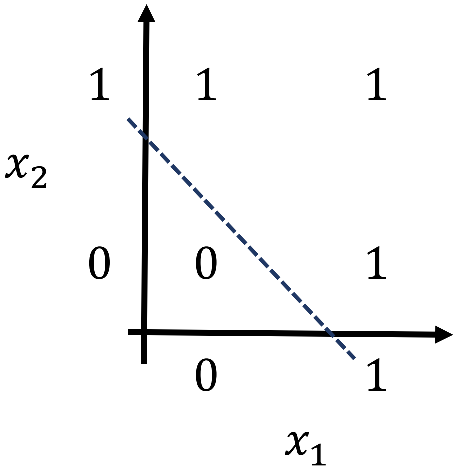
Artificial Intelligence

Shallow Neural Network

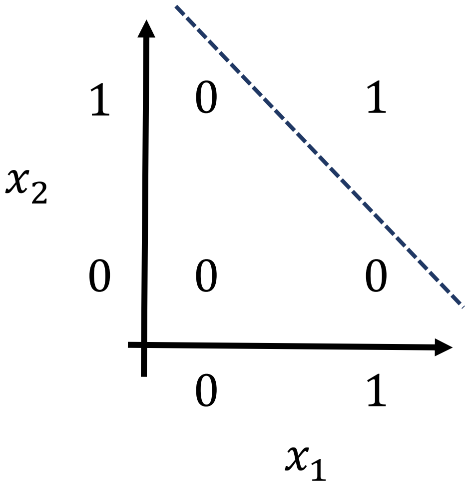
Woohwan Jung

HANYANG UNIVERSITY
Data Science Lab

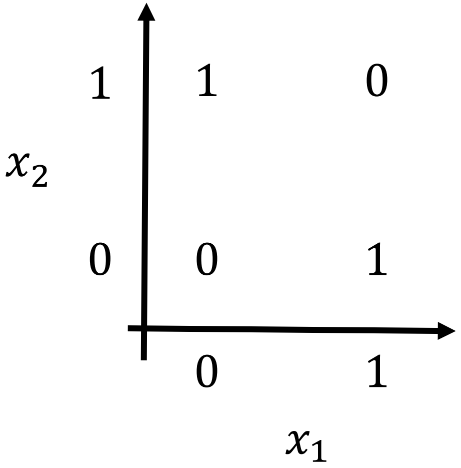
or



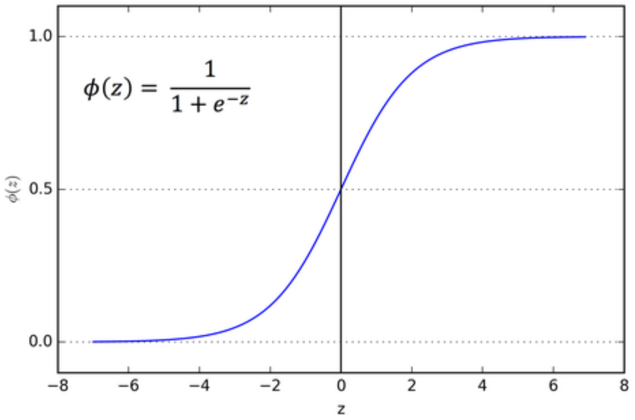
and



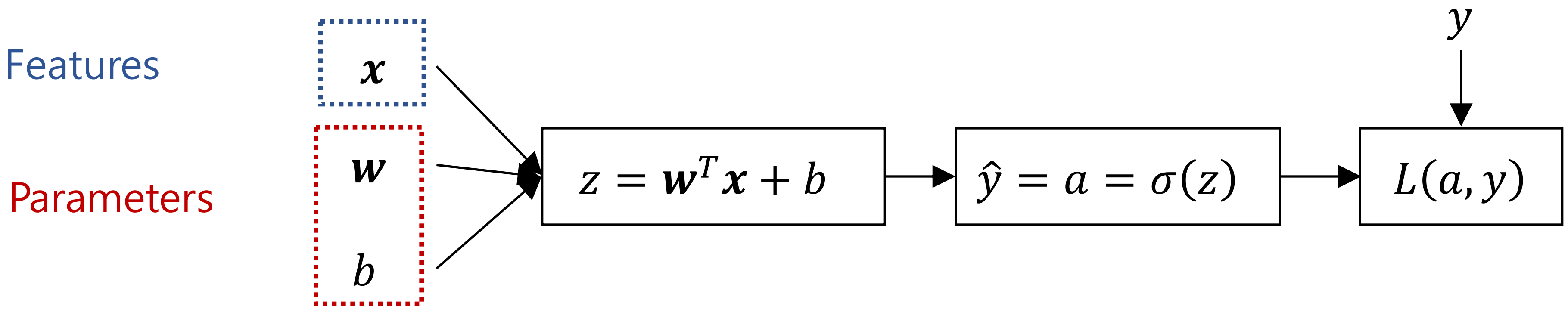
xor



Logistic Regression



- Output: $\hat{y} = \sigma\left(\boldsymbol{w}^{\top}\boldsymbol{x} + b\right)$ where $\sigma(z) = \frac{1}{1+e^{-z}}$
- Loss: $L(\hat{y}, y) = -y \log \hat{y} - (1 - y) \log(1 - \hat{y})$



Features

Parameters

$\boldsymbol{x}$

$\boldsymbol{w}$

b

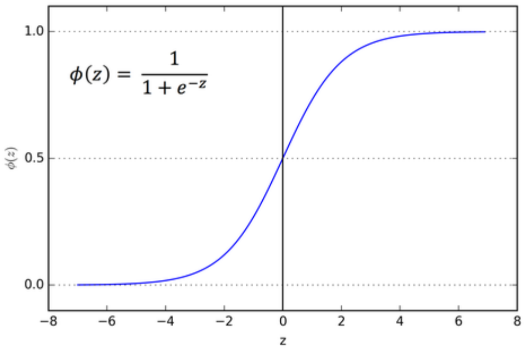
$z = \boldsymbol{w}^T\boldsymbol{x} + b$

$\hat{y} = a = \sigma(z)$

y

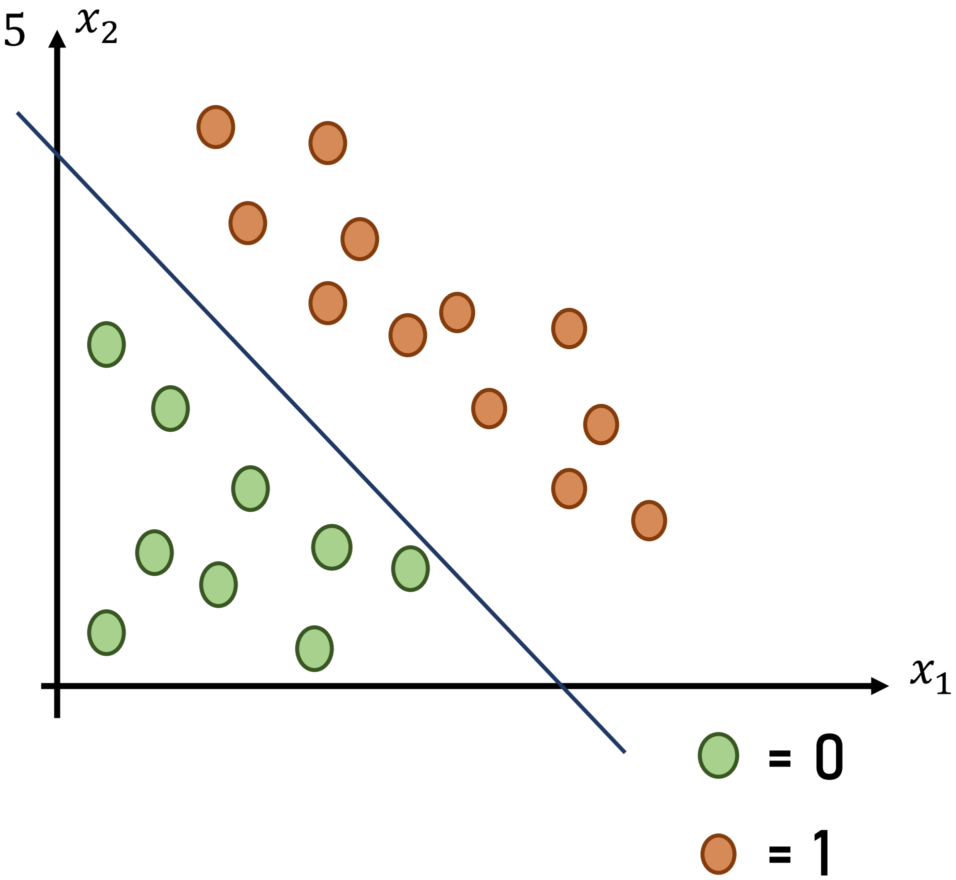
$L(a, y)$

Decision boundary of the logistic regression



$$\boldsymbol{w}^\top \boldsymbol{x} + b = 0$$

$$\sigma\left(\boldsymbol{w}^\top \boldsymbol{x} + b\right) = 0.5$$

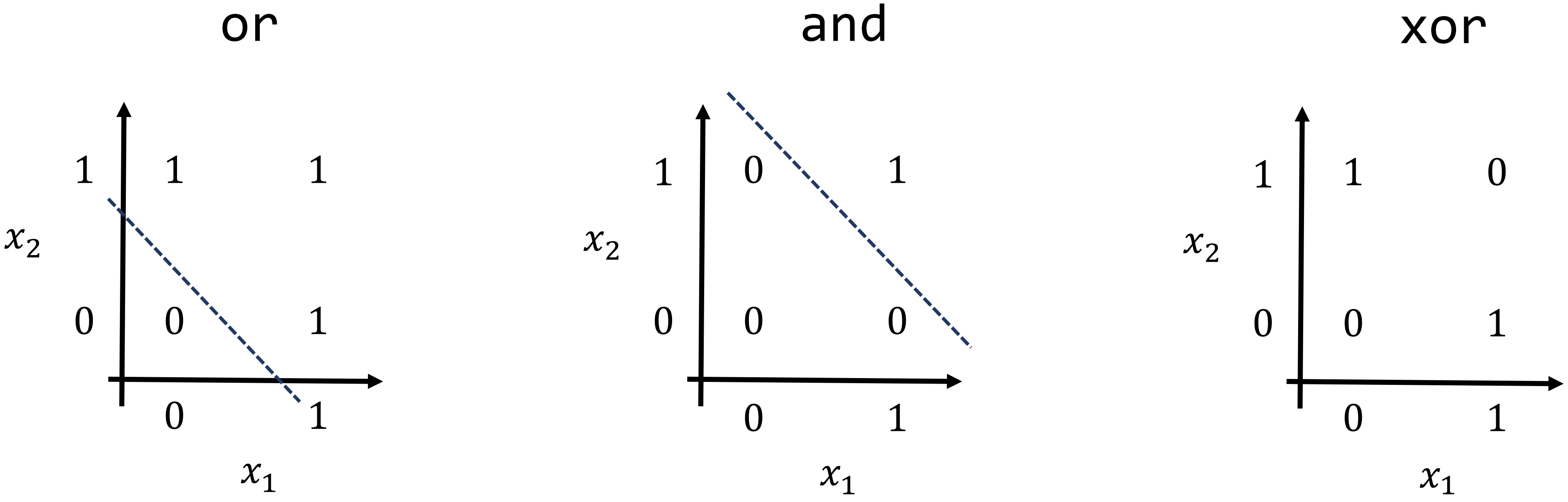


$$\hat{y} = \sigma\left(\boldsymbol{w}^\top \boldsymbol{x} + b\right) \text{ where } \sigma(z) = \frac{1}{1+e^{-z}}$$

$$P(y = 1|\boldsymbol{x}) = \hat{y} = \sigma\left(\boldsymbol{w}^\top \boldsymbol{x} + b\right) > 0.5$$

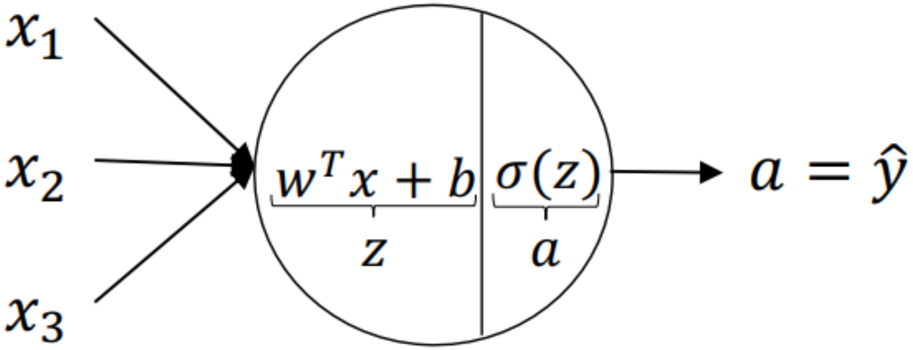
$$\Leftrightarrow \boldsymbol{w}^\top \boldsymbol{x} + b > 0$$

(Simple) XOR problem: linearly separable?



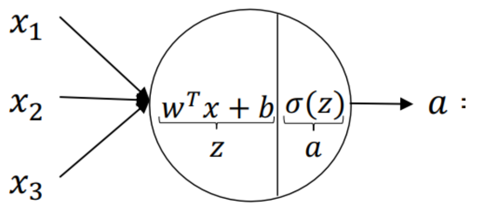
Solution: make it more complicated

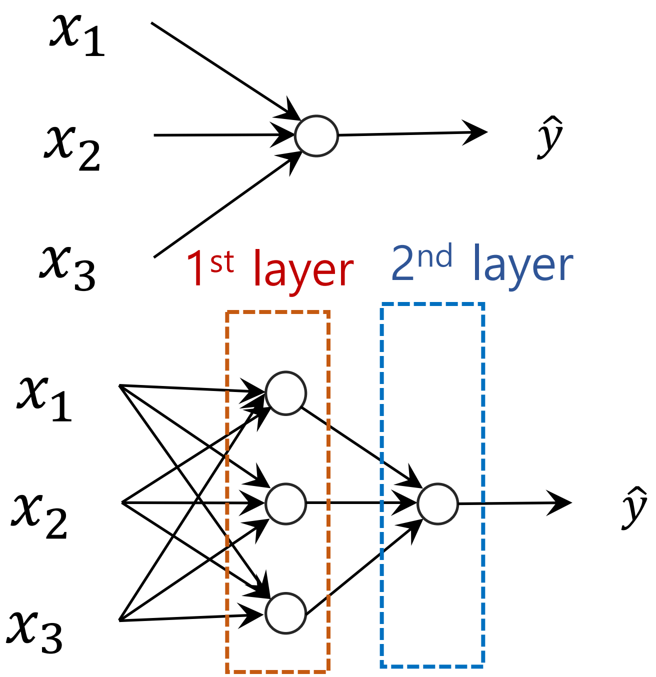
Neuron

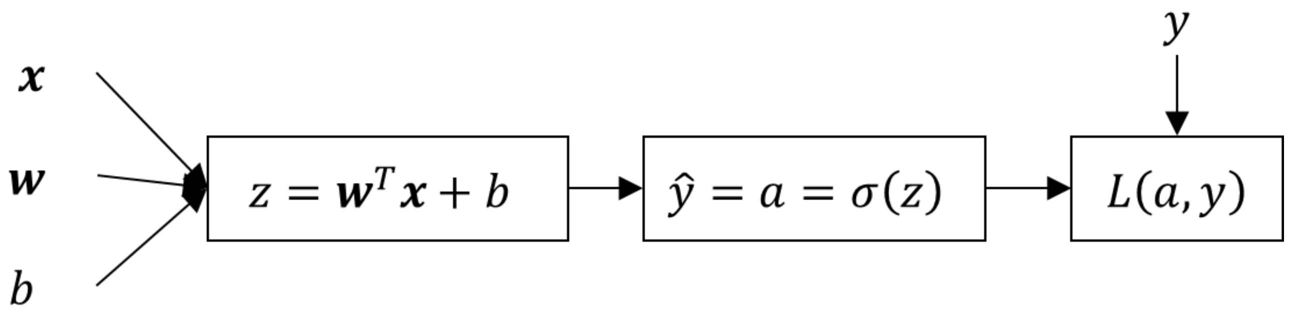


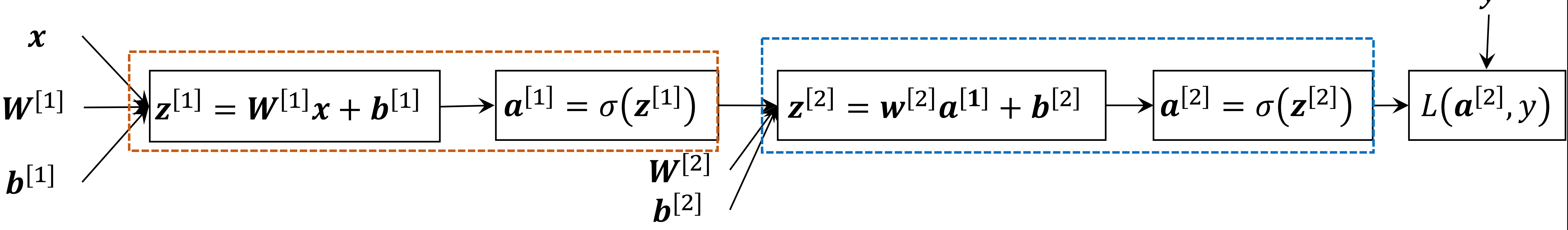
What is a Neural Network?

Neuron

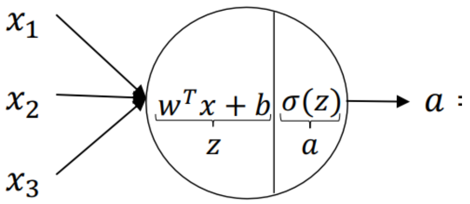






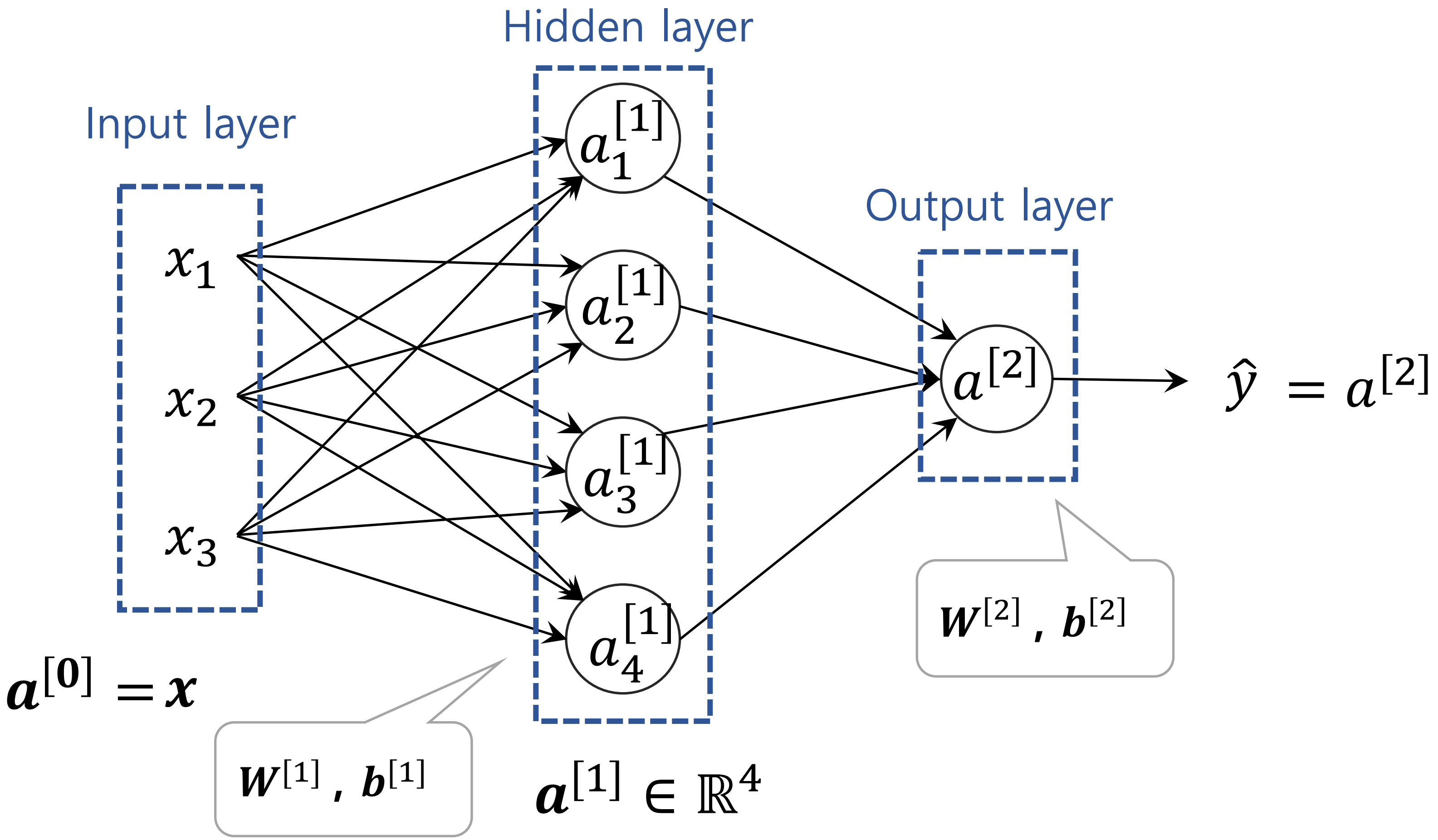


Neuron



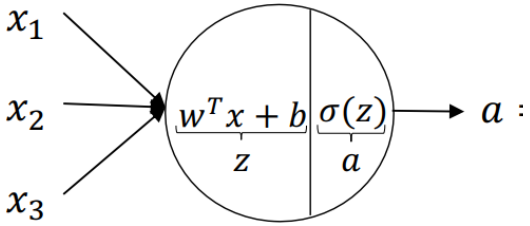
Neural Network Representation

- 2-layer neural network



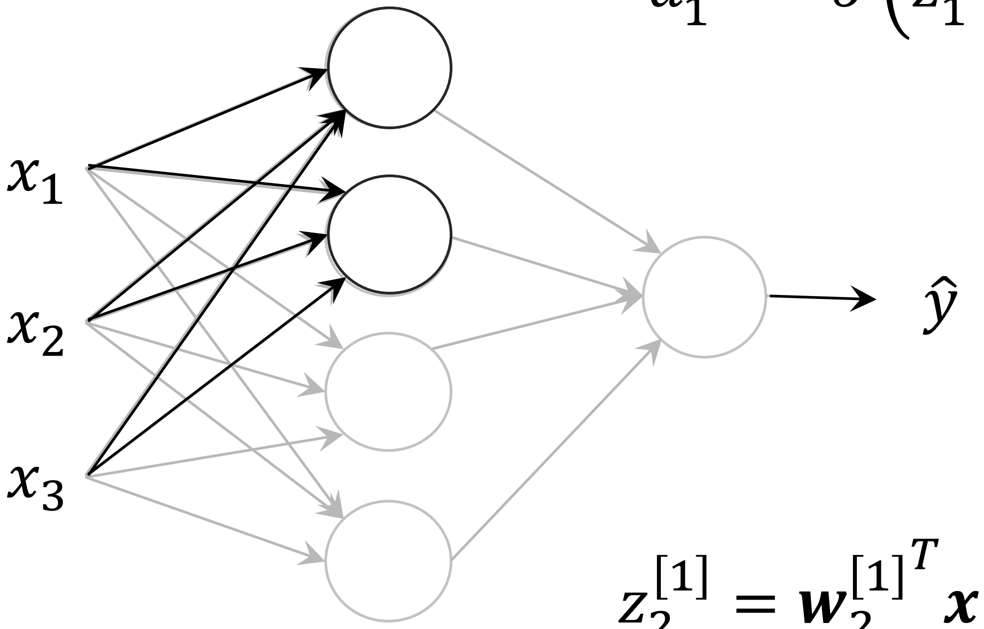
Neural Network Representation

Neuron



$$z_1^{[1]} = \boldsymbol{W}_1^{[1]}\boldsymbol{x} + b_1^{[1]}$$

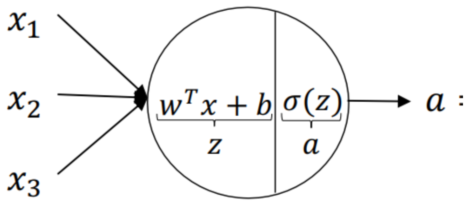
$$a_1^{[1]} = \sigma\left(z_1^{[1]}\right)$$

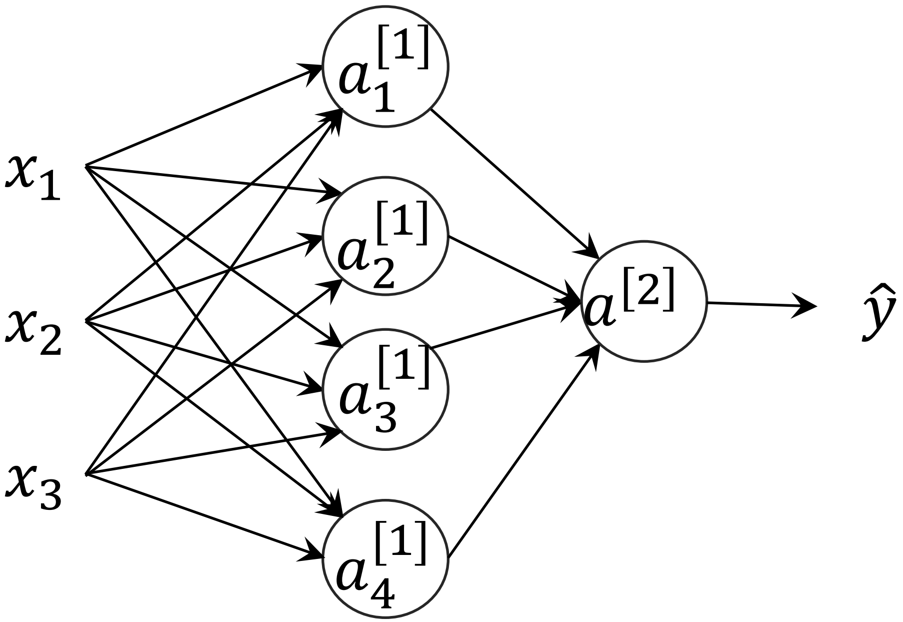


$$z_2^{[1]} = \boldsymbol{w}_2^{[1]^T}\boldsymbol{x} + b_2^{[1]}$$

$$a_2^{[1]} = \sigma\left(z_2^{[1]}\right)$$

Neural Network Representation





$$\boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

$$z_1^{[1]} = \boldsymbol{w}_1^{[1]^T}\boldsymbol{x} + b_1^{[1]} \qquad a_1^{[1]} = \sigma\left(z_1^{[1]}\right)$$

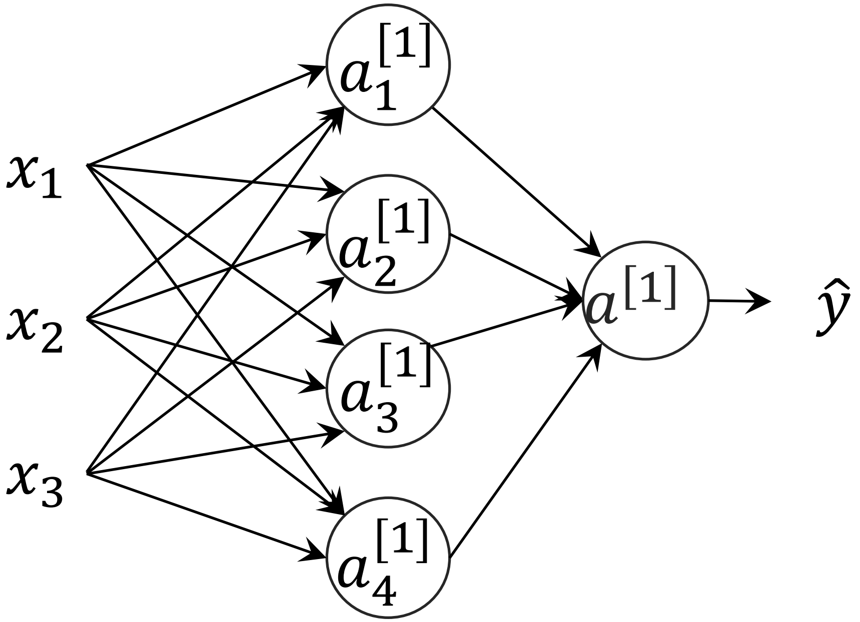
$$z_2^{[1]} = \boldsymbol{w}_2^{[1]^T}\boldsymbol{x} + b_2^{[1]} \qquad a_2^{[1]} = \sigma\left(z_2^{[1]}\right)$$

$$z_3^{[1]} = \boldsymbol{w}_3^{[1]^T}\boldsymbol{x} + b_3^{[1]} \qquad a_3^{[1]} = \sigma\left(z_3^{[1]}\right)$$

$$z_4^{[1]} = \boldsymbol{w}_4^{[1]^T}\boldsymbol{x} + b_4^{[1]} \qquad a_4^{[1]} = \sigma\left(z_4^{[1]}\right)$$

$$z^{[2]} = \boldsymbol{w}^{[2]^T}\boldsymbol{a}^{[1]} + b^{[2]} \qquad a^{[2]} = \sigma\left(z^{[2]}\right)$$

Neural Network with a Hidden Layer: Almost Done!



Input:

$$\boldsymbol{x} \in \mathbb{R}^n$$

Parameters:

$$\boldsymbol{W}^{[1]} \in \mathbb{R}^{h \times n} \qquad \boldsymbol{w}^{[2]} \in \mathbb{R}^h$$

$$\boldsymbol{b}^{[1]} \in \mathbb{R}^h \qquad b^{[2]} \in \mathbb{R}$$

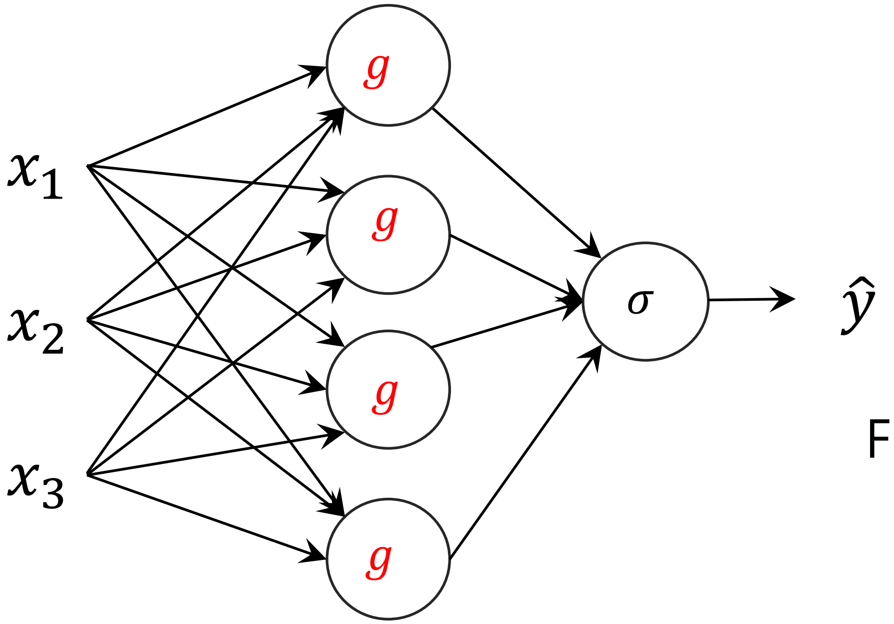
Forward pass:

$$\left.\begin{aligned} \boldsymbol{z}^{[1]} &= \boldsymbol{W}^{[1]}\boldsymbol{x} + \boldsymbol{b}^{[1]} \\ \boldsymbol{a}^{[1]} &= \sigma\left(\boldsymbol{z}^{[1]}\right) \end{aligned}\right\} \quad \text{Hidden layer}$$

$$\left.\begin{aligned} z^{[2]} &= \boldsymbol{w}^{[2]^T}\boldsymbol{a}^{[1]} + b^{[2]} \\ \hat{y} = a^{[2]} &= \sigma\left(z^{[2]}\right) \end{aligned}\right\} \quad \text{Output layer}$$

Activation Functions

Neural Network with a Hidden Layer: Almost Done!



Input:

$\boldsymbol{x} \in \mathbb{R}^n$

Parameters:

$\boldsymbol{W}^{[1]} \in \mathbb{R}^{h \times n}$ $\boldsymbol{w}^{[2]} \in \mathbb{R}^h$

$\boldsymbol{b}^{[1]} \in \mathbb{R}^h$ $b^{[2]} \in \mathbb{R}$

Forward pass:

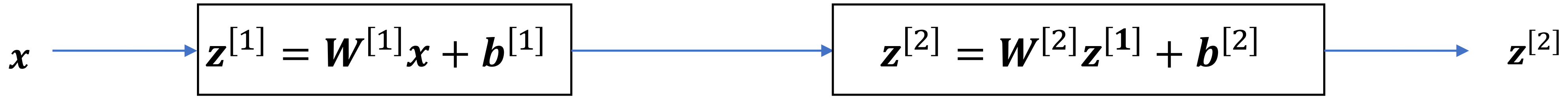
$$\boldsymbol{z}^{[1]} = \boldsymbol{W}^{[1]}\boldsymbol{x} + \boldsymbol{b}^{[1]}$$

$\boldsymbol{a}^{[1]} = g\left(\boldsymbol{z}^{[1]}\right)$ where $g(.)$ is an activation function

$$z^{[2]} = \boldsymbol{w}^{[2]^T}\boldsymbol{a}^{[1]} + b^{[2]}$$

$$\hat{y} = a^{[2]} = \sigma\left(z^{[2]}\right)$$

Why we need to use non-linear activation functions?



$\boldsymbol{x}$

$\boldsymbol{z}^{[1]} = \boldsymbol{W}^{[1]}\boldsymbol{x} + \boldsymbol{b}^{[1]}$

$\boldsymbol{z}^{[2]} = \boldsymbol{W}^{[2]}\boldsymbol{z}^{[1]} + \boldsymbol{b}^{[2]}$

$\boldsymbol{z}^{[2]}$

$$\begin{aligned}\boldsymbol{z}^{[2]} &= \boldsymbol{W}^{[2]}(\boldsymbol{W}^{[1]}\boldsymbol{x} + \boldsymbol{b}^{[1]}) + \boldsymbol{b}^{[2]} \\ &= \boldsymbol{W}^{[2]}\boldsymbol{W}^{[1]}\boldsymbol{x} + (\boldsymbol{W}^{[2]}\boldsymbol{b}^{[1]} + \boldsymbol{b}^{[2]}) \\ &= \boldsymbol{W}'\boldsymbol{x} + \boldsymbol{b}'\end{aligned}$$

Where $\boldsymbol{W}' = \boldsymbol{W}^{[2]}\boldsymbol{W}^{[1]}$ and $\boldsymbol{b}' = \boldsymbol{W}^{[2]}\boldsymbol{b}^{[1]} + \boldsymbol{b}^{[2]}$

Composition of linear functions => linear function

Activation functions

There are so many activation functions ..

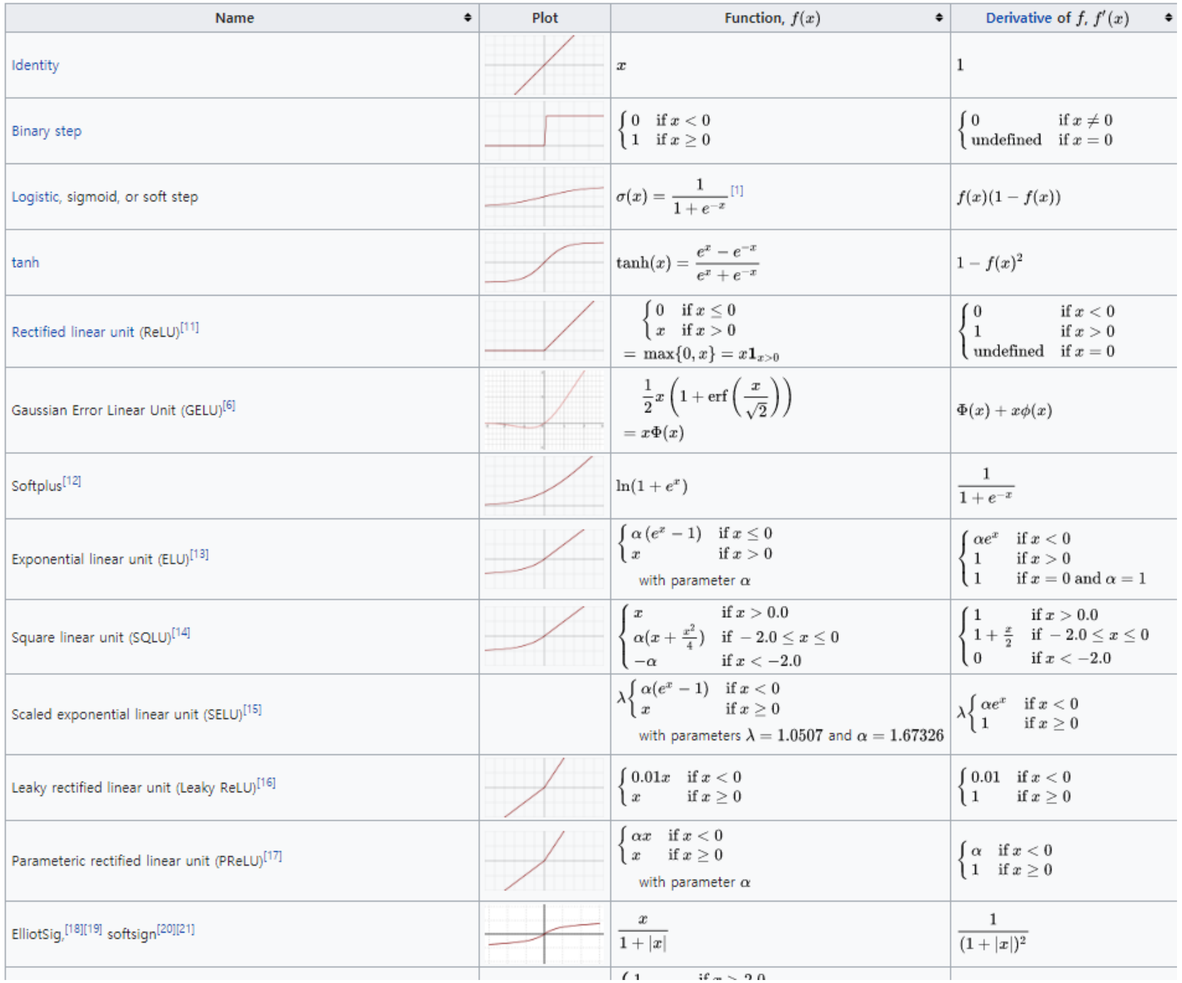
We'll cover some important and currently widely used activation functions

Sigmoid

tanh

ReLU

LeakyReLU



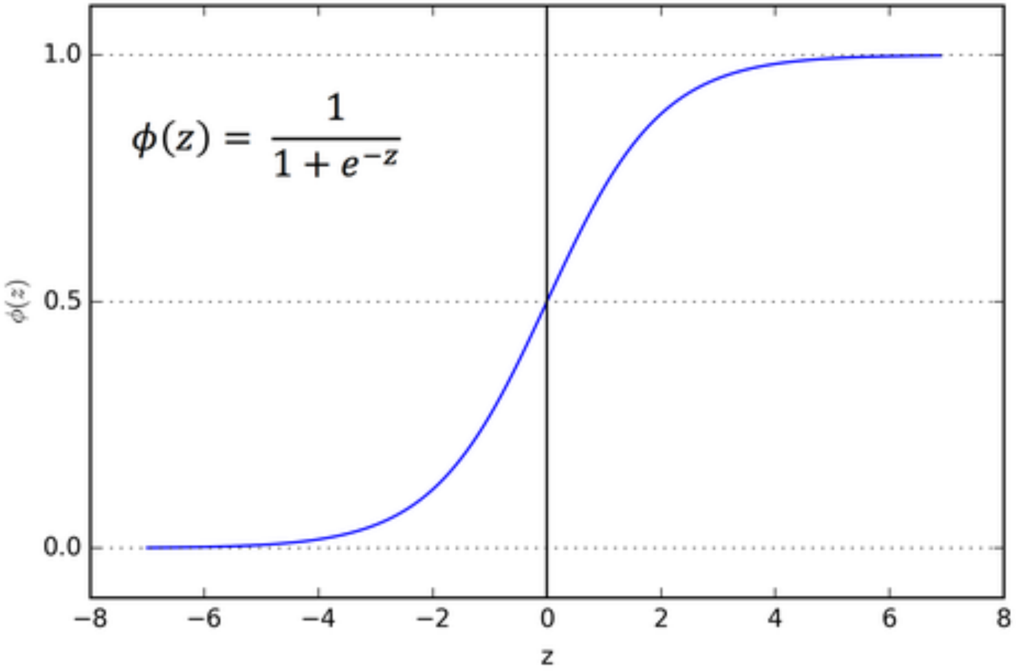
Name	Plot	Function, $f(x)$	Derivative of f, $f'(x)$
Identity		x	1
Binary step		$\begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x \geq 0 \end{cases}$	$\begin{cases} 0 & \text{if } x \neq 0 \\ \text{undefined} & \text{if } x = 0 \end{cases}$
Logistic, sigmoid, or soft step		$\sigma(x) = \frac{1}{1+e^{-x}}$[1]	$f(x)(1-f(x))$
tanh		$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$	$1 - f(x)^2$
Rectified linear unit (ReLU)[11]		$\begin{cases} 0 & \text{if } x \leq 0 \\ x & \text{if } x > 0 \end{cases}$ $= \max\{0, x\} = x\mathbf{1}_{x>0}$	$\begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x > 0 \\ \text{undefined} & \text{if } x = 0 \end{cases}$
Gaussian Error Linear Unit (GELU)[6]		$\frac{1}{2}x\left(1 + \text{erf}\left(\frac{x}{\sqrt{2}}\right)\right)$ $= x\Phi(x)$	$\Phi(x) + x\phi(x)$
Softplus[12]		$\ln(1 + e^x)$	$\frac{1}{1+e^{-x}}$
Exponential linear unit (ELU)[13]		$\begin{cases} \alpha(e^x - 1) & \text{if } x \leq 0 \\ x & \text{if } x > 0 \end{cases}$ with parameter α	$\begin{cases} \alpha e^x & \text{if } x < 0 \\ 1 & \text{if } x > 0 \\ 1 & \text{if } x = 0 \text{ and } \alpha = 1 \end{cases}$
Square linear unit (SQLU)[14]		$\begin{cases} x & \text{if } x > 0.0 \\ \alpha(x + \frac{x^2}{4}) & \text{if } -2.0 \leq x \leq 0 \\ -\alpha & \text{if } x < -2.0 \end{cases}$	$\begin{cases} 1 & \text{if } x > 0.0 \\ 1 + \frac{x}{2} & \text{if } -2.0 \leq x \leq 0 \\ 0 & \text{if } x < -2.0 \end{cases}$
Scaled exponential linear unit (SELU)[15]		$\lambda\begin{cases} \alpha(e^x - 1) & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases}$ with parameters $\lambda = 1.0507$ and $\alpha = 1.67326$	$\lambda\begin{cases} \alpha e^x & \text{if } x < 0 \\ 1 & \text{if } x \geq 0 \end{cases}$
Leaky rectified linear unit (Leaky ReLU)[16]		$\begin{cases} 0.01x & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases}$	$\begin{cases} 0.01 & \text{if } x < 0 \\ 1 & \text{if } x \geq 0 \end{cases}$
Parameteric rectified linear unit (PReLU)[17]		$\begin{cases} \alpha x & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases}$ with parameter α	$\begin{cases} \alpha & \text{if } x < 0 \\ 1 & \text{if } x \geq 0 \end{cases}$
ElliotSig,[18][19] softsign[20][21]		$\frac{x}{1 + \lvert x \rvert}$	$\frac{1}{(1 + \lvert x \rvert)^2}$

Sigmoid

- $\sigma(x) = \frac{1}{1+e^{-x}}$
- Range: $(0,1)$
- Derivative

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

  - $0 < \frac{d\sigma(x)}{dx} \leq 0.25$



Hyperbolic tangent: tanh

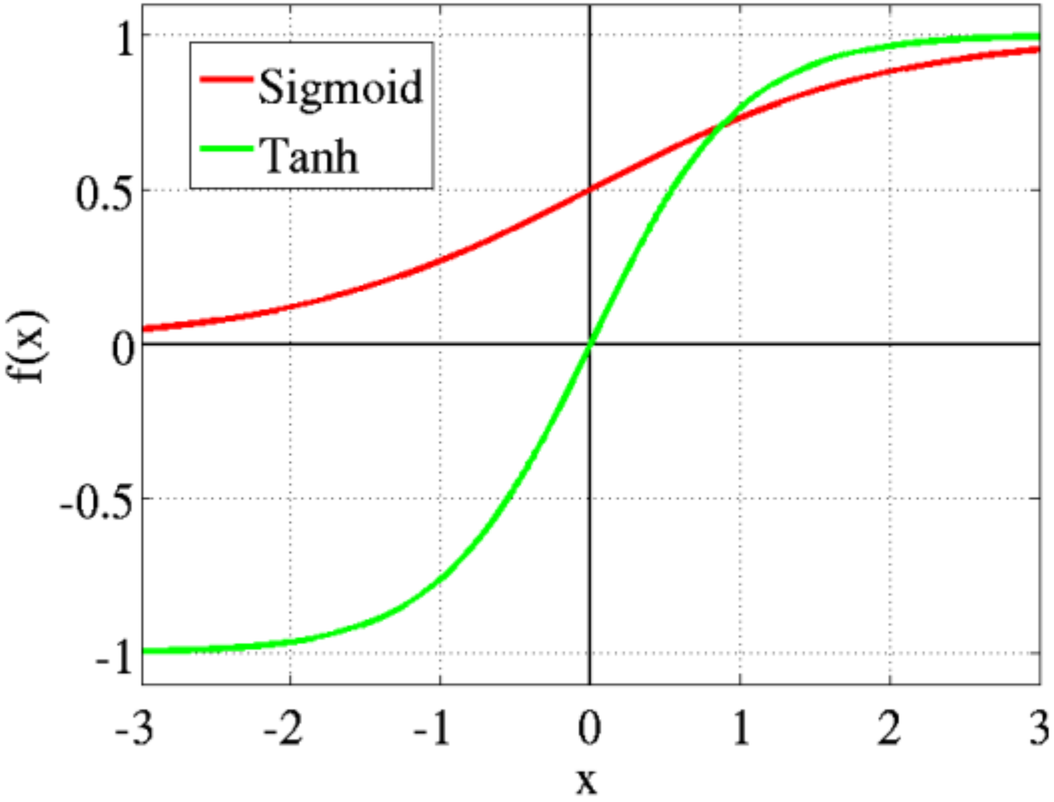
- $\tanh x = \frac{e^x - e^{-x}}{e^x + e^{-x}}$

  $= 2\sigma(2x) - 1$

- Range: $(-1,1)$
- Derivative

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

  - $0 < \frac{d\tanh x}{dx} \leq 1$



Vanishing gradient

> "The term vanishing gradient refers to the fact that in a feedforward network (FFN) the backpropagated error signal typically decreases (or increases) exponentially as a function of the distance from the final layer" by Jason Brownlee

https://machinelearningmastery.com/how-to-fix-vanishing-gradients-using-the-rectified-linear-activation-function/

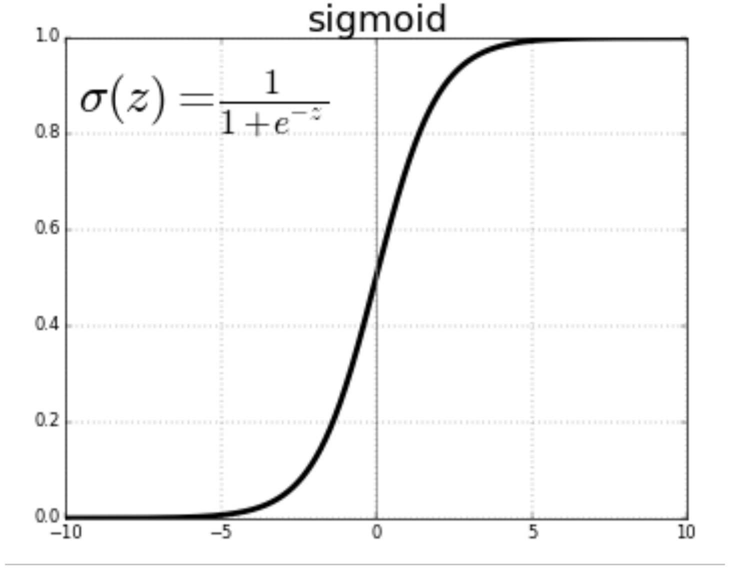
- Chain rule (for a Deep NN)

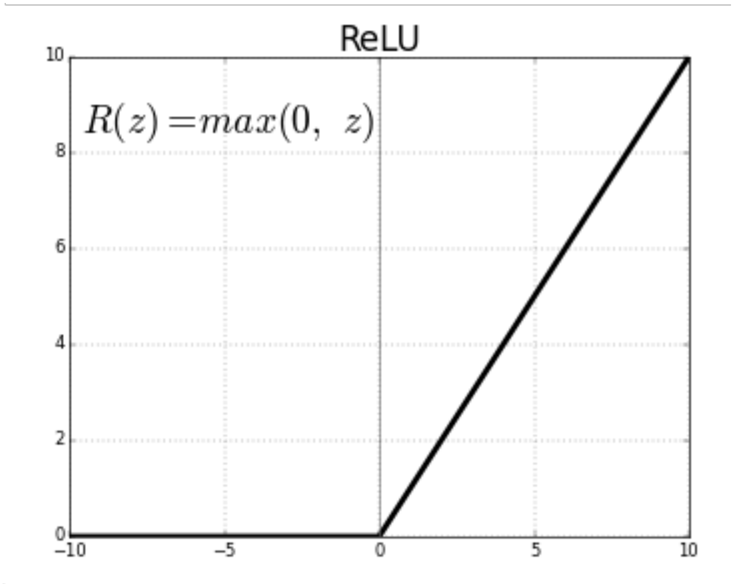
$$\frac{\partial L}{\partial z^{[n]}} \frac{\partial z^{[n]}}{\partial z^{[n-1]}} \cdots \frac{\partial z^{[3]}}{\partial z^{[2]}} \frac{\partial z^{[2]}}{\partial z^{[1]}} \frac{\partial z^{[1]}}{\partial w}$$

Rectified linear unit: ReLU

- $f(x) = \max\{0, x\}$
- Range: $[0, \infty)$
- Derivative

$$\frac{df(x)}{dx} = \begin{cases} 0 & if\ x < 0 \\ 1 & if x > 0 \\ undefined & if\ x = 0 \end{cases}$$



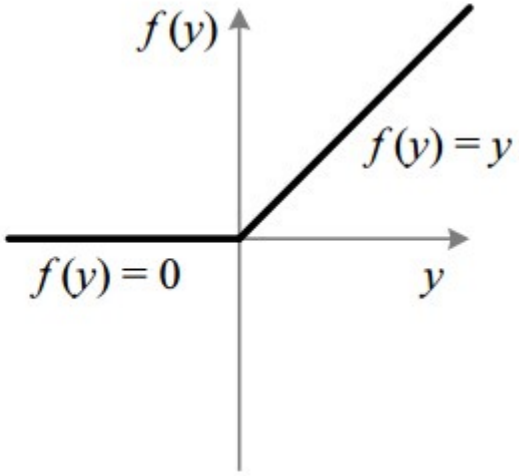


Leaky ReLU

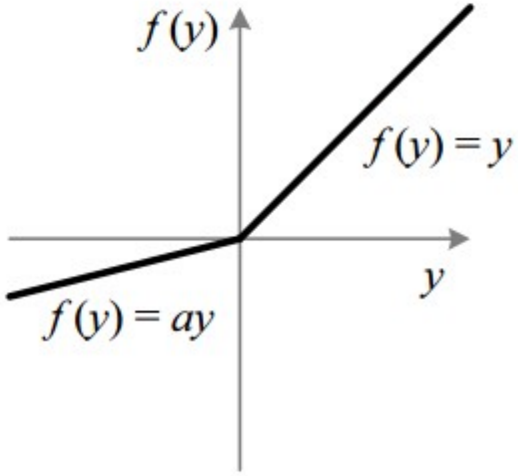
- $f(x) = \begin{cases} ax & if\ x < 0 \\ x & if\ x \geq 0 \end{cases}$
    - $a \ll 1$ (e.g, $a = 0.01$)
- Range: $(-\infty, \infty)$
- Derivative

$$\frac{df(x)}{dx} = \begin{cases} a & if\ x < 0 \\ 1 & ifx > 0 \\ undefined & if\ x = 0 \end{cases}$$

ReLU



Leaky ReLU

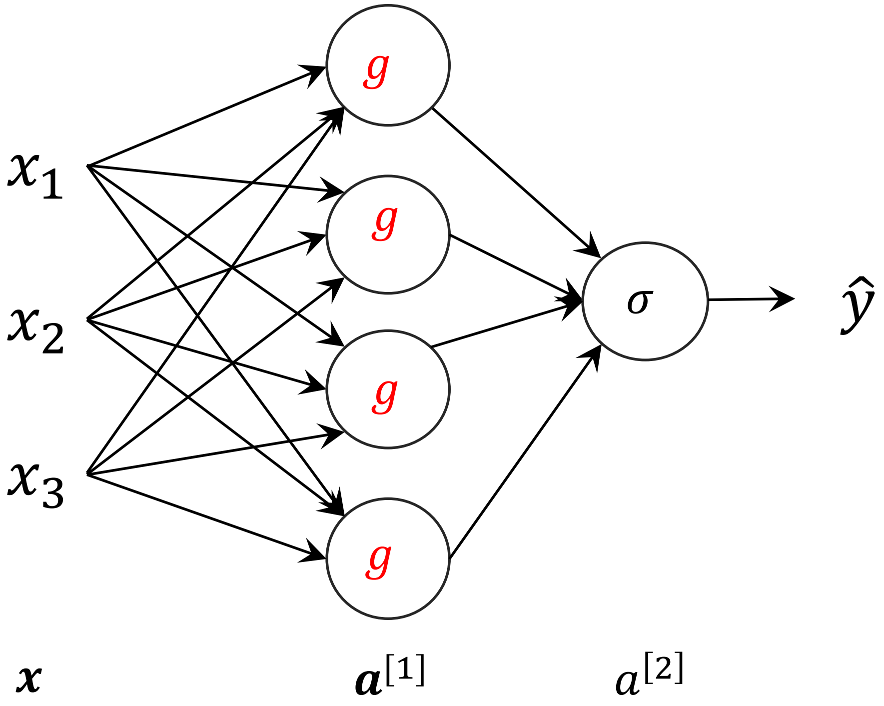


Activation functions

- There is no rule but … in many cases
- Output layer
  - Sigmoid
- Hidden layer
  - tanh, ReLU, LeakyReLU

My personal opinion:
If the performance of two things are similar,
Use the simpler one! (Use ReLU!)

Neural Network with a Hidden Layer: Done!



$$\boldsymbol{z}^{[1]} = \boldsymbol{W}^{[1]}\boldsymbol{x} + \boldsymbol{b}^{[1]}$$

$\boldsymbol{a}^{[1]} = g\left(\boldsymbol{z}^{[1]}\right)$ where $g(.)$ is an activation function

$$z^{[2]} = \boldsymbol{w}^{[2]^T}\boldsymbol{a}^{[1]} + b^{[2]}$$

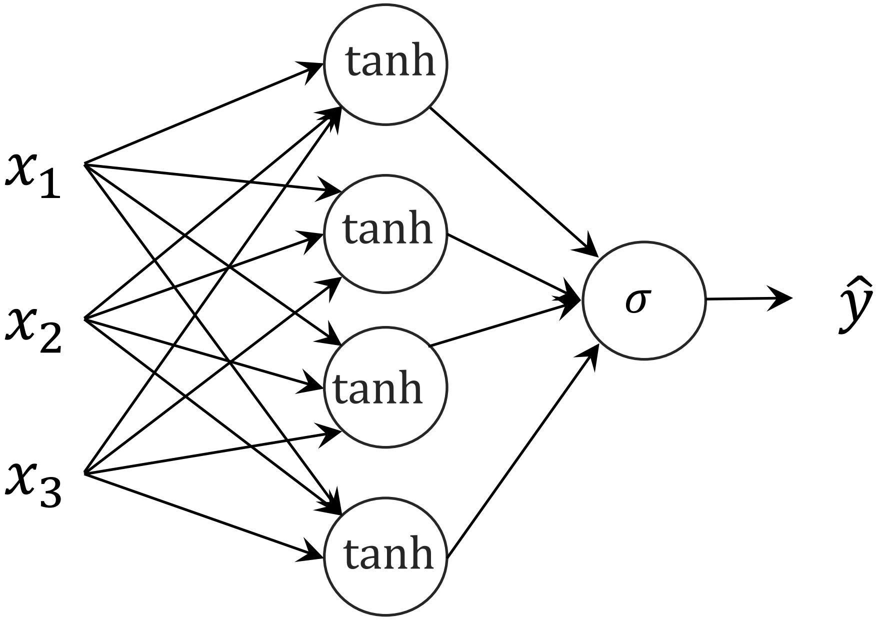
$$\hat{y} = a^{[2]} = \sigma\left(z^{[2]}\right)$$

You may use **tanh, ReLU, Leaky ReLU** as g

Gradient Descent

Shallow Neural Network with tanh

Shallow Neural Network



$$\boldsymbol{z}^{[1]} = \boldsymbol{W}^{[1]}\boldsymbol{x} + \boldsymbol{b}^{[1]}$$

$$\boldsymbol{a}^{[1]} = \tanh \boldsymbol{z}^{[1]}$$

$$z^{[2]} = \boldsymbol{w}^{[2]^T}\boldsymbol{a}^{[1]} + b^{[2]}$$

$$\hat{y} = a^{[2]} = \sigma\left(z^{[2]}\right)$$

Parameters: $\boldsymbol{\theta} = \{\boldsymbol{W}^{[1]}, \boldsymbol{b}^{[1]}, \boldsymbol{w}^{[2]}, b^{[2]}\}$

Loss $L(\hat{y}, y)$
Cost $J(\theta) = \sum_{q=1}^{m} L\left(\hat{y}^{(q)}, y^{(q)}\right)$

Gradient descent

$$\boldsymbol{\theta} \coloneqq \boldsymbol{\theta} - \eta \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \boldsymbol{\theta} - \eta \frac{\sum_{i=1}^{m} \nabla_{\boldsymbol{\theta}} L\left(\hat{y}^{(q)}, y^{(q)}\right)}{m}$$

So, we will compute $\nabla_{\boldsymbol{\theta}} L\left(\hat{y}^{(q)}, y^{(q)}\right)$ For $1 \le q \le m$

Element-wise representations

$$\frac{\partial L\left(a^{[2]}, y\right)}{\partial b^{[2]}}$$

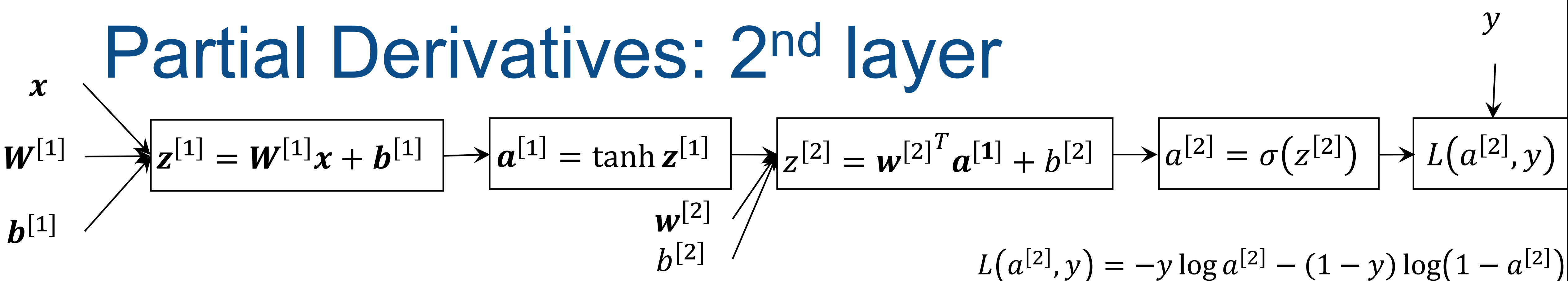
$$\frac{\partial L\left(a^{[2]}, y\right)}{\partial w_i^{[2]}} \quad \text{For } 1 \le i \le h$$

$$\frac{\partial L\left(a^{[2]}, y\right)}{\partial b_i^{[1]}} \quad \text{For } 1 \le i \le h$$

$$\frac{\partial L\left(a^{[2]}, y\right)}{\partial W_{ij}^{[1]}} \quad \text{For } 1 \le i \le h,\ 1 \le j \le n$$

Parameters $\boldsymbol{W}^{[1]}$, $\boldsymbol{b}^{[1]}$, $\boldsymbol{w}^{[2]}$, $b^{[2]}$

Partial Derivatives: 2nd layer



y

$\boldsymbol{x}$

$\boldsymbol{W}^{[1]}$

$\boldsymbol{b}^{[1]}$

$\boldsymbol{z}^{[1]} = \boldsymbol{W}^{[1]}\boldsymbol{x} + \boldsymbol{b}^{[1]}$

$\boldsymbol{a}^{[1]} = \tanh \boldsymbol{z}^{[1]}$

$\boldsymbol{w}^{[2]}$

$b^{[2]}$

$z^{[2]} = \boldsymbol{w}^{[2]^T}\boldsymbol{a}^{[1]} + b^{[2]}$

$a^{[2]} = \sigma(z^{[2]})$

$L(a^{[2]}, y)$

$$L(a^{[2]}, y) = -y \log a^{[2]} - (1-y)\log(1-a^{[2]})$$

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

- $$\frac{\partial L(a^{[2]},y)}{\partial b^{[2]}} = \frac{\partial L(a^{[2]},y)}{\partial a^{[2]}}\frac{\partial a^{[2]}}{\partial z^{[2]}}\frac{\partial z^{[2]}}{\partial b^{[2]}}$$

$$= \left(\frac{-y}{a^{[2]}} + \frac{1-y}{1-a^{[2]}}\right)\sigma(z^{[2]})\left(1-\sigma(z^{[2]})\right)$$

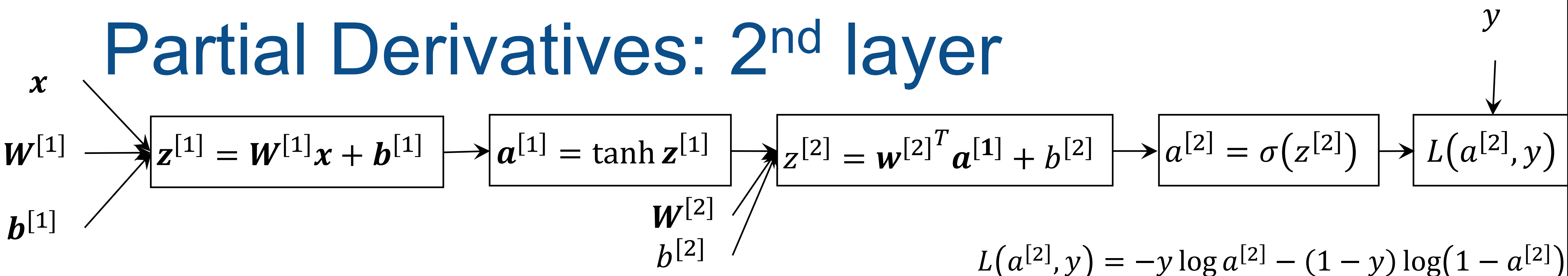
$$= \left(\frac{-y}{a^{[2]}} + \frac{1-y}{1-a^{[2]}}\right)a^{[2]}(1-a^{[2]})$$

$$= -y(1-a^{[2]}) + a^{[2]}(1-y) = a^{[2]} - \mathrm{y}$$

Do you remember?

Partial Derivatives: 2nd layer

Parameters $\boldsymbol{W}^{[1]}$, $\boldsymbol{b}^{[1]}$, $\boldsymbol{w}^{[2]}$, $b^{[2]}$



$$L(a^{[2]}, y) = -y \log a^{[2]} - (1-y)\log(1-a^{[2]})$$

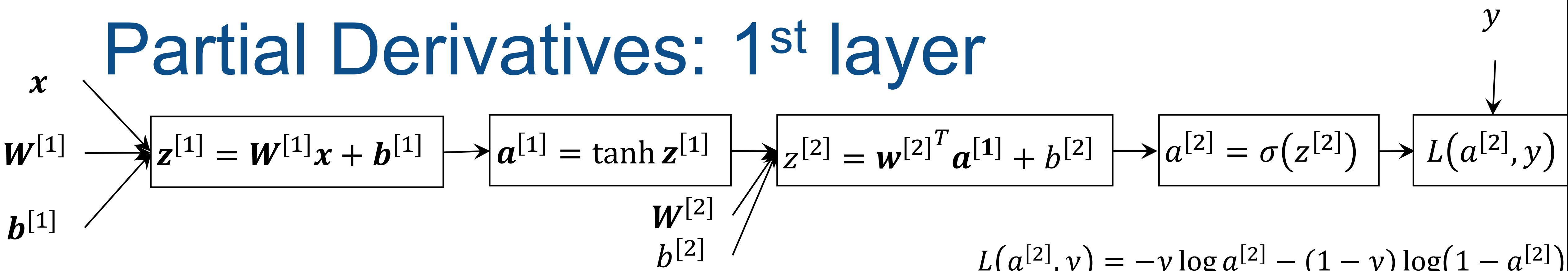
- $$\frac{\partial L(a^{[2]},y)}{\partial w_i^{[2]}} = \frac{\partial L(a^{[2]},y)}{\partial a^{[2]}} \frac{\partial a^{[2]}}{\partial z^{[2]}} \frac{\partial z^{[2]}}{\partial w_i^{[2]}}$$

$$= (a^{[2]} - y)\frac{\partial z^{[2]}}{\partial w_i^{[2]}}$$

$$= (a^{[2]} - y)\ a_i^{[1]}$$

Parameters $\boldsymbol{W}^{[1]}$, $\boldsymbol{b}^{[1]}, \boldsymbol{w}^{[2]}$, $b^{[2]}$

Partial Derivatives: 1st layer



$$L(a^{[2]}, y) = -y \log a^{[2]} - (1 - y) \log(1 - a^{[2]})$$

- $$\frac{\partial L(a^{[2]}, y)}{\partial b_i^{[1]}} = \frac{\partial L(a^{[2]}, y)}{\partial a^{[2]}} \frac{\partial a^{[2]}}{\partial z^{[2]}} \frac{\partial z^{[2]}}{\partial a_i^{[1]}} \frac{\partial a_i^{[1]}}{\partial z_i^{[1]}} \frac{\partial z_i^{[1]}}{\partial b_i^{[1]}}$$

$$= (a^{[2]} - y) \frac{\partial z^{[2]}}{\partial a_i^{[1]}} \frac{\partial a_i^{[1]}}{\partial z_i^{[1]}} \frac{\partial z_i^{[1]}}{\partial b_i^{[1]}}$$

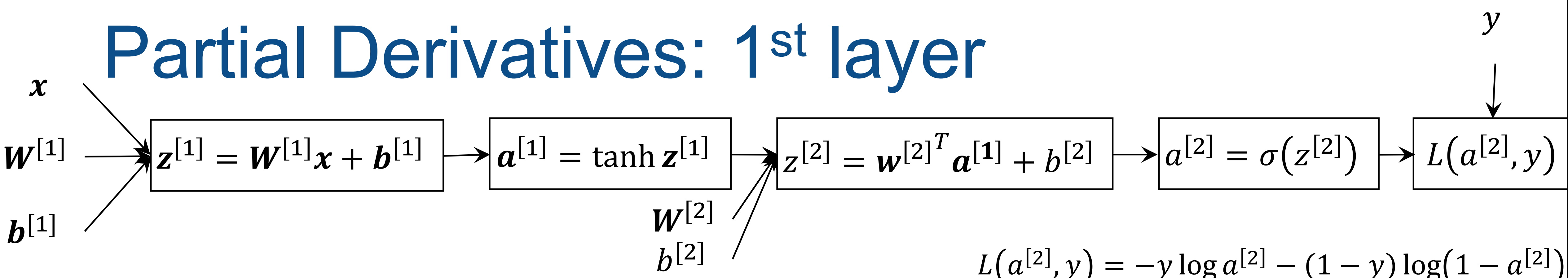
$$= (a^{[2]} - y) w_i^{[2]} \left(1 - \tanh^2 z_i^{[1]}\right) \cdot 1$$

$$= (a^{[2]} - y) w_i^{[2]} \left(1 - a_i^{[1]^2}\right)$$

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

Parameters $\boldsymbol{W}^{[1]}$, $\boldsymbol{b}^{[1]}, \boldsymbol{w}^{[2]}$, $b^{[2]}$

Partial Derivatives: 1st layer



$$L(a^{[2]}, y) = -y \log a^{[2]} - (1-y)\log(1-a^{[2]})$$

- $$\frac{\partial L(a^{[2]},y)}{\partial W_{ij}^{[1]}} = \frac{\partial L(a^{[2]},y)}{\partial a^{[2]}} \frac{\partial a^{[2]}}{\partial z^{[2]}} \frac{\partial z^{[2]}}{\partial a_i^{[1]}} \frac{\partial a_i^{[1]}}{\partial z_i^{[1]}} \frac{\partial z_i^{[1]}}{\partial W_{ij}^{[1]}}$$

$$= (a^{[2]} - y) w_i^{[2]} \left(1 - a_i^{[1]^2}\right) \frac{\partial z_i^{[1]}}{\partial W_{ij}^{[1]}}$$

$$= (a^{[2]} - y) w_i^{[2]} \left(1 - a_i^{[1]^2}\right) x_j$$

Note

$$z_i^{[1]} = \boldsymbol{w}_i^{[1]^T}\boldsymbol{x} + b_i^{[1]}$$

$$= \sum_{j=1}^{m} W_{ij}^{[1]} x_j + b_i^{[1]}$$

$$\Rightarrow \frac{\partial z_i^{[1]}}{\partial W_{ij}^{[1]}} = x_j$$

Partial derivatives

- $\frac{\partial L(a^{[2]},y)}{\partial b^{[2]}} = a^{[2]} - \mathrm{y}$
- $\frac{\partial L(a^{[2]},y)}{\partial w_i^{[2]}} = \left(a^{[2]} - y\right)\, a_i^{[1]}$
- $\frac{\partial L(a^{[2]},y)}{\partial b_i^{[1]}} = \left(a^{[2]} - y\right) w_i^{[2]} \left(1 - {a_i^{[1]}}^2\right)$
- $\frac{\partial L(a^{[2]},y)}{\partial W_{ij}^{[1]}} = \left(a^{[2]} - y\right) w_i^{[2]} \left(1 - {a_i^{[1]}}^2\right) x_j$

Gradient Descent

- Let $\boldsymbol{x} \in \mathbb{R}^n$
- We have h hidden units
- Randomly initialize the parameters
  - $\boldsymbol{W}^{[1]} \in \mathbb{R}^{h \times n}$
  - $\boldsymbol{b}^{[1]} \in \mathbb{R}^h$
  - $\boldsymbol{w}^{[2]} \in \mathbb{R}^h$
  - $b^{[2]} \in \mathbb{R}$

- For each epoch
  - $d\boldsymbol{W}^{[1]} = \boldsymbol{0} \in \mathbb{R}^{h \times n}$, $d\boldsymbol{b}^{[1]} = \boldsymbol{0} \in \mathbb{R}^h$
  - $d\boldsymbol{w}^{[2]} = \boldsymbol{0} \in \mathbb{R}^h$, $db^{[2]} = 0 \in \mathbb{R}$
  - For $(x, y) \in D$
    - $dw_i^{[2]} \mathrel{+}= \frac{\partial L(a^{[2]}, y)}{\partial w_i^{[2]}}$ for $1 \leq i \leq h$
    - $db^{[2]} \mathrel{+}= \frac{\partial L(a^{[2]}, y)}{\partial b^{[2]}}$
    - $dW_{ij}^{[1]} \mathrel{+}= \frac{\partial L(a^{[2]}, y)}{\partial w_{ij}^{[1]}}$ for $1 \leq i \leq n$, $1 \leq j \leq h$
    - $db_i^{[1]} \mathrel{+}= \frac{\partial L(a^{[2]}, y)}{\partial b_i^{[1]}}$ for $1 \leq i \leq h$,
  - $\boldsymbol{W}^{[1]} \mathrel{-}= \eta \cdot d\boldsymbol{W}^{[1]} / |D|$
  - $\boldsymbol{b}^{[1]} \mathrel{-}= \eta \cdot d\boldsymbol{b}^{[1]} / |D|$
  - $\boldsymbol{w}^{[2]} \mathrel{-}= \eta \cdot d\boldsymbol{w}^{[1]} / |D|$
  - $b^{[2]} \mathrel{-}= \eta \cdot db^{[2]} / |D|$

Vectorization: what and why?

- Vectorization: operations are applied to whole arrays instead of individual elements
- Why?
    - Simpler
        - easy to implement and understand
    - Much faster
        - Modern CPUs GPUs implements an instruction set where its instructions are designed to operate efficiently and effectively on large vectors

https://en.wikipedia.org/wiki/Vectorization

https://en.wikipedia.org/wiki/Vector_processor

https://www.quantifisolutions.com/vectorization-part-2-why-and-what/

Partial Derivatives and Gradients

Partial derivatives w.r.t. $w_i^{[2]}$

$$\frac{\partial L\left(a^{[2]}, y\right)}{\partial w_i^{[2]}} = \left(a^{[2]} - y\right)\ a_i^{[1]}$$

for $1 \le i \le h$

Gradients w.r.t. $\boldsymbol{w}^{[2]} = \left[w_1^{[2]} \quad w_2^{[2]} \quad \dots \quad w_h^{[2]}\right]^T$

$$\nabla_{\boldsymbol{w}^{[2]}} L\left(a^{[2]}, y\right) = \left[\frac{\partial L\left(a^{[2]}, y\right)}{\partial w_1^{[2]}} \quad \frac{\partial L\left(a^{[2]}, y\right)}{\partial w_2^{[2]}} \quad \dots \quad \frac{\partial L\left(a^{[2]}, y\right)}{\partial w_h^{[2]}}\right]^T$$

$$= \left[\left(a^{[2]} - y\right)a_1^{[1]} \quad \left(a^{[2]} - y\right)a_2^{[1]} \quad \dots \quad \left(a^{[2]} - y\right)a_h^{[1]}\right]$$

$$= \left(a^{[2]} - y\right)\left[a_1^{[1]} \quad a_2^{[1]} \quad \dots \quad a_h^{[1]}\right]^T$$

$$= \left(a^{[2]} - y\right)\boldsymbol{a}^{[1]}$$

Partial Derivatives and Gradients

Outer product

$$\mathbf{u} \otimes \mathbf{v} = \mathbf{u}\mathbf{v}^T = \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \end{bmatrix} \begin{bmatrix} v_1 & v_2 & v_3 \end{bmatrix} = \begin{bmatrix} u_1 v_1 & u_1 v_2 & u_1 v_3 \\ u_2 v_1 & u_2 v_2 & u_2 v_3 \\ u_3 v_1 & u_3 v_2 & u_3 v_3 \\ u_4 v_1 & u_4 v_2 & u_4 v_3 \end{bmatrix}$$

Partial Derivatives

$$\frac{\partial L\left(a^{[2]}, y\right)}{\partial b^{[2]}} = a^{[2]} - \mathrm{y}$$

$$\frac{\partial L\left(a^{[2]}, y\right)}{\partial w_i^{[2]}} = \left(a^{[2]} - y\right)\ a_i^{[1]}$$

$$\frac{\partial L\left(a^{[2]}, y\right)}{\partial b_i^{[1]}} = \left(a^{[2]} - y\right) w_i^{[2]} \left(1 - {a_i^{[1]}}^2\right)$$

$$\frac{\partial L\left(a^{[2]}, y\right)}{\partial W_{ij}^{[1]}} = \left(a^{[2]} - y\right) w_i^{[2]} \left(1 - {a_i^{[1]}}^2\right) x_j$$

Gradients

$$\frac{\partial L\left(a^{[2]}, y\right)}{\partial b^{[2]}} = a^{[2]} - \mathrm{y}$$

$$\nabla_{\boldsymbol{w}^{[2]}} L\left(a^{[2]}, y\right) = \left(a^{[2]} - y\right) \boldsymbol{a}^{[1]}$$

$$\nabla_{\boldsymbol{b}^{[1]}} L\left(a^{[2]}, y\right) = \left(a^{[2]} - y\right) \boldsymbol{w}^{[2]} \odot \boldsymbol{e}^{[1]}$$

$$\nabla_{\boldsymbol{W}^{[1]}} L\left(a^{[2]}, y\right) = \left(a^{[2]} - y\right) \left(\boldsymbol{w}^{[2]} \odot \boldsymbol{e}^{[1]}\right) \otimes \boldsymbol{x}$$

where $\boldsymbol{e}^{[1]} = \left(1 - \boldsymbol{a}^{[1]} \odot \boldsymbol{a}^{[1]}\right)$

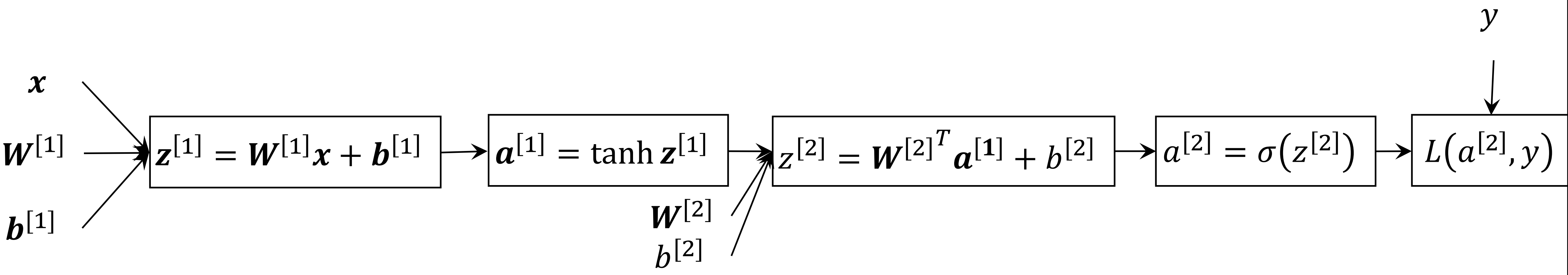
$\odot$: element-wise product (a.k.a. Hadamard pro
$\otimes$: outer product

Gradient Descent

- Let $\boldsymbol{x} \in \mathbb{R}^n$
- We have h hidden units
- Randomly initialize the parameters
    - $\boldsymbol{W}^{[1]} \in \mathbb{R}^{h \times n}$
    - $\boldsymbol{b}^{[1]} \in \mathbb{R}^{h}$
    - $\boldsymbol{w}^{[2]} \in \mathbb{R}^{h}$
    - $b^{[2]} \in \mathbb{R}$

- For each epoch
    - $d\boldsymbol{W}^{[1]} = \boldsymbol{0} \in \mathbb{R}^{h \times n}$, $d\boldsymbol{b}^{[1]} = \boldsymbol{0} \in \mathbb{R}^{h}$
    - $d\boldsymbol{w}^{[2]} = \boldsymbol{0} \in \mathbb{R}^{h}$, $db^{[2]} = 0 \in \mathbb{R}$
    - For $(x, y) \in D$
        - $d\boldsymbol{w}^{[2]} \mathrel{+}= \nabla_{\boldsymbol{w}^{[2]}} L\left(a^{[2]}, y\right)$
        - $db^{[2]} \mathrel{+}= \frac{\partial L\left(a^{[2]}, y\right)}{\partial b^{[2]}}$
        - $d\boldsymbol{W}^{[1]} \mathrel{+}= \nabla_{\boldsymbol{W}^{[1]}} L\left(a^{[2]}, y\right)$
        - $d\boldsymbol{b}^{[1]} \mathrel{+}= \nabla_{\boldsymbol{b}^{[1]}} L\left(a^{[2]}, y\right)$
    - $\boldsymbol{W}^{[1]} \mathrel{-}= \eta \cdot d\boldsymbol{W}^{[1]} / |D|$
    - $\boldsymbol{b}^{[1]} \mathrel{-}= \eta \cdot d\boldsymbol{b}^{[1]} / |D|$
    - $\boldsymbol{w}^{[2]} \mathrel{-}= \eta \cdot d\boldsymbol{w}^{[1]} / |D|$
    - $b^{[2]} \mathrel{-}= \eta \cdot db^{[2]} / |D|$

Initialize with the same value?



$\boldsymbol{x}$

$\boldsymbol{W}^{[1]}$

$\boldsymbol{b}^{[1]}$

$\boldsymbol{z}^{[1]} = \boldsymbol{W}^{[1]}\boldsymbol{x} + \boldsymbol{b}^{[1]}$

$\boldsymbol{a}^{[1]} = \tanh \boldsymbol{z}^{[1]}$

$\boldsymbol{W}^{[2]}$

$b^{[2]}$

$z^{[2]} = \boldsymbol{W}^{[2]^T}\boldsymbol{a}^{[1]} + b^{[2]}$

$a^{[2]} = \sigma\left(z^{[2]}\right)$

y

$L\left(a^{[2]}, y\right)$

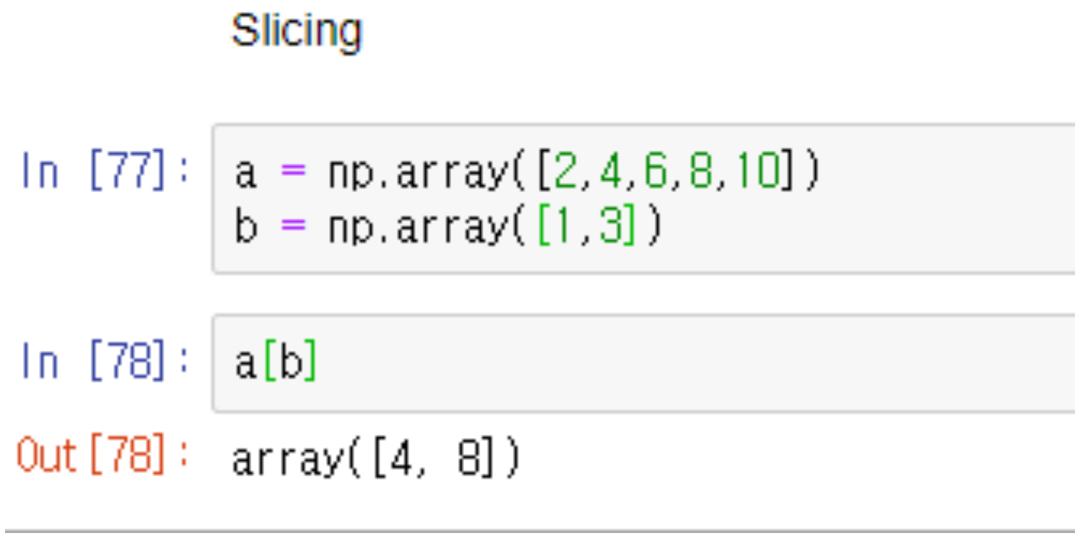
$$\frac{\partial L\left(a^{[2]}, y\right)}{\partial b_i^{[1]}} = \left(1 - {a_i^{[1]}}^2\right) w_i^{[2]}\left(a^{[2]} - y\right)$$

$$\frac{\partial L\left(a^{[2]}, y\right)}{\partial b_1^{[1]}} = \frac{\partial L\left(a^{[2]}, y\right)}{\partial b_2^{[1]}} = \frac{\partial L\left(a^{[2]}, y\right)}{\partial b_3^{[1]}} = \cdots$$

Programming Exercise

Shallow neural network for XOR

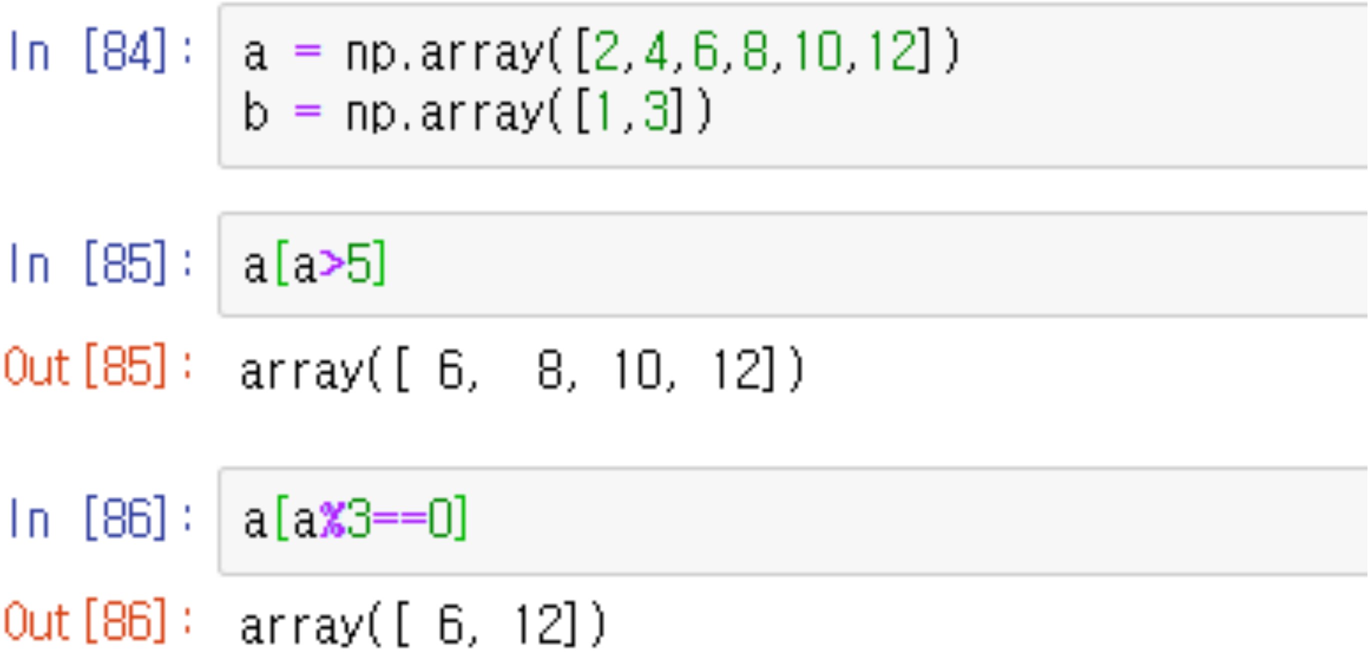
Slicing a numpy array



Slicing

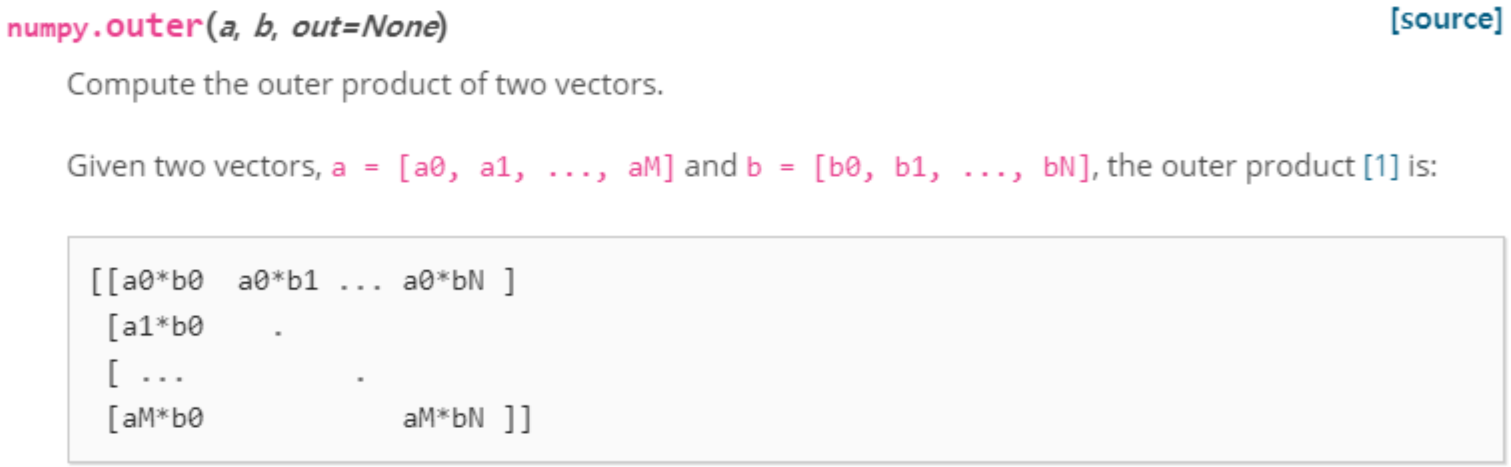
```
In [77]: a = np.array([2,4,6,8,10])
         b = np.array([1,3])

In [78]: a[b]

Out[78]: array([4, 8])
```

Slicing a numpy array with condition



```
In [84]: a = np.array([2,4,6,8,10,12])
         b = np.array([1,3])

In [85]: a[a>5]

Out[85]: array([ 6,  8, 10, 12])

In [86]: a[a%3==0]

Out[86]: array([ 6, 12])
```

Outer product

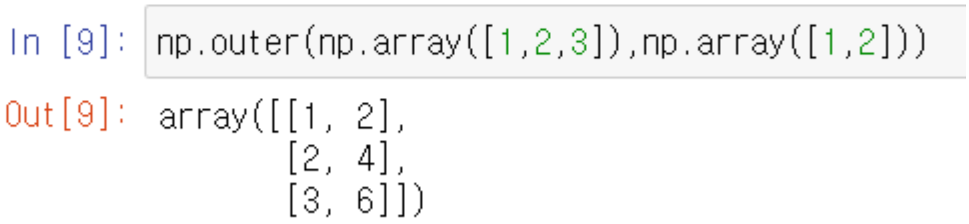


`numpy.outer(a, b, out=None)` [source]

Compute the outer product of two vectors.

Given two vectors, `a = [a0, a1, ..., aM]` and `b = [b0, b1, ..., bN]`, the outer product [1] is:

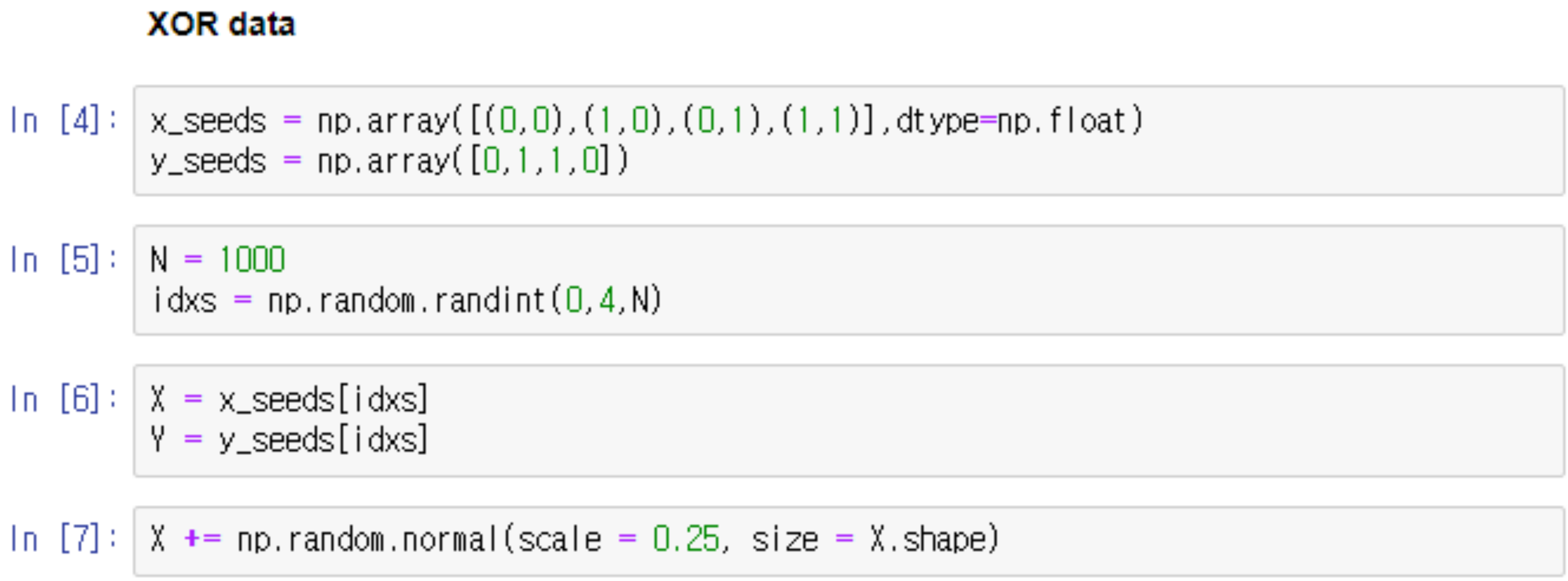
```
[[a0*b0  a0*b1 ... a0*bN ]
 [a1*b0    .
 [ ...          .
 [aM*b0            aM*bN ]]
```



```
In [9]: np.outer(np.array([1,2,3]),np.array([1,2]))

Out[9]: array([[1, 2],
               [2, 4],
               [3, 6]])
```

$$\frac{\partial L\left(a^{[2]},y\right)}{\partial W_{ij}^{[1]}} = \left(a^{[2]}-y\right)w_i^{[2]}\left(1-a_i^{[1]^2}\right)x_j$$

Data preparation



XOR data

```
In [4]: x_seeds = np.array([(0,0),(1,0),(0,1),(1,1)],dtype=np.float)
        y_seeds = np.array([0,1,1,0])
```

```
In [5]: N = 1000
        idxs = np.random.randint(0,4,N)
```

```
In [6]: X = x_seeds[idxs]
        Y = y_seeds[idxs]
```

```
In [7]: X += np.random.normal(scale = 0.25, size = X.shape)
```

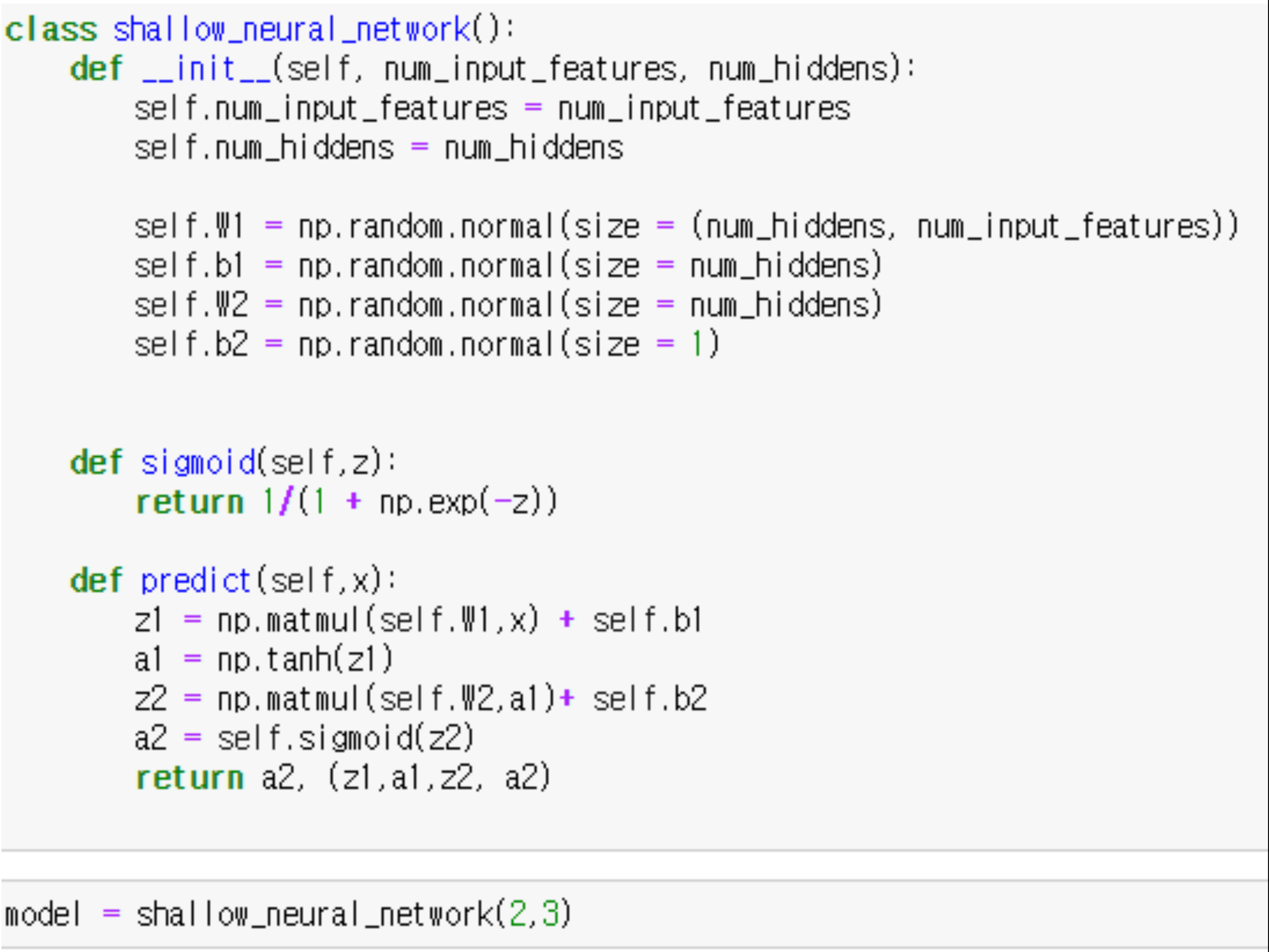
Model

$$\boldsymbol{z}^{[1]} = \boldsymbol{W}^{[1]}\boldsymbol{x} + \boldsymbol{b}^{[1]}$$

$$\boldsymbol{a}^{[1]} = \tanh \boldsymbol{z}^{[1]}$$

$$z^{[2]} = \boldsymbol{w}^{[2]^T}\boldsymbol{a}^{[1]} + b^{[2]}$$

$$\hat{y} = a^{[2]} = \sigma\left(z^{[2]}\right)$$



```
class shallow_neural_network():
    def __init__(self, num_input_features, num_hiddens):
        self.num_input_features = num_input_features
        self.num_hiddens = num_hiddens

        self.W1 = np.random.normal(size = (num_hiddens, num_input_features))
        self.b1 = np.random.normal(size = num_hiddens)
        self.W2 = np.random.normal(size = num_hiddens)
        self.b2 = np.random.normal(size = 1)


    def sigmoid(self,z):
        return 1/(1 + np.exp(-z))

    def predict(self,x):
        z1 = np.matmul(self.W1,x) + self.b1
        a1 = np.tanh(z1)
        z2 = np.matmul(self.W2,a1)+ self.b2
        a2 = self.sigmoid(z2)
        return a2, (z1,a1,z2, a2)
```

```
model = shallow_neural_network(2,3)
```

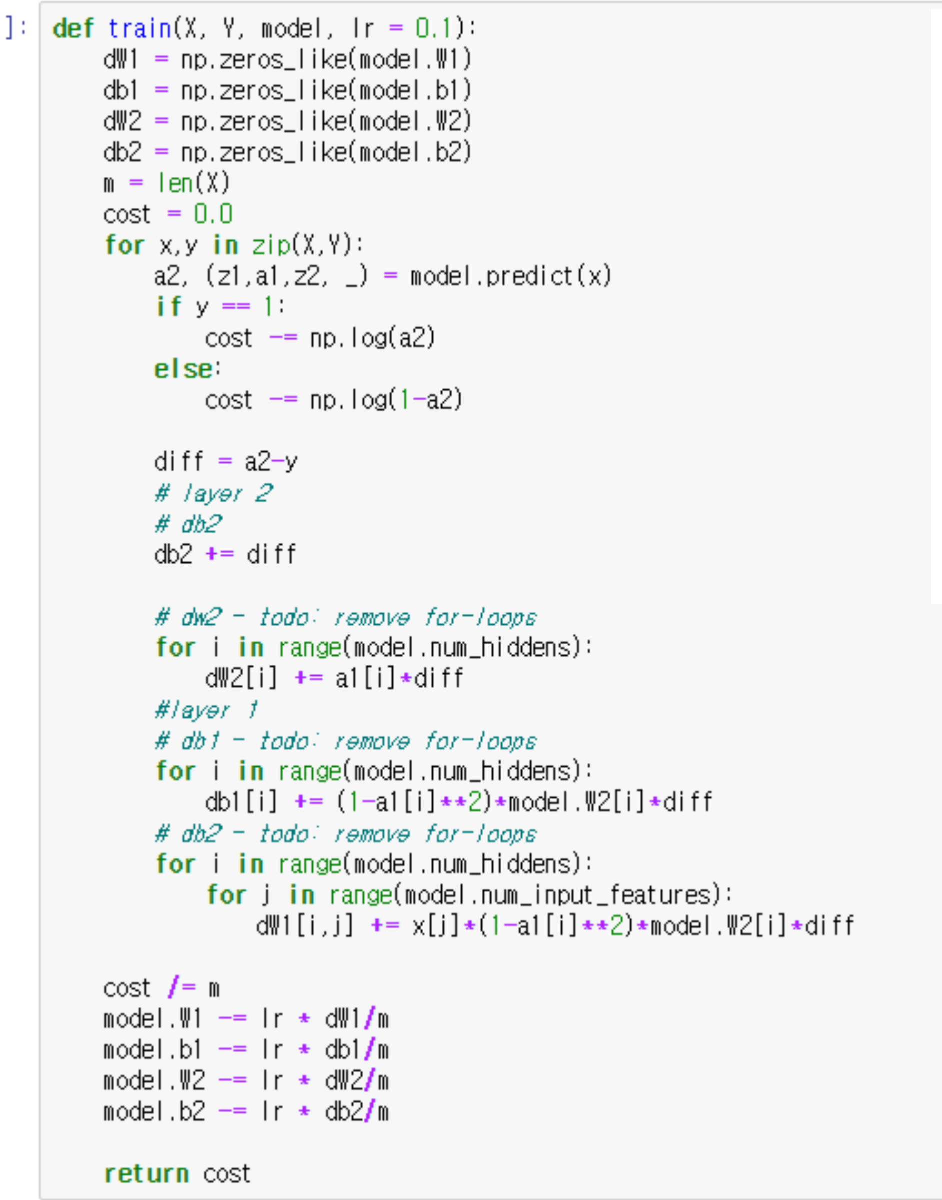
Train (with element-wise operations)

$$\frac{\partial L(a^{[2]},y)}{\partial b^{[2]}} = a^{[2]} - \mathrm{y}$$

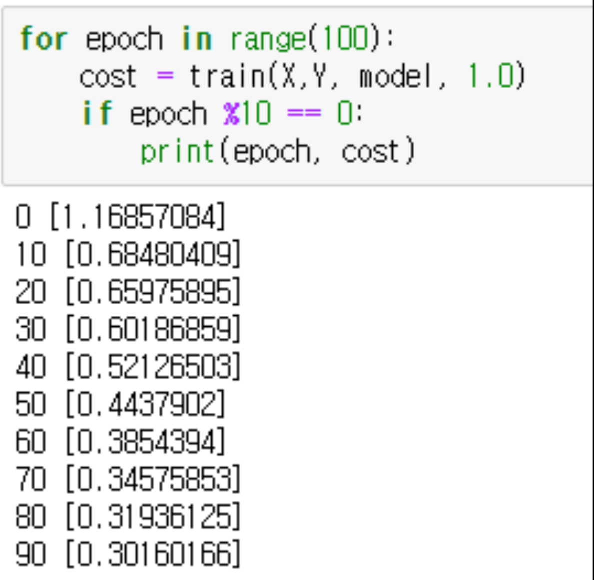
$$\frac{\partial L(a^{[2]},y)}{\partial w_i^{[2]}} = \left(a^{[2]} - y\right)\, a_i^{[1]}$$

$$\frac{\partial L(a^{[2]},y)}{\partial b_i^{[1]}} = \left(a^{[2]} - y\right) w_i^{[2]} \left(1 - {a_i^{[1]}}^2\right)$$

$$\frac{\partial L(a^{[2]},y)}{\partial W_{ij}^{[1]}} = \left(a^{[2]} - y\right) w_i^{[2]} \left(1 - {a_i^{[1]}}^2\right) x_j$$



```
]: def train(X, Y, model, lr = 0.1):
       dW1 = np.zeros_like(model.W1)
       db1 = np.zeros_like(model.b1)
       dW2 = np.zeros_like(model.W2)
       db2 = np.zeros_like(model.b2)
       m = len(X)
       cost = 0.0
       for x,y in zip(X,Y):
           a2, (z1,a1,z2, _) = model.predict(x)
           if y == 1:
               cost -= np.log(a2)
           else:
               cost -= np.log(1-a2)

           diff = a2-y
           # layer 2
           # db2
           db2 += diff

           # dw2 - todo: remove for-loops
           for i in range(model.num_hiddens):
               dW2[i] += a1[i]*diff
           #layer 1
           # db1 - todo: remove for-loops
           for i in range(model.num_hiddens):
               db1[i] += (1-a1[i]**2)*model.W2[i]*diff
           # db2 - todo: remove for-loops
           for i in range(model.num_hiddens):
               for j in range(model.num_input_features):
                   dW1[i,j] += x[j]*(1-a1[i]**2)*model.W2[i]*diff

       cost /= m
       model.W1 -= lr * dW1/m
       model.b1 -= lr * db1/m
       model.W2 -= lr * dW2/m
       model.b2 -= lr * db2/m

       return cost
```



```
for epoch in range(100):
    cost = train(X,Y, model, 1.0)
    if epoch %10 == 0:
        print(epoch, cost)
```

```
0 [1.16857084]
10 [0.68480409]
20 [0.65975895]
30 [0.60186859]
40 [0.52126503]
50 [0.4437902]
60 [0.3854394]
70 [0.34575853]
80 [0.31936125]
90 [0.30160166]
```

Train (with vector operations)

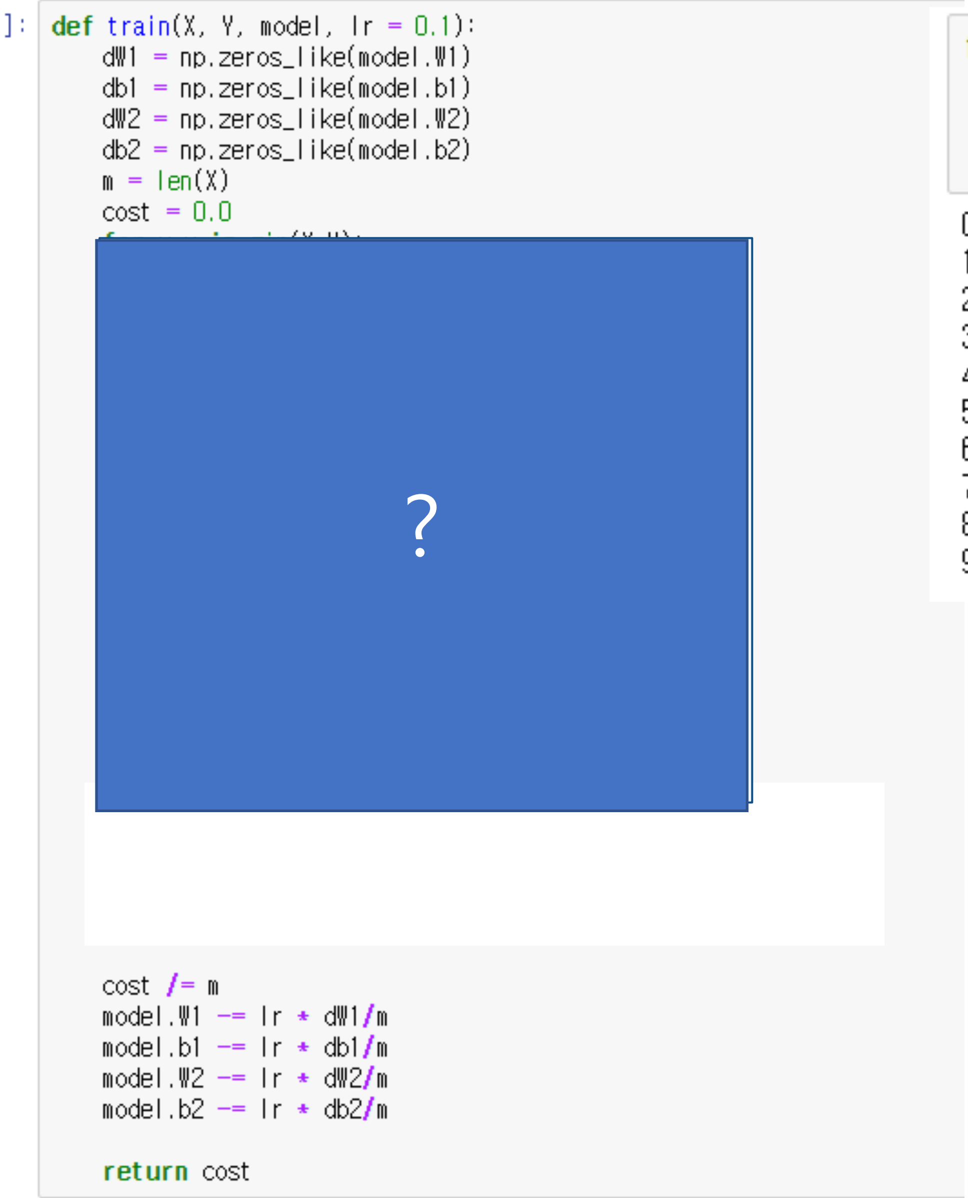
$$\frac{\partial L(a^{[2]}, y)}{\partial b^{[2]}} = a^{[2]} - y$$

$$\nabla_{\boldsymbol{w}^{[2]}} L(a^{[2]}, y) = (a^{[2]} - y)\boldsymbol{a}^{[1]}$$

$$\nabla_{\boldsymbol{b}^{[1]}} L(a^{[2]}, y) = (a^{[2]} - y)\boldsymbol{w}^{[2]} \odot \boldsymbol{e}^{[1]}$$

$$\begin{aligned}\nabla_{\boldsymbol{W}^{[1]}} L(a^{[2]}, y) &= (a^{[2]} - y)(\boldsymbol{w}^{[2]} \odot \boldsymbol{e}^{[1]}) \otimes \boldsymbol{x} \\ &= \nabla_{\boldsymbol{b}^{[1]}} L(a^{[2]}, y) \otimes \boldsymbol{x}\end{aligned}$$

$$\boldsymbol{e}^{[1]} = (1 - \boldsymbol{a}^{[1]} \odot \boldsymbol{a}^{[1]})$$

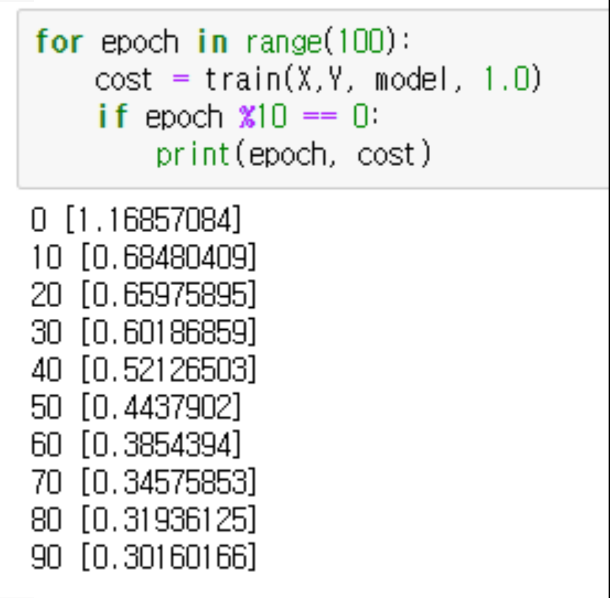


```
]: def train(X, Y, model, lr = 0.1):
       dW1 = np.zeros_like(model.W1)
       db1 = np.zeros_like(model.b1)
       dW2 = np.zeros_like(model.W2)
       db2 = np.zeros_like(model.b2)
       m = len(X)
       cost = 0.0
```

[redacted]

?

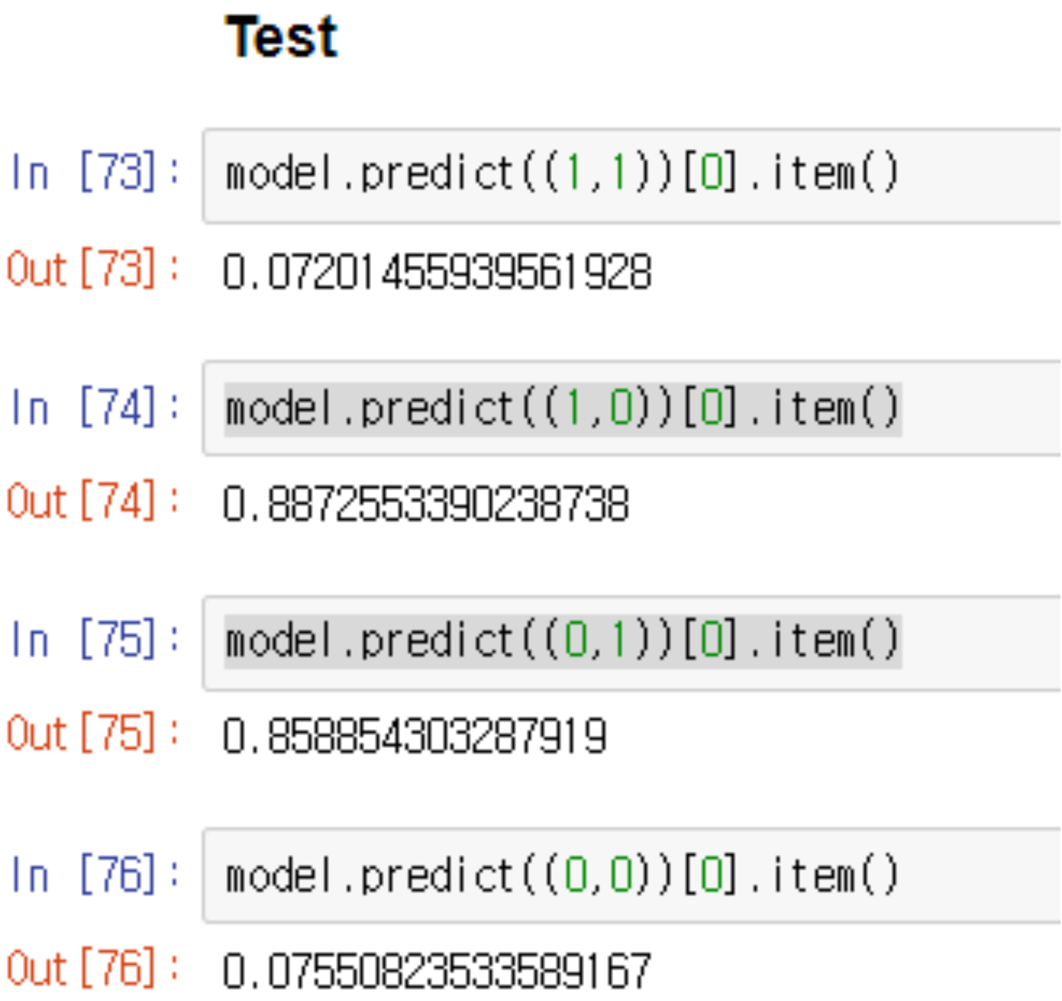
```
       cost /= m
       model.W1 -= lr * dW1/m
       model.b1 -= lr * db1/m
       model.W2 -= lr * dW2/m
       model.b2 -= lr * db2/m

       return cost
```



```
for epoch in range(100):
    cost = train(X,Y, model, 1.0)
    if epoch %10 == 0:
        print(epoch, cost)
```

```
0 [1.16857084]
10 [0.68480409]
20 [0.65975895]
30 [0.60186859]
40 [0.52126503]
50 [0.4437902]
60 [0.3854394]
70 [0.34575853]
80 [0.31936125]
90 [0.30160166]
```

Test

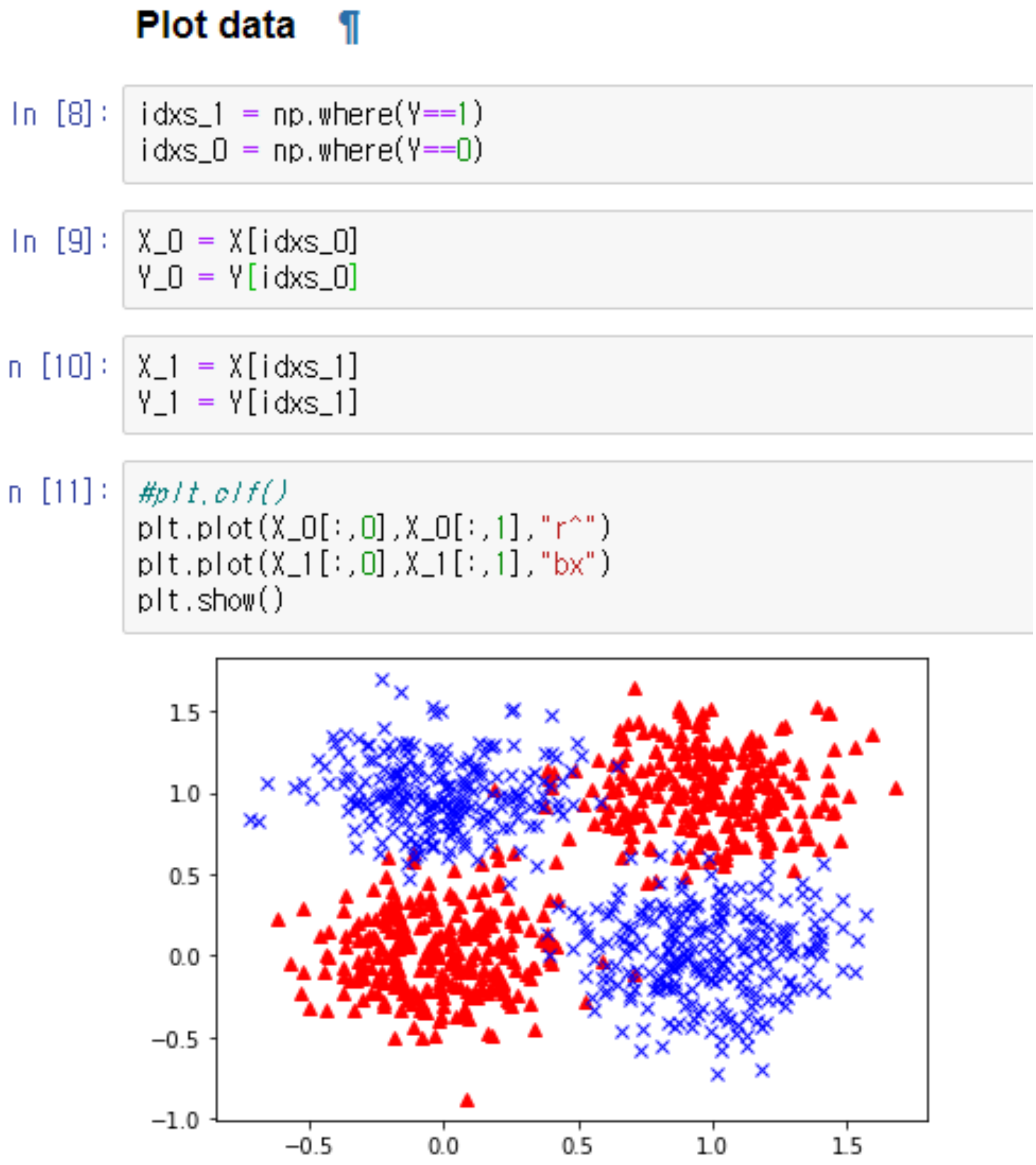


Test

```
In [73]: model.predict((1,1))[0].item()
Out[73]: 0.07201455939561928

In [74]: model.predict((1,0))[0].item()
Out[74]: 0.8872553390238738

In [75]: model.predict((0,1))[0].item()
Out[75]: 0.858854303287919

In [76]: model.predict((0,0))[0].item()
Out[76]: 0.07550823533589167
```

Data plotting

- import matplotlib.pyplot as plt



Plot data ¶

In [8]:

```
idxs_1 = np.where(Y==1)
idxs_0 = np.where(Y==0)
```

In [9]:

```
X_0 = X[idxs_0]
Y_0 = Y[idxs_0]
```

n [10]:

```
X_1 = X[idxs_1]
Y_1 = Y[idxs_1]
```

n [11]:

```
#plt.clf()
plt.plot(X_0[:,0],X_0[:,1],"r^")
plt.plot(X_1[:,0],X_1[:,1],"bx")
plt.show()
```