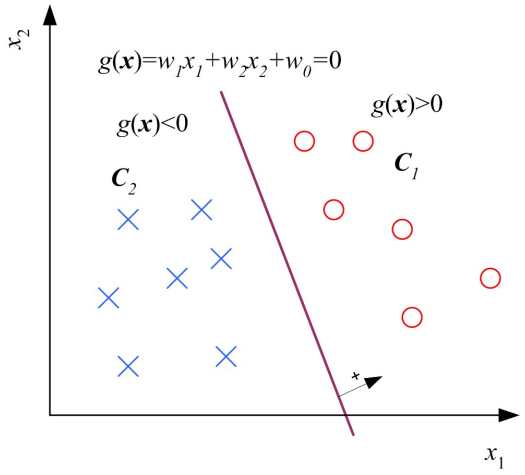
教材《机器学习与大语言模型》配套电子教案

机器学习

第八讲: 多层感知机

顾小东
上海交通大学计算机学院

Recall: Linear Classifiers

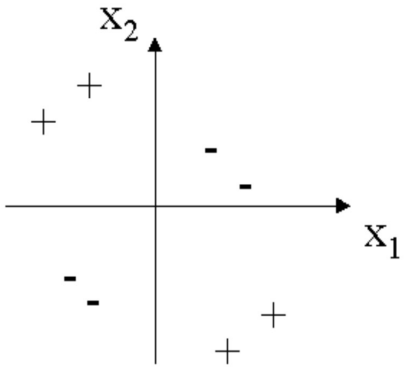


Perceptron

Logistic Regression

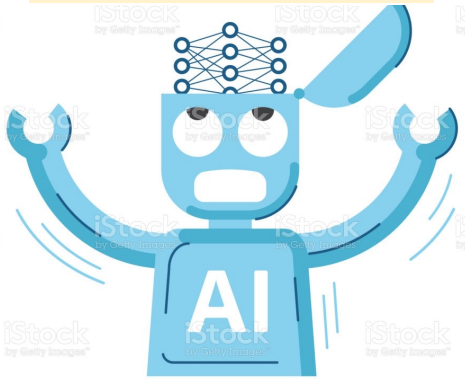
SVM

The curse of nonlinearity



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

Neural Networks



本讲提纲

人工神经网络基本原理

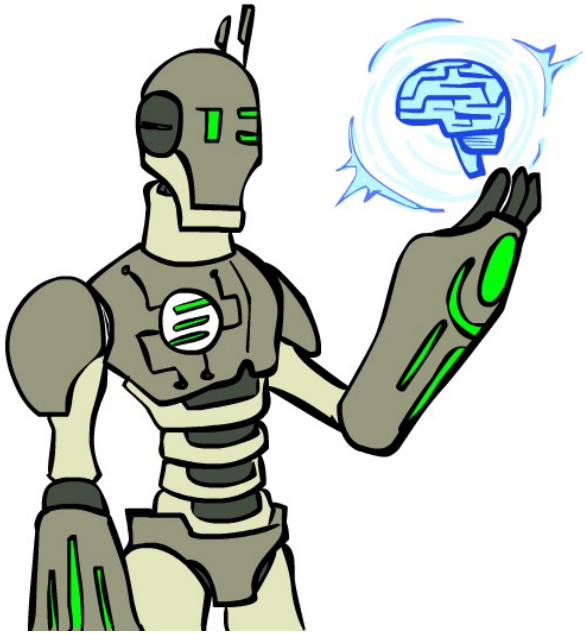
- ▻ 感知机
- ▻ 突破非线性魔咒
- ▻ 多层感知机
- ▻ 神经网络的直观理解

人工神经网络的训练方法

- ▻ 反向传播算法

深度学习概述

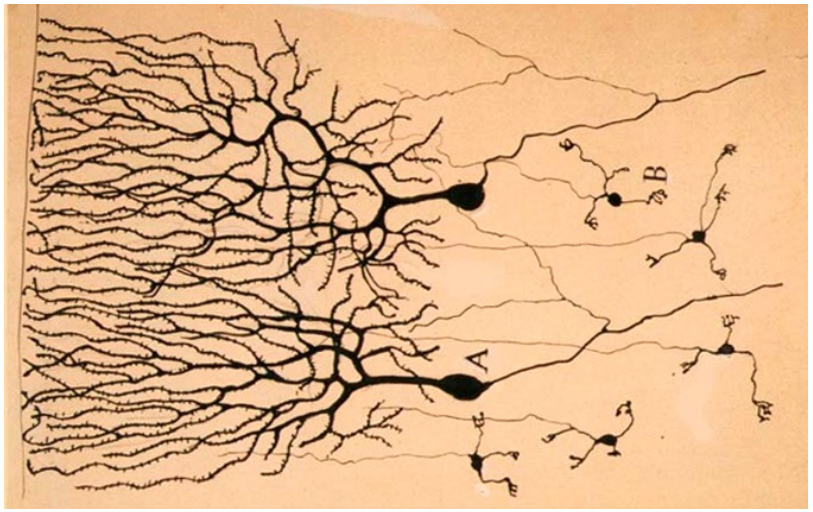
- ▻ 神经网络的深浅之争
- ▻ 深度学习的概念和核心内涵



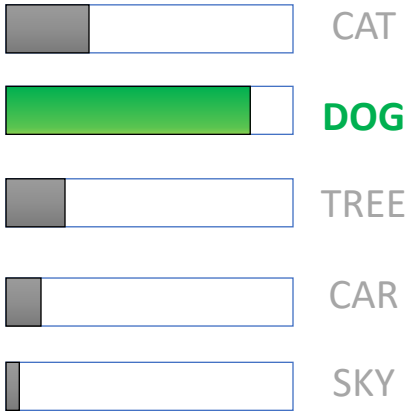
人工神经网络的起源

从人脑的神经处理机制获得启发

输入

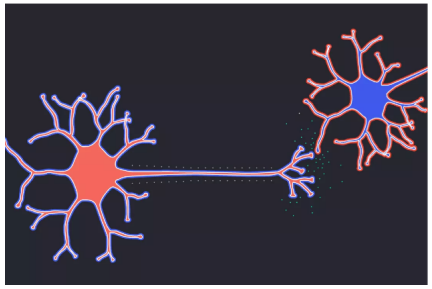


输出



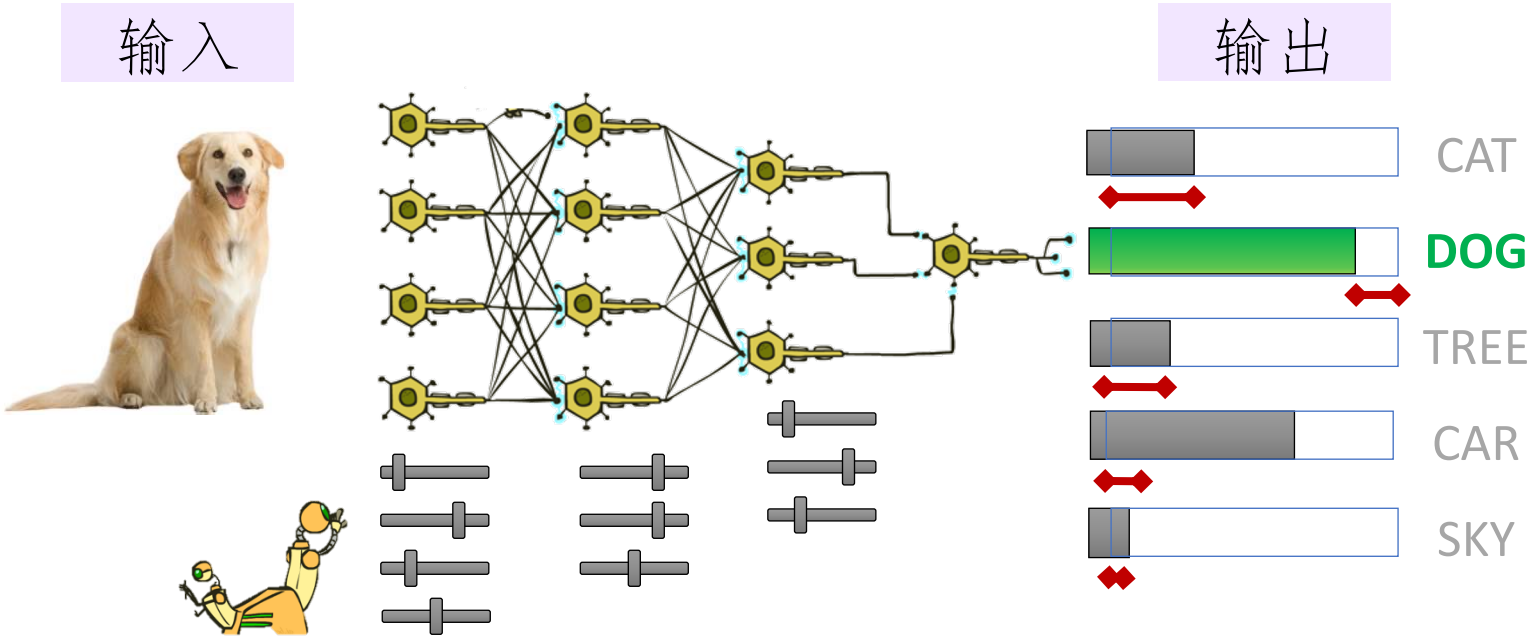
人类大脑拥有:

- 海量(10^{11})神经细胞作为基本处理单元
- 海量(10^{4})突触作为存储单元
- 并行处理能力
- 分布式计算/存储
- 对噪声和错误高容忍



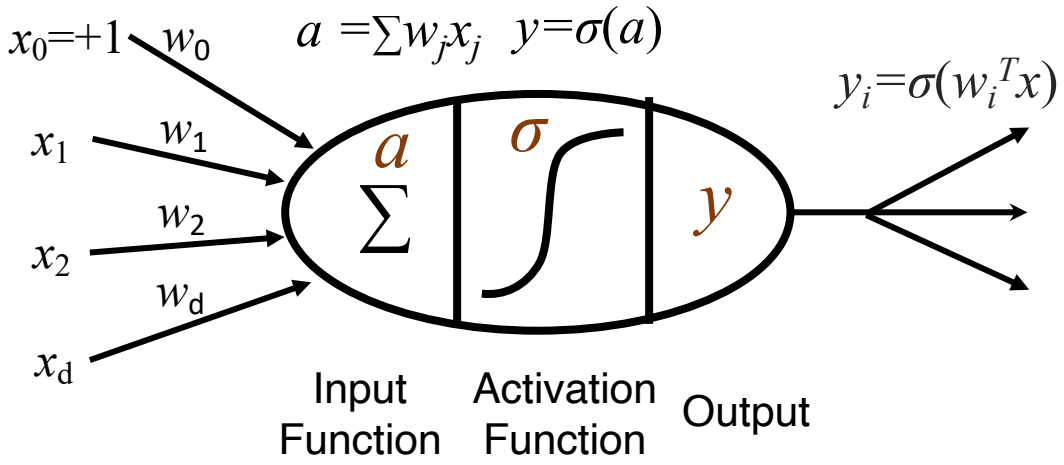
人工神经网络的起源

- 人工神经网络 (ANN)：模拟人类大脑的计算原理而设计的机器学习模型。



感知机——大脑仿造计划的开始

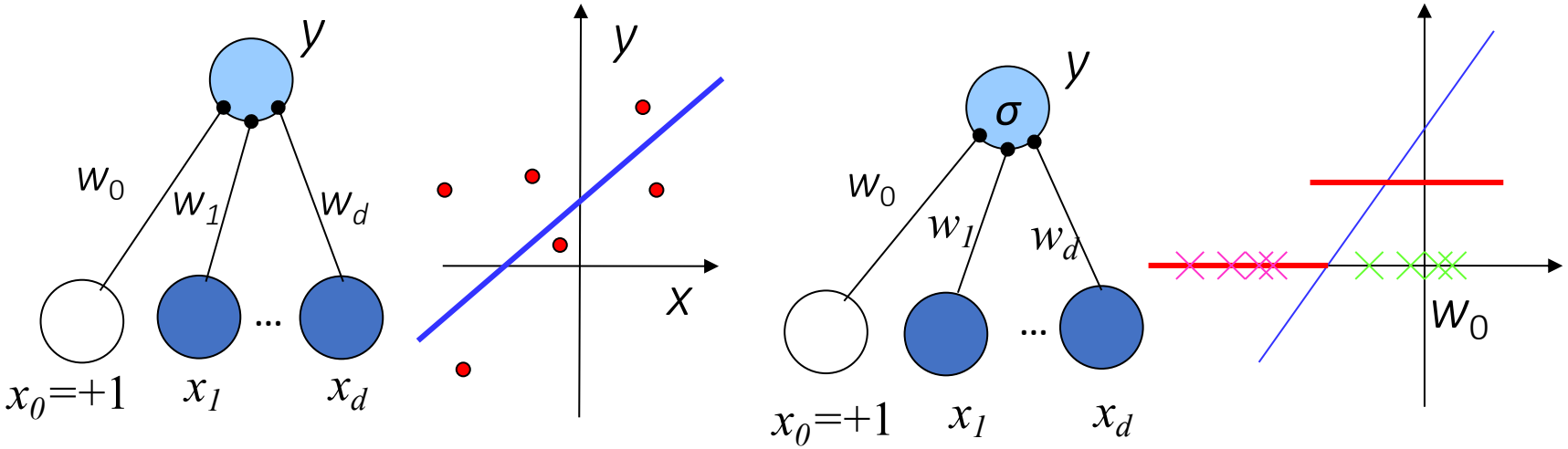
[Frank Rosenblatt, 1957]



- 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.

感知机的使用场景

- 回归模型: $y=\boldsymbol{w}x+w_0$
- 分类模型: $y=\text{sign}(\boldsymbol{w}x+w_0)$



- 对于分类模型，还需要一个额外的阈值函数，称为激活函数：

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

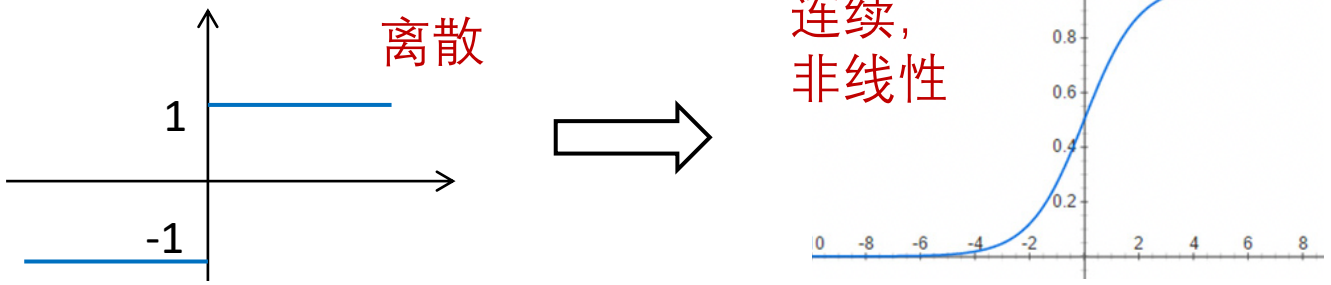
定义了分类的决策规则：

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

解决可微问题——连续激活函数

- 将离散的阈值函数改为连续的非线性函数，称为**激活函数。**

例如：Sigmoid函数



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

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

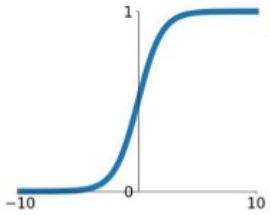
- Sigmoid can be seen as a continuous, differentiable version of thresholding.
- The output may be interpreted as the posterior probability that the input x belongs to C_1.

激活函数

- 一般的，为了给感知机引入非线性，我们引入连续的阈值函数，称为**激活函数**。

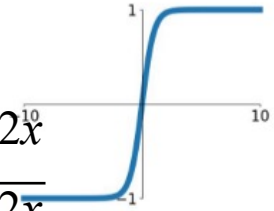
Sigmoid

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



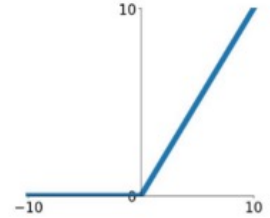
tanh

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



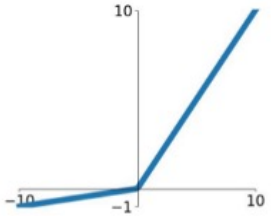
ReLU

$$\max(0, x)$$



Leaky ReLU

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

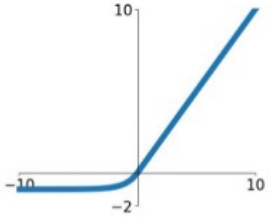


Maxout

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

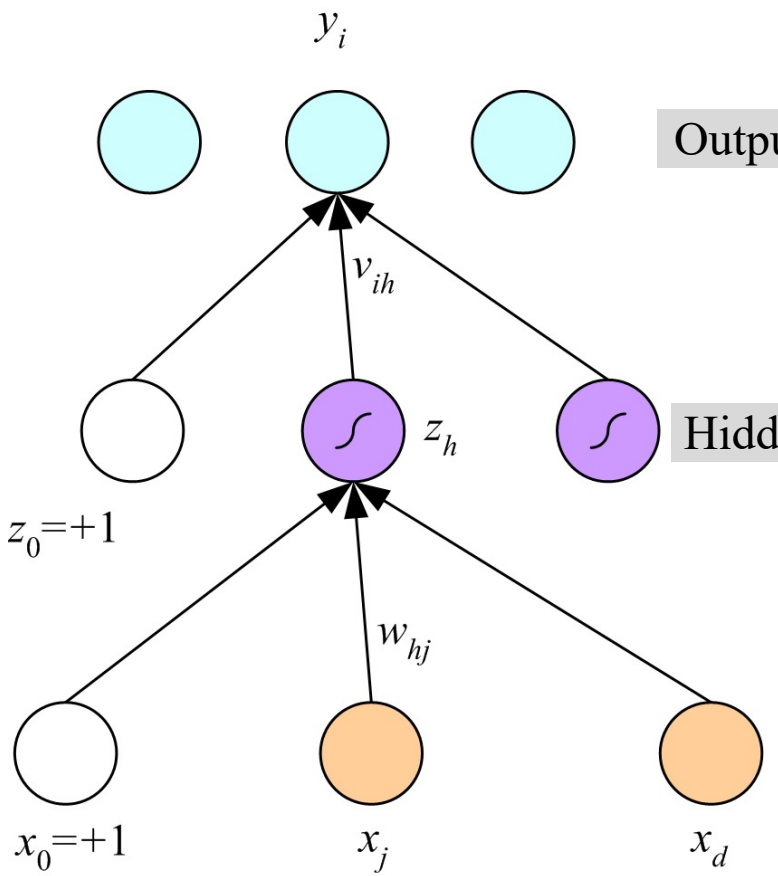
ELU

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



多层感知机(multilayer perceptron, MLP)

- 一种由多个感知机逐层复合而成的神经网络，包括输入层，隐含层，输出层。



a linear function in the new H-dim space.

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

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

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

no computation.

MLP如何训练？

训练优化MLP模型

令神经网络参数集为 $\theta=\{w_0,w_1,\ldots,w_n\}$

基本思路：按照梯度下降法迭代 θ

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

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

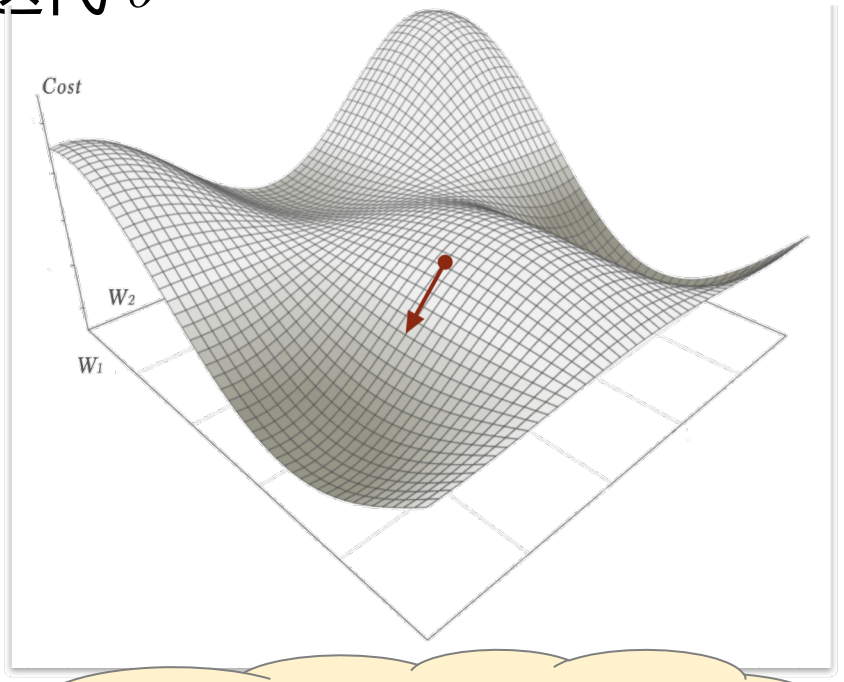
$$\theta^2 = \theta^1 - \eta \nabla \mathrm{L}\left(\theta^1\right)$$

$$\vdots$$

其中，$\nabla \mathrm{L}(\theta)$

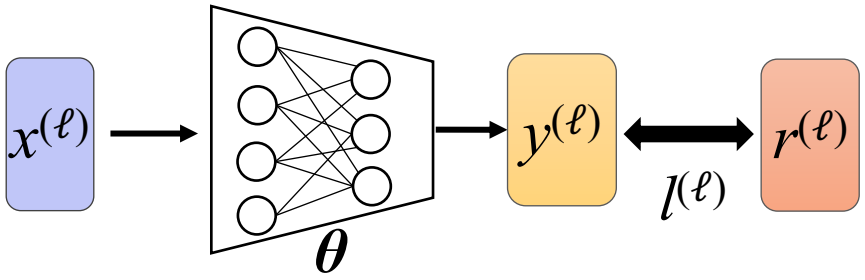
$$\begin{bmatrix} \partial \mathrm{L}(\theta)/\partial w_1 \\ \partial \mathrm{L}(\theta)/\partial w_2 \\ \vdots \end{bmatrix}$$

Millions of Parameters



核心问题：如何获得每个参数的梯度？

损失函数 (以回归问题为例)

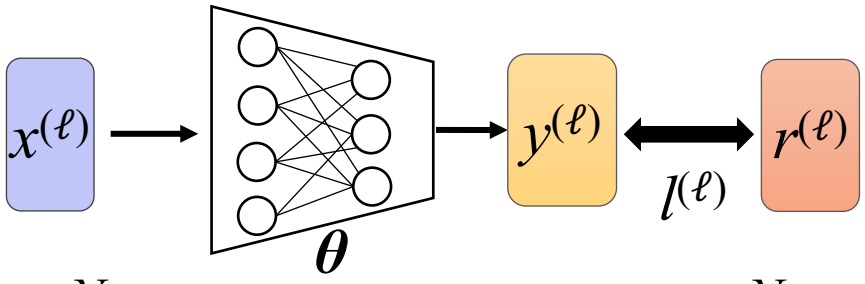


- 给定数据集 $D = \{(x^{(1)}, r^{(1)}), \ldots, (x^{(N)}, r^{(N)})\}$ 其中 $x^{(\ell)}$ 和 $r^{(\ell)}$ 分别代表第 ℓ个输入和目标输出。
- 对每个输入样本$x^{(\ell)}$, 模型预测值为 $y^{(\ell)}$.
- 学习目标是最小化 **MSE** 损失函数:

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

损失函数

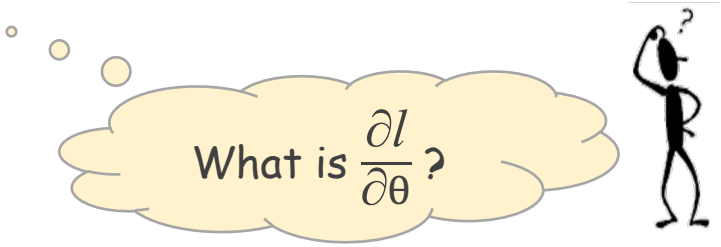
- 考虑数据样本间的独立性，我们认为总体损失函数是单个样本损失函数的平均值。



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

- 对于单个数据样本 (x, r)，其损失函数为:

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



Recall: Chain Rule

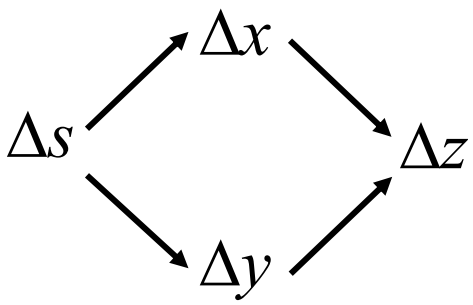
Case 1

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

$$\Delta x \rightarrow \Delta y \rightarrow \Delta z \qquad \frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}$$

Case 2

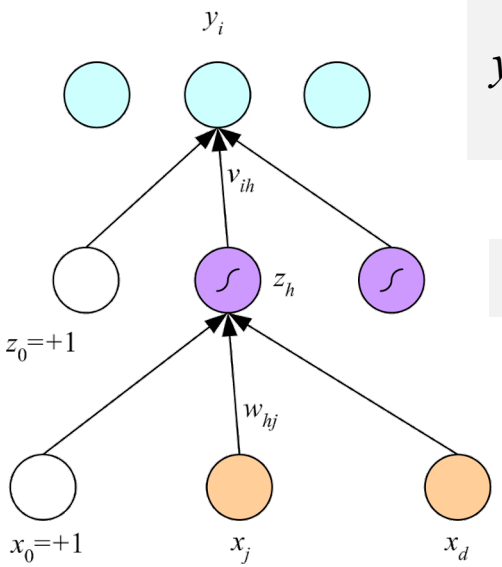
$$x = g(s) \qquad y = h(s) \qquad z = k(x, y)$$



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

Gradient Descend for 2-Layer MLP

Regression



forward

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

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

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

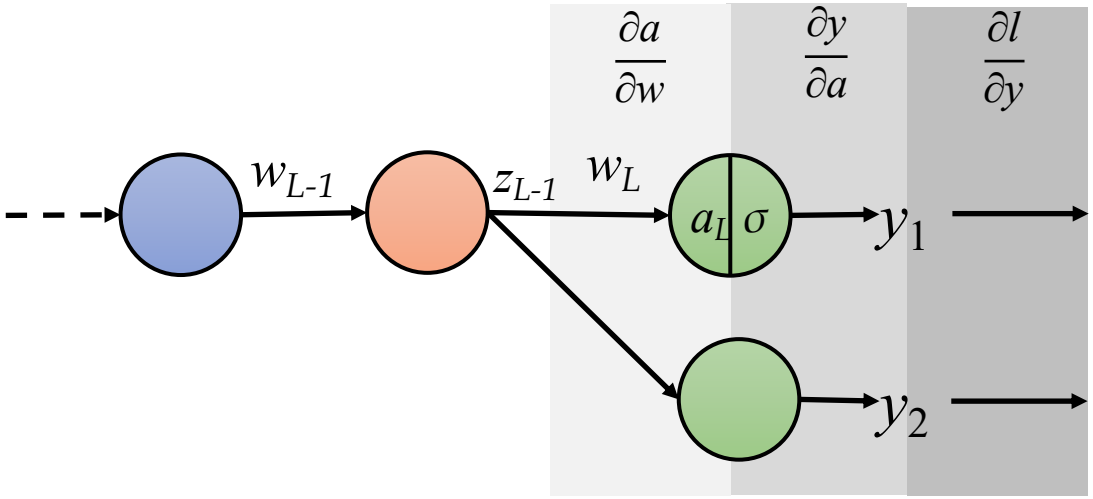
$$l(\boldsymbol{w}, \boldsymbol{v}) = \frac{1}{2} \sum_i (r_i - y_i)^2$$

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?



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

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

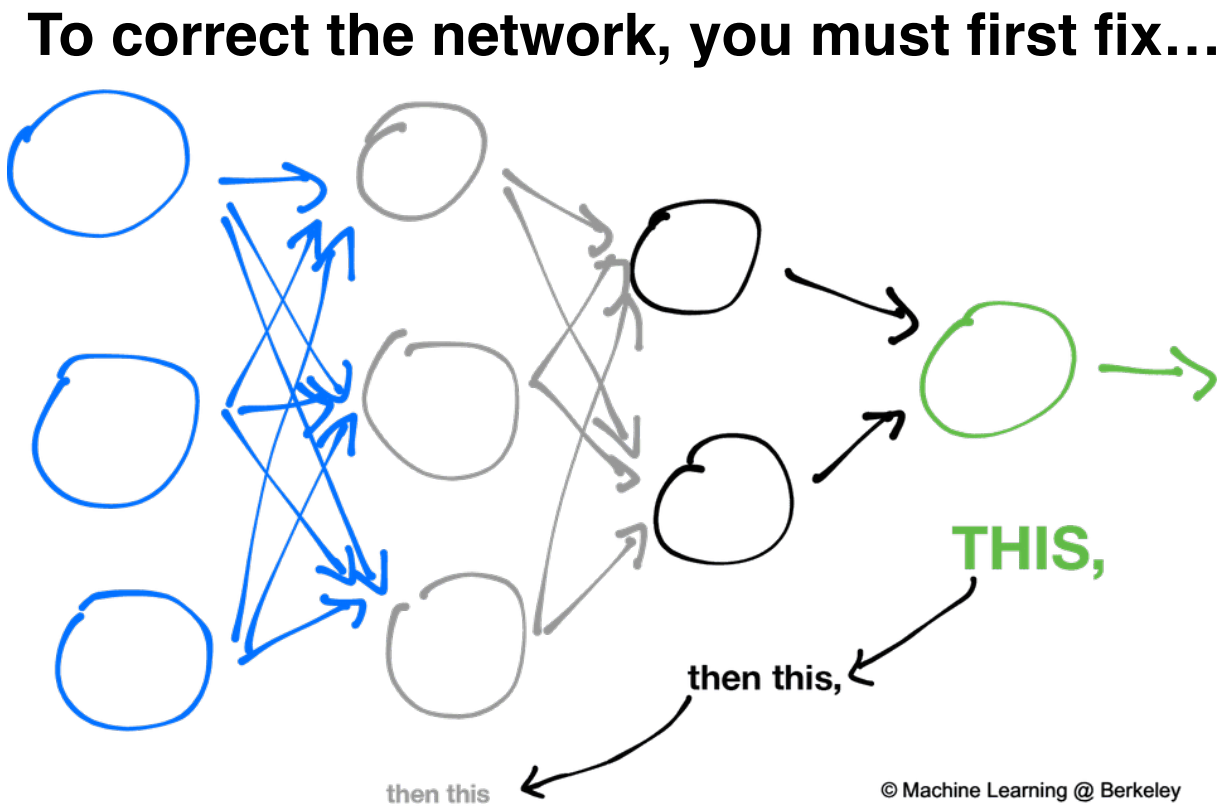
Problems of deriving $\nabla_w L$ directly:

- 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

破解之道 — 反向传播

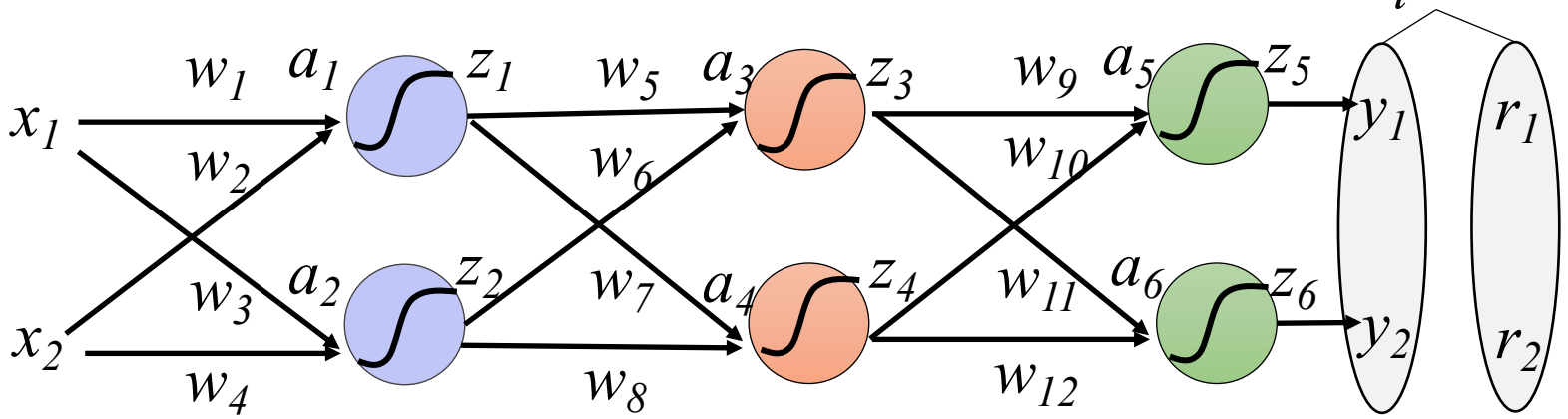
- 一种高效计算神经网络梯度的方法。
- **基本思想**：将误差从网络的最后端逐层向前传播分配。



Backpropagation

- Consider a simple, general MLP

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

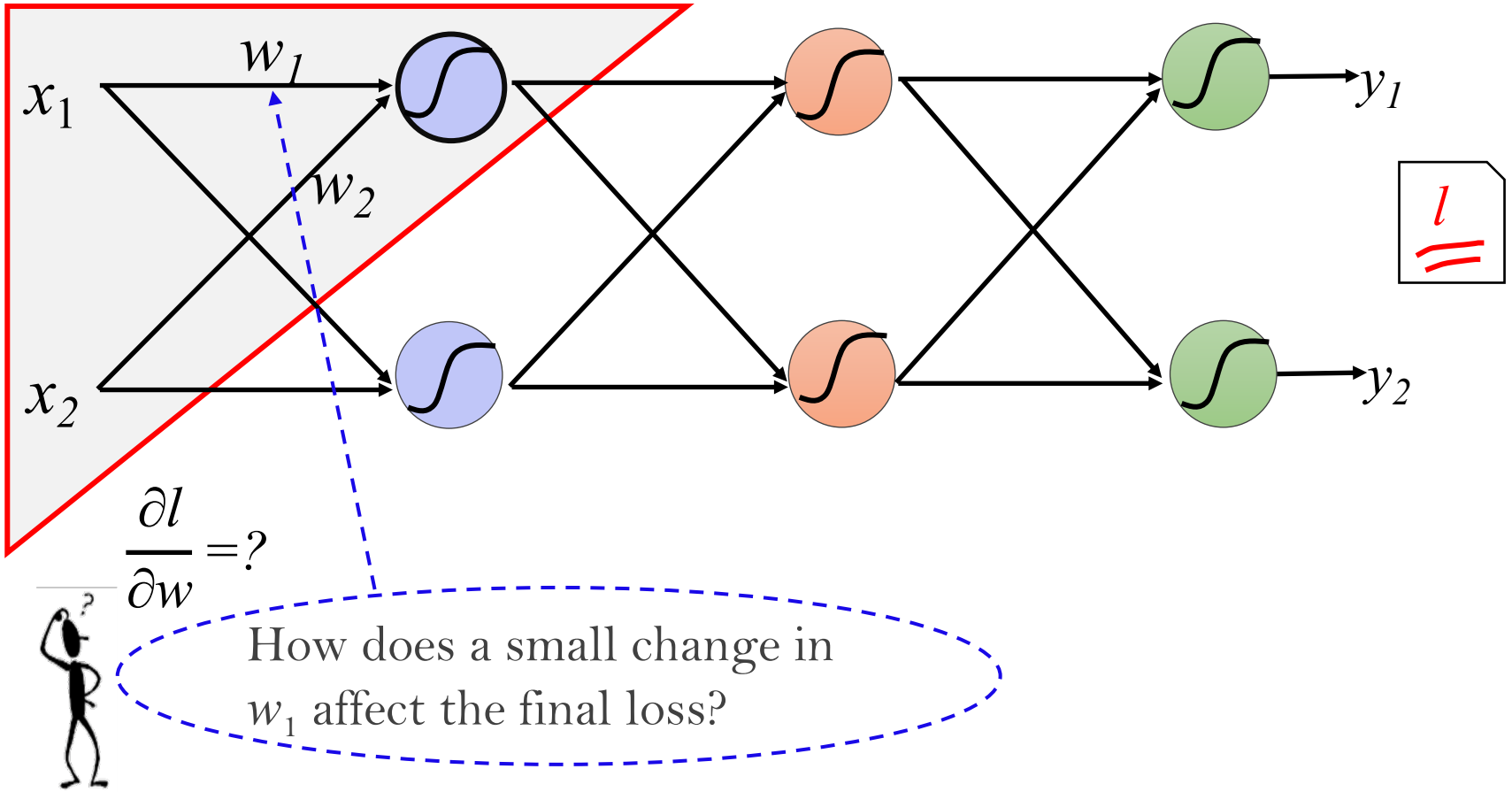


$$\frac{\partial l}{\partial w} = ?$$

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

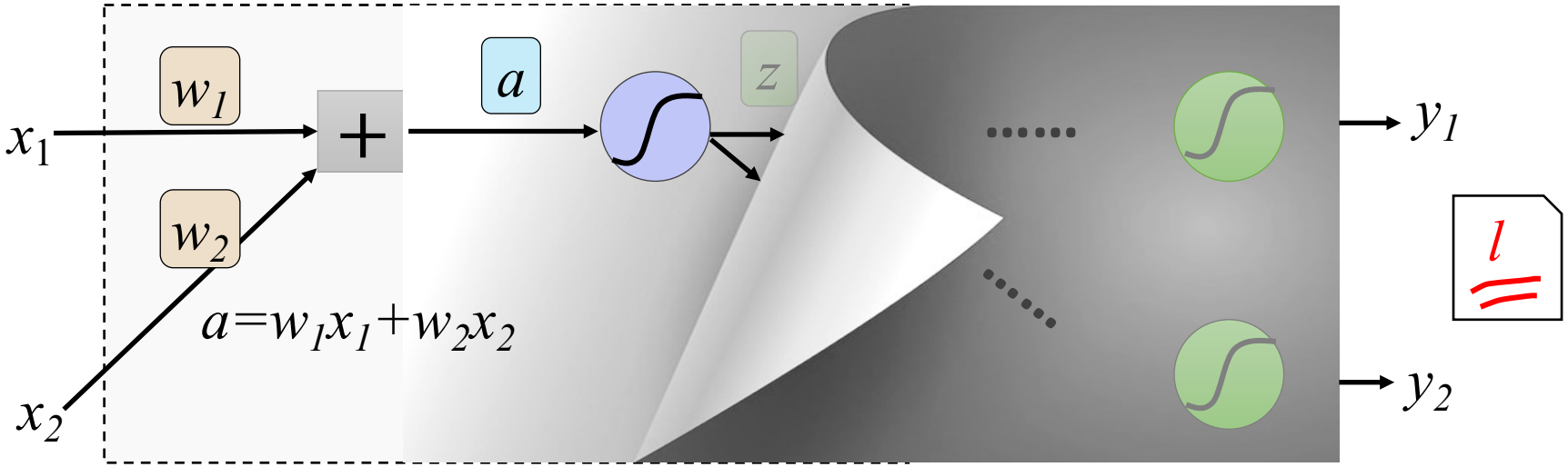
Backpropagation

Initially, consider the first neuron



Backpropagation

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



(chain rule)

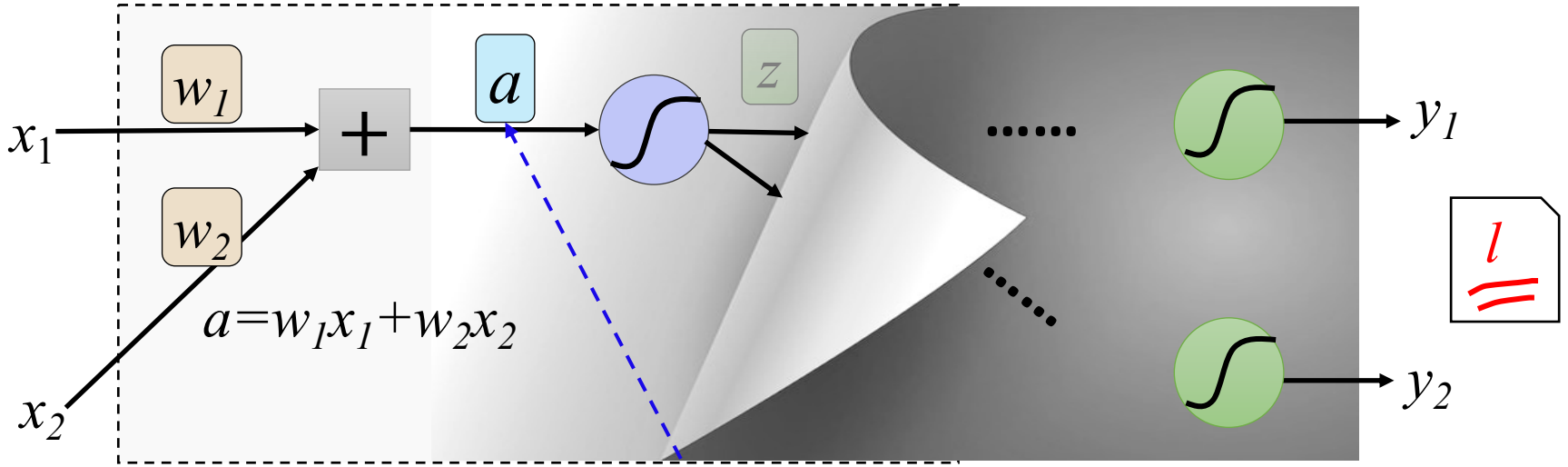
$$\frac{\partial l}{\partial w} = ? \quad \frac{\partial a}{\partial w} \cdot \frac{\partial l}{\partial a}$$

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.



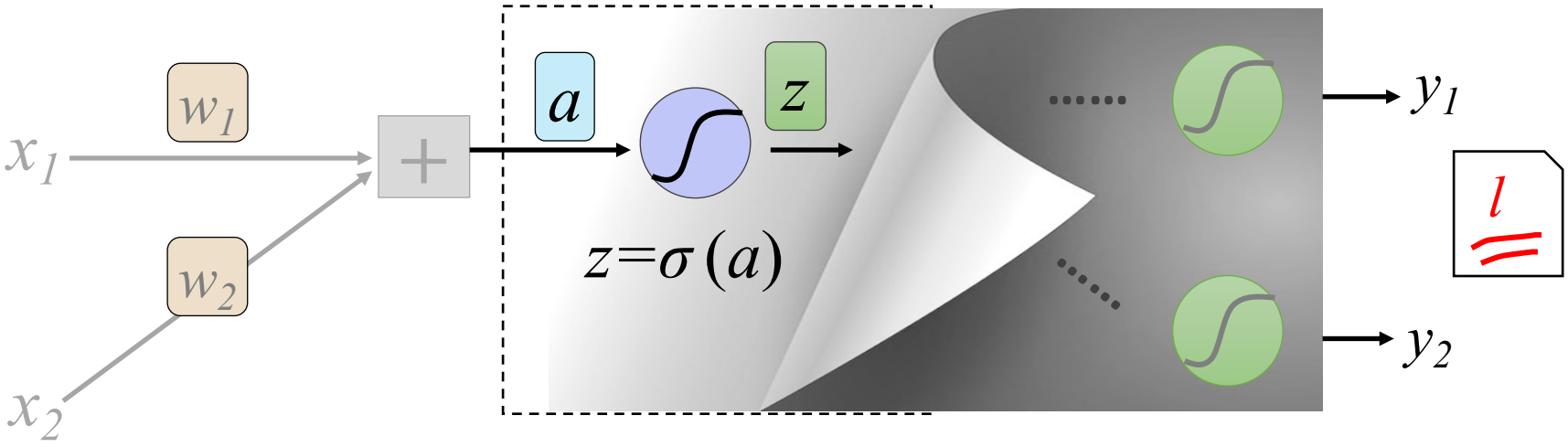
$$\frac{\partial l}{\partial w} = x \cdot \frac{\partial l}{\partial a}$$

?

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

Backpropagation – Backward Pass

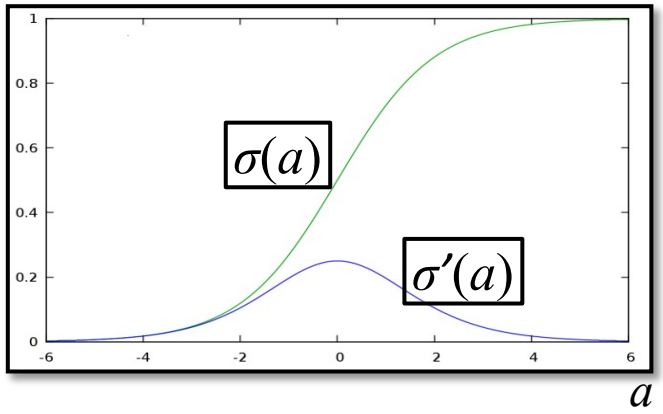
- Compute $\partial l/\partial a$ for the linearity output a.



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

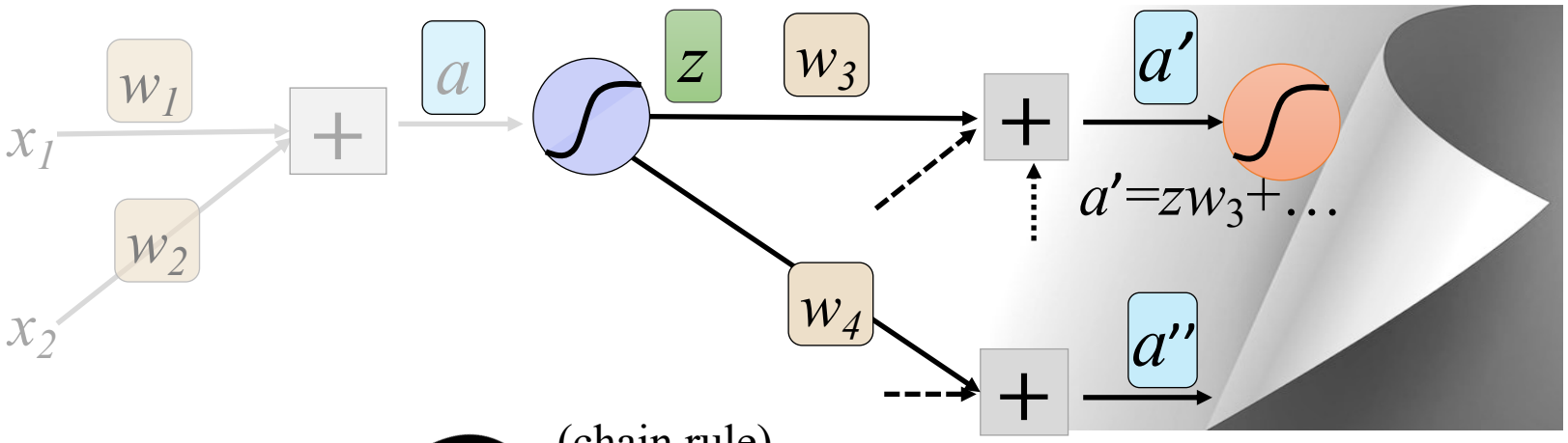
$$\frac{\partial z}{\partial a} = \sigma'(a)$$

Example



Backpropagation – Backward Pass

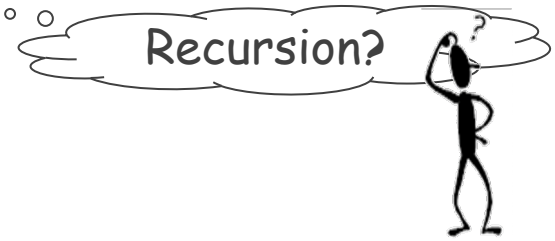
- Compute $\partial l/\partial z$ for nonlinearity outputs z.



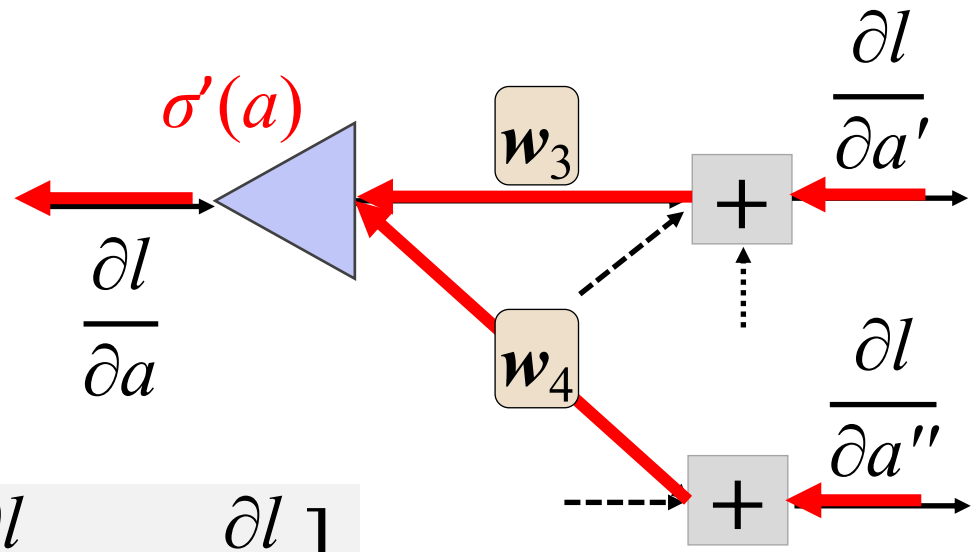
(chain rule)

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

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



Backpropagation – Backward pass



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

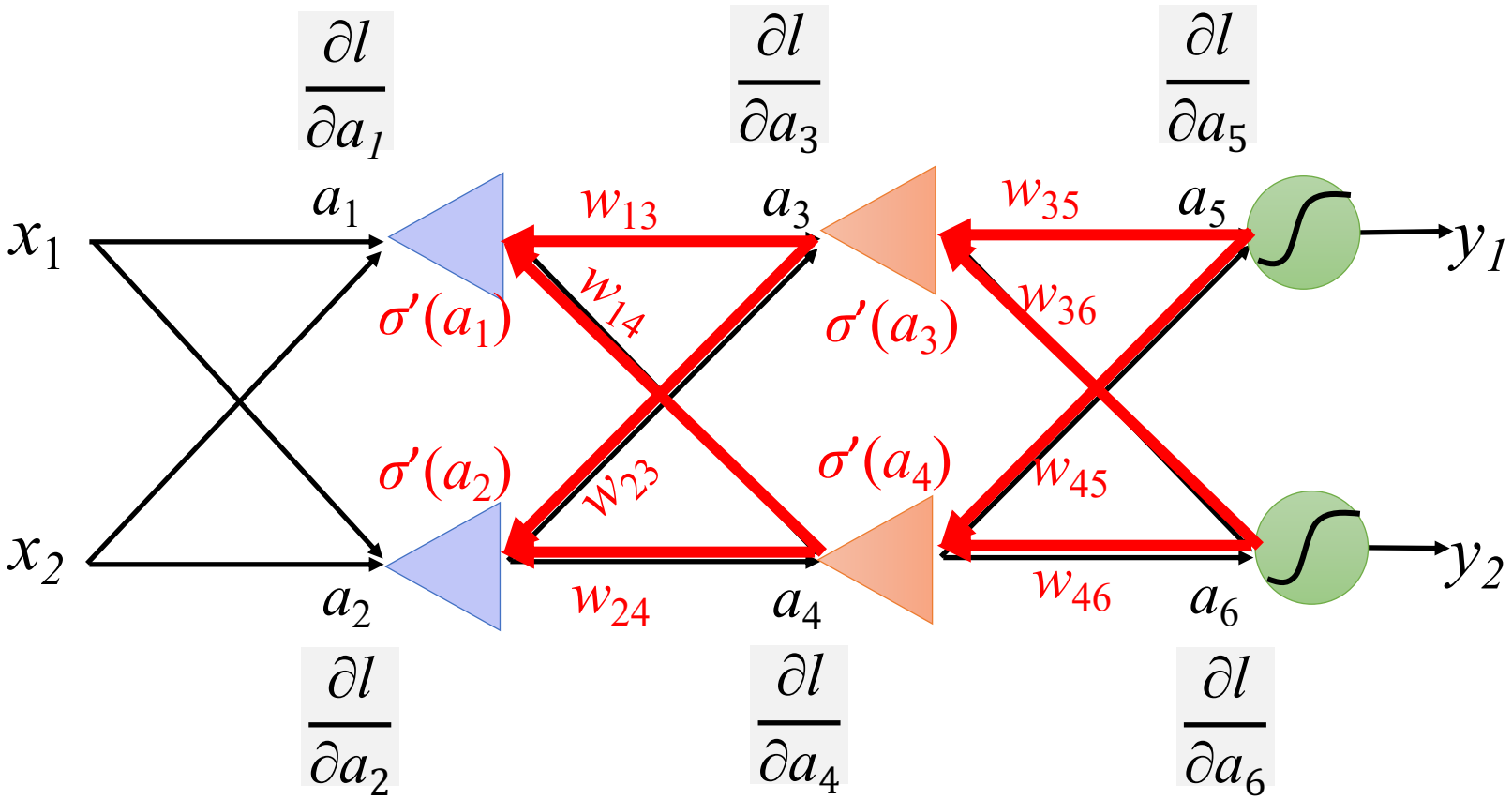
constant because a has already been determined in the forward pass.

Backward pass:

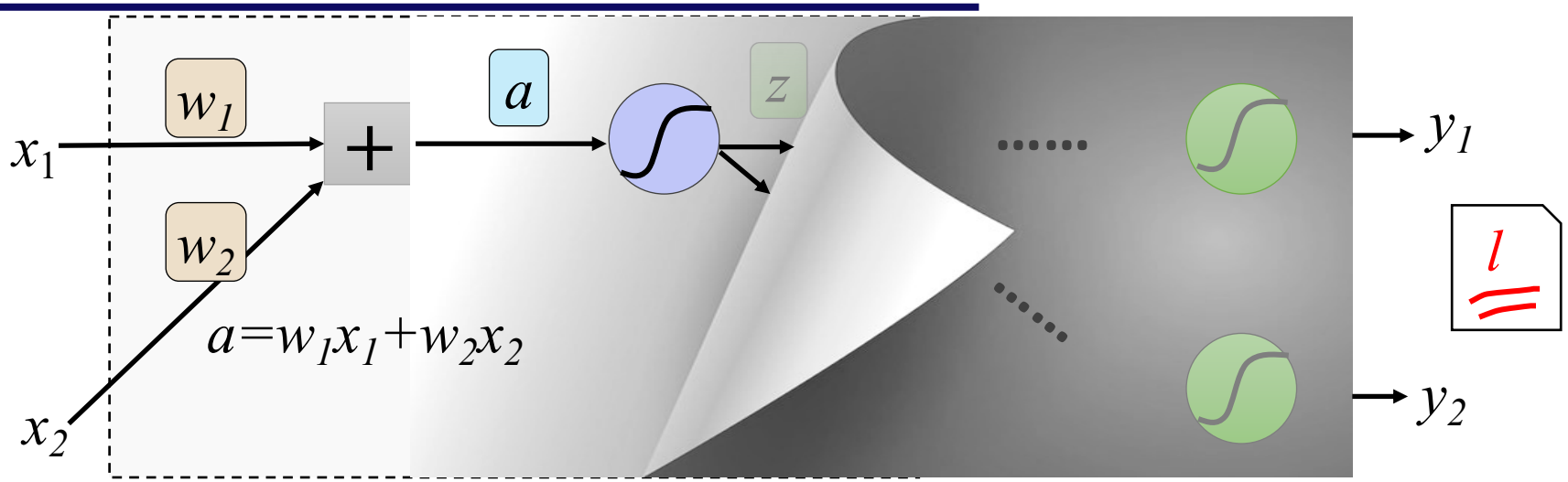
compute $\partial l / \partial a$ for all linearity outputs a recursively.

Backpropagation – Backward Pass

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



Backpropagation – Forward Pass



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

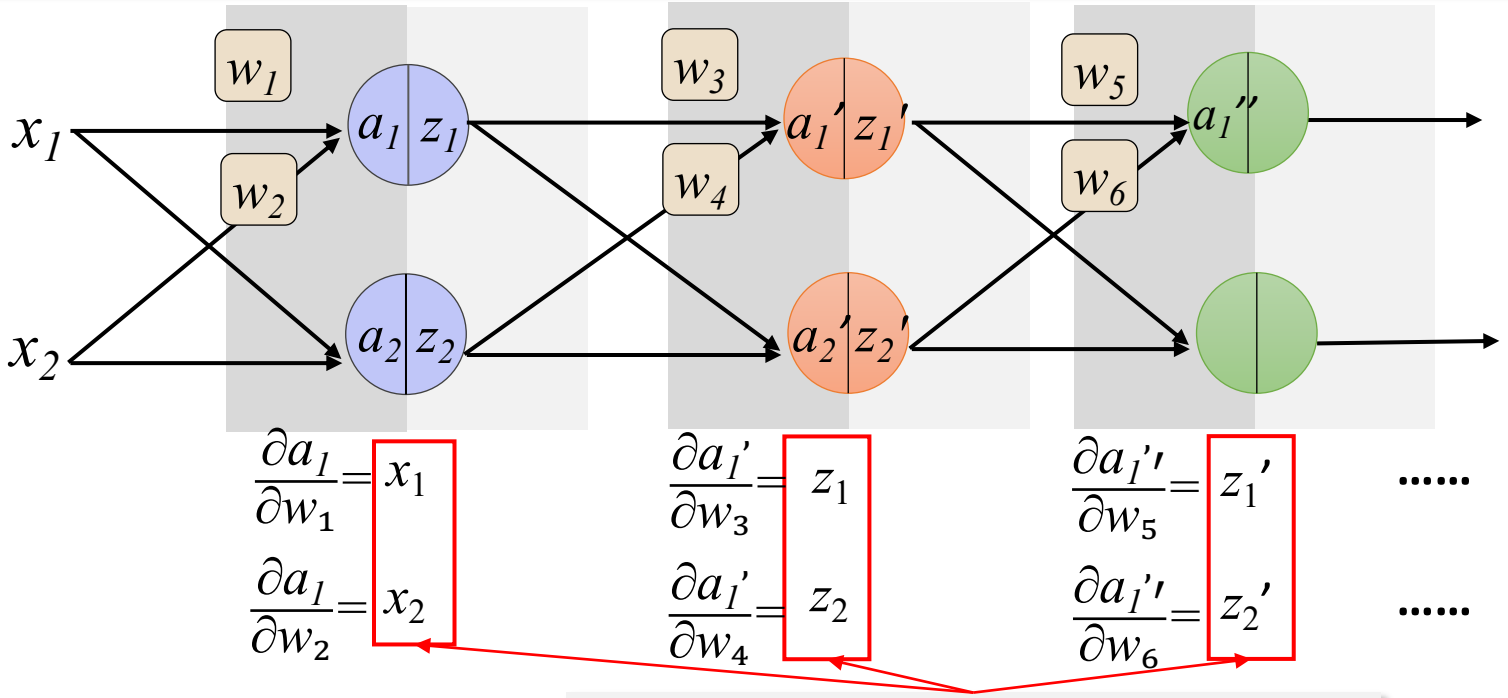
easy

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

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

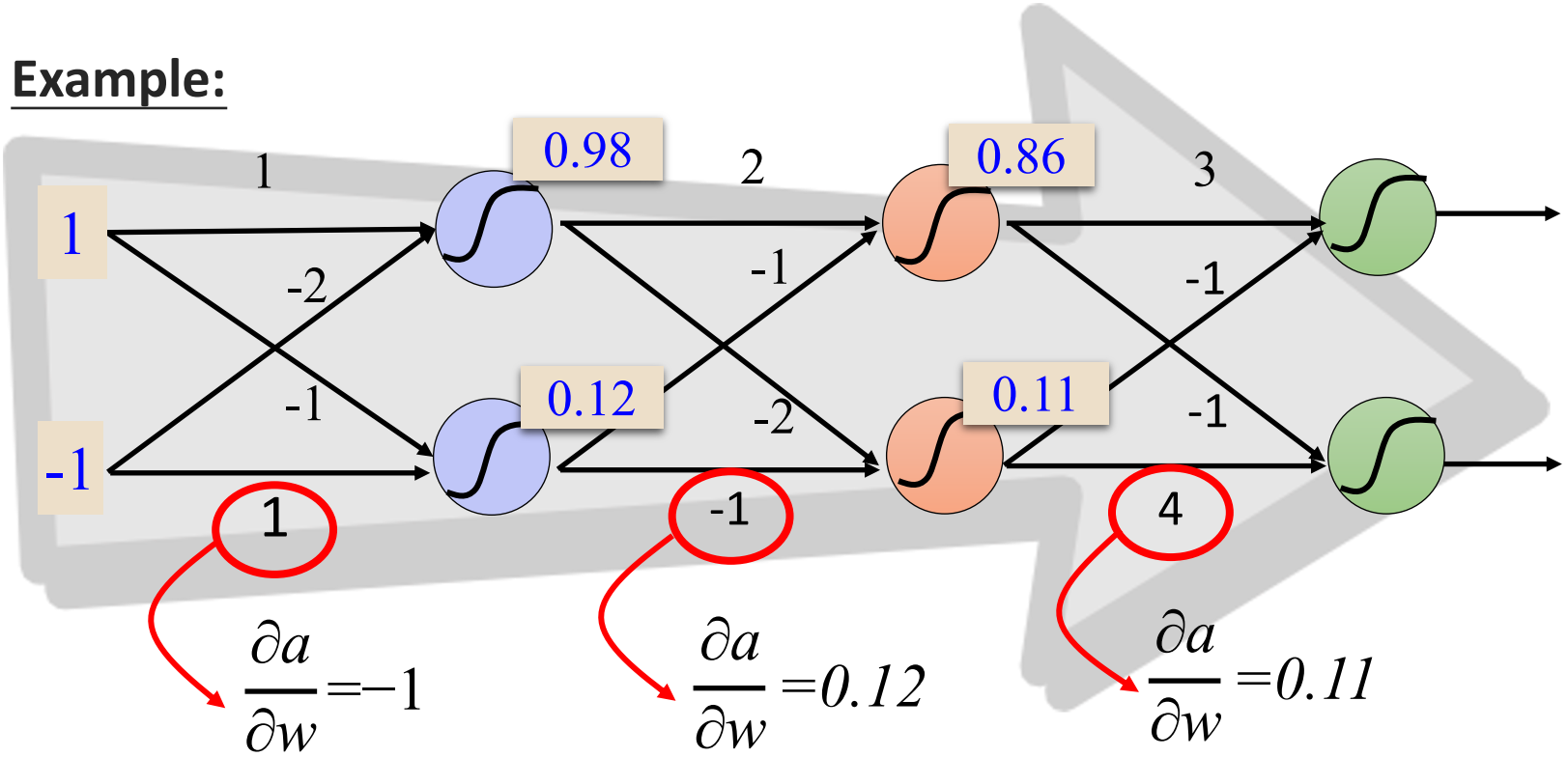
Idea: since the linear unit for each neuron is easy and fixed, we can pre-compute their gradients $\partial a/\partial w$ straightforward.

Backpropagation – Forward Pass

Forward pass:

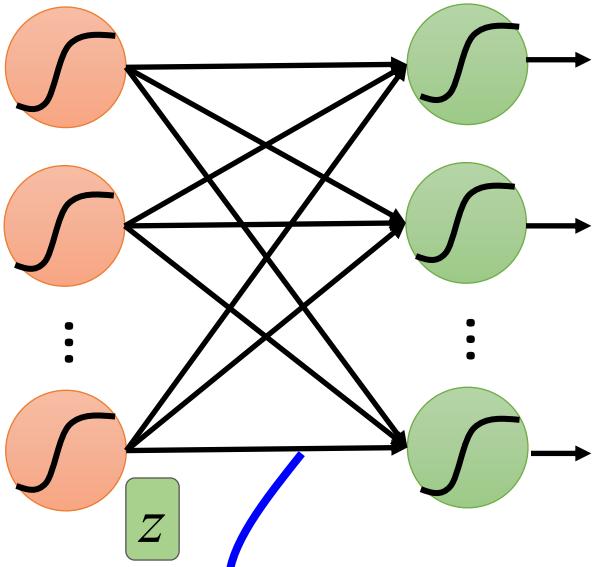
Before backward pass, run MLP forward, obtain $\partial a/\partial w$ $(=z)$ for all $\boldsymbol{w}$

Example:

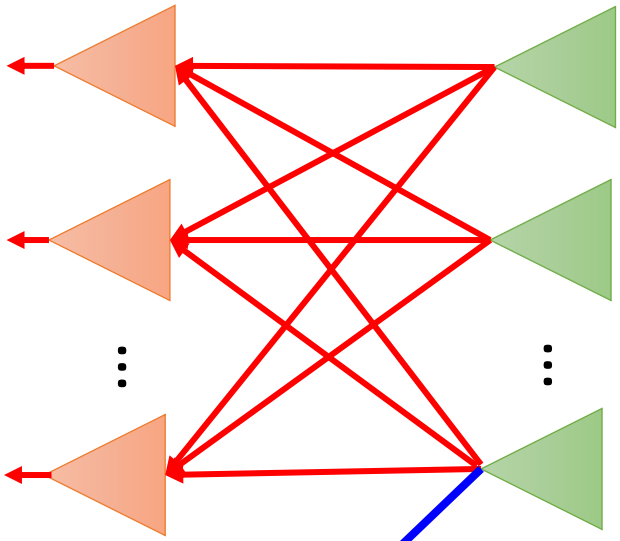


Backpropagation – Summary

Forward Pass



Backward Pass



$$\frac{\partial a}{\partial w} = z \qquad \times \qquad \frac{\partial l}{\partial a} \qquad = \frac{\partial l}{\partial w}$$

for all $\boldsymbol{w}$

算法总结

1. Forward propagation:

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

$$a_j = \sum_i w_{ji} z_i \qquad z_j = \sigma(a_j) \qquad j=1,\ldots d$$

2. Backward propagation:

- Evaluate the derivatives of the loss function with respect to the weights (errors).
- Errors are propagated backwards through the network.

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

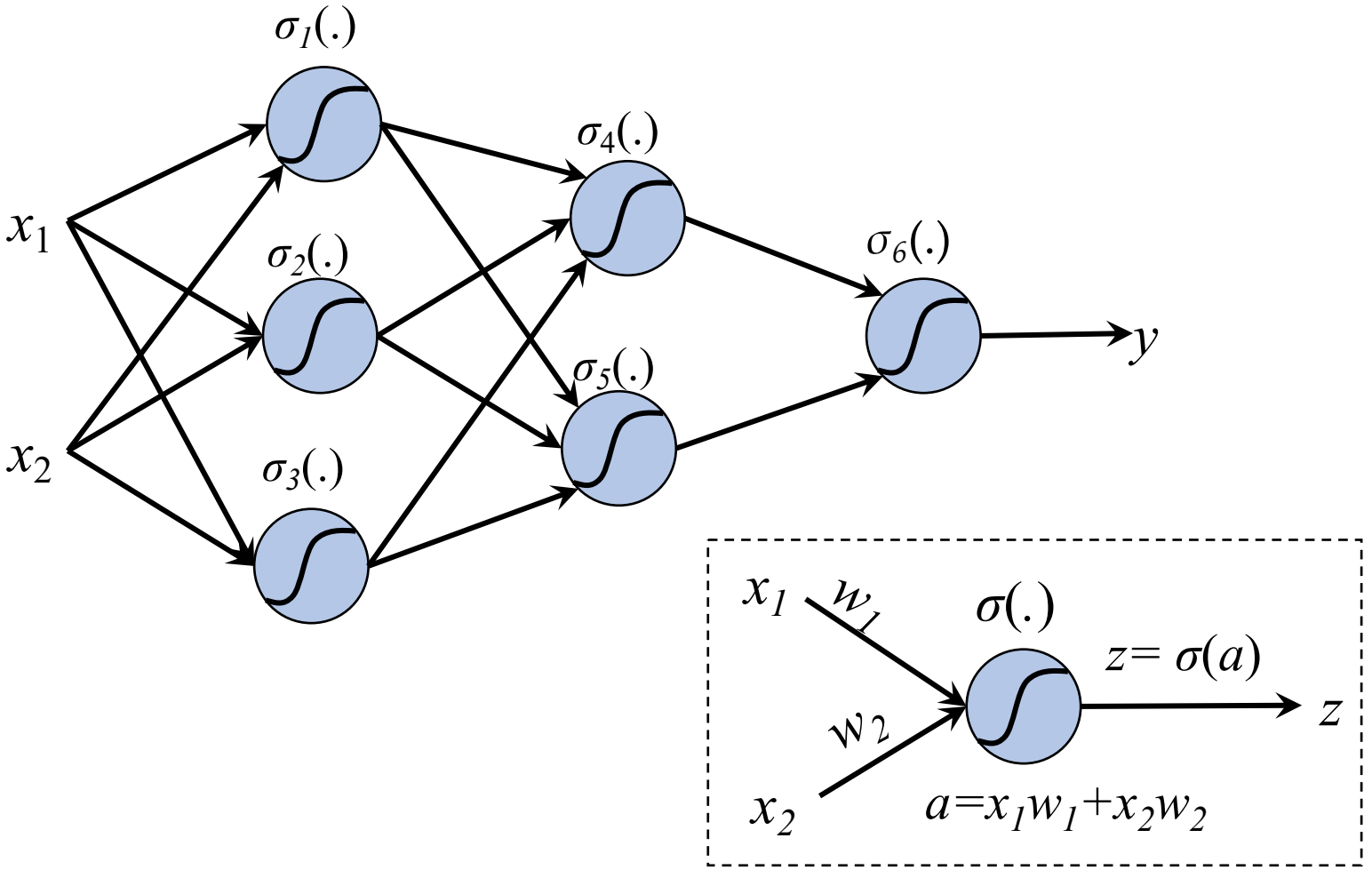
3. Parameter update:

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

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

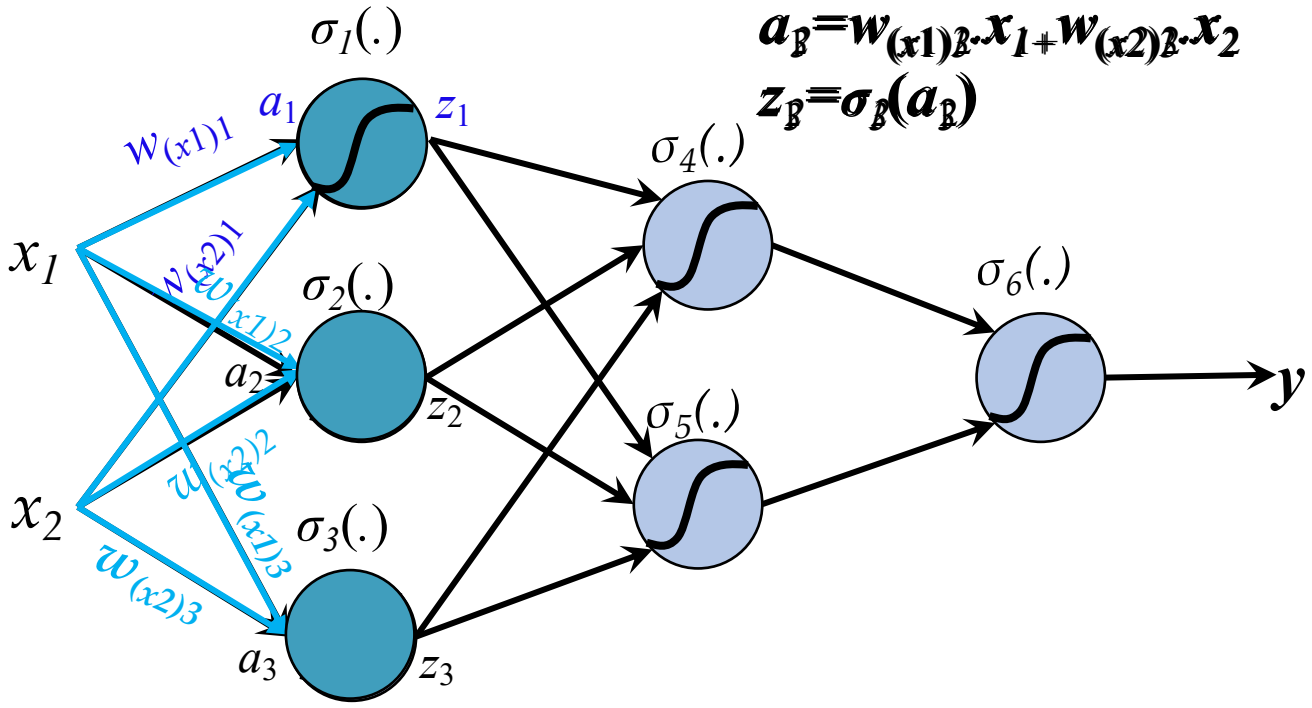
$$w'_{ij} = w_{ij} + \eta \delta_j z_i$$

BP Example

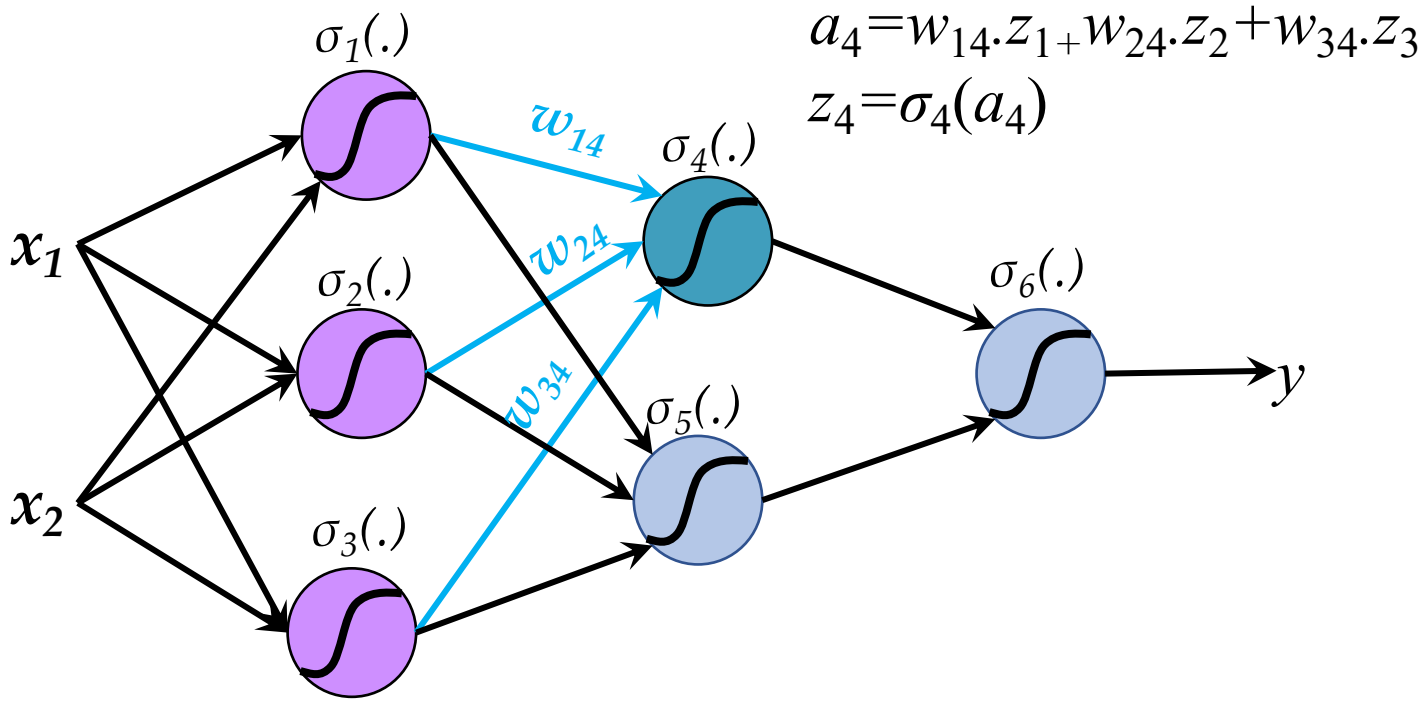

$\sigma_1(.)$
$\sigma_2(.)$
$\sigma_3(.)$
$\sigma_4(.)$
$\sigma_5(.)$
$\sigma_6(.)$
x_1
x_2
y
x_1
w_1
$\sigma(.)$
$z = \sigma(a)$
z
w_2
x_2
$a = x_1 w_1 + x_2 w_2$

BP Example

Forward Pass

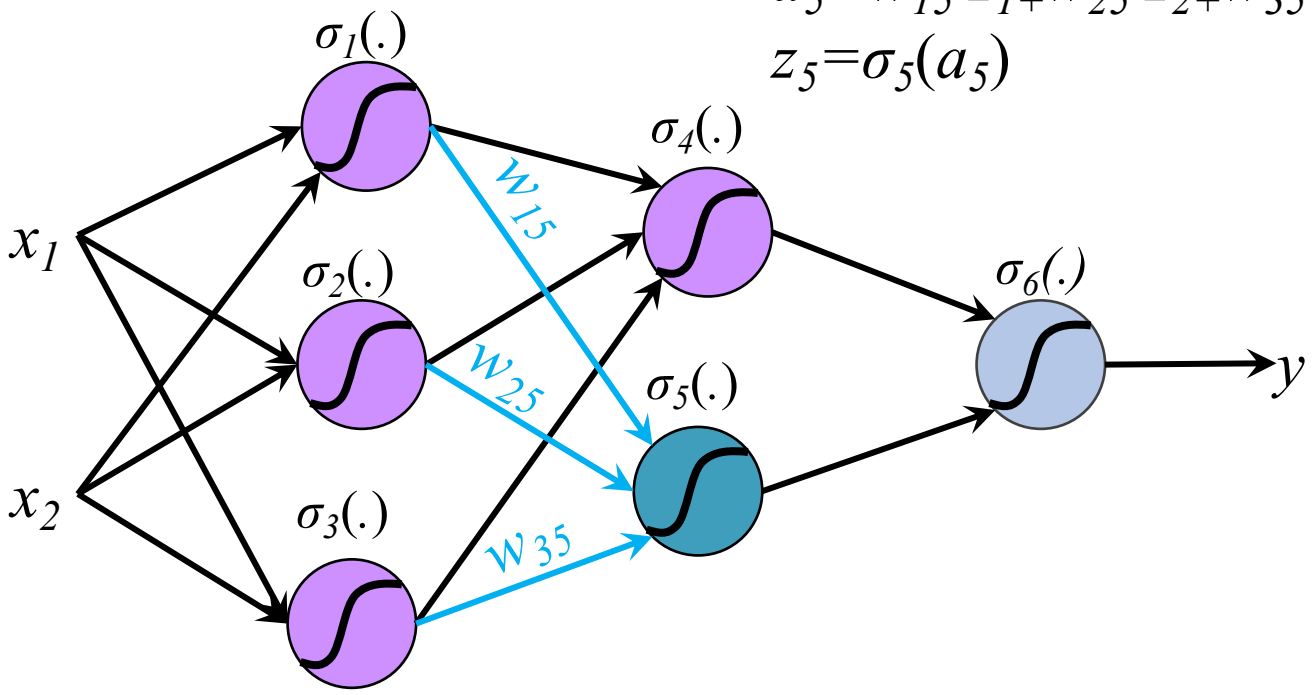


Forward Pass

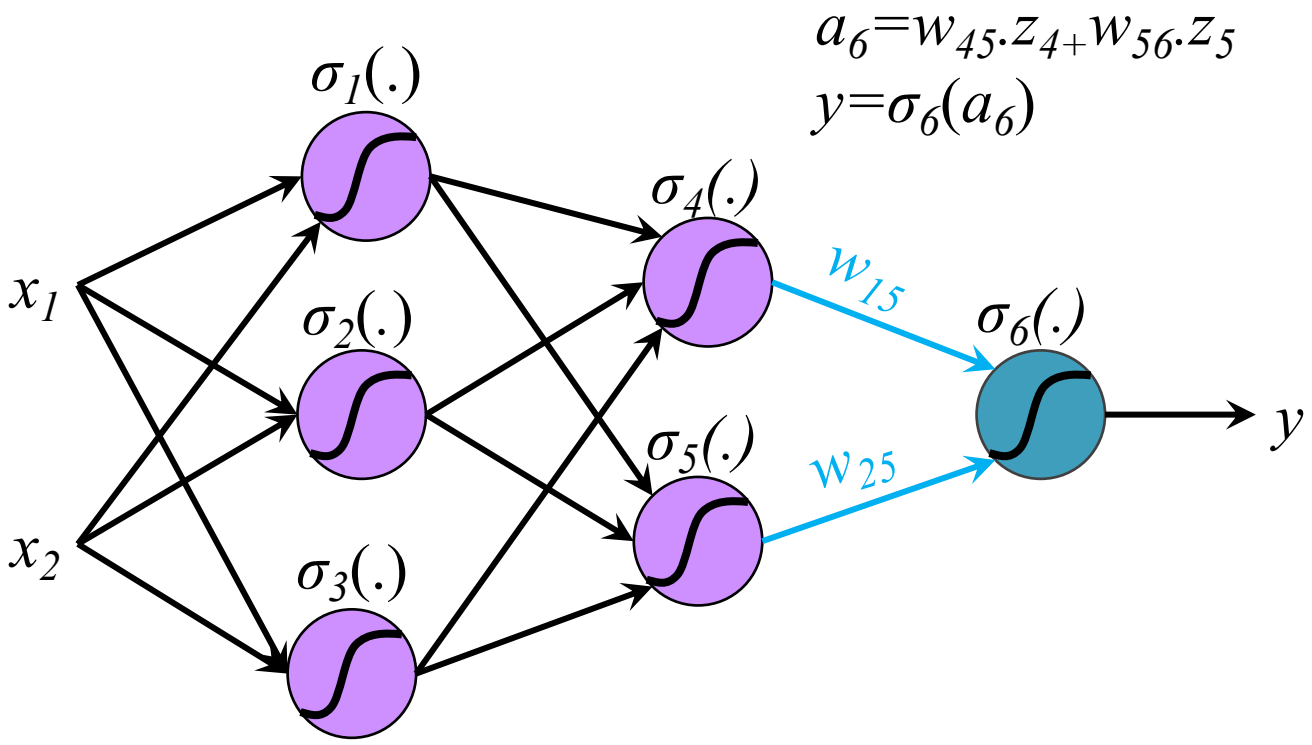


Forward Pass

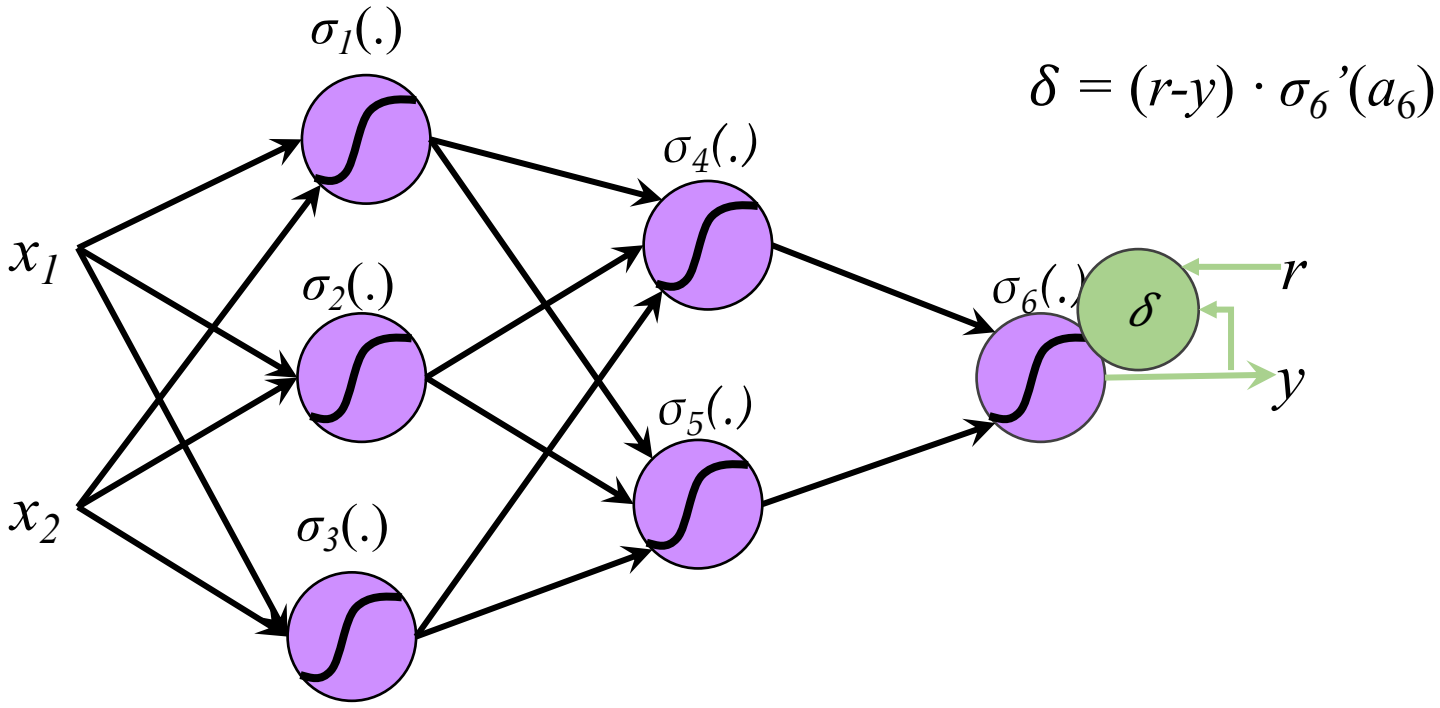
$$a_5 = w_{15} \cdot z_1 + w_{25} \cdot z_2 + w_{35} \cdot z_3$$
$$z_5 = \sigma_5(a_5)$$



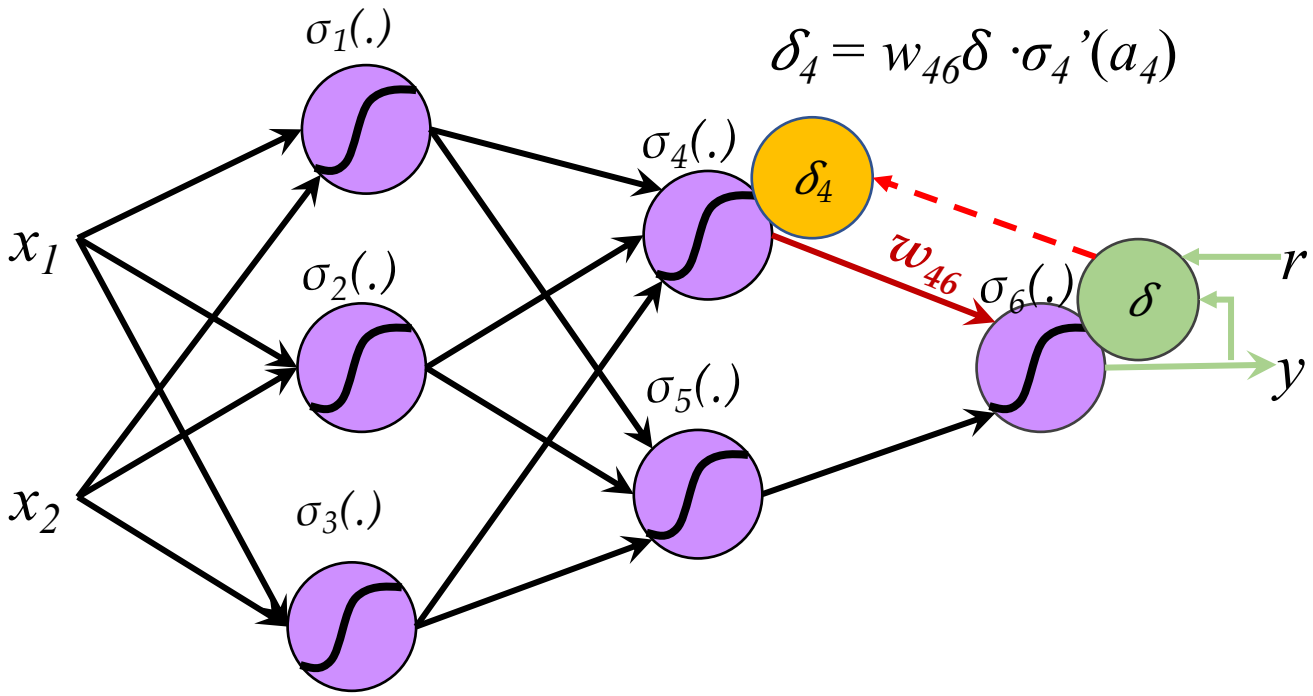
Forward Pass



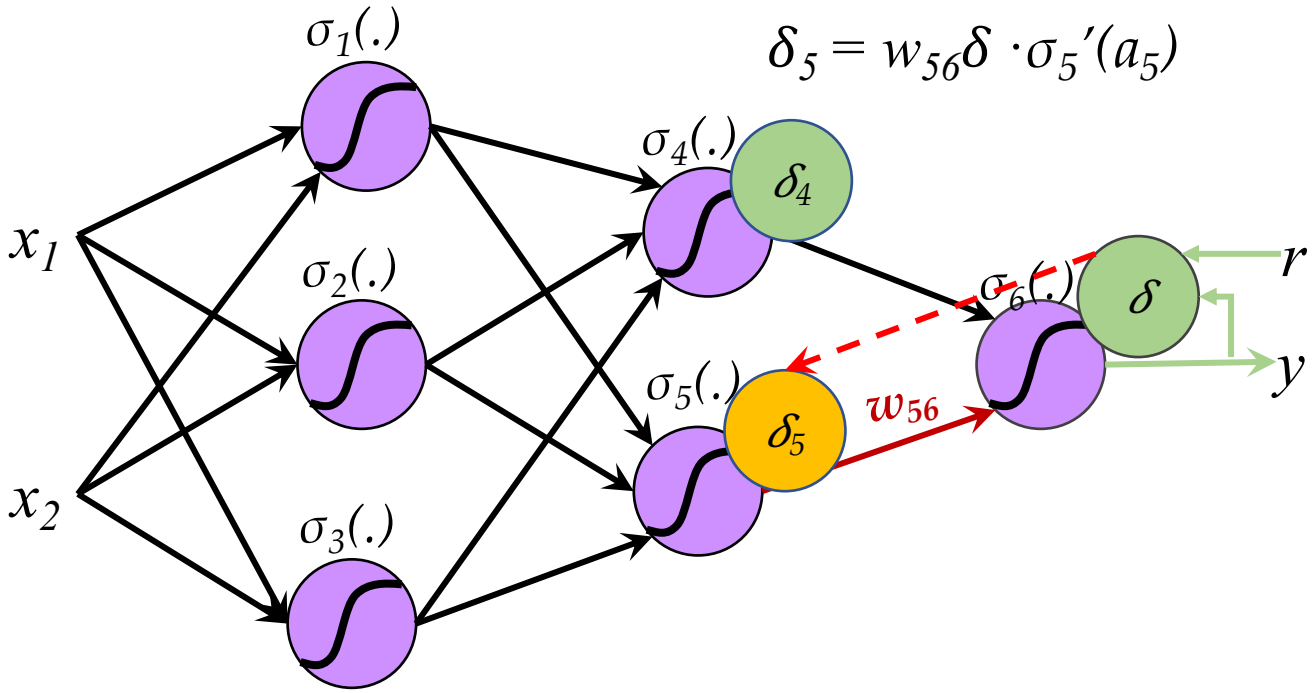
Backward Pass – Error propagation



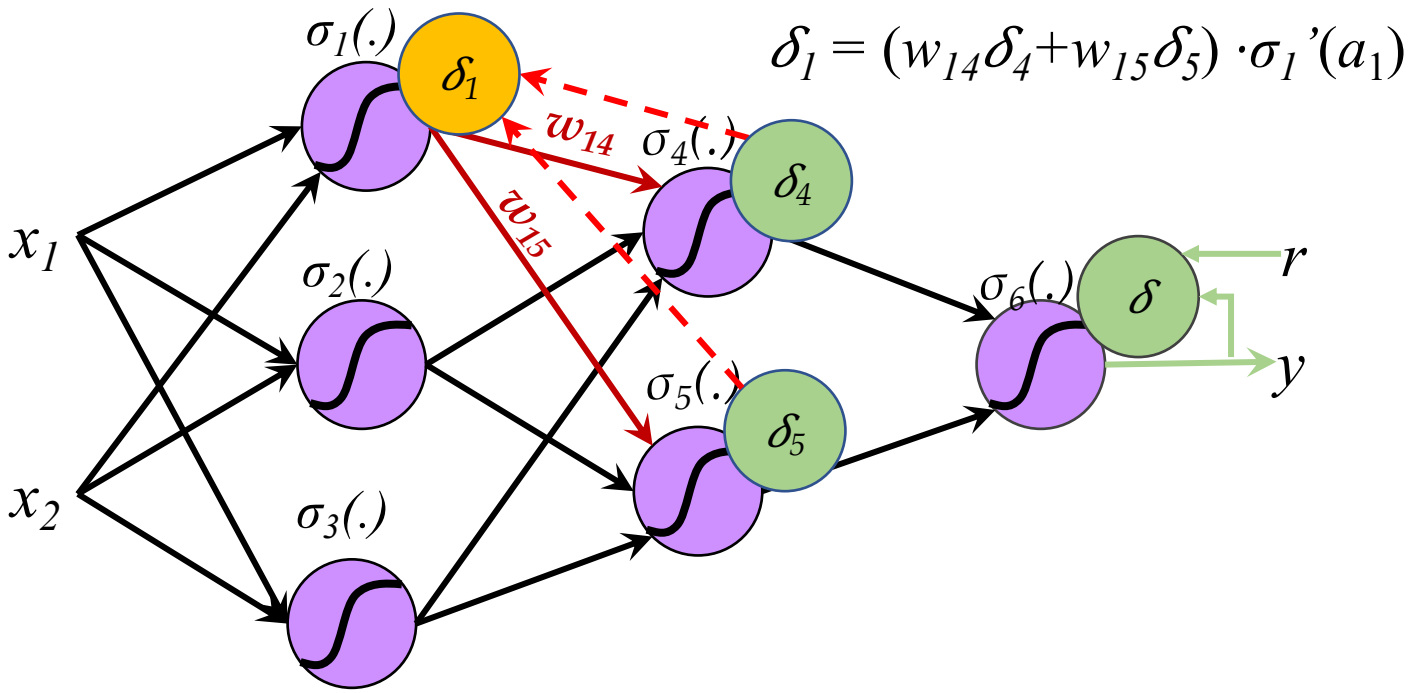
Backward Pass – error propagation



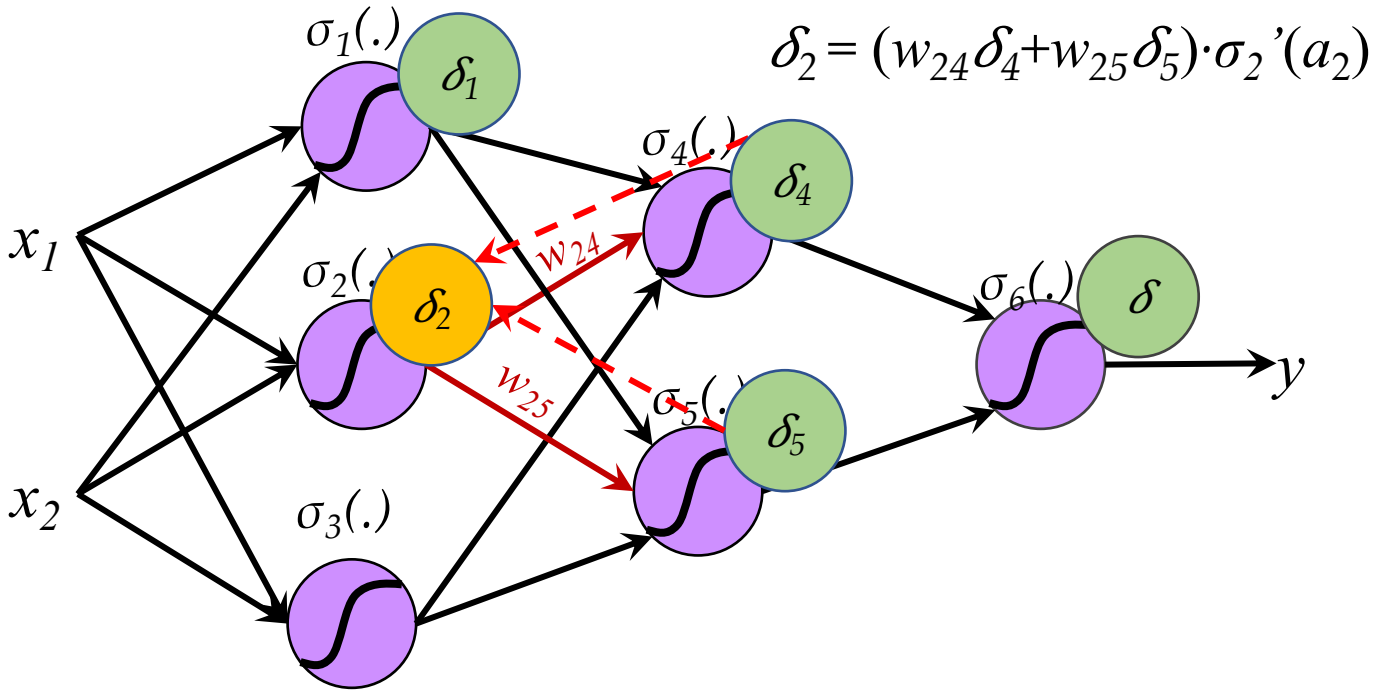
Backward Pass – error propagation



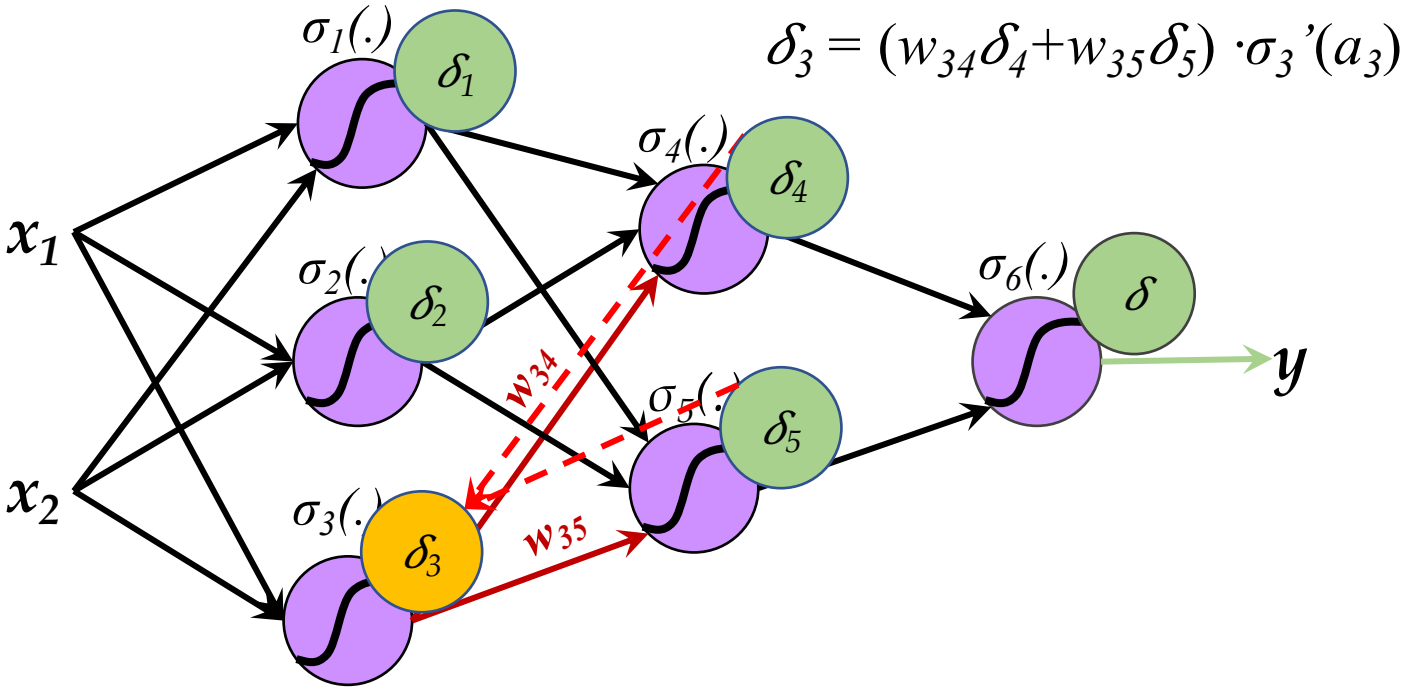
Backward Pass – error propagation



Backward Pass – error propagation



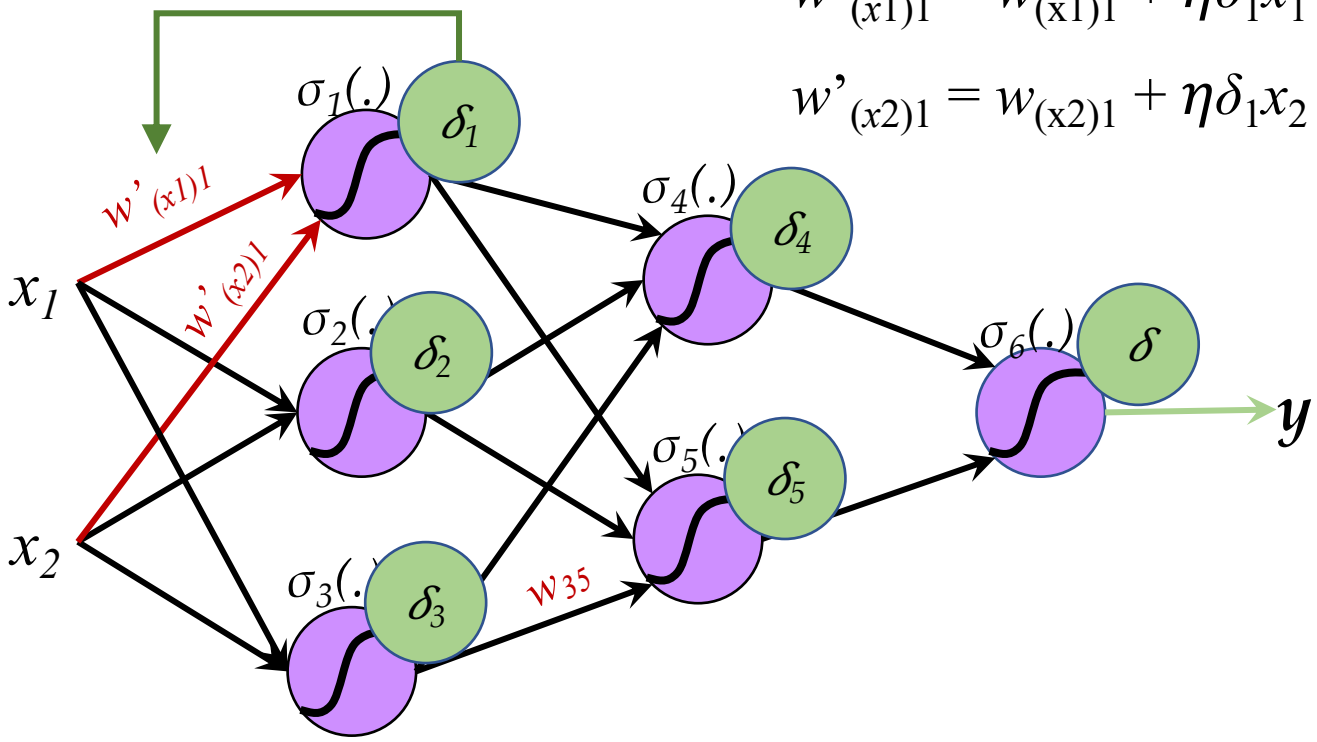
Backward Pass – error propagation



Optimization step – parameter update

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

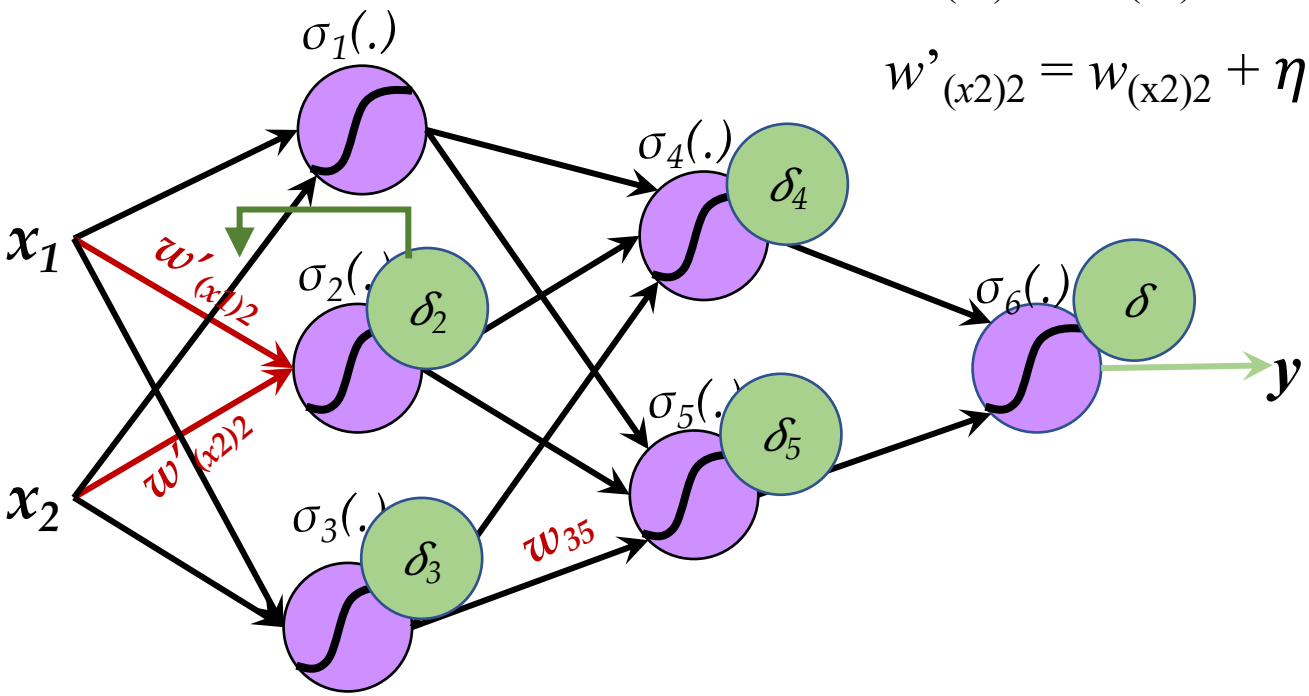
$$w'_{(x2)1} = w_{(x2)1} + \eta\delta_1 x_2$$



Optimization step – parameter update

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

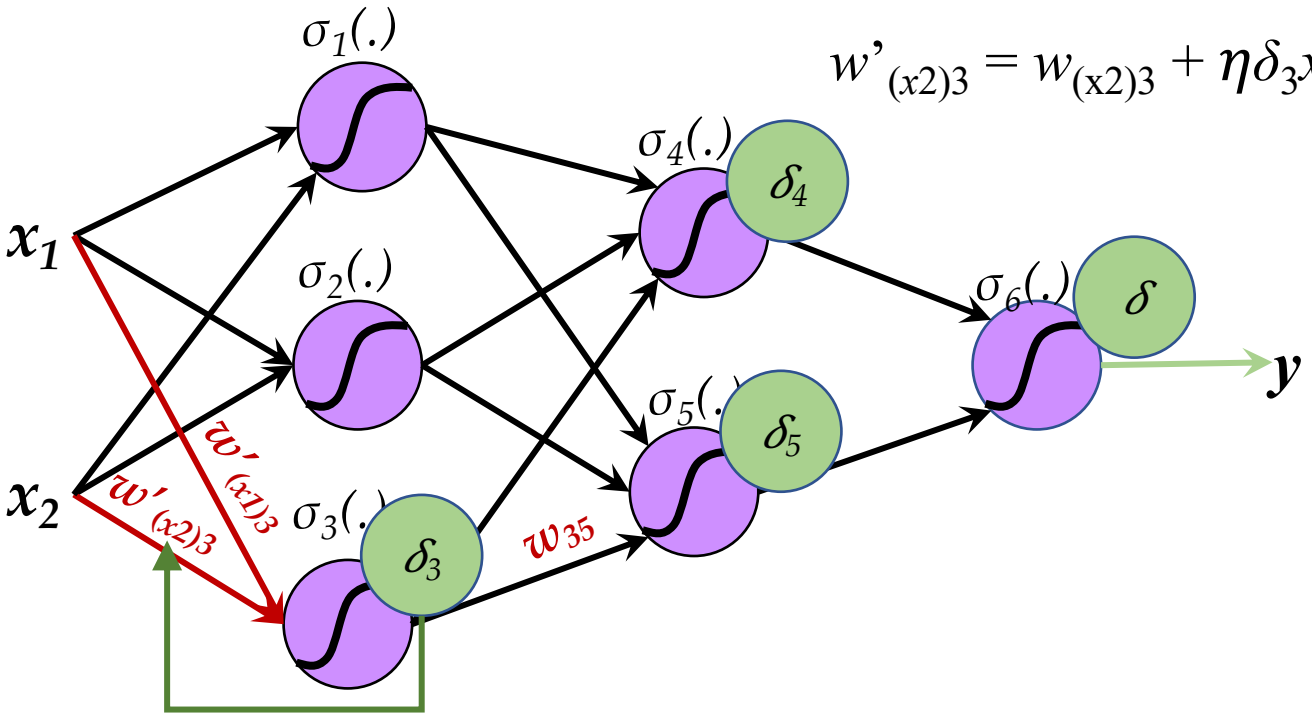
$$w'_{(x2)2} = w_{(x2)2} + \eta\delta_2 x_2$$



Optimization step – parameter update

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

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

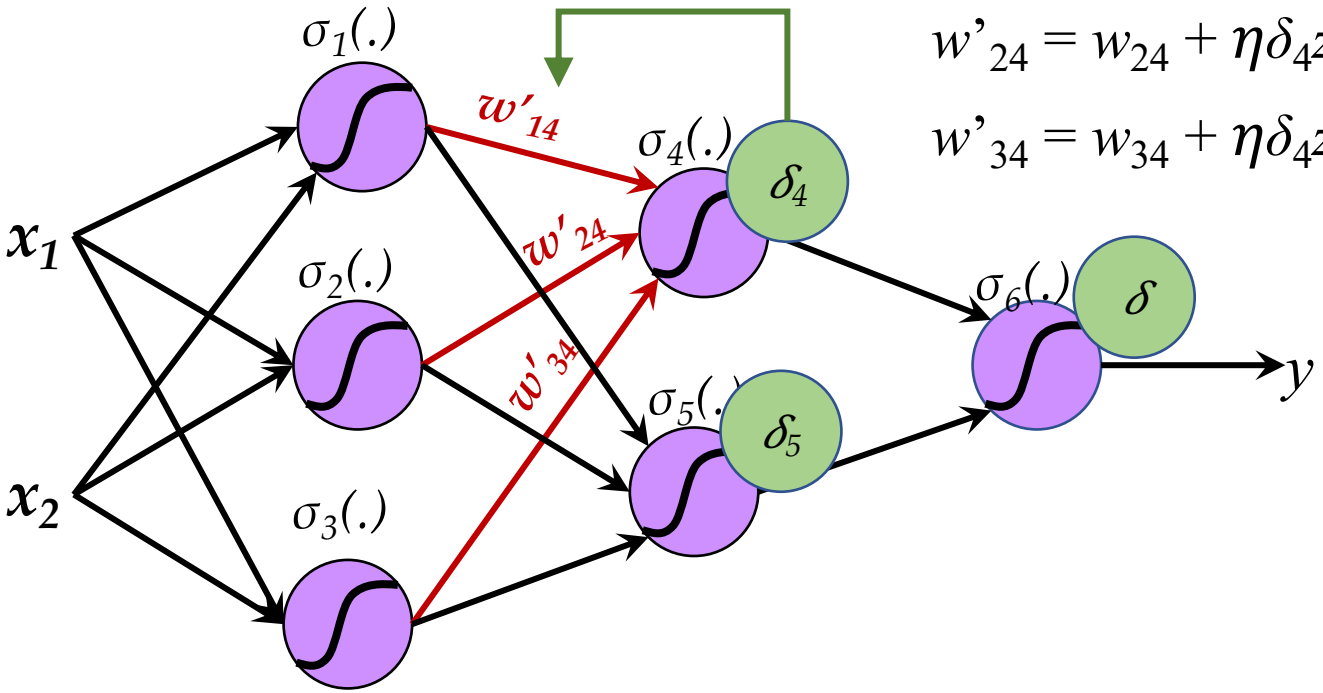


Optimization step – parameter update

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

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

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

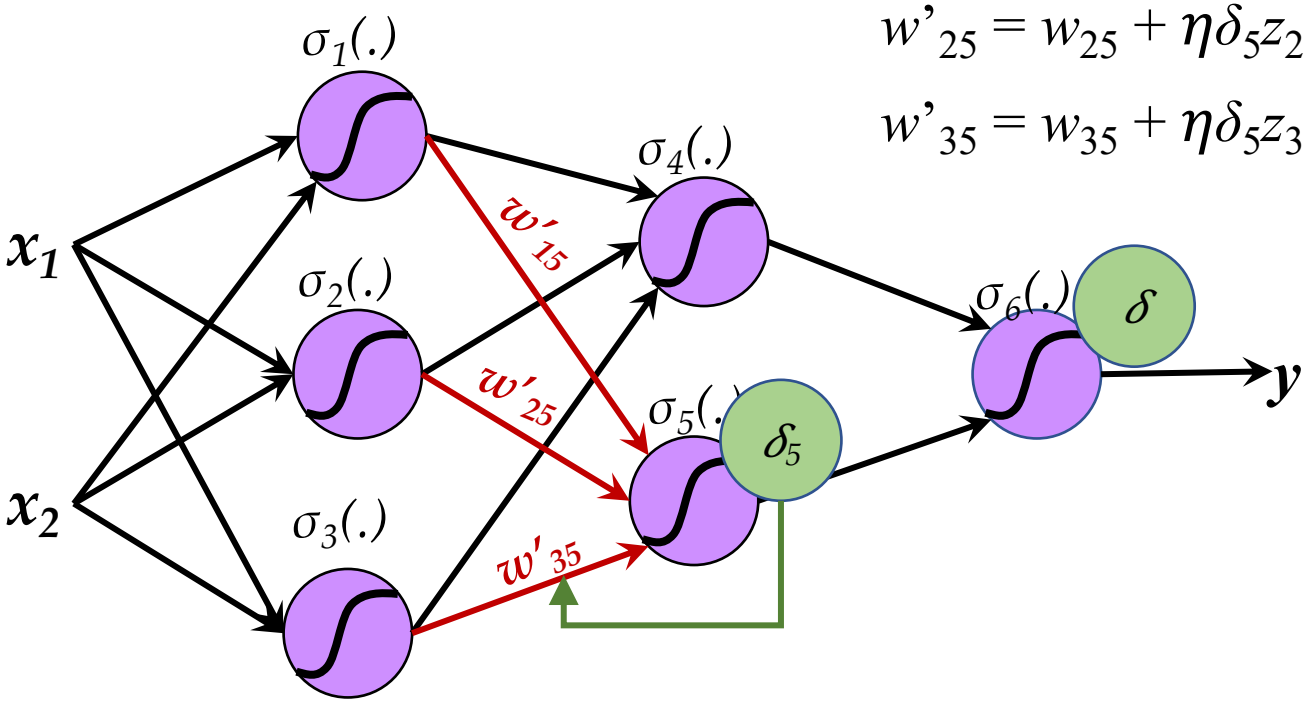


Optimization step – parameter update

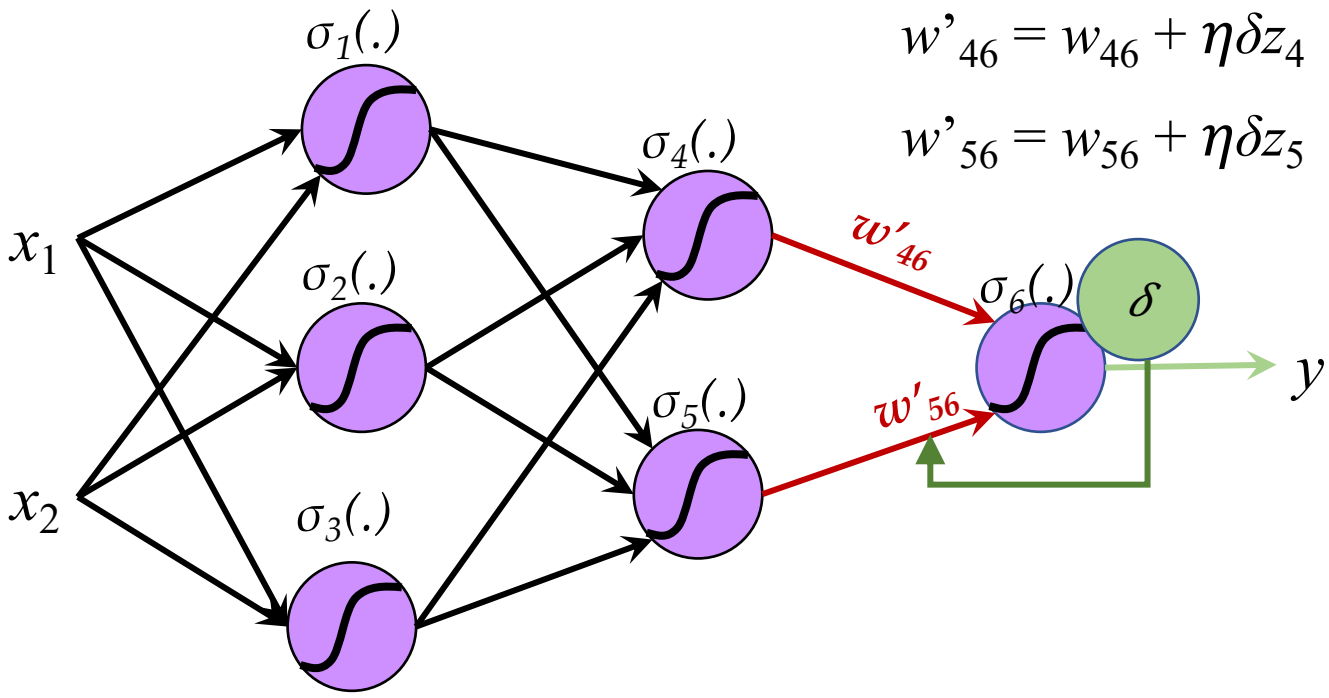
$$w'_{15} = w_{15} + \eta\delta_5 z_1$$

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

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



Optimization step – parameter update



Time for Coding

- **BP**:

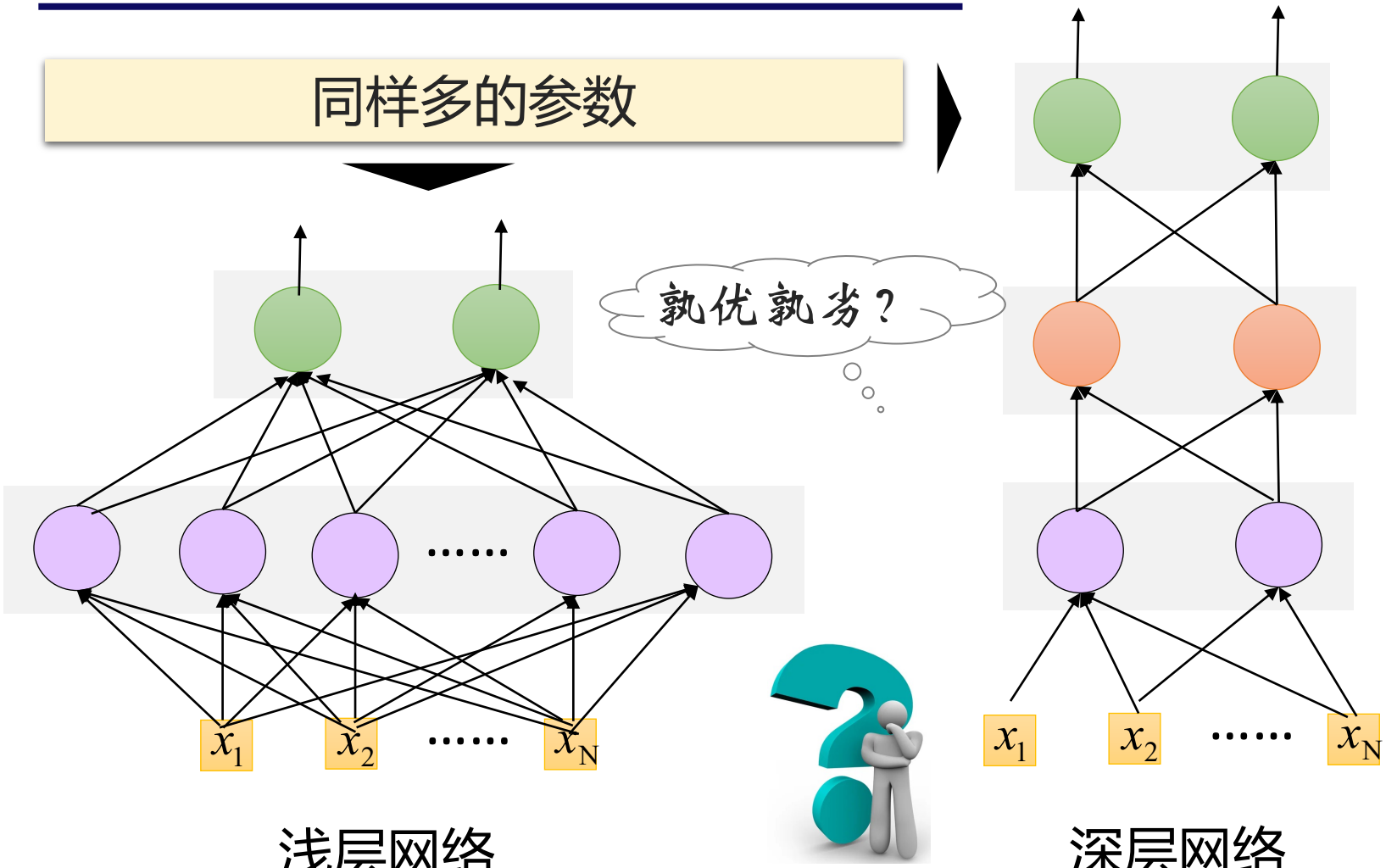
  https://www.kaggle.com/code/kaushal3663/backpropagation

- **MLP using PyTorch**

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

神经网络的深浅之争

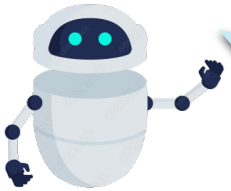
同样多的参数



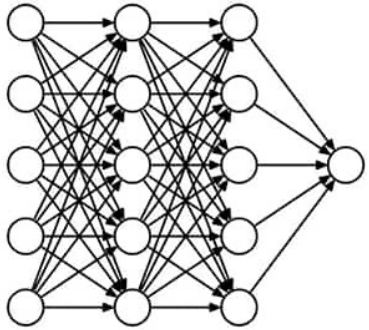
浅层网络

深层网络

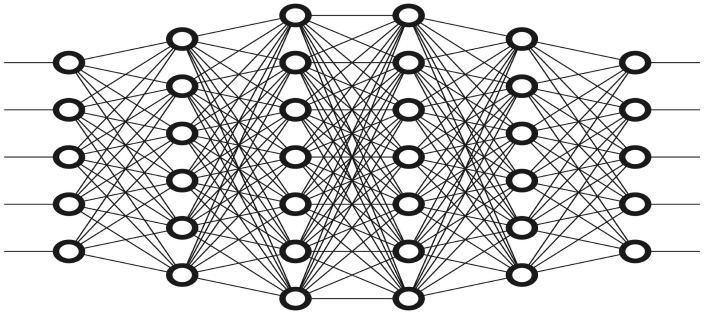
The Deeper, The Better

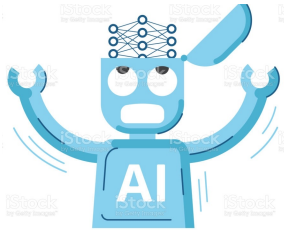


So, shallow networks can represent any functions.



However, using deep structures is more effective.





Reference for the reason: http://neuralnetworksanddeeplearning.com/chap4.html

深度学习（Deep Learning）

什么是深度学习？

深度学习(也叫深度表征学习, DL) 是一种机器学习技术，采用深层的神经网络实现对数据深层次特征的自动提取与表达

什么是深度学习？

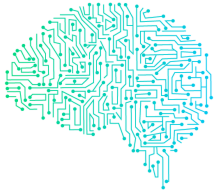
Artificial Intelligence

Computer systems that perform tasks that would usually require human intelligence

Machine Learning

Statistical techniques that learn from a series of inputs and outputs

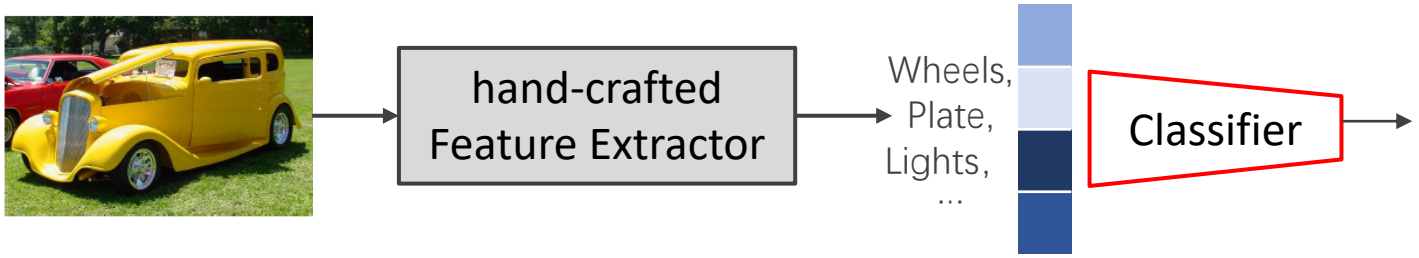
Deep Learning



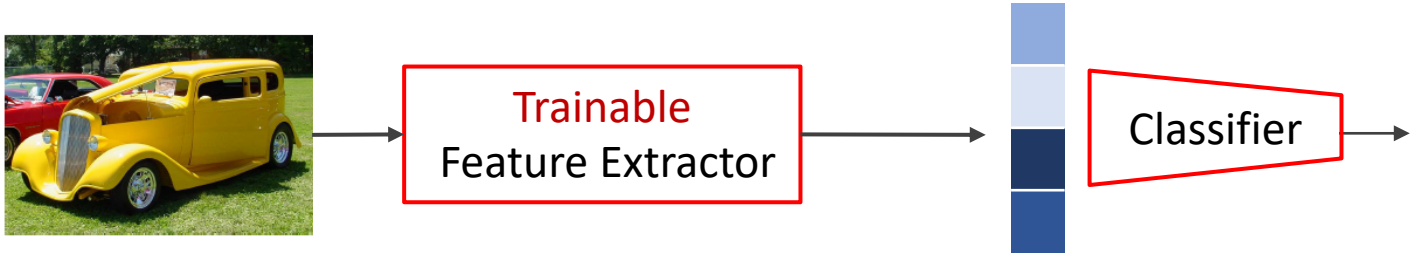
Machine learning algorithms that enable self learning of **data representations**

深度学习 = 学习特征表征

- 传统机器学习或模式识别技术
  - ▶ **fixed/engineered** features + **trainable** classifier

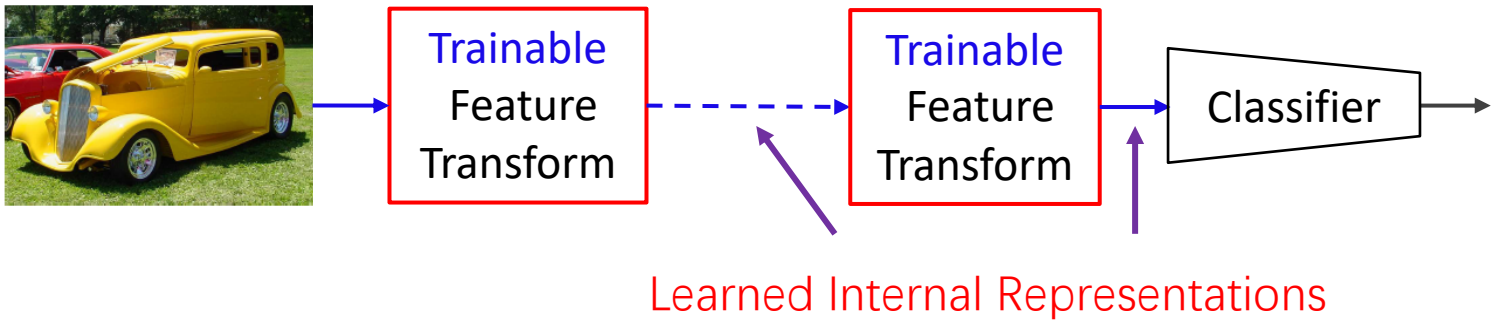


- 深度学习
  - ▶ **trainable** features + **trainable** classifier



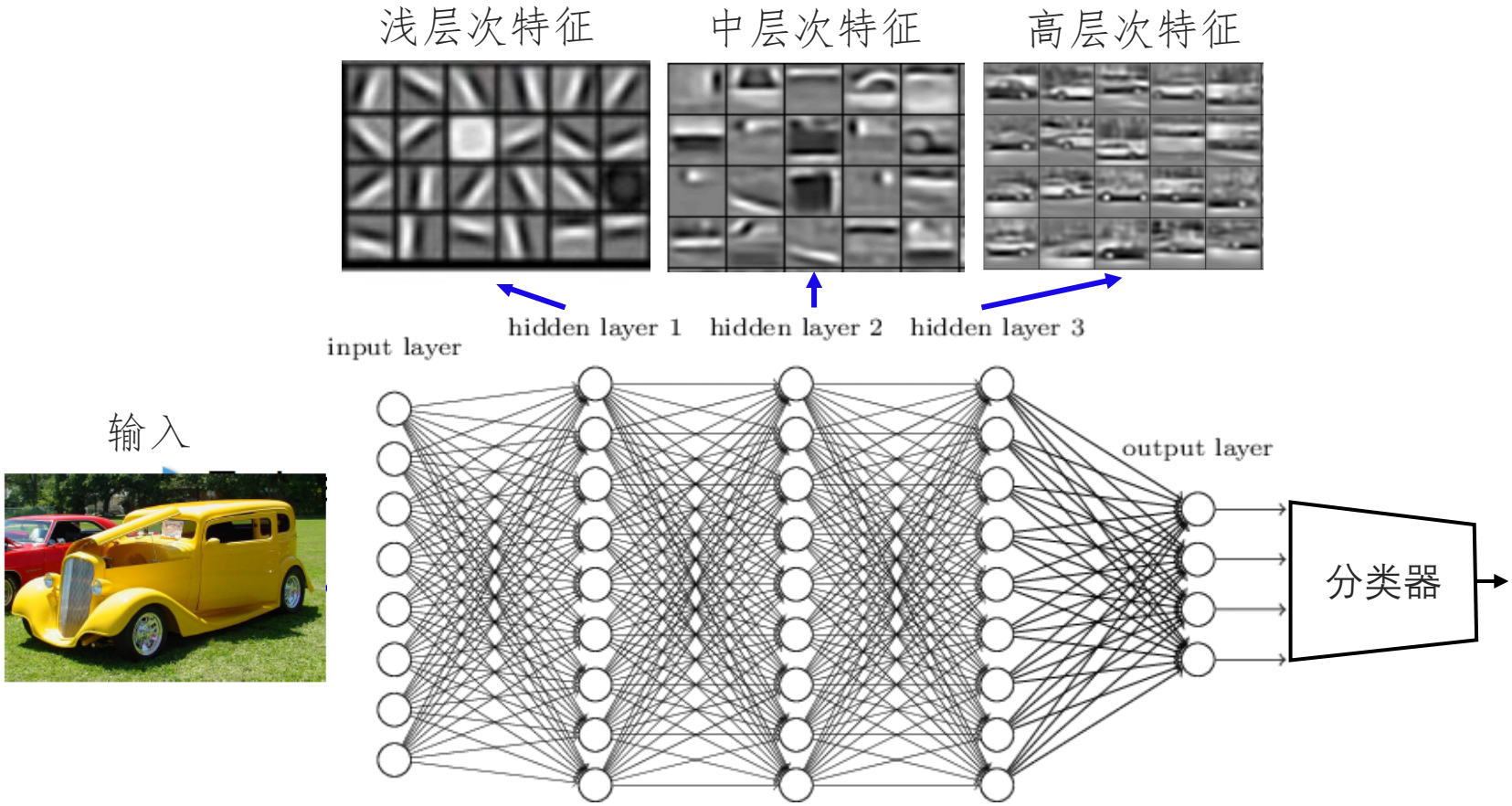
深度学习 = 学习数据的层次化表征

- A hierarchy of trainable feature transforms



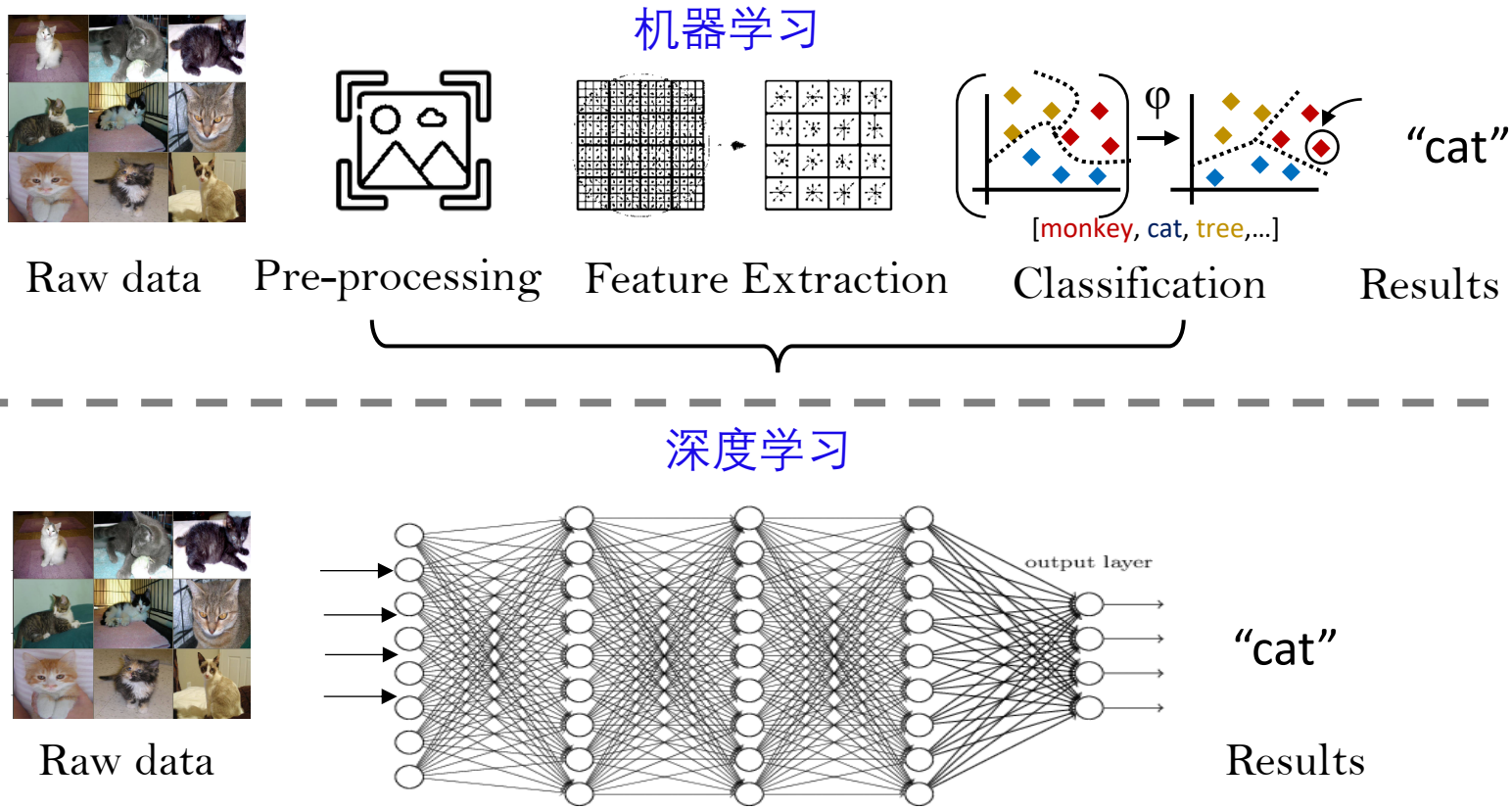
- Each module transforms its input representations into a higher-level one.
- High-level features are more global and more invariant
- Low-level features are shared among categories

深度学习 = 学习数据的层次化表征



深度学习 = 端到端学习(End-to-end Learning)

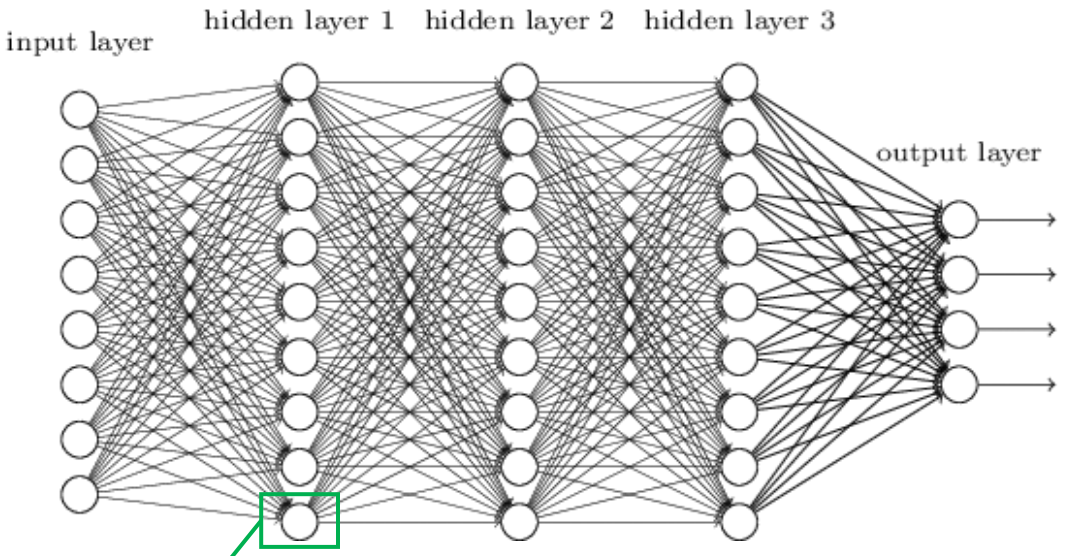
- 模型学习从输入端到输出端的所有操作。



典型的深度神经网络

- 全连接神经网络
- 卷积神经网络
- 循环神经网络
- …

深度(前向)神经网络

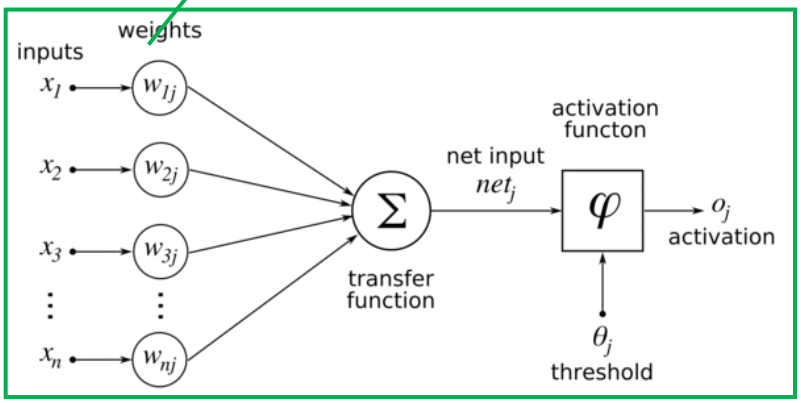


损失函数:

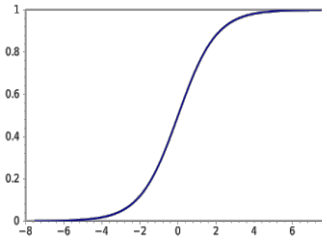
- Square error
- Cross entropy
- Hinge loss
- Ranking loss
- …

训练:

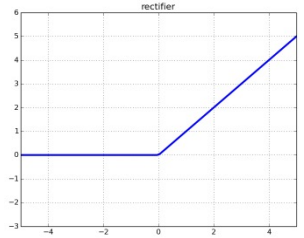
Backpropagation



Activation functions

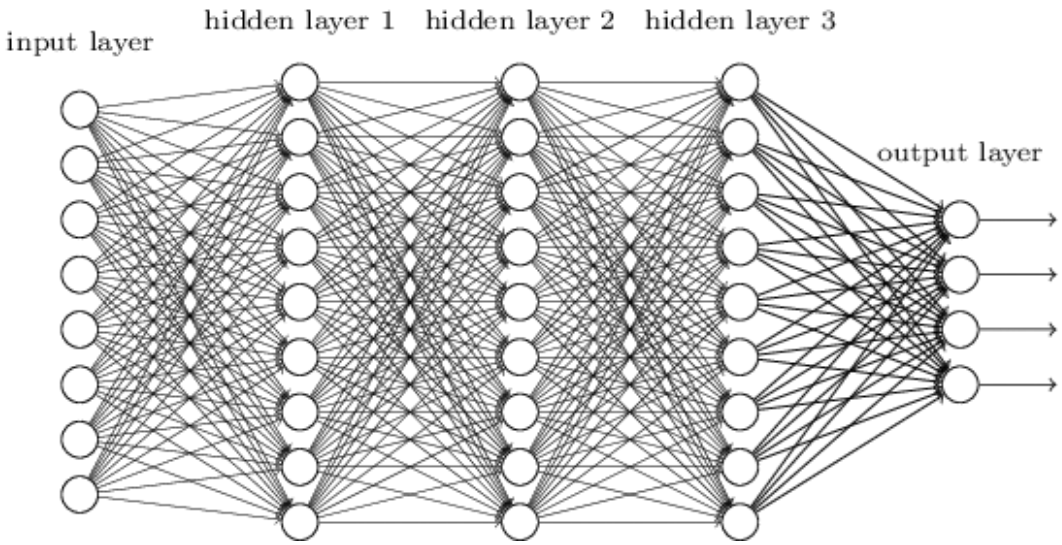


Sigmoid



ReLU

前向神经网络的缺点



前向DNN通常不单独作为模型使用

- 难训练 (too many parameters, gradient vanishing, etc.)
- 输入敏感
- …

而经常用于中间层，作为辅助单元 (e.g., nonlinear transformation)

现实中常用的DNN

卷积神经网络 (CNN)

循环神经网络 (RNN)

Transformer

…