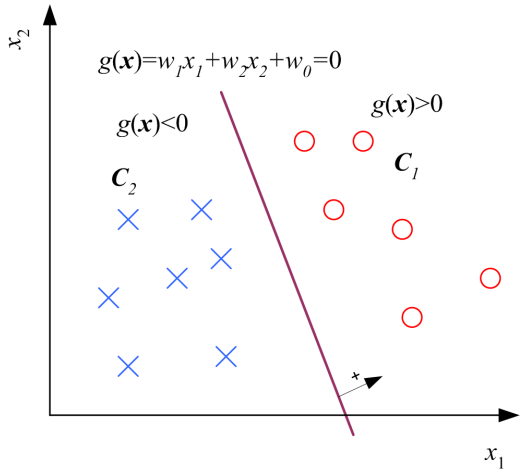
Machine Learning

Lecture 8: Multi-layer Perceptrons

Fall 2023

Instructor: Xiaodong Gu

Recall: Linear Classifiers

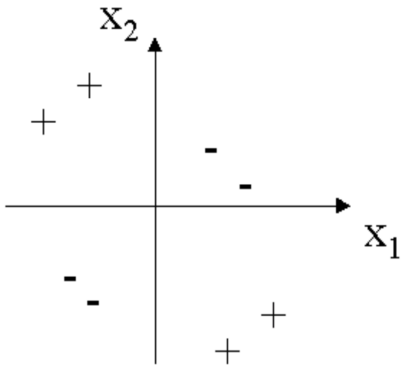


Perceptron

Logistic Regression

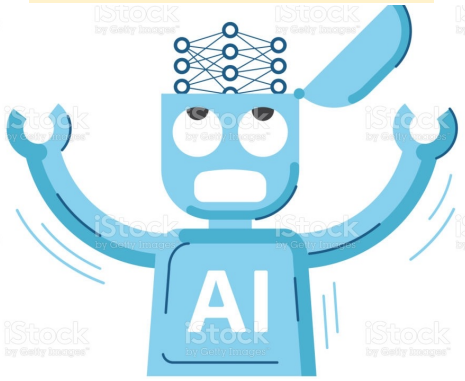
SVM

The curse of nonlinearity



I can approximate **any** non-linear functions !

Neural Networks

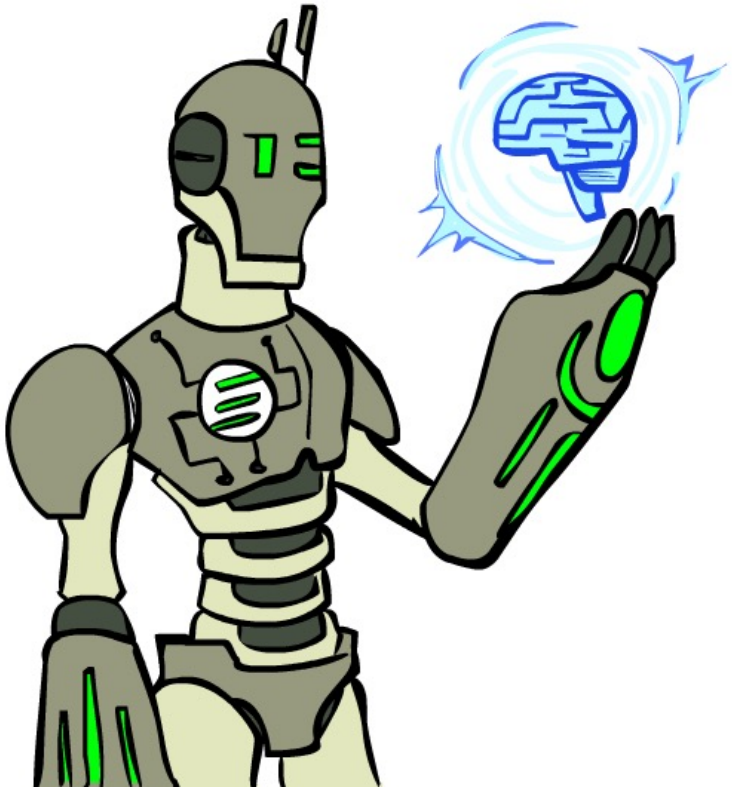


Today

Artificial Neural Networks

Multi-layer Perceptrons

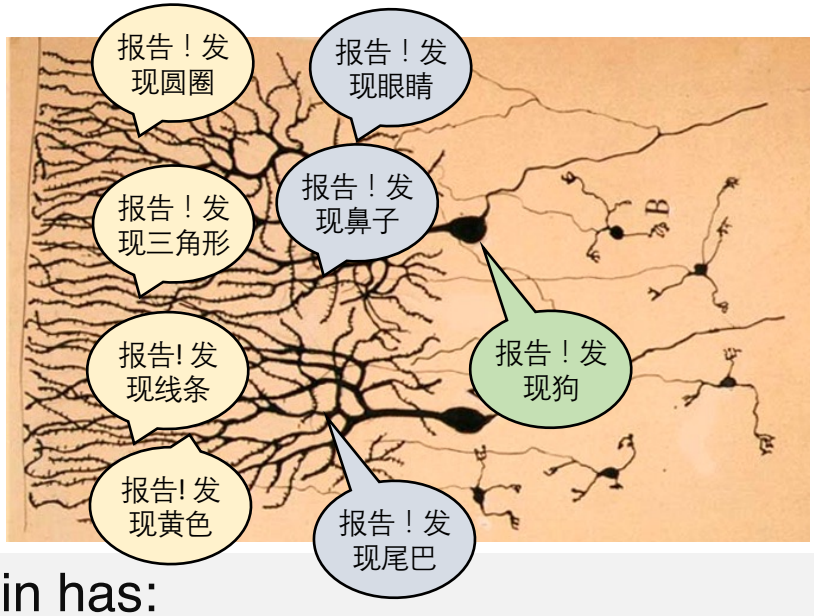
Backpropagation



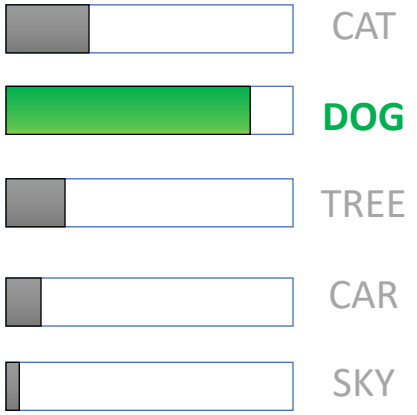
Artificial Neural Networks

Inspired by actual human brain

Input

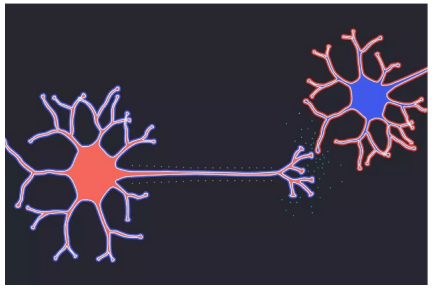


Output



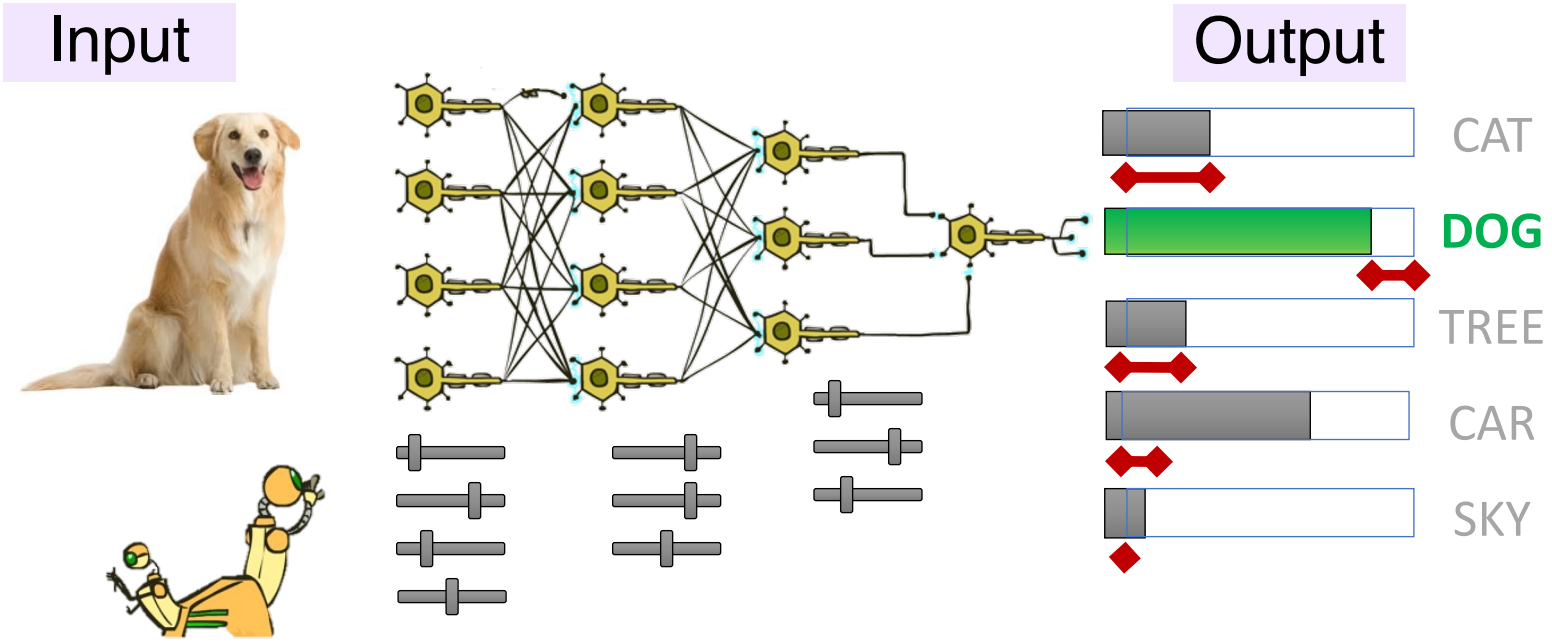
A human brain has:

- Large number (10^{11}) of neurons as processing units
- Large number (10^{4}) of synapses per neuron as memory units
- Parallel processing capabilities
- Distributed computation/memory
- High robustness to noise and failure



Artificial Neural Networks

- **Artificial Neural Networks (ANN)** mimic some characteristics of the human brain, especially with regard to the computational aspects.

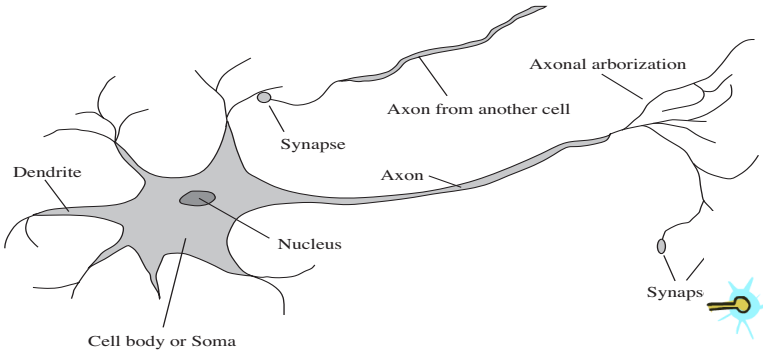


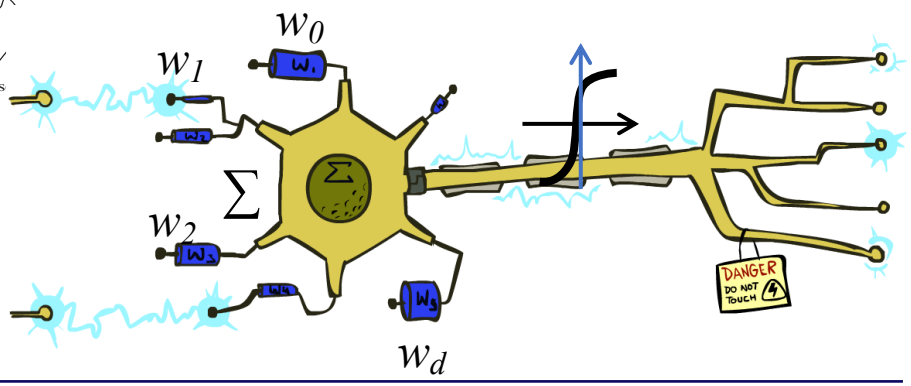
Artificial Neural Network

Neuroscientists vs. Computer scientists

- **Cognitive scientists & neuroscientists**
  Aim to understand the functioning of the brain by building models of the natural neural networks.

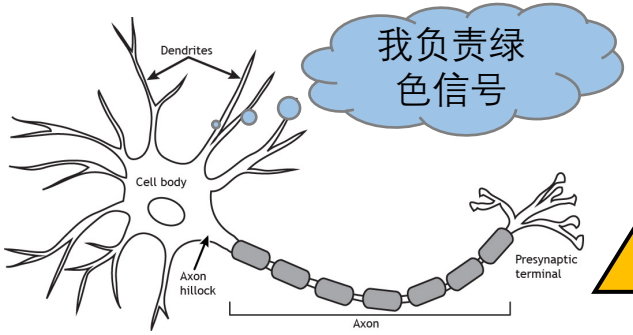
- **Computer scientists**
  Aim to build better computer systems based on inspiration from studying the brain.





Perceptron (感知机) [Frank Rosenblatt, 1957]

- Basic modeling of the "**Neuron**".



我负责绿色信号

报告大脑,发现目标!

1	0	9	0	0
3	2	5	0	2
5	5	0	8	0
0	4	6	0	1
0	0	3	0	0

x_1 x_2 x_3 x_4 w_0 w_1 w_2 w_d

$a = \sum w_j x_j$ $y = \sigma(a)$

z σ y

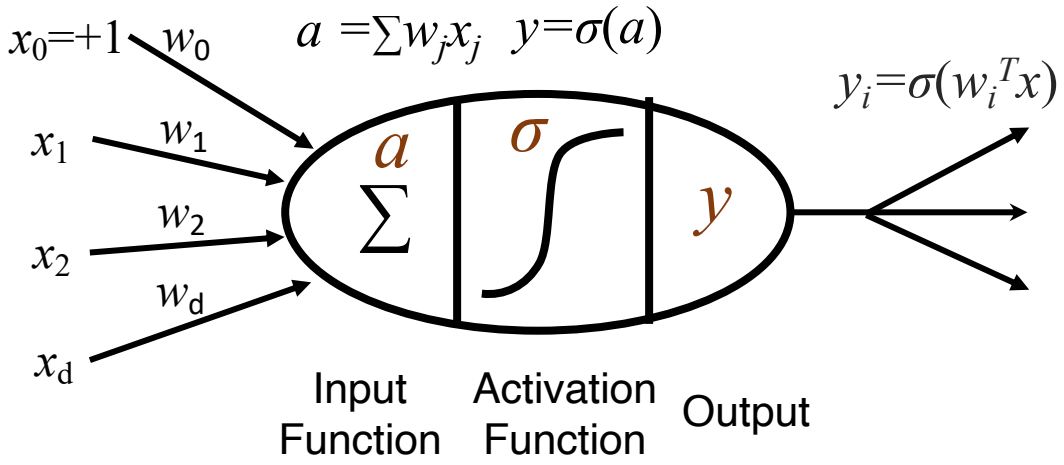
输入函数 激活函数 输出

$y = \begin{cases} 0 & \text{没有绿色} \\ 1 & \text{发现绿色} \end{cases}$

通过计算我也能识别绿色

Perceptron

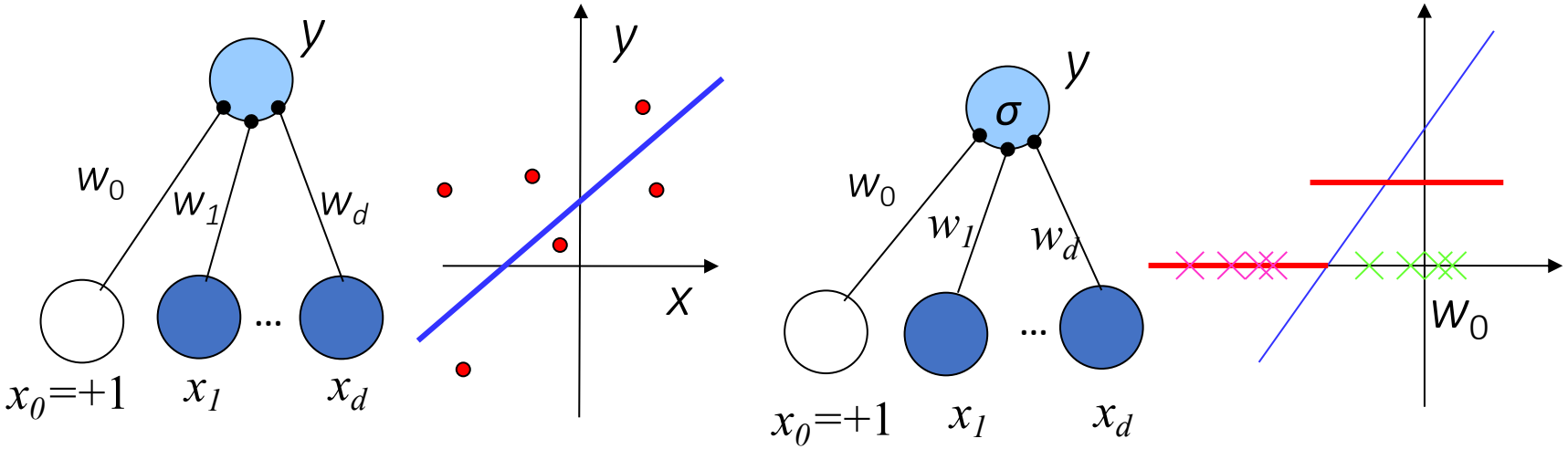
[Frank Rosenblatt, 1957]



- The output y is an activation of a linear weighted sum of the inputs $x = (x_0,\ldots,x_d)^T$ where x_0=1 is a special bias unit and $\boldsymbol{w}=(w_0,\ldots,w_d)^T$ are the connection (synaptic) weights.

What Can a Perceptron Be Used for?

- Regression: $y=\boldsymbol{w}x+w_0$
- Classification: $y=\text{sign}(\boldsymbol{w}x+w_0)$



- For classification, we need a threshold function:

$$\sigma(a)=\begin{cases}1 & \text{if } a>0\\ -1 & \text{otherwise}\end{cases}$$

which defines the decision rule: Choose $\begin{cases}C_1 & \text{if } \sigma(\boldsymbol{w}^{\mathrm{T}}x)=1\\ C_2 & \text{otherwise}\end{cases}$

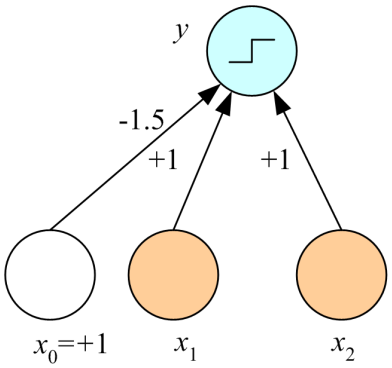
Perceptron and Boolean Functions

- Learning a Boolean function is equivalent to a two-class classification problem.

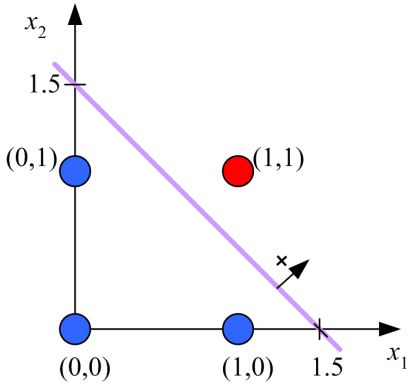
Boolean function

x_1	x_2	AND
0	0	0
0	1	0
1	0	0
1	1	1

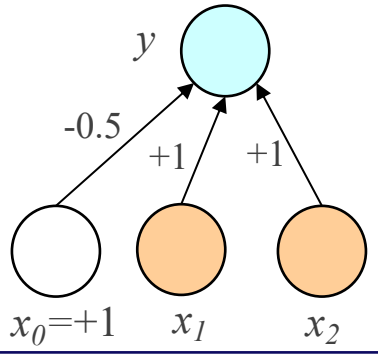
Perceptron

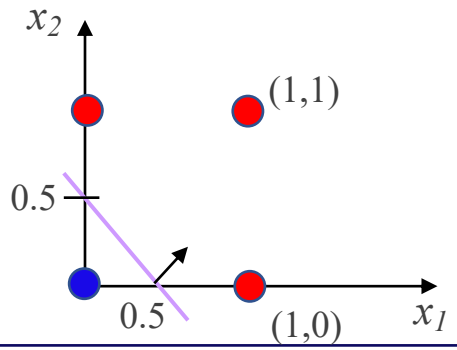


Geometric Interpretation



x_1	x_2	OR
0	0	0
0	1	1
1	0	1
1	1	1

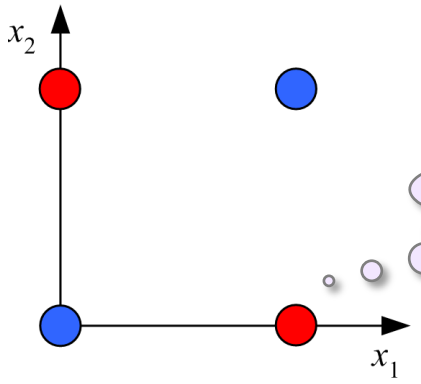


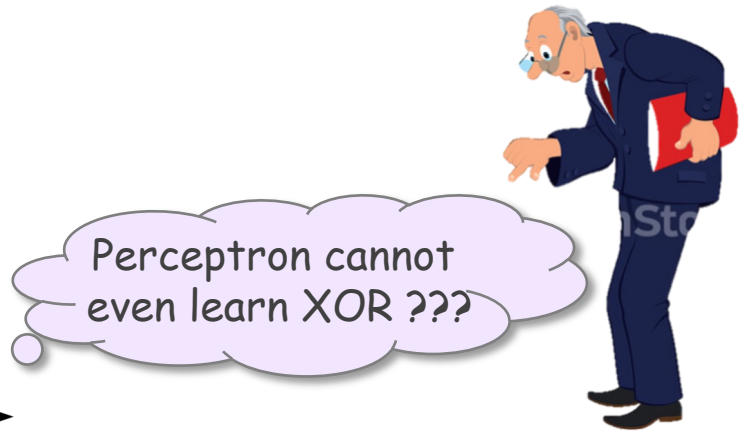


The XOR Puzzle and the Dark Ages of NN

- **However,** Minsky and Papert [1969] proved that some rather elementary computations, such as the XOR problem, could not be done by Rosenblatt's one-layer perceptron.

x_1	x_2	XOR
0	0	0
0	1	1
1	0	1
1	1	0

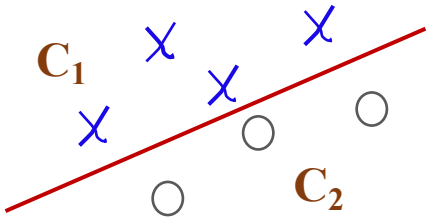


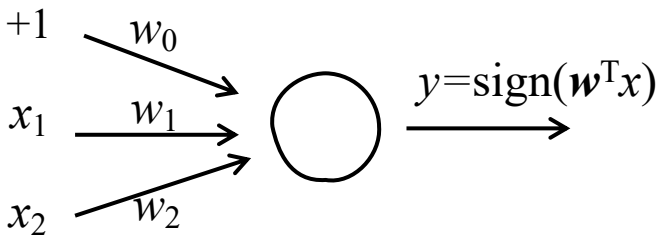


Limitation1: perceptron cannot learn data that are not linearly separable

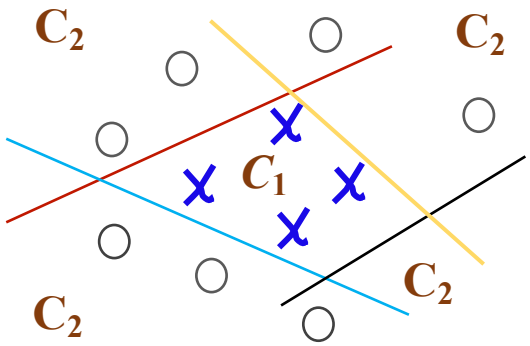
The Magic of Hidden Layers !

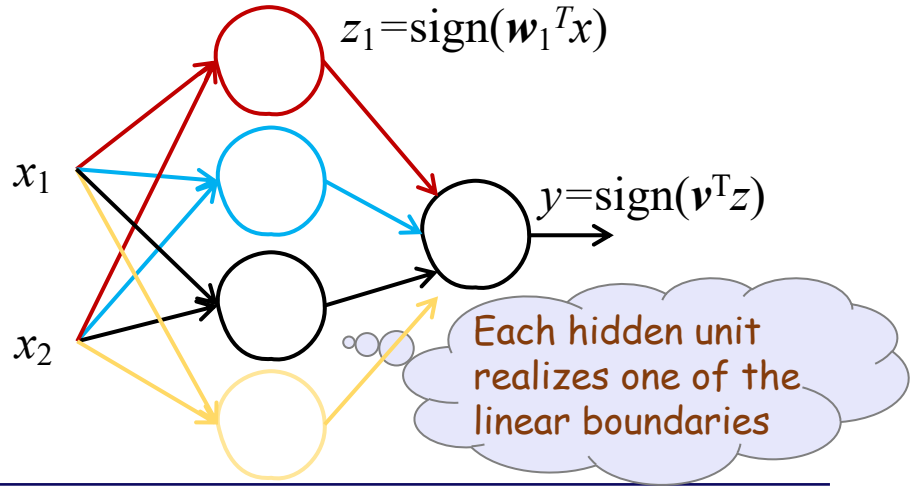
- **But,** adding hidden layer(s) (internal presentation) allows for learning a mapping that is not constrained by linear separability.





decision boundary: $x_1w_1+x_2w_2+w_0=0$



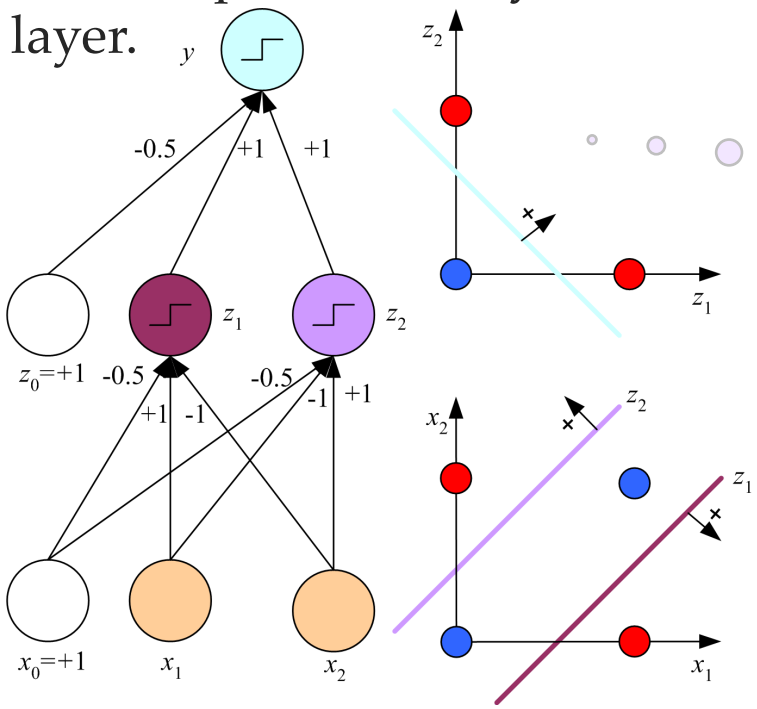


The Magic of Hidden Layers !

- Any Boolean function can be represented as a disjunction of conjunctions, e.g.

$$x_1 \text{ XOR } x_2 = (x_1 \text{ AND } \sim x_2) \text{ or } (\sim x_1 \text{ AND } x_2)$$

which can be implemented by a multi-layer perceptron with one hidden layer.

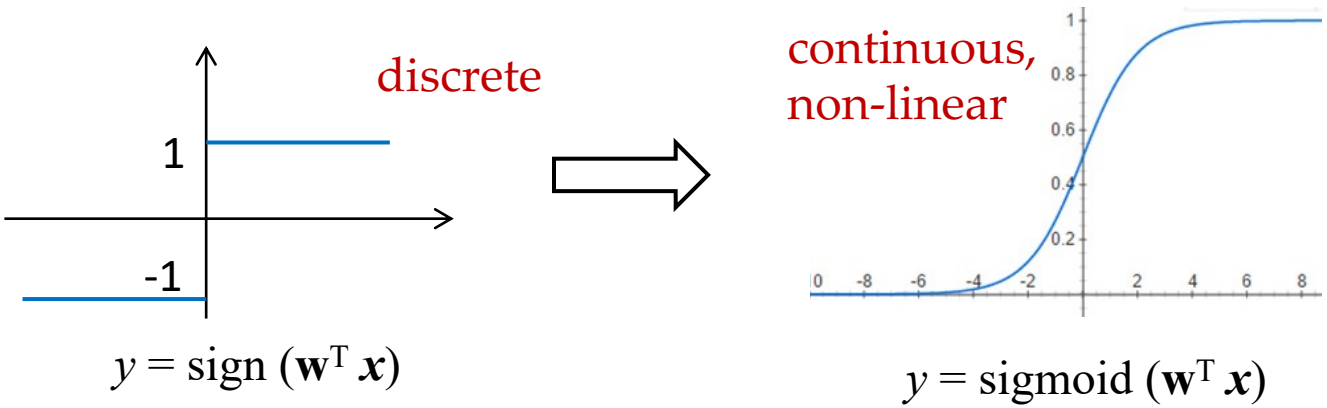


Any other problems?

non-differentiable thresholding functions !

Continuous Thresholding Function

- Use **sigmoid** instead of discrete thresholding:



$y = \text{sign}(\mathbf{w}^T \boldsymbol{x})$ $\qquad$ $y = \text{sigmoid}(\mathbf{w}^T \boldsymbol{x})$

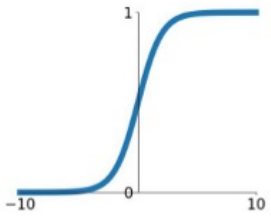
- Sigmoid can be seen as a continuous, differentiable version of thresholding.
- The output may be interpreted as the posterior probability that the input x belongs to C_1.

Activation Functions

- Generally, we can apply many other continuous thresholding functions, called **activation functions**, to bring nonlinearity into perceptrons.

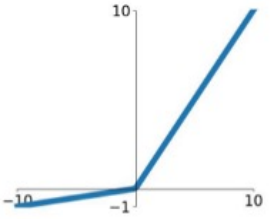
Sigmoid

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



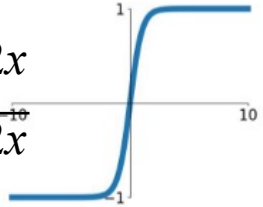
Leaky ReLU

$$\max(0.1x, x)$$



tanh

$$\tanh(x) = \frac{1-e^{-2x}}{1+e^{-2x}}$$

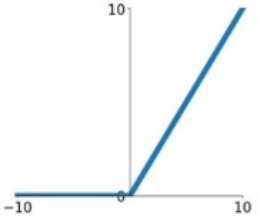


Maxout

$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

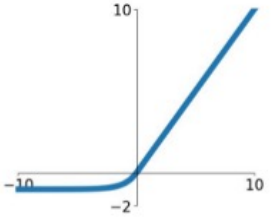
ReLU

$$\max(0, x)$$



ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



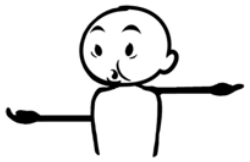
Sigmoid

$$y = \frac{1}{1+e^{-x}}$$

Tanh

$$y = \tanh(x)$$

Step Function



$$y = \begin{cases} 0, & x<n \\ 1, & x \geq n \end{cases}$$

Softplus

$$y = \ln(1+e^{x})$$

ReLU

$$y = \begin{cases} 0, & x<0 \\ x, & x \geq 0 \end{cases}$$

Softsign

$$y = \frac{x}{(1+|x|)}$$

ELU

$$y = \begin{cases} \alpha(e^{x}-1), & x<0 \\ x, & x \geq 0 \end{cases}$$

Log of Sigmoid

$$y = \ln\left(\frac{1}{1+e^{-x}}\right)$$

Swish

$$y = \frac{x}{1+e^{-x}}$$

Sinc

$$y = \frac{\sin(x)}{x}$$

Leaky ReLU

$$y = \max(0.1x, x)$$

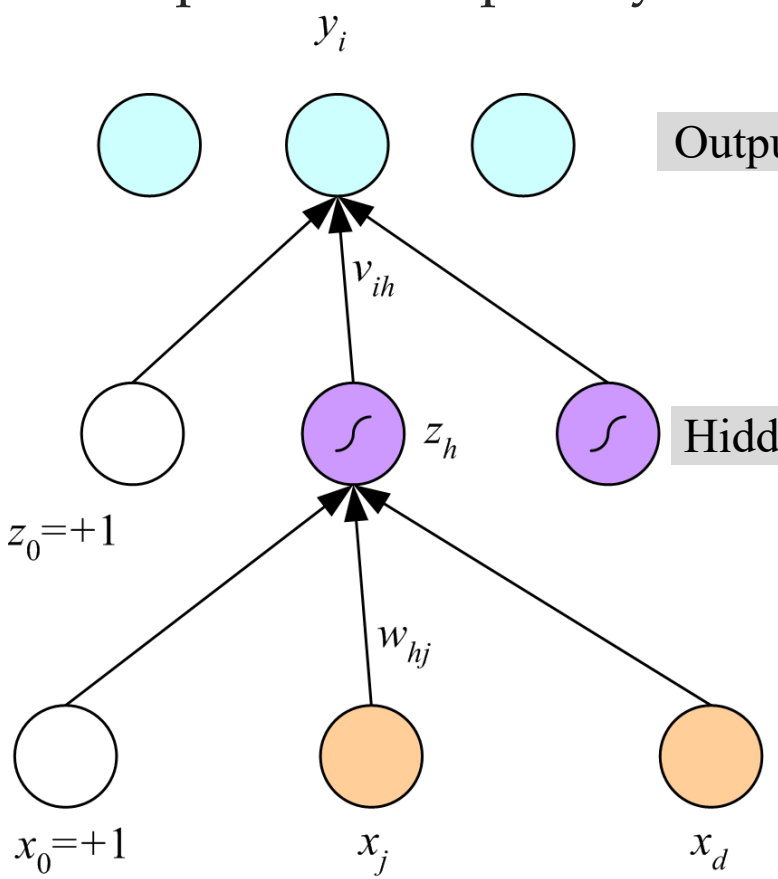
Mish

$$y = x(\tanh(\text{softplus}(x)))$$

https://sefiks.com/2020/02/02/dance-moves-of-deep-learning-activation-functions/

Multilayer Perceptrons (多层感知机)

- A **multilayer perceptron** (**MLP**) has a hidden layer between the input and output layers.



Output

a linear function in the new H-dim space.

$$y_i = \mathbf{v}_i^T \mathbf{z} = \sum_{h=1}^{H} v_{ih} z_h + v_{i0}$$

Hidden

a **nonlinear** transformation from the d-dim input space to the H-dim space spanned by the hidden units.

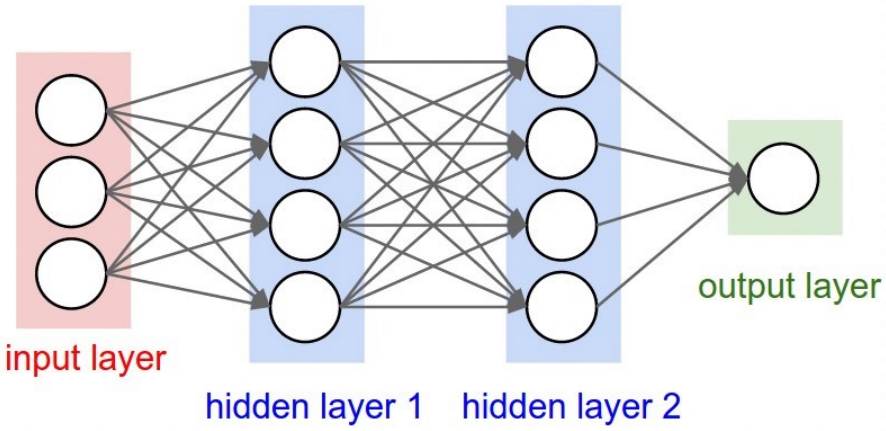
$$z_h = \text{sigmoid}(\mathbf{w}_h^T \mathbf{x}) = \frac{1}{1 + \exp\left[-\left(\sum_{j=1}^{d} w_{hj} x_j + w_{h0}\right)\right]}$$

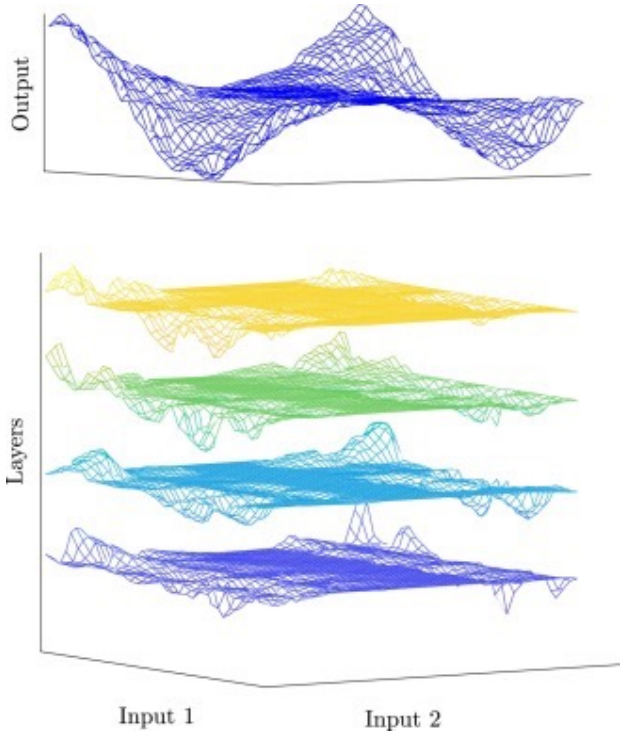
Input

no computation.

What Does an MLP Do?

- MLP is a universal approximator

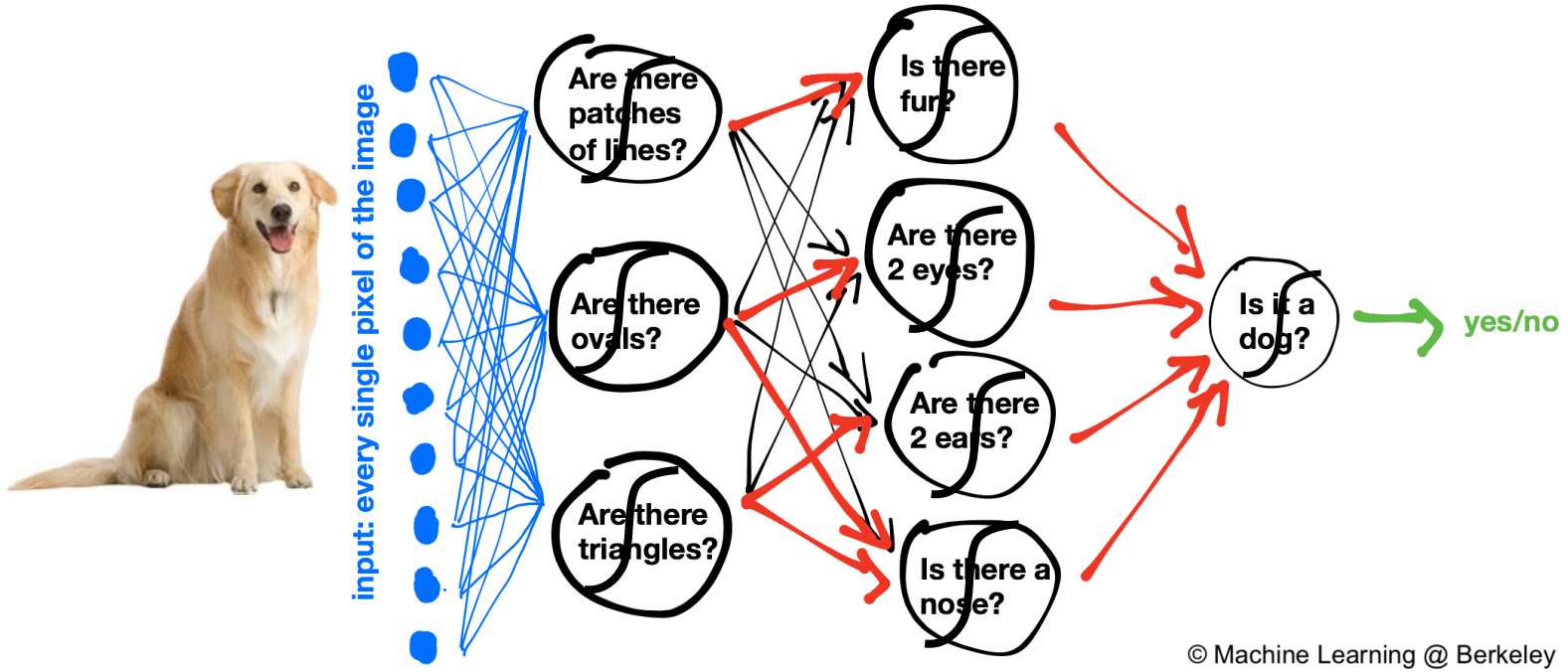




An MLP with one hidden layer can approximate **any** continuous functions of the input given sufficiently many hidden units.

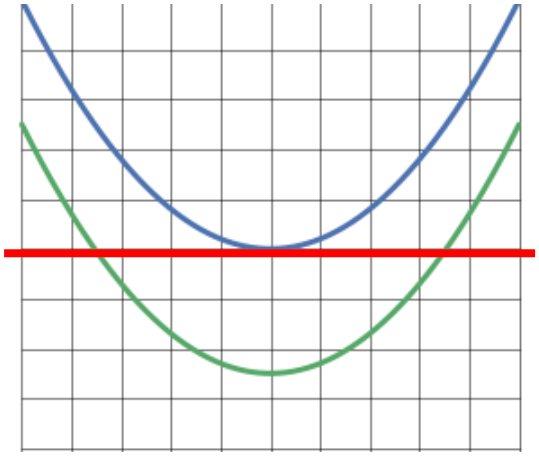
What Does an MLP Do?

Explanation 1: MLP = continuous (differentiable) decisions that are easy to optimize.



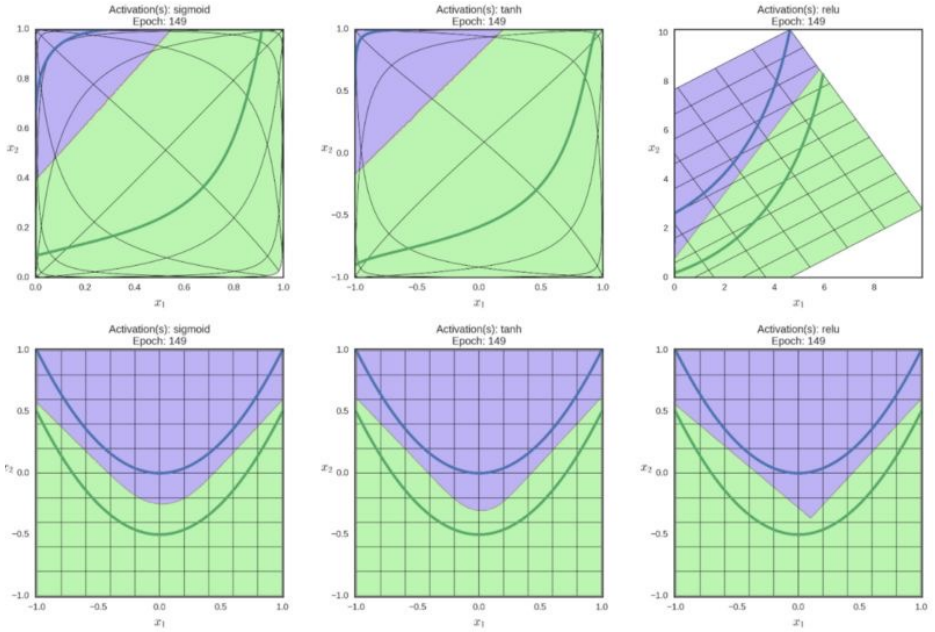
What Does an MLP do?

Explanation 2: MLP transform a nonlinear feature space by scaling, squeezing, and deformation. Find linear boundaries in the new space.



$$y = \begin{cases} 0, & (x_1, x_2) \in (x, x^2) \\ 1, & (x_1, x_2) \in (x, x^2 - 0.5) \end{cases}$$

Linear classifier cannot separate the two classes



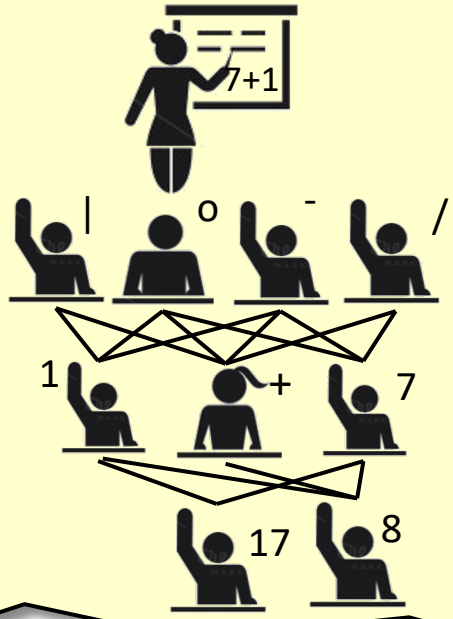
Linearly separable by squeezing and deformations,...

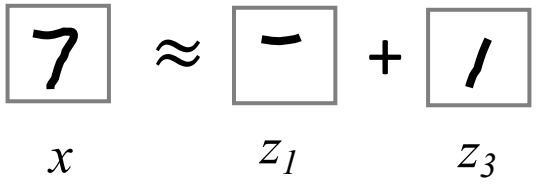
https://www.zhihu.com/question/22334626

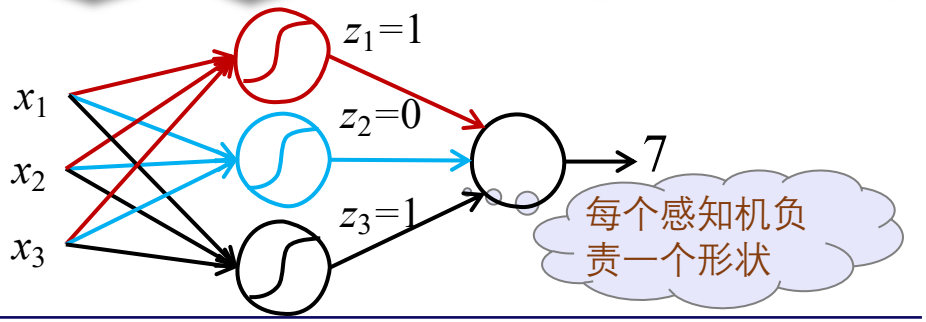
Game Time – 模拟神经网络识别算术

老师演示数术给第一排学生，每个学生模拟一个神经元(感知机)，负责感知一个基本形状，发现目标就激活(举手)，后一层的神经元根据前一层激活的是哪几个神经元判断是否数字1、+、7并向后激活。

目标: 最后一位学生猜对老师给的算术







How to train an MLP?

Optimizing MLP

Network Parameters: $\theta=\{w_0,w_1,\ldots,w_n\}$

$$\theta^0 \rightarrow \theta^1 \rightarrow \theta^2 \rightarrow \ldots$$

$$\theta^1 = \theta^0 - \eta\nabla \mathrm{L}\left(\theta^0\right)$$

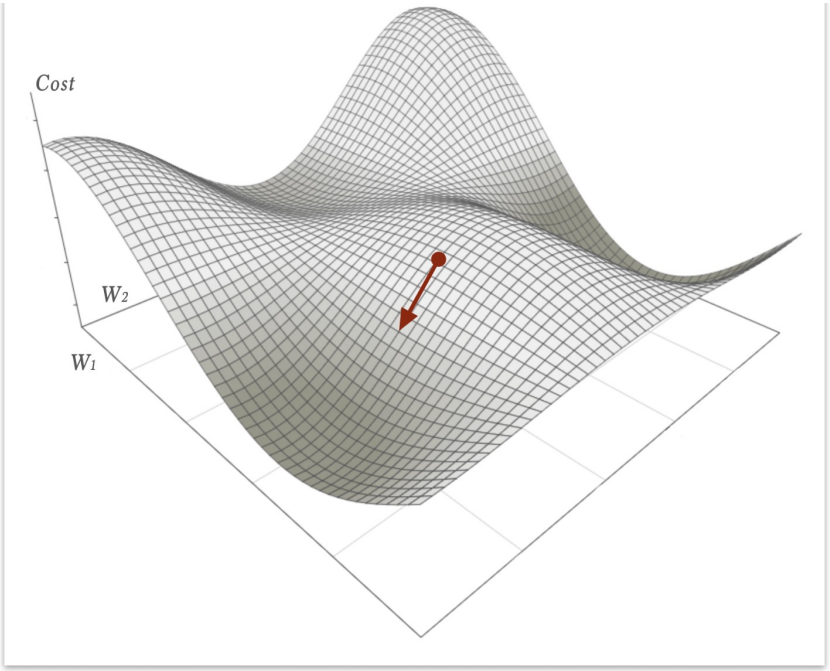
$$\theta^2 = \theta^1 - \eta\nabla \mathrm{L}\left(\theta^1\right)$$

$$\vdots$$

$$\nabla \mathrm{L}(\theta)$$

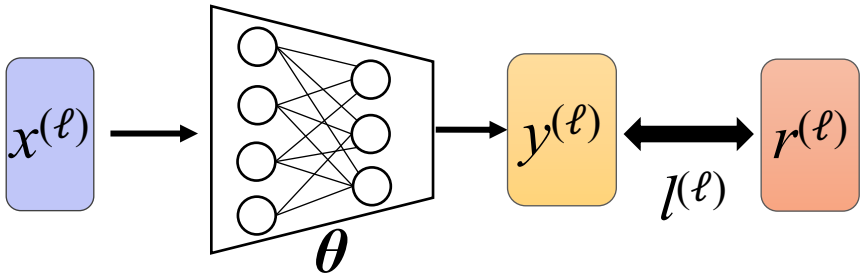
$$\begin{bmatrix} \partial \mathrm{L}(\theta)/\partial w_1 \\ \partial \mathrm{L}(\theta)/\partial w_2 \\ \vdots \end{bmatrix}$$

Millions of Parameters



how to obtain gradients in NN?

Loss Function (Regression)

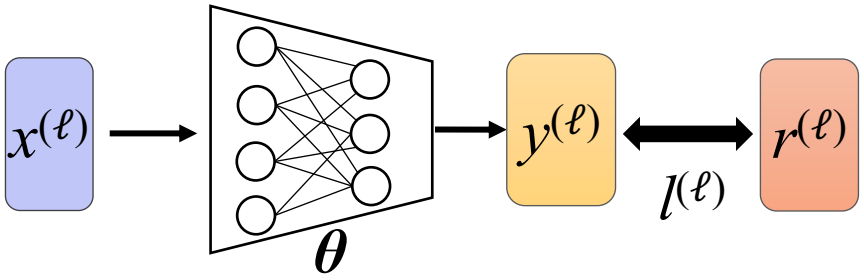


- Given a dataset $D = \{(x^{(1)}, r^{(1)}), \ldots, (x^{(N)}, r^{(N)})\}$ where $x^{(\ell)}$ and $r^{(\ell)}$ represent the ℓ-th input and the corresponding target, respectively.
- For each $x^{(\ell)}$, the model predicts a value $y^{(\ell)}$.
- Our goal is to minimize the **MSE** loss:

$$L(\theta|D) = \frac{1}{2}\sum_{\ell}\sum_{i}(r_i^{(\ell)} - y_i^{(\ell)})^2$$

Loss Function

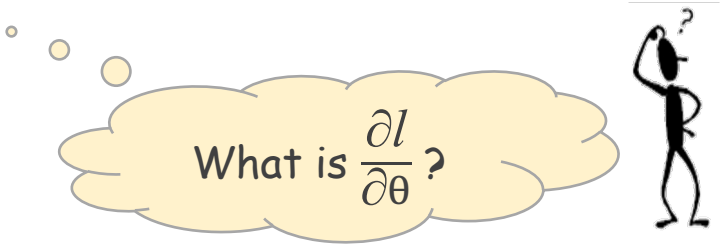
- Consider the loss function for a single sample

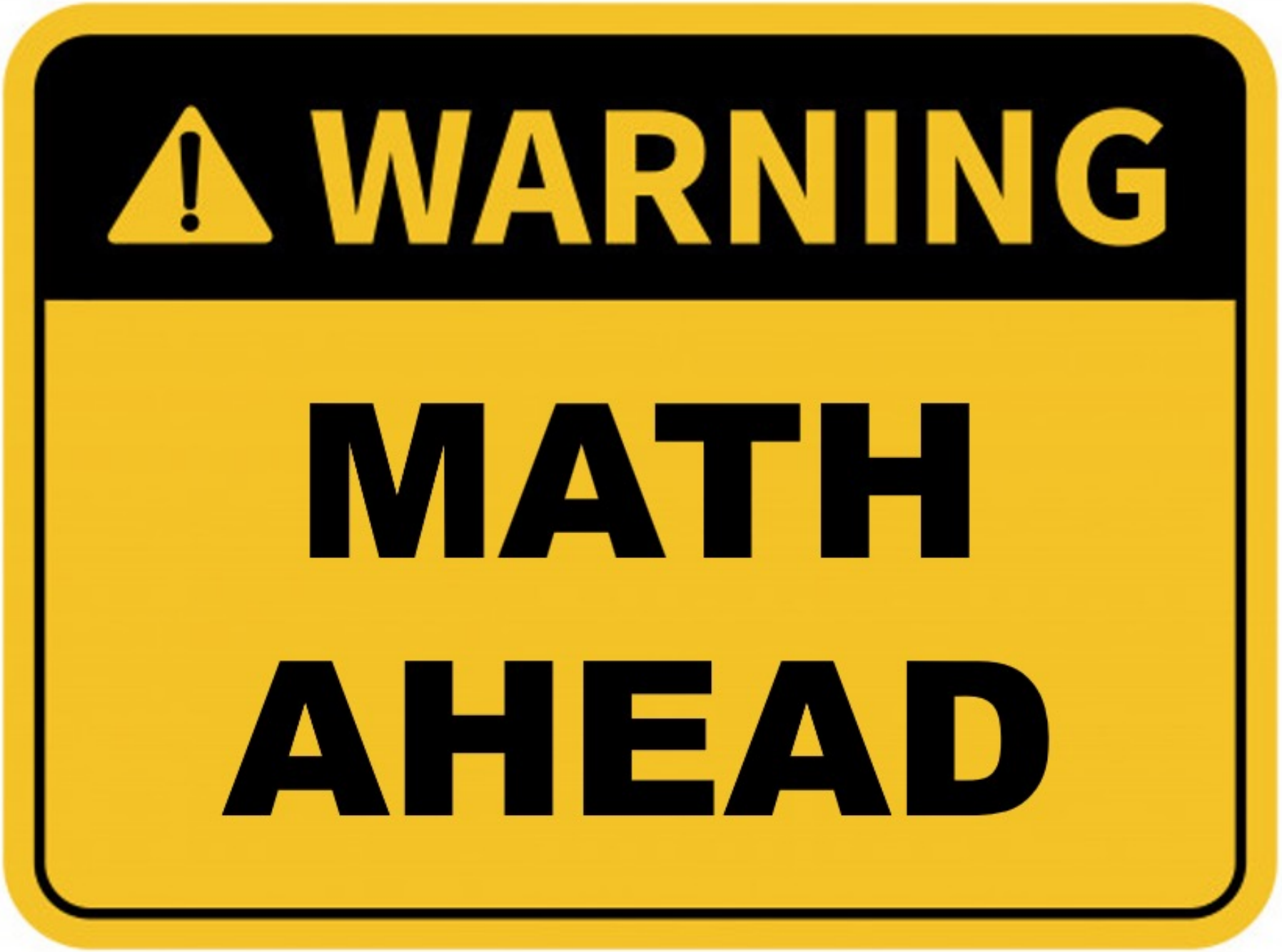


$$L(\theta)=\sum_{\ell=1}^{N} l^{(\ell)}(\theta) \quad \Rightarrow \quad \frac{\partial L(\theta)}{\partial w}=\sum_{\ell=1}^{N}\frac{\partial l^{(\ell)}(\theta)}{\partial w}$$

- For a single training example (x, r):

$$l(\theta|x, r)=\frac{1}{2}\sum_{i}(r_i-y_i)^2$$




WARNING
MATH
AHEAD

Recall: Chain Rule

Case 1

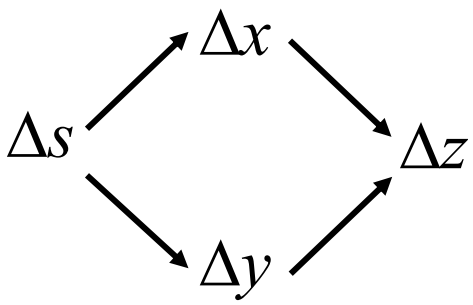
$$y = g(x) \quad z = h(y)$$

$$\Delta x \to \Delta y \to \Delta z$$

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

Case 2

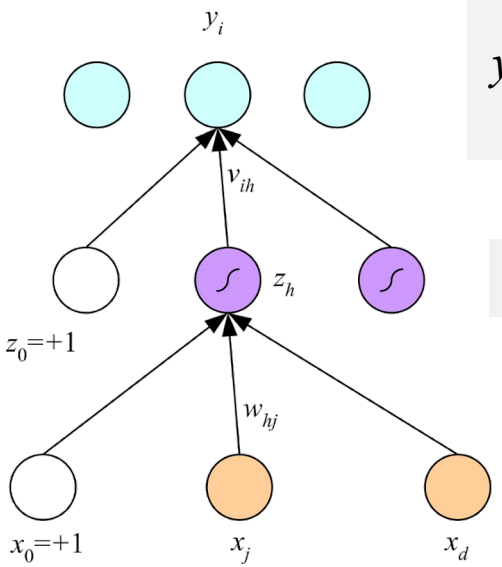
$$x = g(s) \qquad y = h(s) \qquad z = k(x, y)$$



$$\frac{dz}{ds} = \frac{\partial z}{\partial x}\frac{dx}{ds} + \frac{\partial z}{\partial y}\frac{dy}{ds}$$

Gradient Descend for 2-Layer MLP

Regression



forward

$$y_i = \sum_{h=1}^{H} v_{ih} z_h + v i_0$$

$$z_h = \text{sigmoid}(a_h)$$

$$a_h = \sum_{j=1}^{d} W_{hj} x_j$$

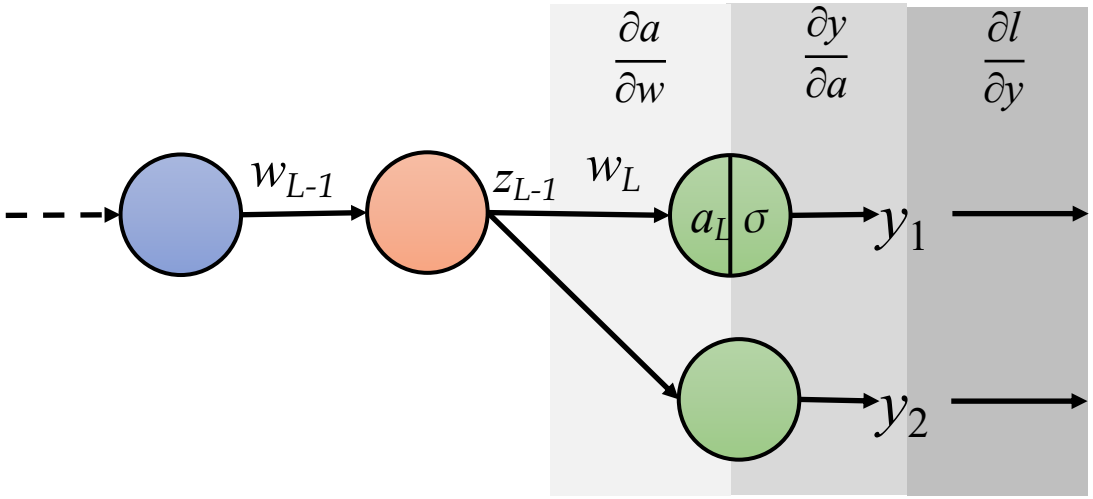
$$l(\boldsymbol{w}, \boldsymbol{v}) = \frac{1}{2} \sum_i (r_i - y_i)^2$$

backward

$$\frac{\partial l}{\partial v_{ih}} = \frac{\partial l}{\partial y_i} \frac{\partial y_i}{\partial v_{ih}} = (r_i - y_i) z_h$$

$$\frac{\partial l}{\partial w_{hj}} = \sum_i \frac{\partial l}{\partial y_i} \frac{\partial y_i}{\partial w_{hj}} = \sum_i \frac{\partial L}{\partial y_i} \frac{\partial y_i}{\partial z_h} \frac{\partial z_h}{\partial a_h} \frac{\partial a_h}{\partial w_{hj}} = \left[\sum_i (ri - yi) v_{ih} \right] z_h (1 - z_h) x_j$$

Gradient Descend for L>2 Layers?



$$\frac{\partial l}{\partial w_L} = \frac{\partial a_L}{\partial w_L}\frac{\partial y_1}{\partial a_L}\frac{\partial l}{\partial y_1}$$

$$\frac{\partial l}{\partial wl_{-1}} = ?$$

$\vdots$

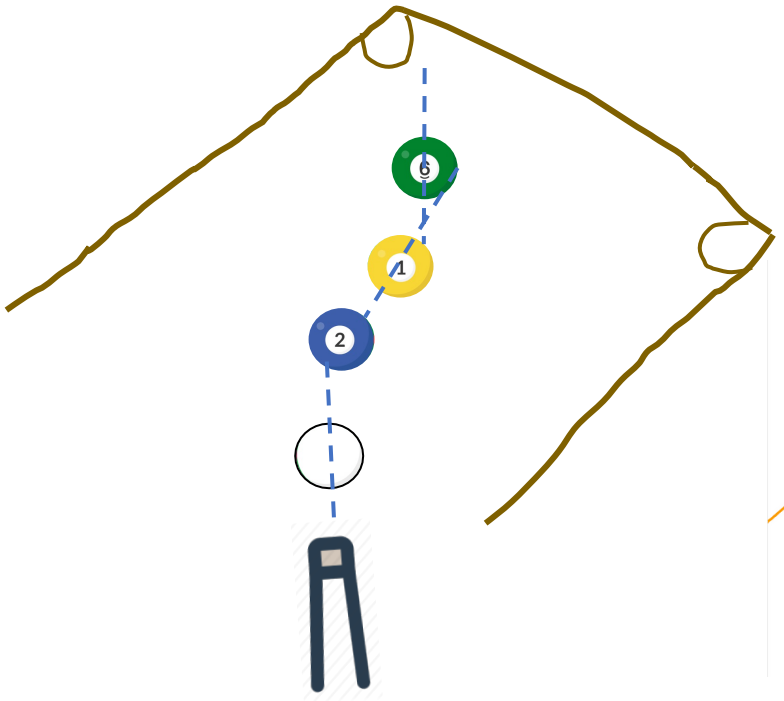
Problems of deriving $\nabla_w L$ directly:

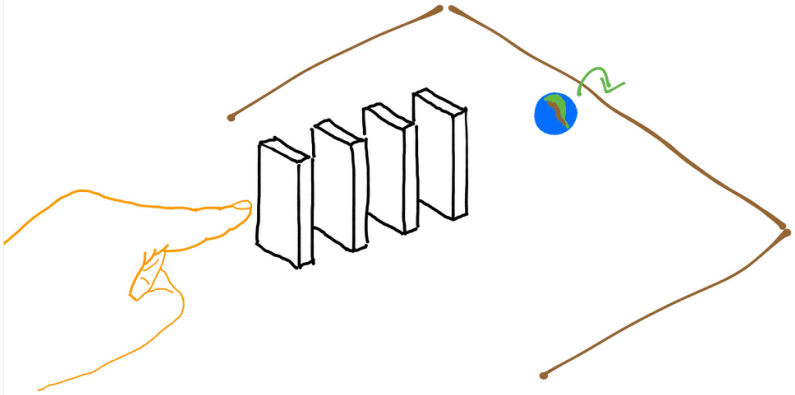
- Very tedious: lots of matrix calculus, need lots of paper …
- What if we want to change loss? E.g. use SVM instead of softmax? Need to re-derive from scratch.
- Not feasible for very complex models!

Straightforward derivation of gradients in MLP is tedious, inflexible and often infeasible, due to the dependences between gradients

Any New Ideas ?

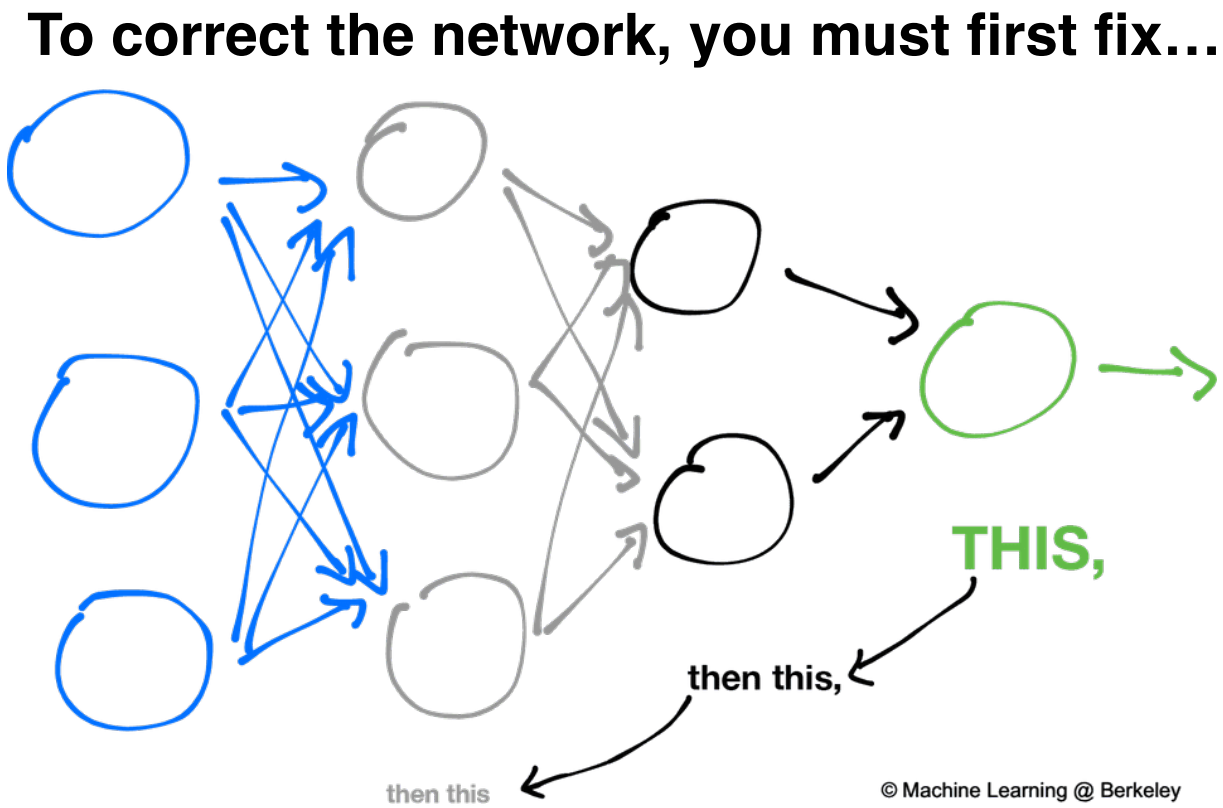
- Straightforward computation of gradients in MLP is **computationally infeasible**, due to the **dependences** between gradients.





Backpropagation

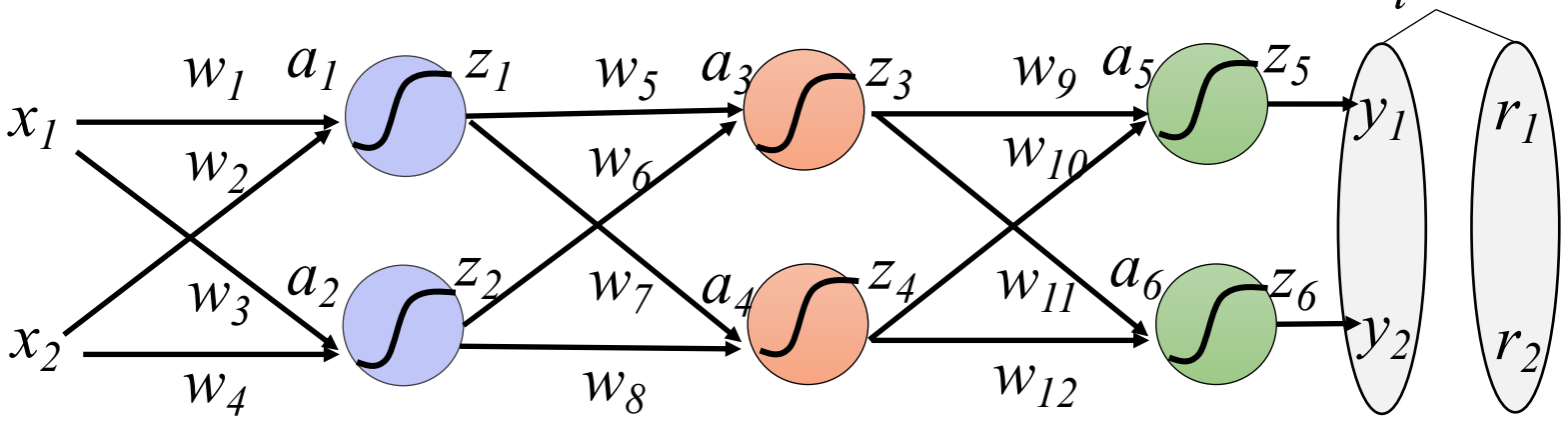
- A **computationally efficient** approach for computing derivatives



Backpropagation

- Consider a simple, general MLP

$$l=\frac{1}{2}\sum_i (r_i-y_i)^2$$

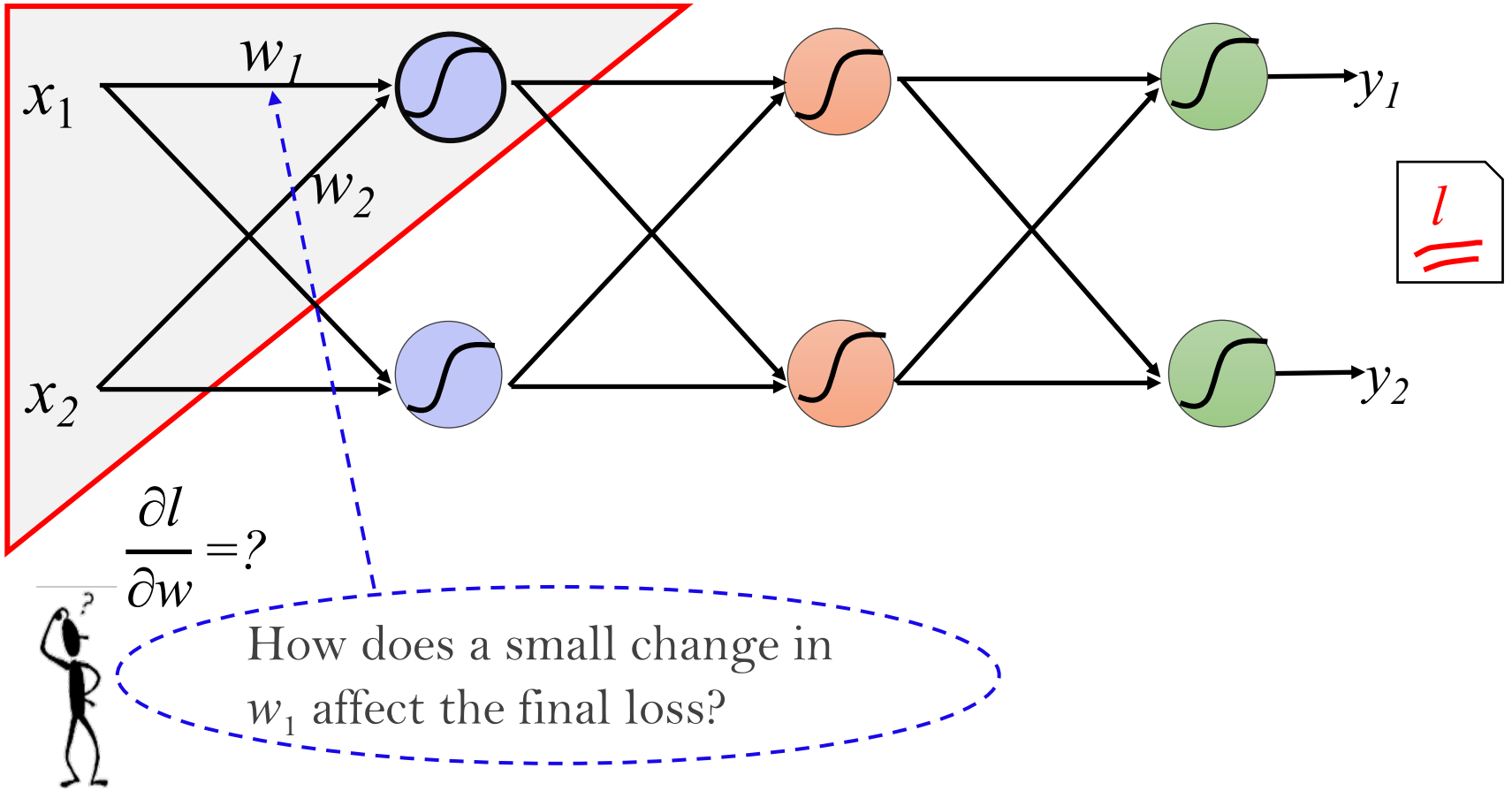


$$\frac{\partial l}{\partial w}=?$$

How does a small change in each weight (e.g., w_i) affect the final loss?

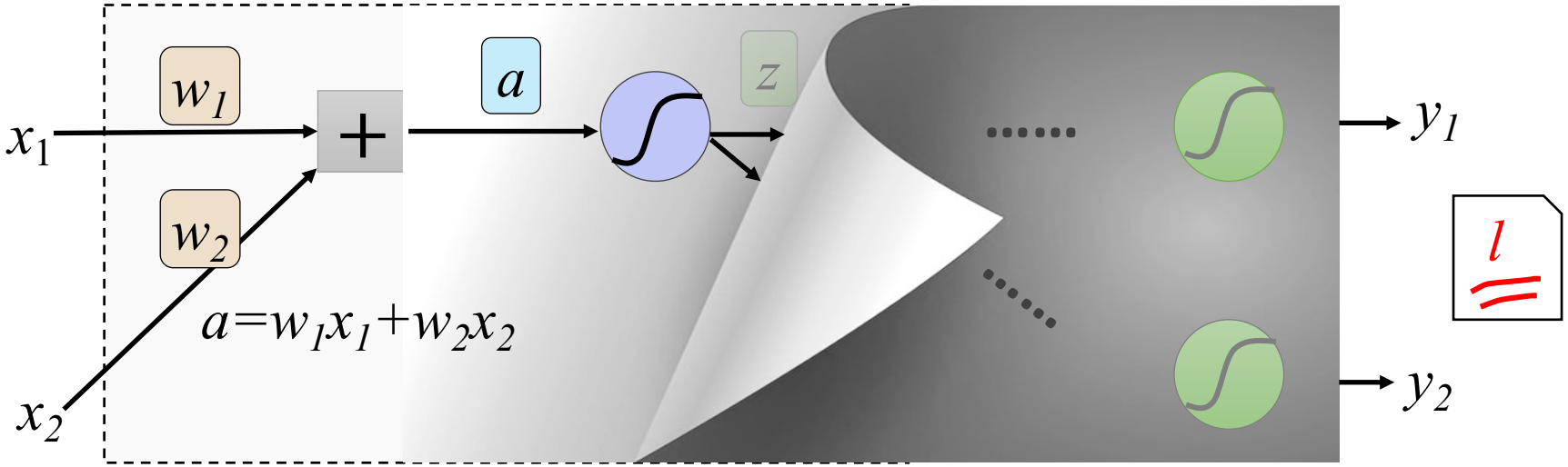
Backpropagation

Initially, consider the first neuron



Backpropagation

Compute $\partial l/\partial w$ for each weight w.



(chain rule)

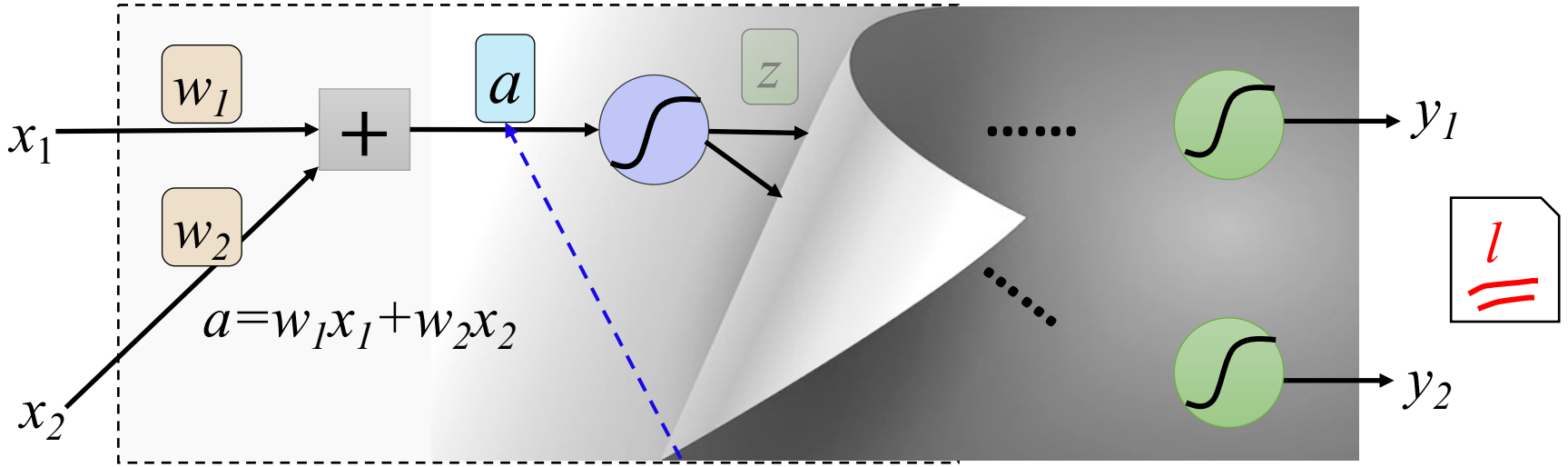
$$\frac{\partial l}{\partial w} = ? \quad \frac{\partial a}{\partial w} \cdot \frac{\partial l}{\partial a}$$

easy

$$\partial a/\partial w_1 = x_1$$
$$\partial a/\partial w_2 = x_2$$

Backpropagation

- Compute $\partial l/\partial w$ for each weight w.



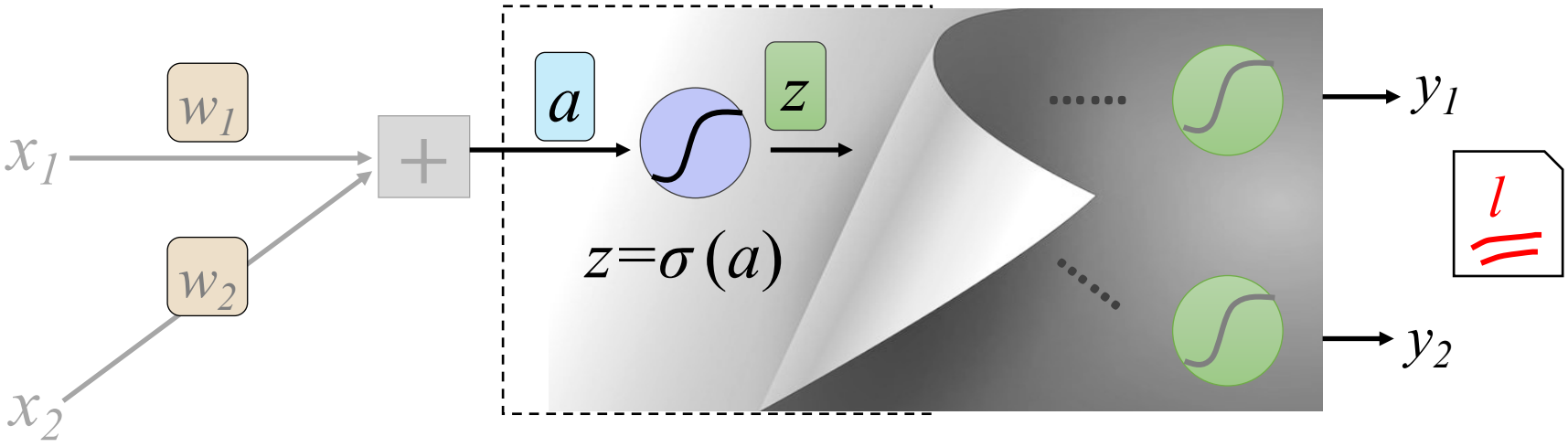
$$\frac{\partial l}{\partial w} = x \cdot \frac{\partial l}{\partial a}$$

?

How does a small change in a affect the final loss?

Backpropagation – Backward Pass

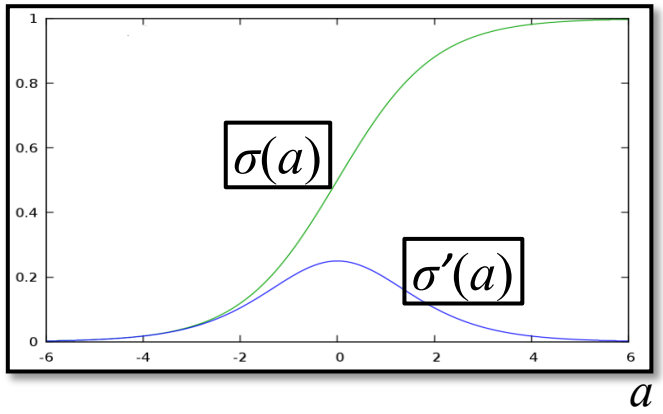
- Compute $\partial l/\partial a$ for the linearity output a.



$$\frac{\partial l}{\partial a} = \frac{\partial z}{\partial a} \cdot \frac{\partial l}{\partial z}$$

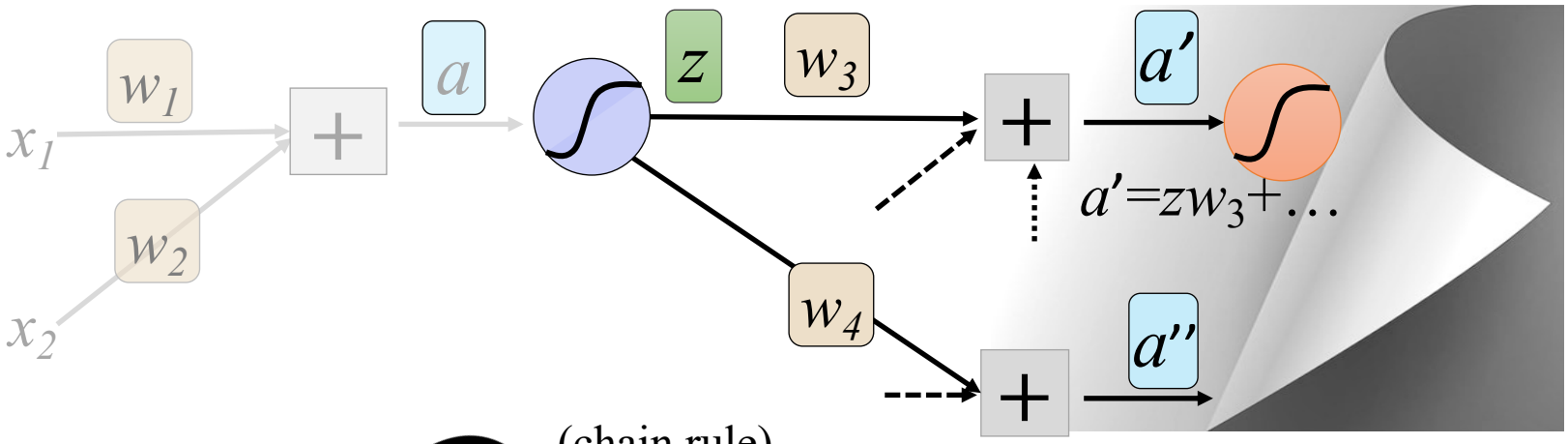
$$\frac{\partial z}{\partial a} = \sigma'(a)$$

Example



Backpropagation – Backward Pass

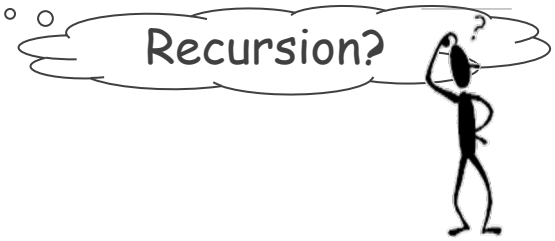
- Compute $\partial l/\partial z$ for nonlinearity outputs z.



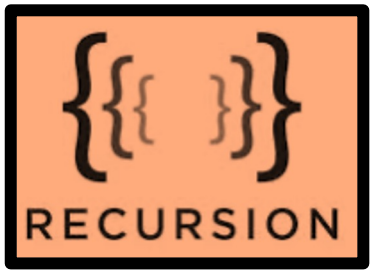
(chain rule)

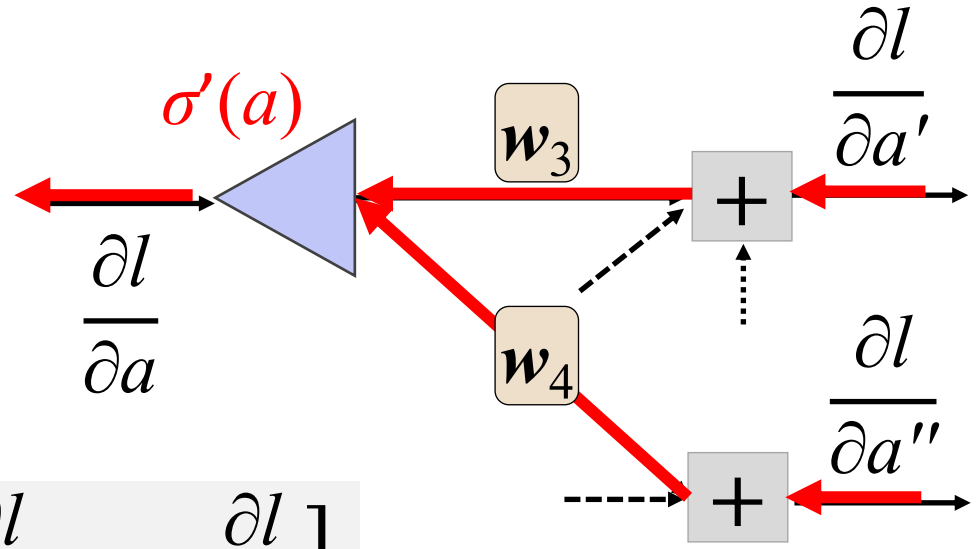
$$\frac{\partial l}{\partial a} = \sigma'(a) \cdot \frac{\partial l}{\partial z} = \frac{\partial a'}{\partial z} \cdot \frac{\partial l}{\partial a'} + \frac{\partial a''}{\partial z} \cdot \frac{\partial l}{\partial a''} \circ \circ$$

$\frac{\partial a'}{\partial z} = w_3$, $\frac{\partial l}{\partial a'}$ = ?, $\frac{\partial a''}{\partial z} = w_4$, $\frac{\partial l}{\partial a''}$ = ?



Backpropagation – Backward pass





$$\frac{\partial l}{\partial a} = \sigma'(a)\left[w_3 \frac{\partial l}{\partial a'} + w_4 \frac{\partial l}{\partial a''}\right]$$

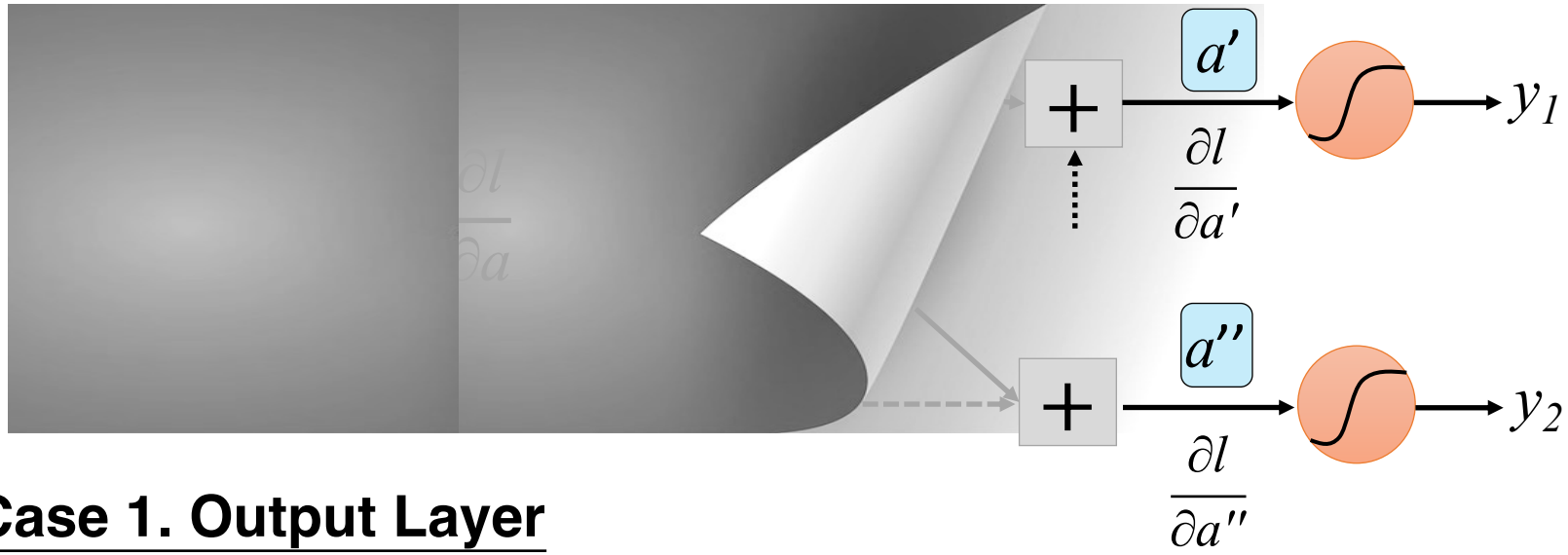
constant because a has already been determined in the forward pass.

Backward pass:

compute $\partial l / \partial a$ for all linearity outputs a recursively.

Backpropagation – Backward Pass

- Compute $\partial l/\partial a$ for all linearity outputs a **recursively**



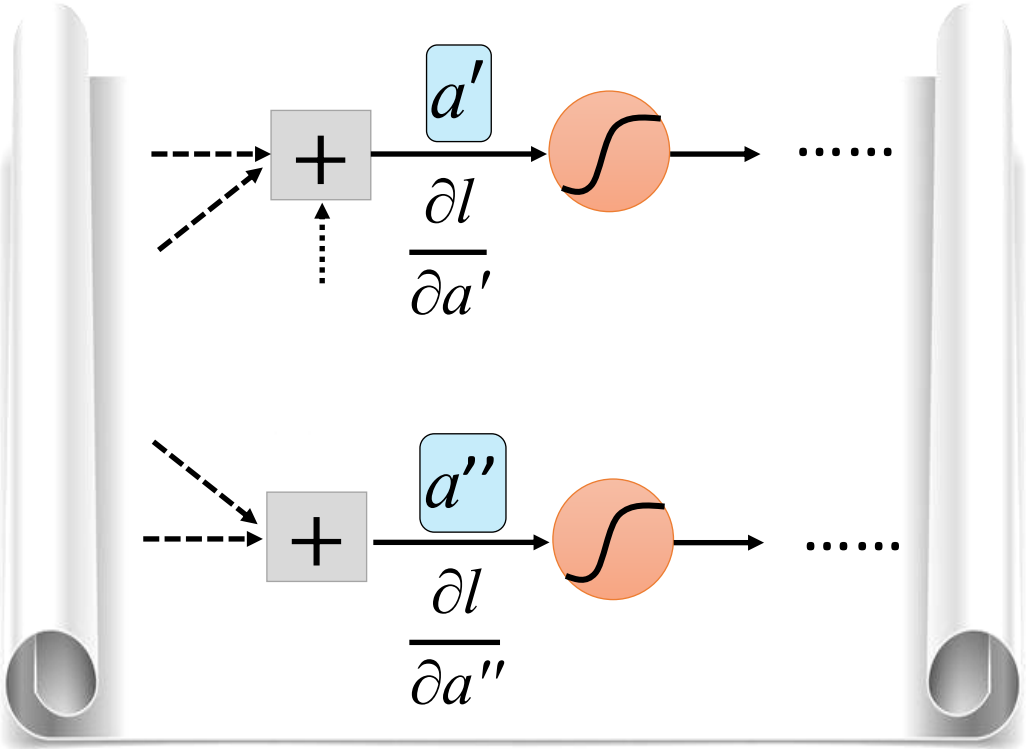
Case 1. Output Layer

$$\frac{\partial l}{\partial a'} = \frac{\partial y_1}{\partial a'}\frac{\partial l}{\partial y_1} \qquad \frac{\partial l}{\partial a''} = \frac{\partial y_2}{\partial a''}\frac{\partial l}{\partial y_2}$$

DONE

Backpropagation – Backward pass

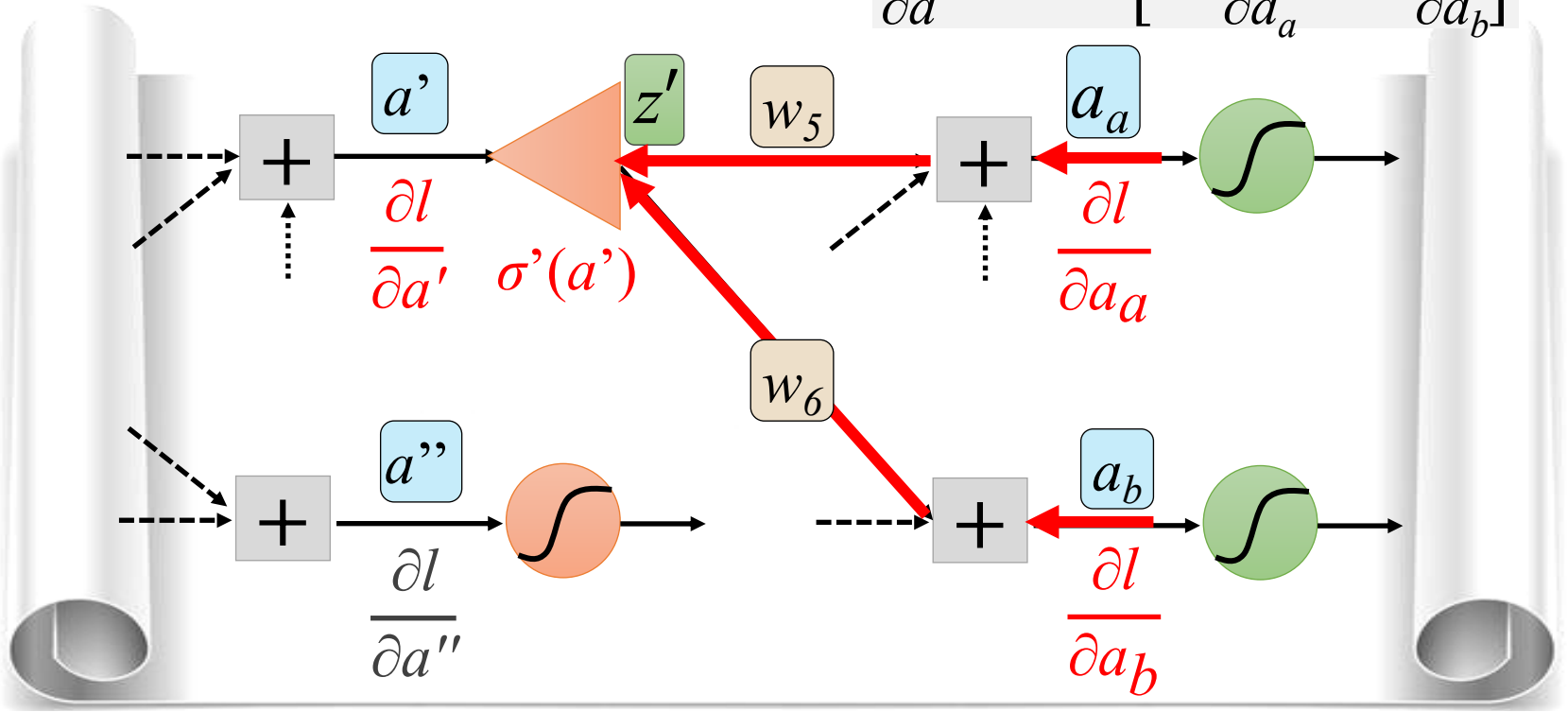
Case 2. Not an Output Layer



Backpropagation – Backward pass

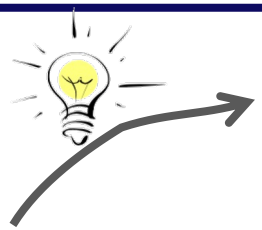
Case 2. Not an Output Layer

$$\frac{\partial l}{\partial a'} = \sigma'(a') \left[w_5 \frac{\partial l}{\partial a_a} + w_6 \frac{\partial l}{\partial a_b} \right]$$



Compute $\partial l / \partial a$ recursively until reaching the output layer…

Backpropagation – Backward Pass

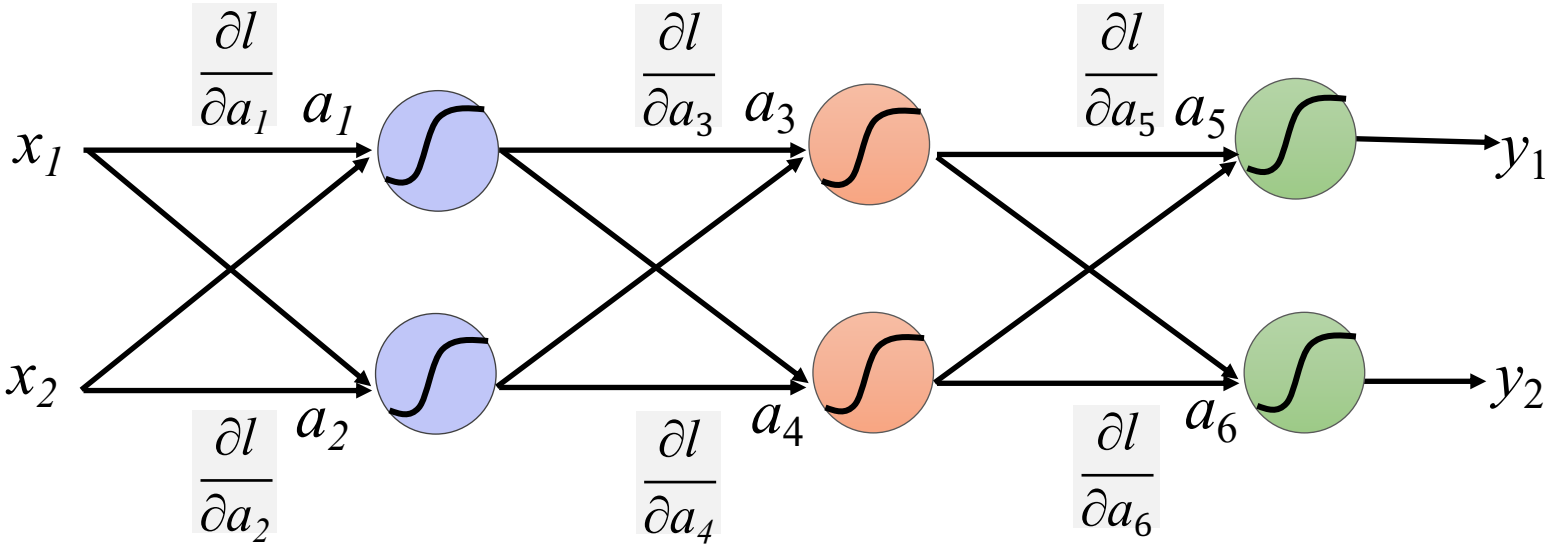


Problem

Recursion is often computationally inefficient

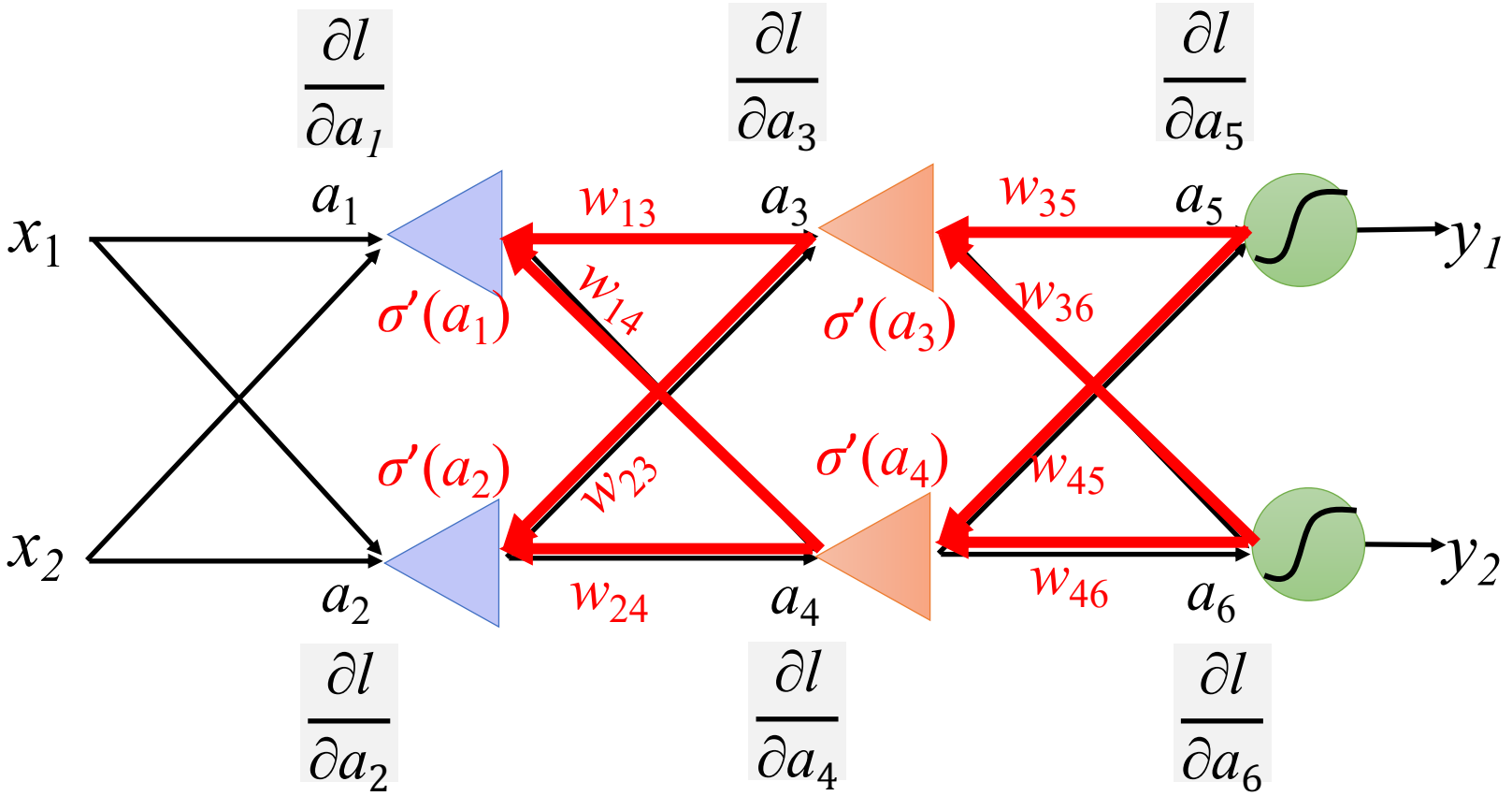
Idea: dynamic programming

Instead of computing $\partial l/\partial a$ recursively, we compute them from the output layer **backward** to the input layer.

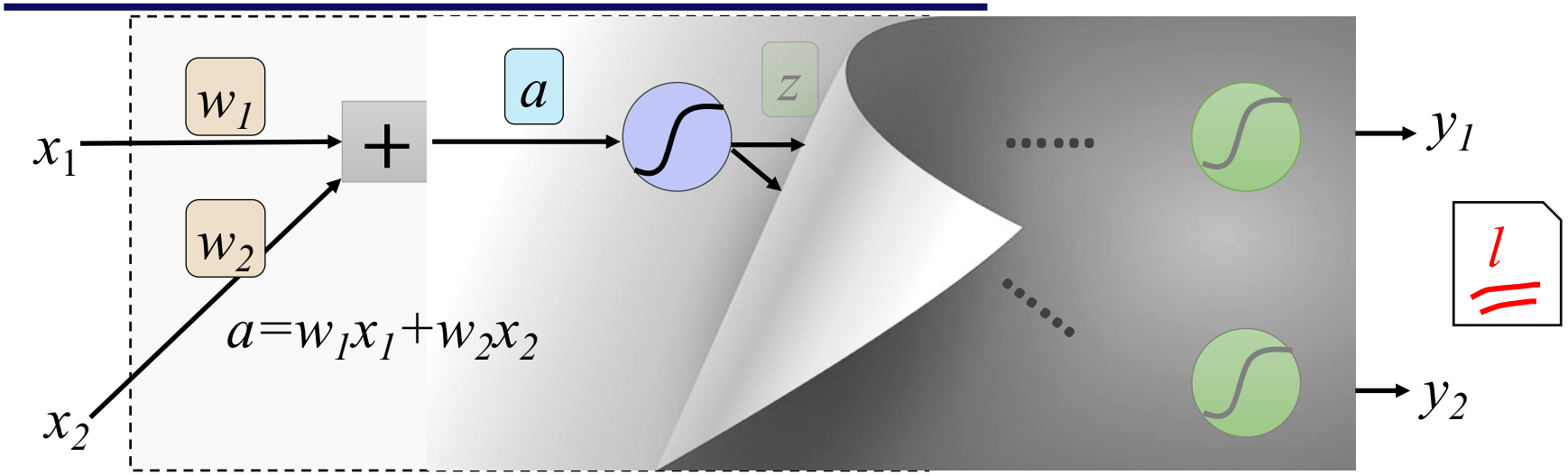


Backpropagation – Backward Pass

- Compute $\partial l/\partial a$ backward from the output layer.



Backpropagation – Forward Pass



$$\frac{\partial l}{\partial w} = \frac{\partial a}{\partial w} \cdot \frac{\partial l}{\partial a}$$

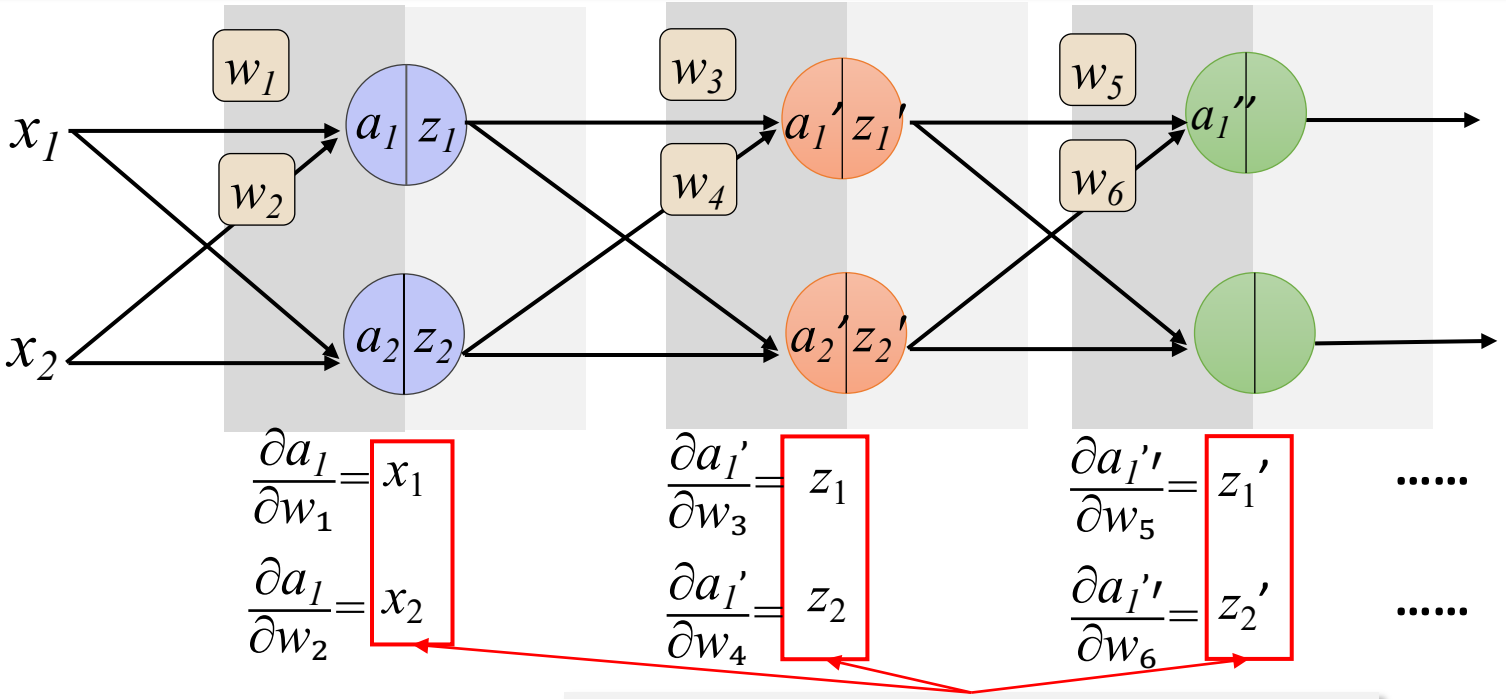
easy

$$\partial a/\partial w_1 = x_1$$
$$\partial a/\partial w_2 = x_2$$

For every w, there is a $\frac{\partial a}{\partial w}$ …

Backpropagation – Forward Pass

For every w, we need to compute $\partial a/\partial w$ within each linearity unit.



$\partial a/\partial w$ = the input connected to each w

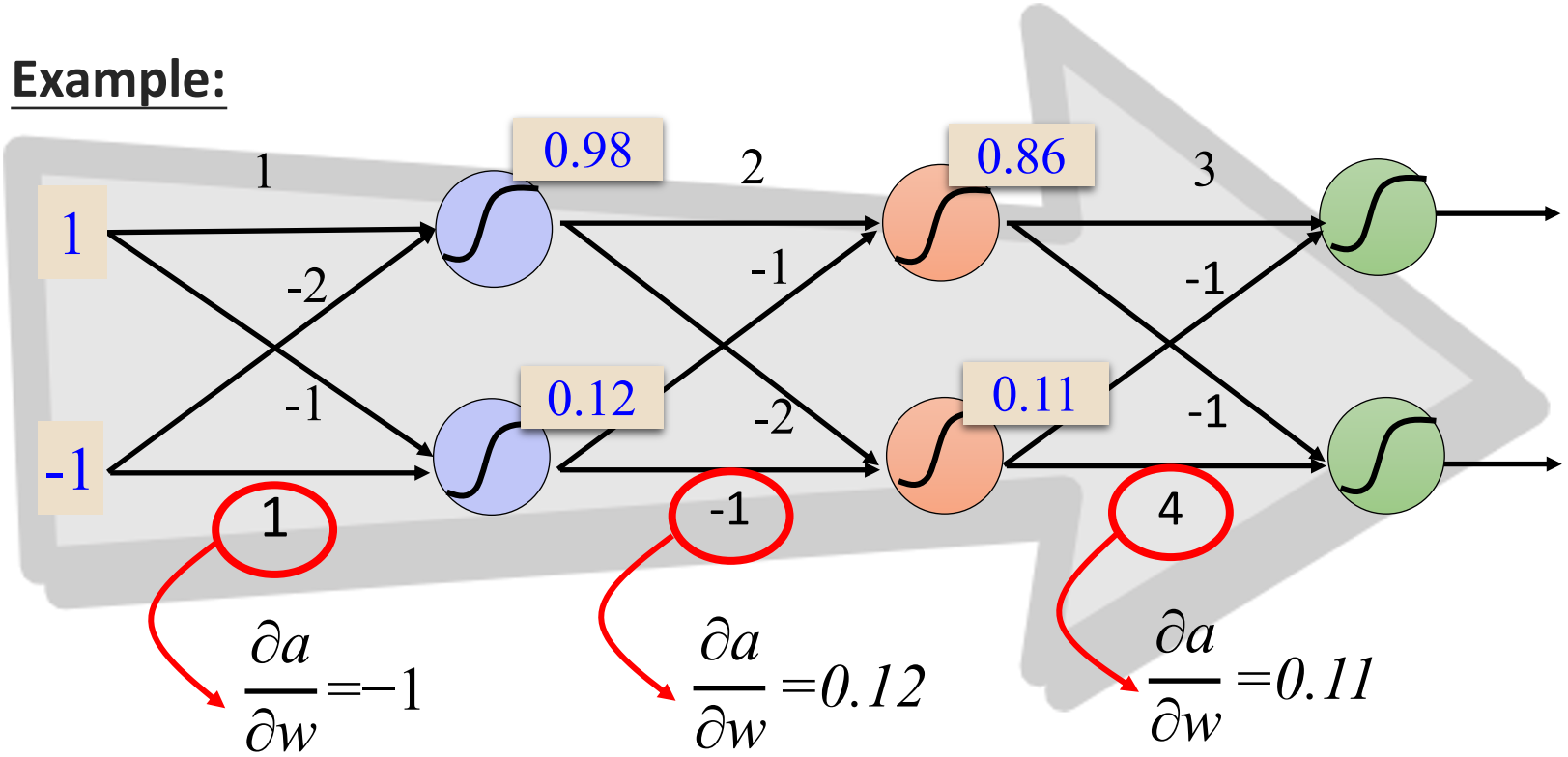
Idea: since the linear unit for each neuron is easy and fixed, we can pre-compute their gradients $\partial a/\partial w$ straightforward.

Backpropagation – Forward Pass

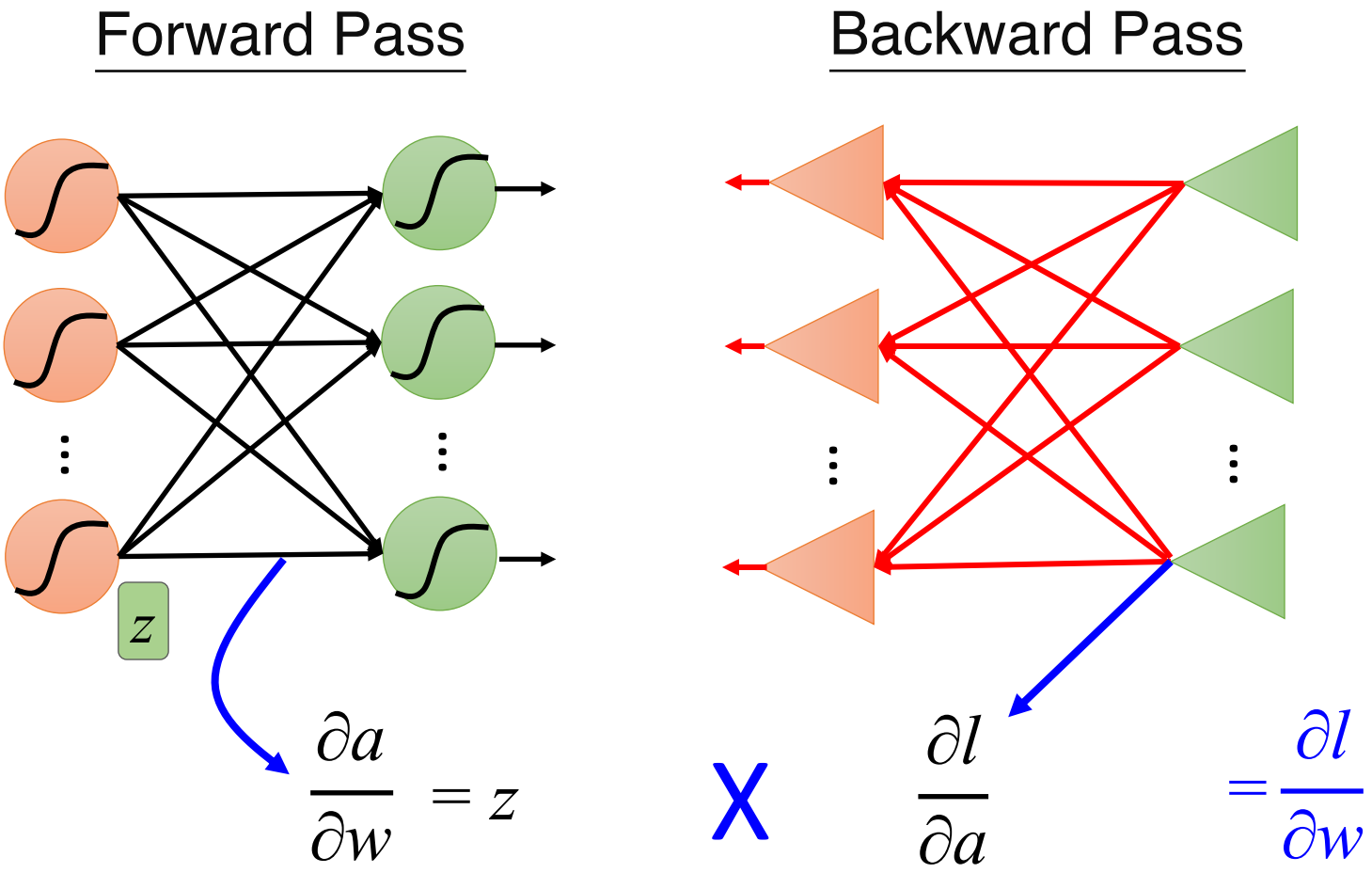
Forward pass:

Before backward pass, run MLP forward, obtain $\partial a/\partial w$ $(=z)$ for all $\boldsymbol{w}$

Example:



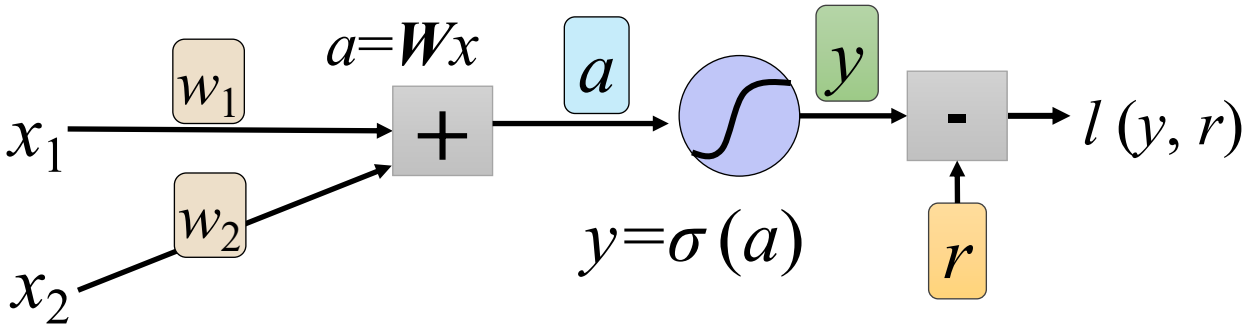
Backpropagation – Summary



Perspective from Error-Backpropagation

Recall: Logistic Regression

$$y = \sigma(\boldsymbol{W} x)$$



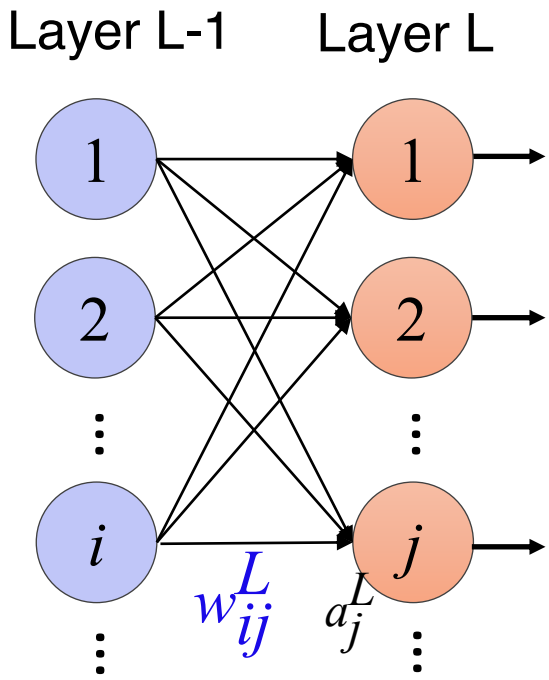
$$\frac{\partial L}{\partial w_j} = (r-y)\, x_j$$

error input

The update with respect to each weight is the product of the error and the input (signal)

Perspective from Error-Propagation

- The update to each weight is the product of the error and the input (signal).



$$\frac{\partial l}{\partial w_{ij}^L} = \frac{\partial a_j^L}{\partial w_{ij}^L} \frac{\partial l}{\partial a_j^L}$$

Input signal

$$\begin{cases} z_i^{L-1} & L > 1 \\ x_i & L = 1 \end{cases}$$

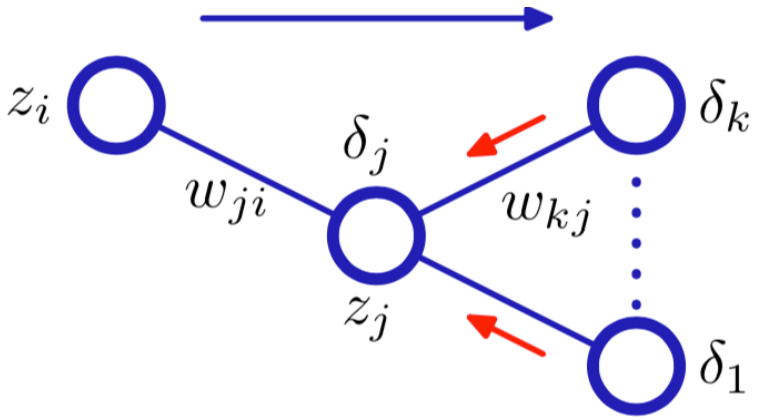
δ_j^L

?

Consider $\frac{\partial l}{\partial a_j} \equiv \delta_j$ as the error signal for all w_{ij}

Perspective from Error-Propagation

- Clearly, for the output unit, we have $\delta_k = l\,(y,\, r)$



- Recursively,

$$\delta_j = \frac{\partial l}{\partial a_j} = \sum_k \frac{\partial l}{\partial a_k}\frac{\partial a_k}{\partial a_j} \quad \text{(Chain Rule)}$$

$$= \sigma'(a_j) \sum_k w_{kj}\delta_k$$

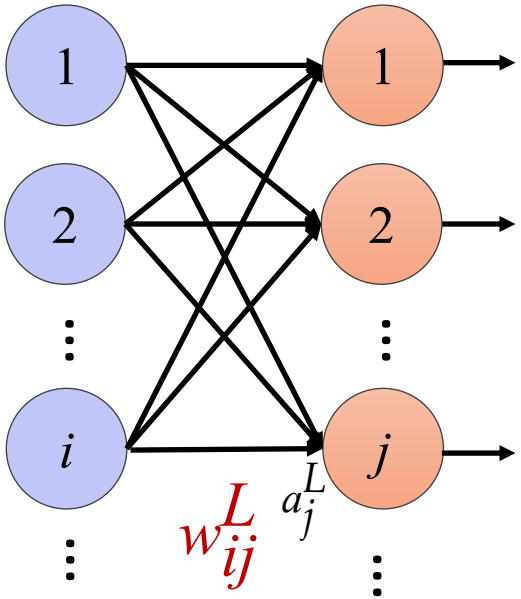
$=\delta_k$

$=w_{kj}\sigma'(a_j)$

"Amplifier"

- The value of δ for a particular hidden unit can be obtained by propagating the δ's backwards from units higher up in the network.

Perspective from Error-Propagation



$$\frac{\partial l}{\partial w_{ij}^L} = \frac{\partial a_j^l}{\partial w_{ij}^l} \frac{\partial l}{\partial a_j^L}$$

Input signal

Error signal

$$\begin{cases} z_i^{L-1} & L > 1 \\ x_i & L = 1 \end{cases}$$

$$\delta_j^L$$

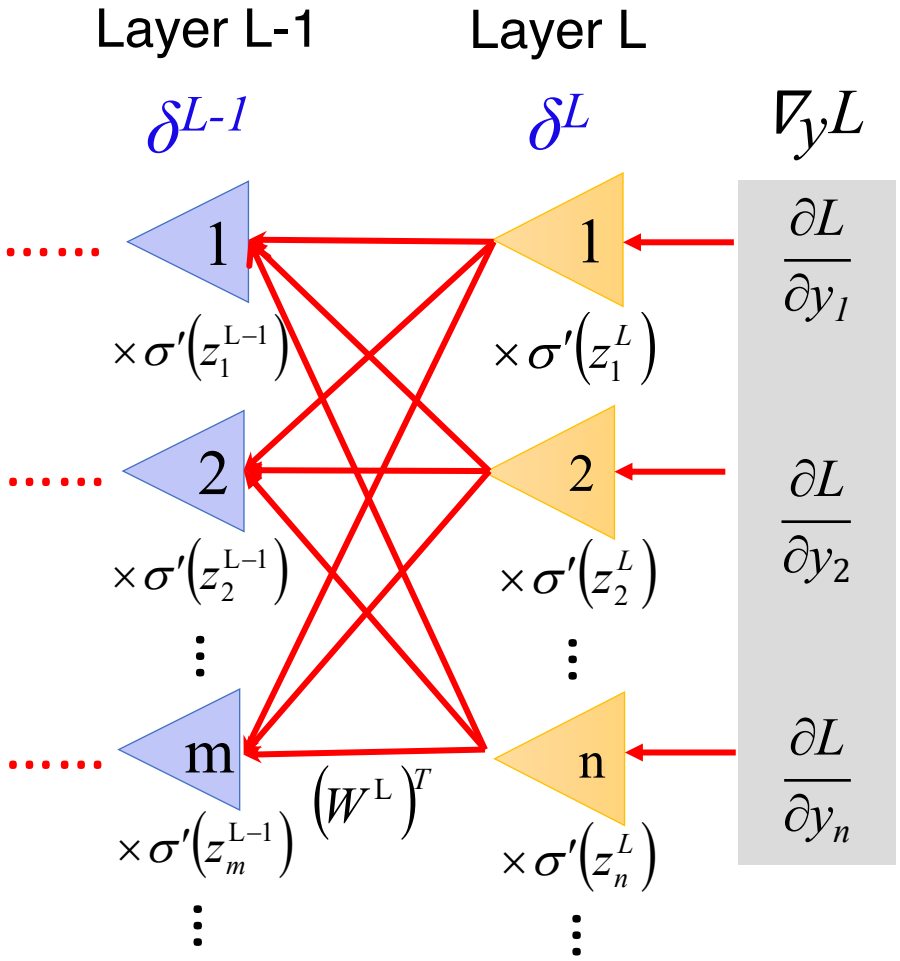
Forward Pass

$$z^1 = W^1 x$$
$$a^1 = \sigma(z^1)$$
$$\cdots\cdots$$
$$z^{l-1} = W^{l-1} a^{l-2}$$
$$a^{l-1} = \sigma(z^{l-1})$$

Backward Pass

$$\delta^L = \sigma'(z^L) \bullet \nabla_y Loss$$
$$\delta^{L-1} = \sigma'\left(z^{L-1}\right) \bullet \left(W^L\right)^T \delta^L$$
$$\cdots\cdots$$
$$\delta^l = \sigma'\left(z^l\right) \bullet \left(W^{l+1}\right)^T \delta^{l+1}$$
$$\cdots\cdots$$

Perspective from Error-Propagation



(We do not use softmax here)

The Algorithm

1. Forward propagation:

- apply the input vector to the network and evaluate the activations of all hidden and output units.

$$a_j = \sum_i w_{ji} z_i \qquad z_j = \sigma(a_j)$$

2. Backward propagation:

- evaluate the derivatives of the loss function with respect to the weights (errors).

$$\delta_j^L = \frac{\partial l}{\partial a_j^L}$$

- errors are propagated backwards through the network.

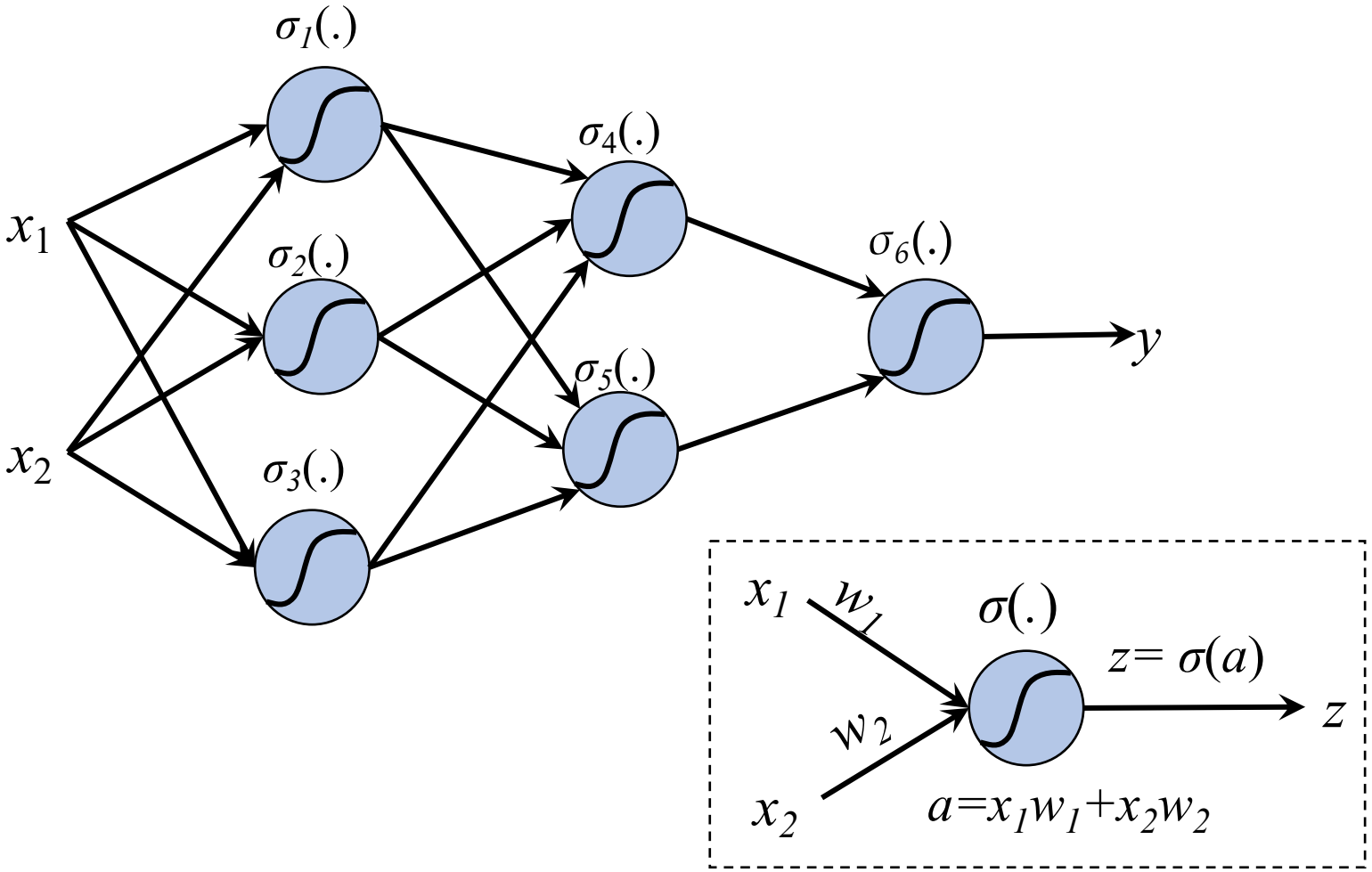
$$\delta^{L-1} = \sigma'(z^{L-1})(W^L)^T \delta^L$$

3. Parameter update:

- the evaluated derivatives (errors) are then used to compute the adjustments to be made to the parameters.

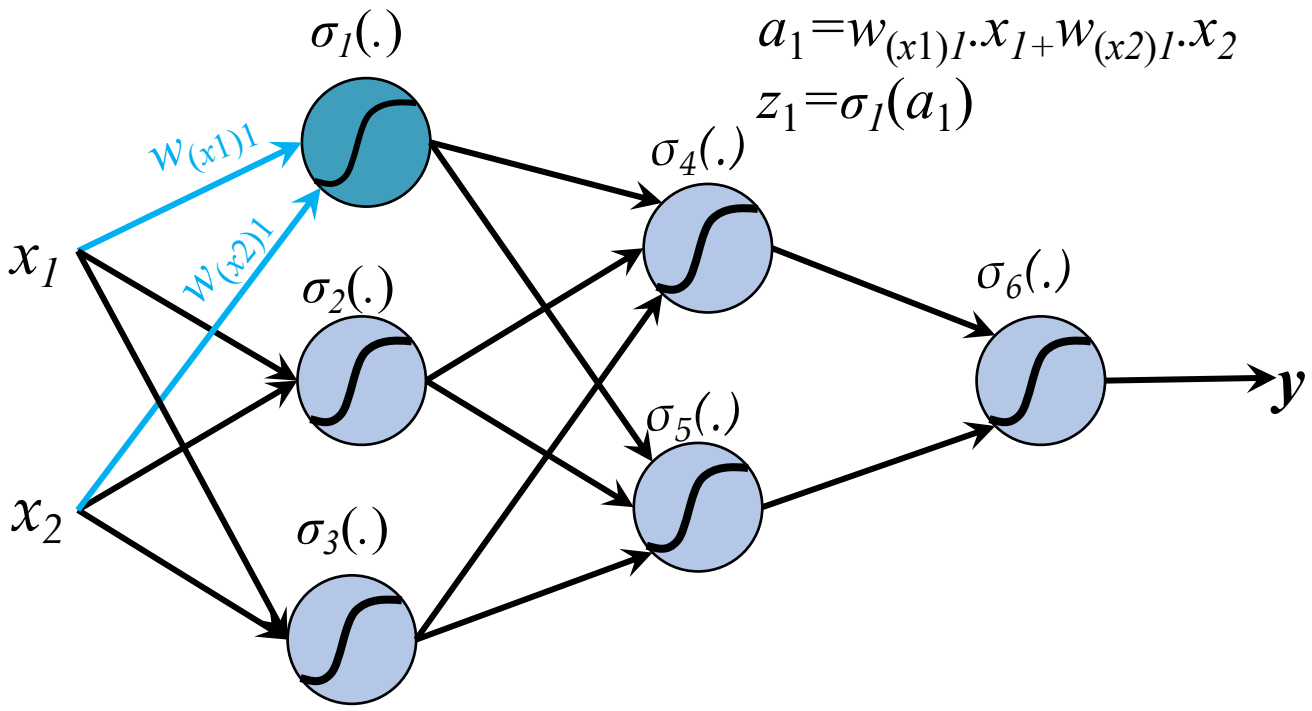
$$w'_{ij} = w_{ij} + \eta \delta_j z_i$$

BP Example

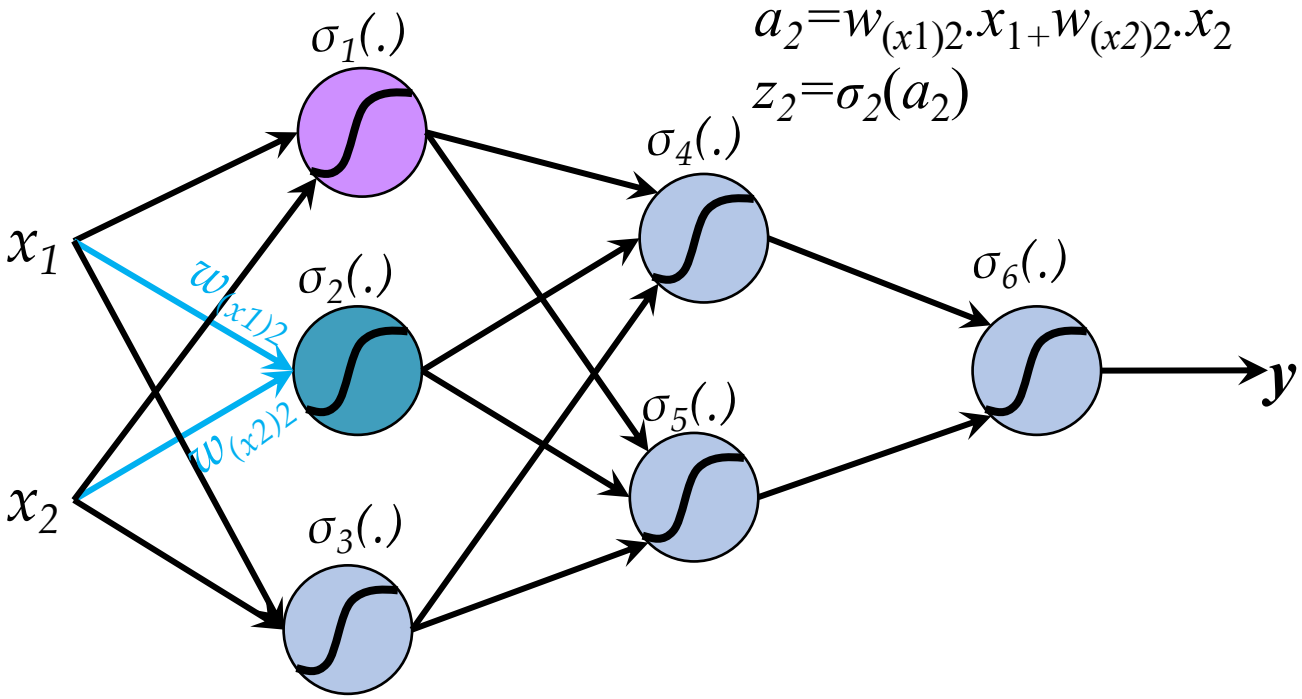

$\sigma_1(.)$
$\sigma_2(.)$
$\sigma_3(.)$
$\sigma_4(.)$
$\sigma_5(.)$
$\sigma_6(.)$
x_1
x_2
y
x_1
w_1
$\sigma(.)$
$z= \sigma(a)$
w_2
z
x_2
$a=x_1w_1+x_2w_2$

BP Example

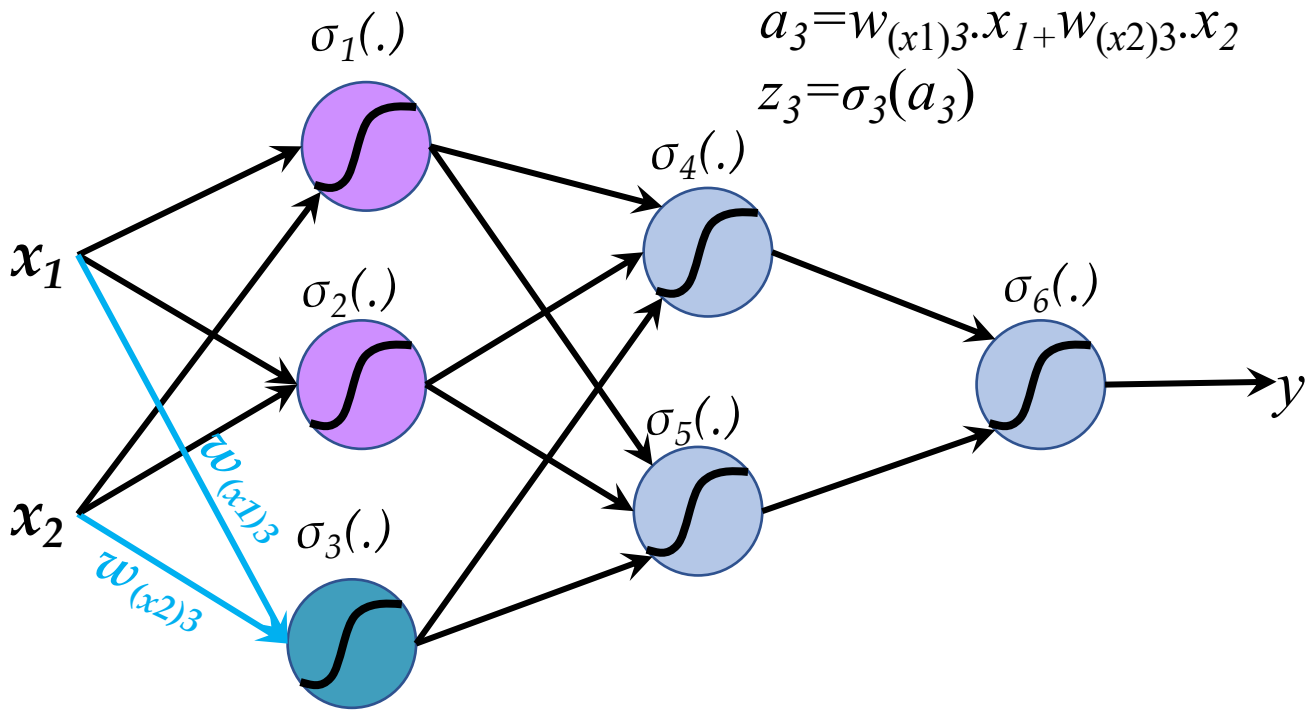
Forward Pass



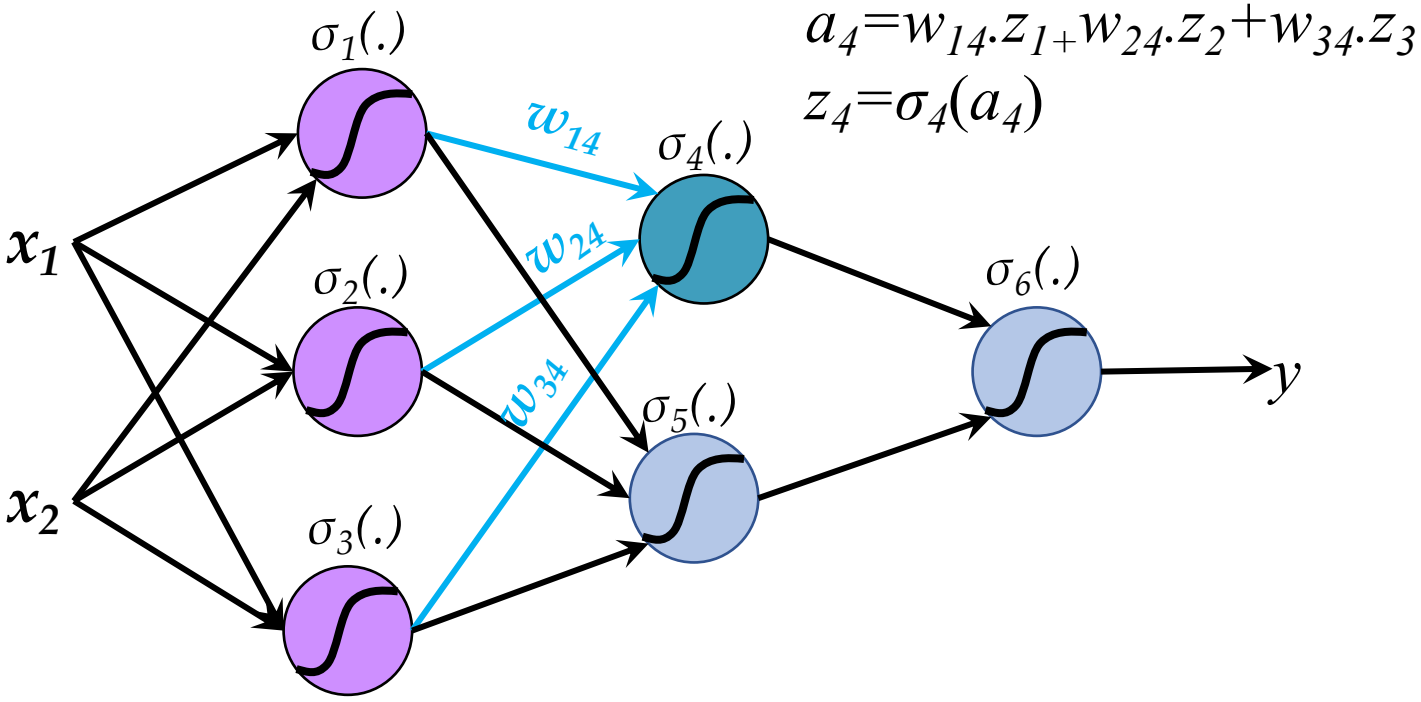
Forward Pass



Forward Pass

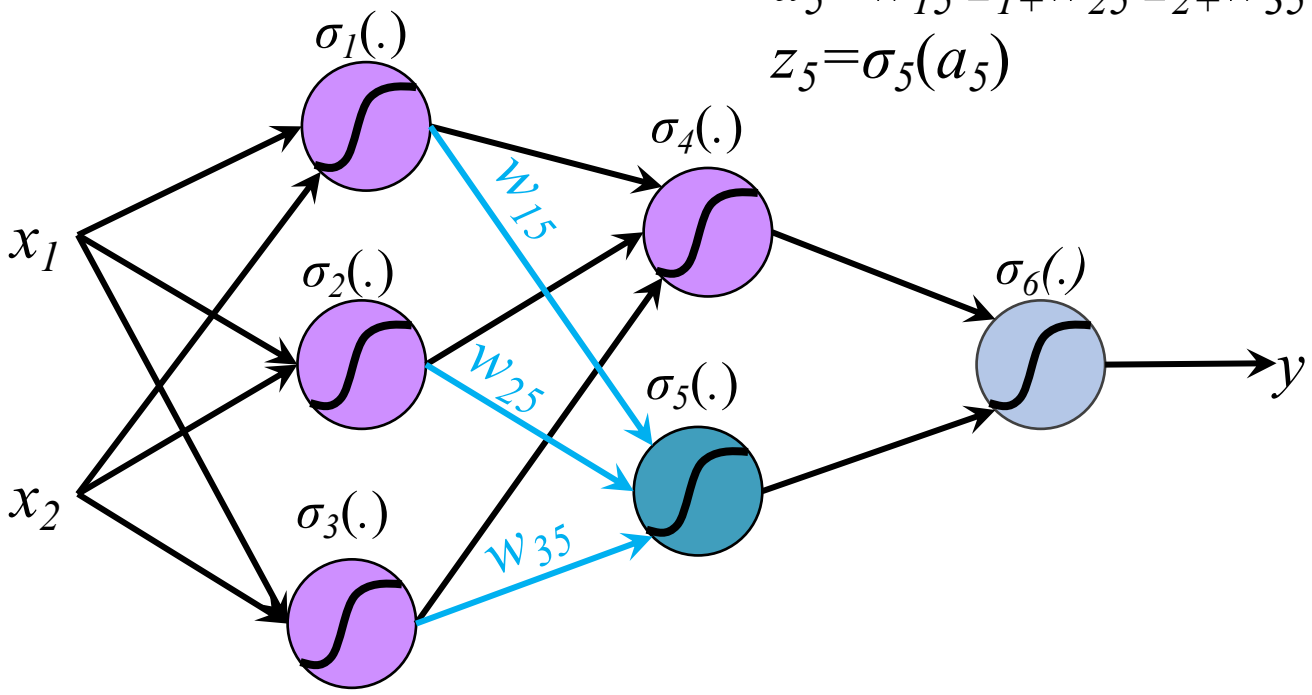


Forward Pass

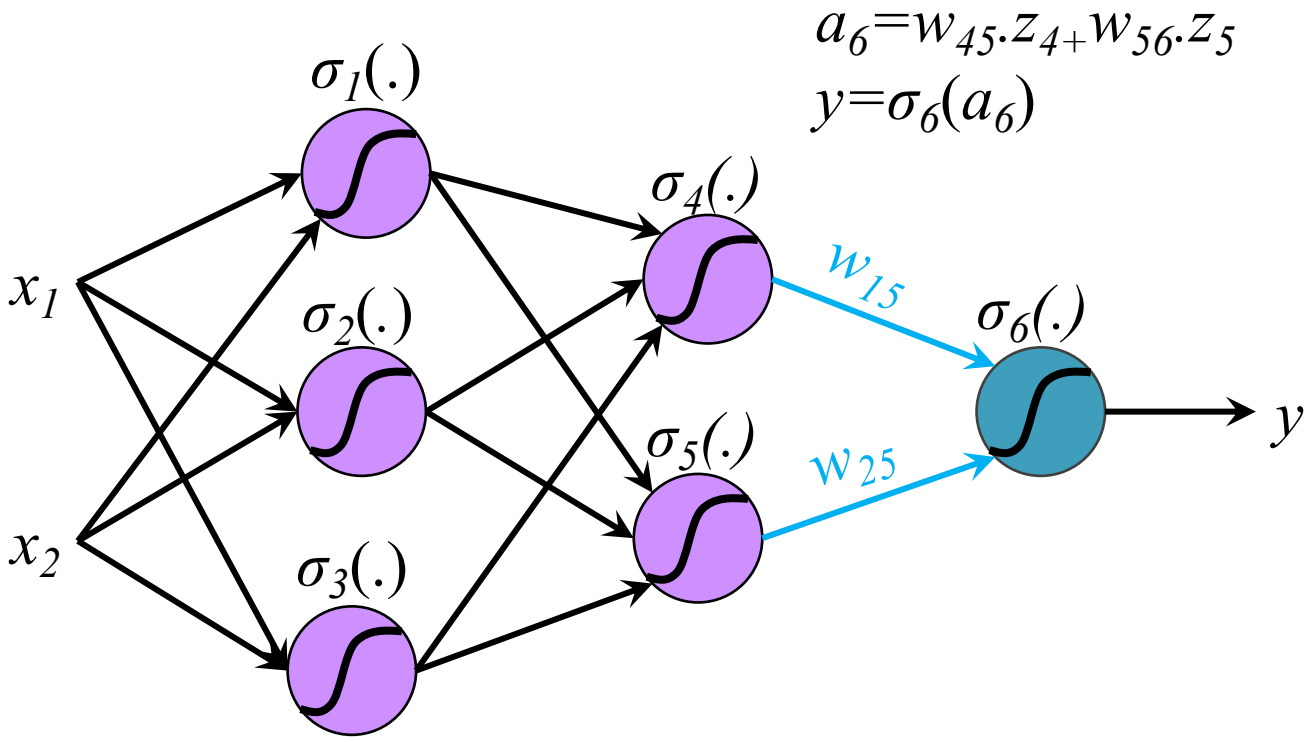


Forward Pass

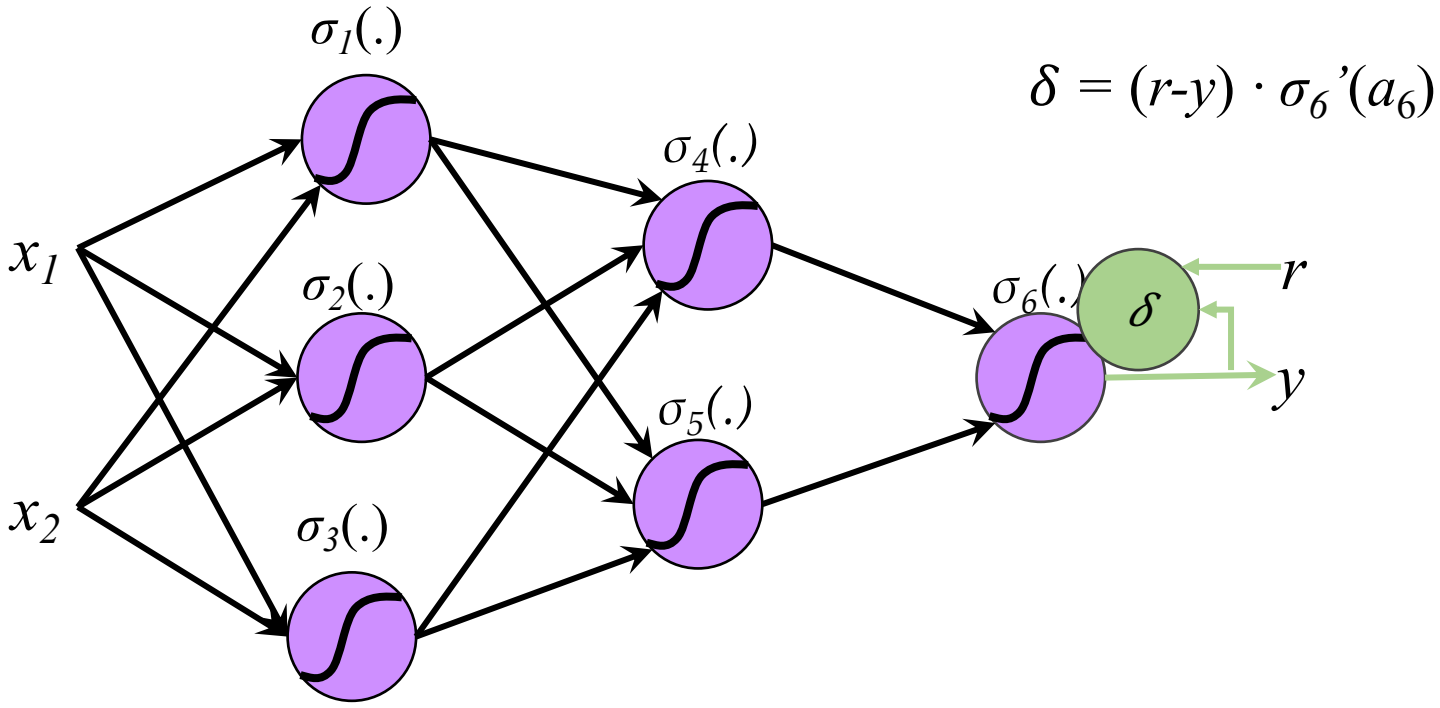
$$a_5 = w_{15} \cdot z_1 + w_{25} \cdot z_2 + w_{35} \cdot z_3$$
$$z_5 = \sigma_5(a_5)$$



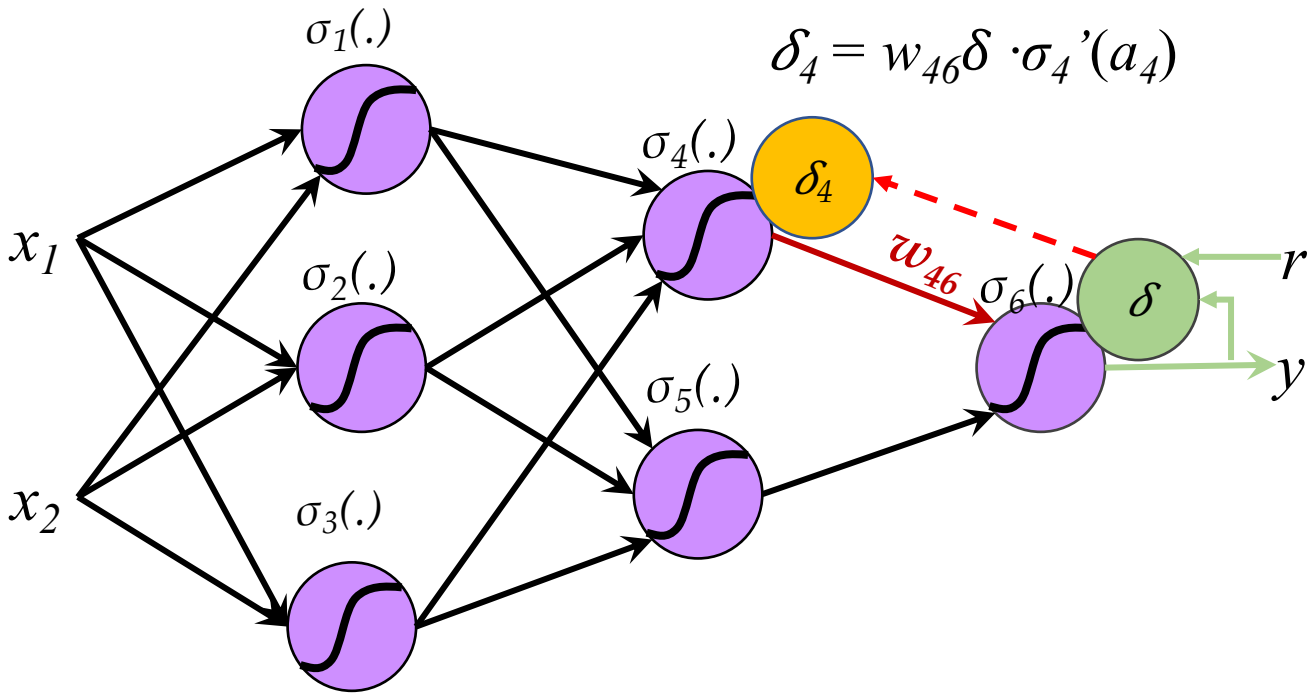
Forward Pass



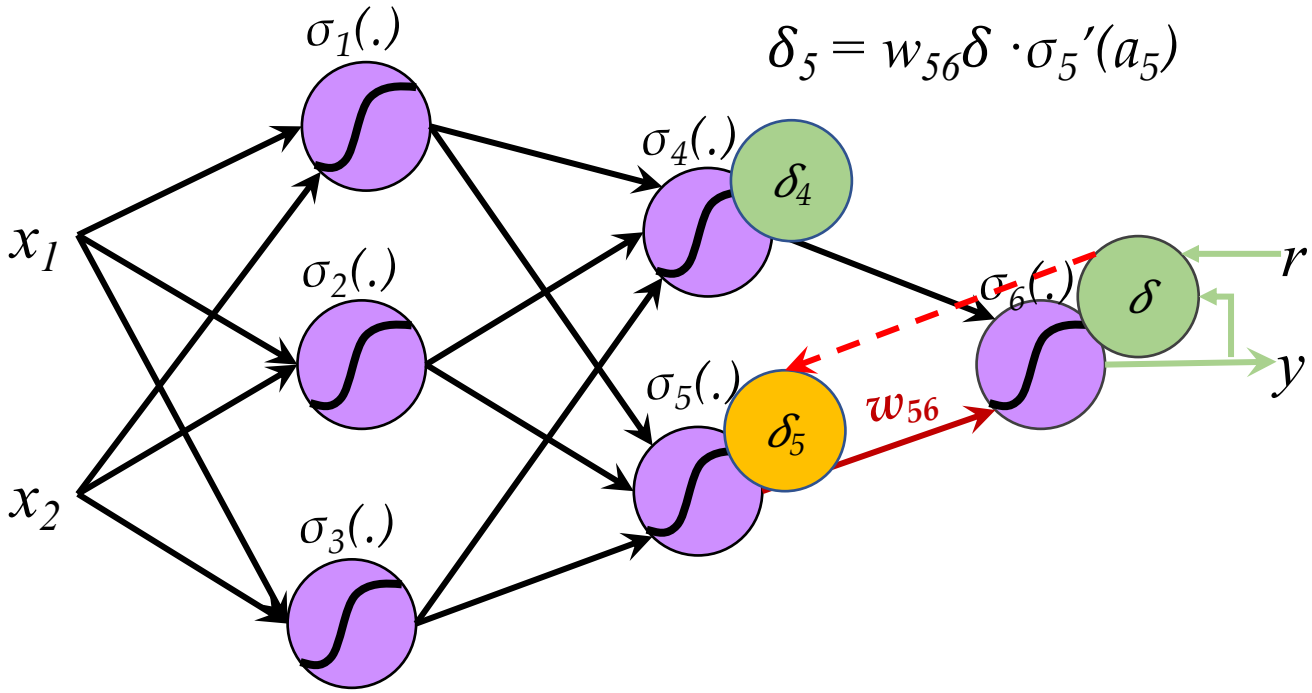
Backward Pass – error propagation



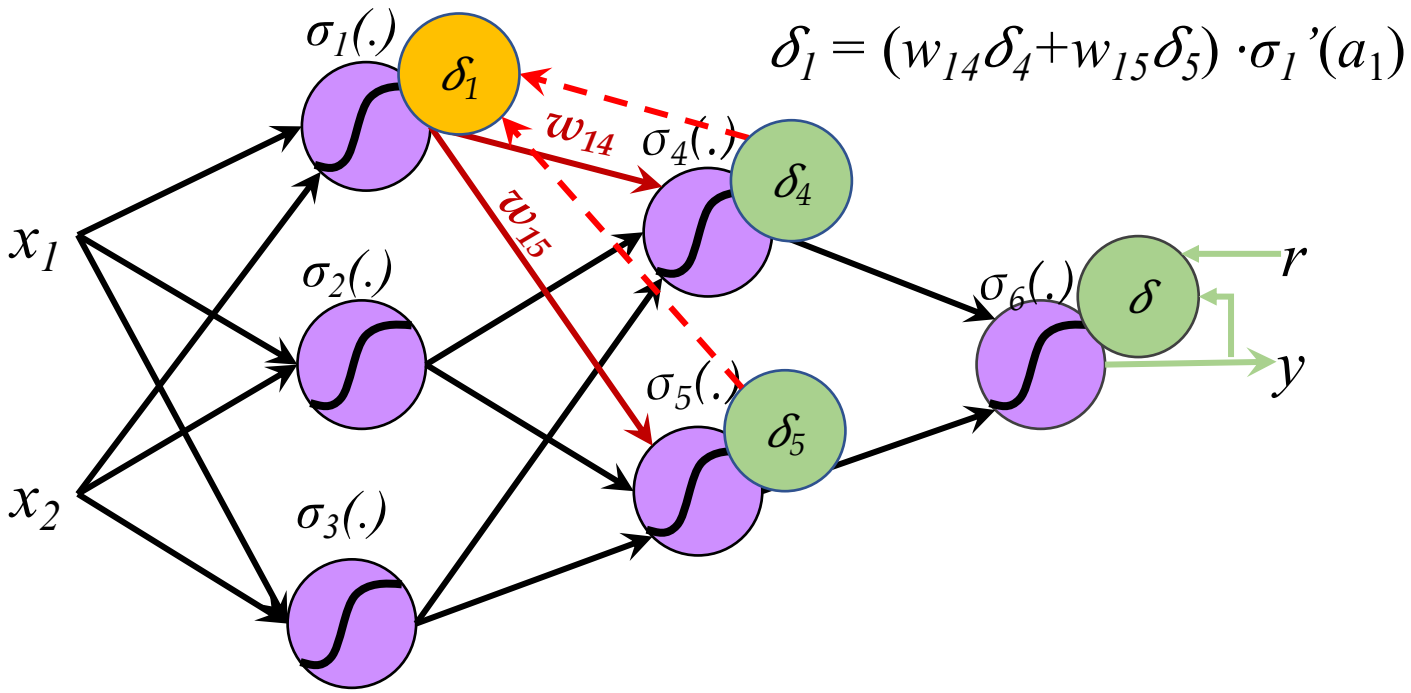
Backward Pass – error propagation



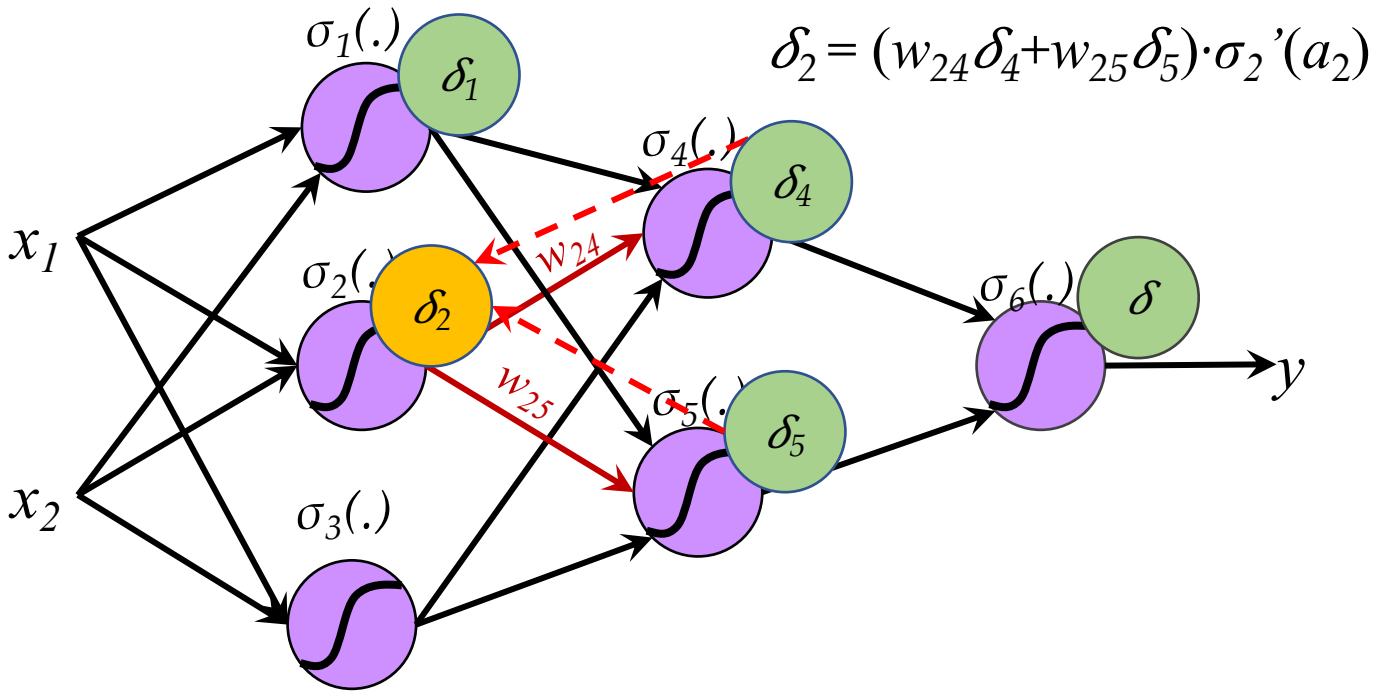
Backward Pass – error propagation



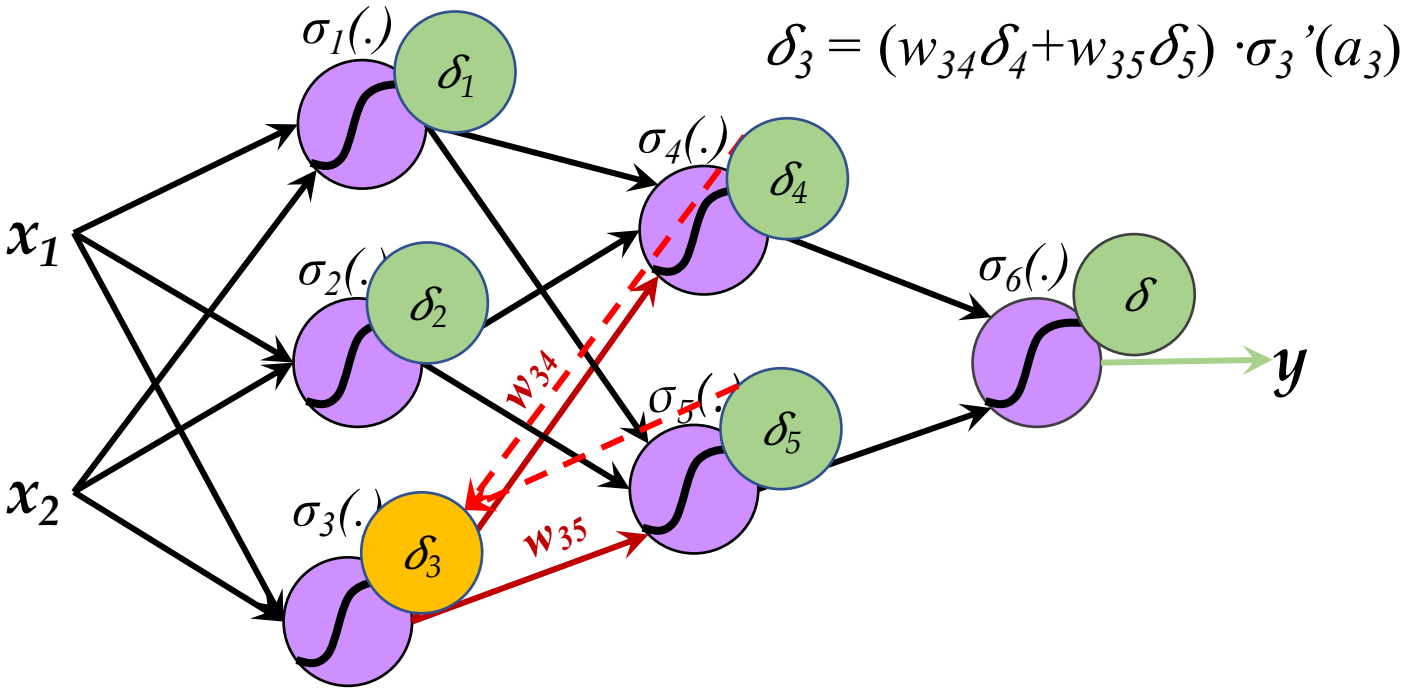
Backward Pass – error propagation



Backward Pass – error propagation



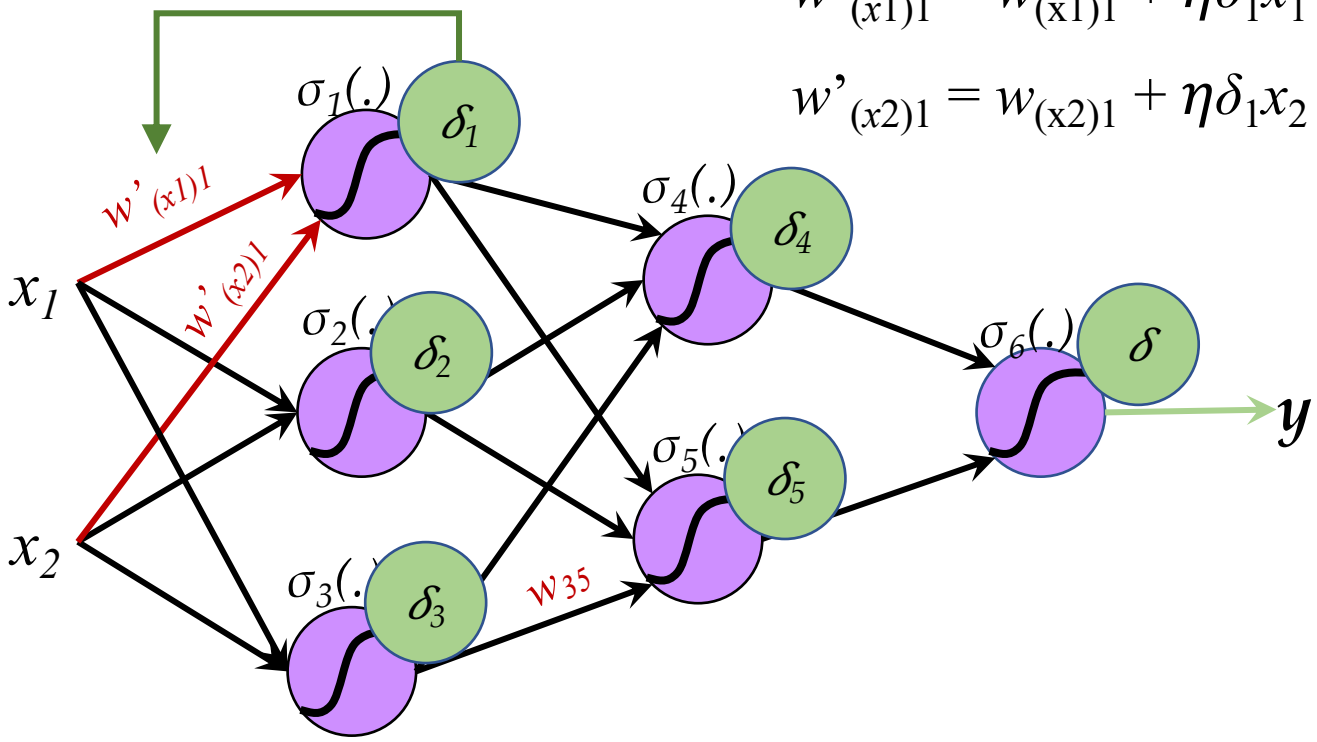
Backward Pass – error propagation



Optimization step – parameter update

$$w'_{(x1)1} = w_{(x1)1} + \eta\delta_1 x_1$$

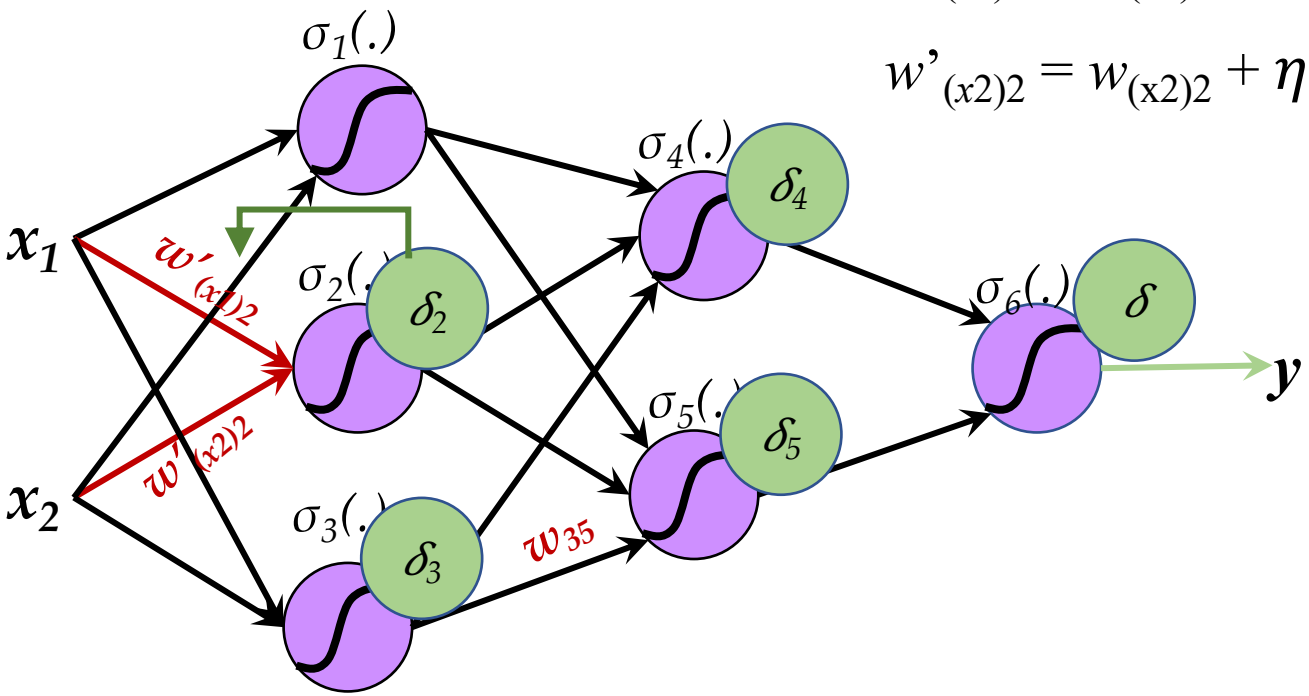
$$w'_{(x2)1} = w_{(x2)1} + \eta\delta_1 x_2$$



Optimization step – parameter update

$$w'_{(x1)2} = w_{(x1)2} + \eta\delta_2 x_1$$

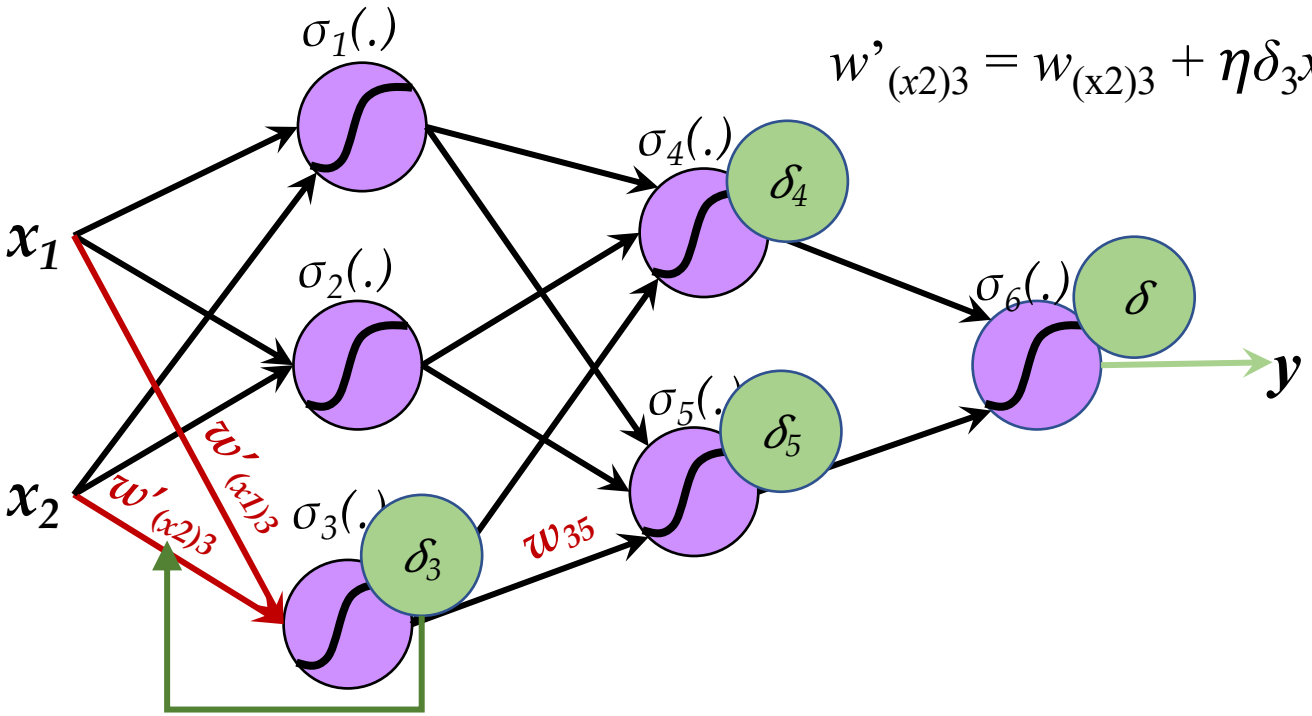
$$w'_{(x2)2} = w_{(x2)2} + \eta\delta_2 x_2$$



Optimization step – parameter update

$$w'_{(x1)3} = w_{(x1)3} + \eta\delta_3 x_1$$

$$w'_{(x2)3} = w_{(x2)3} + \eta\delta_3 x_2$$

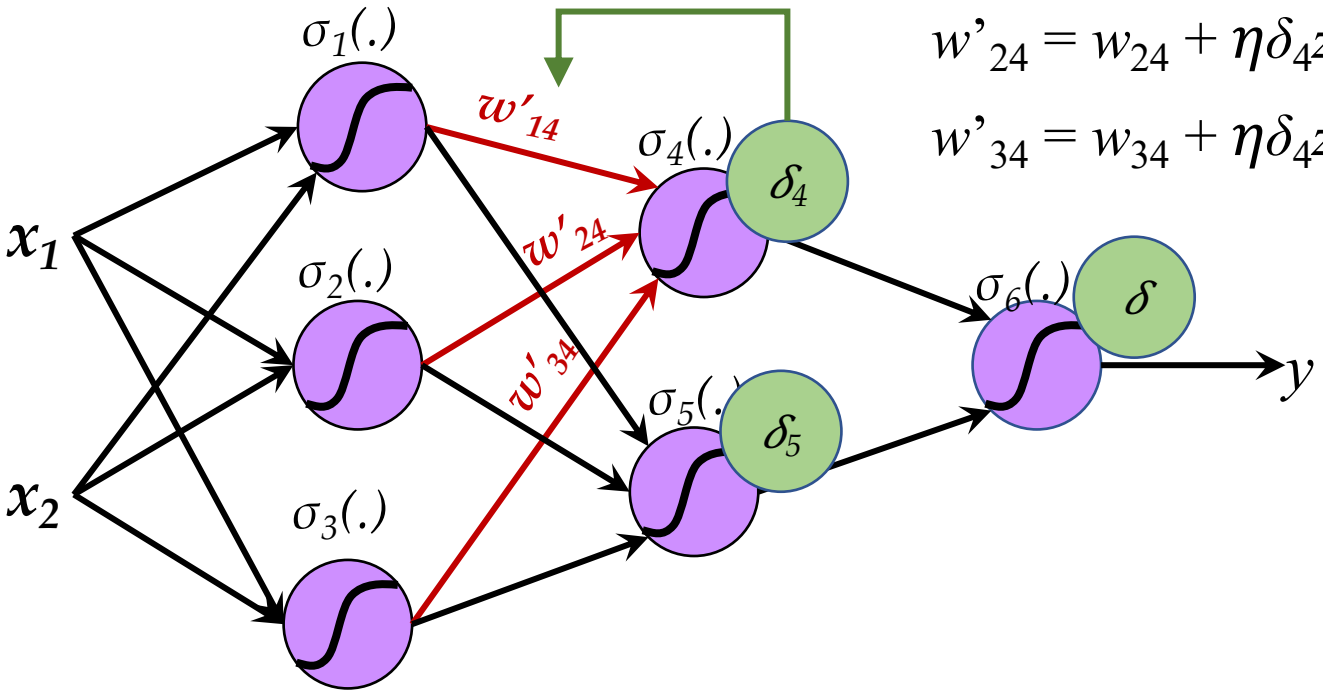


Optimization step – parameter update

$$w'_{14} = w_{14} + \eta\delta_4 z_1$$

$$w'_{24} = w_{24} + \eta\delta_4 z_2$$

$$w'_{34} = w_{34} + \eta\delta_4 z_3$$

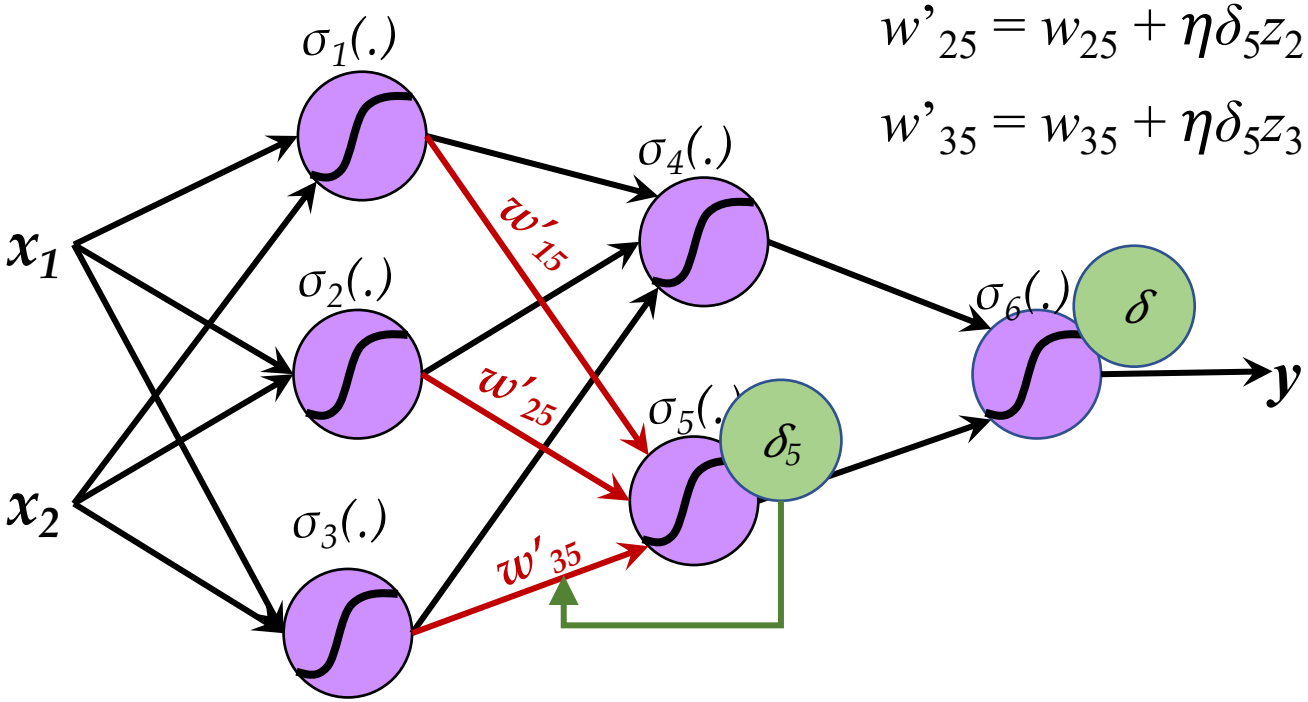


Optimization step – parameter update

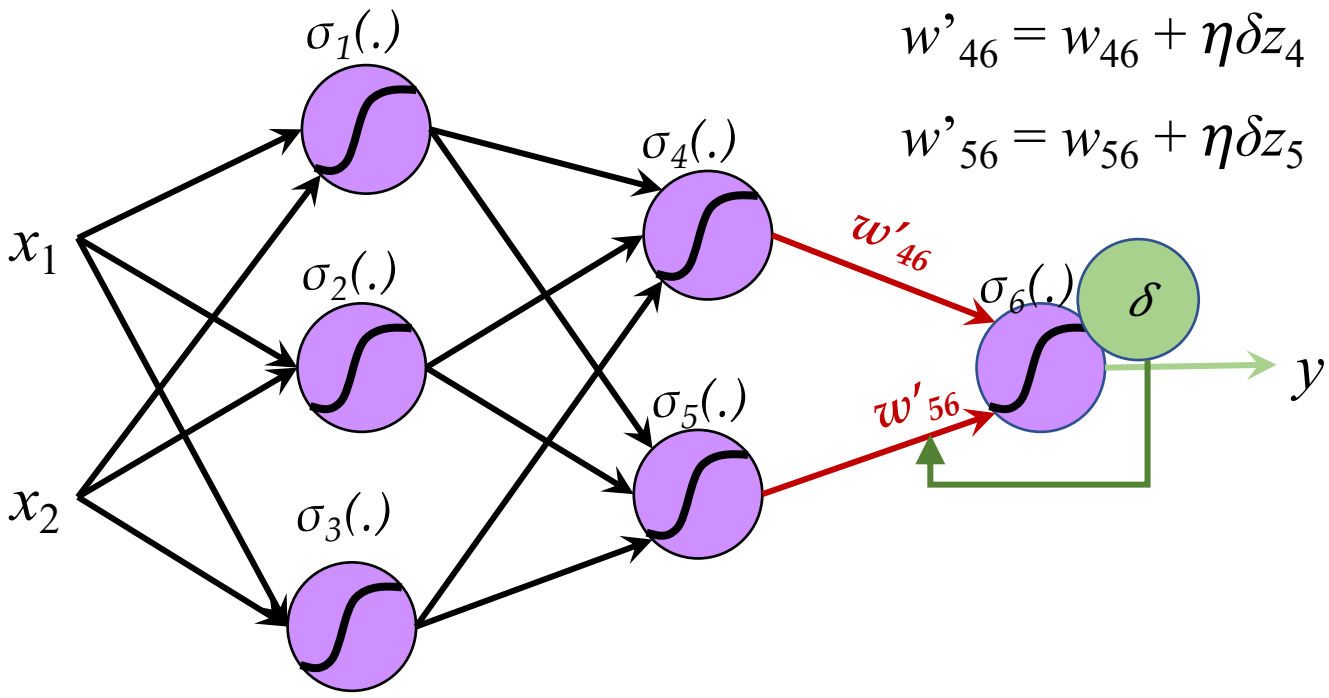
$$w'_{15} = w_{15} + \eta\delta_5 z_1$$

$$w'_{25} = w_{25} + \eta\delta_5 z_2$$

$$w'_{35} = w_{35} + \eta\delta_5 z_3$$



Optimization step – parameter update



$$w'_{46} = w_{46} + \eta\delta z_4$$

$$w'_{56} = w_{56} + \eta\delta z_5$$

Time for Coding

- **BP**:

  https://www.kaggle.com/code/kaushal3663/backpropagation

- **MLP using PyTorch**

  https://www.kaggle.com/code/pinocookie/pytorch-simple-mlp/notebook