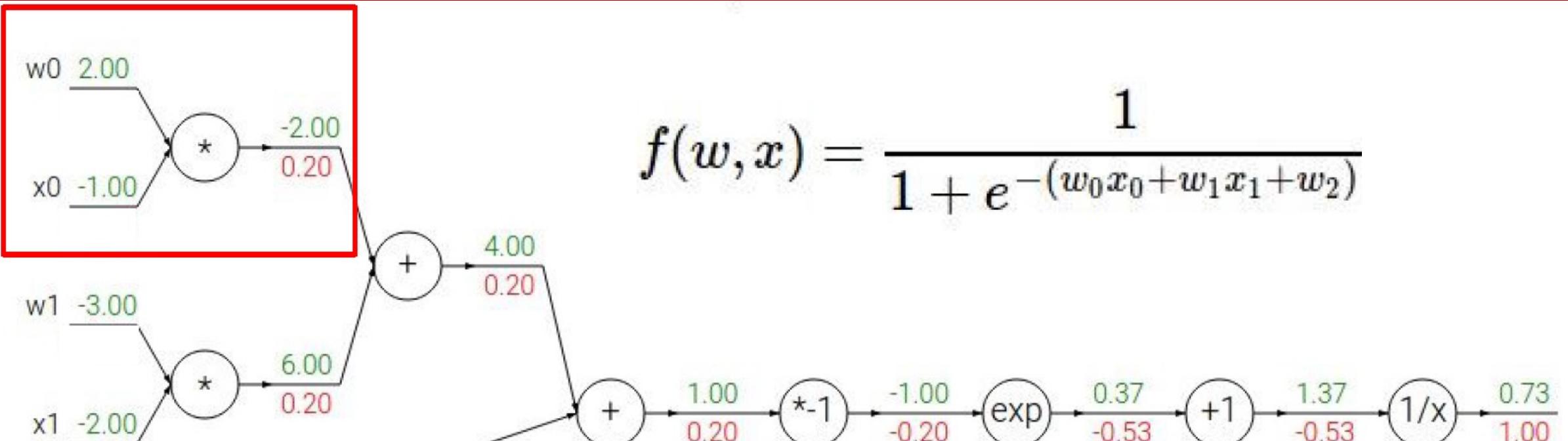
$$f_c(x) = c + x$$

\rightarrow

$$\frac{df}{dx} = -1/x^2$$

$$\frac{df}{dx} = 1$$

Another example



$$f(x) = e^x$$

→

$$\frac{df}{dx} = e^x$$

$$f_a(x) = ax$$

→

$$\frac{df}{dx} = a$$

$$f(x) = \frac{1}{x}$$

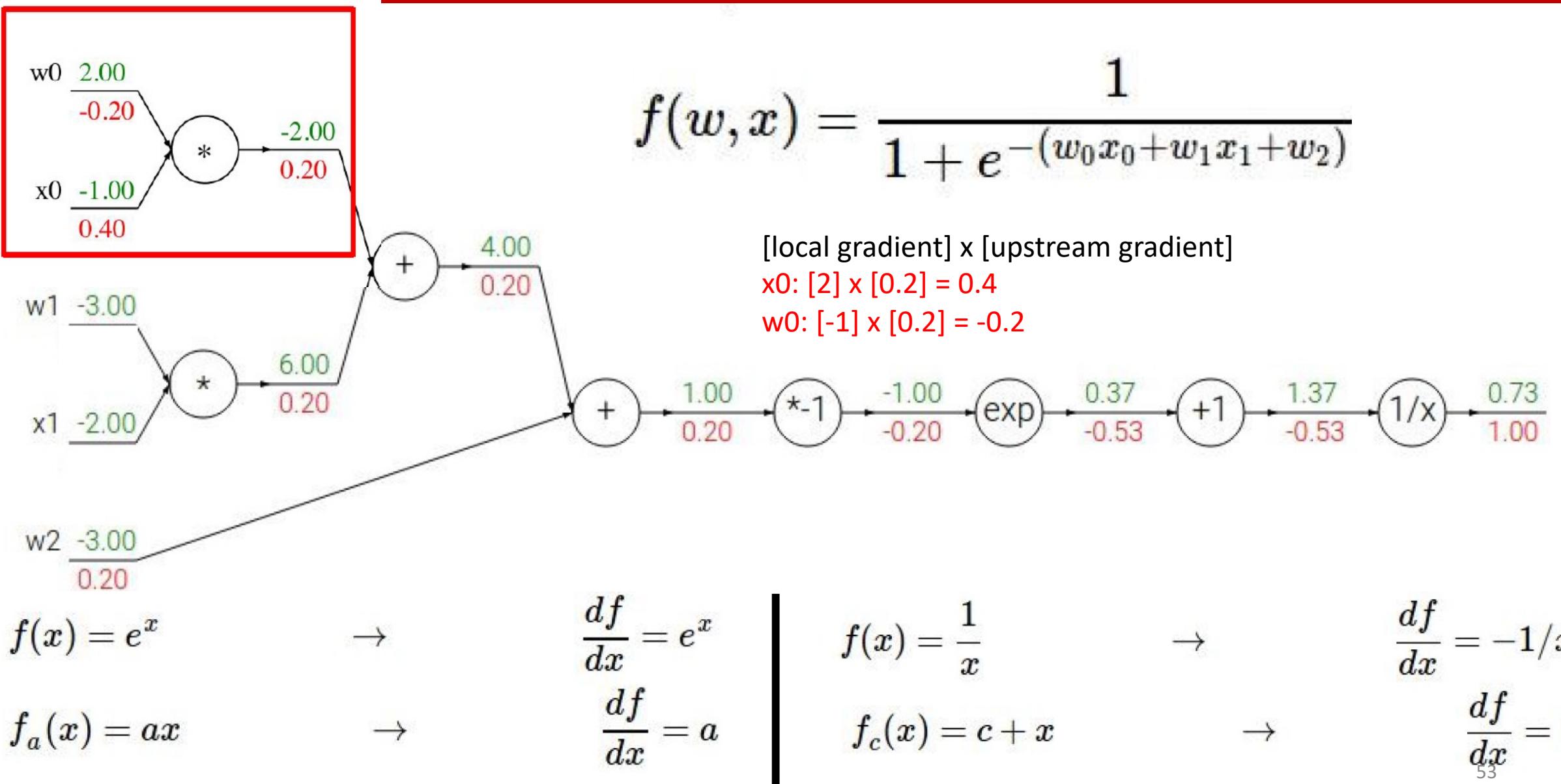
$$f_c(x) = c + x$$

→

$$\frac{df}{dx} = -1/x^2$$

$$\frac{df}{dx} = 1$$

Another example



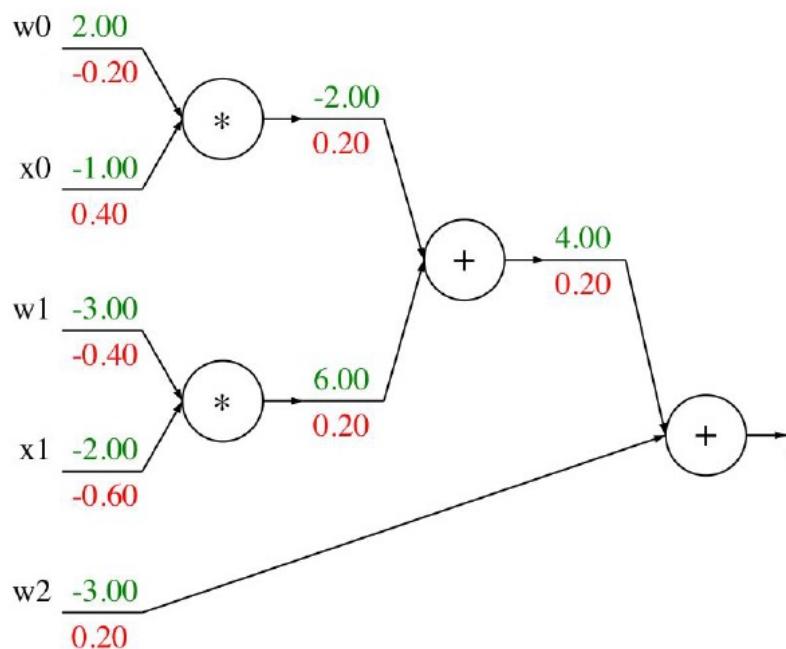
Another example: sigmoid

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

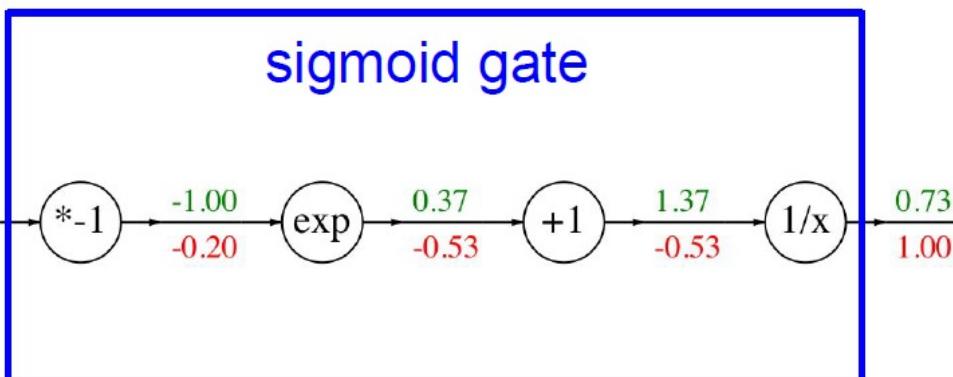
$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

sigmoid function

$$\frac{d\sigma(x)}{dx} = \frac{e^{-x}}{(1+e^{-x})^2} = \left(\frac{1+e^{-x}-1}{1+e^{-x}}\right) \left(\frac{1}{1+e^{-x}}\right) = (1-\sigma(x))\sigma(x)$$



Computational graph representation may not be unique. Choose one where local gradients at each node can be easily expressed!



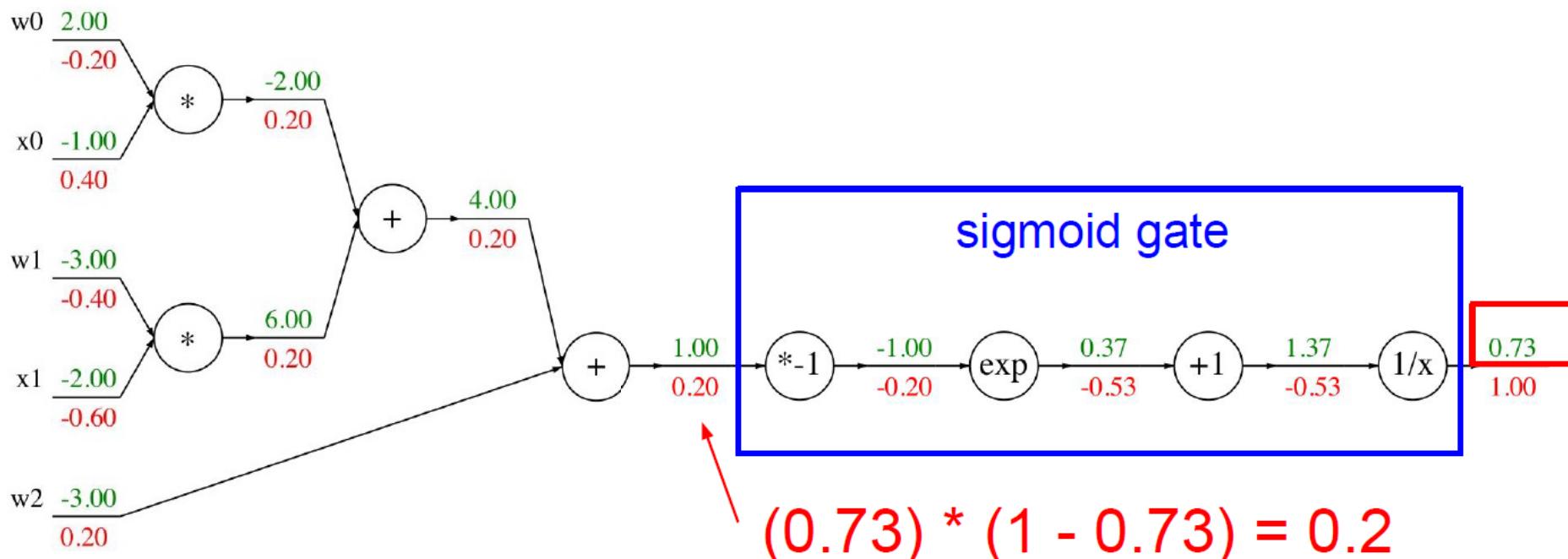
Another example: sigmoid

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

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

sigmoid function

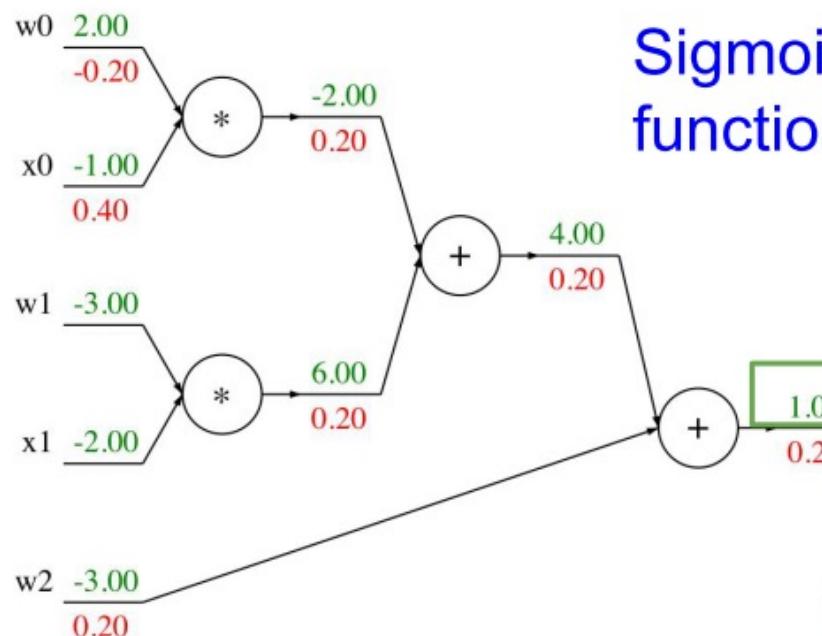
$$\frac{d\sigma(x)}{dx} = \frac{e^{-x}}{(1 + e^{-x})^2} = \left(\frac{1 + e^{-x} - 1}{1 + e^{-x}} \right) \left(\frac{1}{1 + e^{-x}} \right) = (1 - \sigma(x))\sigma(x)$$



Another example: sigmoid

Another example:

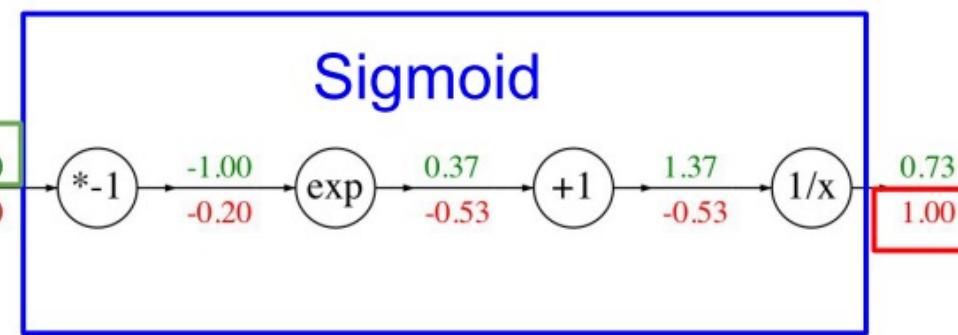
$$f(w, x) = \frac{1}{1 + e^{-(w_0x_0 + w_1x_1 + w_2)}}$$



Sigmoid
function

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

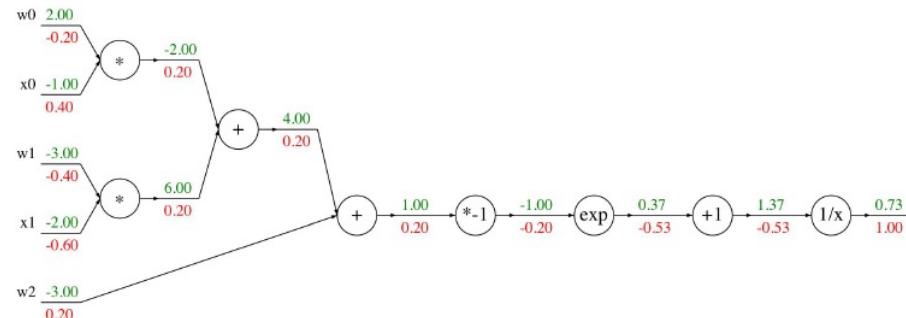
Computational graph representation may not be unique. Choose one where local gradients at each node can be easily expressed!



$$[\text{upstream gradient}] \times [\text{local gradient}]$$
$$[1.00] \times [(1 - 1/(1+e^{-1})) (1/(1+e^{-1}))] = 0.2$$

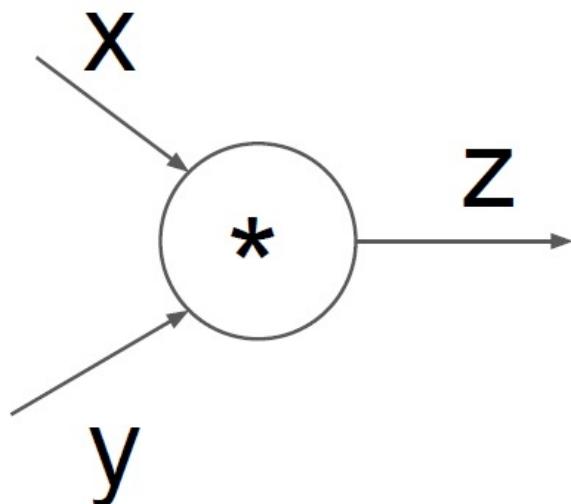
Modularized implementation: forward / backward API

Graph (or Net) object (*rough psuedo code*)



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

Modularized implementation: forward / backward API



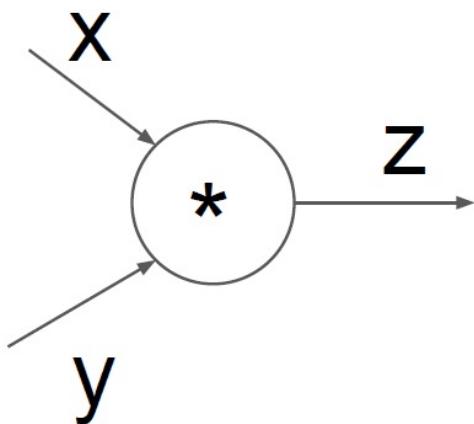
(x,y,z are scalars)

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

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

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

Modularized implementation: forward / backward API

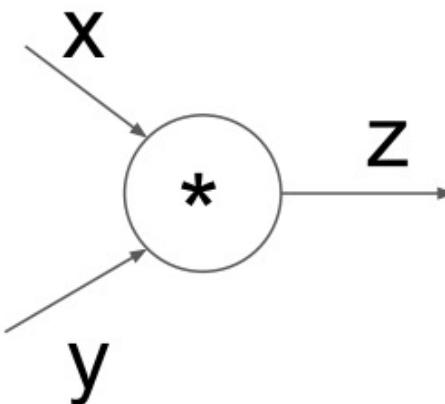


(x, y, z are scalars)

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

Modularized implementation: forward / backward API

Gate / Node / Function object: Actual PyTorch code



(x,y,z are scalars)

```
class Multiply(torch.autograd.Function):  
    @staticmethod  
    def forward(ctx, x, y):  
        ctx.save_for_backward(x, y) ← Need to cache some values for use in backward  
        z = x * y  
        return z  
  
    @staticmethod  
    def backward(ctx, grad_z): ← Upstream gradient  
        x, y = ctx.saved_tensors  
        grad_x = y * grad_z # dz/dx * dL/dz  
        grad_y = x * grad_z # dz/dy * dL/dz  
        return grad_x, grad_y
```

Need to cache some values for use in backward

Upstream gradient

Multiply upstream and local gradients

Example: PyTorch operators

[pytorch / pytorch](#)

Watch 1,221 Unstar 26,770 Fork 6,340

Code

Issues 2,286

Pull requests 561

Projects 4

Wiki

Insights

Tree: 517c7c9861 ▾ [pytorch / aten / src / THNN / generic /](#)

Create new file Upload files Find file History

 ezyang and facebook-github-bot Canonicalize all includes in PyTorch. (#14849) ...

Latest commit 517c7c9 on Dec 8, 2018

..		
AbsCriterion.c	Canonicalize all includes in PyTorch. (#14849)	4 months ago
BCECriterion.c	Canonicalize all includes in PyTorch. (#14849)	4 months ago
ClassNLLCriterion.c	Canonicalize all includes in PyTorch. (#14849)	4 months ago
Col2Im.c	Canonicalize all includes in PyTorch. (#14849)	4 months ago
ELU.c	Canonicalize all includes in PyTorch. (#14849)	4 months ago
FeatureLPPooling.c	Canonicalize all includes in PyTorch. (#14849)	4 months ago
GatedLinearUnit.c	Canonicalize all includes in PyTorch. (#14849)	4 months ago
HardTanh.c	Canonicalize all includes in PyTorch. (#14849)	4 months ago
Im2Col.c	Canonicalize all includes in PyTorch. (#14849)	4 months ago
IndexLinear.c	Canonicalize all includes in PyTorch. (#14849)	4 months ago
LeakyReLU.c	Canonicalize all includes in PyTorch. (#14849)	4 months ago
LogSigmoid.c	Canonicalize all includes in PyTorch. (#14849)	4 months ago
MSECriterion.c	Canonicalize all includes in PyTorch. (#14849)	4 months ago
MultiLabelMarginCriterion.c	Canonicalize all includes in PyTorch. (#14849)	4 months ago
MultiMarginCriterion.c	Canonicalize all includes in PyTorch. (#14849)	4 months ago
RReLU.c	Canonicalize all includes in PyTorch. (#14849)	4 months ago
Sigmoid.c	Canonicalize all includes in PyTorch. (#14849)	4 months ago
SmoothL1Criterion.c	Canonicalize all includes in PyTorch. (#14849)	4 months ago
SoftMarginCriterion.c	Canonicalize all includes in PyTorch. (#14849)	4 months ago

[SpatialClassNLLCriterion.c](#)

Canonicalize all includes in PyTorch. (#14849)

[SpatialConvolutionMM.c](#)

Canonicalize all includes in PyTorch. (#14849)

[SpatialDilatedConvolution.c](#)

Canonicalize all includes in PyTorch. (#14849)

[SpatialDilatedMaxPooling.c](#)

Canonicalize all includes in PyTorch. (#14849)

[SpatialFractionalMaxPooling.c](#)

Canonicalize all includes in PyTorch. (#14849)

[SpatialFullDilatedConvolution.c](#)

Canonicalize all includes in PyTorch. (#14849)

[SpatialMaxUnpooling.c](#)

Canonicalize all includes in PyTorch. (#14849)

[SpatialReflectionPadding.c](#)

Canonicalize all includes in PyTorch. (#14849)

[SpatialReplicationPadding.c](#)

Canonicalize all includes in PyTorch. (#14849)

[SpatialUpSamplingBilinear.c](#)

Canonicalize all includes in PyTorch. (#14849)

[SpatialUpSamplingNearest.c](#)

Canonicalize all includes in PyTorch. (#14849)

[THNN.h](#)

Canonicalize all includes in PyTorch. (#14849)

[Tanh.c](#)

Canonicalize all includes in PyTorch. (#14849)

[TemporalReflectionPadding.c](#)

Canonicalize all includes in PyTorch. (#14849)

[TemporalReplicationPadding.c](#)

Canonicalize all includes in PyTorch. (#14849)

[TemporalRowConvolution.c](#)

Canonicalize all includes in PyTorch. (#14849)

[TemporalUpSamplingLinear.c](#)

Canonicalize all includes in PyTorch. (#14849)

[TemporalUpSamplingNearest.c](#)

Canonicalize all includes in PyTorch. (#14849)

[VolumetricAdaptiveAveragePoolin...](#)

Canonicalize all includes in PyTorch. (#14849)

[VolumetricAdaptiveMaxPooling.c](#)

Canonicalize all includes in PyTorch. (#14849)

[VolumetricAveragePooling.c](#)

Canonicalize all includes in PyTorch. (#14849)

[VolumetricConvolutionMM.c](#)

Canonicalize all includes in PyTorch. (#14849)

[VolumetricDilatedConvolution.c](#)

Canonicalize all includes in PyTorch. (#14849)

[VolumetricDilatedMaxPooling.c](#)

Canonicalize all includes in PyTorch. (#14849)

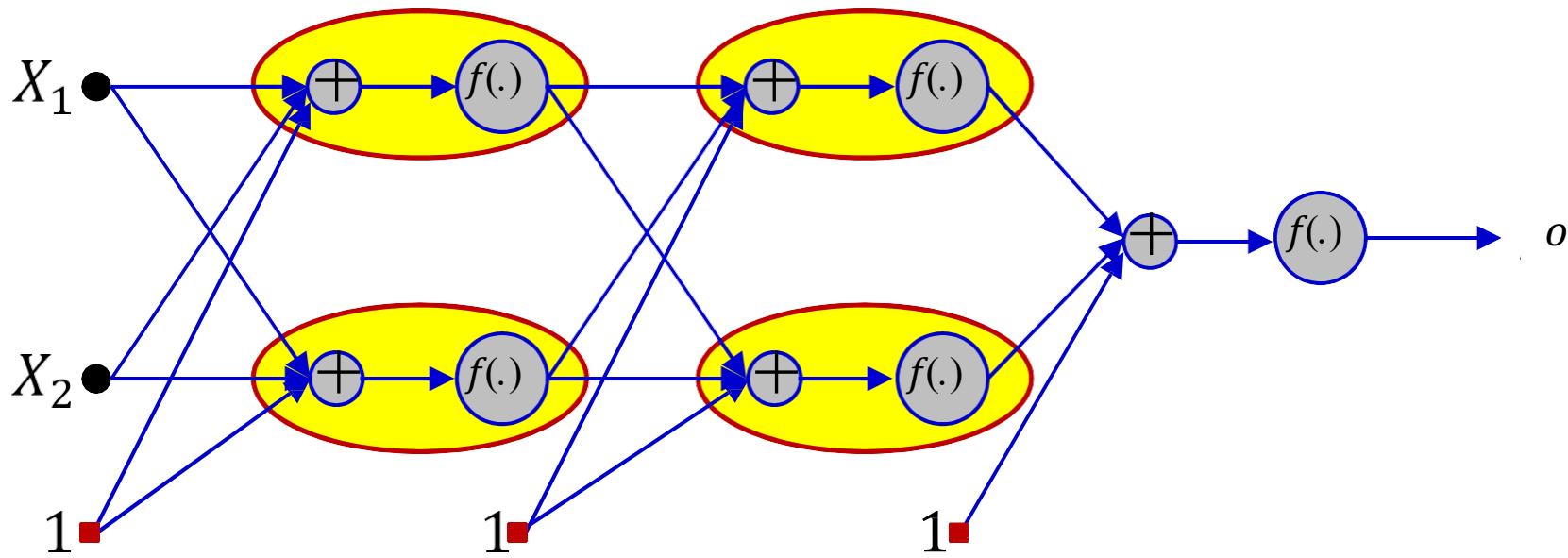
[VolumetricFractionalMaxPooling.c](#)

Canonicalize all includes in PyTorch. (#14849)

[VolumetricFullDilatedConvolution.c](#)

Canonicalize all includes in PyTorch. (#14849)

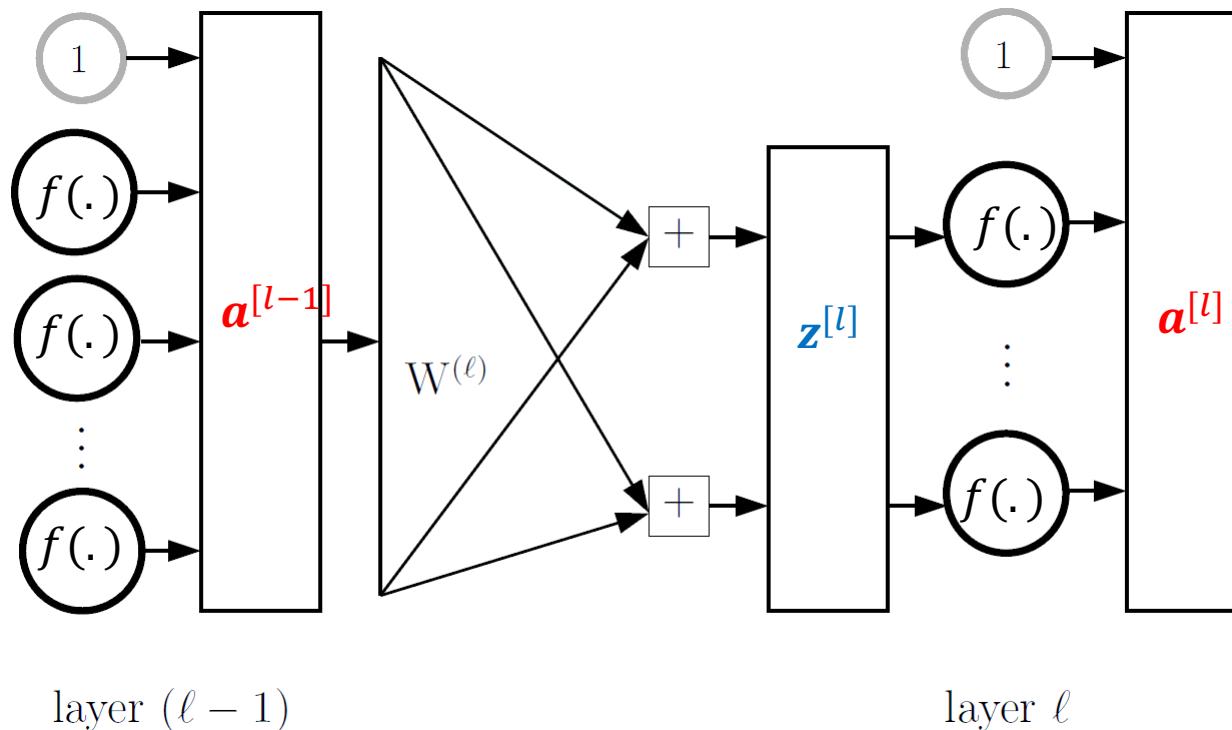
A first closer look at the network



- Showing a tiny 2-input network for illustration
 - Actual network would have many more neurons and inputs
- Explicitly separating the weighted sum of inputs from the activation

Backpropagation: Notation

- $\mathbf{a}^{[0]} \leftarrow Input$
- $output \leftarrow \mathbf{a}^{[L]}$



Backpropagation: Last layer gradient

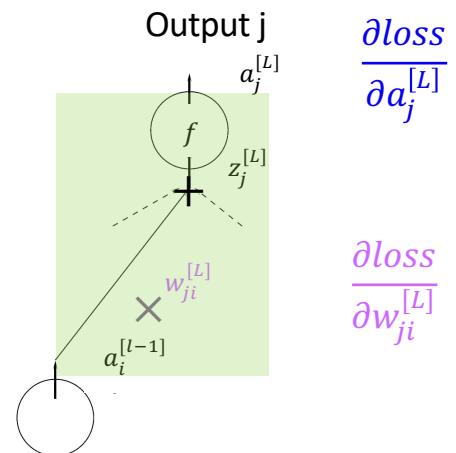
For squared error loss:

$$loss = \frac{1}{2} \sum_j (o_j - y_j)^2$$
$$o_j = a_j^{[L]}$$

$$\frac{\partial loss}{\partial a_j^{[L]}} = (a_j^{[L]} - y_j)$$

$$a_i^{[L]} = f(z_i^{[L]})$$
$$z_j^{[L]} = \sum_{i=0}^M w_{ji}^{[L]} a_i^{[L-1]}$$

$$\frac{\partial loss}{\partial w_{ji}^{[L]}} = ?$$



Backpropagation: Last layer gradient

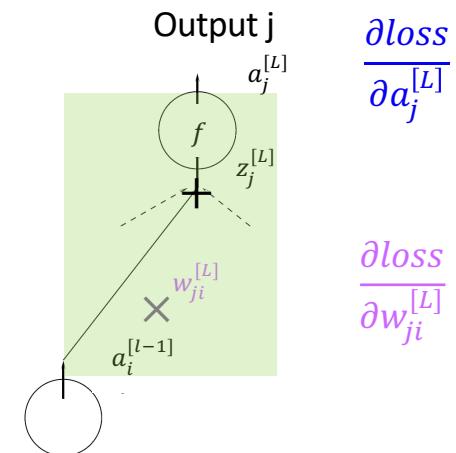
$$\frac{\partial \text{loss}}{\partial a_j^{[L]}} = (a_j^{[L]} - y_j)$$

$$a_j^{[L]} = f(z_j^{[L]})$$

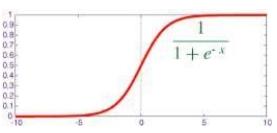
$$z_j^{[L]} = \sum_{i=0}^M w_{ji}^{[L]} a_i^{[L-1]}$$

$$\frac{\partial \text{loss}}{\partial w_{ji}^{[L]}} = \frac{\partial \text{loss}}{\partial a_j^{[L]}} f'(z_j^{[L]}) \frac{\partial z_j^{[L]}}{\partial w_{ji}^{[L]}}$$

$$= \frac{\partial \text{loss}}{\partial a_j^{[L]}} f'(z_j^{[L]}) a_i^{[L-1]}$$

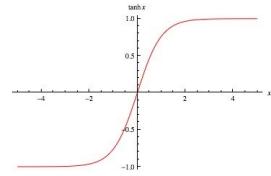


Activations and their derivatives



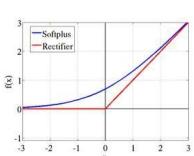
$$f(z) = \frac{1}{1 + \exp(-z)}$$

$$f'(z) = f(z)(1 - f(z))$$



$$f(z) = \tanh(z)$$

$$f'(z) = 1 - f^2(z)$$



$$f(z) = \begin{cases} 0, & z < 0 \\ z, & z \geq 0 \end{cases}$$

$$f'(z) = \begin{cases} 1, & z \geq 0 \\ 0, & z < 0 \end{cases}$$

$$f(z) = \log(1 + \exp(z))$$

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

- Some popular activation functions and their derivatives

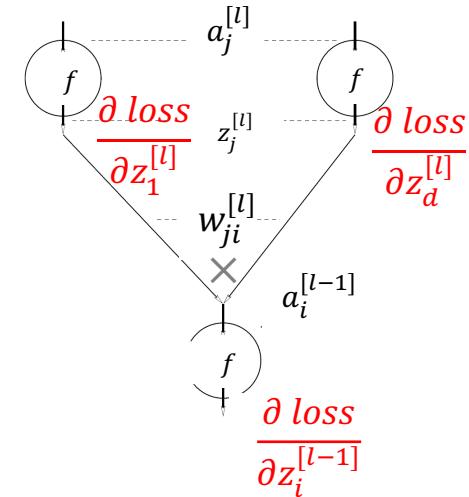
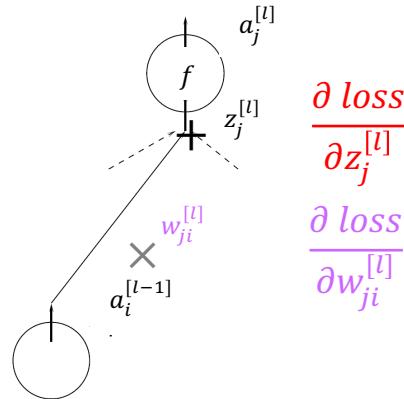
Previous layers gradients

$$\frac{\partial \text{loss}}{\partial w_{ji}^{[l]}} = \frac{\partial \text{loss}}{\partial z_j^{[l]}} \frac{\partial z_j^{[l]}}{\partial w_{ji}^{[l]}} = \frac{\partial \text{loss}}{\partial z_j^{[l]}} a_i^{[l-1]}$$

$$\begin{aligned}\frac{\partial \text{loss}}{\partial z_i^{[l-1]}} &= \frac{\partial a_i^{[l-1]}}{\partial z_i^{[l-1]}} \sum_{j=1}^{d^{[l]}} \frac{\partial \text{loss}}{\partial z_j^{[l]}} \times \frac{\partial z_j^{[l]}}{\partial a_i^{[l-1]}} \\ &= f'(z_i^{[l-1]}) \sum_{j=1}^{d^{[l]}} \frac{\partial \text{loss}}{\partial z_j^{[l]}} \times w_{ji}^{[l]}\end{aligned}$$

$$a_j^{[l]} = f(z_j^{[l]})$$

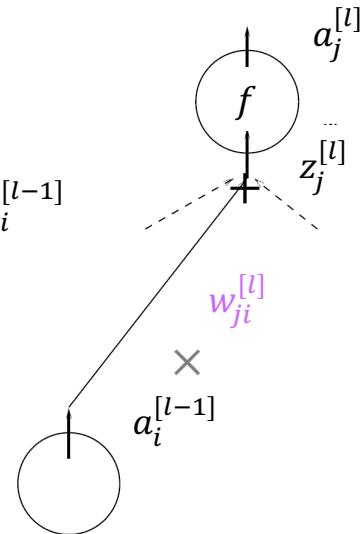
$$z_j^{[l]} = \sum_{i=0}^M w_{ji}^{[l]} a_i^{[l-1]}$$



Backpropagation:

$$\begin{aligned}\frac{\partial \text{loss}}{\partial w_{ji}^{[l]}} &= \boxed{\frac{\partial \text{loss}}{\partial z_j^{[l]}}} \times \frac{\partial z_j^{[l]}}{\partial w_{ji}^{[l]}} \\ &= \boxed{\delta_j^{[l]}} \times a_i^{[l-1]}\end{aligned}$$

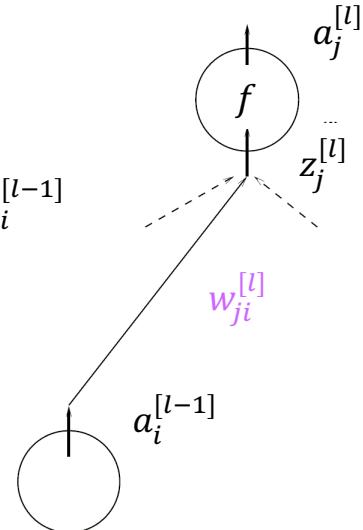
$$\begin{aligned}a_j^{[l]} &= f(z_j^{[l]}) \\ z_j^{[l]} &= \sum_{i=0}^M w_{ji}^{[l]} a_i^{[l-1]}\end{aligned}$$



Backpropagation:

$$\begin{aligned}\frac{\partial \text{loss}}{\partial w_{ji}^{[l]}} &= \boxed{\frac{\partial \text{loss}}{\partial z_j^{[l]}}} \times \frac{\partial z_j^{[l]}}{\partial w_{ji}^{[l]}} \\ &= \boxed{\delta_j^{[l]}} \times \boxed{a_i^{[l-1]}}\end{aligned}$$

$$\begin{aligned}a_j^{[l]} &= f(z_j^{[l]}) \\ z_j^{[l]} &= \sum_{i=0}^M w_{ji}^{[l]} a_i^{[l-1]}\end{aligned}$$



- $\delta_j^{[l]} = \frac{\partial \text{loss}}{\partial z_j^{[l]}}$ is the **sensitivity** of the loss to $z_j^{[l]}$
- Sensitivity vectors can be obtained by running a backward process in the network architecture (hence the name backpropagation.)

We will compute $\delta^{[l-1]}$ from $\delta^{[l]}$:

$$\delta_i^{[l-1]} = f'(z_i^{[l-1]}) \sum_{j=1}^{d^{[l]}} \delta_j^{[l]} \times w_{ji}^{[l]}$$

Backward process on sensitivity vectors

- For the final layer $l = L$:

$$\delta_j^{[L]} = \frac{\partial \text{loss}}{\partial z_j^{[L]}}$$

- Compute $\delta^{[l-1]}$ from $\delta^{[l]}$: by running a backward process in the network architecture:

$$\delta_i^{[l-1]} = f' \left(z_i^{[l-1]} \right) \sum_{j=1}^{d^{[l]}} \delta_j^{[l]} \times w_{ji}^{[l]}$$

Backpropagation Algorithm

- Initialize all weights to small random numbers.
- **While not satisfied**
- **For** each training example **do**:
 1. Feed forward the training example to the network and compute the outputs of all units in forward step (z and a) and the loss
 2. For each unit find its δ in the backward step
 3. Update each network weight $w_{ji}^{[l]}$ as $w_{ji}^{[l]} \leftarrow w_{ji}^{[l]} - \eta \frac{\partial \text{loss}}{\partial w_{ji}^{[l]}}$ where $\frac{\partial \text{loss}}{\partial w_{ji}^{[l]}} = \delta_j^{[l]} \times a_i^{[l-1]}$

Vector formulation

- For layered networks it is generally simpler to think of the process in terms of vector operations
 - Simpler arithmetic
 - Fast matrix libraries make operations *much* faster
- We can restate the entire process in vector terms
 - This is what is *actually* used in any real system

The Jacobian

- The derivative of a vector function w.r.t. vector input is called a *Jacobian*
- It is the matrix of partial derivatives given below

$$\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{bmatrix} = f \left(\begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_d \end{bmatrix} \right)$$

Using vector notation

$$\mathbf{y} = f(\mathbf{x})$$

$$\frac{\partial \mathbf{y}}{\partial \mathbf{x}} = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \frac{\partial y_2}{\partial x_1} & \cdots & \frac{\partial y_m}{\partial x_1} \\ \frac{\partial y_1}{\partial x_2} & \frac{\partial y_2}{\partial x_2} & \cdots & \frac{\partial y_m}{\partial x_2} \\ \cdots & \cdots & \ddots & \cdots \\ \frac{\partial y_1}{\partial x_d} & \frac{\partial y_2}{\partial x_d} & \cdots & \frac{\partial y_m}{\partial x_d} \end{bmatrix}$$

Matrix calculus

- Scalar-by-Vector

$$\frac{\partial y}{\partial \mathbf{x}} = \begin{bmatrix} \frac{\partial y}{\partial x_1} & \dots & \frac{\partial y}{\partial x_n} \end{bmatrix}^T$$

- Vector-by-Vector

$$\frac{\partial \mathbf{y}}{\partial \mathbf{x}} = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \dots & \frac{\partial y_m}{\partial x_1} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_1}{\partial x_n} & \dots & \frac{\partial y_m}{\partial x_n} \end{bmatrix}$$

- Scalar-by-Matrix

$$\frac{\partial y}{\partial A} = \begin{bmatrix} \frac{\partial y}{\partial A_{11}} & \dots & \frac{\partial y}{\partial A_{1n}} \\ \vdots & \ddots & \vdots \\ \frac{\partial y}{\partial A_{m1}} & \dots & \frac{\partial y}{\partial A_{mn}} \end{bmatrix}$$

- Vector-by-Matrix

$$\frac{\partial \mathbf{y}}{\partial A_{ij}} = \frac{\partial y}{\partial \mathbf{z}} \frac{\partial \mathbf{z}}{\partial A_{ij}}$$

Vector derivatives

Scalar to Scalar

$$x \in \mathbb{R}, y \in \mathbb{R}$$

Regular derivative:

$$\frac{\partial y}{\partial x} \in \mathbb{R}$$

If x changes by a small amount, how much will y change?

Vector to Scalar

$$x \in \mathbb{R}^N, y \in \mathbb{R}$$

Derivative is **Gradient**:

$$\frac{\partial y}{\partial x} \in \mathbb{R}^N \quad \left(\frac{\partial y}{\partial x} \right)_n = \frac{\partial y}{\partial x_n}$$

For each element of x , if it changes by a small amount then how much will y change?

Vector to Vector

$$x \in \mathbb{R}^N, y \in \mathbb{R}^M$$

Derivative is **Jacobian**:

$$\frac{\partial y}{\partial x} \in \mathbb{R}^{N \times M} \quad \left(\frac{\partial y}{\partial x} \right)_{n,m} = \frac{\partial y_m}{\partial x_n}$$

For each element of x , if it changes by a small amount then how much will each element of y change?

Examples of derivatives

$$\mathbf{z} = \mathbf{Wx}$$

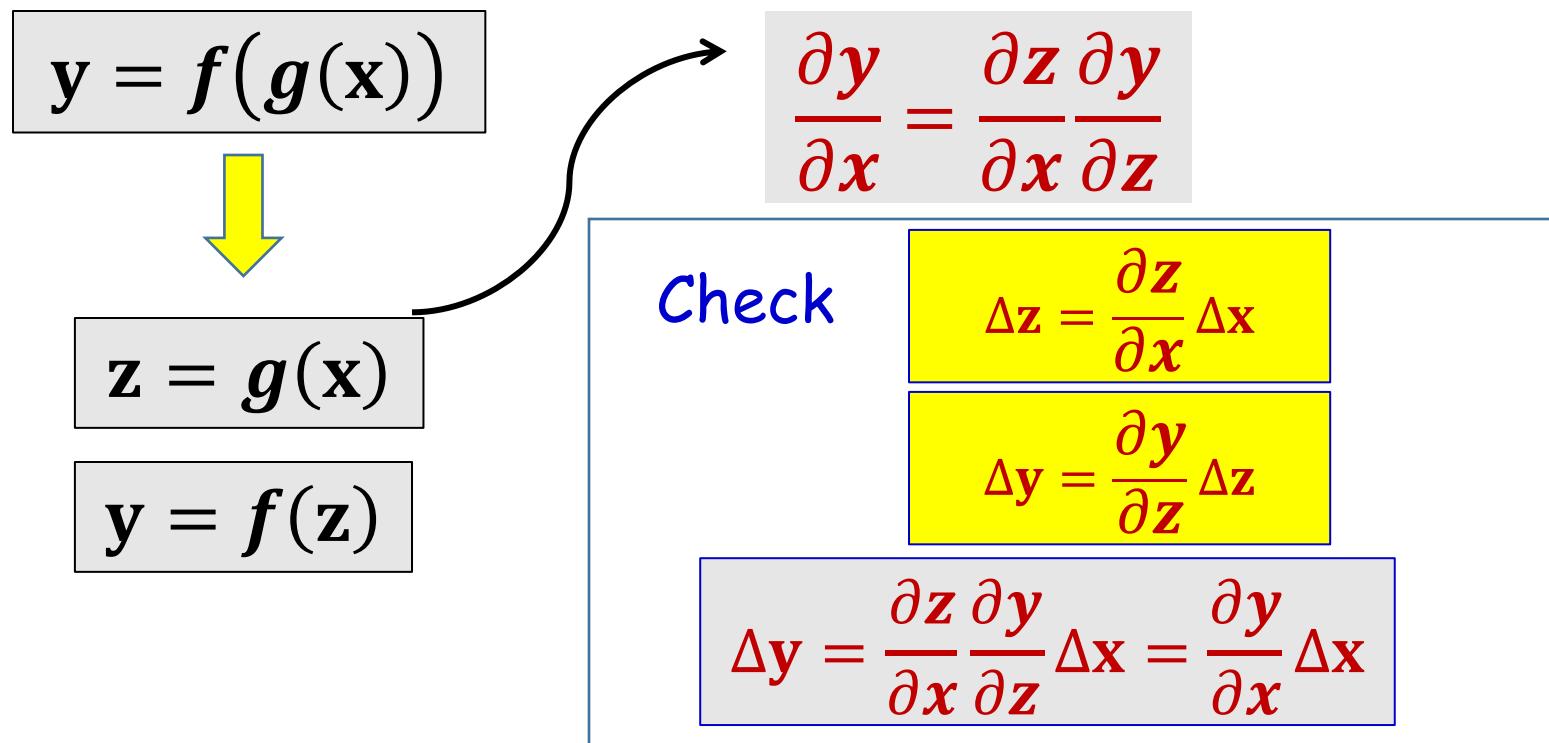
$$\frac{\partial \mathbf{z}}{\partial \mathbf{x}} = \mathbf{W}^T$$

$$z_i = \sum_j W_{ij} x_j$$

$$\frac{\partial z_i}{\partial x_j} = W_{ij}$$

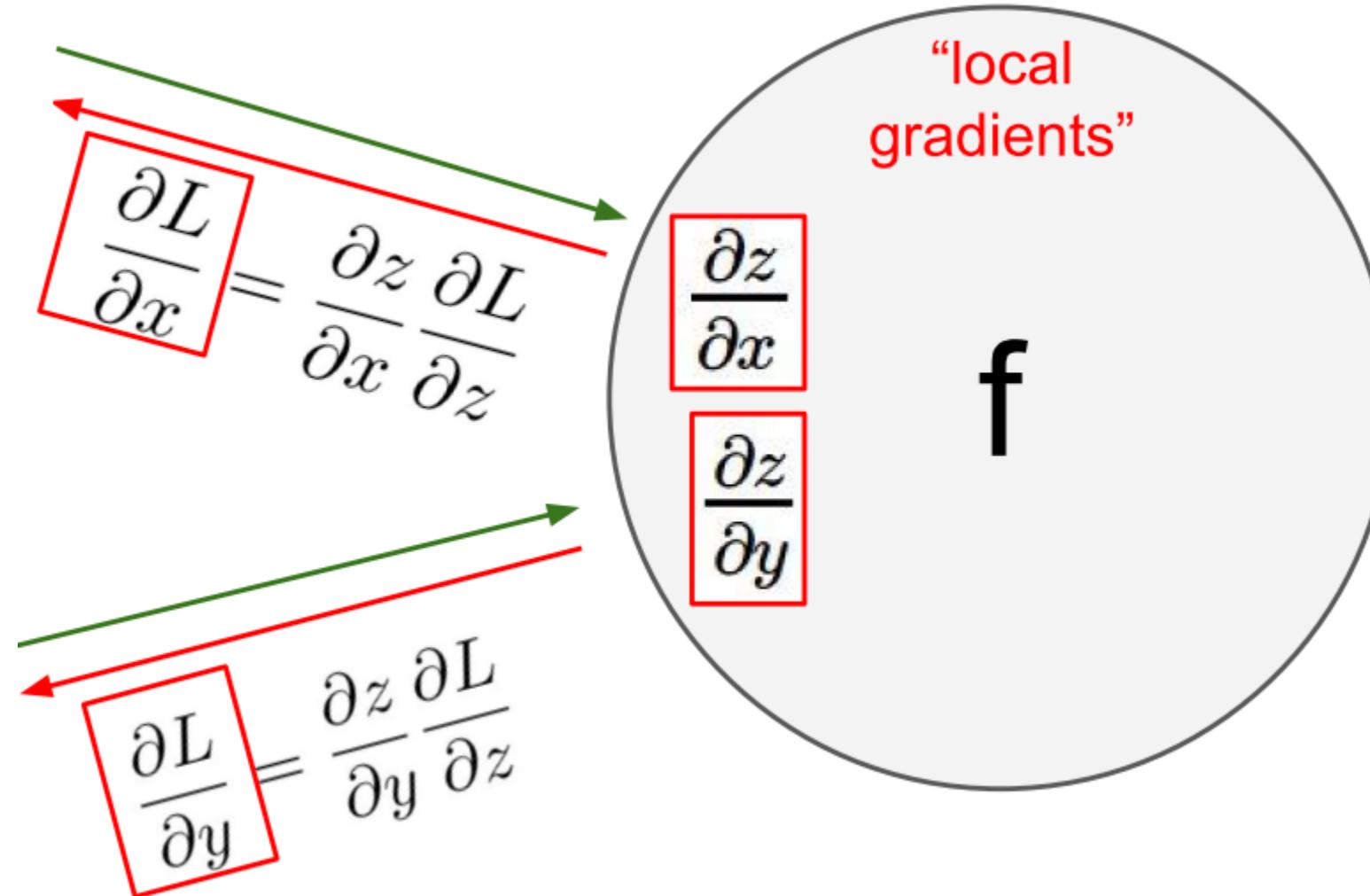
Vector derivatives: Chain rule

- We can define a chain rule for Jacobians
- **For vector functions of vector inputs:**



Note the order: The derivative of the inner function comes first

Backprop with Vectors



Loss L still a scalar!

z

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

"Upstream gradient"

For each element of z , how much does it influence L ?

Examples of derivatives

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

$$\frac{\partial L}{\partial \mathbf{z}} = \boldsymbol{\delta}$$

$$\frac{\partial L}{\partial \mathbf{W}} = \boldsymbol{\delta}\mathbf{x}^T$$

$$\frac{\partial \mathbf{z}}{\partial W_{ij}} = \begin{bmatrix} 0 \\ \vdots \\ 0 \\ x_j \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

$$\frac{\partial L}{\partial W_{ij}} = \delta_i x_j$$

Backpropagation shape rule

- When you take gradients against a **scalar**, the gradient at each intermediate step has **shape of denominator**

$$X \in \mathbb{R}^{m \times n} \Leftrightarrow \frac{\partial L}{\partial W} \in \mathbb{R}^{m \times n}$$

Dimension balancing

$$Z = XW$$

$$[m \times w]$$

$$\frac{\partial Loss}{\partial Z} = \delta$$

$$W$$

$$[n \times w]$$

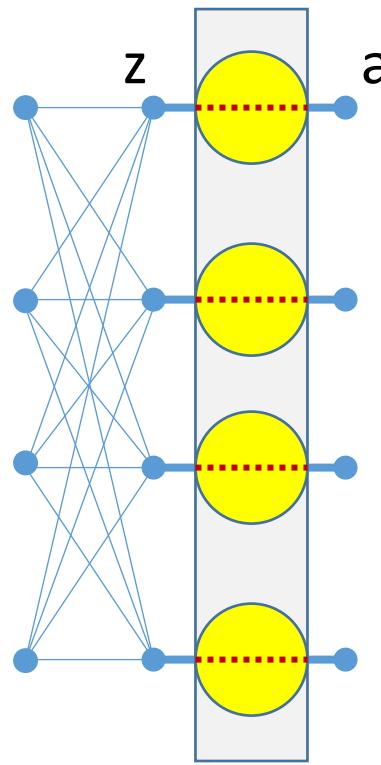
$$\frac{\partial Loss}{\partial W} = X^\top \delta$$

$$X$$

$$[m \times n]$$

$$\frac{\partial Loss}{\partial X} = \delta W^T$$

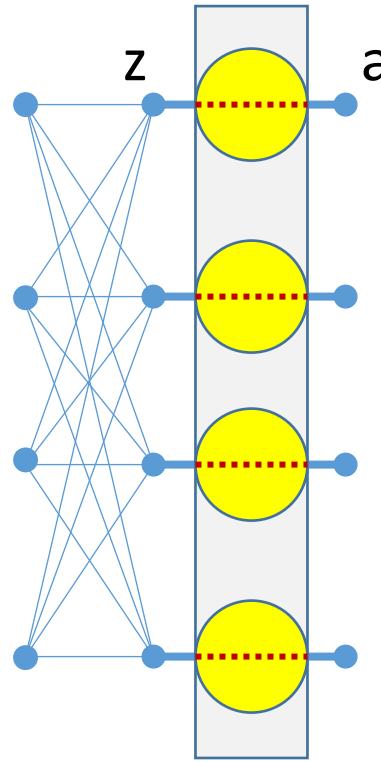
Jacobians can describe derivatives of activations w.r.t their input



$$\frac{\partial \mathbf{a}}{\partial \mathbf{z}} = \begin{bmatrix} \frac{da_1}{dz_1} & 0 & \dots & 0 \\ 0 & \frac{da_2}{dz_2} & \dots & 0 \\ \dots & \dots & \ddots & \dots \\ 0 & 0 & \dots & \frac{da_m}{dz_m} \end{bmatrix}$$

- **For Scalar activations**
 - Number of outputs is identical to the number of inputs
- Jacobian is a diagonal matrix
 - Diagonal entries are individual derivatives of outputs w.r.t inputs
 - Not showing the superscript “[k]” in equations for brevity

Activation function: sigmoid

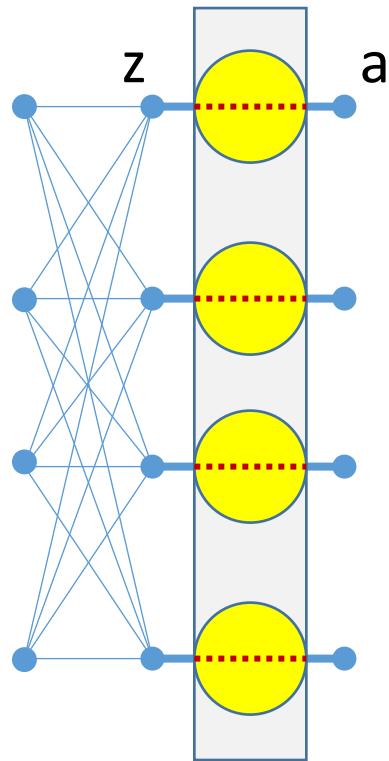


$$a_i = f(z_i)$$

$$\frac{\partial \mathbf{a}}{\partial \mathbf{z}} = \begin{bmatrix} f'(z_1) & 0 & \cdots & 0 \\ 0 & f'(z_2) & \cdots & 0 \\ \cdots & \cdots & \ddots & \cdots \\ 0 & 0 & \cdots & f'(z_m) \end{bmatrix}$$

- **For scalar activations (shorthand notation):**
 - Jacobian is a diagonal matrix
 - Diagonal entries are individual derivatives of outputs w.r.t inputs

Activation function: sigmoid

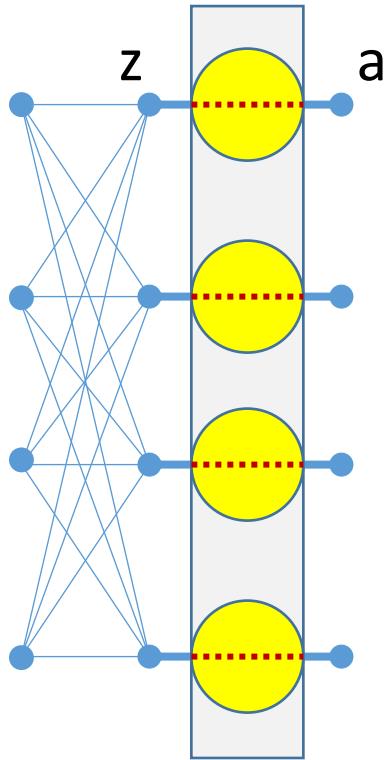


$$a_i = \sigma(z_i)$$

$$\frac{\partial a}{\partial z} = \begin{bmatrix} \sigma(z_1)(1 - \sigma(z_1)) & 0 & \cdots & 0 \\ 0 & \sigma(z_2)(1 - \sigma(z_2)) & \cdots & 0 \\ \cdots & \cdots & \ddots & \cdots \\ 0 & 0 & \cdots & \sigma(z_m)(1 - \sigma(z_m)) \end{bmatrix}$$

- **For scalar activations (shorthand notation):**
 - Jacobian is a diagonal matrix
 - Diagonal entries are individual derivatives of outputs w.r.t inputs

Activation function: ReLU



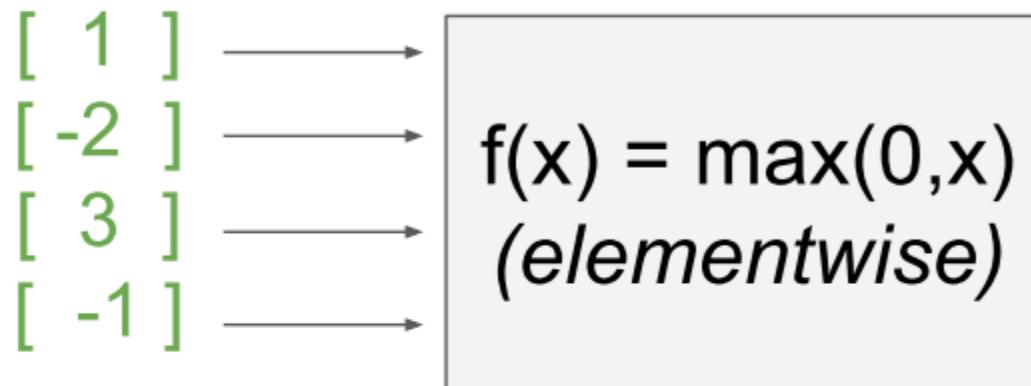
$$a_i = \max(z_i, 0)$$

$$\frac{\partial a}{\partial z} = \begin{bmatrix} I(z_1 > 0) & 0 & \cdots & 0 \\ 0 & I(z_2 > 0) & \cdots & 0 \\ \cdots & \cdots & \ddots & \cdots \\ 0 & 0 & \cdots & I(z_m > 0) \end{bmatrix}$$

- **For scalar activations (shorthand notation):**
 - Jacobian is a diagonal matrix
 - Diagonal entries are individual derivatives of outputs w.r.t inputs

Activation function: ReLU example

4D input x :



4D output z :



4D dL/dx :

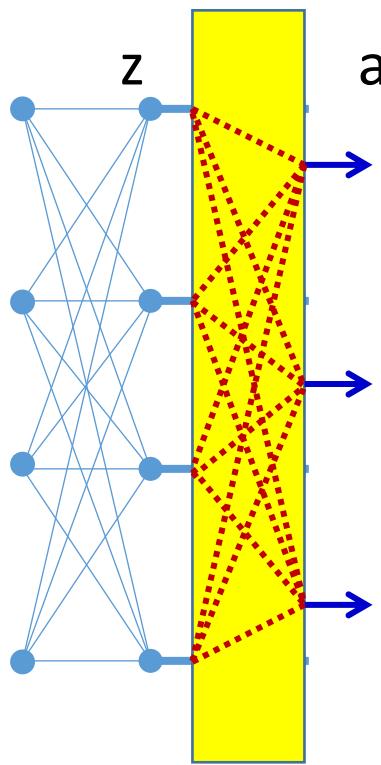
$$\begin{bmatrix} 4 \\ 0 \\ 5 \\ 0 \end{bmatrix} \leftarrow \begin{bmatrix} 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 4 \\ -1 \\ 5 \\ 9 \end{bmatrix}$$

4D dL/dz :

$$\begin{bmatrix} 4 \\ -1 \\ 5 \\ 9 \end{bmatrix} \leftarrow \begin{bmatrix} 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 4 \\ -1 \\ 5 \\ 9 \end{bmatrix}$$

Upstream
gradient

For Vector activations

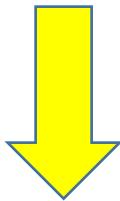


$$\frac{\partial \mathbf{a}}{\partial \mathbf{z}} = \begin{bmatrix} \frac{\partial a_1}{\partial z_1} & \frac{\partial a_2}{\partial z_1} & \dots & \frac{\partial a_m}{\partial z_1} \\ \frac{\partial a_1}{\partial z_2} & \frac{\partial a_2}{\partial z_2} & \dots & \frac{\partial a_m}{\partial z_2} \\ \dots & \dots & \ddots & \dots \\ \frac{\partial a_1}{\partial z_n} & \frac{\partial a_2}{\partial z_n} & \dots & \frac{\partial a_m}{\partial z_n} \end{bmatrix}$$

- Jacobian is a full matrix
 - Entries are partial derivatives of individual outputs w.r.t individual inputs

Special case: Affine functions

$$\mathbf{z}^{[l]} = \mathbf{W}^{[l]} \mathbf{a}^{[l-1]} + \mathbf{b}^{[l]}$$



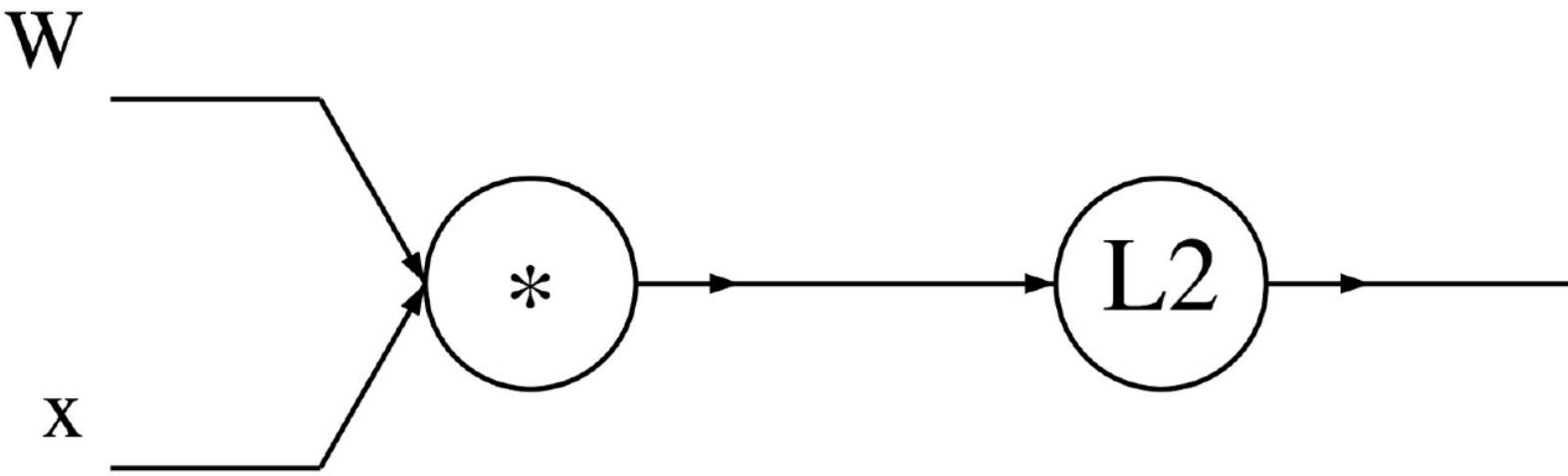
$$\frac{\partial \mathbf{z}^{[l]}}{\partial \mathbf{a}^{[l-1]}} = \mathbf{W}^{[l]T}$$

- Matrix \mathbf{W} and bias \mathbf{b} operating on vector $\mathbf{a}^{[l-1]}$ to produce vector $\mathbf{z}^{[l]}$

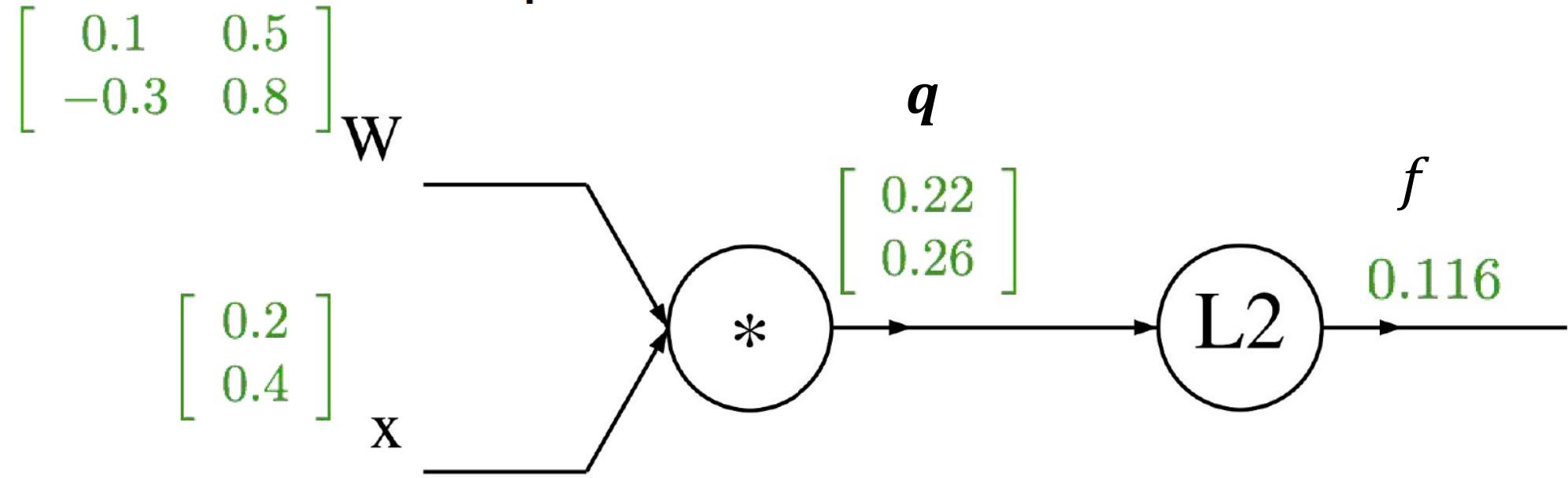
A vectorized example: $f(x, W) = ||W \cdot x||^2$

$$\begin{matrix} & \\ \downarrow & \downarrow \\ \in \mathbb{R}^n & \in \mathbb{R}^{n \times n} \end{matrix}$$

A vectorized example: $f(x, W) = ||W \cdot x||^2$



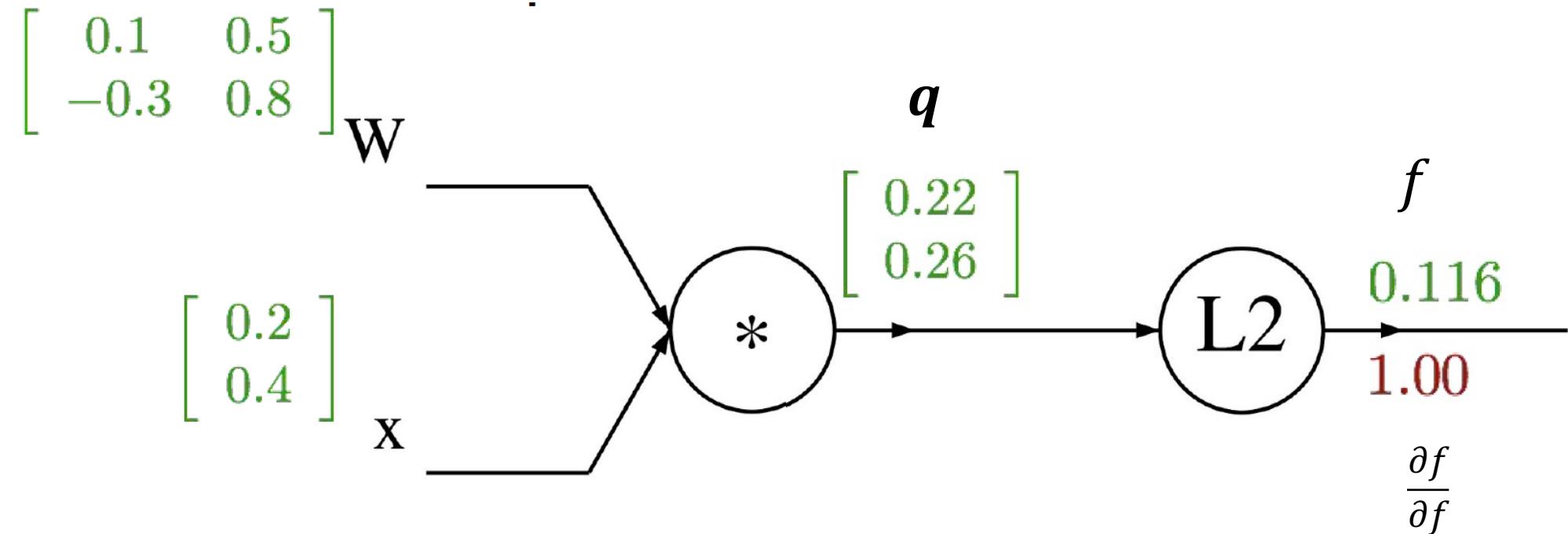
A vectorized example: $f(x, W) = \|W \cdot x\|^2$



$$q = Wx$$

$$f(q) = \|q\|^2 = q^T q$$

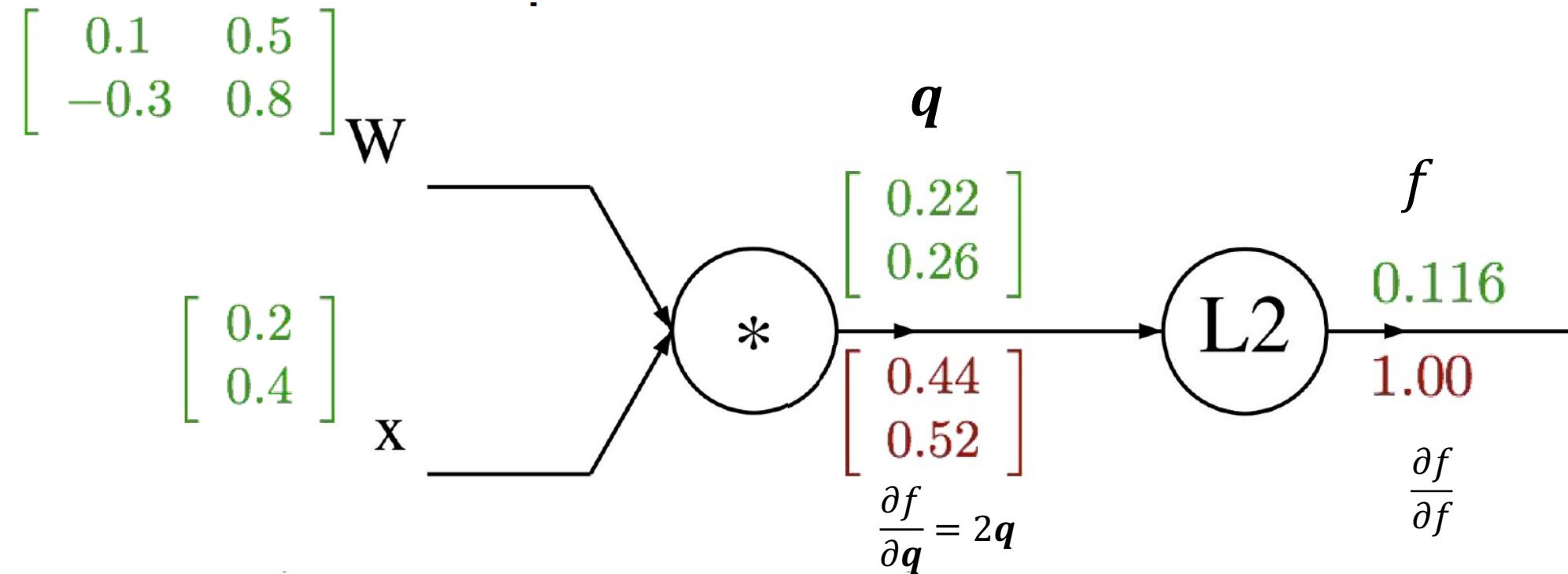
A vectorized example: $f(x, W) = \|W \cdot x\|^2 = \sum_{i=1}^n (W \cdot x)_i^2$



$$q = Wx$$

$$f(q) = \|q\|^2 = q^T q$$

A vectorized example: $f(x, W) = \|W \cdot x\|^2 = \sum_{i=1}^n (W \cdot x)_i^2$



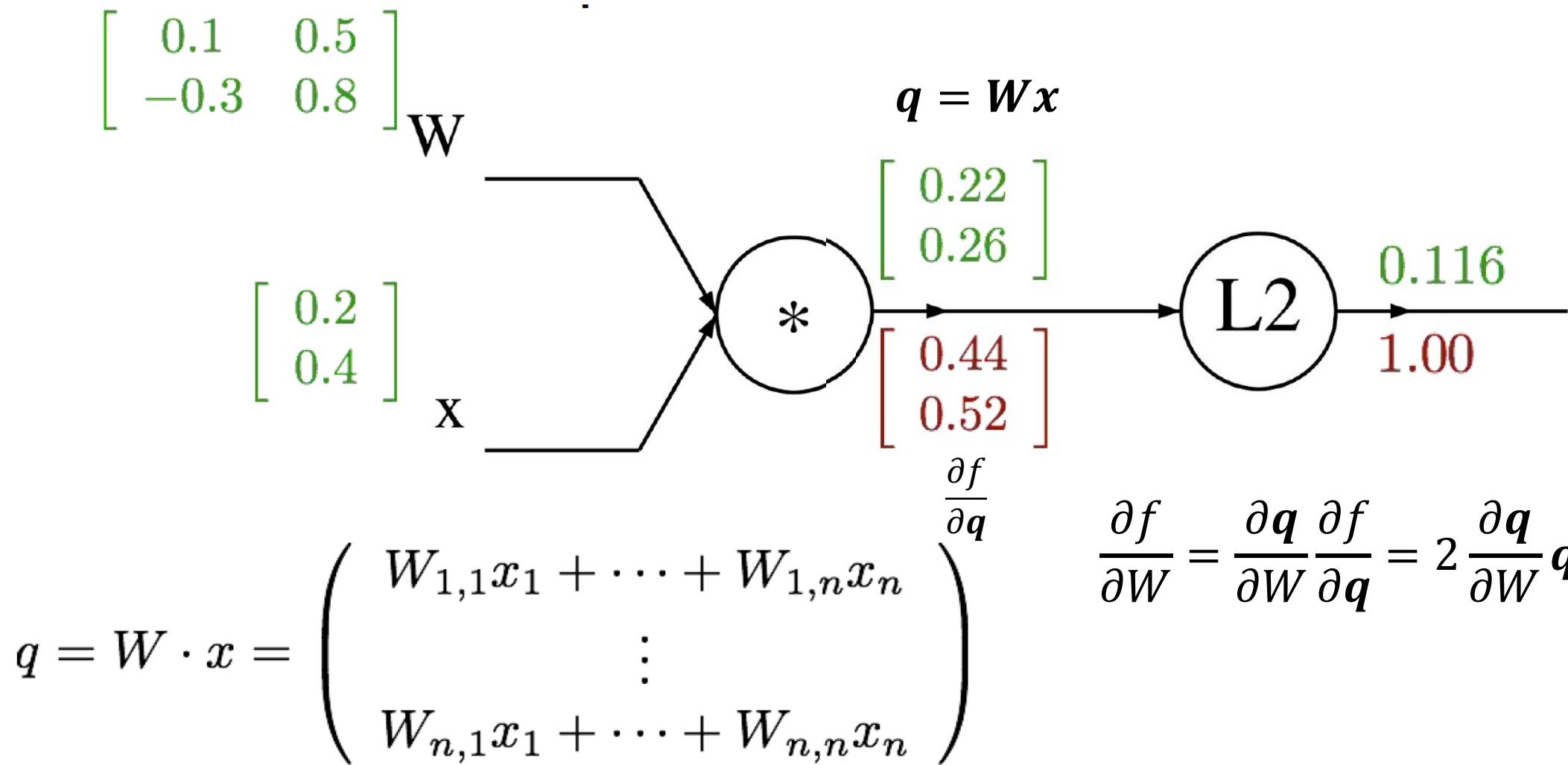
$$q = Wx$$

$$f(q) = \|q\|^2 = q^T q = q_1^2 + \dots + q_m^2$$

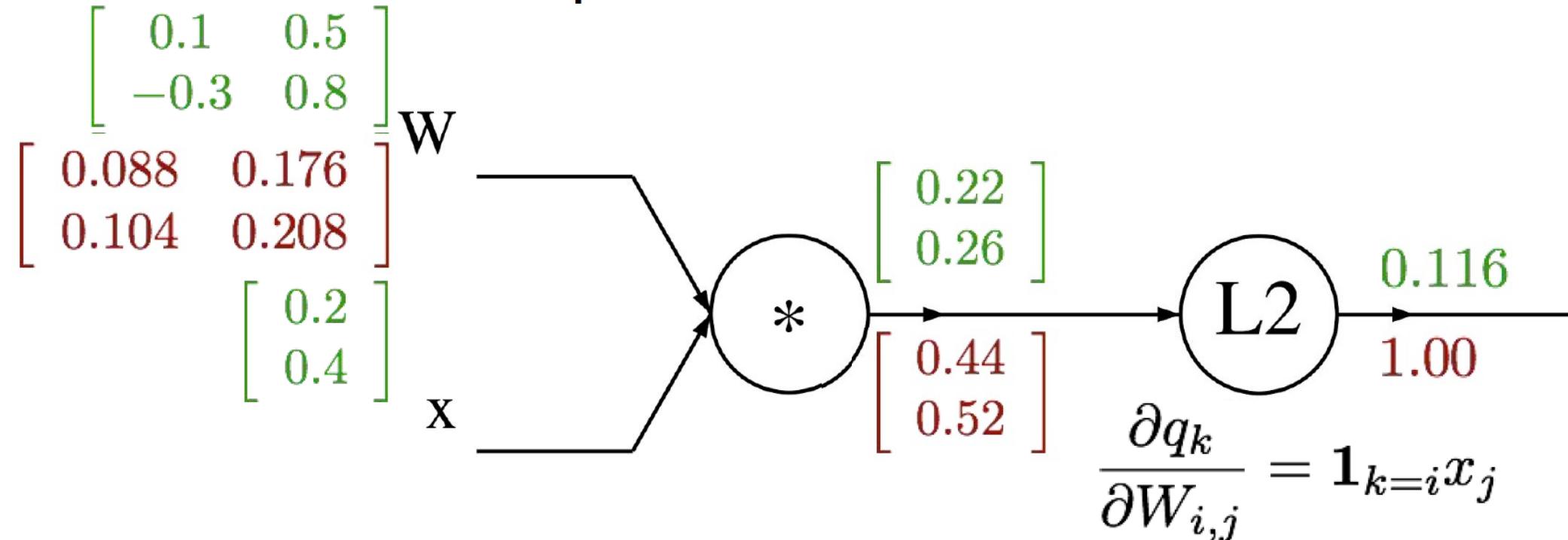
$$\frac{\partial f}{\partial q_i} = 2q_i$$

$$\frac{\partial f}{\partial x} = 2W^T q$$

A vectorized example: $f(x, W) = ||W \cdot x||^2 = \sum_{i=1}^n (W \cdot x)_i^2$



A vectorized example: $f(x, W) = ||W \cdot x||^2 = \sum_{i=1}^n (W \cdot x)_i^2$

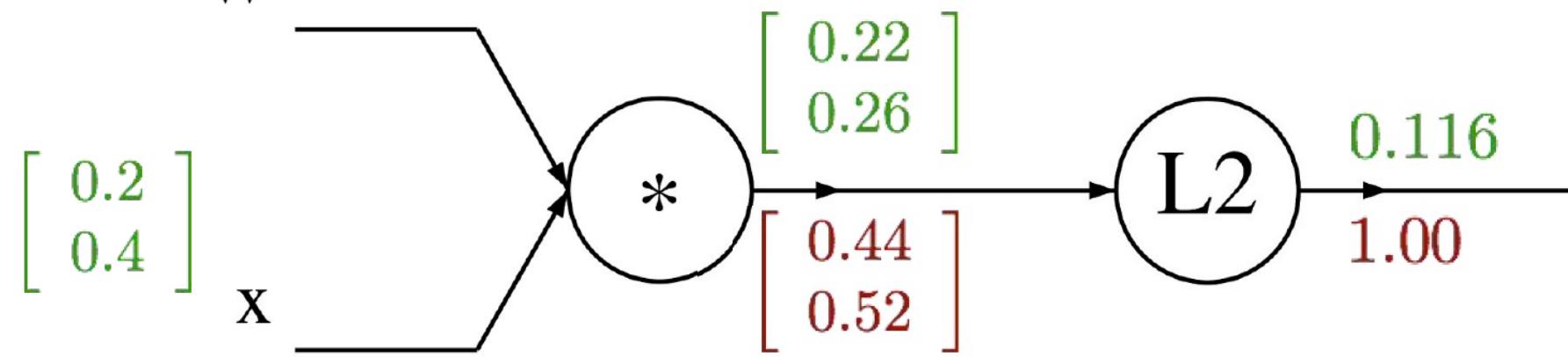


Always check: The gradient with respect to a variable should have the same shape as the Variable

$$\begin{aligned} \frac{\partial f}{\partial W_{i,j}} &= \sum_k \frac{\partial f}{\partial q_k} \frac{\partial q_k}{\partial W_{i,j}} \\ &= \sum_k (2q_k) (\mathbf{1}_{k=i} x_j) \\ &= 2q_i x_j \end{aligned}$$

A vectorized example: $f(x, W) = ||W \cdot x||^2 = \sum_{i=1}^n (W \cdot x)_i^2$

$$\begin{bmatrix} 0.1 & 0.5 \\ -0.3 & 0.8 \end{bmatrix} W$$

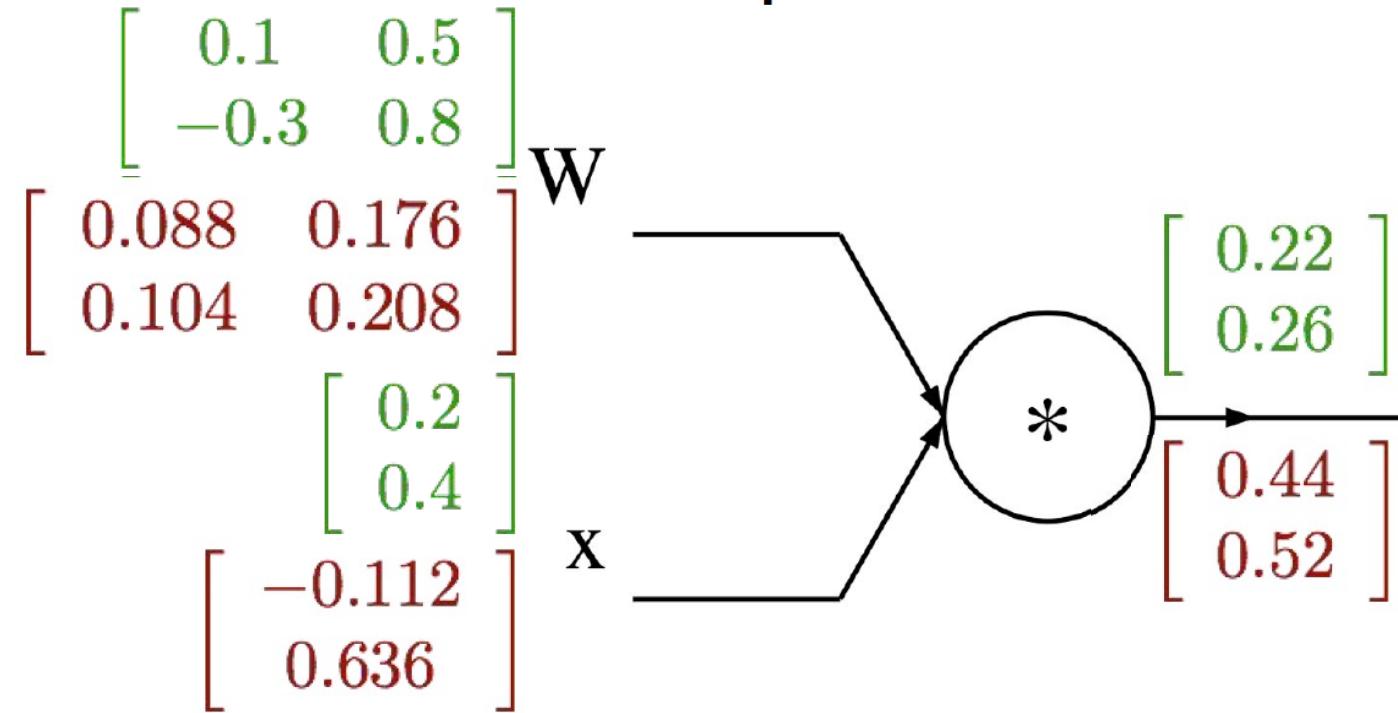


$$q = W \cdot x = \begin{pmatrix} W_{1,1}x_1 + \cdots + W_{1,n}x_n \\ \vdots \\ W_{n,1}x_1 + \cdots + W_{n,n}x_n \end{pmatrix}$$

$$f(q) = ||q||^2 = q_1^2 + \cdots + q_n^2$$

$$\frac{\partial f}{\partial W} = \frac{\partial f}{\partial q} \frac{\partial q}{\partial W} = \frac{\partial f}{\partial q} x^T = 2qx^T$$

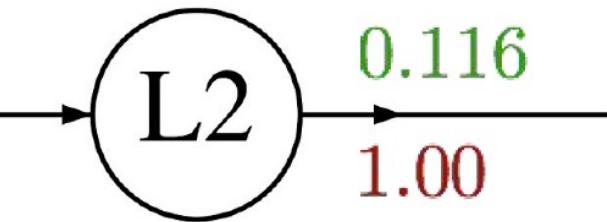
A vectorized example: $f(x, W) = ||W \cdot x||^2 = \sum_{i=1}^n (W \cdot x)_i^2$



$$q = W \cdot x = \begin{pmatrix} W_{1,1}x_1 + \cdots + W_{1,n}x_n \\ \vdots \\ W_{n,1}x_1 + \cdots + W_{n,n}x_n \end{pmatrix}$$

$$f(q) = ||q||^2 = q_1^2 + \cdots + q_n^2$$

$$\nabla_x f = 2W^T \cdot q$$

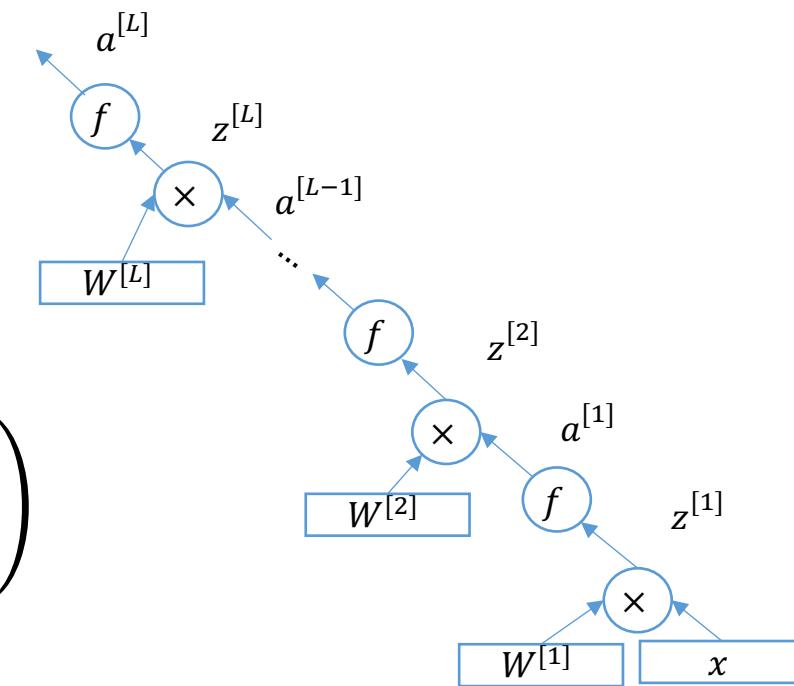


$$\frac{\partial q_k}{\partial x_i} = W_{k,i}$$

$$\begin{aligned} \frac{\partial f}{\partial x_i} &= \sum_k \frac{\partial f}{\partial q_k} \frac{\partial q_k}{\partial x_i} \\ &= \sum_k 2q_k W_{k,i} \end{aligned}$$

Output as a composite function

$$\begin{aligned} \text{Output} &= a^{[L]} \\ &= f(z^{[L]}) \\ &= f(W^{[L]}a^{[L-1]}) \\ &= f(W^{[L]}f(W^{[L-1]}a^{[L-2]})) \\ &= f\left(W^{[L]}f\left(W^{[L-1]} \dots f\left(W^{[2]}f\left(W^{[1]}x\right)\right)\right)\right) \end{aligned}$$



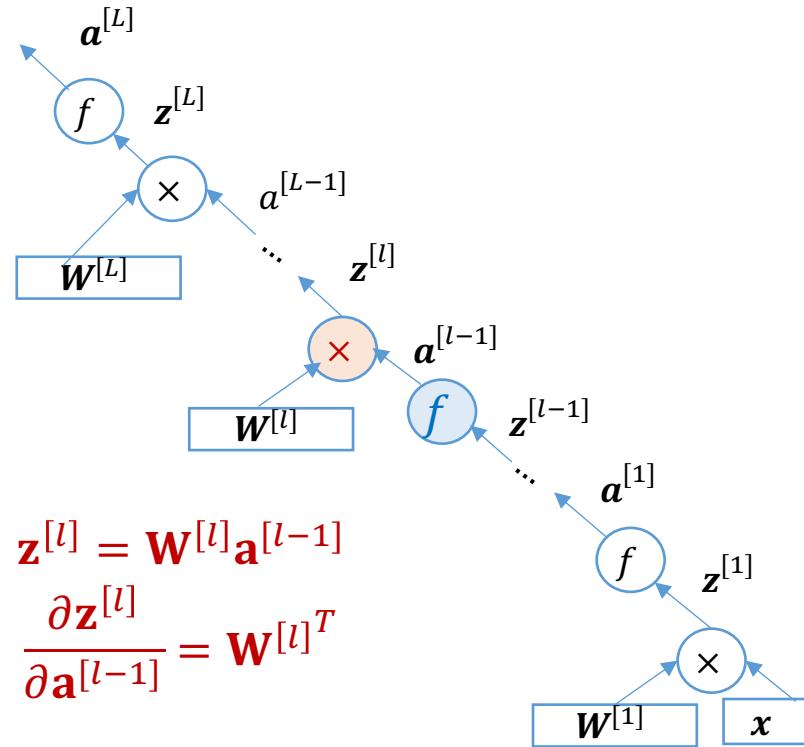
For convenience, we use the same activation functions for all layers.

However, output layer neurons most commonly do not need activation function (they show class scores or real-valued targets.)

Backward-pass vector

- Assume we have $\frac{\partial Loss}{\partial \mathbf{a}^{[L]}}$
- $\frac{\partial Loss}{\partial \mathbf{z}^{[L]}} = \frac{\partial \mathbf{a}^{[L]}}{\partial \mathbf{z}^{[L]}} \frac{\partial Loss}{\partial \mathbf{a}^{[L]}}$
- $\frac{\partial Loss}{\partial \mathbf{W}^{[l]}} = \frac{\partial \mathbf{z}^{[l]}}{\partial \mathbf{W}^{[l]}} \frac{\partial Loss}{\partial \mathbf{z}^{[l]}} = \frac{\partial Loss}{\partial \mathbf{z}^{[l]}} \mathbf{a}^{[l-1]}^T$
- $\frac{\partial Loss}{\partial \mathbf{z}^{[l-1]}} = \frac{\partial \mathbf{z}^{[l]}}{\partial \mathbf{a}^{[l-1]}} \frac{\partial \mathbf{a}^{[l]}}{\partial \mathbf{z}^{[l]}} \frac{\partial Loss}{\partial \mathbf{z}^{[l]}} = \mathbf{W}^{[l]} \frac{\partial \mathbf{a}^{[l]}}{\partial \mathbf{z}^{[l]}} \frac{\partial Loss}{\partial \mathbf{z}^{[l]}}$

$$\mathbf{a}^{[L]} = output = \hat{y}$$



$$\frac{\partial \mathbf{a}^{[l]}}{\partial \mathbf{z}^{[l]}} = \begin{bmatrix} f'(z_1^{[l]}) & 0 & \cdots & 0 \\ 0 & f'(z_2^{[l]}) & \cdots & 0 \\ \cdots & \cdots & \ddots & \cdots \\ 0 & 0 & \cdots & f'(z_m^{[l]}) \end{bmatrix}$$

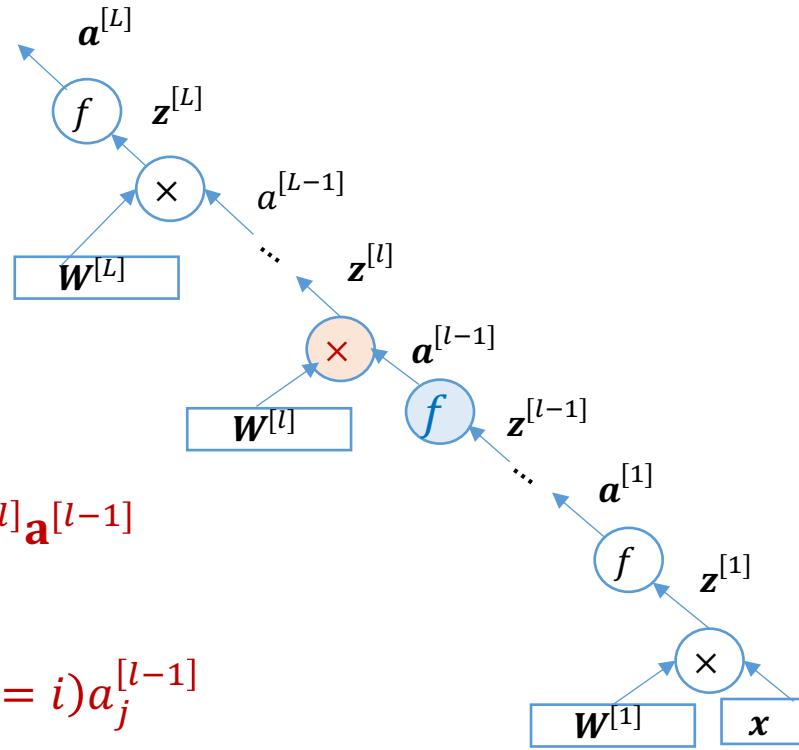
Backward-pass vector

- Assume we have $\frac{\partial Loss}{\partial \mathbf{a}^{[L]}}$

- $\frac{\partial Loss}{\partial \mathbf{z}^{[L]}} = \frac{\partial \mathbf{a}^{[L]}}{\partial \mathbf{z}^{[L]}} \frac{\partial Loss}{\partial \mathbf{a}^{[L]}}$

- $\frac{\partial Loss}{\partial W_{ij}^{[l]}} = \frac{\partial Loss}{\partial \mathbf{z}^{[l]}} \frac{\partial \mathbf{z}^{[l]}}{\partial W_{ij}^{[l]}} = \frac{\partial Loss}{\partial z_i^{[l]}} \frac{\partial z_i^{[l]}}{\partial W_{ij}^{[l]}}$

$$\mathbf{a}^{[L]} = output = \hat{y}$$



Find and save $\delta^{[L]}$

- Called error, computed recursively in backward manner
- For the final layer $l = L$:

$$\delta^{[L]} = \frac{\partial Loss}{\partial z^{[L]}}$$

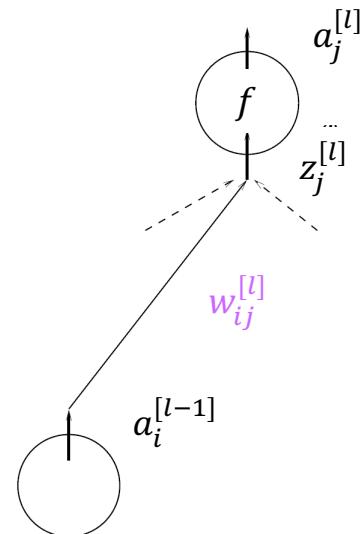
Compute $\delta^{[l-1]}$ from $\delta^{[l]}$

- $\delta^{[l]} = \frac{\partial \text{loss}}{\partial z^{[l]}}$ is the **sensitivity** of the output to $z^{[l]}$

$$a_i^{[l]} = f(z_i^{[l]})$$
$$z_j^{[l]} = \sum_{i=0}^M w_{ij}^{[l]} a_i^{[l-1]}$$

- Sensitivity vectors can be obtained by running a backward process in the network architecture (hence the name backpropagation.)

$$\delta^{[l-1]} = W^{[l]}{}^T f'(a^{[l-1]}) \delta^{[l]}$$



Mini-batch SGD

- Loop:
 1. **Sample** a batch of data
 2. **Forward** prop it through the graph (network), get loss
 3. **Backprop** to calculate the gradients
 4. **Update** the parameters using the gradient

Summary

- Neural nets may be very large: impractical to write down gradient formula by hand for all parameters
- **Backpropagation** = recursive application of the chain rule along a computational graph to compute the gradients of all inputs/parameters/intermediates + Dynamic Programming
- Implementations maintain a graph structure, where the nodes implement the **forward()** / **backward()** API
 - **forward**: compute result of an operation and save any intermediates needed for gradient computation in memory
 - **backward**: apply the chain rule to compute the gradient of the loss function with respect to the inputs

Converting error derivatives into a learning procedure

- The backpropagation algorithm is an efficient way of computing the gradient of the error function w.r.t. weights and biases.
- There are many other decisions to be made to have a learning procedure from these derivatives:
 - Convergence or optimization issues: How do we use the error derivatives?
 - Generalization issues: How can we improve its decisions on unseen data?