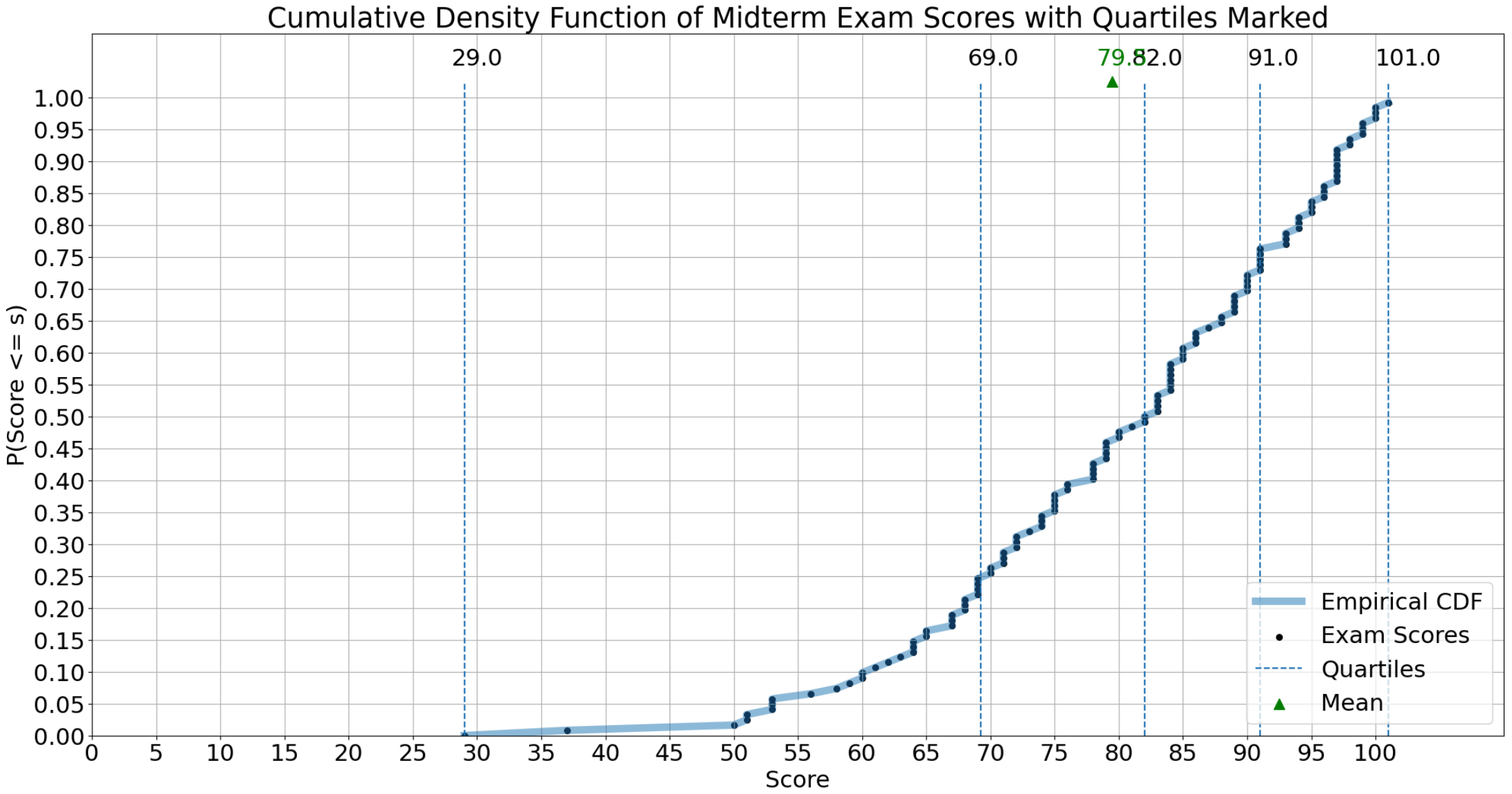
Machine Learning and Data Mining

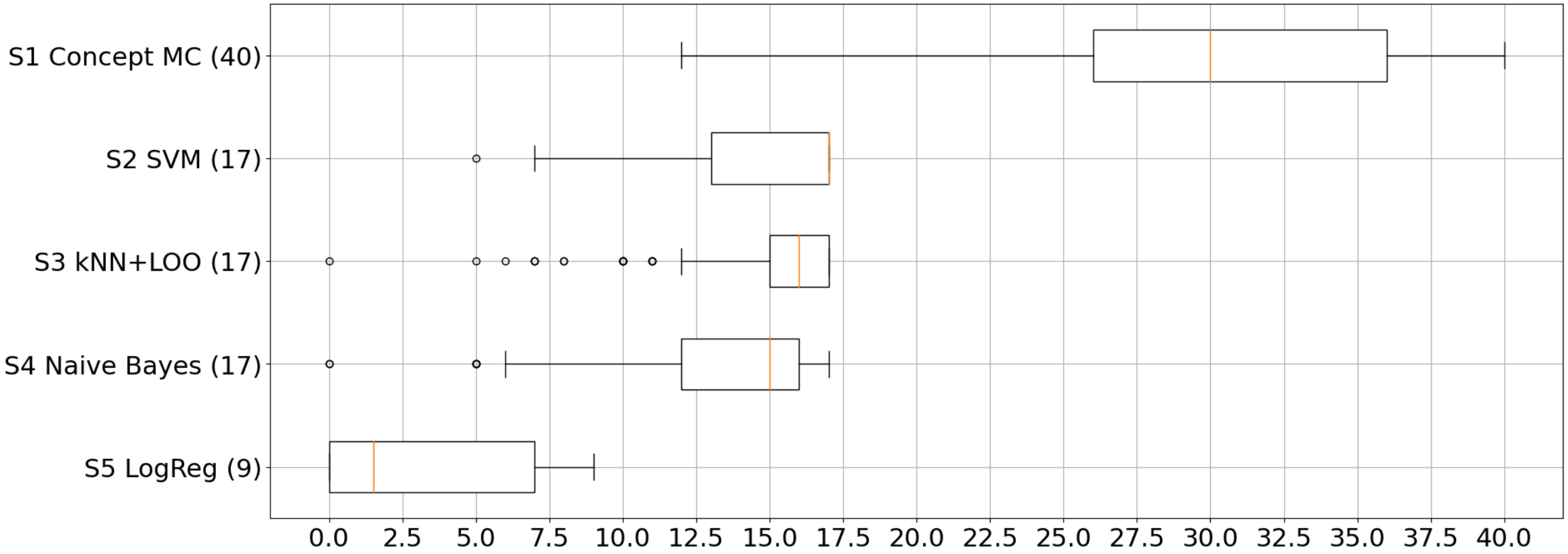
Lecture 7.1: Neural Networks (II)

CS 434

Midterm Analysis



Midterm Analysis



Midterm

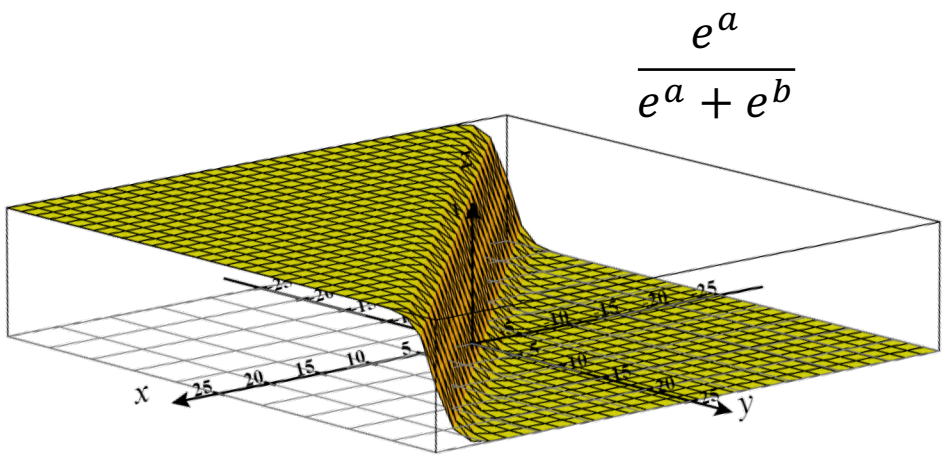
RECAP

From Last Lecture

A Note About The Softmax Function

This functional form has a name – the softmax function. It turns a real-valued vector $\mathbb{R}^d$ to a categoric distribution -- i.e. each element in $(0,1)$ and it sums to 1.

$$Softmax\left(\begin{bmatrix} z_1 \\ z_2 \\ \vdots \\ z_d \end{bmatrix}\right) = \begin{bmatrix} e^{z_1} \\ e^{z_2} \\ \vdots \\ e^{z_d} \end{bmatrix} \frac{1}{\sum_{i=1}^{d} e^{z_i}}$$



"**max**" – amplifies probability of largest x_i
"**soft**" – differentiable and still assigns non-zero probability to other entries

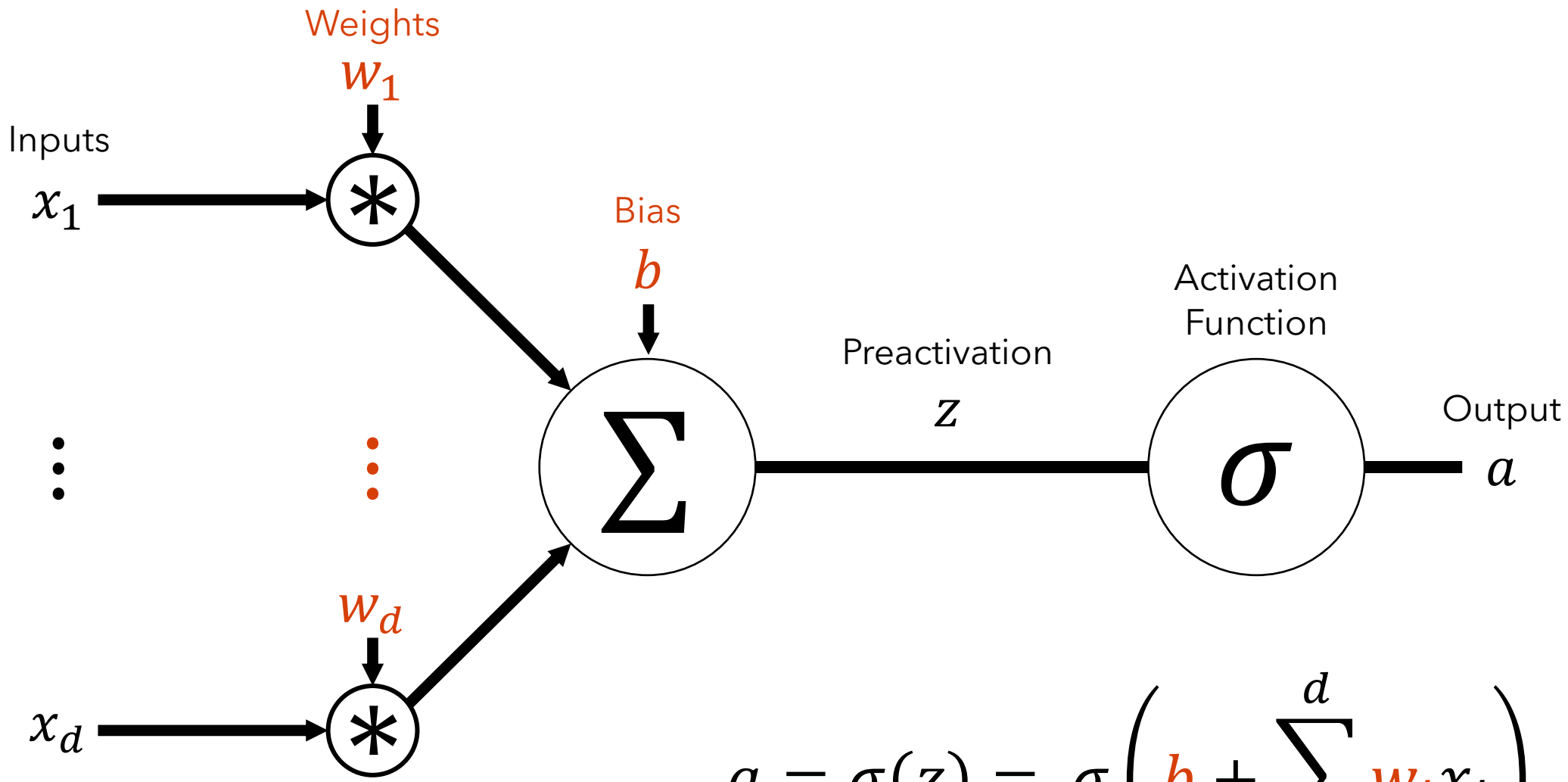
https://colab.research.google.com/drive/1aCyd04JhiLZEmDlTmsty_fmd-Ud2pOda?usp=sharing&authuser=1#scrollTo=XrU9awzZBqSk

New Topic: Neural Networks

Neural networks are a class of biologically-inspired models for classification and regression

- **Key Concepts:**
  - Artificial Neurons
    - Activation Functions
  - Neural Networks
    - Linear Layers
    - Universal Approximators (in the limit)
  - Training Neural Networks
    - Backpropagation
    - Stochastic Gradient Descent

The Artificial Neuron



$$a = \sigma(z) = \sigma\left(b + \sum_{i=1}^{d} w_i x_i\right)$$

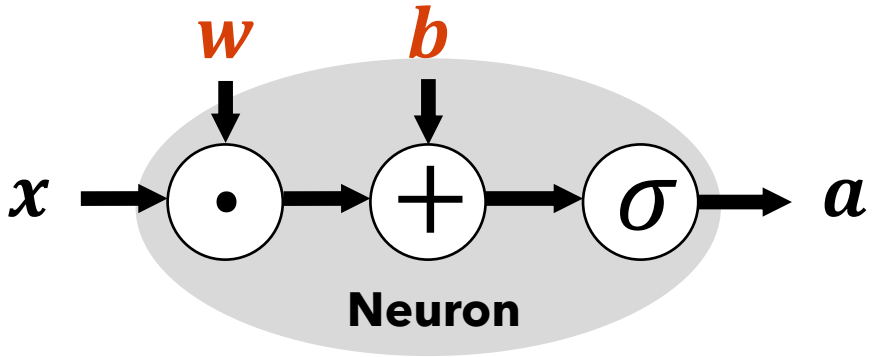
The Humble Neuron Vectorized

$$a = \sigma\left(b + \sum_{i=1}^{d} w_i x_i\right) = \sigma(b + \boldsymbol{w}^T \boldsymbol{x})$$

$$\underset{1\times1}{\boldsymbol{a}} \quad \begin{bmatrix} a \end{bmatrix} = \sigma\left(\begin{bmatrix} w_1 \\ \vdots \\ w_d \end{bmatrix}^{\boldsymbol{T}} \begin{bmatrix} x_1 \\ \vdots \\ x_d \end{bmatrix} + \underset{1\times1}{\boldsymbol{b}} \begin{bmatrix} b \end{bmatrix} \right)$$

$\boldsymbol{w}$ $d\times1$ $\quad$ $\boldsymbol{x}$ $d\times1$

Neuron



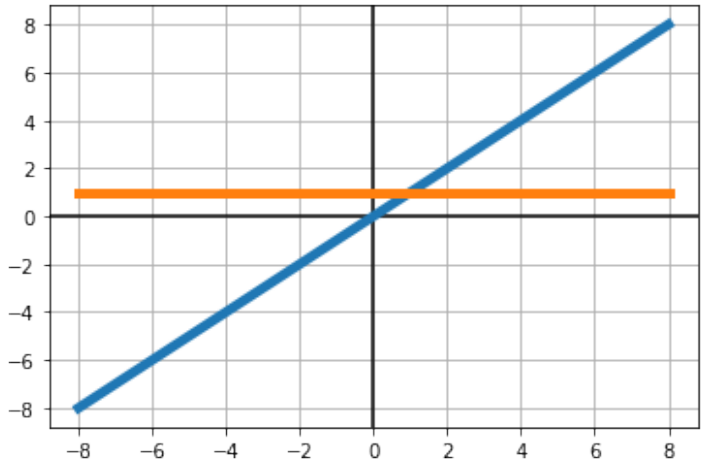
Neuron is a linear function of its input followed by a (typically) non-linear activation function to produce output. $a = \sigma\left(\boldsymbol{w}^{\boldsymbol{T}}\boldsymbol{x} + b\right)$

Hyperparameters : activation function (σ)

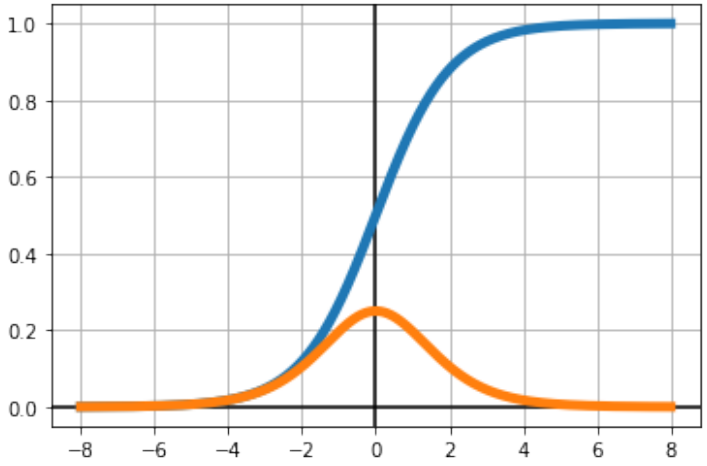
Learnable Parameters: $\mathbf{w} \in \mathbb{R}^d$ and $\mathbf{b} \in \mathbb{R}$

Activation Functions σ

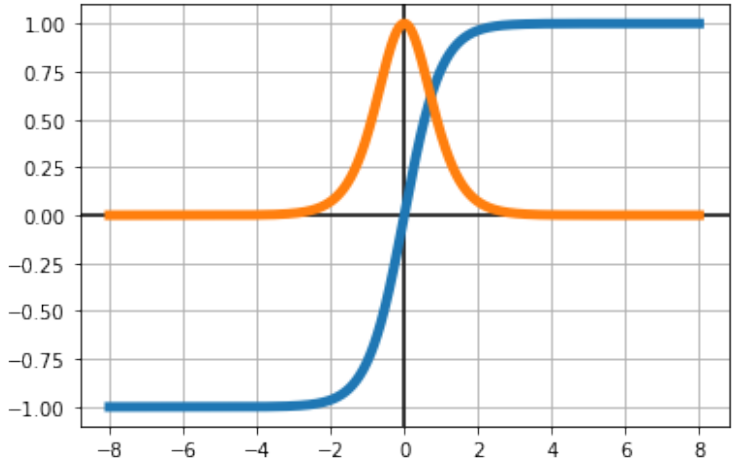
None $\sigma(z) = z$



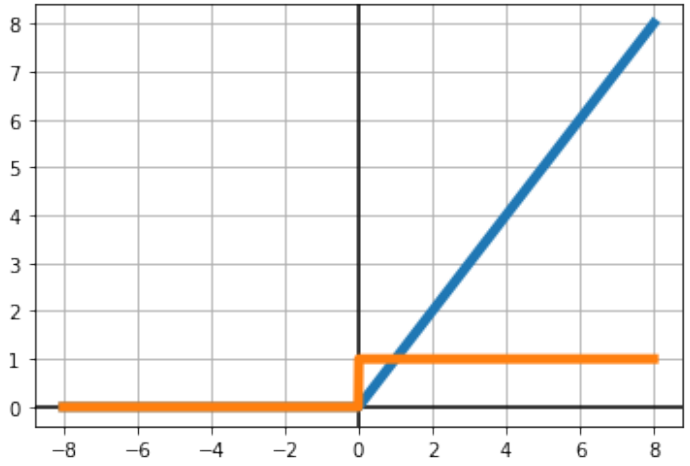
Sigmoid $\sigma(z) = \frac{1}{1+e^{-z}}$



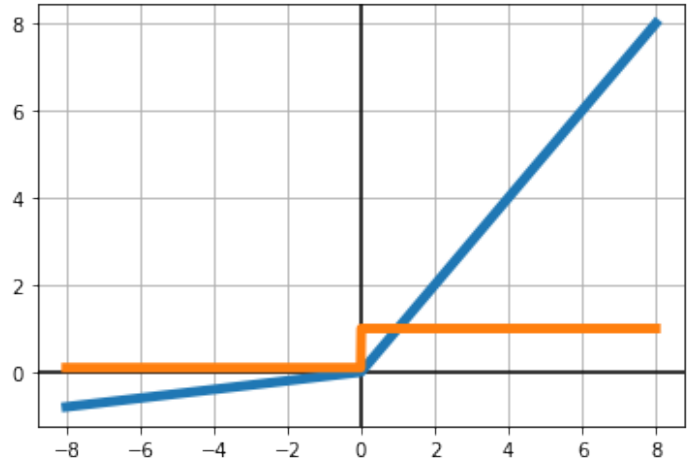
tanh $\sigma(z) = \frac{e^{z}-e^{-z}}{e^{z}+e^{-z}}$



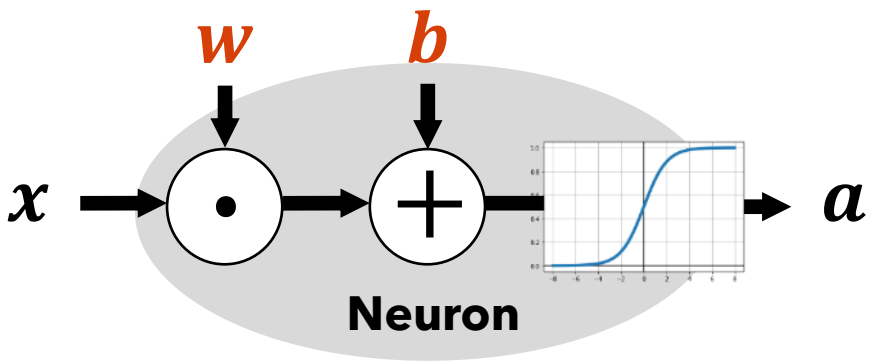
ReLU $\sigma(z) = \max(0, z)$



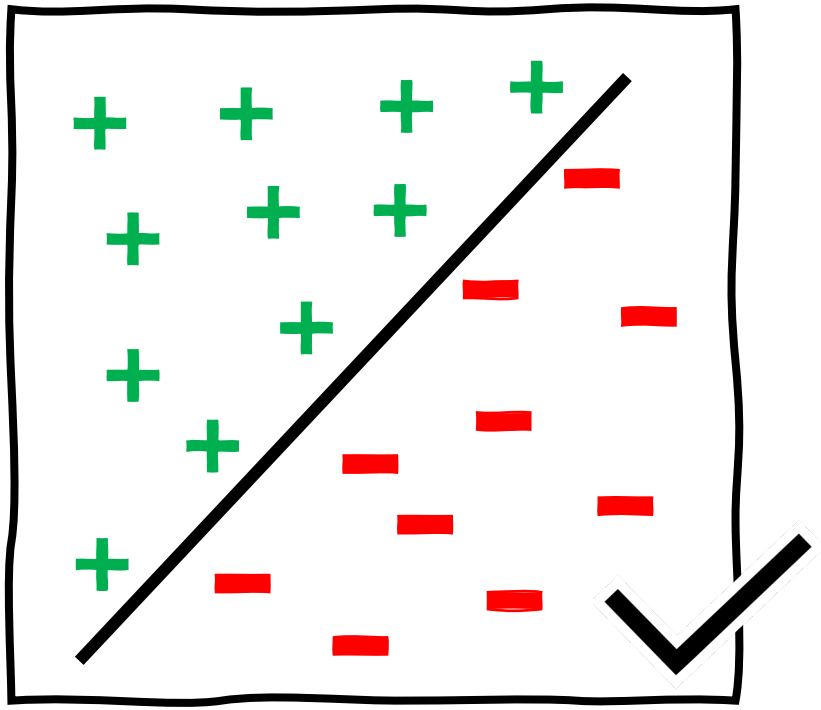
Leaky ReLU $\sigma(z) = \begin{cases} \alpha z & \text{if } z < 0 \\ z & \text{if } z \geq 0 \end{cases}$

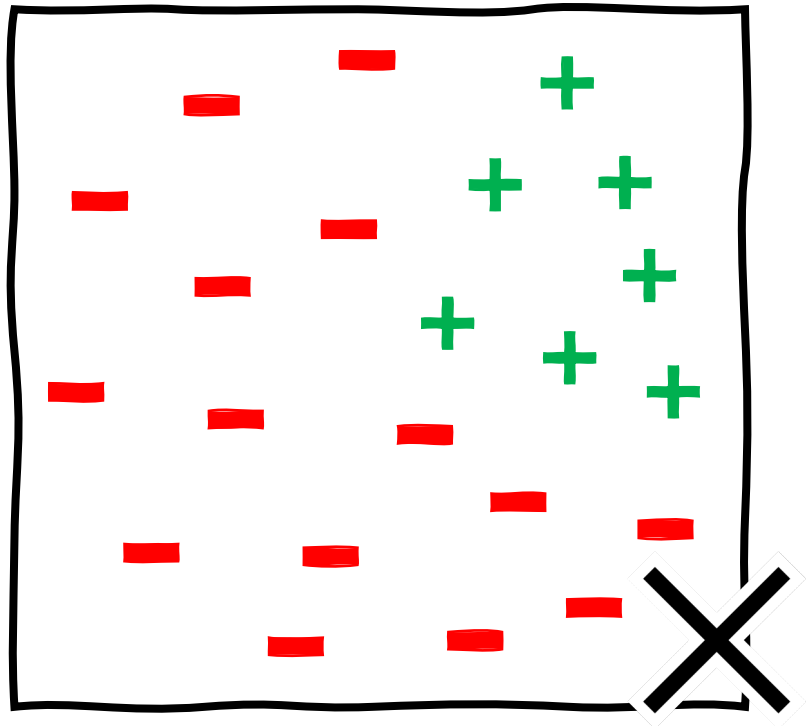


What can one neuron do?



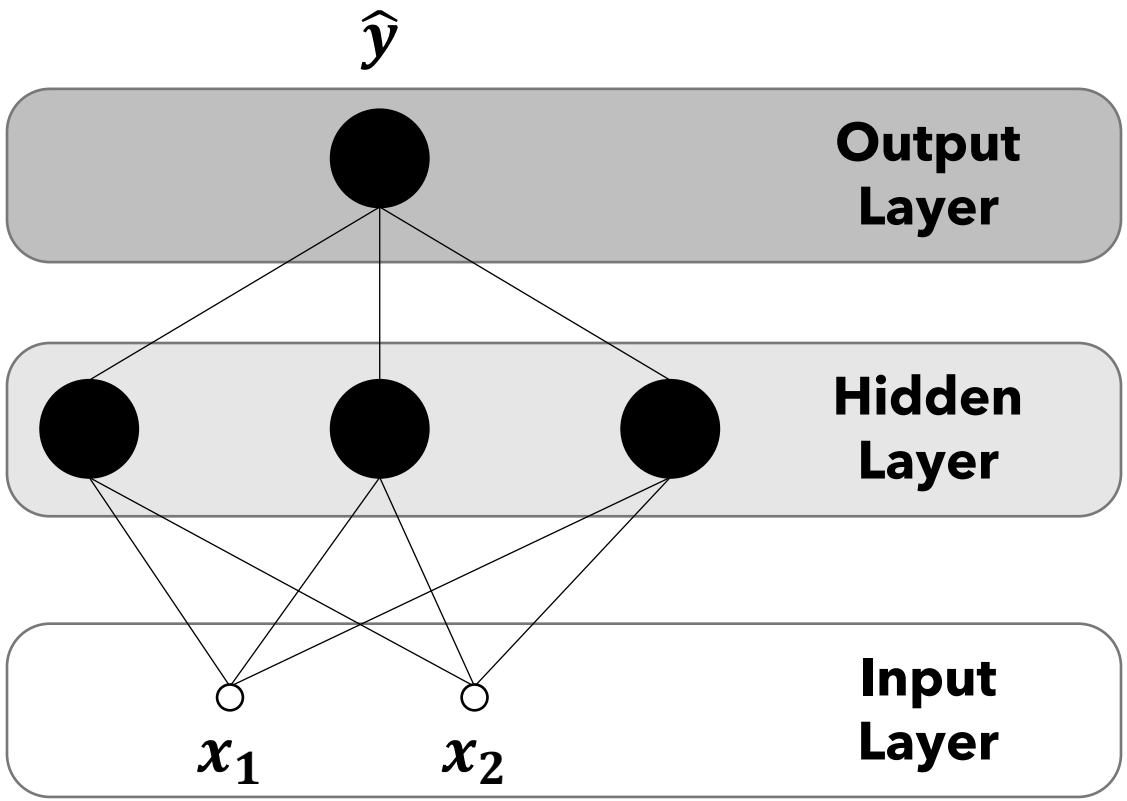
$$\sigma(b + w^T x) = \frac{1}{1 + e^{-(w^T x + b)}}$$





Basic Multilayer Neural Network

A Neural Network is a set of connected neurons. A very typical arrangement is a feed-forward multilayer neural network like the one shown below.



Each layer receives its input from the previous layer and forwards its output to the next – thus the **feed-forward** description.

The layers of neurons between the input and output are referred to as **hidden layers**.

- This network for instance is **a 2-layer neural** network (1 hidden and 1 output)
- Number of neurons in a layer is referred to as its width.

Activation functions in different layers can be heterogenous.

- Output layer's activation is task dependent
  - Linear for regression
  - Sigmoid or softmax for classification

Today's Learning Objectives

Be able to answer:

- What is a fully-connected layer?
- What is a loss function?

- (Quick review of vector calculus)

- What is forward / reverse mode differentiation?
- How does backpropagation work?

- What is a computation graph?

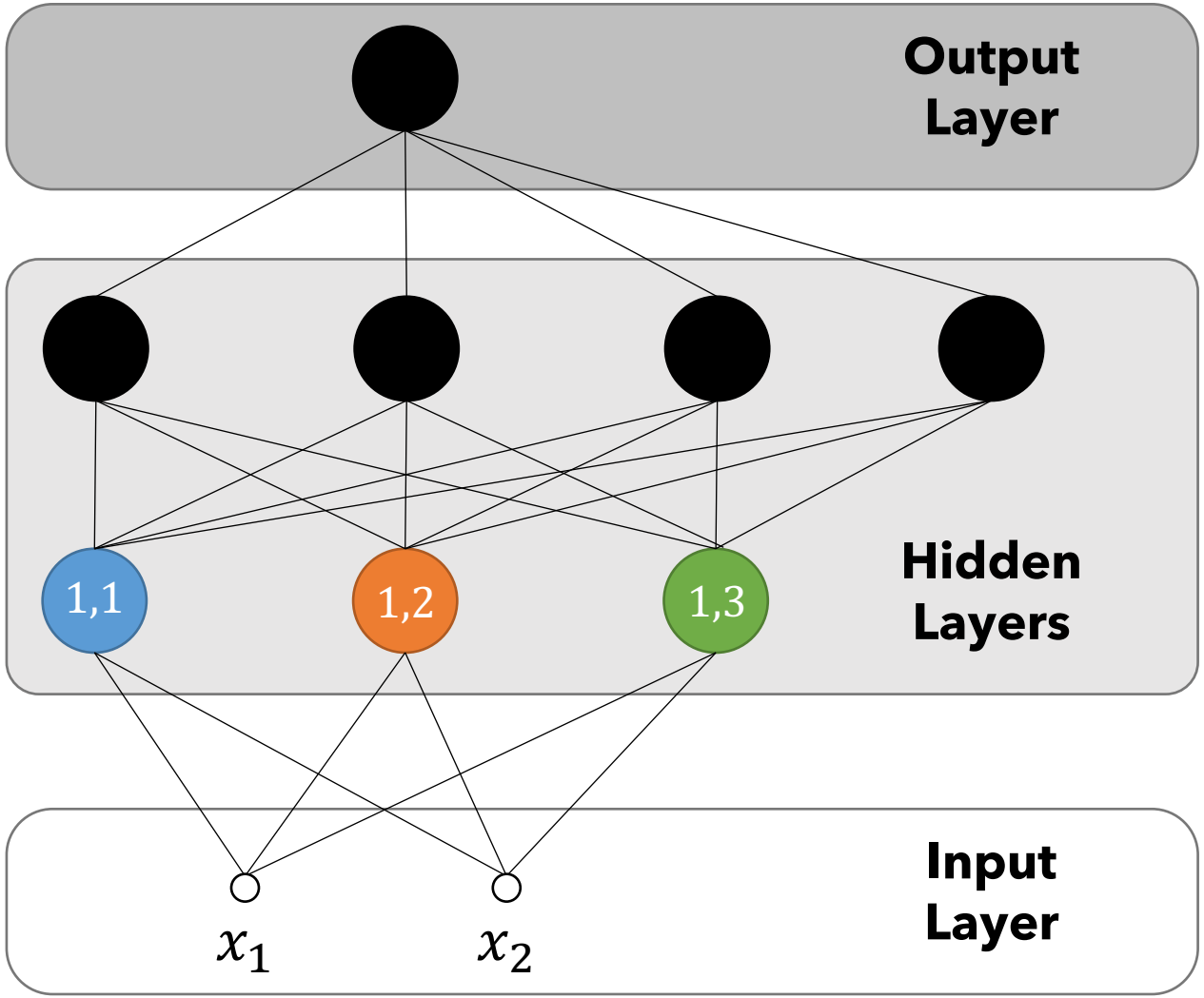
Working on Notation for Feed-Forward Neural Networks

To tidy up our notation, I want to describe the output of a whole layer at once rather than write out a bunch of equations for each neuron.

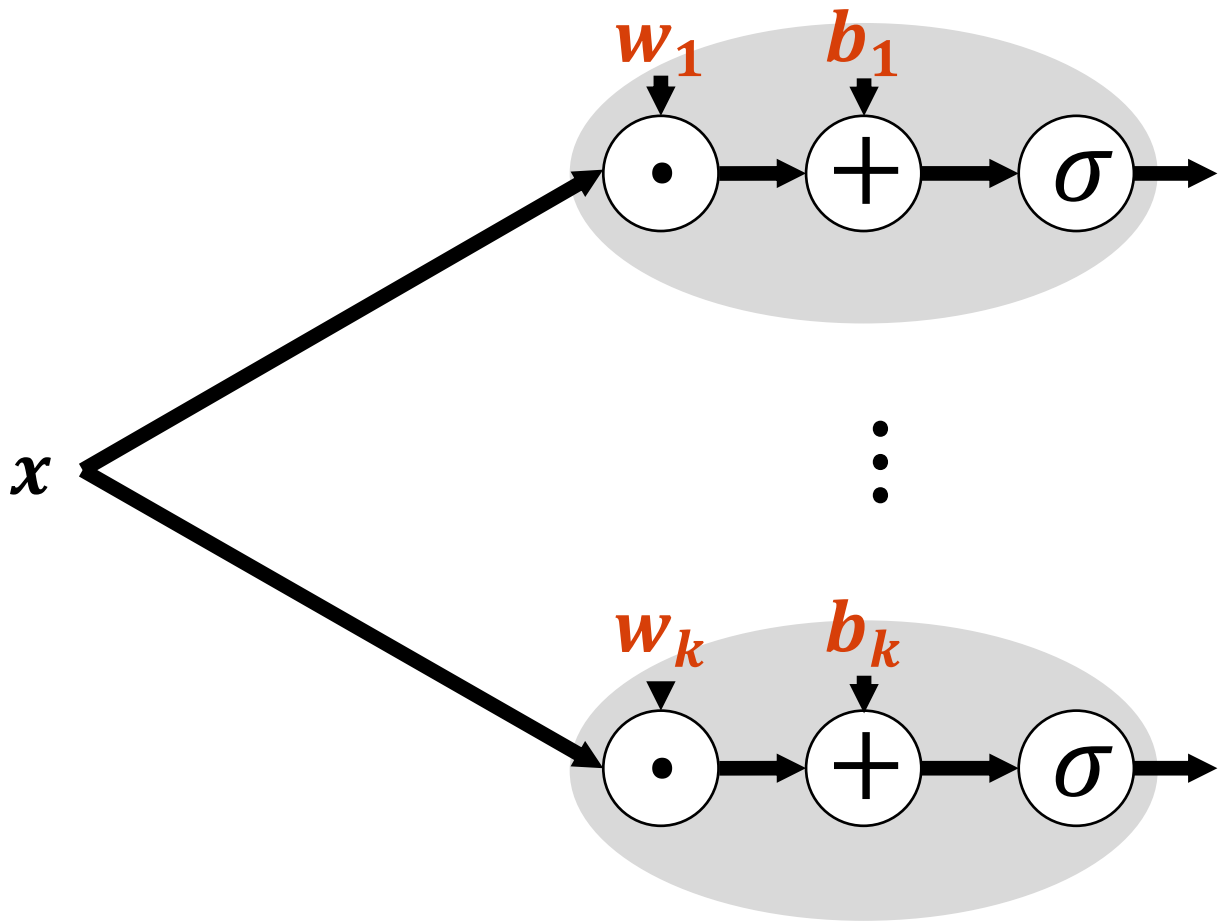
$$a_{11} = \sigma(b_{11} + w_{11}^T x)$$

$$a_{12} = \sigma(b_{12} + w_{12}^T x)$$

$$a_{13} = \sigma(b_{13} + w_{13}^T x)$$



Fully-Connected Layer of Neurons



$$a_1 = \sigma(b_1 + w_1^T x)$$

$$a_k = \sigma(b_k + w_k^T x)$$

Fully-Connected Layer of Neurons Vectorized

$$\boldsymbol{a} = \sigma(\boldsymbol{b} + \boldsymbol{W}\boldsymbol{x})$$

$$\underset{k\times 1}{\boldsymbol{a}} = \begin{bmatrix} a^{(1)} \\ \vdots \\ a^{(k)} \end{bmatrix} = \sigma\left(\underset{k\times 1}{\boldsymbol{b}} \begin{bmatrix} b^{(1)} \\ \vdots \\ b^{(k)} \end{bmatrix} + \underset{k\times d}{\boldsymbol{W}} \begin{bmatrix} w_{11} & w_{12} & \cdots & w_{1d} \\ \vdots & \vdots & \cdots & \vdots \\ w_{k1} & w_{k2} & \cdots & w_{kd} \end{bmatrix} \underset{d\times 1}{\boldsymbol{x}} \begin{bmatrix} x_1 \\ \vdots \\ x_d \end{bmatrix} \right)$$

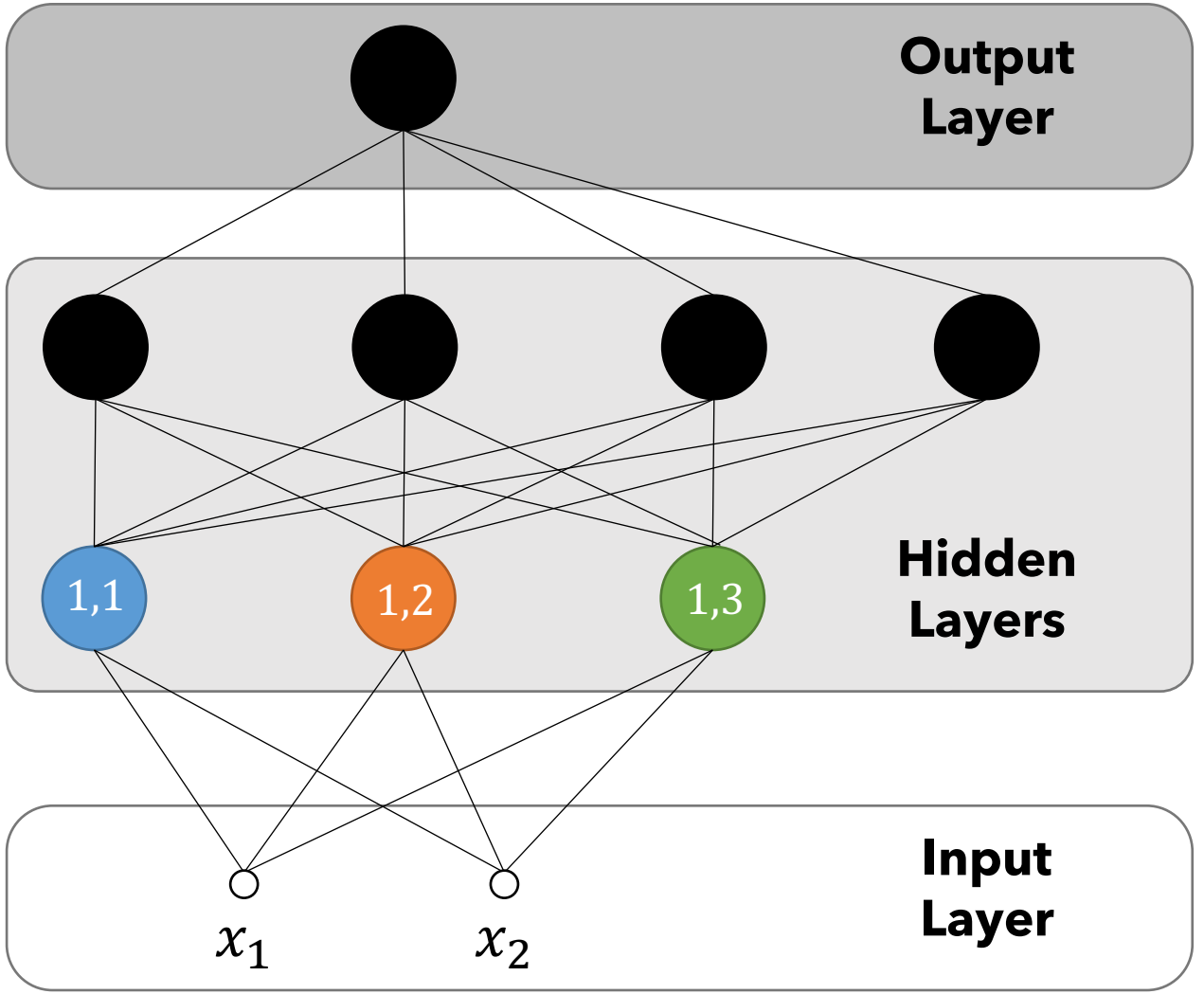
Activation applied elementwise

Working on Notation for Feed-Forward Neural Networks

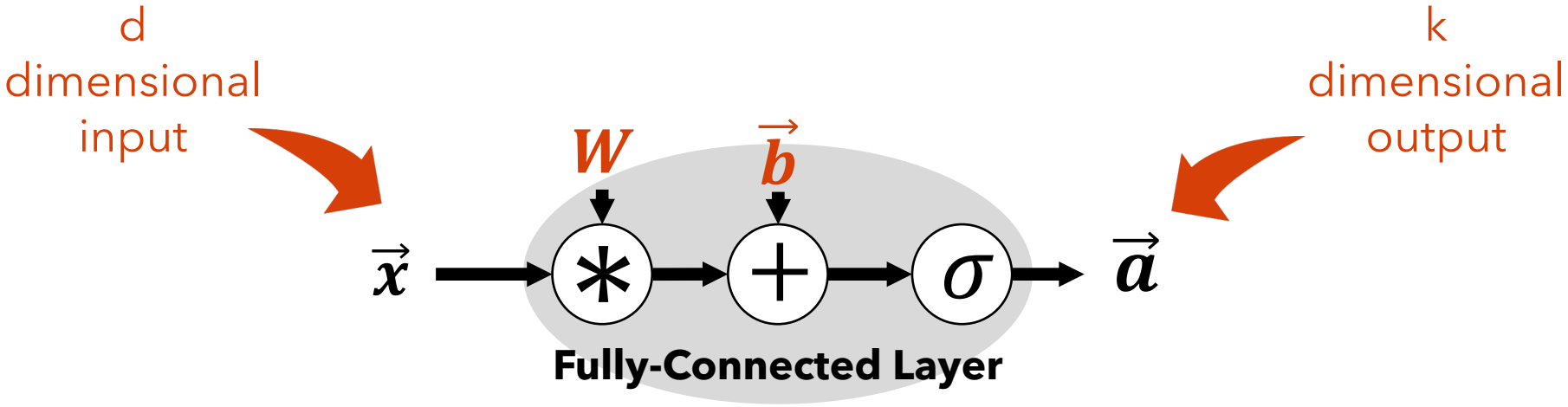
$$a_{11} = \sigma(b_{11} + w_{11}^T x)$$
$$a_{12} = \sigma(b_{12} + w_{12}^T x)$$
$$a_{13} = \sigma(b_{13} + w_{13}^T x)$$

$$\underset{3\times1}{a^{(1)}} = \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} = \sigma\left(\underset{3\times1}{b^{(1)}} + \underset{3\times2}{W^{(1)}} \underset{2\times1}{x} \right) = \sigma\left(\begin{bmatrix} b_{11} \\ b_{12} \\ b_{13} \end{bmatrix} + \begin{bmatrix} — & w_{11}^T & — \\ — & w_{12}^T & — \\ — & w_{13}^T & — \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \right)$$

$$a^{(1)} = \sigma(b^{(1)} + W^{(1)} x)$$



Fully-Connected Layer



Hyperparameters : width of layer (k) and choice of activation functions (σ)

Learnable Parameters: $W \in \mathbb{R}^{k \times d}$ and $\vec{b} \in \mathbb{R}^{k}$

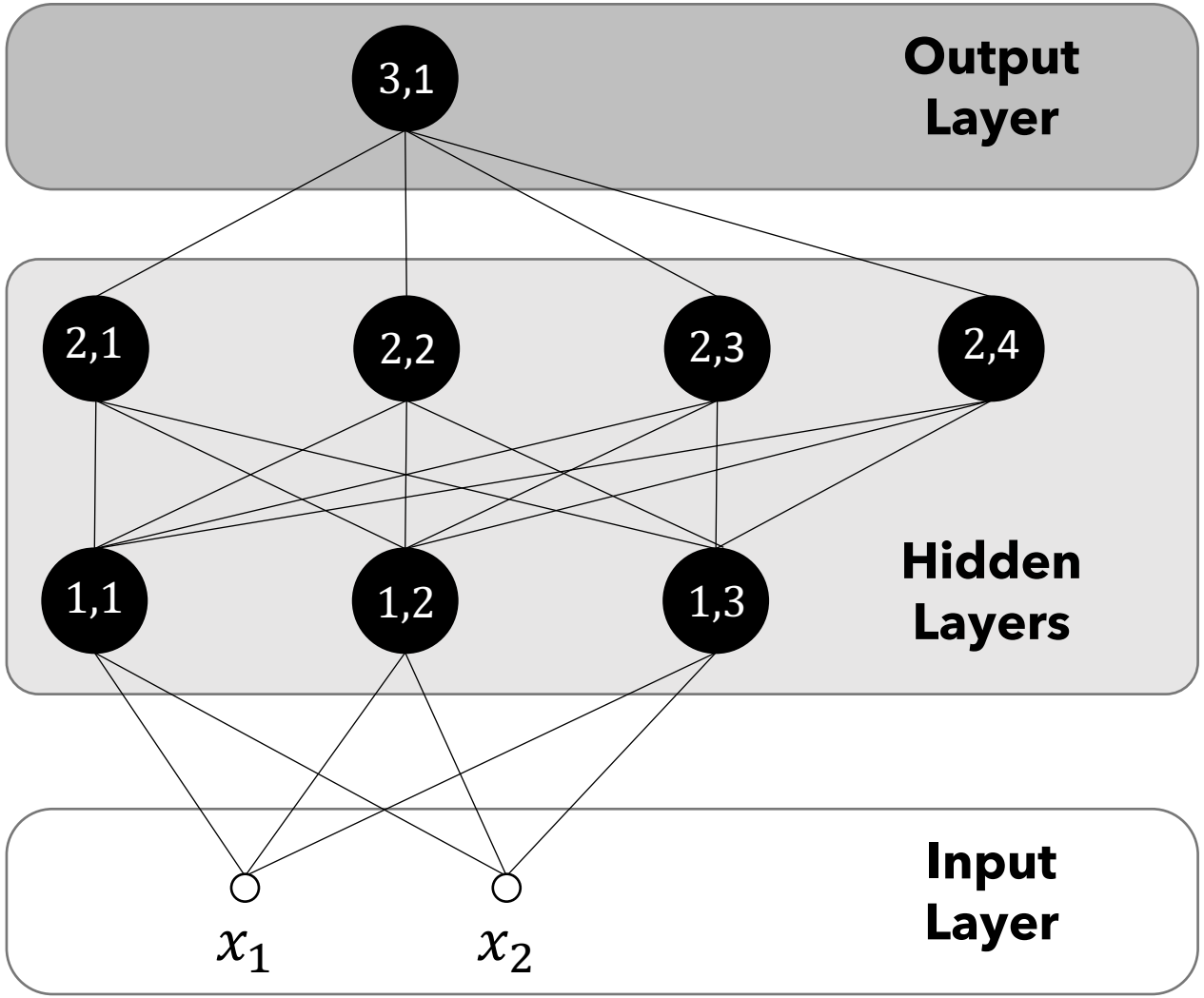
Working on Notation for Feed-Forward Neural Networks

Suppose a network has L layers:

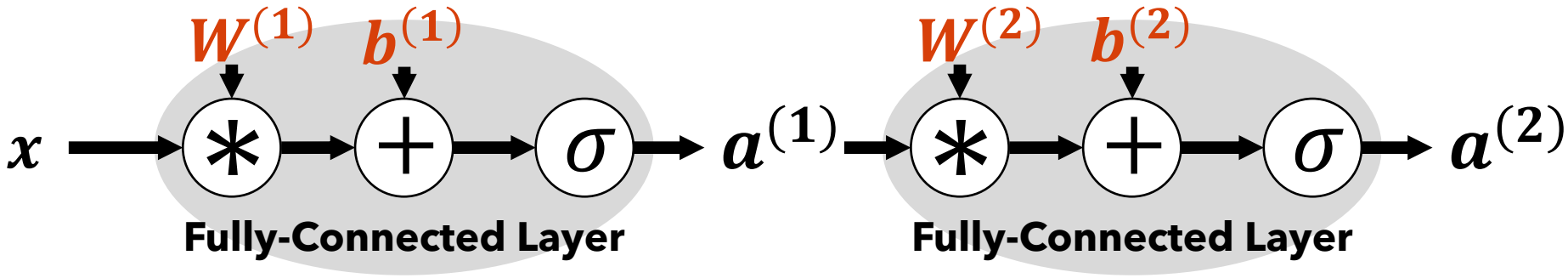
Let $W^{(1)}, W^{(2)}, \ldots, W^{(L)}$ be **weight matrices** for layers 1,2,…,L.

Let $\boldsymbol{b}^{(1)}, \boldsymbol{b}^{(2)}, \ldots, \boldsymbol{b}^{(L)}$ be **bias vectors** for layers 1,2,…,L.

Let $a^{(1)}, a^{(2)}, \ldots, a^{(L)}$ be **vectors of activations** for layers 1,2,…,L.



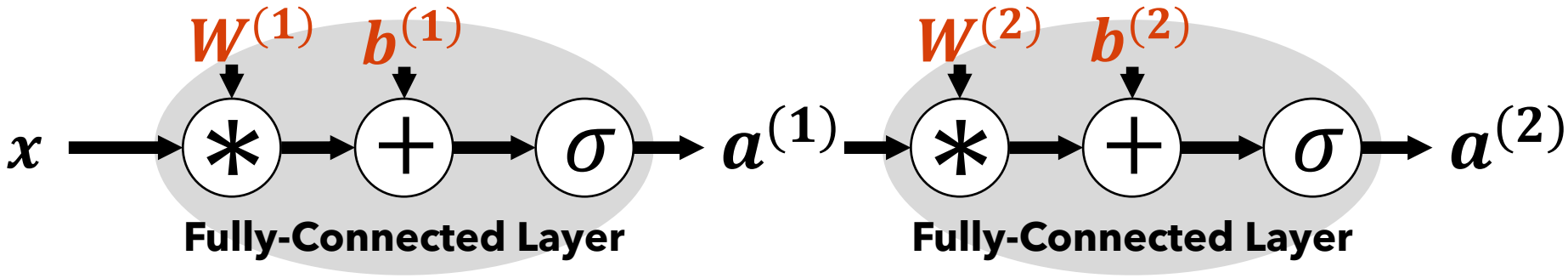
Multi-layer Networks



$$\boldsymbol{a}^{(1)} = \sigma(\boldsymbol{b}^{(1)} + \boldsymbol{W}^{(1)}\boldsymbol{x})$$

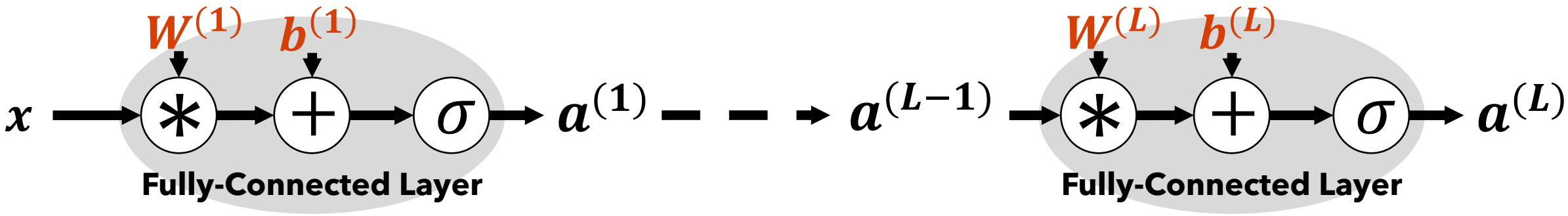
$$\boldsymbol{a}^{(2)} = \sigma(\boldsymbol{b}^{(2)} + \boldsymbol{W}^{(2)}\boldsymbol{a}^{(1)})$$

Multi-layer Networks



$$a^{(2)} = \sigma(b^{(2)} + W^{(2)}\sigma(b^{(1)} + W^{(1)}x))$$

Multi-layer Networks



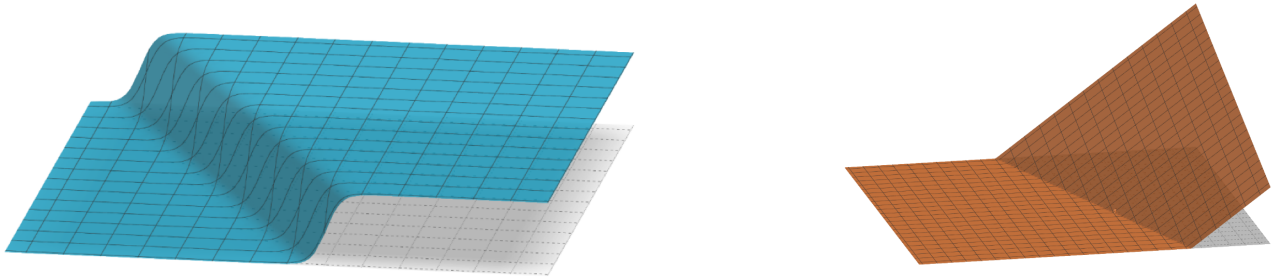
Hyperparameters : depth in layers (L), width of each layer ($k_1, \ldots, k_L$), and choice of activation functions for each layer ($\sigma_1, \ldots, \sigma_L$)

Learnable Parameters: $W_1 \in \mathbb{R}^{k_1 \times d}, W_{i>1} \in \mathbb{R}^{k_i \times k_{i-1}}$ and $\mathrm{b_i} \in \mathbb{R}^{k_i}$

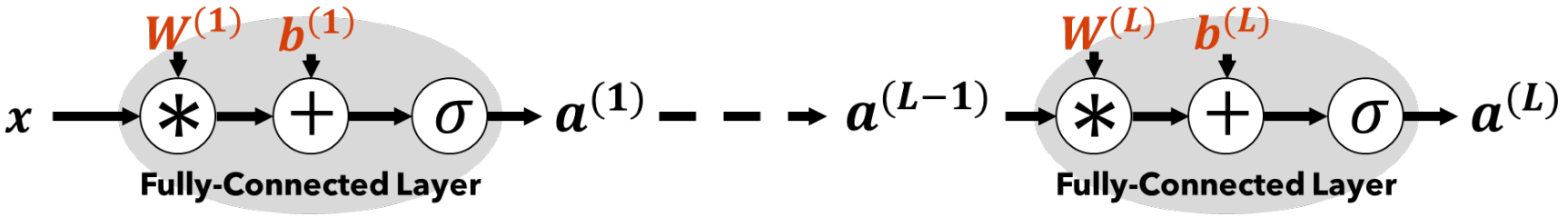
Let's Review Where We are

I introduced a neuron last class and said they weren't very expressive on their own.

- Sigmoid neuron and ReLU neuron outputs below (2d inputs):



Had the great idea to combine them together to make them more powerful – called this a neural network. One common configuration is a feed-forward neural network.



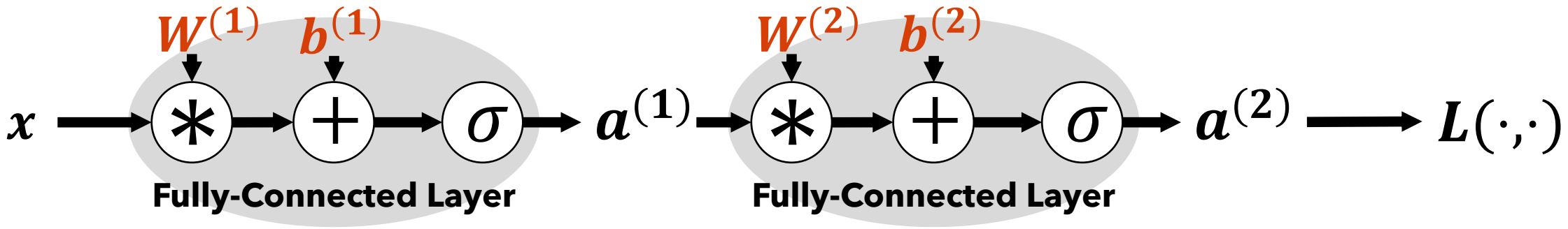
Question remaining: How do I actually get this complicated looking thing to do something useful?

What are Neural Networks good for?

Non-linear Function Learning: As we saw in the demo yesterday, neural networks can learn non-linear decision boundaries. But unlike the kernel methods / basis functions we used before, we don't need to specify the exact from of this non-linear function.

Theoretically Universal Approximators: For any continuous function F over a bounded subspace of D-dimensional space, there exists a two-layer neural network $\hat{F}$ with a finite number of hidden units that approximates F arbitrarily well. That is to say, for all x in the domain of F, $\left|F(x) - \hat{F}(x)\right| < \epsilon$

Incredibly Flexible: Can be used for classification or regression. Or just any problem with a differentiable loss function.



Loss Function *L*– A function measuring "how bad" a network's output is, usually relative to some gold-standard for what the output should be.

Loss Functions - Regression

Squared Error

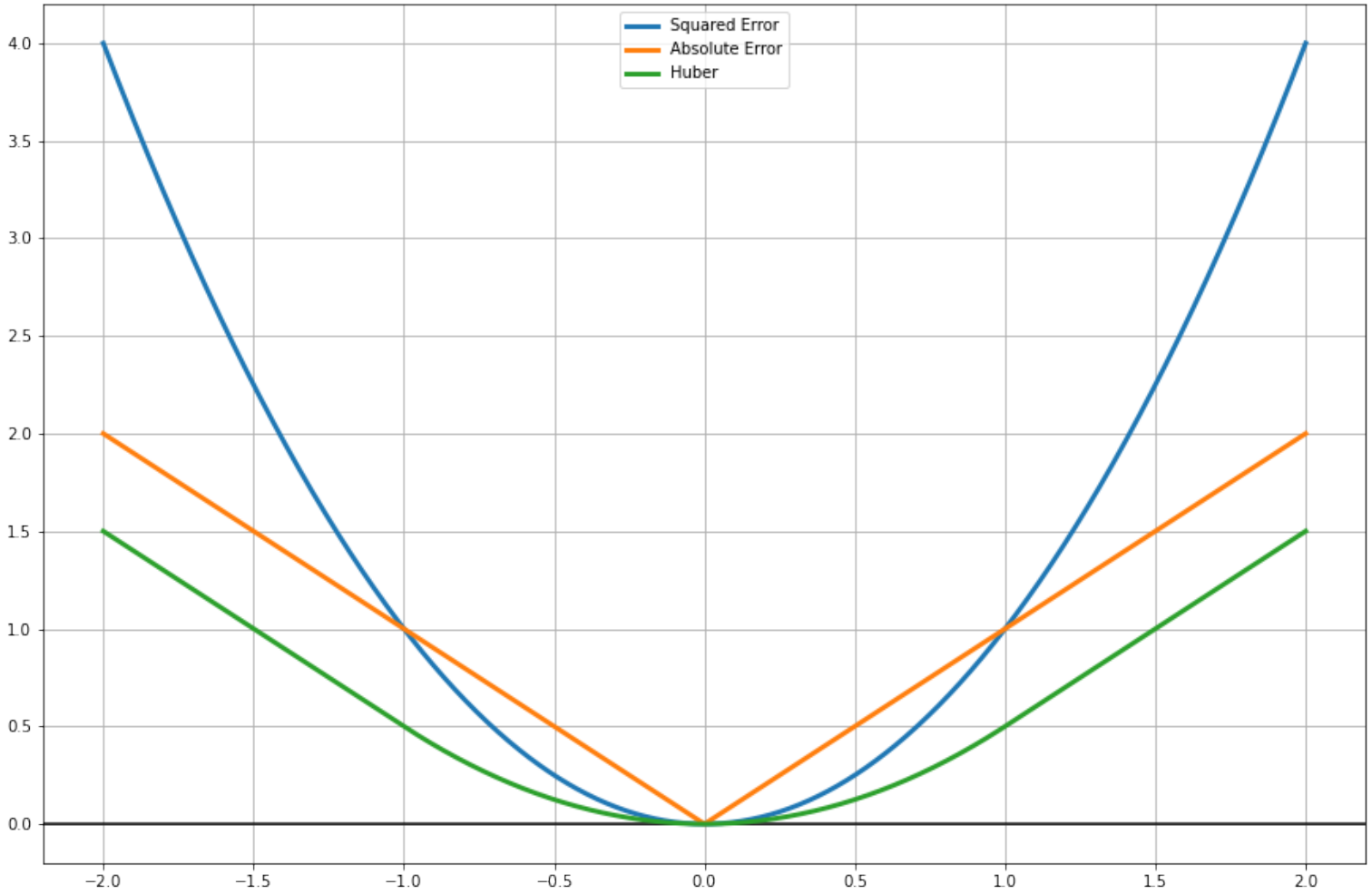
$$L(a, b) = (a - b)^2$$

Absolute Error

$$L(a, b) = |a - b|$$

Huber

$$L(a, b) = \begin{cases} \frac{1}{2}(a-b)^2 & \text{if } |a-b| < \delta \\ \delta|a-b| - \frac{1}{2}\delta^2 & \text{else} \end{cases}$$



Loss Functions - Classification

Cross Entropy $(\vec{a} \in \Delta^C \quad \vec{b} \in \Delta^C)$

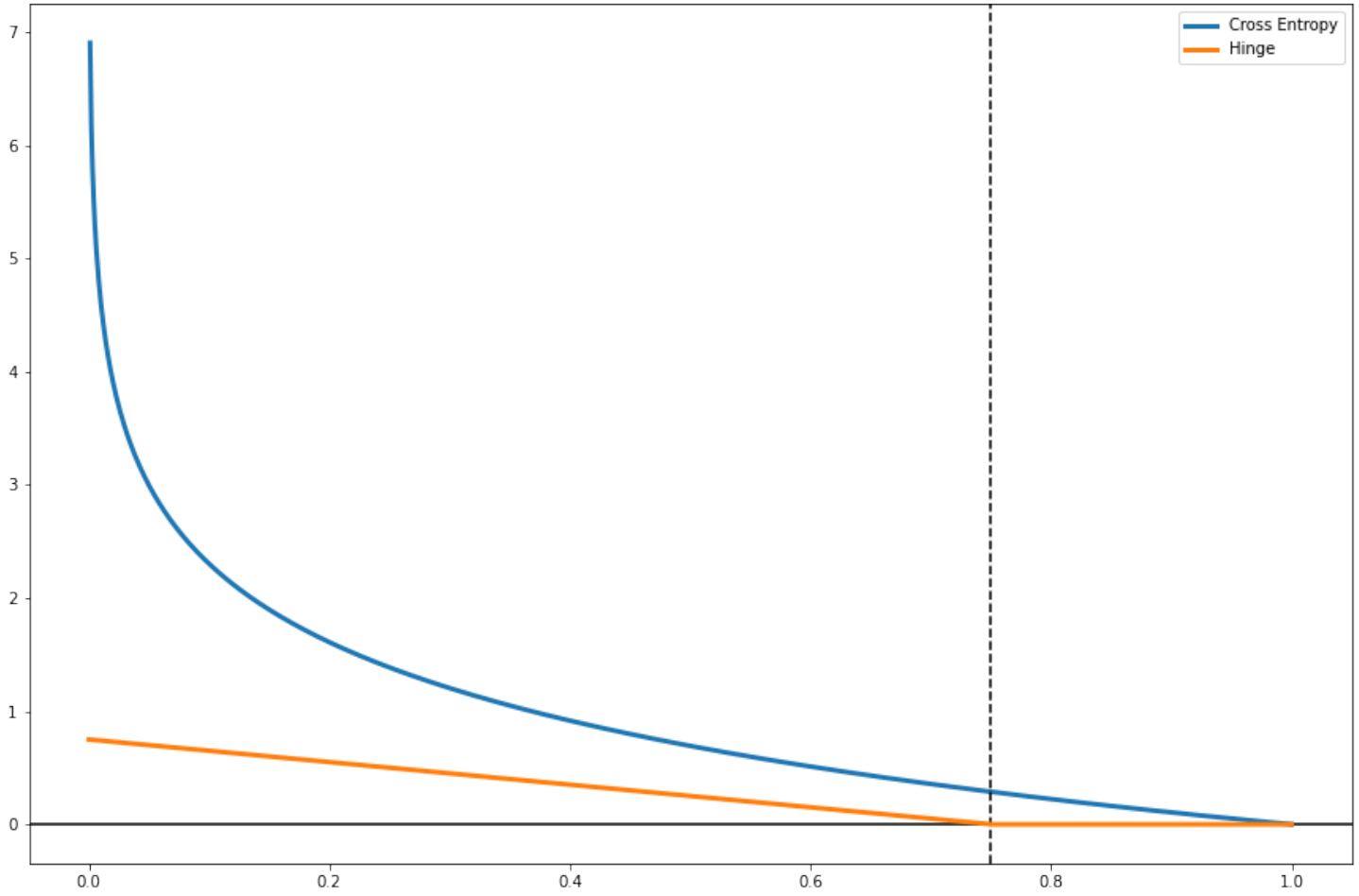
$$L(\vec{a}, \vec{b}) = -E_b[\log(a)] = -\sum_{i=0}^{C} b_i \log(a_i)$$

$$\rightarrow \; if \; \vec{b} = \vec{1}_c \; \rightarrow \; -log(a_c)$$

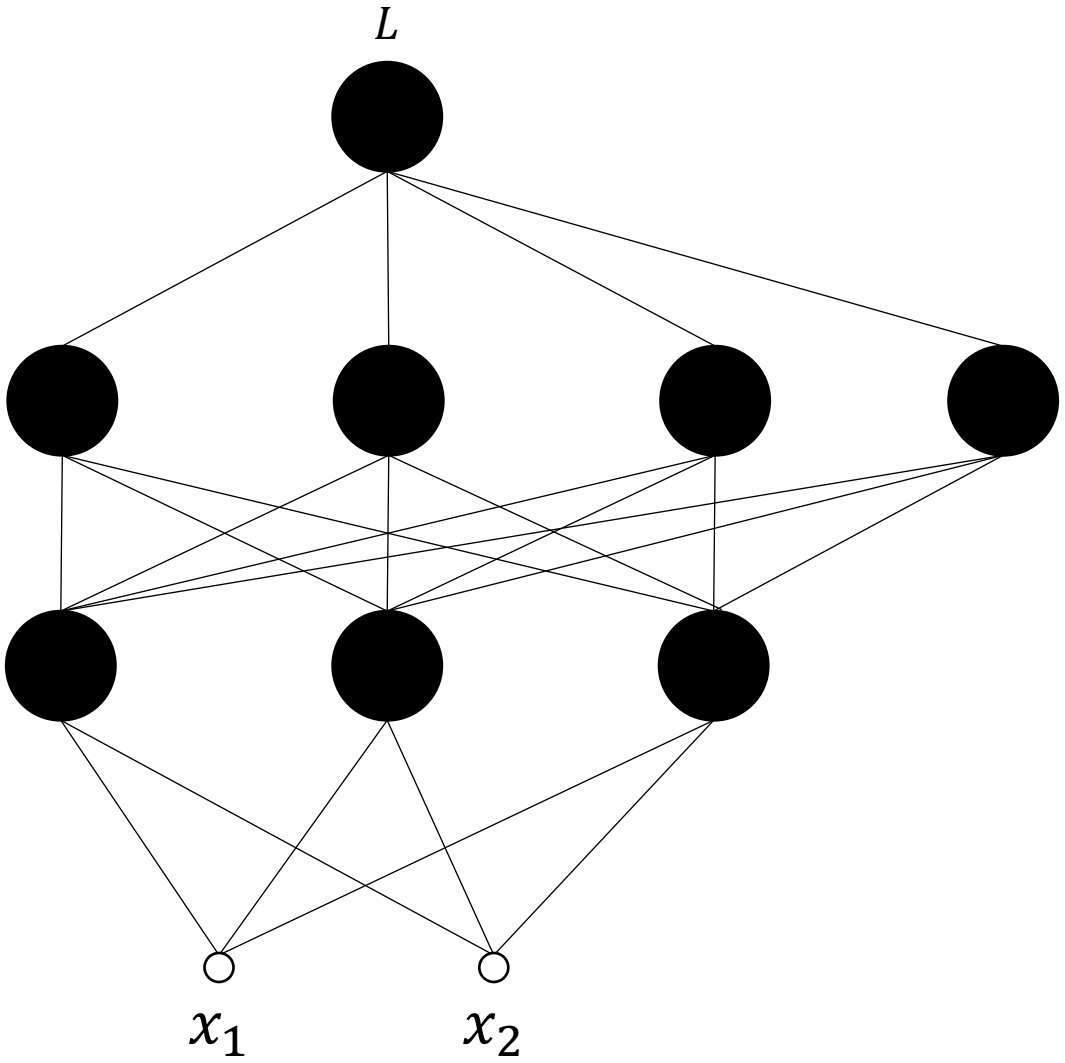
Hinge $(\vec{a} \in \mathbb{R}^C \quad \vec{b} = \vec{1}_c)$

$$L(\vec{a}, \vec{b}) = \sum_{i=0}^{C} \max(0, \delta - b_i a_i)$$

$$\rightarrow \; if \; \vec{b} = \vec{1}_c \; \rightarrow \max(0, \delta - b_c a_c)$$



Intuition For Neural Network Training



Neural Network Training In Words

Forward Pass

1) For each training example:
   - Compute and store all activations
   - Compute loss

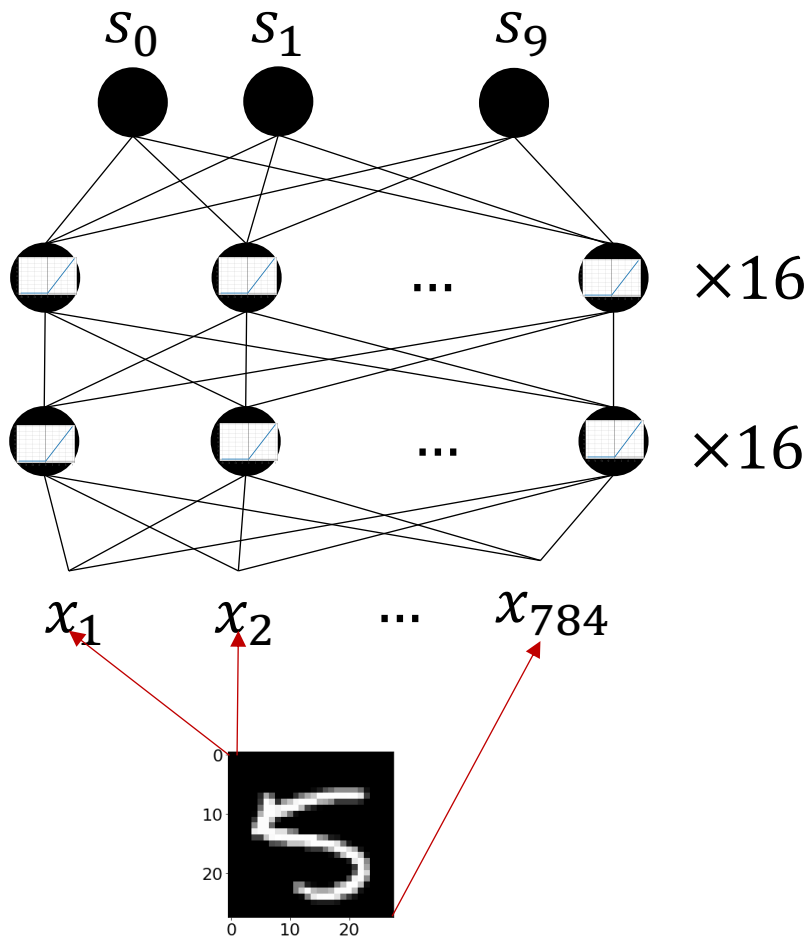
Backward Pass

2) Compute gradient of the loss with respect to all network parameters
   - Will do this efficiently with an algorithm called **backpropagation**

Update

Take a step of gradient descent to minimize the loss

An Example from the Homework



Input: an image represented as a 784 dimensional vector of real values

Model: 2 hidden layers with ReLU activations and widths of 16 neurons

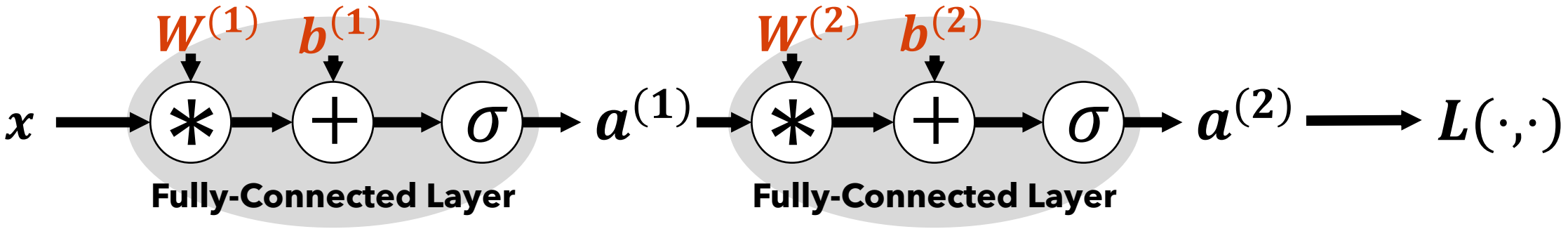
Output: scores for each digit class. Can compute probability for each as:

$$p_{0|x} = \frac{e^{s_0}}{\sum_{c=0}^{9} e^{s_c}}$$

Loss: cross entropy, for this example:

$$\mathcal{L} = -log\ p_{5|x}$$

Time to Pay the Math Tax



$$\frac{\delta L}{\delta W^{(1)}} = ?$$

Recall from Machine Learning / Calculus / Optimization

Our loss function $L: \theta \rightarrow \mathbb{R}$ defines a "loss surface".
Optimization is about getting as low as possible on this surface.

Recall: Gradient (vector of partial derivatives) point in the direction of steepest ascent.

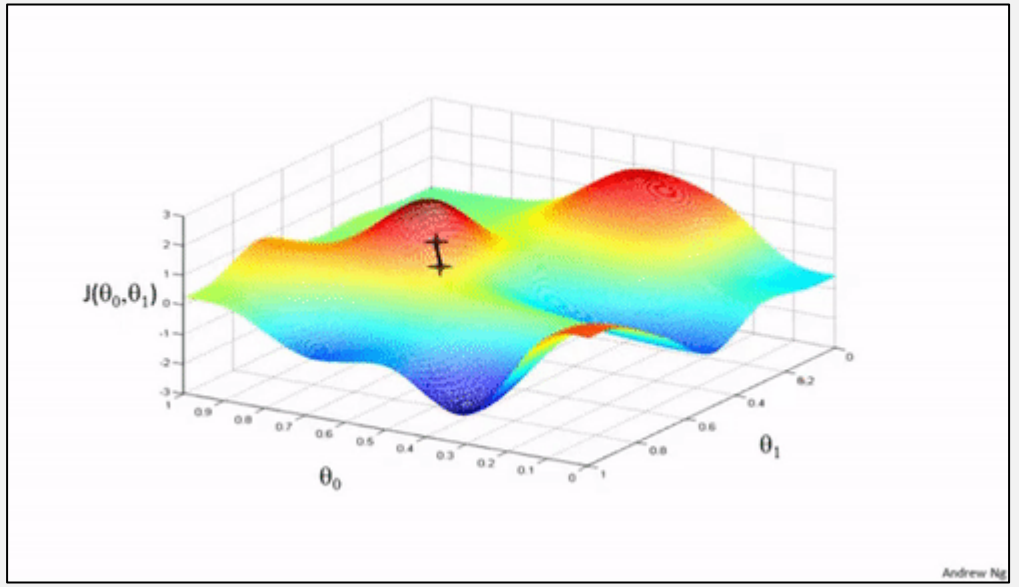
Stochastic Gradient Descent Algorithm

$\theta \leftarrow random(\quad)$

$Sample\ batch\ B$

while $|\nabla_\theta L| > \epsilon\ or\ iters\ remain$

$\theta = \theta - \alpha \nabla_\theta L(B; \theta)$



Today's Learning Objectives

Be able to answer:

- ~~What is a fully-connected layer?~~
- ~~What is a loss function?~~

- (Quick review of vector calculus)

- What is forward / reverse mode differentiation?
- How does backpropagation work?

- What is a computation graph?

Recall from Calculus: Scalar Partial Derivatives

Derivative of f with respect to x:

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

Partial Derivatives of f with respect to x and y:

$$\frac{\delta}{\delta x} f(x,y) = \lim_{h \to 0} \frac{f(x+h,y) - f(x,y)}{h}$$

$$\frac{\delta}{\delta y} f(x,y) = \lim_{h \to 0} \frac{f(x,y+h) - f(x,y)}{h}$$

Notation for Vector Calculus

Scalars: $y, x \in \mathbb{R}$ Vectors: $\vec{\boldsymbol{y}} \in \mathbb{R}^d, \vec{\boldsymbol{x}} \in \mathbb{R}^c$ Matrices: $\mathbf{X} \in \mathbb{R}^{m \times n}, \boldsymbol{Y} \in \mathbb{R}^{o \times p}$

Our convention: Numerator dim → rows, Denominator dim → cols

$$\frac{\boldsymbol{\delta \text{vector}}}{\boldsymbol{\delta \text{scalar}}}$$

$$\frac{\delta \vec{\boldsymbol{y}}}{\delta x} = \begin{bmatrix} \frac{\delta y_1}{\delta x} \\ \frac{\delta y_2}{\delta x} \\ \vdots \\ \frac{\delta y_d}{\delta x} \end{bmatrix}_{d \times 1}$$

$$\frac{\boldsymbol{\delta \text{scalar}}}{\boldsymbol{\delta \text{vector}}}$$

$$\frac{\delta y}{\delta \vec{\boldsymbol{x}}} = \begin{bmatrix} \frac{\delta y}{\delta x_1} & \frac{\delta y}{\delta x_2} & \cdots & \frac{\delta y}{\delta x_c} \end{bmatrix}_{1 \times c}$$

Notation for Vector Calculus

Scalars: $y, x \in \mathbb{R}$ Vectors: $\vec{\boldsymbol{y}} \in \mathbb{R}^d, \vec{\boldsymbol{x}} \in \mathbb{R}^c$ Matrices: $\mathbf{X} \in \mathbb{R}^{m \times n}, \boldsymbol{Y} \in \mathbb{R}^{o \times p}$

Our convention: Numerator dim → rows, Denominator dim → cols

$$\frac{\delta \vec{\boldsymbol{y}}}{\delta x} = \frac{\boldsymbol{\delta \text{vector}}}{\boldsymbol{\delta \text{scalar}}} = \begin{bmatrix} \frac{\delta y_1}{\delta x} \\ \frac{\delta y_2}{\delta x} \\ \vdots \\ \frac{\delta y_d}{\delta x} \end{bmatrix}_{d \times 1}$$

$$\vec{\boldsymbol{y}}(x) = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} -5x \\ x^2 \end{bmatrix}_{2 \times 1}$$

Notation for Vector Calculus

Scalars: $y, x \in \mathbb{R}$ Vectors: $\vec{\boldsymbol{y}} \in \mathbb{R}^d, \vec{\boldsymbol{x}} \in \mathbb{R}^c$ Matrices: $\mathbf{X} \in \mathbb{R}^{m \times n}, \boldsymbol{Y} \in \mathbb{R}^{o \times p}$

Our convention: Numerator dim → rows, Denominator dim → cols

$$\frac{\delta \vec{\boldsymbol{y}}}{\delta x} = \begin{bmatrix} \frac{\delta y_1}{\delta x} \\ \frac{\delta y_2}{\delta x} \\ \vdots \\ \frac{\delta y_d}{\delta x} \end{bmatrix}_{d \times 1} \qquad \frac{\boldsymbol{\delta \text{vector}}}{\boldsymbol{\delta \text{scalar}}}$$

$$\vec{\boldsymbol{y}}(x) = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} -5x \\ x^2 \end{bmatrix}_{2 \times 1}$$

$$\frac{\delta \vec{\boldsymbol{y}}}{\delta x} = \begin{bmatrix} -5 \\ 2x \end{bmatrix}_{2 \times 1}$$

Notation for Vector Calculus

Scalars: $y, x \in \mathbb{R}$ Vectors: $\vec{\boldsymbol{y}} \in \mathbb{R}^d, \vec{\boldsymbol{x}} \in \mathbb{R}^c$ Matrices: $\mathbf{X} \in \mathbb{R}^{m \times n}, \boldsymbol{Y} \in \mathbb{R}^{o \times p}$

Our convention: Numerator dim → rows, Denominator dim → cols

$$\frac{\boldsymbol{\delta \text{scalar}}}{\boldsymbol{\delta \text{vector}}}$$

$$\frac{\delta y}{\delta \vec{\boldsymbol{x}}} = \begin{bmatrix} \frac{\delta y}{\delta x_1} & \frac{\delta y}{\delta x_2} & \cdots & \frac{\delta y}{\delta x_c} \end{bmatrix}_{1 \times c}$$

$$y(\vec{\boldsymbol{x}}) = \vec{\boldsymbol{w}}^T \vec{\boldsymbol{x}} = \sum_{\boldsymbol{i}} w_i x_i \qquad \vec{\boldsymbol{w}} \in \mathbb{R}^c$$

Notation for Vector Calculus

Scalars: $y, x \in \mathbb{R}$ Vectors: $\vec{\boldsymbol{y}} \in \mathbb{R}^d, \vec{\boldsymbol{x}} \in \mathbb{R}^c$ Matrices: $\mathbf{X} \in \mathbb{R}^{m \times n}, \boldsymbol{Y} \in \mathbb{R}^{o \times p}$

Our convention: Numerator dim → rows, Denominator dim → cols

$$\frac{\boldsymbol{\delta \text{scalar}}}{\boldsymbol{\delta \text{vector}}}$$

$$\frac{\delta y}{\delta \vec{\boldsymbol{x}}} = \begin{bmatrix} \frac{\delta y}{\delta x_1} & \frac{\delta y}{\delta x_2} & \cdots & \frac{\delta y}{\delta x_c} \end{bmatrix}_{1 \times c}$$

$$y(\vec{\boldsymbol{x}}) = \vec{\boldsymbol{w}}^T \vec{\boldsymbol{x}} = \sum_{\boldsymbol{i}} w_i x_i \qquad \vec{\boldsymbol{w}} \in \mathbb{R}^c$$

$$\frac{\delta y}{\delta \vec{\boldsymbol{x}}} = \begin{bmatrix} w_1 & w_2 & \cdots & w_c \end{bmatrix}_{1 \times c}$$

$$= \vec{\boldsymbol{w}}^T$$

Notation for Vector Calculus

Scalars: $y, x \in \mathbb{R}$ Vectors: $\vec{\boldsymbol{y}} \in \mathbb{R}^d, \vec{\boldsymbol{x}} \in \mathbb{R}^c$ Matrices: $\mathbf{X} \in \mathbb{R}^{m \times n}, \boldsymbol{Y} \in \mathbb{R}^{o \times p}$

Our convention: Numerator dim → rows, Denominator dim → cols

$$\frac{\boldsymbol{\delta \text{vector}}}{\boldsymbol{\delta \text{vector}}}$$

$$\frac{\delta \vec{\boldsymbol{y}}}{\delta \vec{\boldsymbol{x}}} = \begin{bmatrix} \frac{\delta y_1}{\delta x_1} & \frac{\delta y_1}{\delta x_2} & \cdots & \frac{\delta y_1}{\delta x_c} \\ \frac{\delta y_2}{\delta x_1} & \frac{\delta y_2}{\delta x_2} & \cdots & \frac{\delta y_2}{\delta x_c} \\ \vdots & \vdots & \ldots & \vdots \\ \frac{\delta y_d}{\delta x_1} & \frac{\delta y_d}{\delta x_2} & \cdots & \frac{\delta y_d}{\delta x_c} \end{bmatrix}_{d \times c}$$

Called the Jacobian

Notation for Vector Calculus

Scalars: $y, x \in \mathbb{R}$ Vectors: $\vec{\boldsymbol{y}} \in \mathbb{R}^d, \vec{\boldsymbol{x}} \in \mathbb{R}^c$ Matrices: $\mathbf{X} \in \mathbb{R}^{m \times n}, \boldsymbol{Y} \in \mathbb{R}^{o \times p}$

Our convention: Numerator dim → rows, Denominator dim → cols

$$\frac{\delta \vec{\boldsymbol{y}}}{\delta \vec{\boldsymbol{x}}} = \begin{bmatrix} \frac{\delta y_1}{\delta x_1} & \frac{\delta y_1}{\delta x_2} & \cdots & \frac{\delta y_1}{\delta x_c} \\ \frac{\delta y_2}{\delta x_1} & \frac{\delta y_2}{\delta x_2} & \cdots & \frac{\delta y_2}{\delta x_c} \\ \vdots & \vdots & \ldots & \vdots \\ \frac{\delta y_d}{\delta x_1} & \frac{\delta y_d}{\delta x_2} & \cdots & \frac{\delta y_d}{\delta x_c} \end{bmatrix}_{d \times c}$$

Called the Jacobian

$$\vec{\boldsymbol{y}}(\vec{\boldsymbol{x}}) = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} -5x_1 + x_2 \\ x_1^2 \end{bmatrix}$$

Notation for Vector Calculus

Scalars: $y, x \in \mathbb{R}$ Vectors: $\vec{\boldsymbol{y}} \in \mathbb{R}^d, \vec{\boldsymbol{x}} \in \mathbb{R}^c$ Matrices: $\mathbf{X} \in \mathbb{R}^{m \times n}, \boldsymbol{Y} \in \mathbb{R}^{o \times p}$

Our convention: Numerator dim → rows, Denominator dim → cols

$$\frac{\delta \vec{\boldsymbol{y}}}{\delta \vec{\boldsymbol{x}}} = \begin{bmatrix} \frac{\delta y_1}{\delta x_1} & \frac{\delta y_1}{\delta x_2} & \cdots & \frac{\delta y_1}{\delta x_c} \\ \frac{\delta y_2}{\delta x_1} & \frac{\delta y_2}{\delta x_2} & \cdots & \frac{\delta y_2}{\delta x_c} \\ \vdots & \vdots & \ldots & \vdots \\ \frac{\delta y_d}{\delta x_1} & \frac{\delta y_d}{\delta x_2} & \cdots & \frac{\delta y_d}{\delta x_c} \end{bmatrix}_{d \times c}$$

Called the Jacobian

$$\vec{\boldsymbol{y}}(\vec{\boldsymbol{x}}) = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} -5x_1 + x_2 \\ x_1^2 \end{bmatrix}$$

$$\frac{\delta \vec{\boldsymbol{y}}}{\delta \vec{\boldsymbol{x}}} = \begin{bmatrix} \frac{\delta y_1}{\delta x_1} & \frac{\delta y_1}{\delta x_2} \\ \frac{\delta y_2}{\delta x_1} & \frac{\delta y_2}{\delta x_2} \end{bmatrix} = \begin{bmatrix} -5 & 1 \\ 2x_1 & 0 \end{bmatrix}$$

Notation for Vector Calculus

Scalars: $y, x \in \mathbb{R}$ Vectors: $\vec{\boldsymbol{y}} \in \mathbb{R}^d, \vec{\boldsymbol{x}} \in \mathbb{R}^c$ Matrices: $\mathbf{X} \in \mathbb{R}^{m \times n}, \boldsymbol{Y} \in \mathbb{R}^{o \times p}$

Our convention: Numerator dim → rows, Denominator dim → cols

$$\frac{\boldsymbol{\delta \text{vector}}}{\boldsymbol{\delta \text{matrix}}}$$

$$\frac{\delta \vec{\boldsymbol{y}}}{\delta \boldsymbol{X}} = \begin{bmatrix} \frac{\delta y_1}{\delta x_{11}} & \frac{\delta y_1}{\delta x_{12}} & \cdots & \frac{\delta y_1}{\delta x_{mn}} \\ \frac{\delta y_2}{\delta x_{11}} & \frac{\delta y_2}{\delta x_{12}} & \cdots & \frac{\delta y_2}{\delta x_{mn}} \\ \vdots & \vdots & \ldots & \vdots \\ \frac{\delta y_d}{\delta x_{11}} & \frac{\delta y_d}{\delta x_{12}} & \cdots & \frac{\delta y_d}{\delta x_{mn}} \end{bmatrix}_{d \times mn}$$

Vectorize the matrix and treat like vector-vector derivative

Notation for Vector Calculus

Scalars: $y, x \in \mathbb{R}$ Vectors: $\vec{\boldsymbol{y}} \in \mathbb{R}^d, \vec{\boldsymbol{x}} \in \mathbb{R}^c$ Matrices: $\mathbf{X} \in \mathbb{R}^{m \times n}, \boldsymbol{Y} \in \mathbb{R}^{o \times p}$

Our convention: Numerator dim → rows, Denominator dim → cols

Chain Rule

$$L(\vec{\boldsymbol{x}}) = L\left(\vec{\boldsymbol{y}}(\vec{\boldsymbol{x}})\right)$$

$$\underset{1\times c}{\frac{\delta L}{\delta \vec{\boldsymbol{x}}}} = \underset{1\times d}{\frac{\delta L}{\delta \vec{\boldsymbol{y}}}} * \underset{\mathrm{d}\times c}{\frac{\delta \vec{\boldsymbol{y}}}{\delta \vec{\boldsymbol{x}}}}$$

Notation for Vector Calculus

Scalars: $y, x \in \mathbb{R}$ Vectors: $\vec{\boldsymbol{y}} \in \mathbb{R}^d, \vec{\boldsymbol{x}} \in \mathbb{R}^c$ Matrices: $\mathbf{X} \in \mathbb{R}^{m \times n}, \boldsymbol{Y} \in \mathbb{R}^{o \times p}$

Our convention: Numerator dim → rows, Denominator dim → cols

Chain Rule

$$\underbrace{\begin{bmatrix} \frac{\delta L}{\delta x_1} & \frac{\delta L}{\delta x_2} & \cdots & \frac{\delta L}{\delta x_c} \end{bmatrix}}_{1 \times c} = \underbrace{\begin{bmatrix} \frac{\delta L}{\delta y_1} & \frac{\delta L}{\delta y_2} & \cdots & \frac{\delta L}{\delta y_d} \end{bmatrix}}_{1 \times d} \underbrace{\begin{bmatrix} \frac{\delta y_1}{\delta x_1} & \frac{\delta y_1}{\delta x_2} & \cdots & \frac{\delta y_1}{\delta x_c} \\ \frac{\delta y_2}{\delta x_1} & \frac{\delta y_2}{\delta x_2} & \cdots & \frac{\delta y_2}{\delta x_c} \\ \vdots & \vdots & \cdots & \vdots \\ \frac{\delta y_d}{\delta x_1} & \frac{\delta y_d}{\delta x_2} & \cdots & \frac{\delta y_d}{\delta x_c} \end{bmatrix}}_{d \times c}$$

Notation for Vector Calculus

Scalars: $y, x \in \mathbb{R}$ Vectors: $\vec{\boldsymbol{y}} \in \mathbb{R}^d, \vec{\boldsymbol{x}} \in \mathbb{R}^c$ Matrices: $\mathbf{X} \in \mathbb{R}^{m \times n}, \boldsymbol{Y} \in \mathbb{R}^{o \times p}$

Our convention: Numerator dim → rows, Denominator dim → cols

Chain Rule

$$\begin{bmatrix} \frac{\delta L}{\delta x_1} & \frac{\delta L}{\delta x_2} & \cdots & \frac{\delta L}{\delta x_c} \end{bmatrix} = \begin{bmatrix} \frac{\delta L}{\delta y_1} & \frac{\delta L}{\delta y_2} & \cdots & \frac{\delta L}{\delta y_d} \end{bmatrix} \begin{bmatrix} \frac{\delta y_1}{\delta x_1} & \frac{\delta y_1}{\delta x_2} & \cdots & \frac{\delta y_1}{\delta x_c} \\ \frac{\delta y_2}{\delta x_1} & \frac{\delta y_2}{\delta x_2} & \cdots & \frac{\delta y_2}{\delta x_c} \\ \vdots & \vdots & \cdots & \vdots \\ \frac{\delta y_d}{\delta x_1} & \frac{\delta y_d}{\delta x_2} & \cdots & \frac{\delta y_d}{\delta x_c} \end{bmatrix}$$

$$\frac{\delta L}{\delta x_2} = \sum_{i=1}^{d} \frac{\delta L}{\delta y_i} \frac{\delta y_i}{\delta x_2}$$

Today's Learning Objectives

Be able to answer:

- ~~What is a fully-connected layer?~~
- ~~What is a loss function?~~

- ~~(Quick review of vector calculus)~~

- What is forward / reverse mode differentiation?
- How does backpropagation work?

- What is a computation graph?

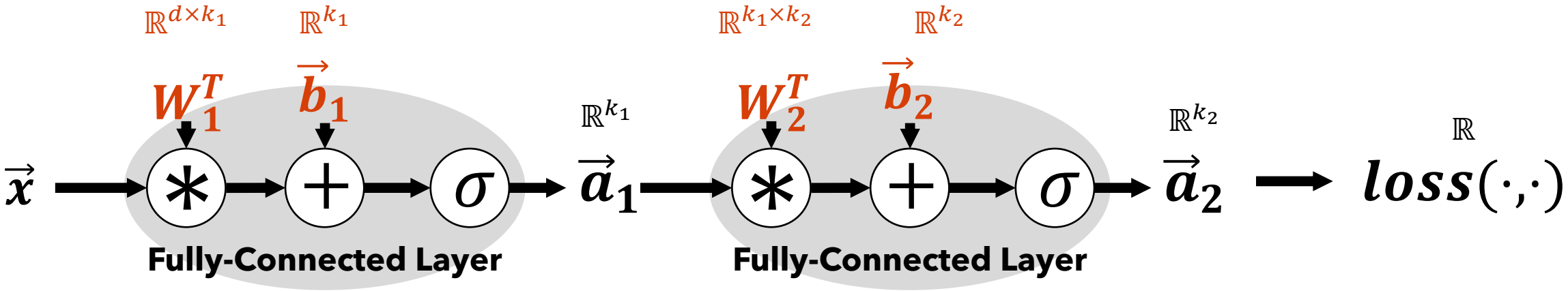
Backpropagation – A reverse-mode automatic differentiation algorithm commonly used to efficiently compute parameter gradients when training neural networks via gradient descent.

Builds off two simple observations / ideas:

1) Neural networks tend to have lower dimensional outputs than inputs.

2) We shouldn't recompute something we already know.

Gradients of Our Parameters



$$\frac{\delta loss}{\delta \boldsymbol{W^{(1)}}} = \frac{\delta loss}{\delta\, \boldsymbol{a^{(2)}}} * \frac{\delta \boldsymbol{a^{(2)}}}{\delta \boldsymbol{a^{(1)}}} * \frac{\delta \boldsymbol{a^{(1)}}}{\delta \boldsymbol{W^{(1)}}}$$

Dim: $1\times(dk_1)$ $1\times k_2$ $k_2\times k_1$ $k_1\times(dk_1)$

Questions Break!

Should I multiply these matrices left-to-right or right-to-left?

Hint: The computational complexity of multiplying an $a \times b$ and $b \times c$ matrix is $O(abc)$

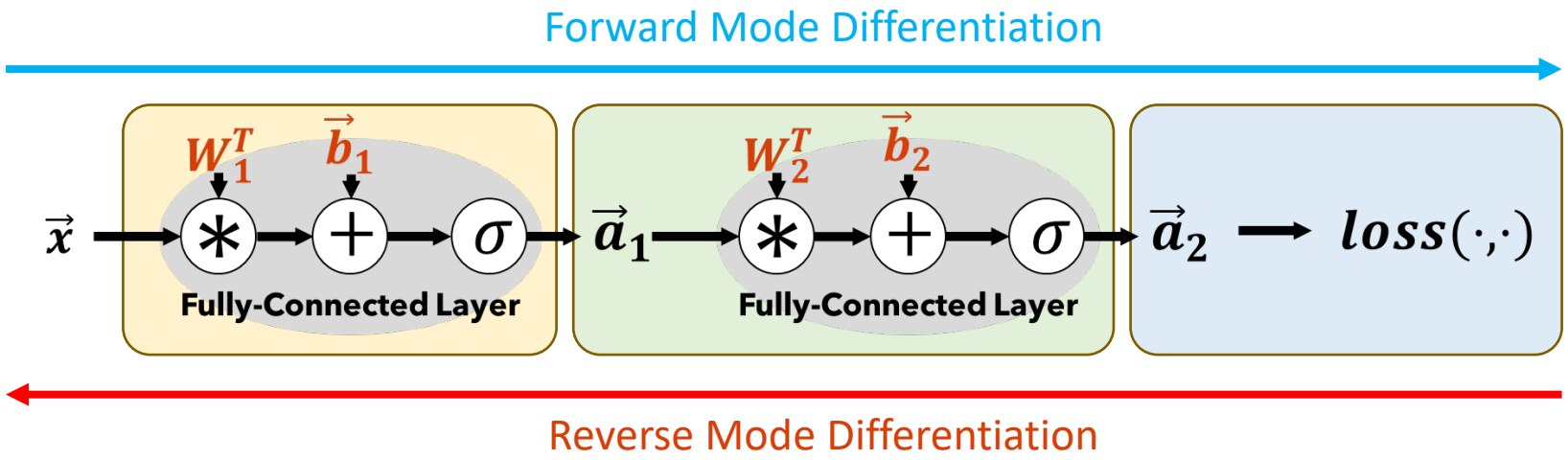
$$\frac{\delta loss}{\delta \boldsymbol{W}^{(1)}} = \frac{\delta loss}{\delta\, \boldsymbol{a}^{(2)}} * \frac{\delta \boldsymbol{a}^{(2)}}{\delta \boldsymbol{a}^{(1)}} * \frac{\delta \boldsymbol{a}^{(1)}}{\delta \boldsymbol{W}^{(1)}}$$

$O(k_2 d k_1)$ $O(k_2 k_1 d k_1)$

$O(k_2 k_1)$ $O(k_1 d k_1)$

Dim: $1 \times k_2$ $k_2 \times k_1$ $k_1 \times (d k_1)$

Seed 1) Forward vs. Reverse Mode Differentiation

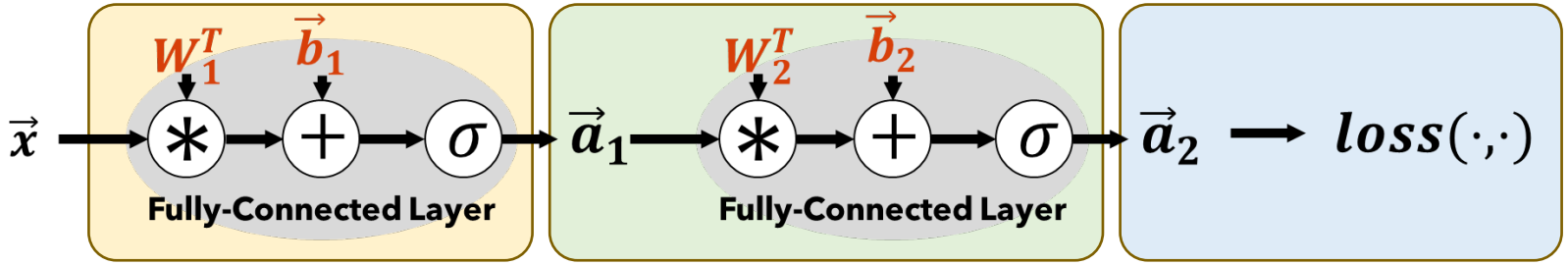


Forward Mode Differentiation
(input to output, dim input << dim output)

$$\frac{\delta loss}{\delta \boldsymbol{W}_1} = \frac{\delta loss}{\delta\, \vec{\boldsymbol{a}}_2} * \frac{\delta \vec{\boldsymbol{a}}_2}{\delta \vec{\boldsymbol{a}}_1} * \frac{\delta \vec{\boldsymbol{a}}_1}{\delta\, \boldsymbol{W}_1}$$

Reverse Mode Differentiation
(output to input, dim output << dim input)

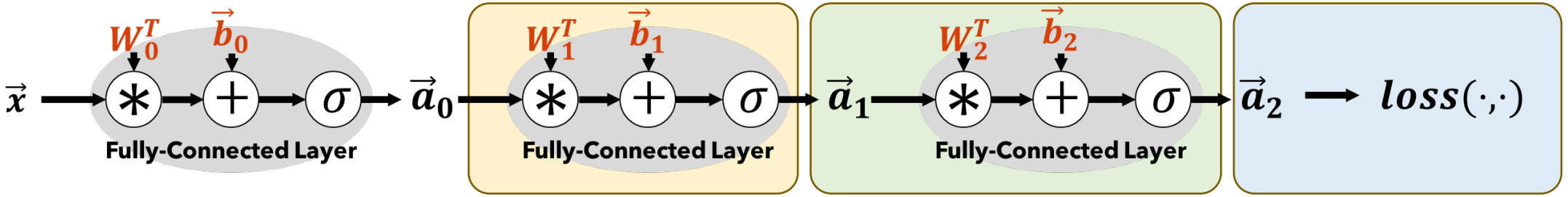
Gradients of Our Parameters / The Seeds of Backprop



$$\frac{\delta L}{\delta \boldsymbol{W}_2} = \frac{\delta L}{\delta \vec{\boldsymbol{a}}_2} * \frac{\delta \vec{\boldsymbol{a}}_2}{\delta \boldsymbol{W}_2} \qquad \frac{\delta L}{\delta \vec{\boldsymbol{b}}_2} = \frac{\delta L}{\delta \vec{\boldsymbol{a}}_2} * \frac{\delta \vec{\boldsymbol{a}}_2}{\delta \vec{\boldsymbol{b}}_2}$$

$$\frac{\delta L}{\delta \boldsymbol{W}_1} = \frac{\delta L}{\delta \vec{\boldsymbol{a}}_2} * \frac{\delta \vec{\boldsymbol{a}}_2}{\delta \vec{\boldsymbol{a}}_1} * \frac{\delta \vec{\boldsymbol{a}}_1}{\delta \boldsymbol{W}_1} \qquad \frac{\delta L}{\delta \vec{\boldsymbol{b}}_1} = \frac{\delta L}{\delta \vec{\boldsymbol{a}}_2} * \frac{\delta \vec{\boldsymbol{a}}_2}{\delta \vec{\boldsymbol{a}}_1} * \frac{\delta \vec{\boldsymbol{a}}_1}{\delta \vec{\boldsymbol{b}}_1}$$

Gradients of Our Parameters / The Seeds of Backprop

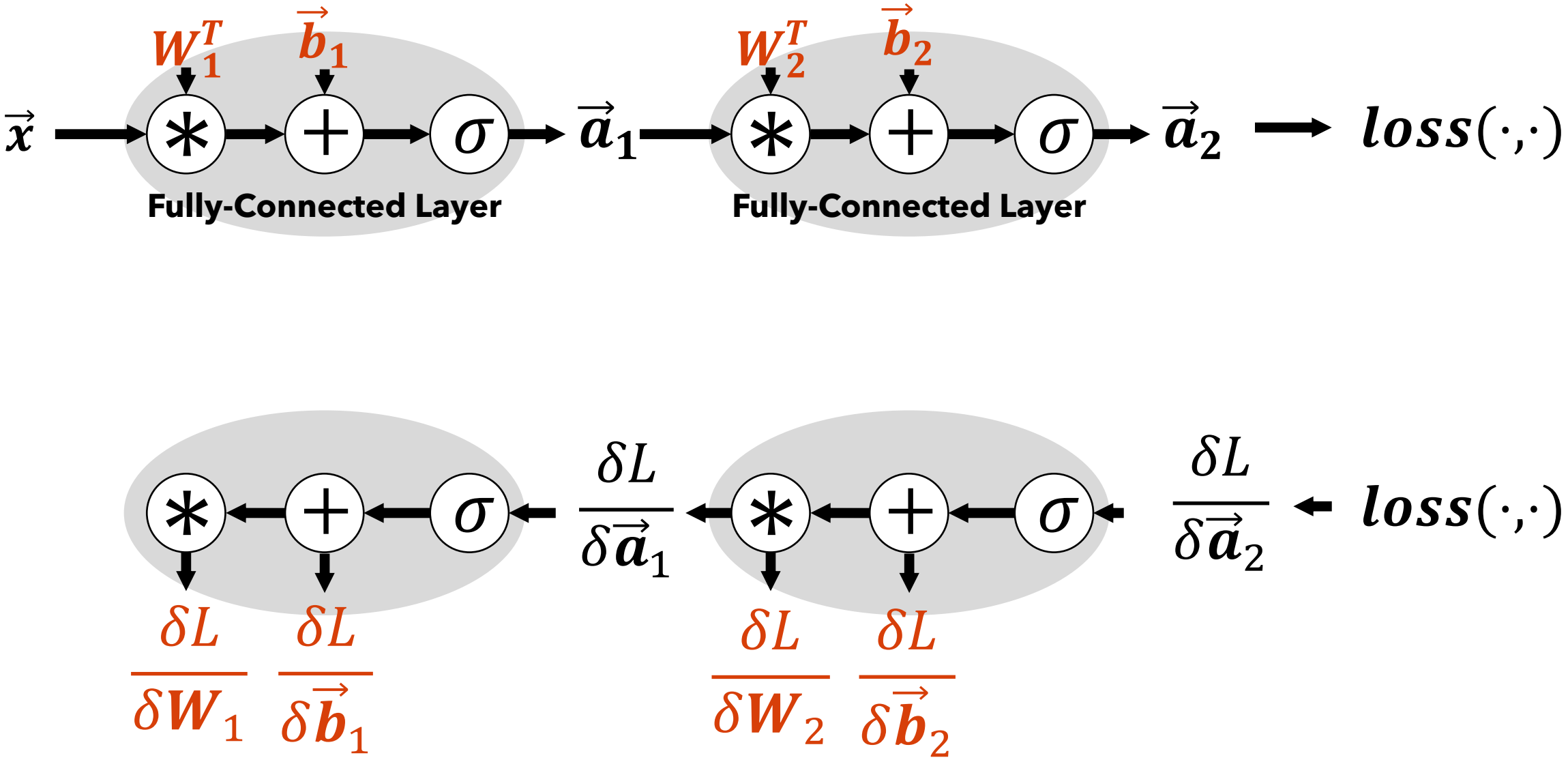


$$\frac{\delta L}{\delta \boldsymbol{W}_2} = \frac{\delta L}{\delta \vec{\boldsymbol{a}}_2} * \frac{\delta \vec{\boldsymbol{a}}_2}{\delta \boldsymbol{W}_2} \qquad \frac{\delta L}{\delta \vec{\boldsymbol{b}}_2} = \frac{\delta L}{\delta \vec{\boldsymbol{a}}_2} * \frac{\delta \vec{\boldsymbol{a}}_2}{\delta \vec{\boldsymbol{b}}_2}$$

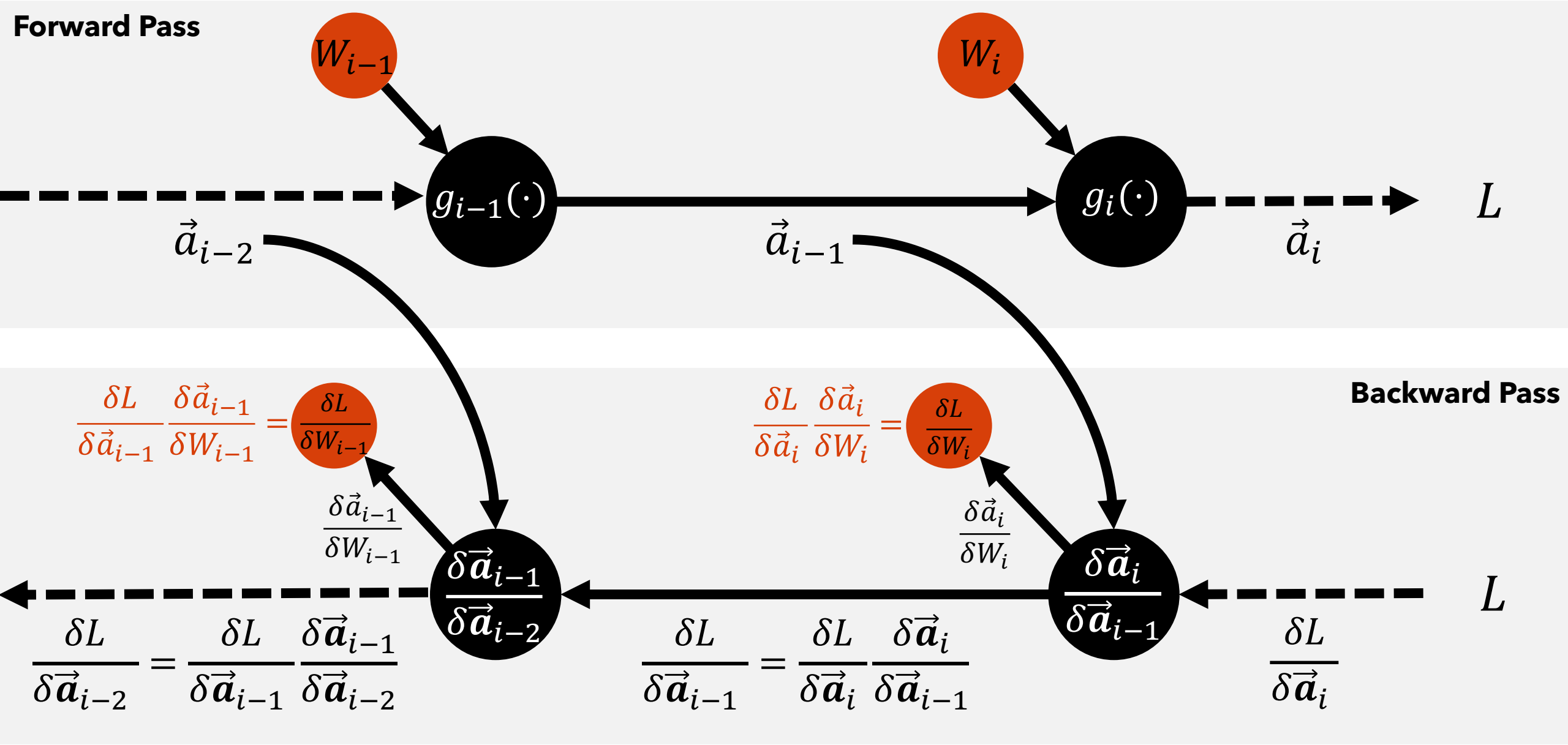
$$\frac{\delta L}{\delta \boldsymbol{W}_1} = \frac{\delta L}{\delta \vec{\boldsymbol{a}}_2} * \frac{\delta \vec{\boldsymbol{a}}_2}{\delta \vec{\boldsymbol{a}}_1} * \frac{\delta \vec{\boldsymbol{a}}_1}{\delta \boldsymbol{W}_1} \qquad \frac{\delta L}{\delta \vec{\boldsymbol{b}}_1} = \frac{\delta L}{\delta \vec{\boldsymbol{a}}_2} * \frac{\delta \vec{\boldsymbol{a}}_2}{\delta \vec{\boldsymbol{a}}_1} * \frac{\delta \vec{\boldsymbol{a}}_1}{\delta \vec{\boldsymbol{b}}_1}$$

$$\frac{\delta L}{\delta \boldsymbol{W}_0} = \frac{\delta L}{\delta \vec{\boldsymbol{a}}_2} * \frac{\delta \vec{\boldsymbol{a}}_2}{\delta \vec{\boldsymbol{a}}_1} * \frac{\delta \vec{\boldsymbol{a}}_1}{\delta \vec{\boldsymbol{a}}_0} * \frac{\delta \vec{\boldsymbol{a}}_1}{\delta \boldsymbol{W}_0} \qquad \frac{\delta L}{\delta \vec{\boldsymbol{b}}_0} = \frac{\delta L}{\delta \vec{\boldsymbol{a}}_2} * \frac{\delta \vec{\boldsymbol{a}}_2}{\delta \vec{\boldsymbol{a}}_1} * \frac{\delta \vec{\boldsymbol{a}}_1}{\delta \vec{\boldsymbol{a}}_0} * \frac{\delta \vec{\boldsymbol{a}}_0}{\delta \vec{\boldsymbol{b}}_0}$$

Backpropagation Algorithm

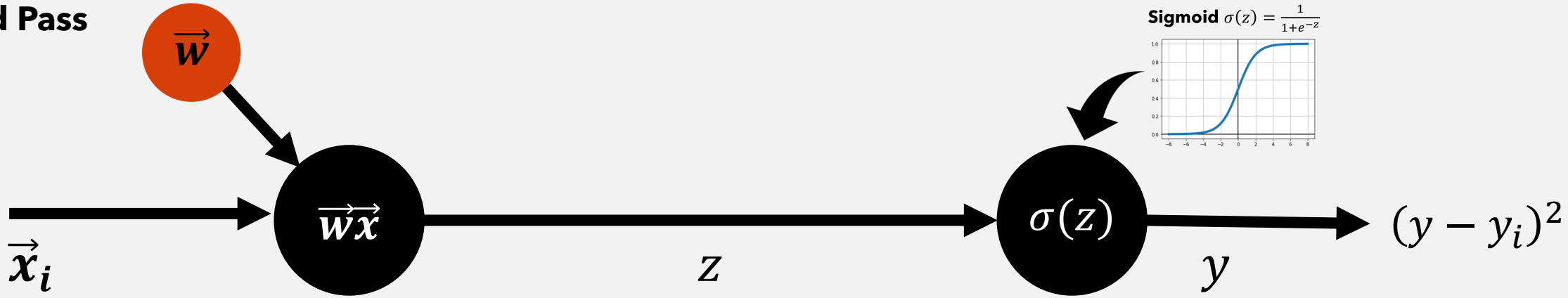


Backpropagation (aka Reverse Mode Auto-differentiation)



Backpropagation (aka Reverse Mode Auto-differentiation)

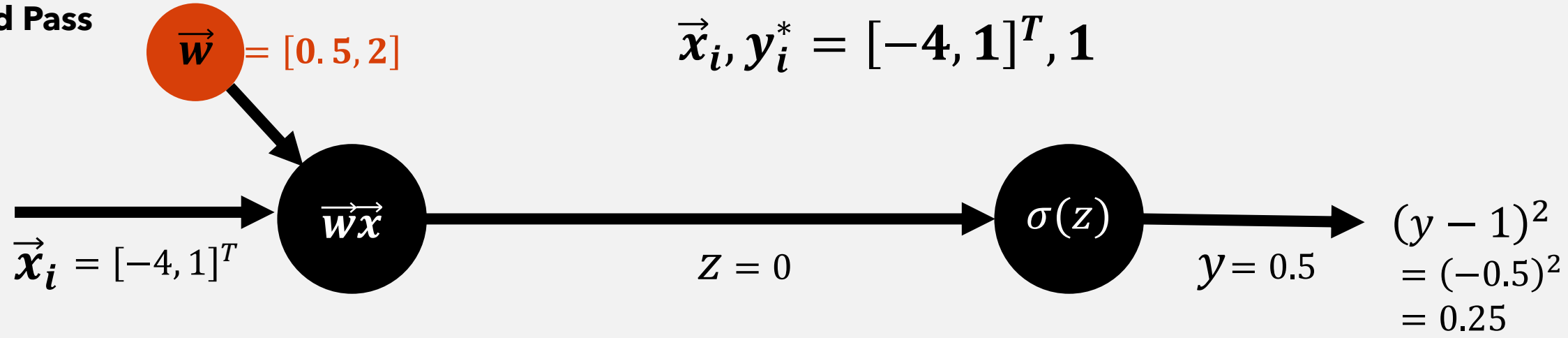
Forward Pass



Backward Pass

Backpropagation (aka Reverse Mode Auto-differentiation)

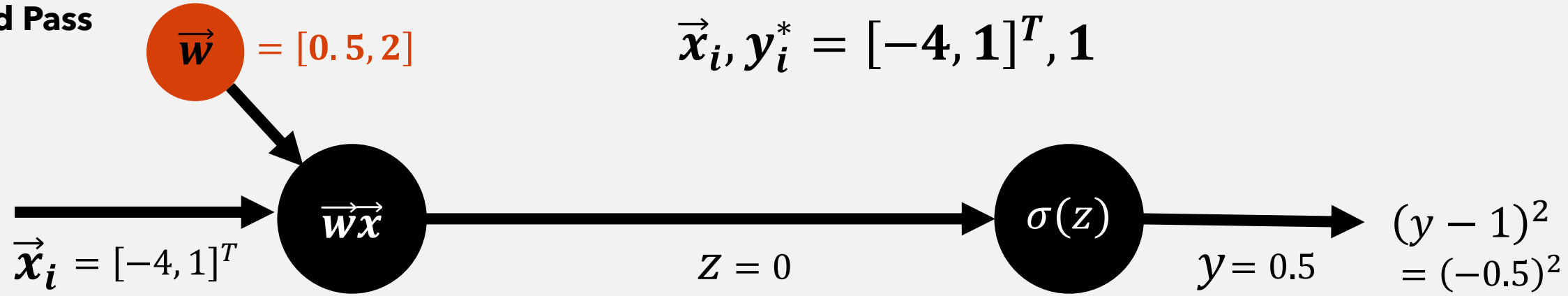
Forward Pass



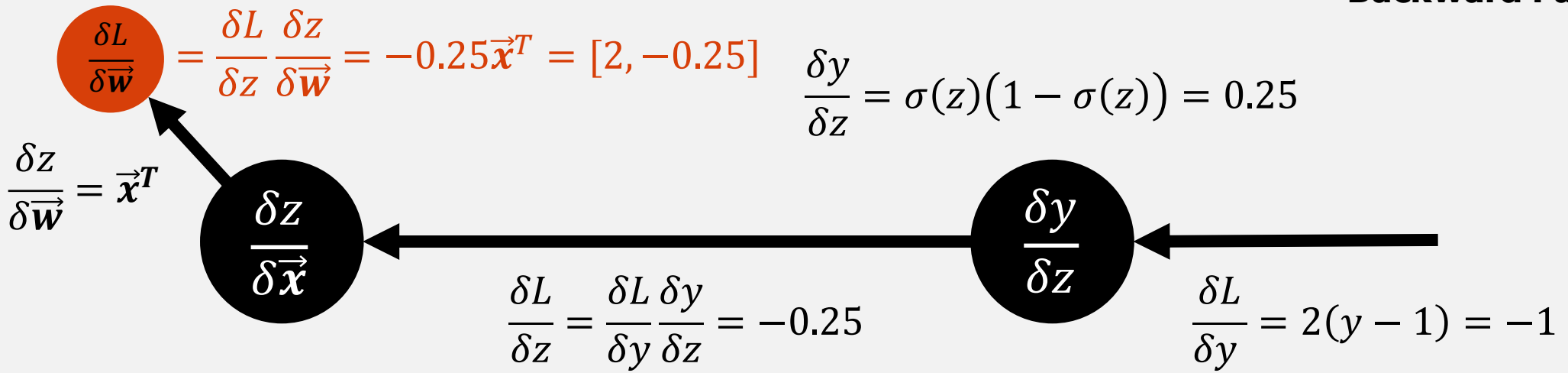
Backward Pass

Backpropagation (aka Reverse Mode Auto-differentiation)

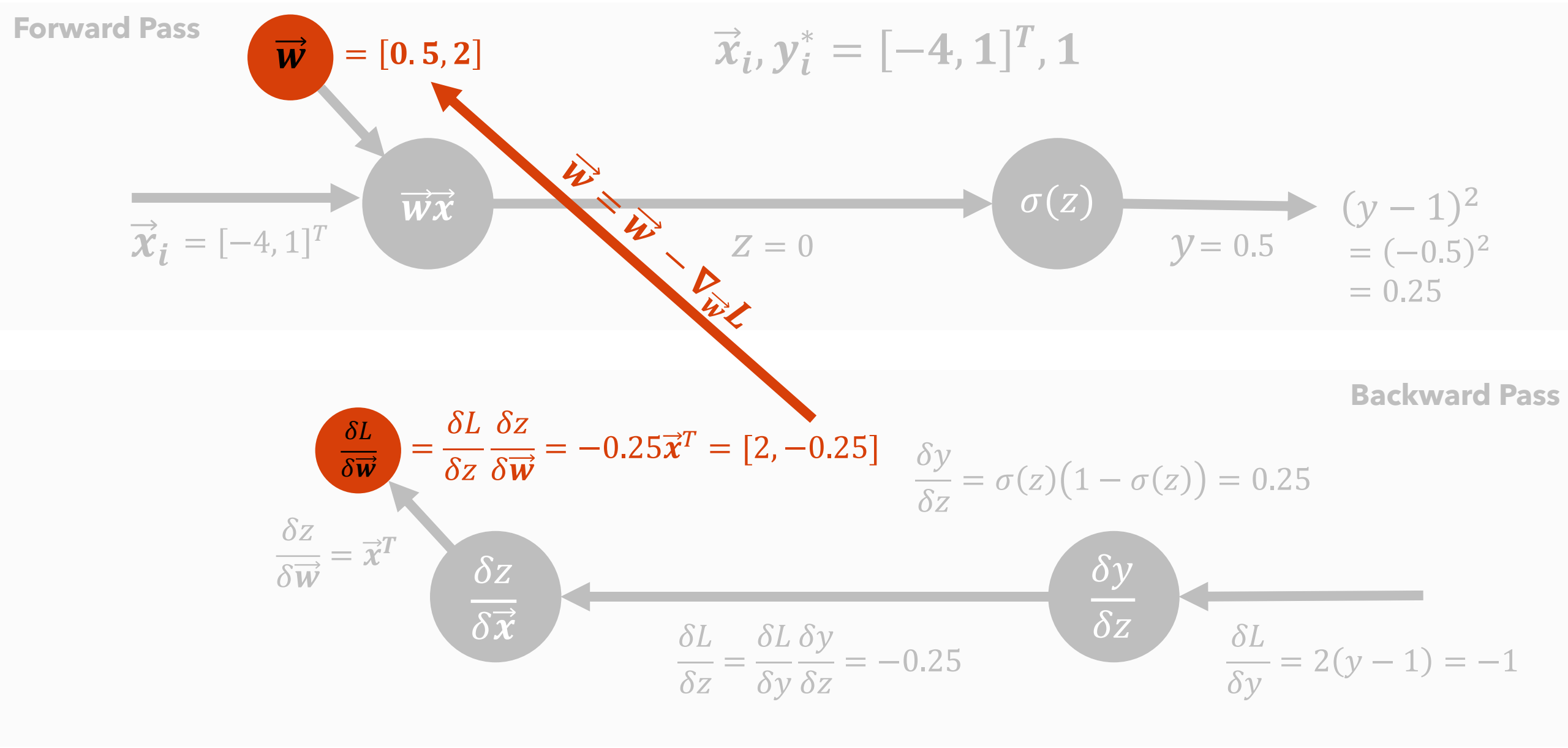
Forward Pass



Backward Pass

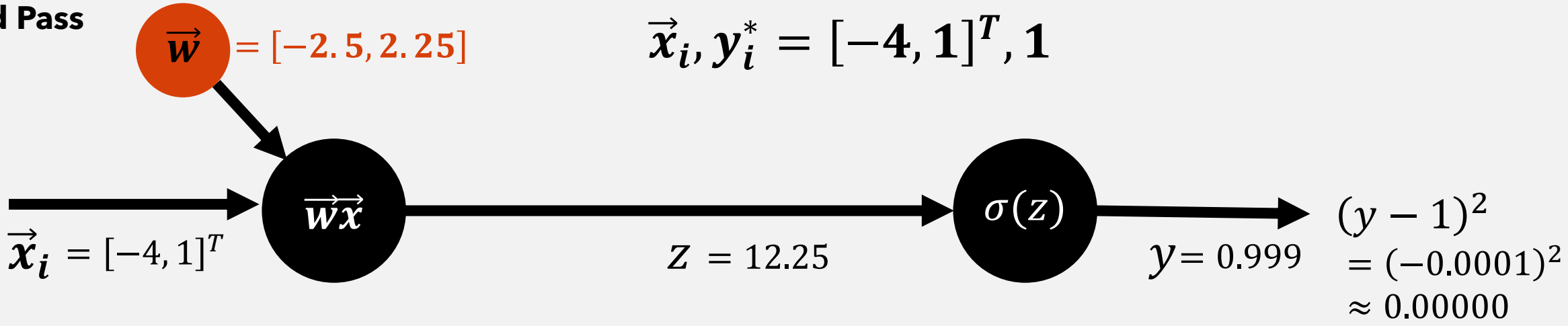


Backpropagation (aka Reverse Mode Auto-differentiation)



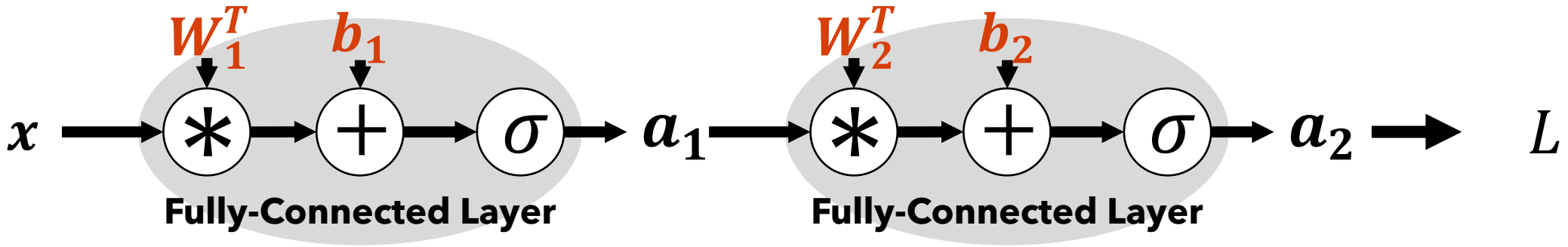
Backpropagation (aka Reverse Mode Auto-differentiation)

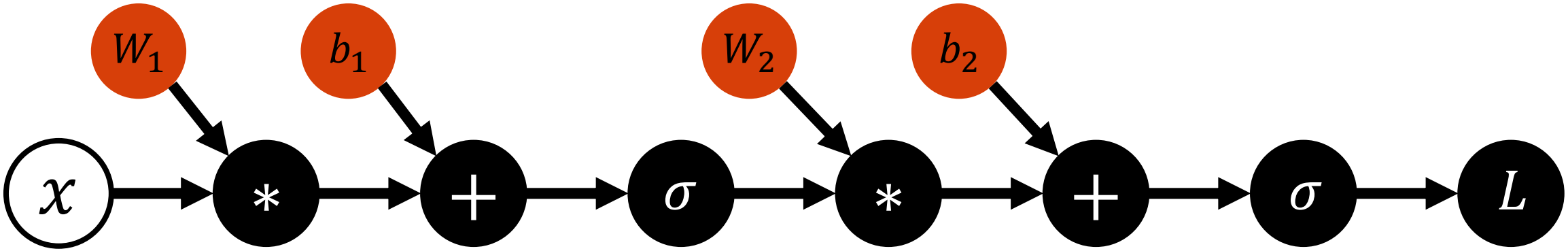
Forward Pass



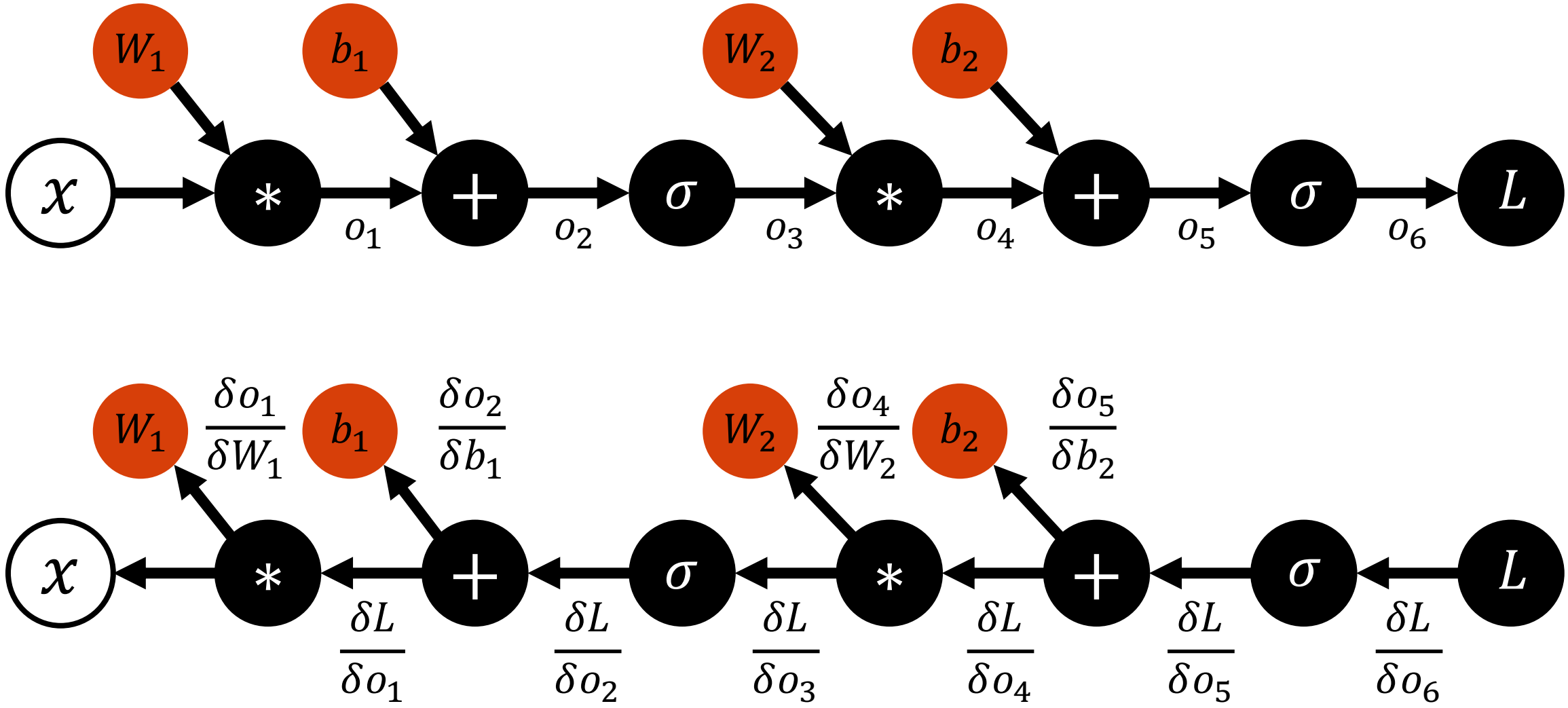
Backward Pass

Neural Networks as Computational Graphs





Neural Networks as Computational Graphs



Today's Learning Objectives

Be able to answer:

- ~~What is a fully-connected layer?~~
- ~~What is a loss function?~~

- ~~(Quick review of vector calculus)~~

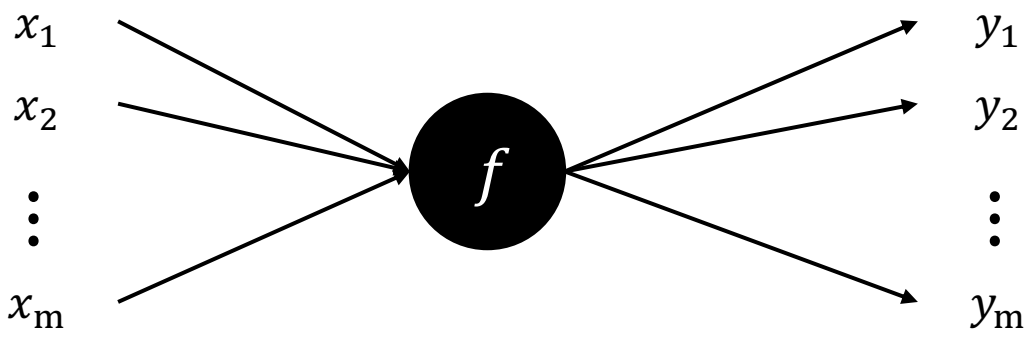
- ~~What is forward / reverse mode differentiation?~~
- ~~How does backpropagation work?~~

- What is a computation graph?

Computational Graph – A directed acyclic graph (DAG) with vertices corresponding to computation and edges to intermediate results of the computation.

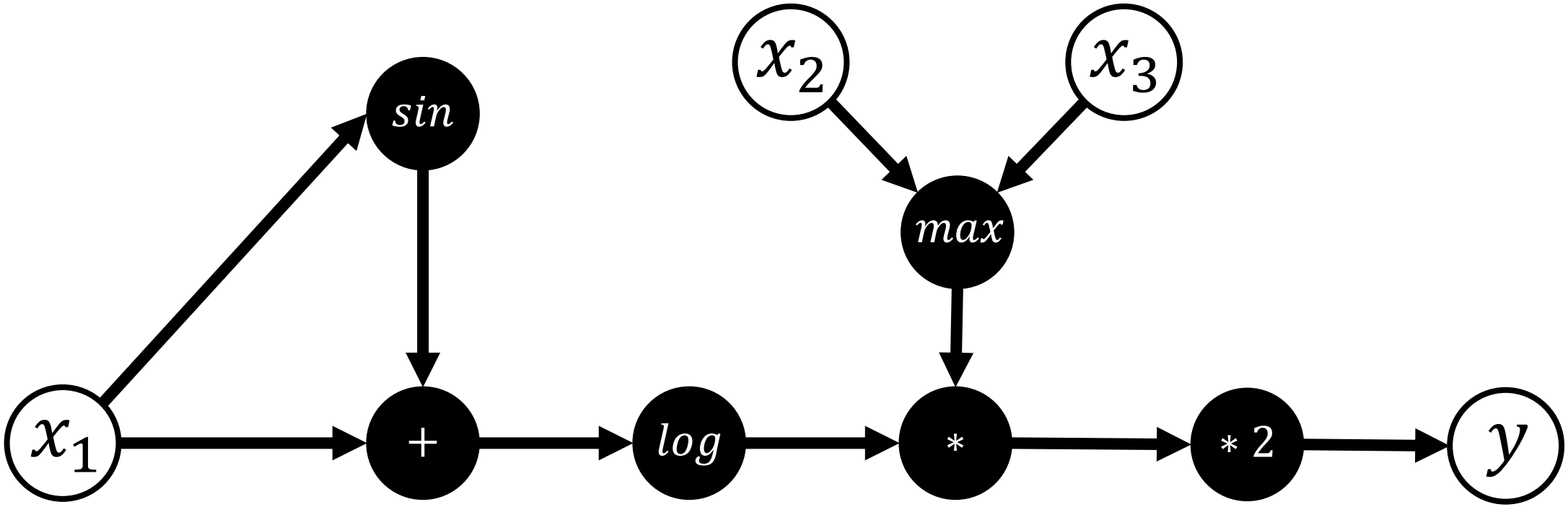
For backprop to work, each node needs to define:

- Its **forward** computation $y_1, \ldots, y_k = f(x_1, \ldots, x_m)$
- Its **backward** computation: $\frac{\delta y_i}{\delta x_j} \; \forall i, j$



Backpropagation on Computational Graphs

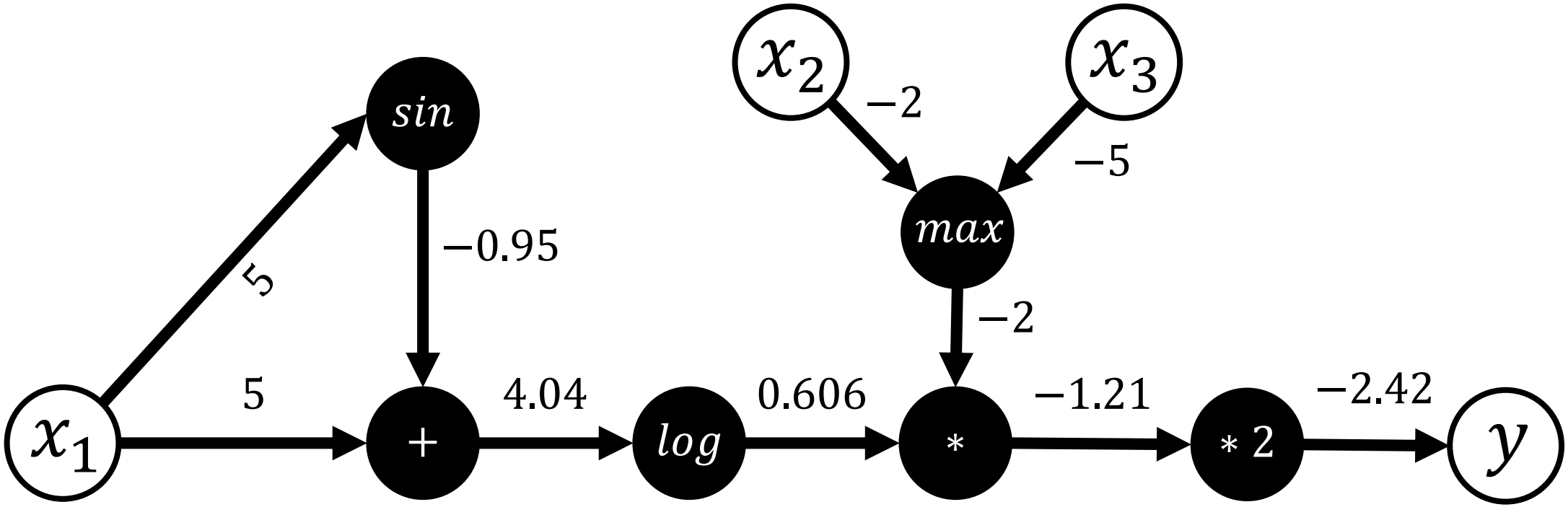
$$y = 2\log(x_1 + \sin(x_1))\max(x_2, x_3)$$



Adapted from Fei-Fei Li, Justin Johnson, Serena Yeung CS 231n

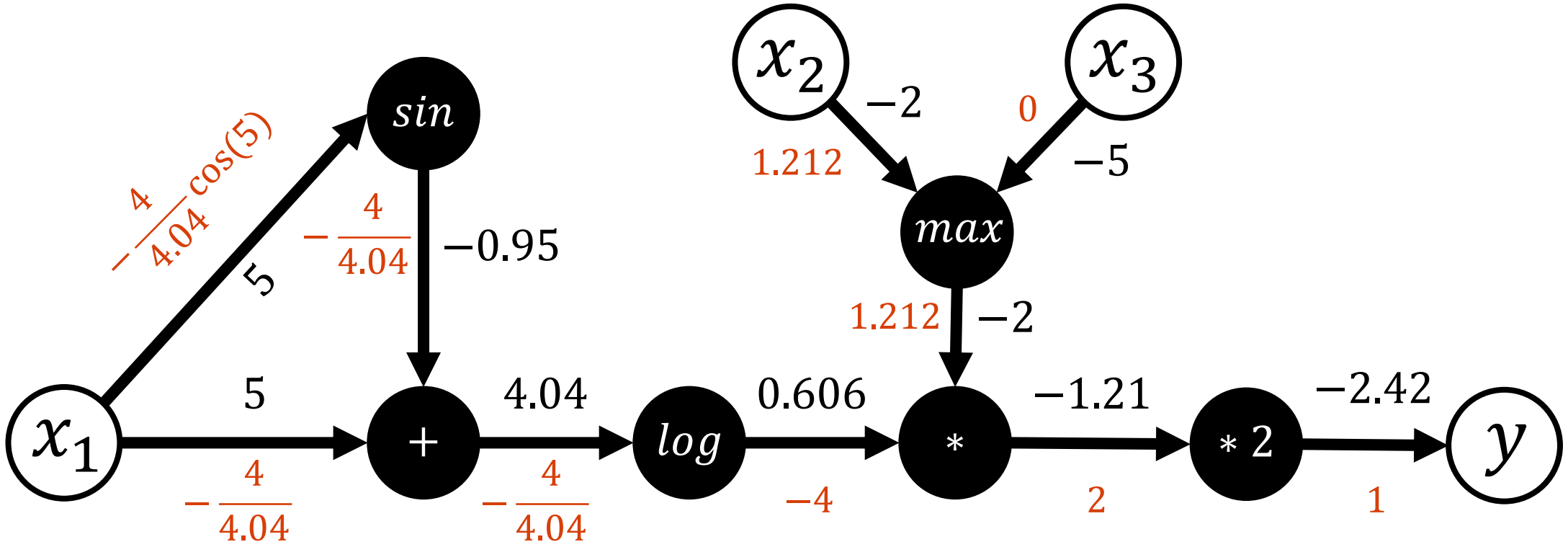
Backpropagation on Computational Graphs

$$y = 2\log(x_1 + \sin(x_1))\max(x_2, x_3)$$



Backpropagation on Computational Graphs

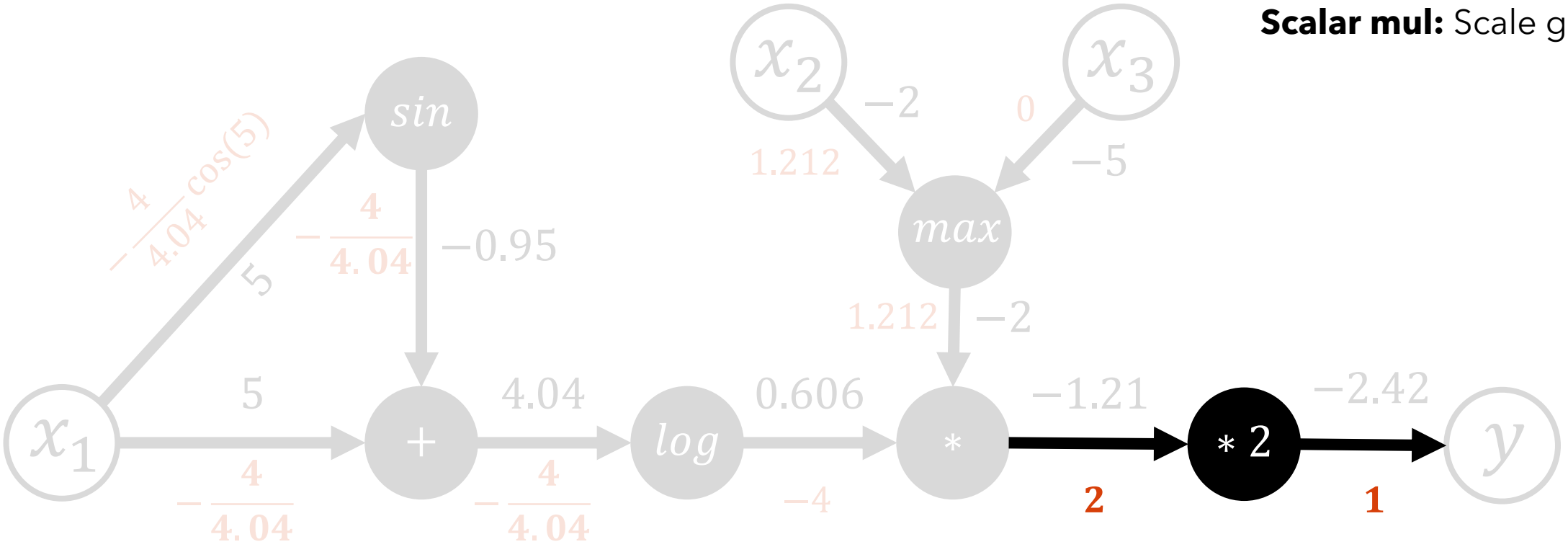
$$y = 2\log(x_1 + \sin(x_1))\max(x_2, x_3)$$



$$\frac{\delta y}{\delta x_1} = -\frac{4}{4.04}\cos(5) - \frac{4}{4.04} \qquad \frac{\delta y}{\delta x_2} = 1.212 \qquad \frac{\delta y}{\delta x_3} = 0$$

Backpropagation on Computational Graphs

$$y = 2\log(x_1 + \sin(x_1))\max(x_2, x_3)$$

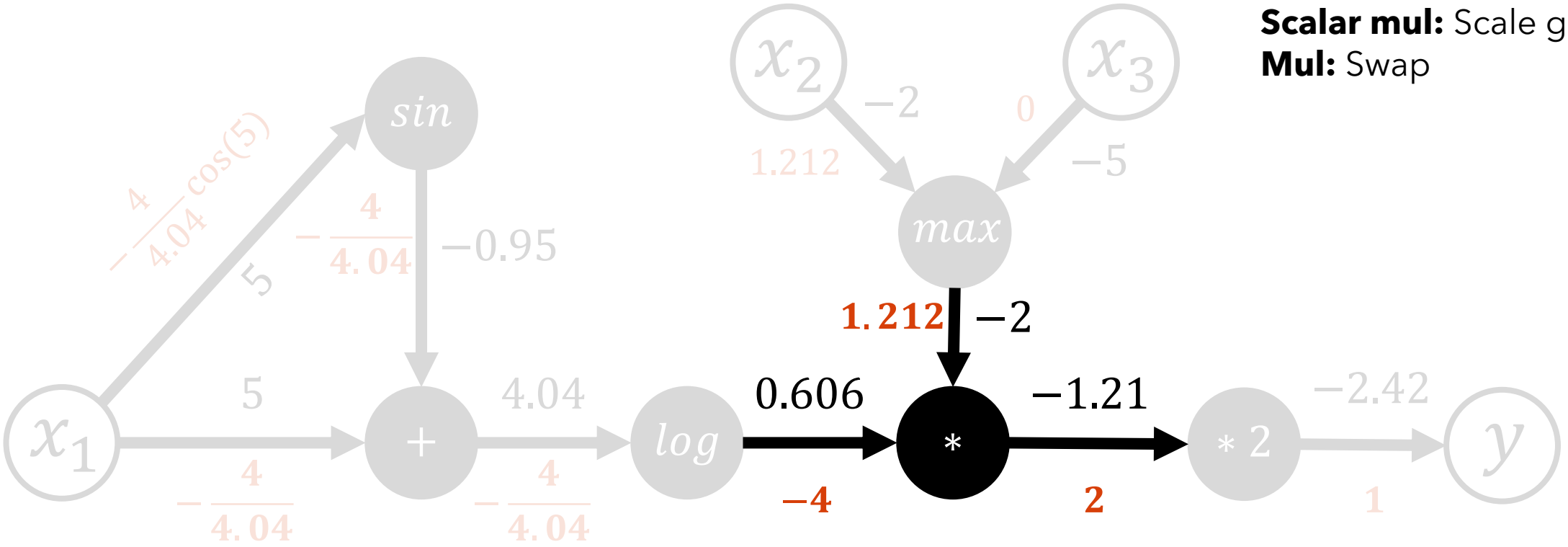


Scalar mul: Scale grad

$$\frac{\delta y}{\delta x_1} = -\frac{4}{4.04}\cos(5) - \frac{4}{4.04} \qquad \frac{\delta y}{\delta x_2} = 1.212 \qquad \frac{\delta y}{\delta x_3} = 0$$

Backpropagation on Computational Graphs

$$y = 2\log(x_1 + \sin(x_1))\max(x_2, x_3)$$

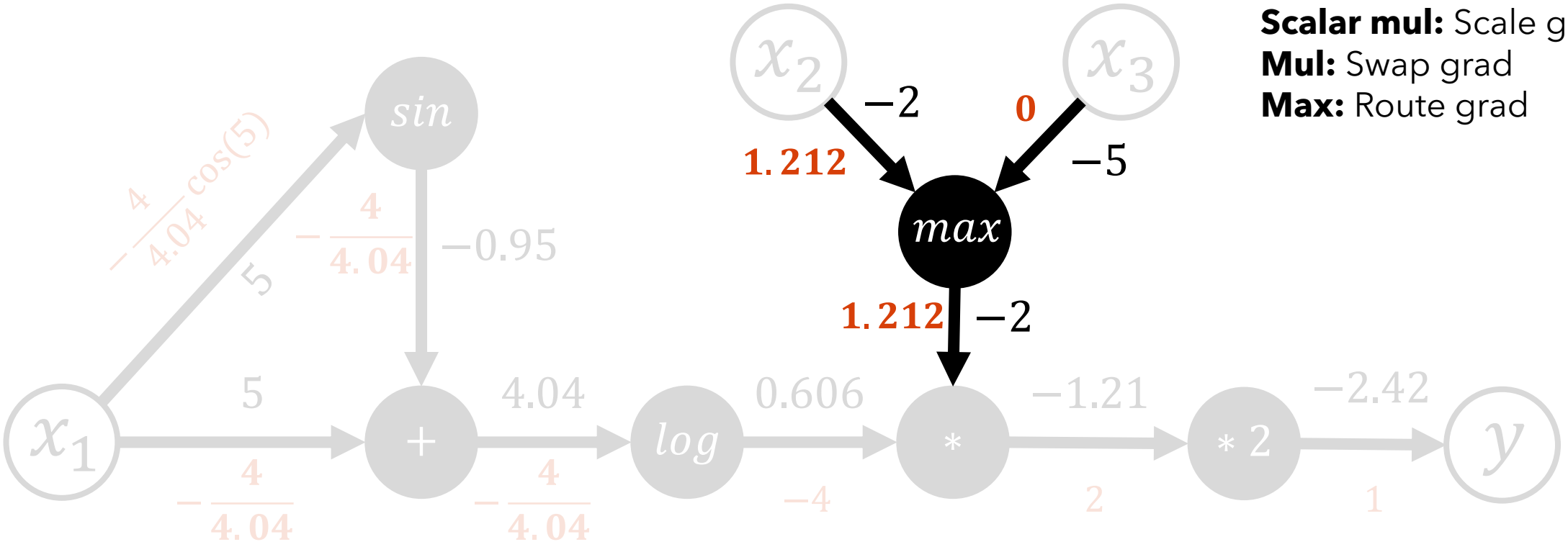


Scalar mul: Scale grad
Mul: Swap

$$\frac{\delta y}{\delta x_1} = -\frac{4}{4.04}\cos(5) - \frac{4}{4.04} \qquad \frac{\delta y}{\delta x_2} = 1.212 \qquad \frac{\delta y}{\delta x_3} = 0$$

Backpropagation on Computational Graphs

$$y = 2\log(x_1 + \sin(x_1))\max(x_2, x_3)$$

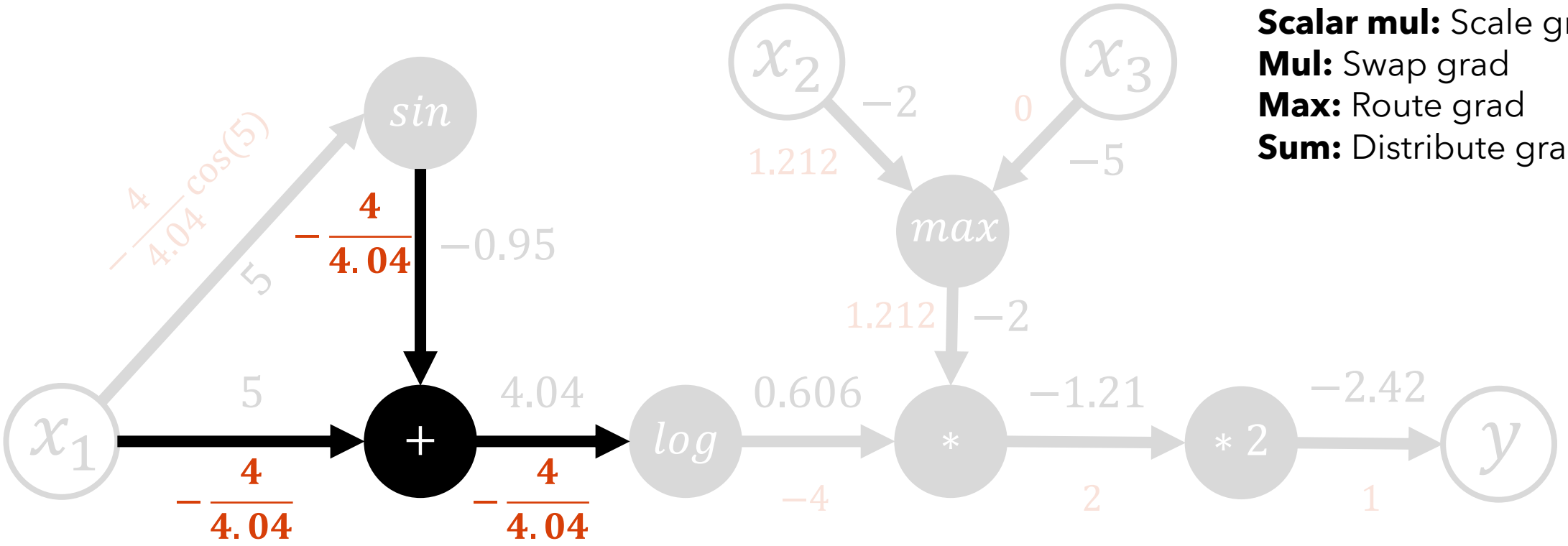


Scalar mul: Scale grad
Mul: Swap grad
Max: Route grad

$$\frac{\delta y}{\delta x_1} = -\frac{4}{4.04}\cos(5) - \frac{4}{4.04} \qquad \frac{\delta y}{\delta x_2} = 1.212 \qquad \frac{\delta y}{\delta x_3} = 0$$

Backpropagation on Computational Graphs

$$y = 2\log(x_1 + \sin(x_1))\max(x_2, x_3)$$

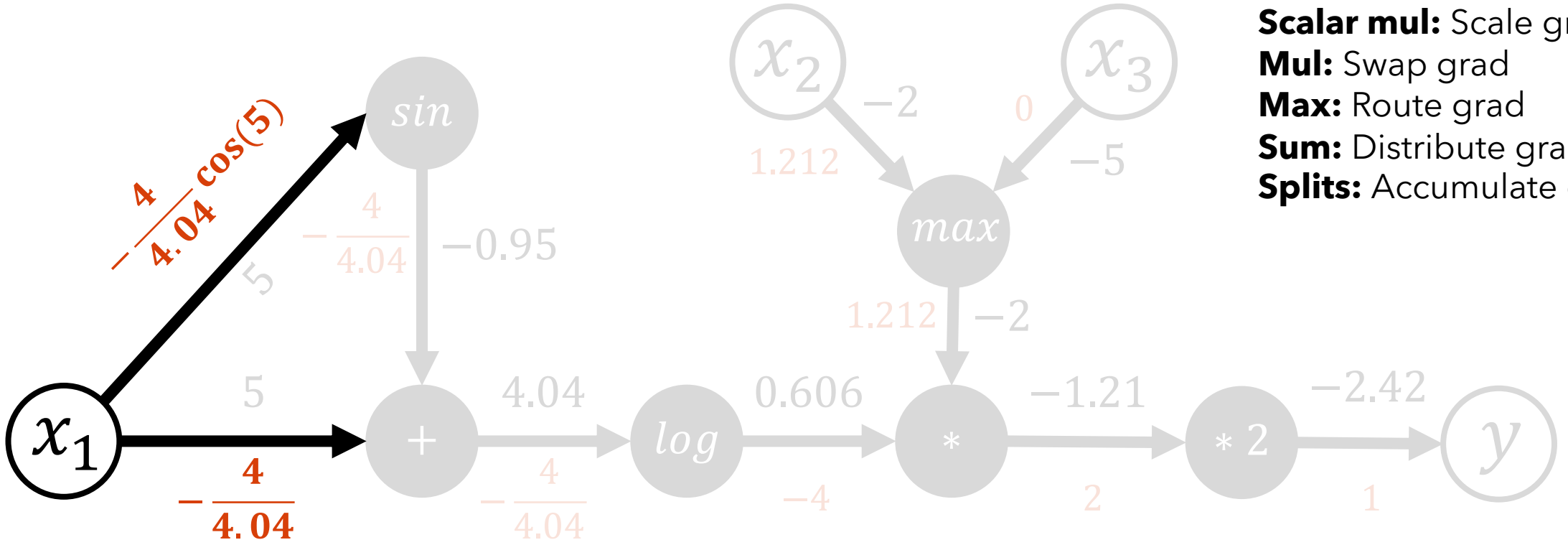


Scalar mul: Scale grad
Mul: Swap grad
Max: Route grad
Sum: Distribute grad

$$\frac{\delta y}{\delta x_1} = -\frac{4}{4.04}\cos(5) - \frac{4}{4.04} \qquad \frac{\delta y}{\delta x_2} = 1.212 \qquad \frac{\delta y}{\delta x_3} = 0$$

Backpropagation on Computational Graphs

$$y = 2\log(x_1 + \sin(x_1))\max(x_2, x_3)$$



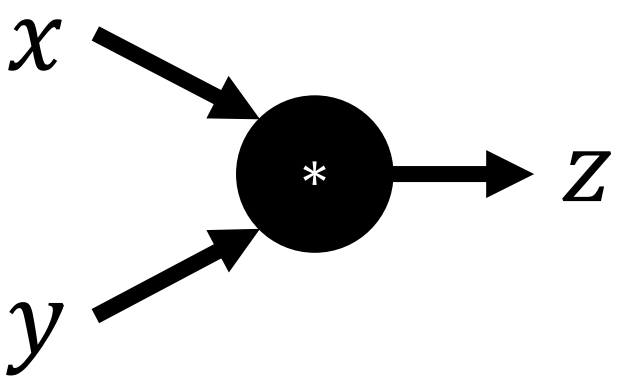
Scalar mul: Scale grad
Mul: Swap grad
Max: Route grad
Sum: Distribute grad
Splits: Accumulate grad

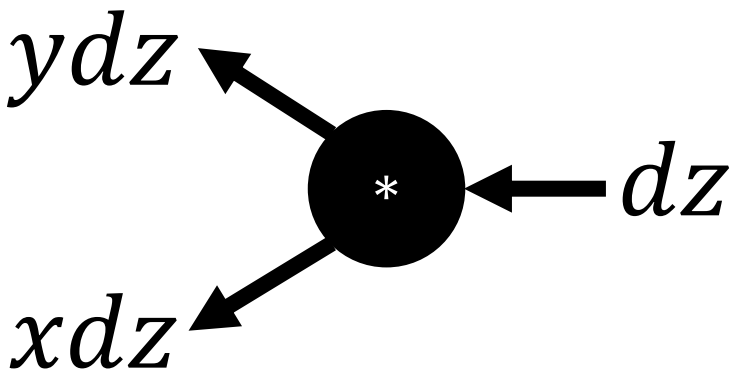
$$\frac{\delta_y}{\delta x_1} = -\frac{4}{4.04}\cos(5) - \frac{4}{4.04}$$

$$\frac{\delta_y}{\delta x_2} = 1.212$$

$$\frac{\delta_y}{\delta x_3} = 0$$

Modularized Implementation: Forward / Backward API






```
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]
```

Modern Autograd Frameworks Make This Easy

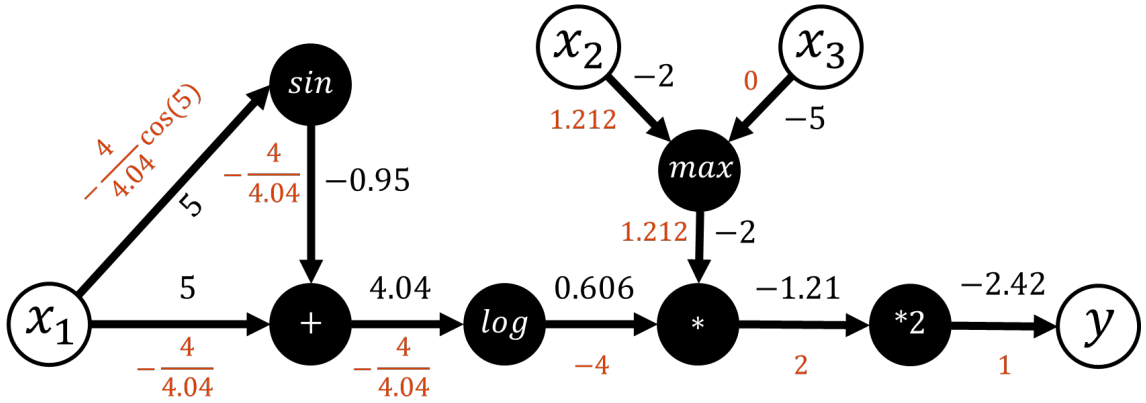
Implement special variable types (e.g., Tensor in PyTorch) that:

- Override most operations with corresponding forward/backward APIs
- Dynamically build a computation graph as the variables are manipulated

Implement engine that can run backpropagation over the computation graph

Modern Autograd Frameworks -- PyTorch

$$y = 2\log(x_1 + \sin(x_1))\max(x_2, x_3)$$



$$\frac{\delta_y}{\delta x_1} = -\frac{4}{4.04}\cos(5) - \frac{4}{4.04} \qquad \frac{\delta_y}{\delta x_2} = 1.213 \qquad \frac{\delta_y}{\delta x_3} = 0$$

```
import torch
import numpy as np

x = torch.tensor([5.0, -2.0, -5.0], requires_grad=True)
y = 2*torch.log( x[0] + torch.sin(x[0]))/np.log(10)*torch.max( x[1:])

y.backward()
print(x.grad)
```

```
tensor([-0.5518,  1.2130,  0.0000])
```

Modern Autograd Frameworks -- PyTorch

```
at::Tensor norm_backward(const at::Tensor & grad, const at::Tensor & self, const optional<at::Scalar> & p_, const at::Tensor & norm);
at::Tensor norm_backward(at::Tensor grad, const at::Tensor & self, const optional<at::Scalar> & p_, at::Tensor norm, at::IntArrayRef dim, bool keepdim);
at::Tensor pow_backward(at::Tensor grad, const at::Tensor & self, const at::Scalar & exponent_);
at::Tensor pow_backward_self(at::Tensor grad, const at::Tensor & self, const at::Tensor & exponent);
at::Tensor pow_backward_exponent(at::Tensor grad, const at::Tensor& self, const at::Tensor& exponent, at::Tensor result);
at::Tensor pow_backward_exponent(at::Tensor grad, const at::Scalar & base, const at::Tensor& exponent, at::Tensor result);
at::Tensor angle_backward(at::Tensor grad, const at::Tensor& self);
at::Tensor mul_tensor_backward(Tensor grad, Tensor other, ScalarType self_st);
at::Tensor div_tensor_self_backward(Tensor grad, Tensor other, ScalarType self_st);
at::Tensor div_tensor_other_backward(Tensor grad, Tensor self, Tensor other);
at::Tensor mvlgamma_backward(at::Tensor grad, const at::Tensor & self, int64_t p);
at::Tensor permute_backwards(const at::Tensor & grad, at::IntArrayRef fwd_dims);
at::Tensor rad2deg_backward(const at::Tensor& grad);
at::Tensor deg2rad_backward(const at::Tensor& grad);
at::Tensor unsqueeze_multiple(const at::Tensor & t, at::IntArrayRef dim, size_t n_dims);
at::Tensor sum_backward(const at::Tensor & grad, at::IntArrayRef sizes, at::IntArrayRef dims, bool keepdim);
at::Tensor nansum_backward(const at::Tensor & grad, const at::Tensor & self, at::IntArrayRef dims, bool keepdim);
std::vector<int64_t> reverse_list(const at::IntArrayRef list);
at::Tensor reverse_dim(const at::Tensor& t, int64_t dim);
at::Tensor prod_safe_zeros_backward(const at::Tensor &grad, const at::Tensor& inp, int64_t dim);
at::Tensor prod_backward(const at::Tensor& grad, const at::Tensor& input, const at::Tensor& result);
at::Tensor prod_backward(at::Tensor grad, const at::Tensor& input, at::Tensor result, int64_t dim, bool keepdim);
at::Tensor solve_backward_self(const at::Tensor & grad, const at::Tensor & self, const at::Tensor & A);
at::Tensor solve_backward_A(const at::Tensor & grad, const at::Tensor & self, const at::Tensor & A, const at::Tensor & solution);
at::Tensor cumsum_backward(const at::Tensor & x, int64_t dim);
at::Tensor logsumexp_backward(at::Tensor grad, const at::Tensor & self, at::Tensor result, at::IntArrayRef dim, bool keepdim);
at::Tensor logcumsumexp_backward(at::Tensor grad, const at::Tensor & self, at::Tensor result, int64_t dim);
```

https://github.com/pytorch/pytorch/blob/master/torch/csrc/autograd/FunctionsM

Modern Autograd Frameworks -- PyTorch

```
Tensor sum_backward(const Tensor & grad, IntArrayRef sizes, IntArrayRef dims, bool keepdim) {
  if (!keepdim && sizes.size() > 0) {
    if (dims.size()==1) {
      return grad.unsqueeze(dims[0]).expand(sizes);
    } else {
      Tensor res = unsqueeze_multiple(grad, dims, sizes.size());
      return res.expand(sizes);
    }
  } else {
    return grad.expand(sizes);
  }
}
```

Next time

Next Time: We'll keep talking about neural networks Thursday and then move on to Decision Trees afterwards.