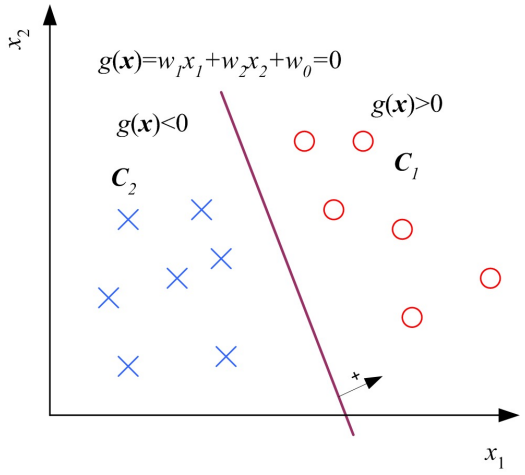
Machine Learning

Chapter 8: Multi-layer Perceptrons

Fall 2021

Instructor: Xiaodong Gu

Recall: Linear Classifiers

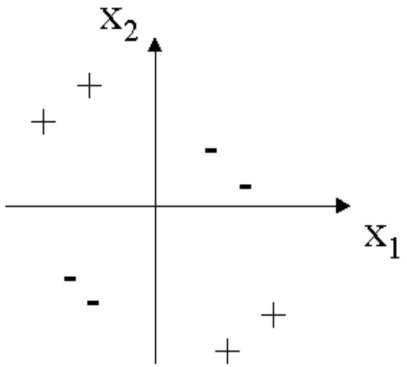


Perceprton

Logistic Regression

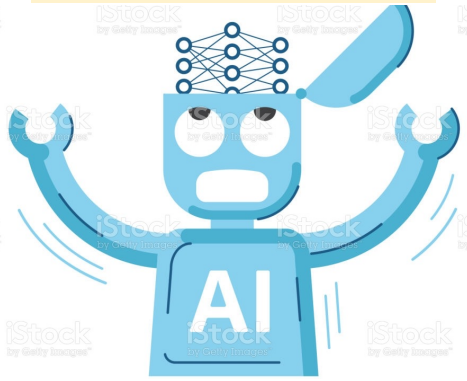
SVM

The curse of nonlinearity



I can approximate any non-linear functions !

Neural Networks

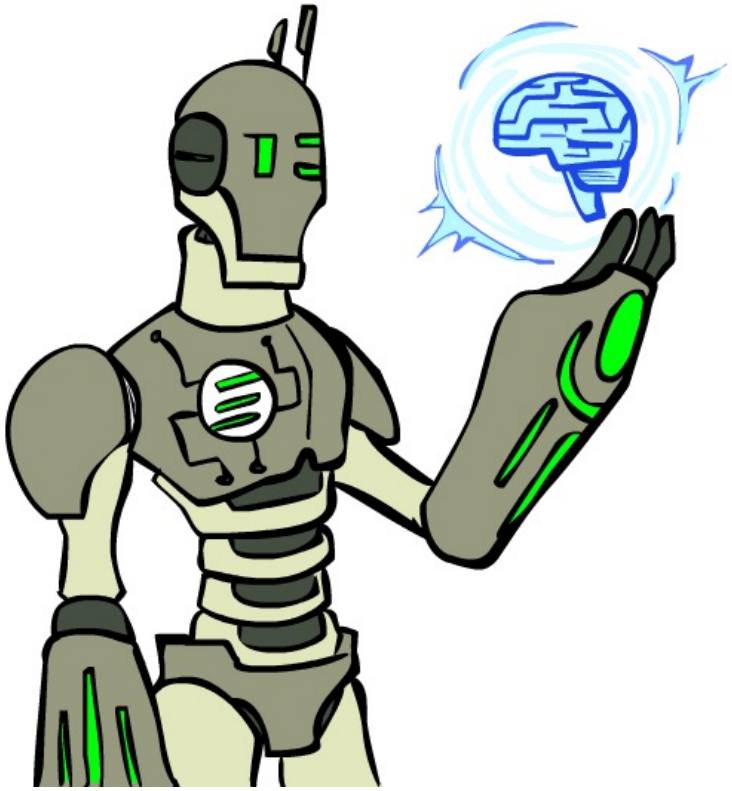


Today

Artificial Neural Networks

Multi-layer Perceptrons

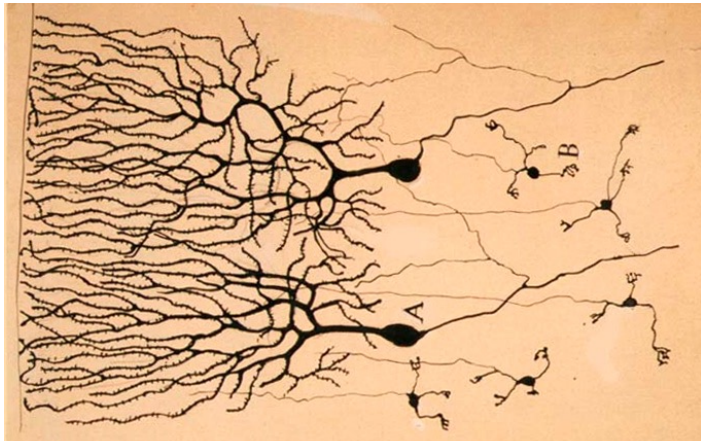
Backpropagation



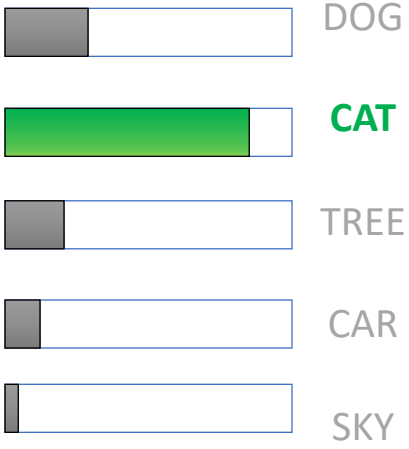
Artificial Neural Networks

Inspired by actual human brain

Input



Output

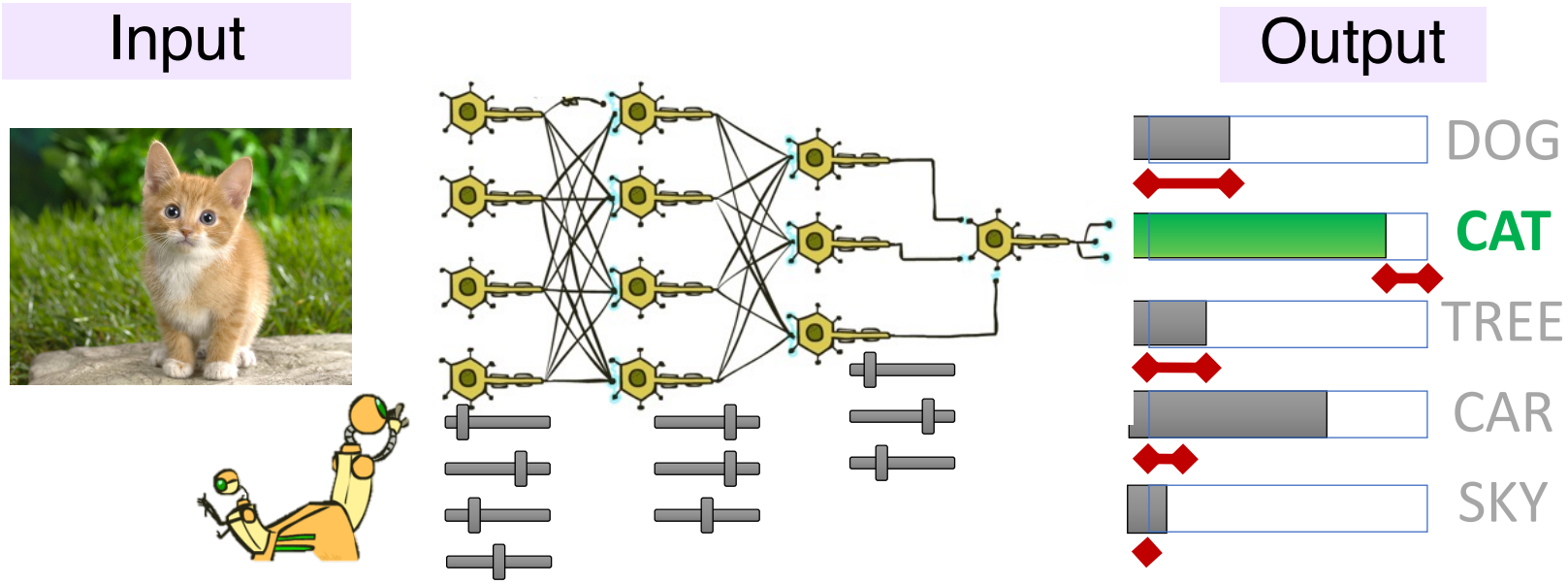


A human brain has:

- Large number (1011) of neurons as processing units
- Large number (104) 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.



Many layers of neurons, millions of parameters

Artificial Neural Network

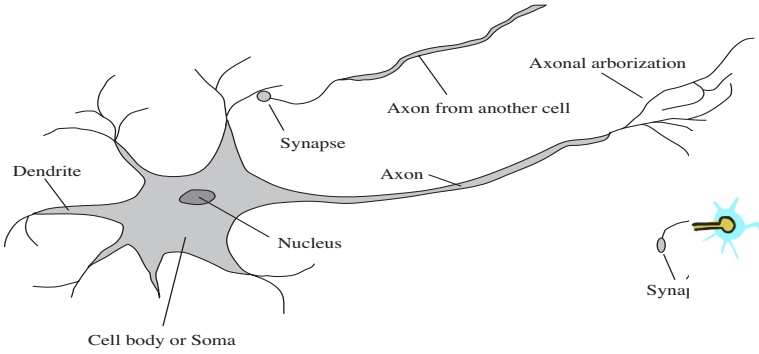
Neuroscientists vs. AI scientists

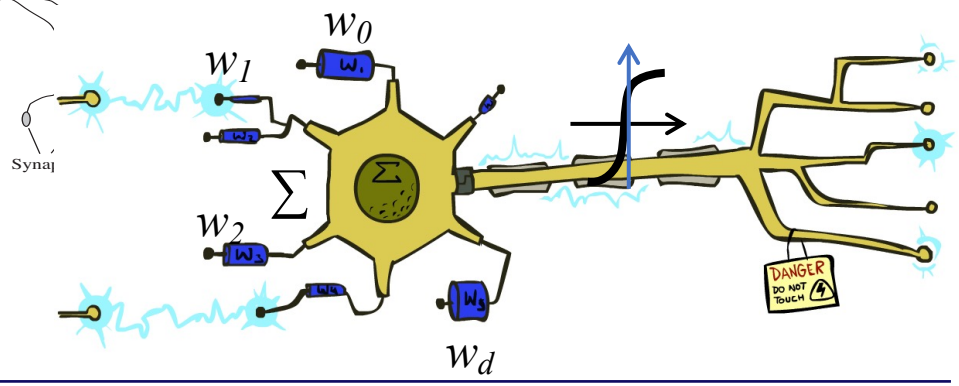
- Cognitive scientists and neuroscientists

aim to understand the functioning of the brain by building models of the natural neural networks.

- AI scientists

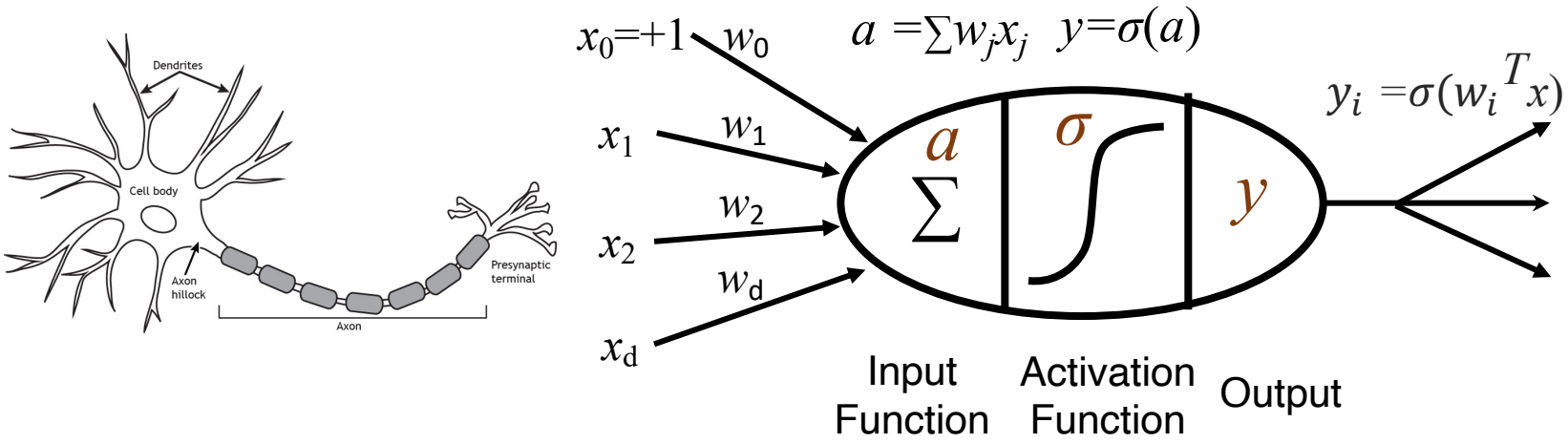
aim to build better computer systems based on inspirations from studying the brain.





Perceptron

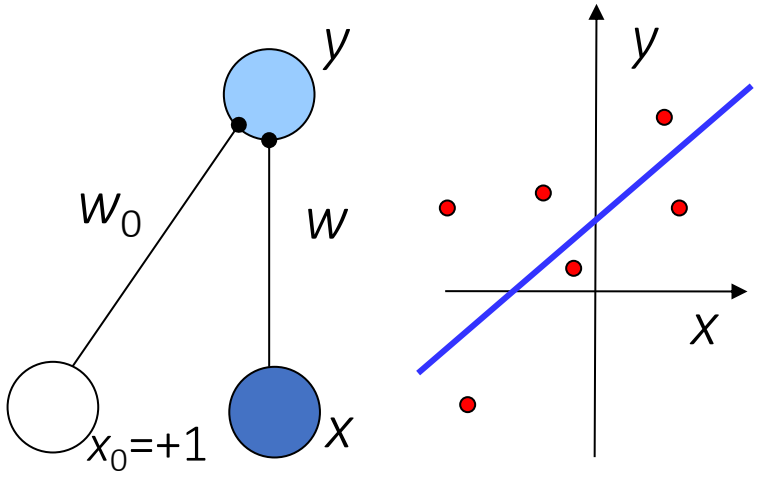
- Basic modeling of "Neuron".

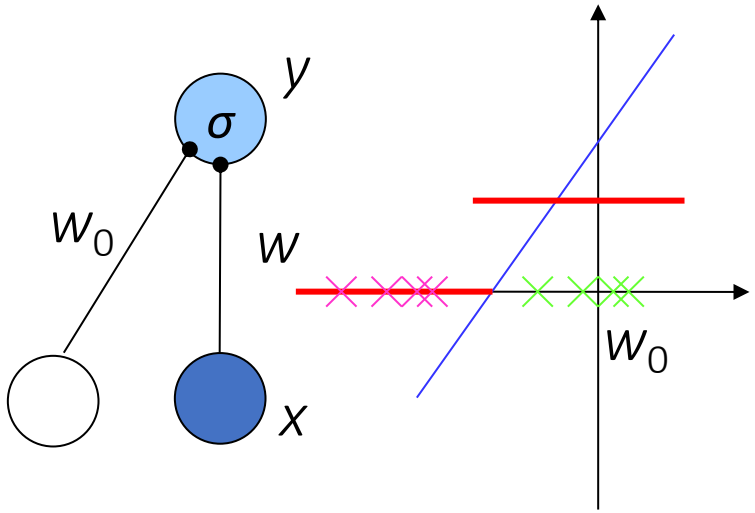


- The output y is an activation of a linear weighted sum of the inputs $x = (x_0,\ldots,x_d)^T$ where x_0 is a special bias unit with x_0=1 and $\boldsymbol{w}=(w_0,\ldots,w_d)^T$ are called the connection weights or synaptic weights.

What a Perceptron Does ?

- Regression: $y=wx+w_0$
- Classification: $y=\text{sign}\,(wx+w_0)$





- To implement a linear classifier, we need the threshold function:

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

to define the following decision rule:

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

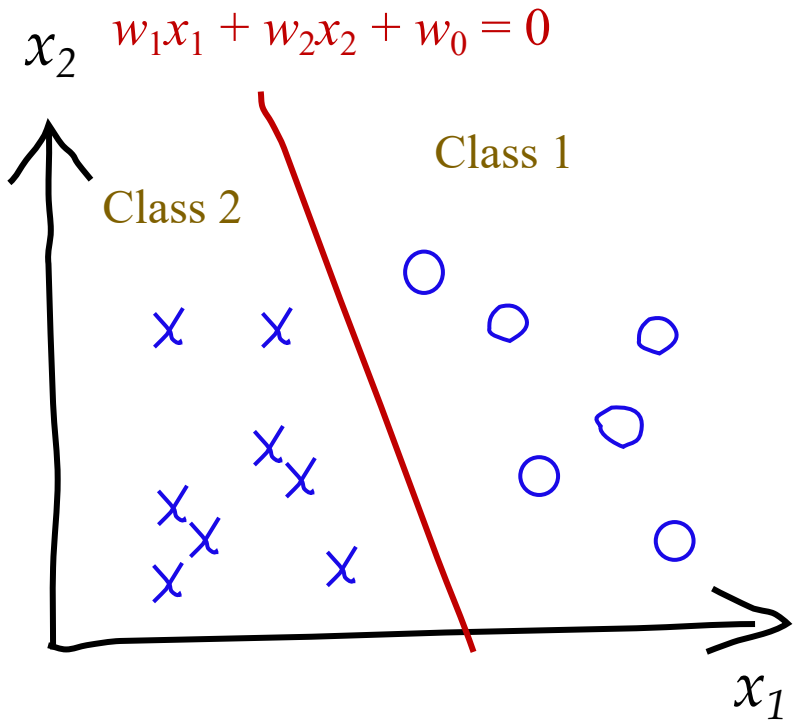
Training a Perceptron

$$w_i = w_i + \eta(y - \hat{y})x_i$$

- Equivalent to rules:
  - If output is correct, do nothing
  - If output is high, lower weights on active inputs
  - If output is low, increase weights on active inputs

Property of Perceptron

- Rosenblatt [1958] proved the convergence of a perceptron learning algorithm if two classes said to be linearly separable (i.e., patterns that lie on opposite sides of a hyperplane)



Property of Perceptron

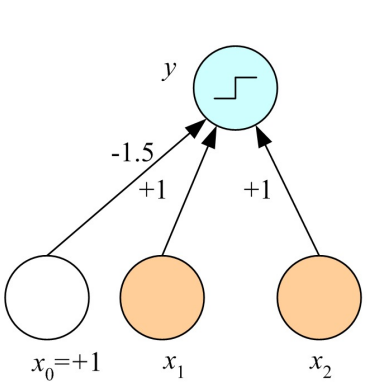
- Perceptron and Boolean functions

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

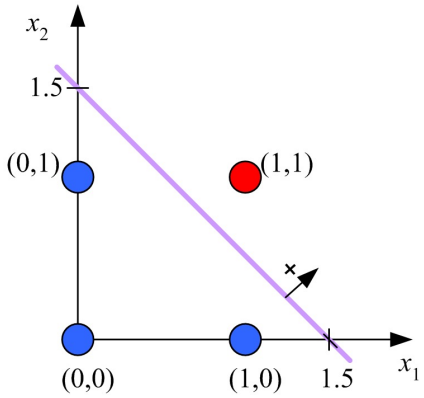
Perceptron for AND and its geometric interpretation:

x_1	x_2	r
0	0	0
0	1	0
1	0	0
1	1	1

AND function with 2 inputs and 1 output



Perceptron for AND



Geometric Interpretation

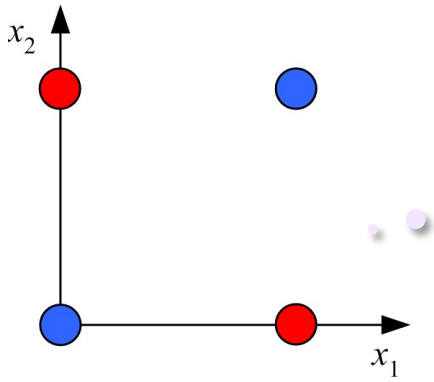
Property of Perceptron

- Perceptron and XOR

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

XOR function with 2 inputs and 1 output

x_1	x_2	r
0	0	0
0	1	1
1	0	1
1	1	0

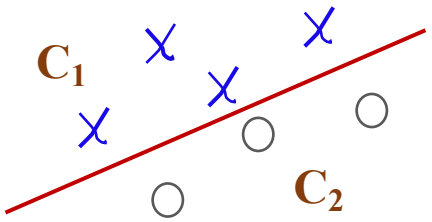


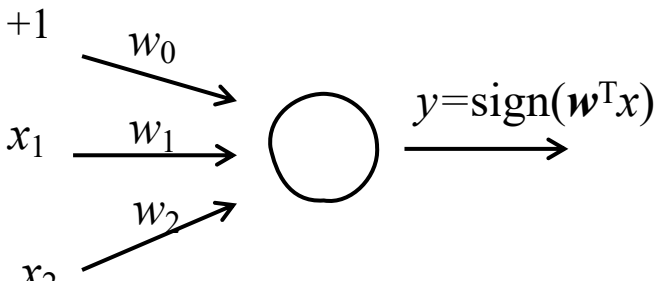
Perceptron cannot even learn XOR ???

Limitation1: perceptron cannot learn data that are not linearly separable

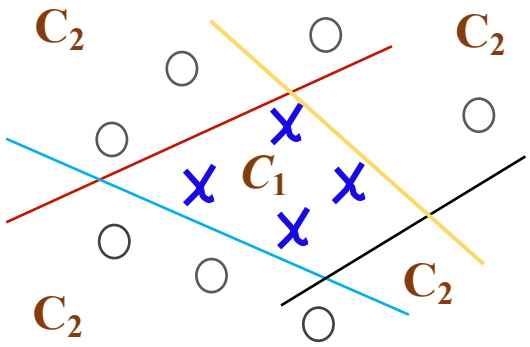
The Magic of Hidden Layers !

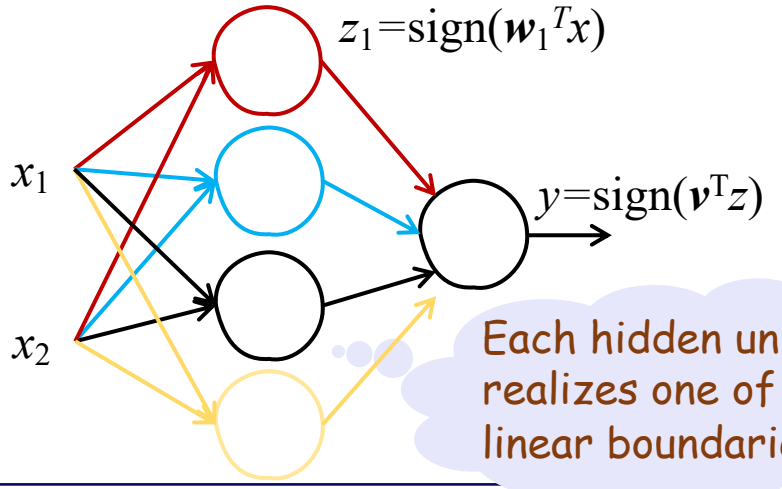
- **But,** adding hidden layer(s) (internal presentation) allows to learn a mapping that is not constrained by linear separability.





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





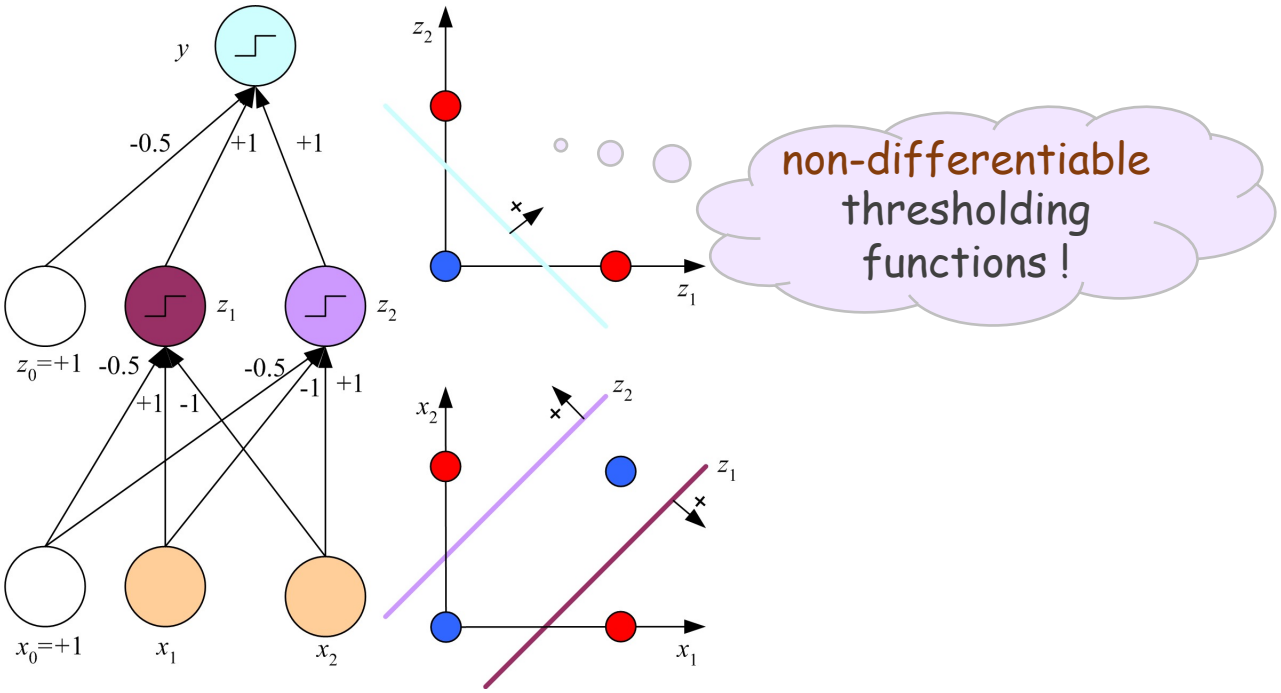
Each hidden unit realizes one of the linear boundaries

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.



Limitation2: thresholding functions are discrete and non-differentiable.

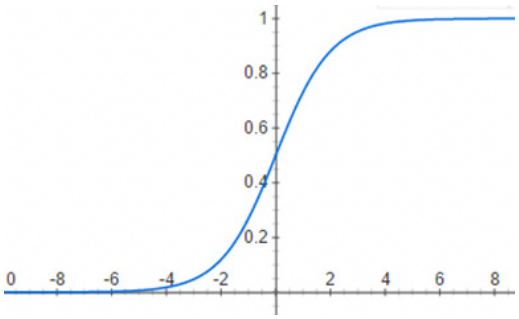
Continuous Thresholding

- Instead of using the threshold function to give a discrete output in {-1,1}, we may use the sigmoid function

$$\text{sigmoid}(a) = \frac{1}{1 + \exp(-a)}$$

to give a continuous output in (0,1):

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



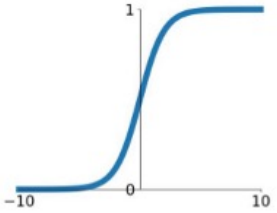
- Sigmoid can be seen as a continuous, differentiable version of thresholding.
- The output may be interpreted as the posterior probability that the input x belongs to C_1.

Activation Functions

In general, we can apply many other continuous thresholding functions, called activation functions, to introduce nonlinearity in perceptrons.

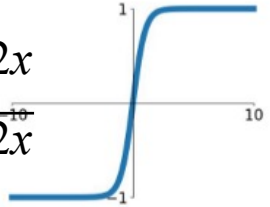
Sigmoid

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



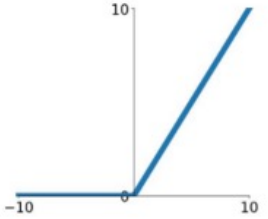
tanh

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



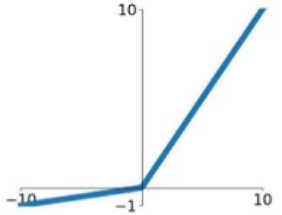
ReLU

$$\max(0, x)$$



Leaky ReLU

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

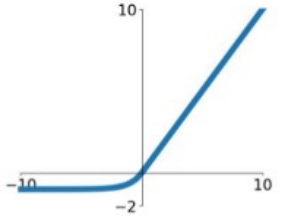


Maxout

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

ELU

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



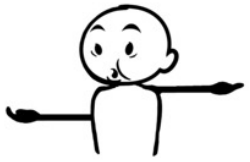
Sigmoid

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

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

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

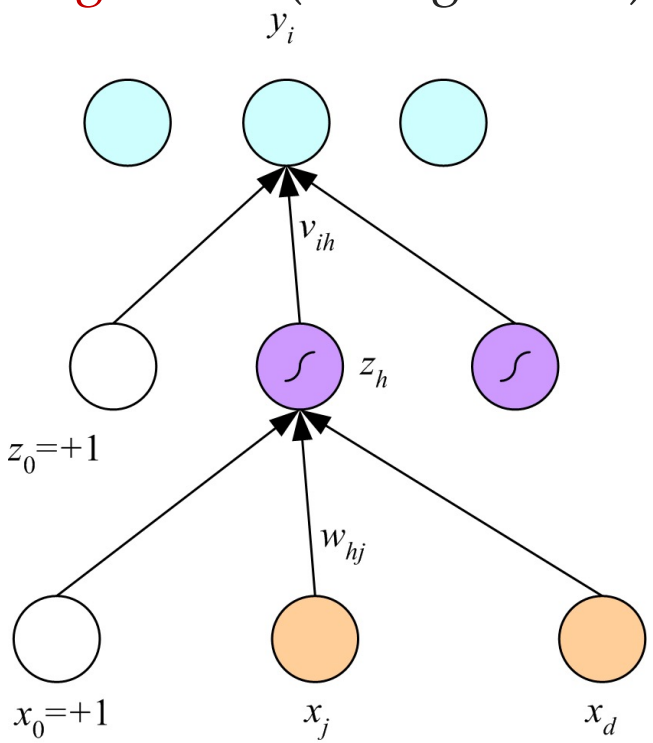
Mish

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

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

Multilayer Perceptrons

- A multilayer perceptron (MLP) has a hidden layer between the input and output layers.
- Can implement nonlinear discriminants (for classification) and regression (for regression) functions.



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.

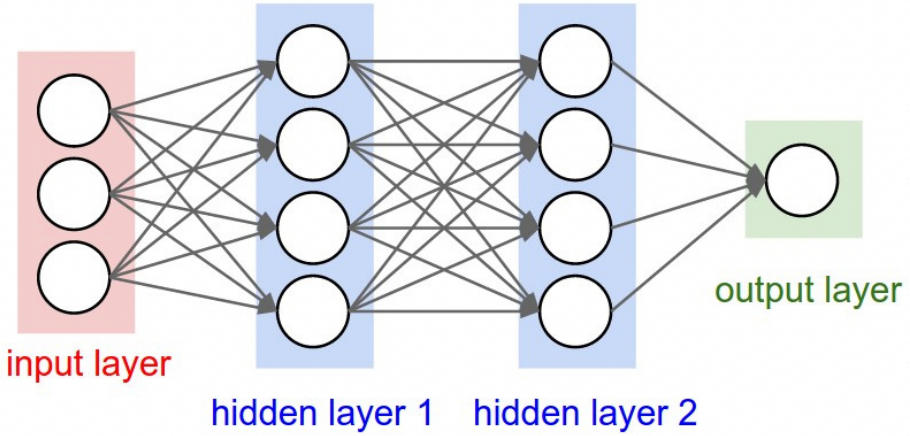
$$z_h = \text{sigmoid}(\mathbf{w}_h^T \mathbf{x}) = \frac{1}{1 + \exp\left[-\left(\sum_{j=1}^{d} w_{hj} x_j + w_{h0}\right)\right]}$$

Input performs no computation.

MLP as a Universal Approximator

- The result for arbitrary Boolean functions can be extended to the continuous case.
- Universal approximation:

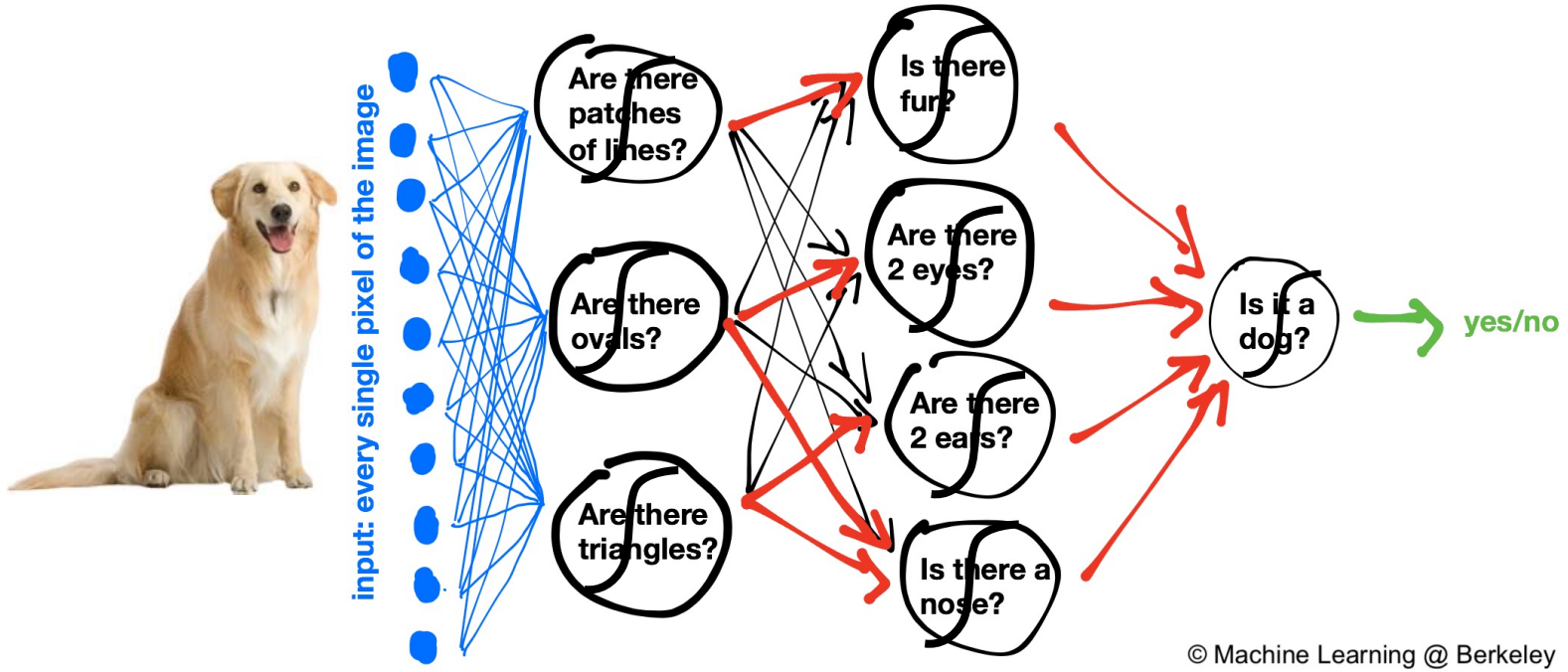
  An MLP with one hidden layer can approximate any nonlinear function of the input given sufficiently many hidden units.



Multi-layer perceptrons approximate **any** continuous functions on compact subset of R^n

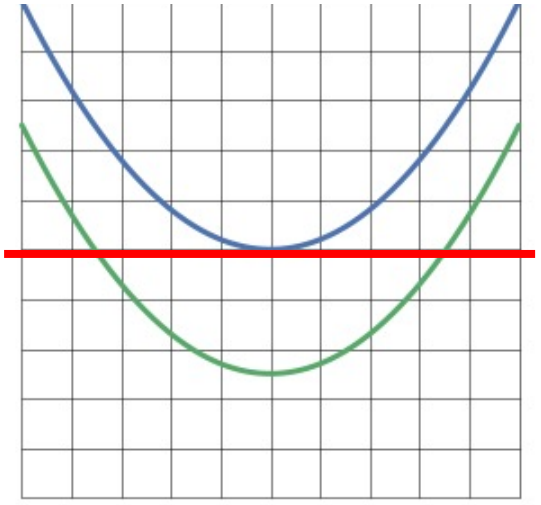
What an MLP does?

Perspective1: Continuous (differentiable) decisions that are easy to optimize.

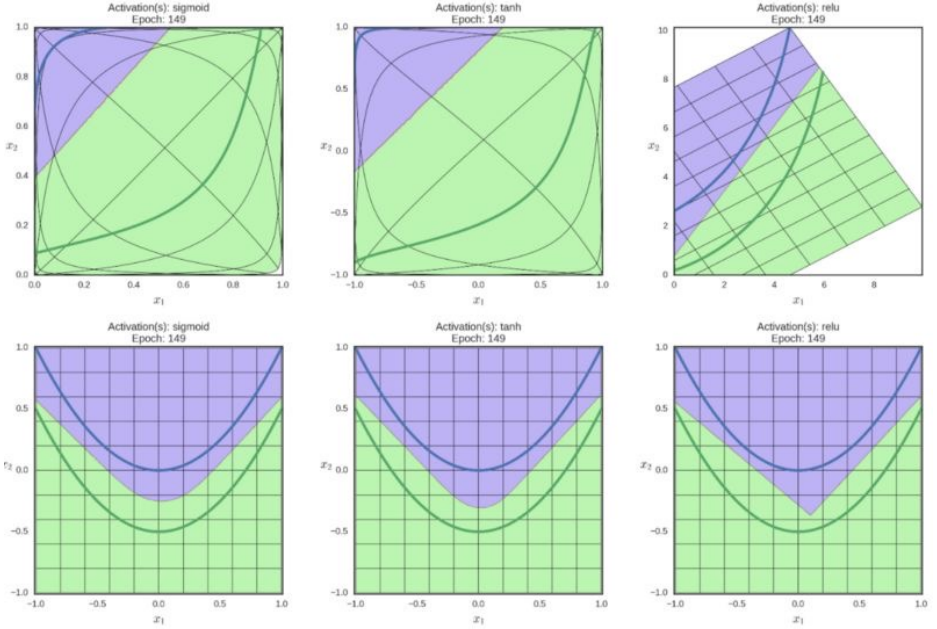


What an MLP does?

Perspective2: Transform the feature space by squeezing, deformation, and . 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

How to train an MLP?

Optimizing MLP

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

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

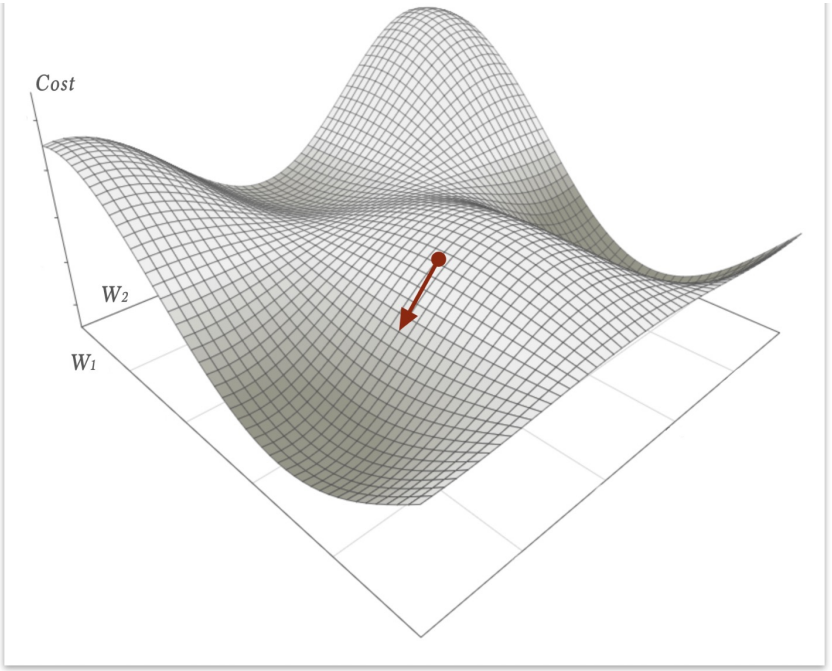
$$\theta^1 = \theta^0 - \eta\nabla L(\theta^0)$$

$$\theta^2 = \theta^1 - \eta\nabla L(\theta^1)$$

$$\vdots$$

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

Millions of Parameters



how to obtain gradients in NN?

Recall: Chain Rule

Case 1

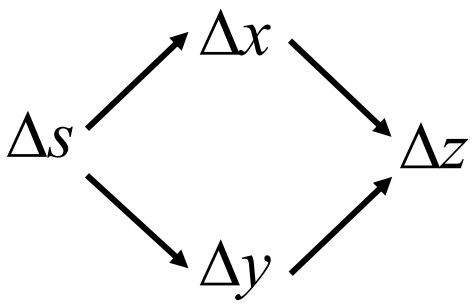
$$y = g(x) \quad z = h(y)$$

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

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

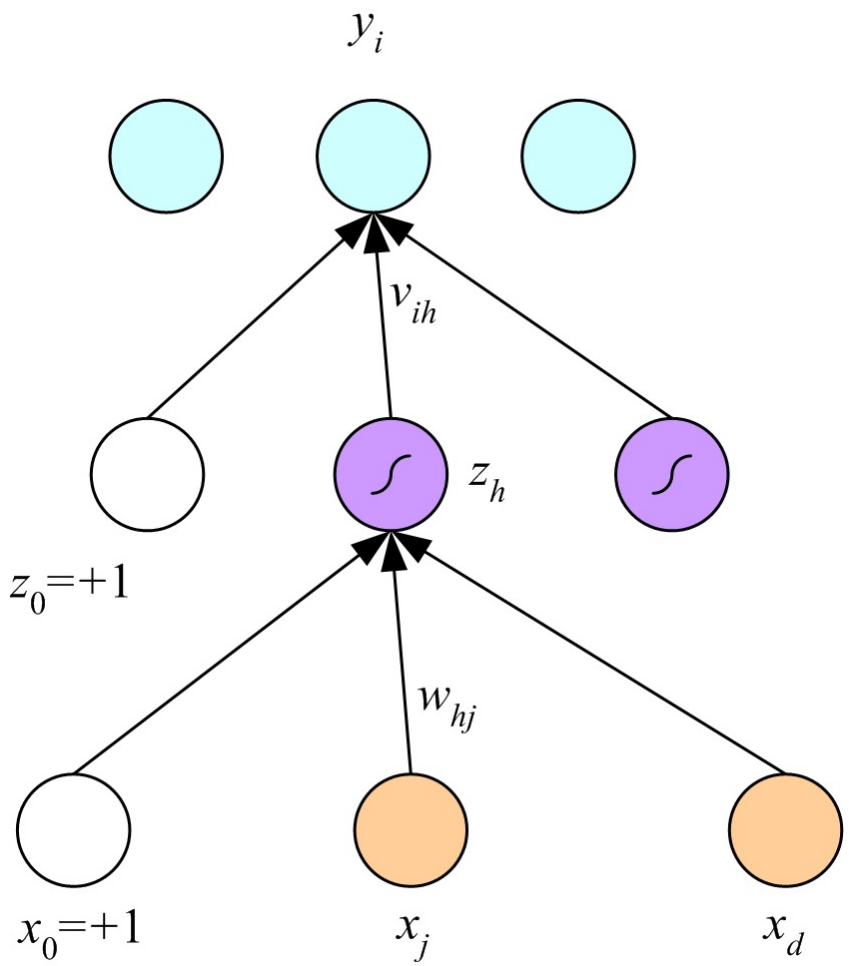
Case 2

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



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

Gradient Descend for 2-Layer MLP (Classification)



$$y_i^{(\ell)} = \frac{\exp(o_i^{(\ell)})}{\sum_k \exp(o_k^{(\ell)})} \equiv P(C_i \mid \mathbf{x}^{(\ell)})$$

$$o_i^{(\ell)} = \sum_{h=0}^{H} V_{ih} z_h^{(\ell)}$$

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

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

$$L(\boldsymbol{W}, \boldsymbol{V} | \chi) = -\sum_{\ell=1}^{N} \sum_{k=1}^{K} r_k^{(\ell)} \log y_k^{(\ell)}$$

Gradient Descend for 2-Layer MLP (Classification)

- Update rule for second-layer weights:

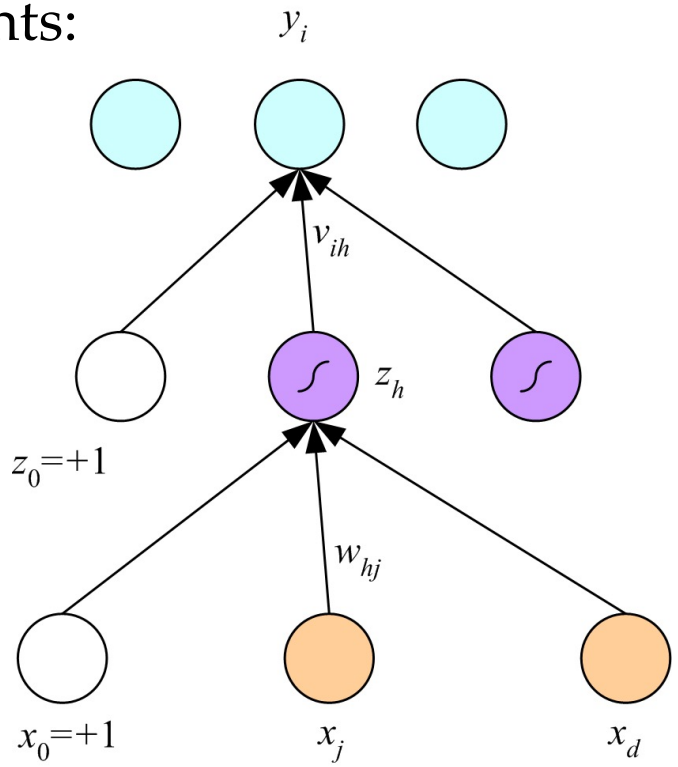
$$\Delta v_{ih} = -\eta \frac{\partial L}{\partial v_{ih}}$$

$$= \eta \sum_{\ell} \sum_{k} \frac{r_k^{(\ell)}}{y_k^{(\ell)}} \frac{\partial y_k^{(\ell)}}{\partial v_{ih}}$$

$$= \eta \sum_{\ell} \sum_{k} \frac{r_k^{(\ell)}}{y_k^{(\ell)}} \frac{\partial y_k^{(\ell)}}{\partial o_i^{(\ell)}} \frac{\partial o_i^{(\ell)}}{\partial v_{ih}}$$

$$= \eta \sum_{\ell} \sum_{k} \frac{r_k^{(\ell)}}{y_k^{(\ell)}} y_k^{(\ell)} (\delta_{ki} - y_i^{(\ell)}) z_h^{(\ell)}$$

$$= \eta \sum_{\ell} \left[\sum_{k} r_k^{(\ell)} (\delta_{ki} - y_i^{(\ell)}) \right] z_h^{(\ell)} \quad = \eta \sum_{\ell} (r_i^{(\ell)} - y_i^{(\ell)}) z_h^{(\ell)}$$



Gradient Descend for 2-Layer MLP (Classification)

- Update rule for first-layer weights:

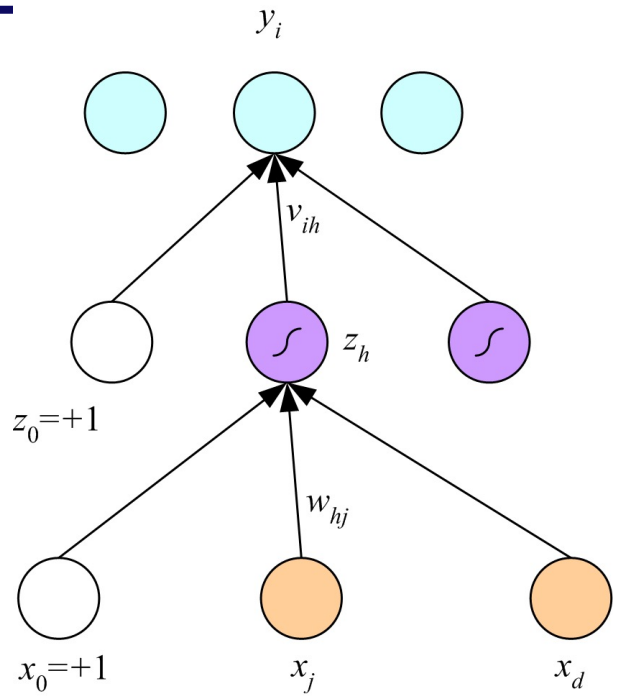
$$\Delta w_{hj} = -\eta \frac{\partial L}{\partial w_{hj}}$$

$$= \eta \sum_{\ell} \sum_{k} \frac{r_k^{(\ell)}}{y_k^{(\ell)}} \frac{\partial y_k^{(\ell)}}{\partial z_h^{(\ell)}} \frac{\partial z_h^{(\ell)}}{\partial w_{hj}}$$

$$= \eta \sum_{\ell} \sum_{k} \sum_{i} \frac{r_k^{(\ell)}}{y_k^{(\ell)}} \frac{\partial y_k^{(\ell)}}{\partial o_i^{(\ell)}} \frac{\partial o_k^{(\ell)}}{\partial z_h^{(\ell)}} \frac{\partial z_h^{(\ell)}}{\partial a_h^{(\ell)}} \frac{\partial a_h^{(\ell)}}{\partial w_{hj}}$$

$$= \eta \sum_{\ell} \sum_{k} \sum_{i} \frac{r_k^{(\ell)}}{y_k^{(\ell)}} y_k^{(\ell)} \left(\delta_{ki} - y_k^{(\ell)}\right) v_{ih} z_h^{(\ell)} \left(1 - z_h^{(\ell)}\right) x_j^{(\ell)}$$

$$= \eta \sum_{\ell} \left[\sum_{i} \left(r_i^{(\ell)} - y_i^{(\ell)}\right) v_{ih} \right] z_h^{(\ell)} \left(1 - z_h^{(\ell)}\right) x_j^{(\ell)}$$



The Algorithm

Gradient Descend for 2-Layer MLP

Initialize all v_{ih} and w_{hj} to rand$(-0.01, 0.01)$

Repeat

  For all $(\boldsymbol{x}^t, r^t) \in \mathcal{X}$ in random order

    For $h = 1, \ldots, H$

      $z_h \leftarrow \text{sigmoid}(\boldsymbol{w}_h^T \boldsymbol{x}^t)$

    For $i = 1, \ldots, K$

      $y_i = \boldsymbol{v}_i^T \boldsymbol{z}$

    For $i = 1, \ldots, K$

      $\Delta \boldsymbol{v}_i = \eta(r_i^t - y_i^t)\boldsymbol{z}$

    For $h = 1, \ldots, H$

      $\Delta \boldsymbol{w}_h = \eta(\sum_i (r_i^t - y_i^t) v_{ih}) z_h (1 - z_h) \boldsymbol{x}^t$

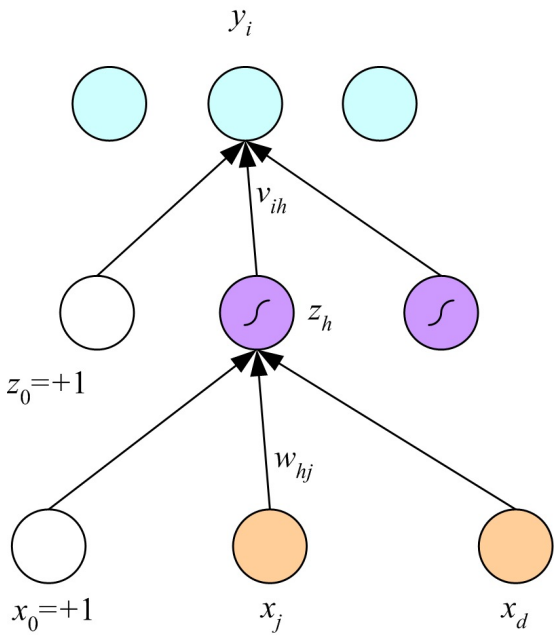
    For $i = 1, \ldots, K$

      $\boldsymbol{v}_i \leftarrow \boldsymbol{v}_i + \Delta \boldsymbol{v}_i$

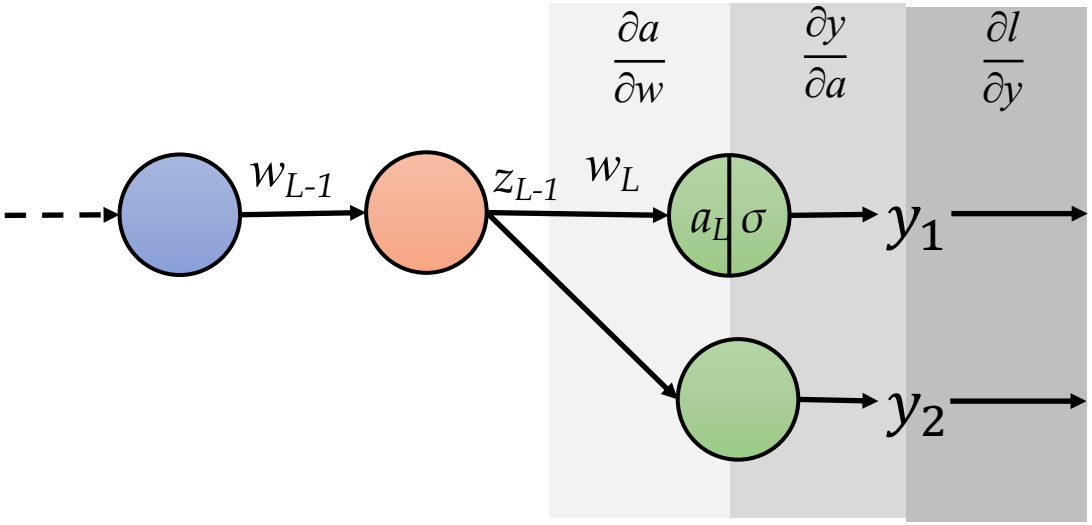
    For $h = 1, \ldots, H$

      $\boldsymbol{w}_h \leftarrow \boldsymbol{w}_h + \Delta \boldsymbol{w}_h$

Until convergence



Gradient Descend for L>2 Layers?



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

$$\frac{\partial l}{\partial w_{L-1}} = ?$$

$\vdots$

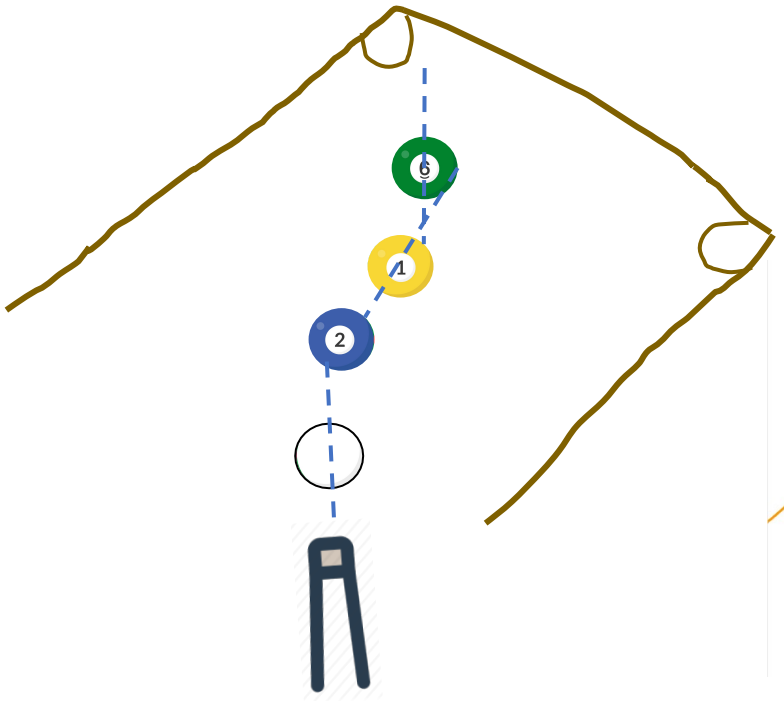
Problems of deriving $\nabla_w L$ directly:

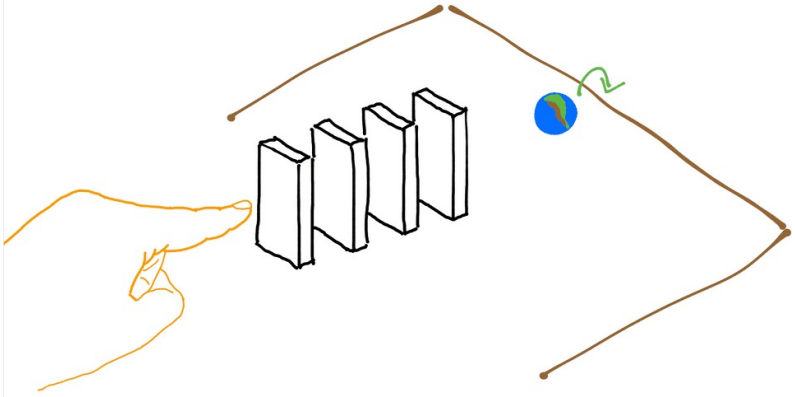
- Very tedious: lots of matrix calculus, need lots of paper …
- What if we want to change loss? E.g. use SVM instead of softmax? Need to re-derive from scratch.
- Not feasible for very complex models!

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

Any New Ideas ?

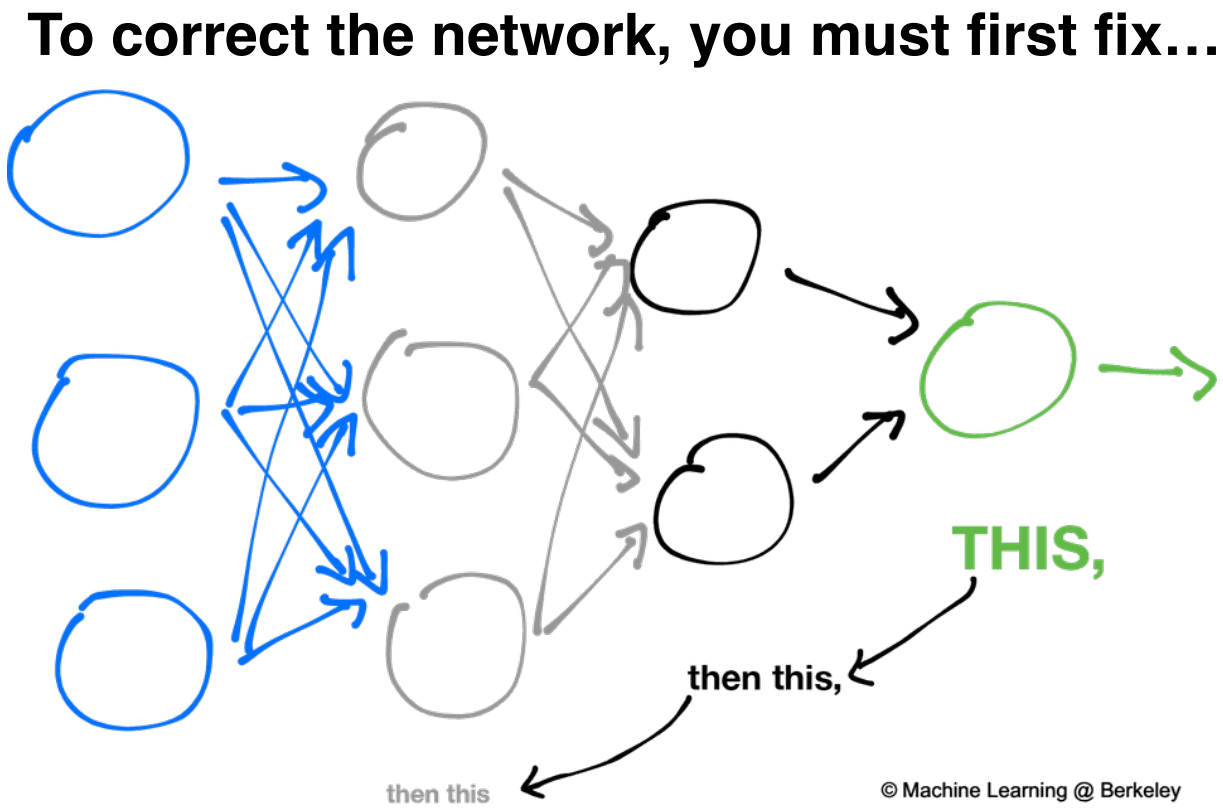
- Straightforward computation of gradients in MLP is computationally infeasible, due to the dependences between gradients.





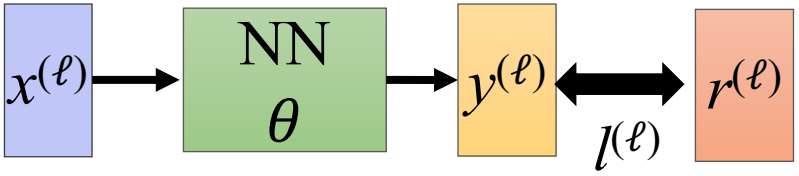
Backpropagation

- A computationally efficient approach for computing derivatives

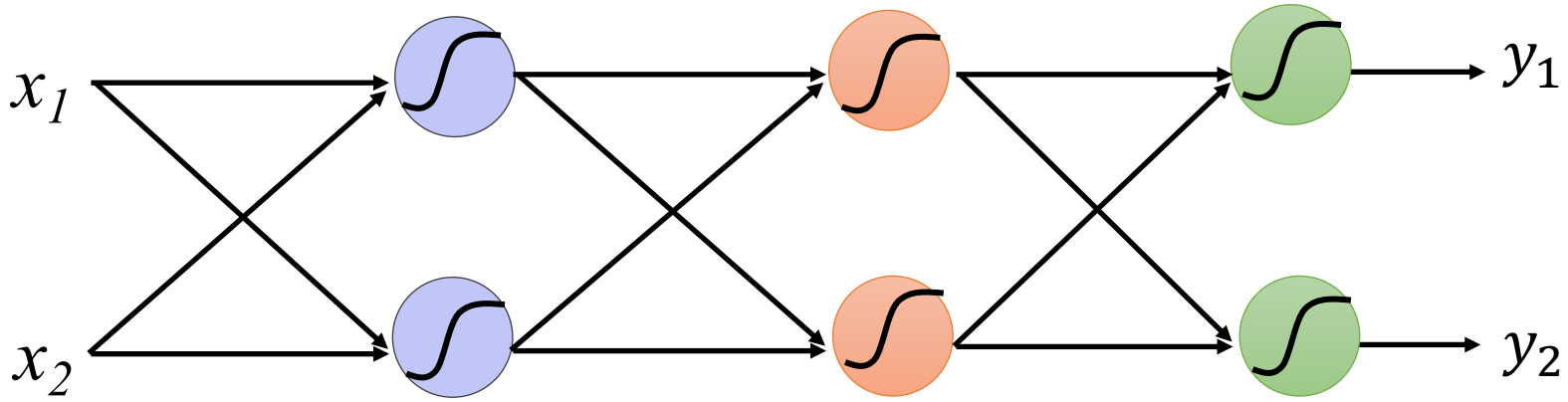


Backpropagation

- Consider the loss function for a single sample

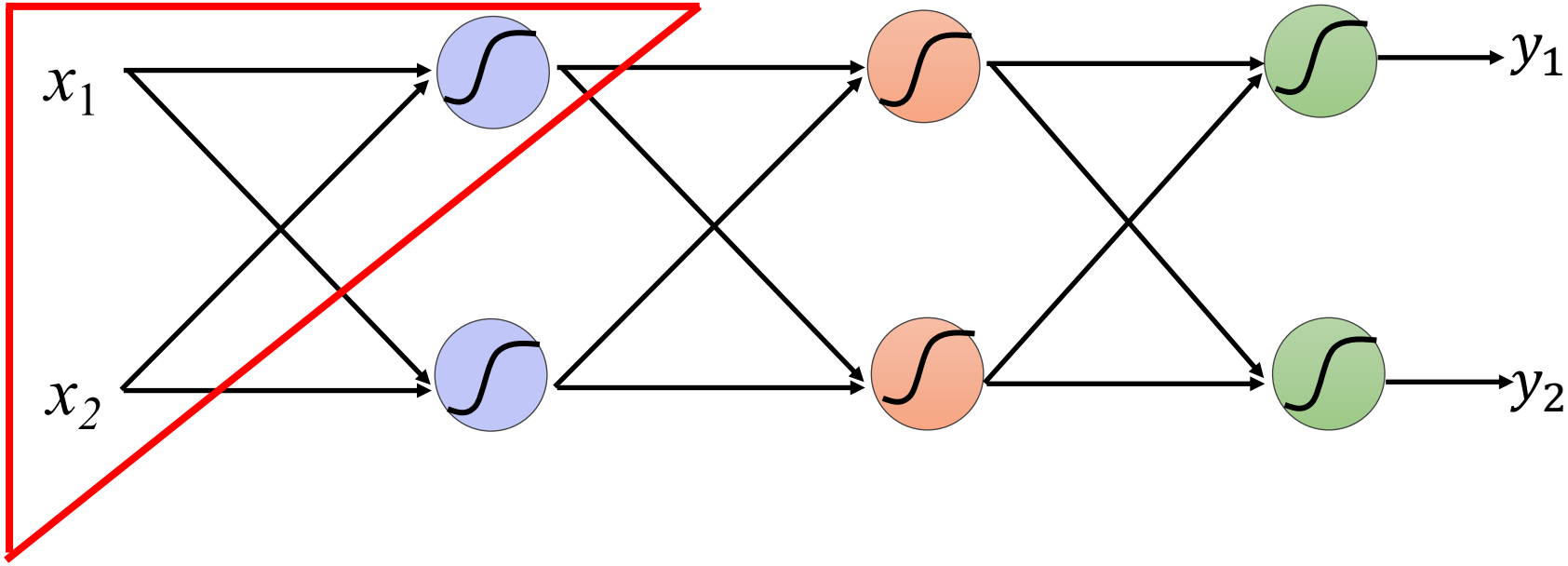


$$L(\theta)=\sum_{\ell=1}^{N} l^{(\ell)}(\theta) \quad \Rightarrow \quad \frac{\partial L(\theta)}{\partial w}=\sum_{\ell=1}^{N}\frac{\partial l^{(\ell)}(\theta)}{\partial w}$$

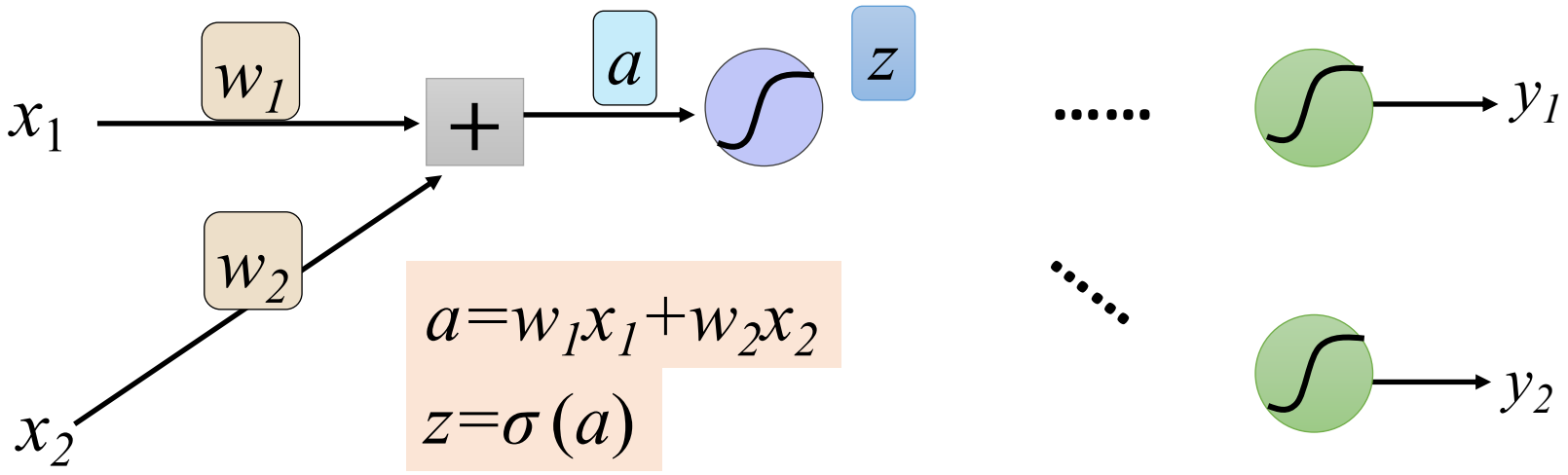


Backpropagation

Initially, consider the first neuron



Backpropagation



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

(Chain rule)

Forward pass:

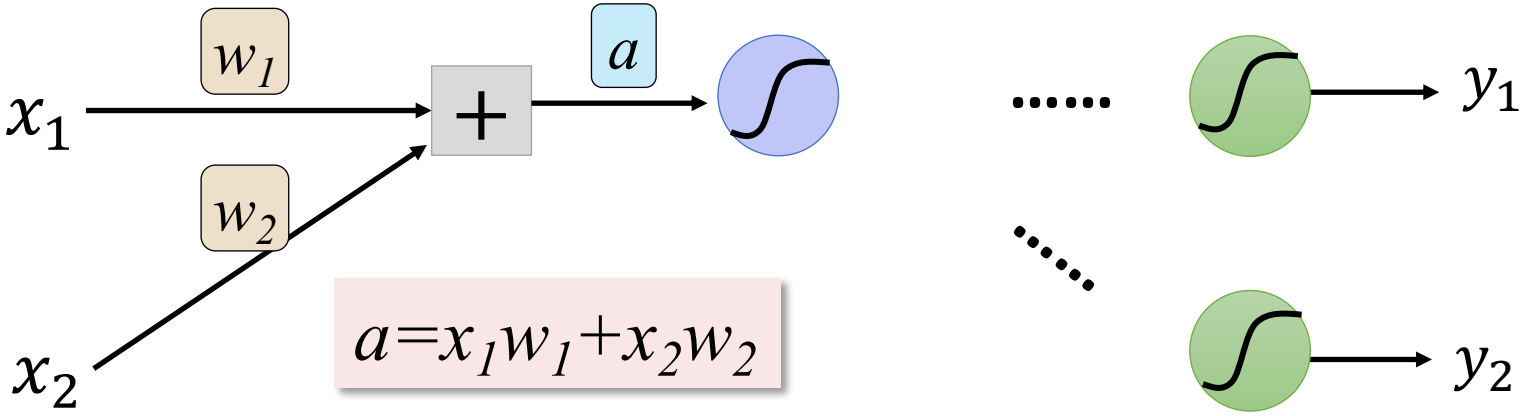
Compute $\partial a/\partial w$ for all parameters $\boldsymbol{w}$

Backward pass:

Compute $\partial l/\partial a$ for all linearity outputs a.

Backpropagation – Forward pass

Compute $\partial a/\partial w$ for all parameters within each linearity unit.



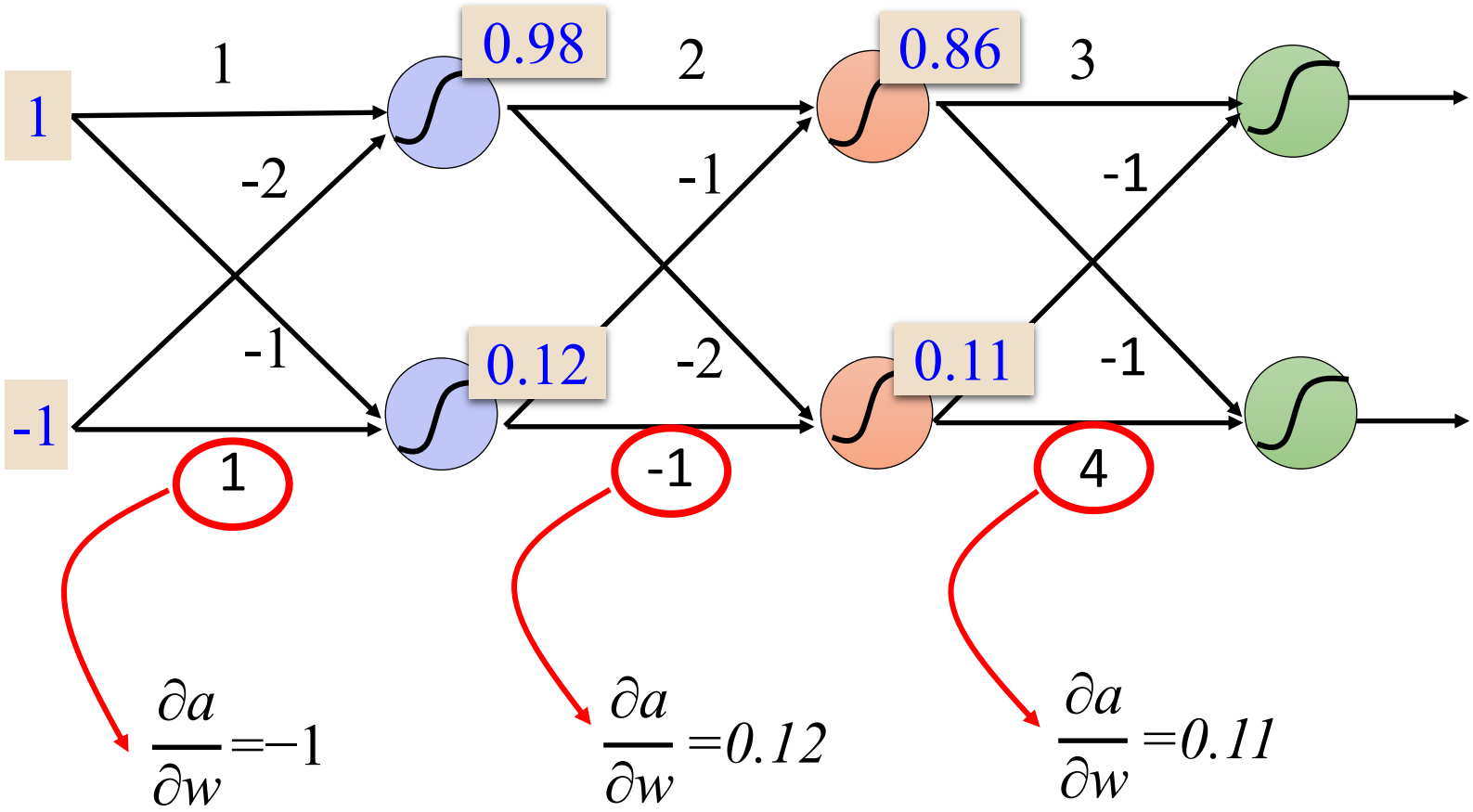
$$\partial a/\partial w_1 = ? \quad x_1$$

$$\partial a/\partial w_2 = ? \quad x_2$$

The input value that is connected to the weight

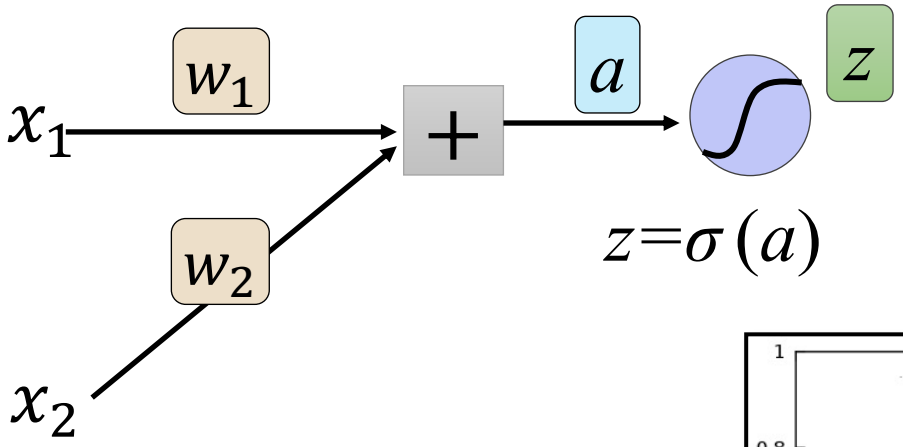
Backpropagation – Forward Pass

Example



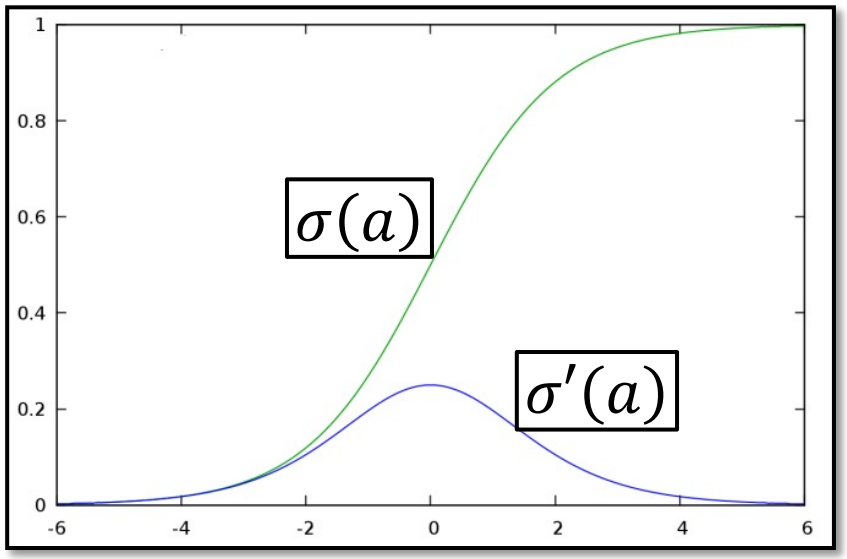
Backpropagation – Backward Pass

Compute $\partial l/\partial a$ for all linearity outputs a.



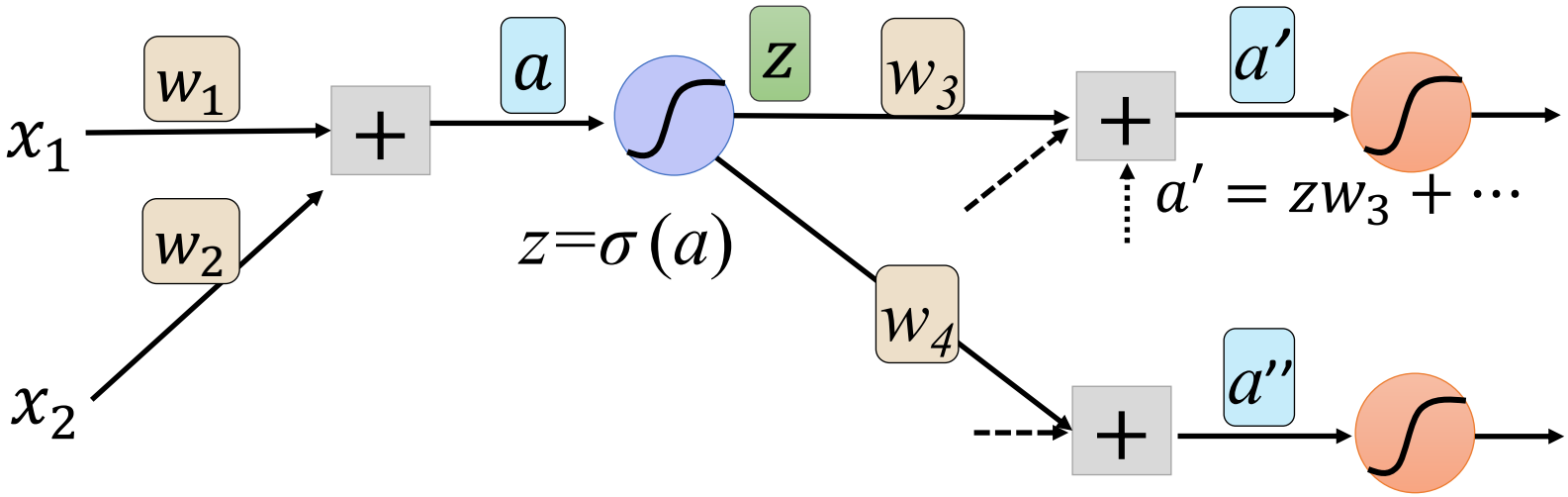
$$\frac{\partial l}{\partial a} = \frac{\partial z}{\partial a} \cdot \frac{\partial l}{\partial z}$$

$$\Rightarrow \sigma'(a)$$



Backpropagation – Backward Pass

Compute $\partial l/\partial z$ for all linearity outputs a.



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

$$\frac{\partial l}{\partial z} = \frac{\partial a'}{\partial z}\frac{\partial l}{\partial a'} + \frac{\partial a''}{\partial z}\frac{\partial l}{\partial a''}$$

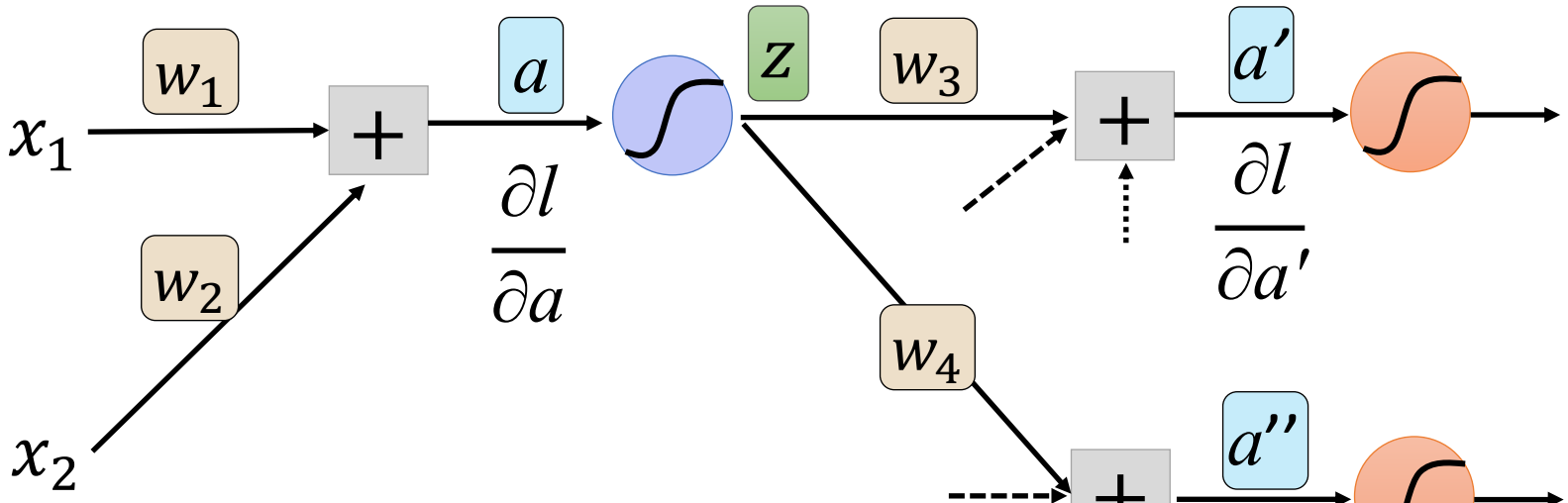
(Chain rule)

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

Recursion?

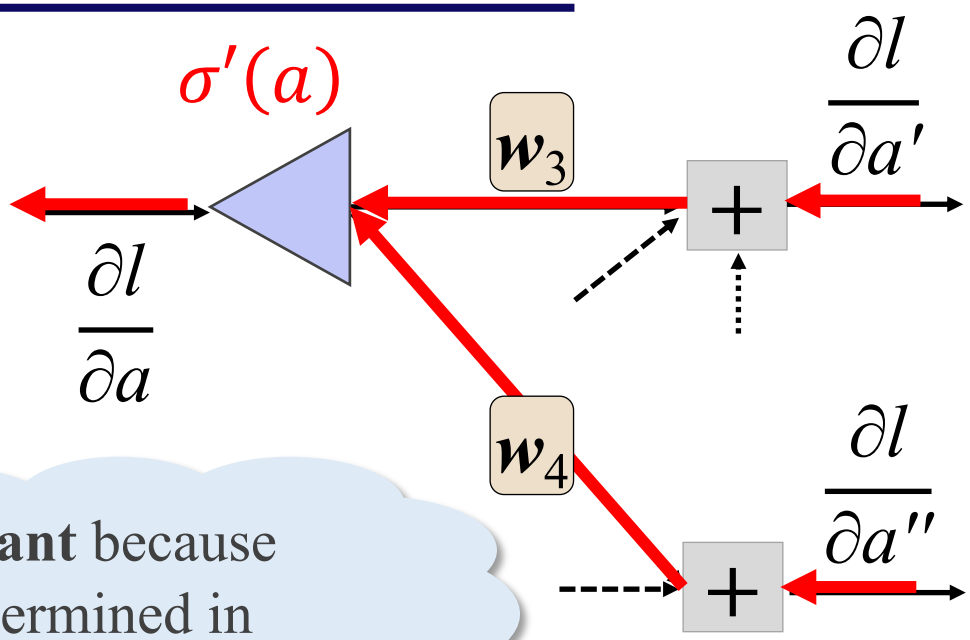
Backpropagation – Backward Pass

Compute $\partial l/\partial a$ for all linearity outputs a.



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

Backpropagation – Backward pass

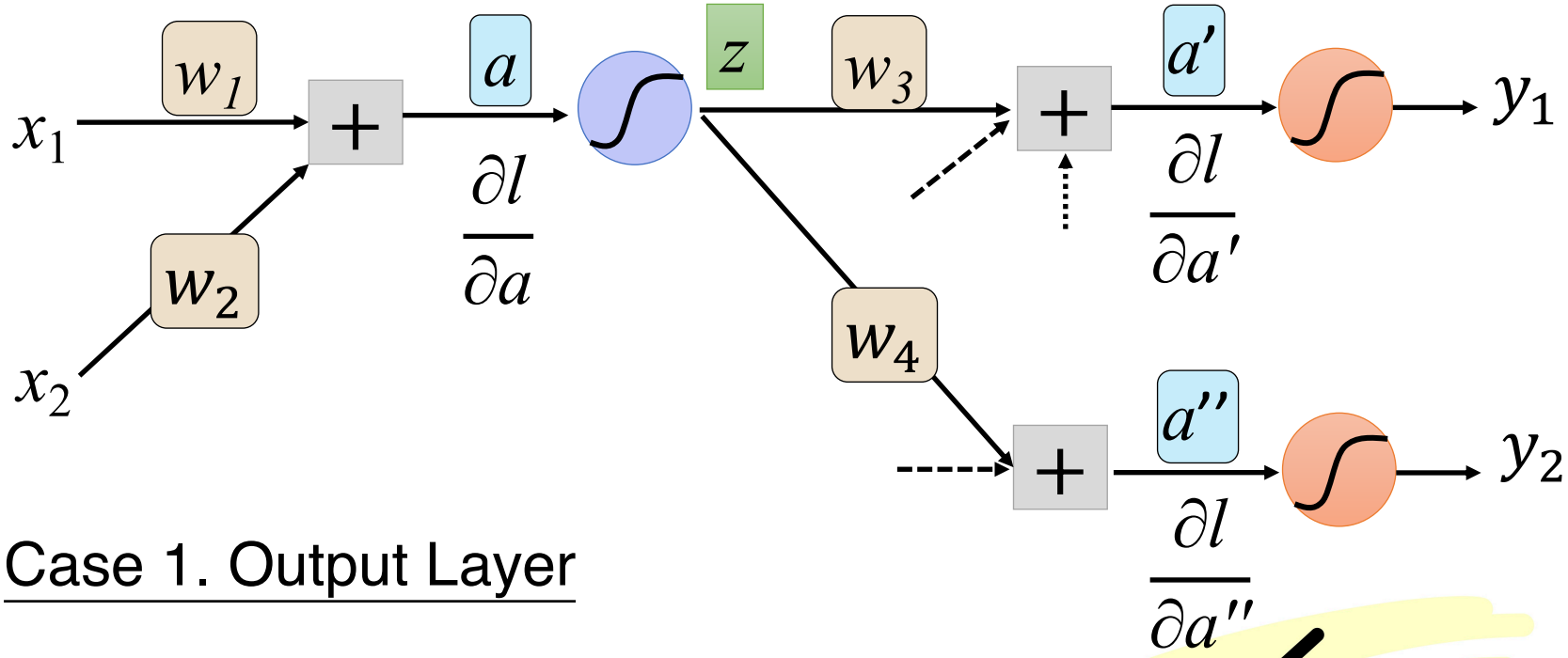


$\sigma'(a)$ is a **constant** because a is already determined in the forward pass.

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

Backpropagation – Backward Pass

Compute $\partial l/\partial a$ for all linearity outputs a.



Case 1. Output Layer

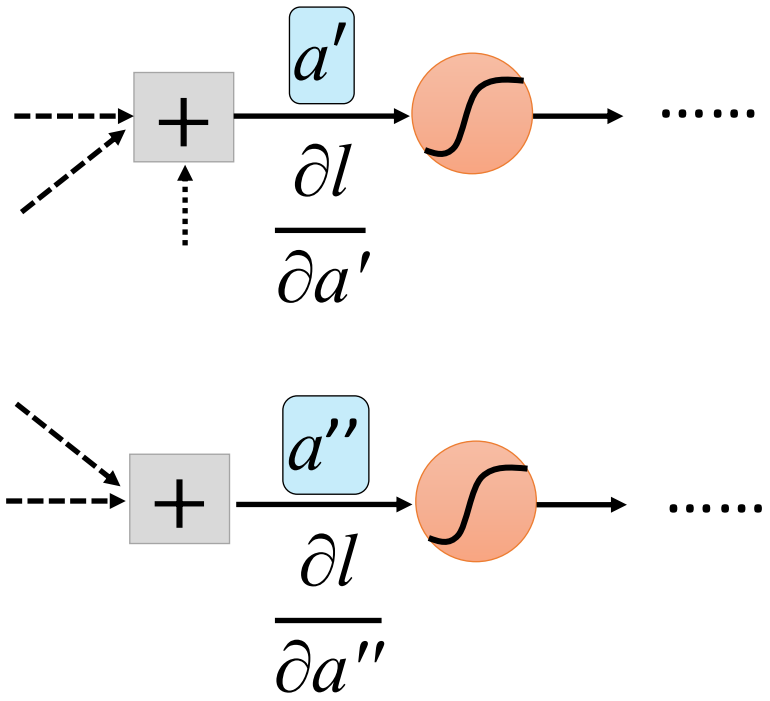
$$\frac{\partial l}{\partial a'} = \frac{\partial y_1}{\partial a'}\frac{\partial l}{\partial y_1} \qquad \frac{\partial l}{\partial a''} = \frac{\partial y_2}{\partial a''}\frac{\partial l}{\partial y_2}$$

DONE

Backpropagation – Backward pass

Compute $\partial l/\partial a$ for all linearity outputs a.

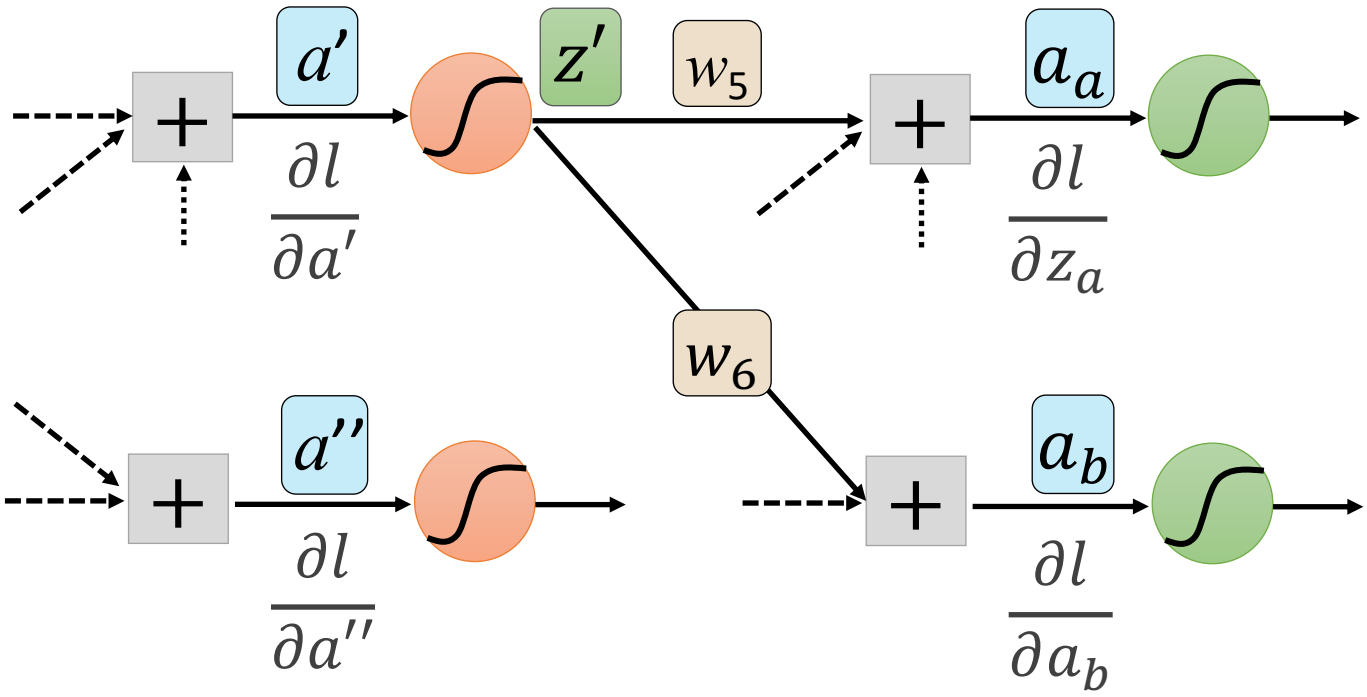
Case 2. Not Output Layer



Backpropagation – Backward pass

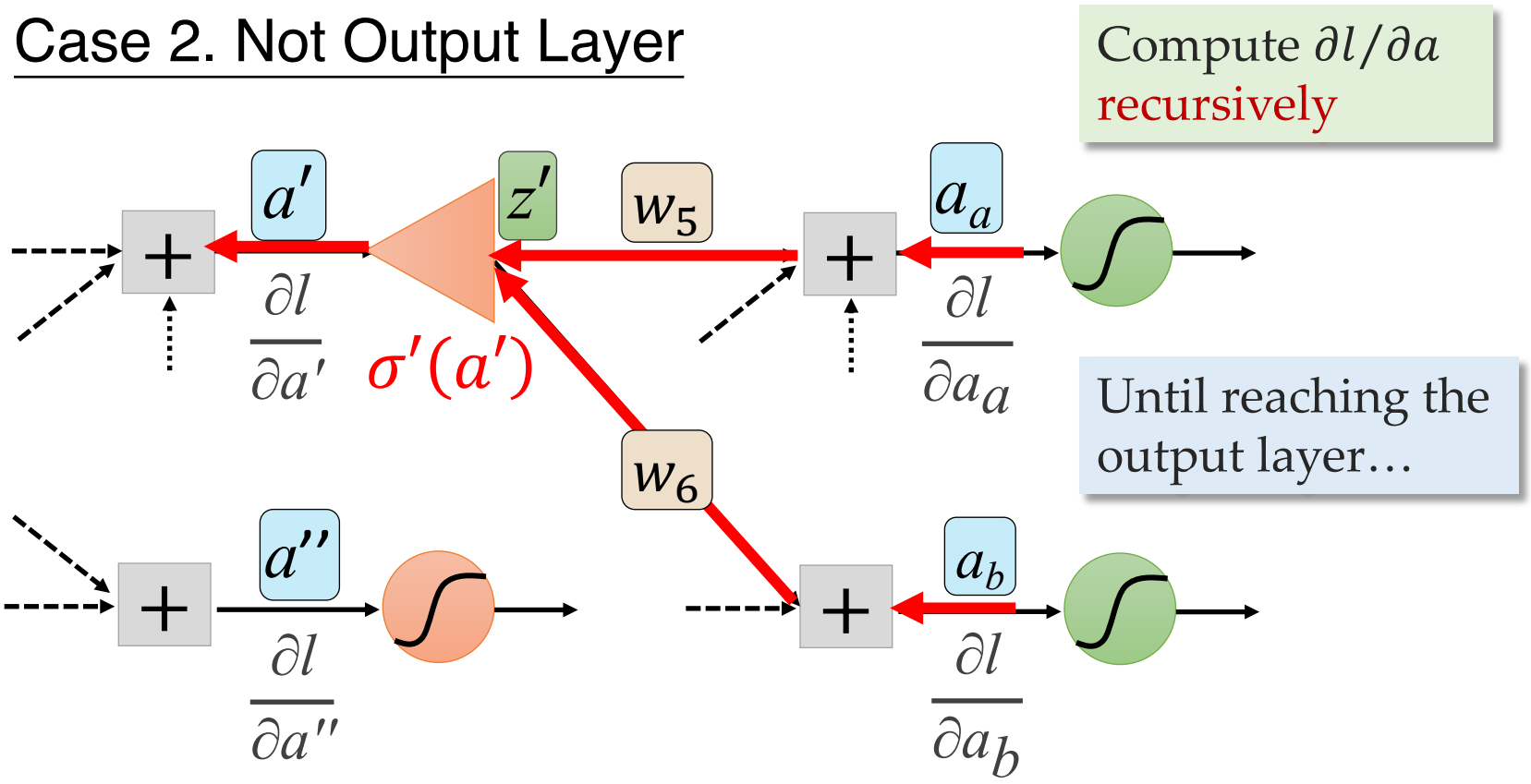
Compute $\partial l/\partial a$ for all linearity outputs a.

Case 2. Not Output Layer



Backpropagation – Backward pass

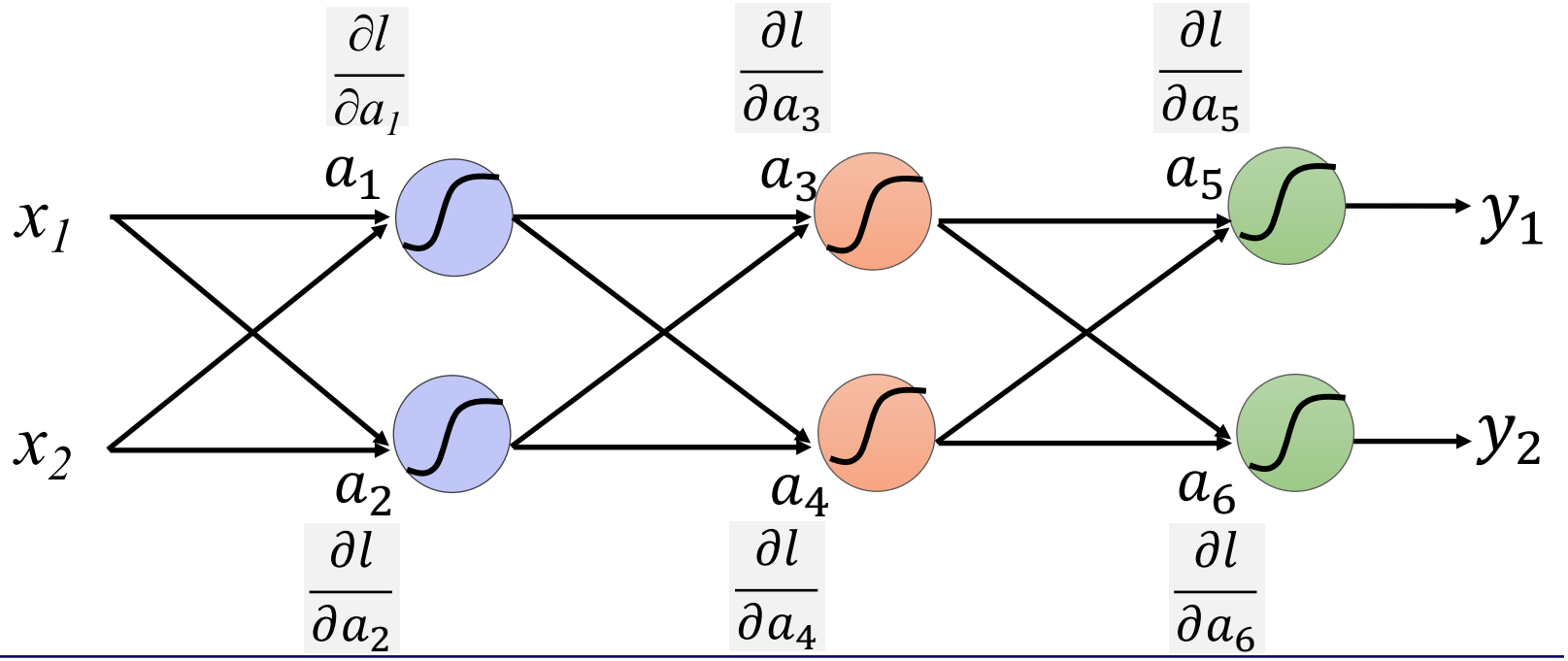
Compute $\partial l/\partial a$ for all linearity outputs a.



Backpropagation – Backward Pass

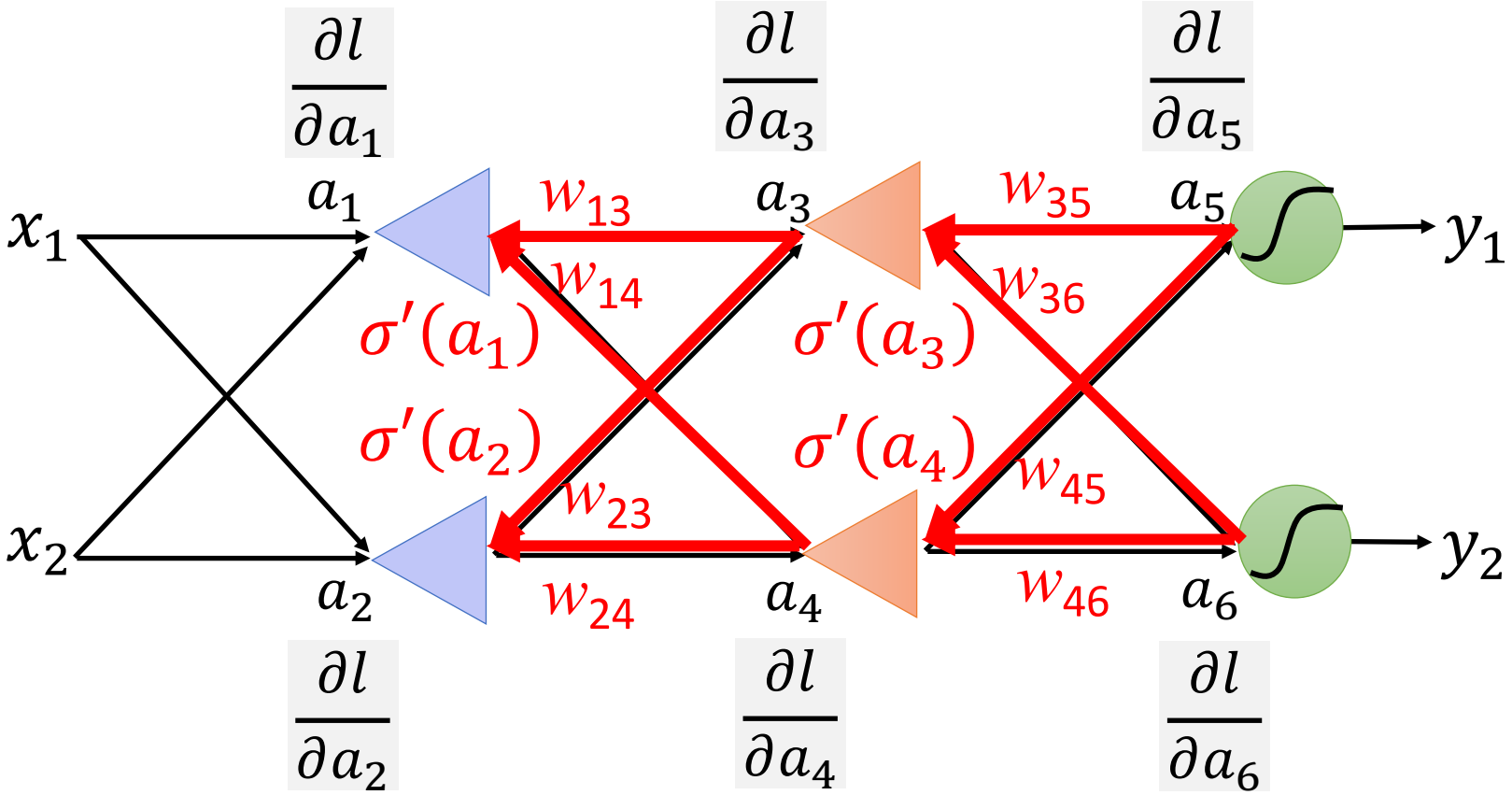
Recursion ⇨ Dynamic Programming

We can compute $\partial l/\partial a$ from the output layer **backward** to the input layer.

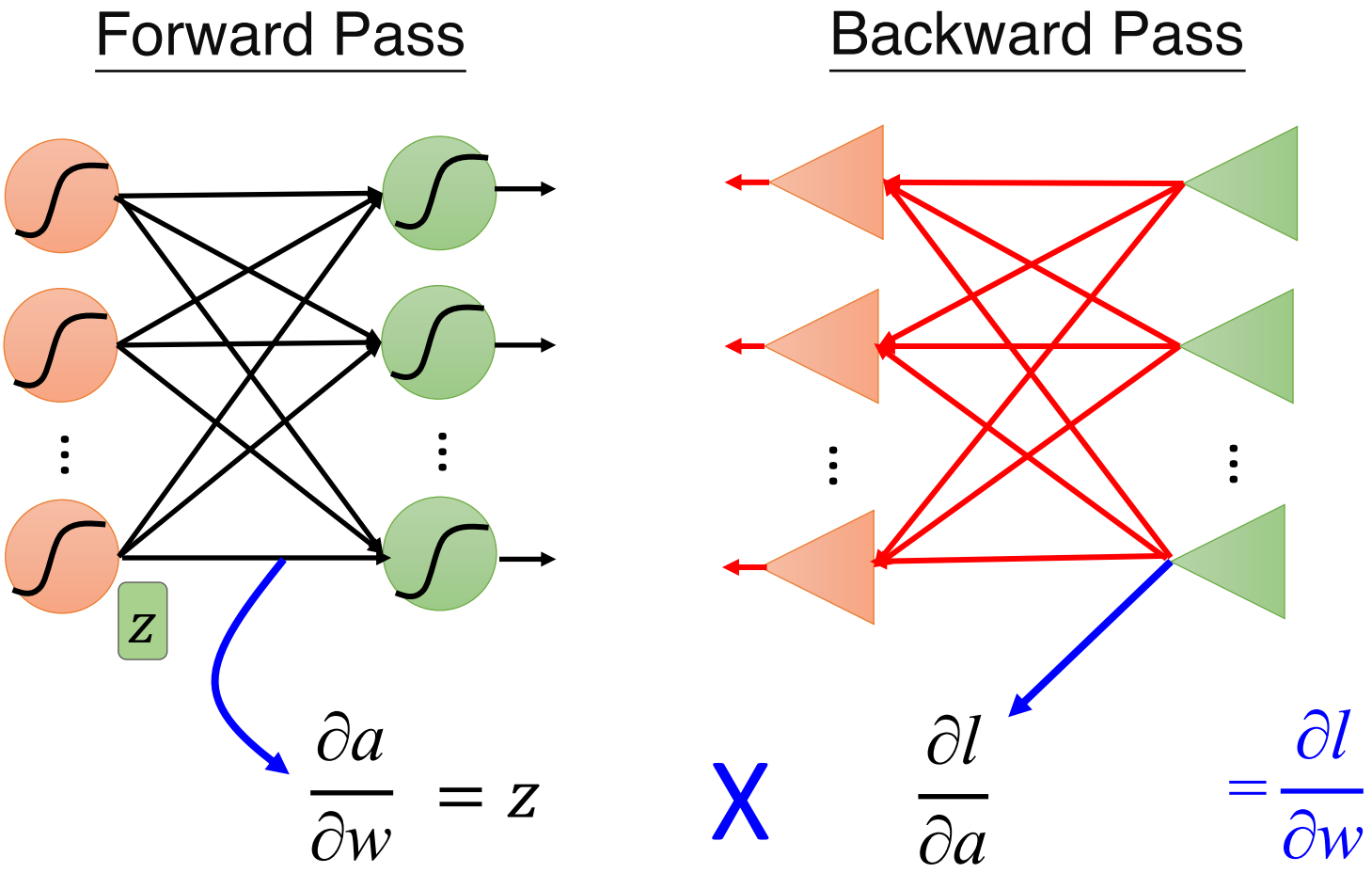


Backpropagation – Backward Pass

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



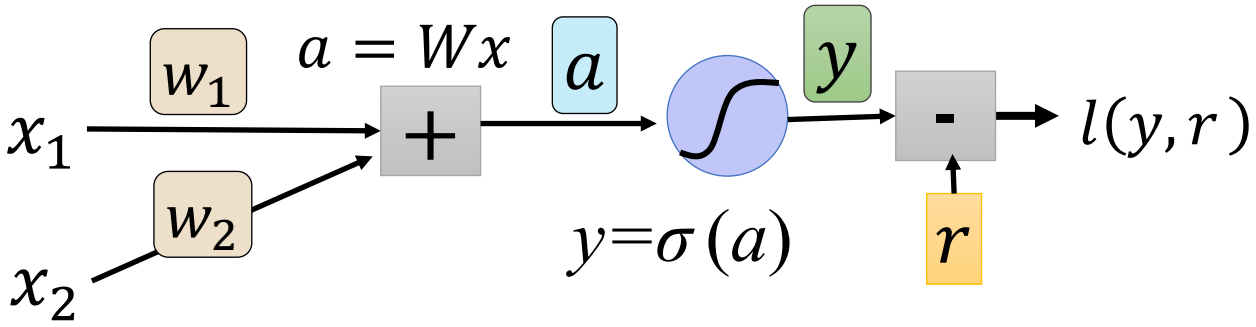
Backpropagation – Summary



Perspective from Error-Backpropagation

Recall: Logistic Regression

$$y = \sigma(\ W\ x\ )$$



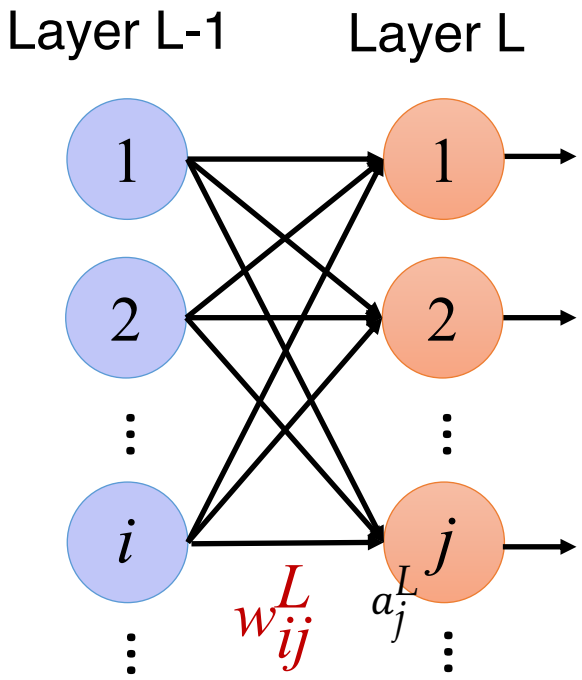
$$\frac{\partial L}{\partial w_j} = -\sum_l (r^{(l)} - y^{(l)})\, x_j^{(l)}$$

error input

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

Perspective from Error-Propagation

Update to each weight is the product of the error and the input (signal).



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

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

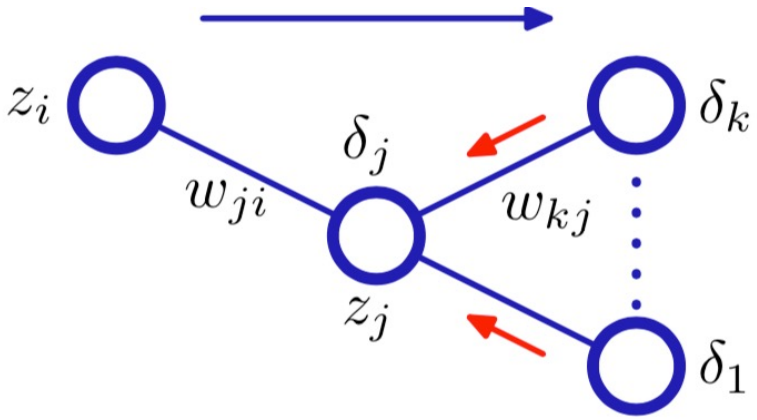
δ_j^L

?

Let $\frac{\partial l}{\partial a_j} \equiv \delta_j$ be the Error Signal for all w_{ij}

Perspective from Error-Propagation

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



- Recursively,

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

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

$=\delta_k$

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

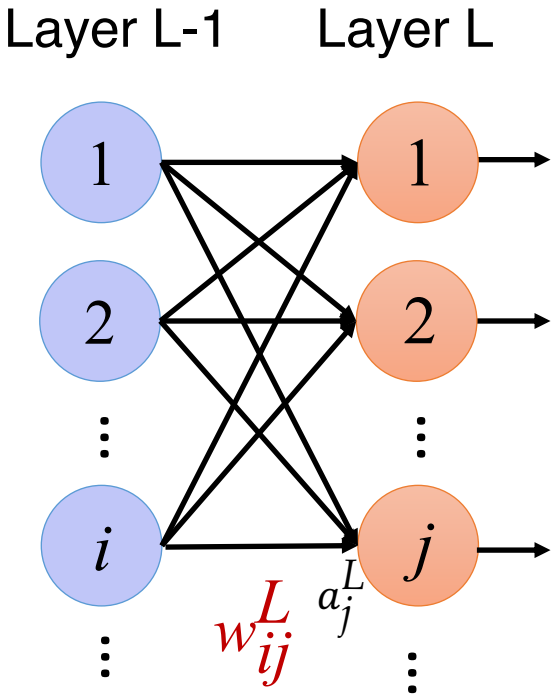
"Amplifier"

- 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

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

Error signal



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

δ_j^{L}

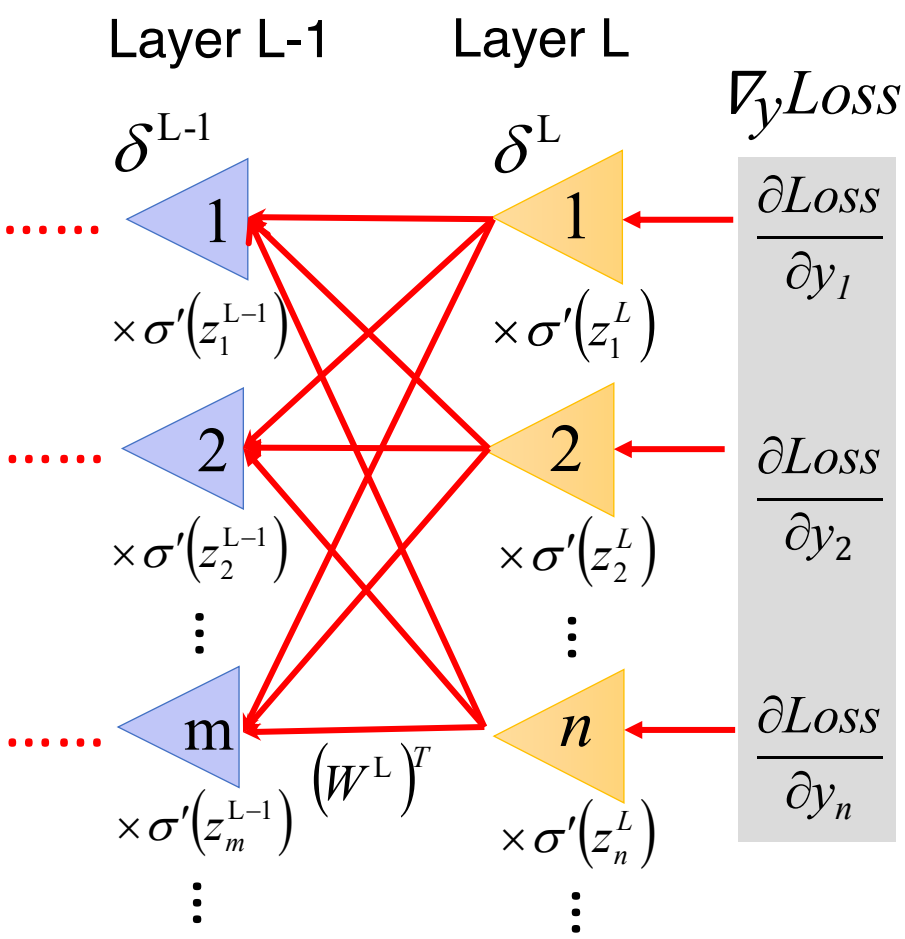
Forward Pass

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

Backward Pass

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

Perspective from Error-Propagation



(we do not use softmax here)

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

error signal

$$\delta_j^L$$

Backward Pass

$$\delta^L = \sigma'(zL) \bullet \nabla_y Loss$$

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

……

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

……

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:

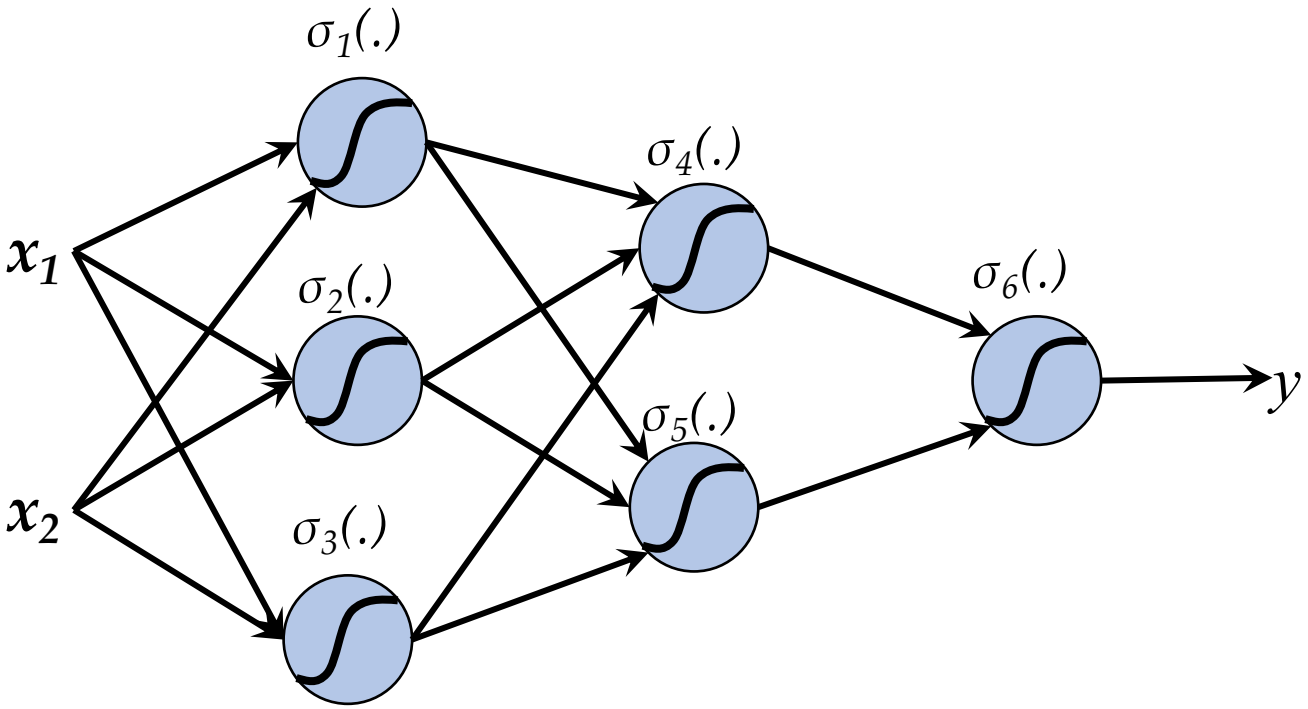
- evaluate the derivatives of the loss function with respect to the weights (errors).
- errors are propagated backwards through the network. $\delta_j^L = \frac{\partial l}{\partial a_j^L}$

3. Parameter update:

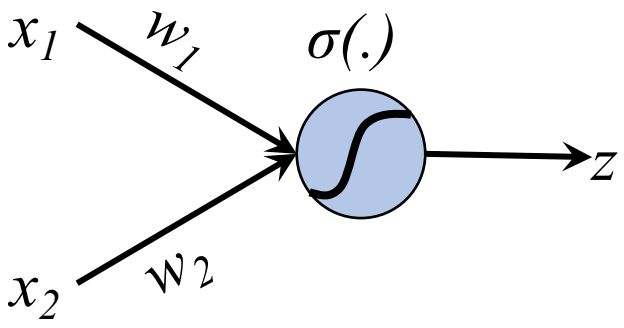
- the evaluated derivatives (errors) are then used to compute the adjustments to be made to the parameters.

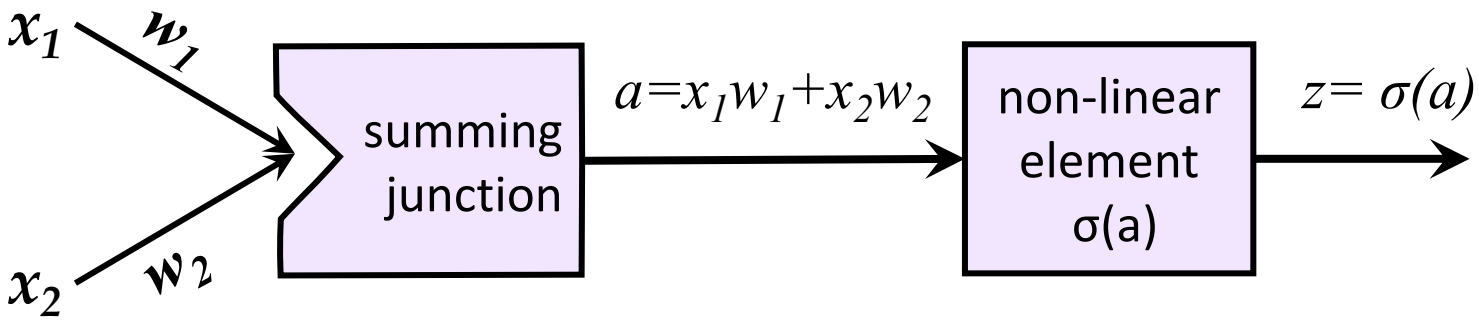
$$w'_{ij} = w_{ij} + \eta \delta_j \sigma'_j(a_j) z_i$$

BP Example



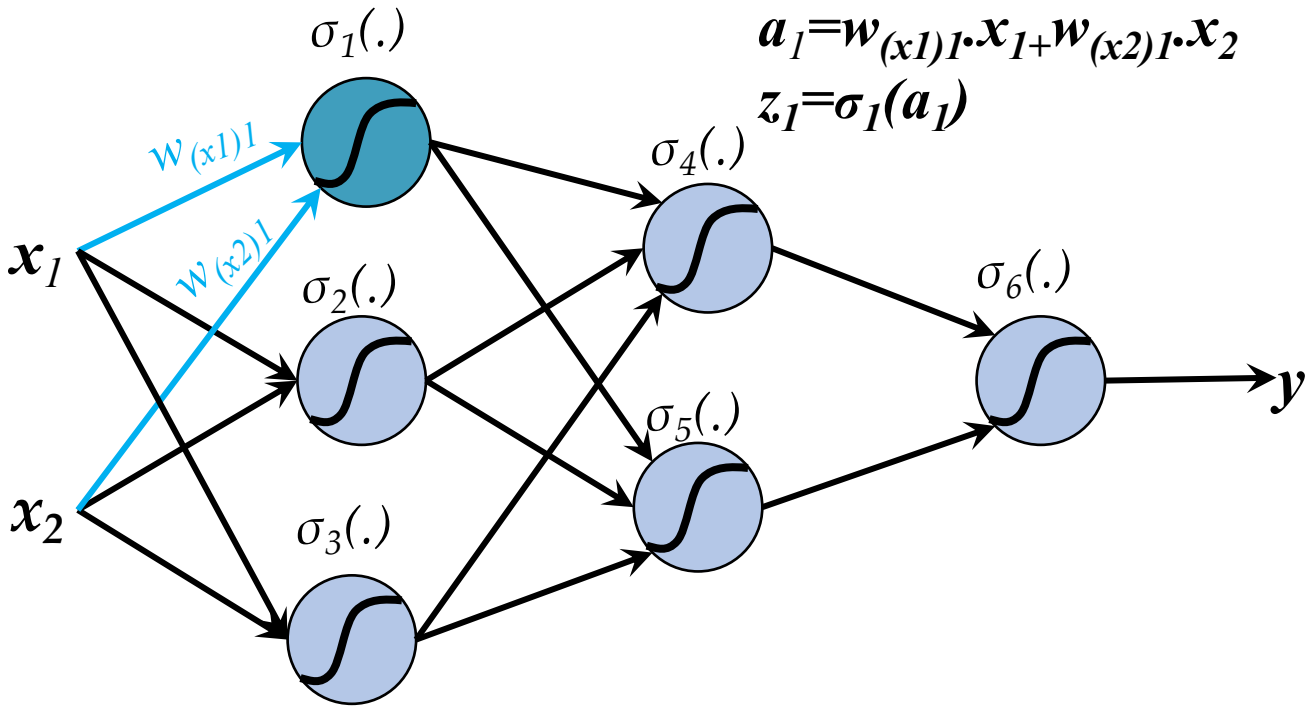
BP Example


x_1
w_1
$\sigma(.)$
z
x_2
w_2

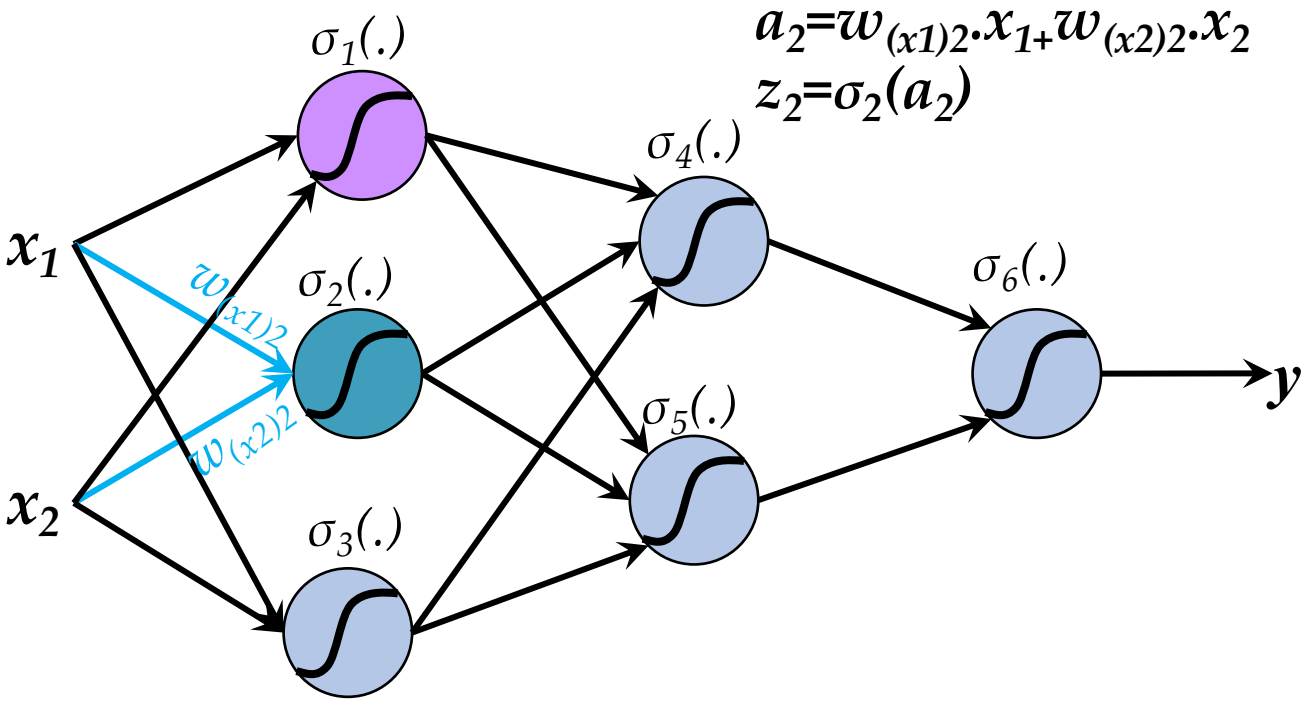

x_1
w_1
x_2
w_2
summing junction
$a=x_1w_1+x_2w_2$
non-linear element σ(a)
$z=\sigma(a)$

BP Example

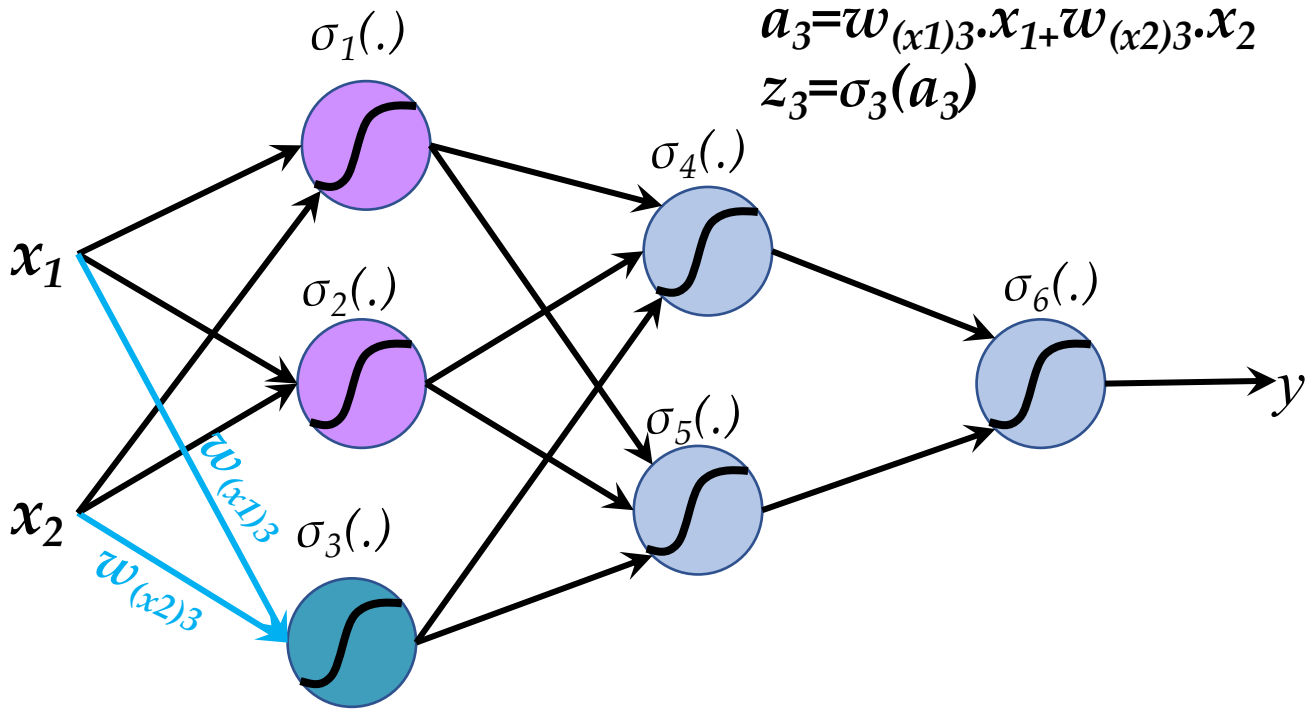
Forward Pass



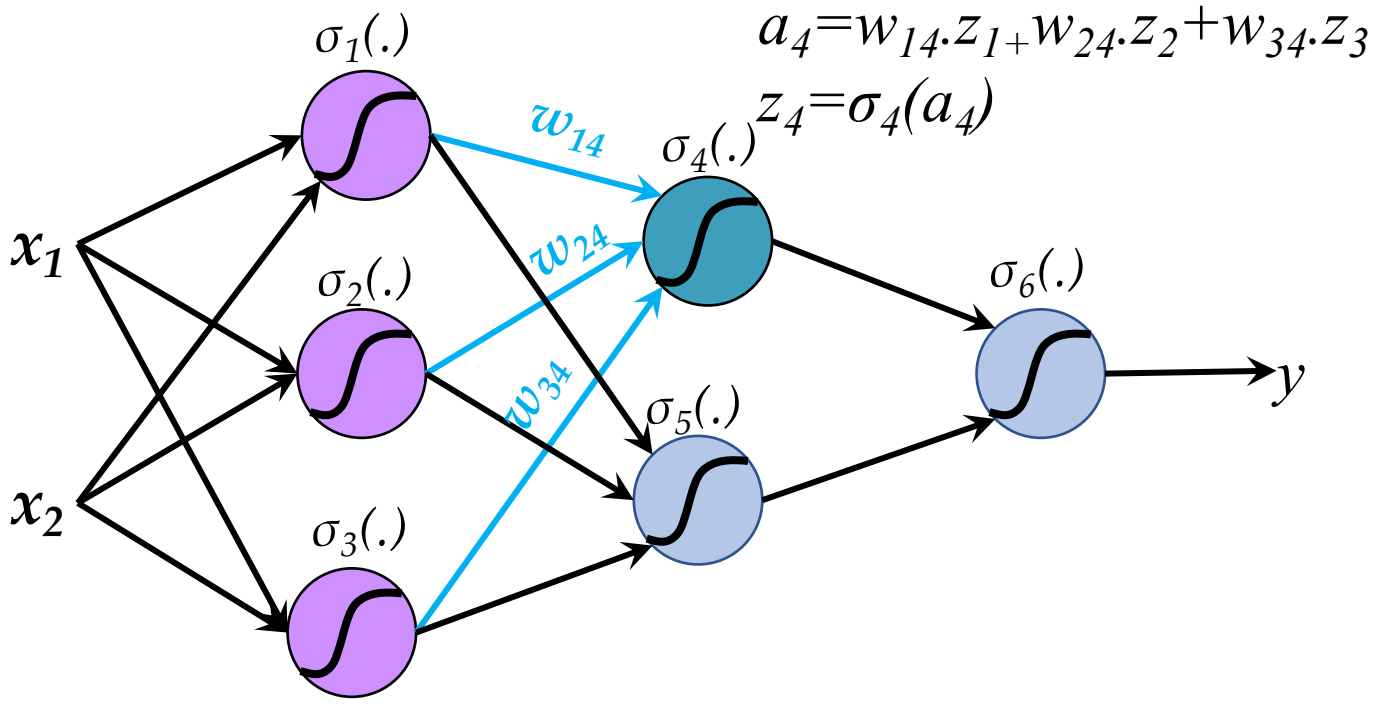
Forward Pass



Forward Pass

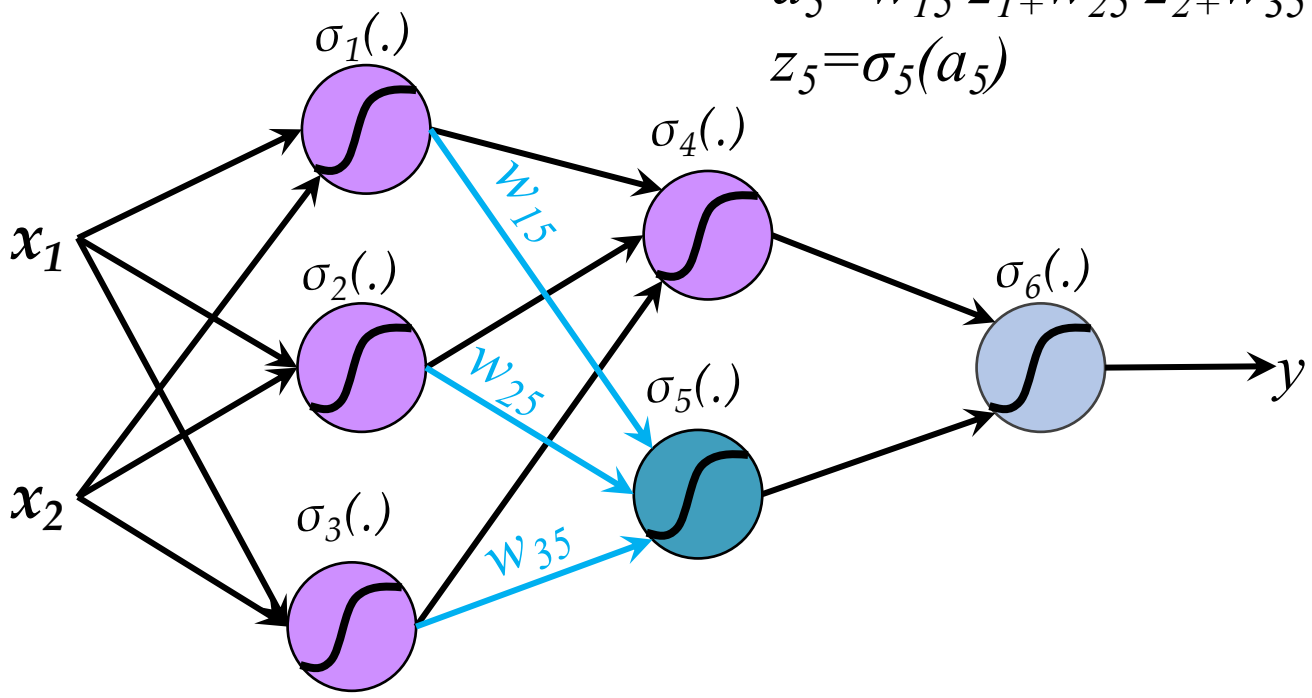


Forward Pass

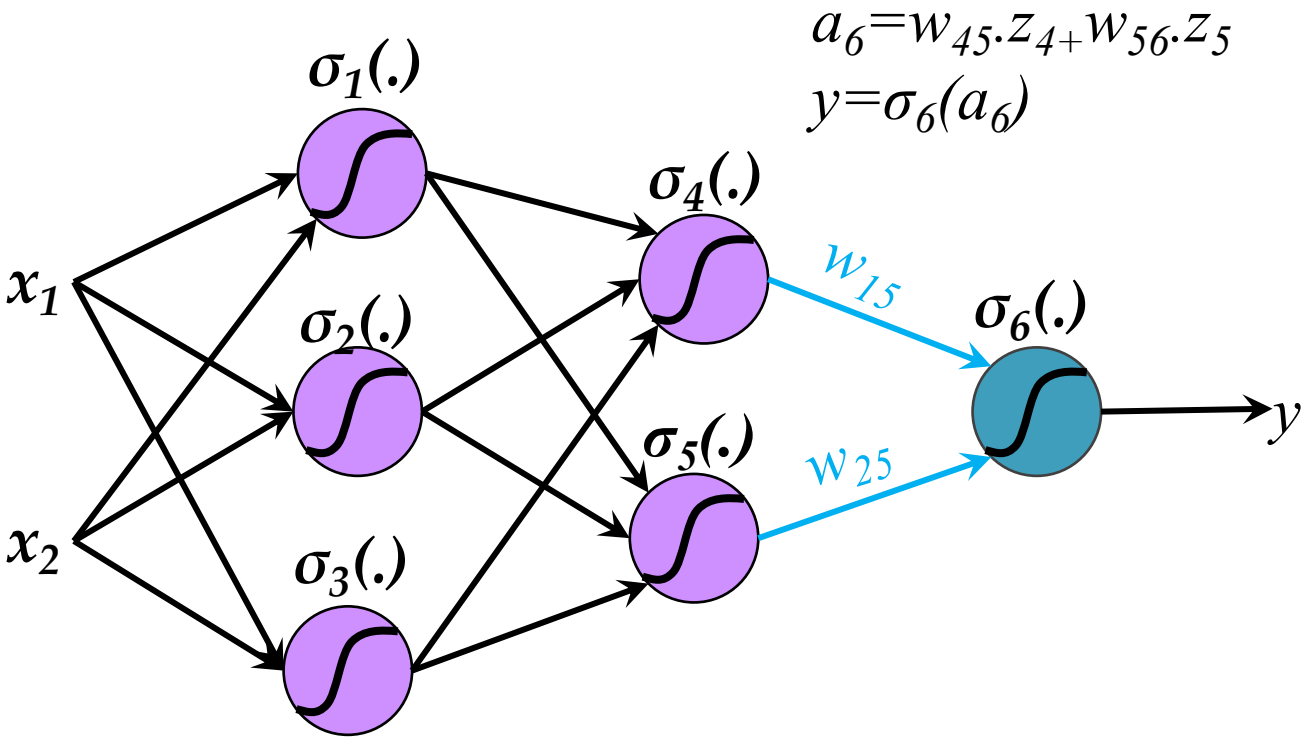


Forward Pass

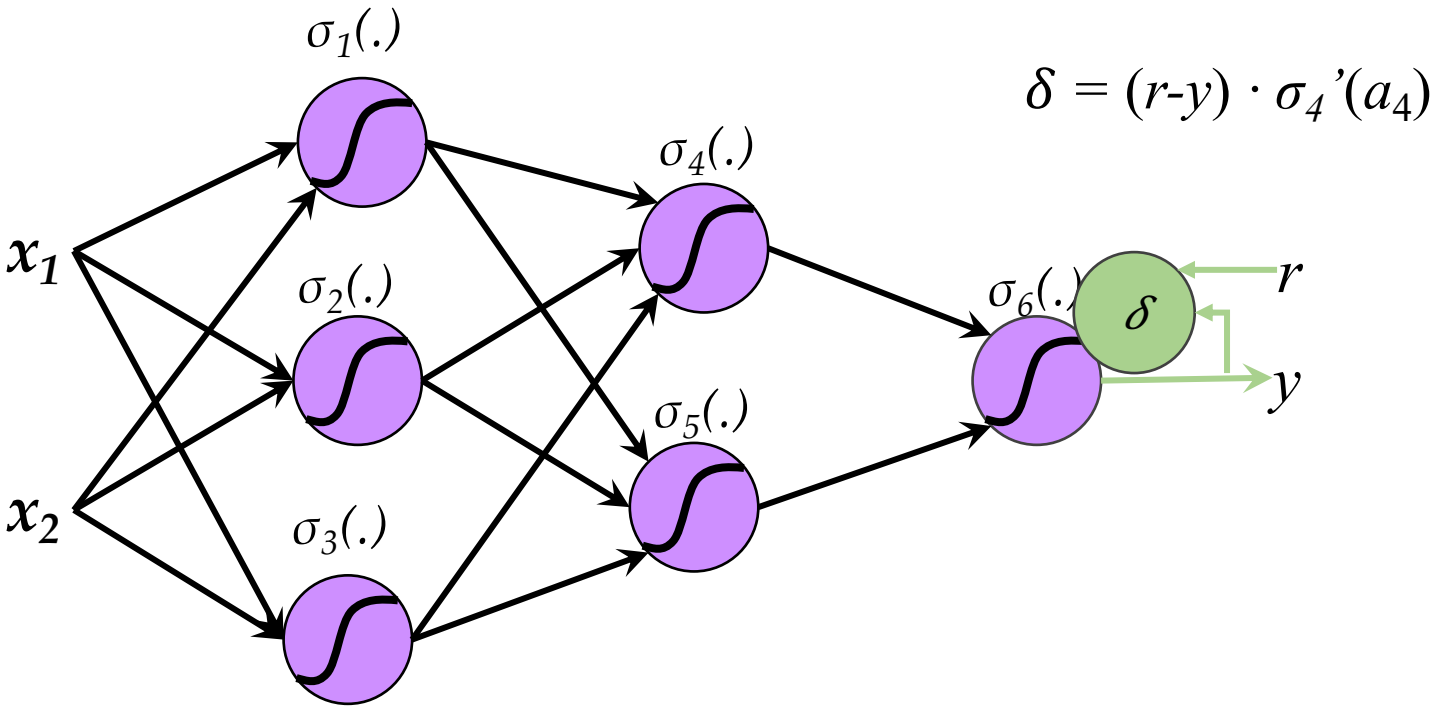
$$a_5 = w_{15} \cdot z_1 + w_{25} \cdot z_2 + w_{35} \cdot z_3$$
$$z_5 = \sigma_5(a_5)$$



Forward Pass

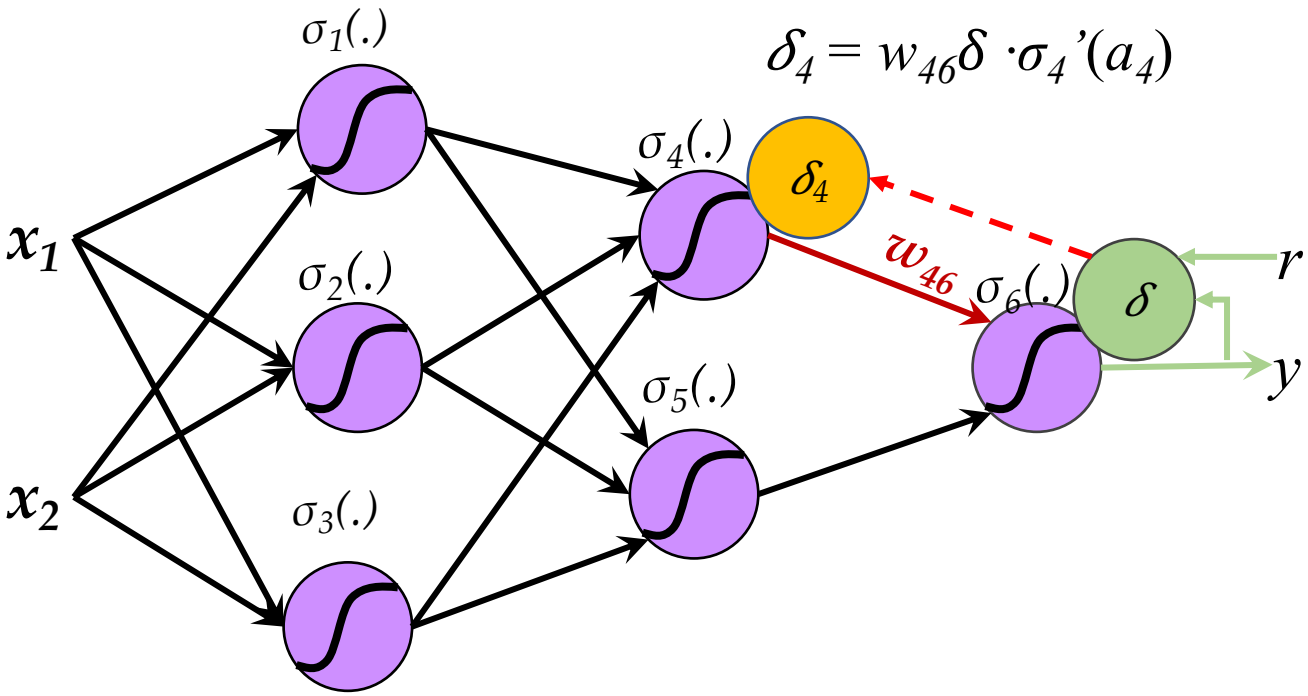


Backward Pass – error propagation

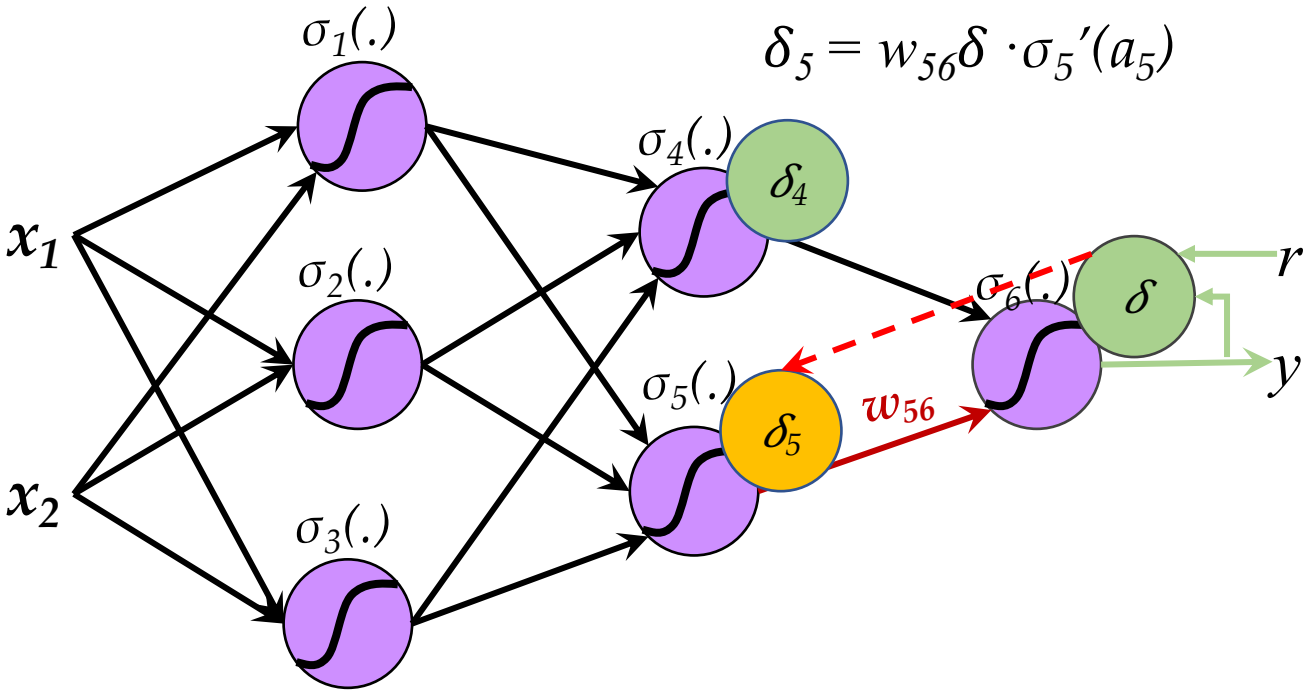


$$\delta = (r\text{-}y) \cdot \sigma_4'(a_4)$$

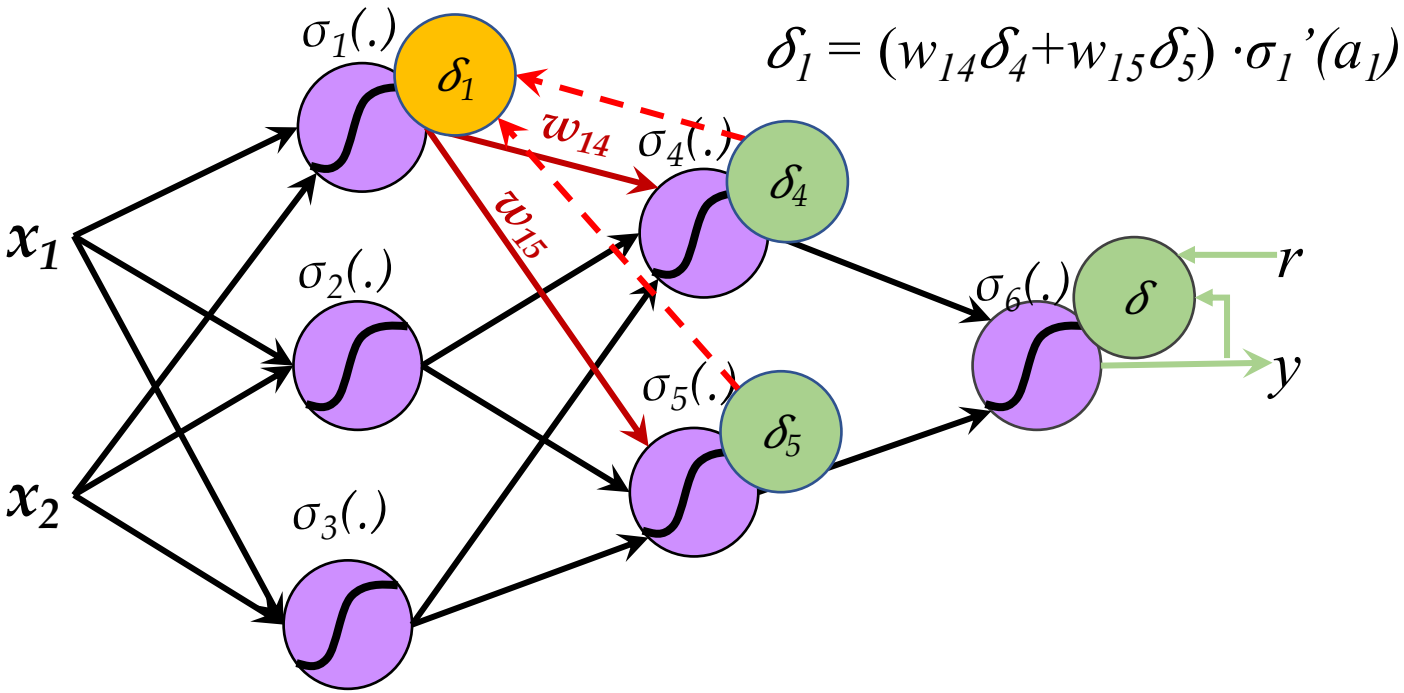
Backward Pass – error propagation



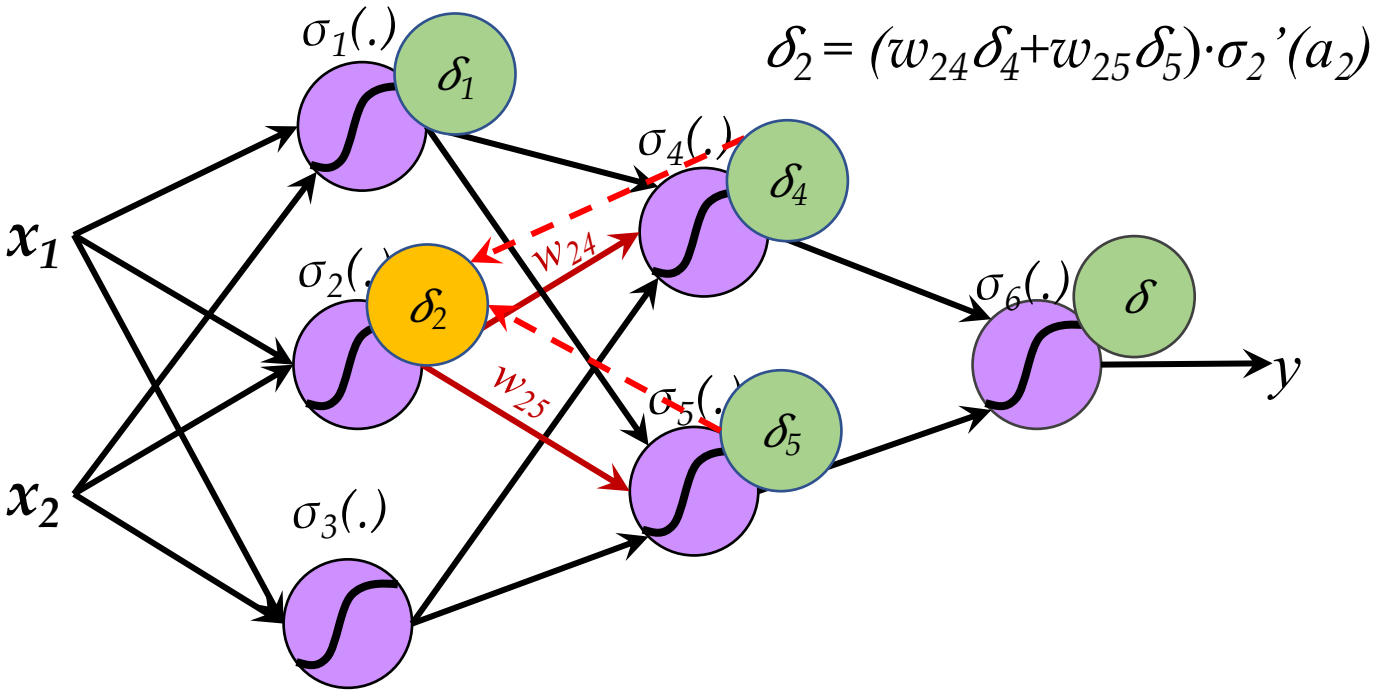
Backward Pass – error propagation



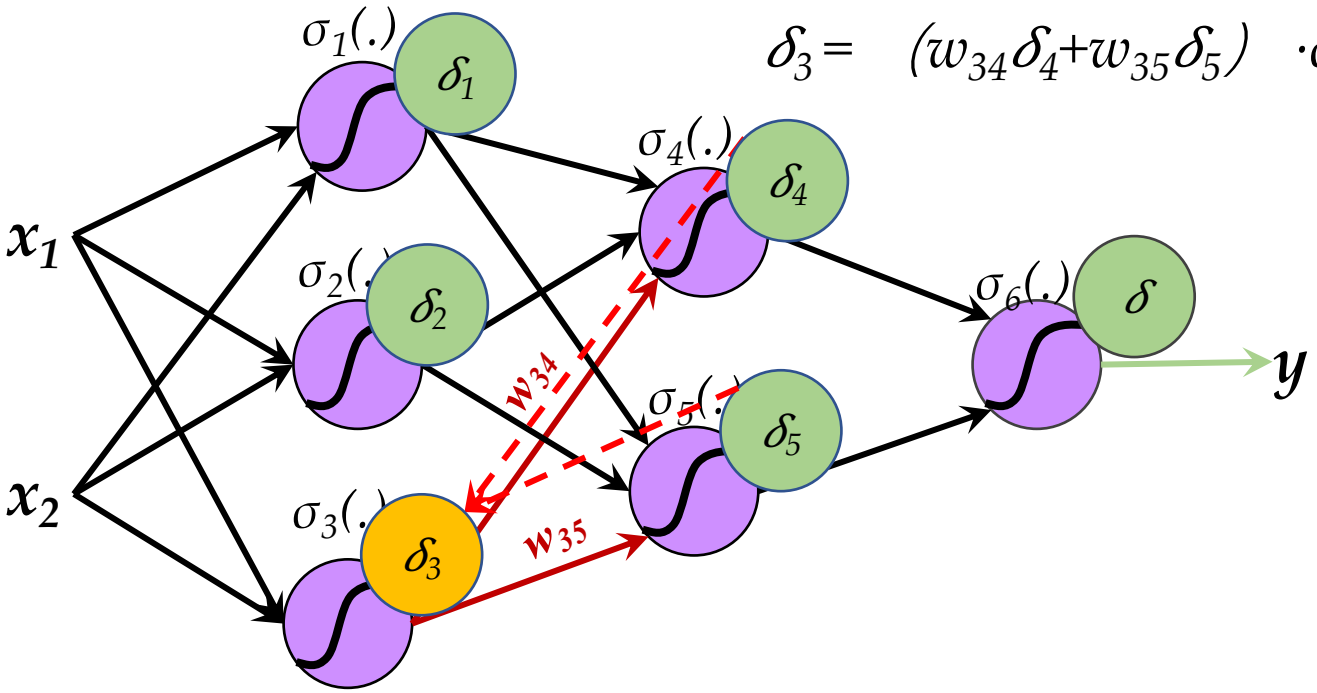
Backward Pass – error propagation



Backward Pass – error propagation

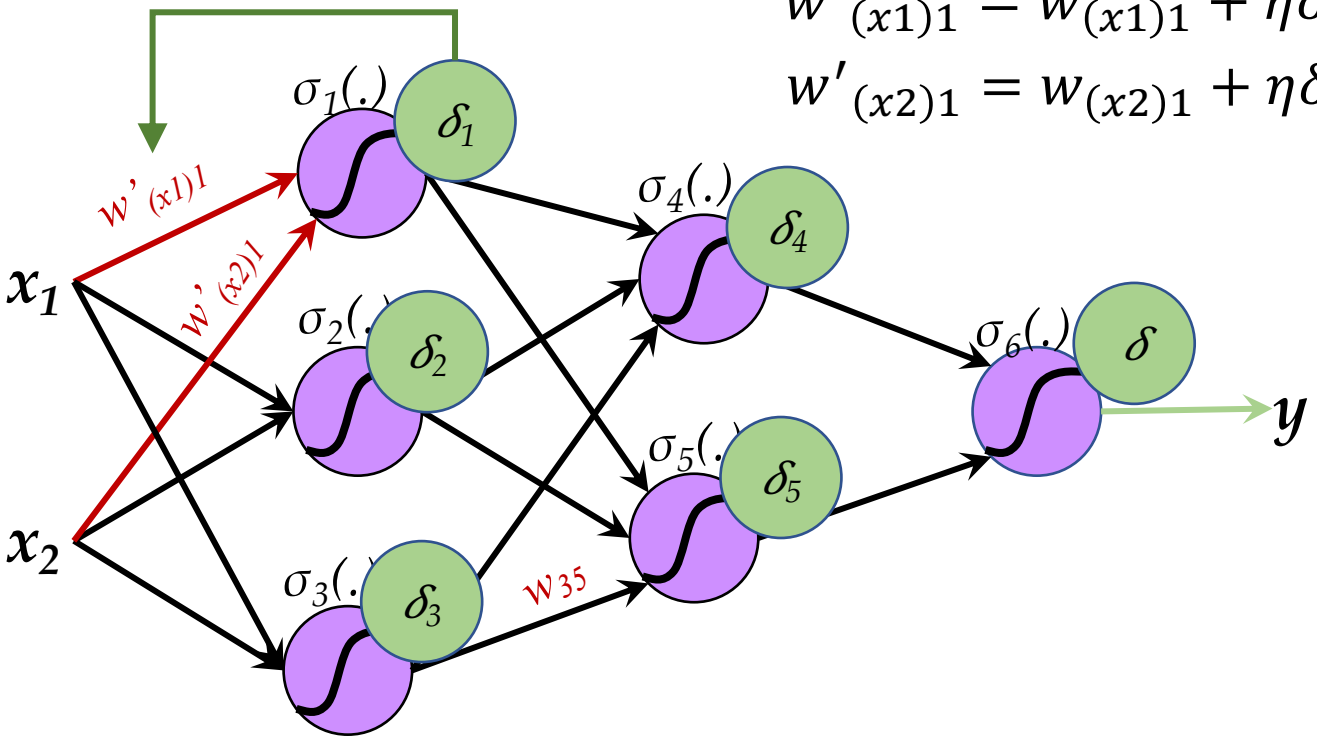


Backward Pass – error propagation



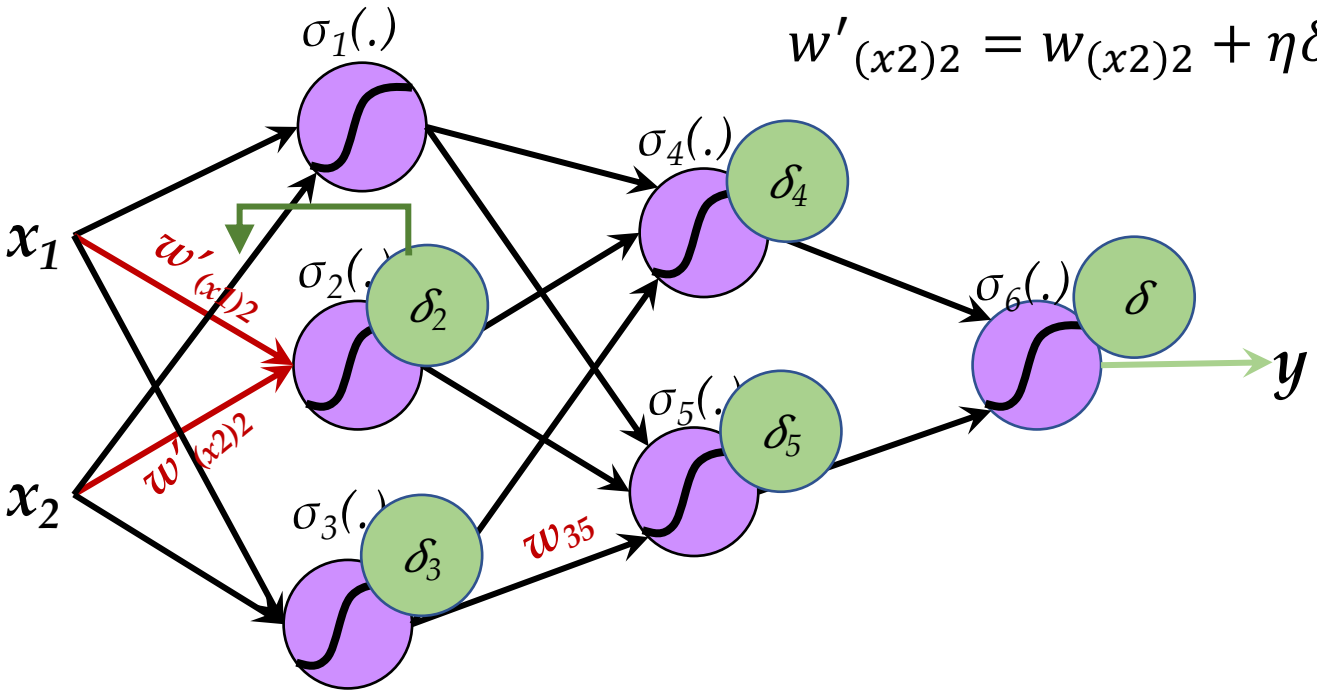
Optimization step – parameter update

$$w'_{(x1)1} = w_{(x1)1} + \eta\delta_1 x_1$$
$$w'_{(x2)1} = w_{(x2)1} + \eta\delta_1 x_2$$



Optimization step – parameter update

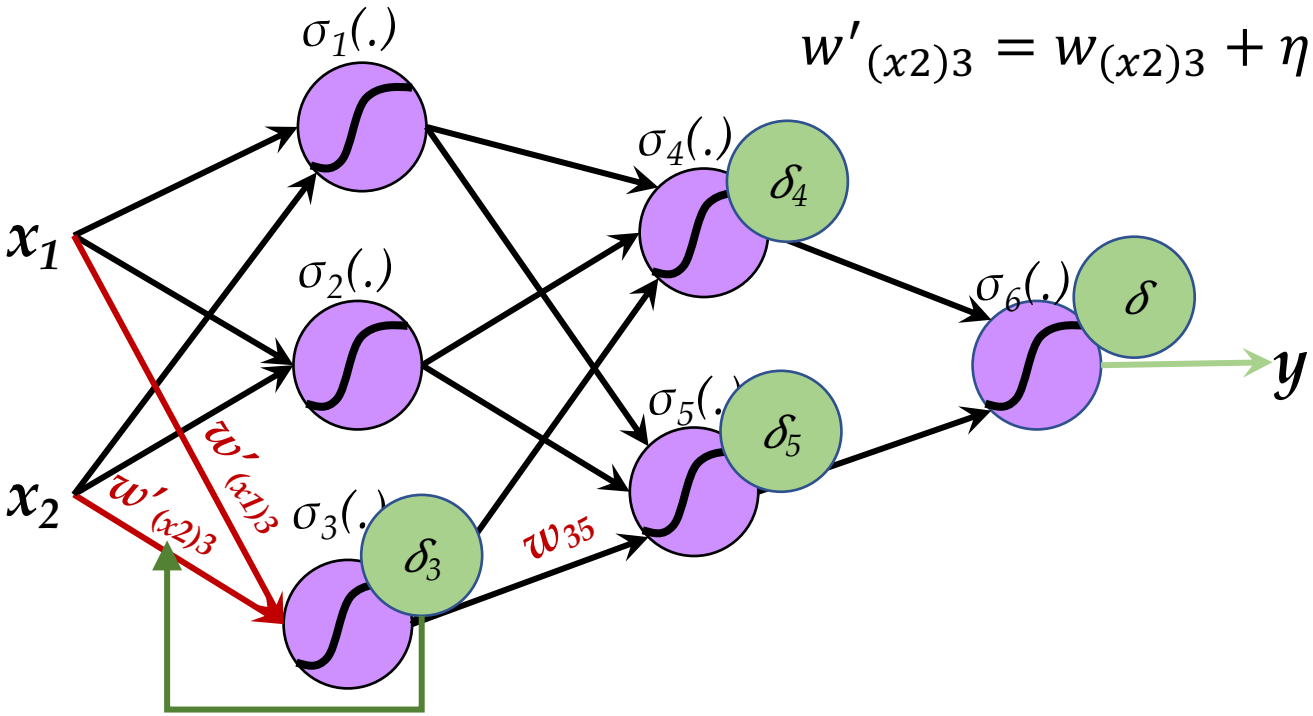
$$w'_{(x1)2} = w_{(x1)2} + \eta\delta_2 x_1$$
$$w'_{(x2)2} = w_{(x2)2} + \eta\delta_2 x_2$$



Optimization step – parameter update

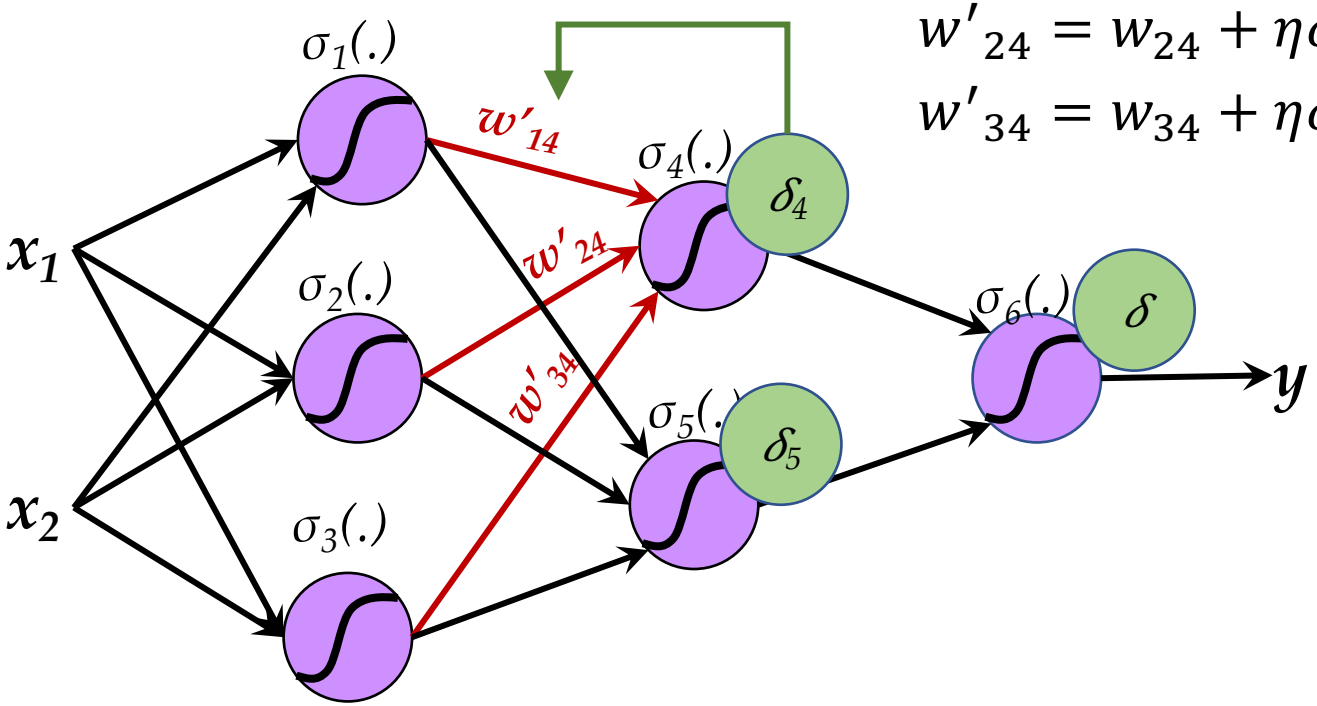
$$w'_{(x1)3} = w_{(x1)3} + \eta\delta_3 x_1$$

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



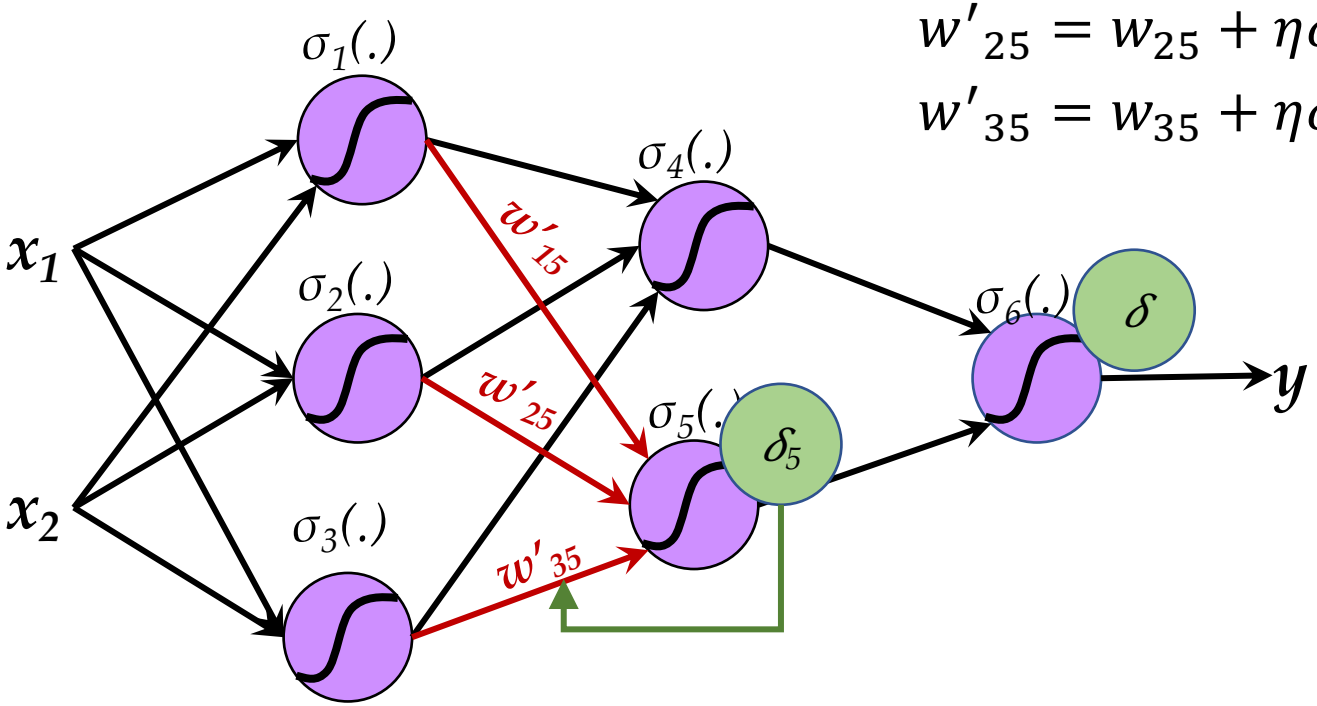
Optimization step – parameter update

$$w'_{14} = w_{14} + \eta\delta_4 z_1$$
$$w'_{24} = w_{24} + \eta\delta_4 z_2$$
$$w'_{34} = w_{34} + \eta\delta_4 z_3$$



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

$$w'_{46} = w_{46} + \eta\delta z_4$$
$$w'_{56} = w_{56} + \eta\delta z_5$$