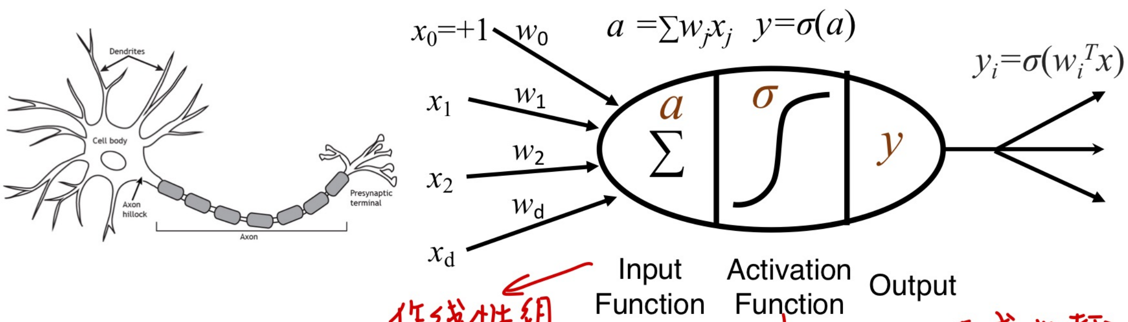
ch8: Multi-layer Perceptrons

Artificial Neural Networks

Perceptron(P6)

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



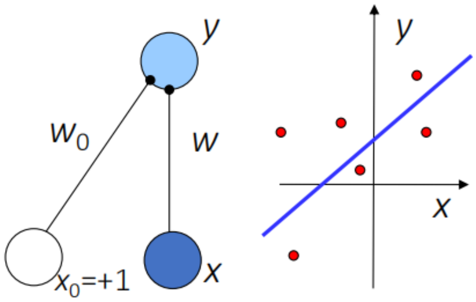
作线性组合

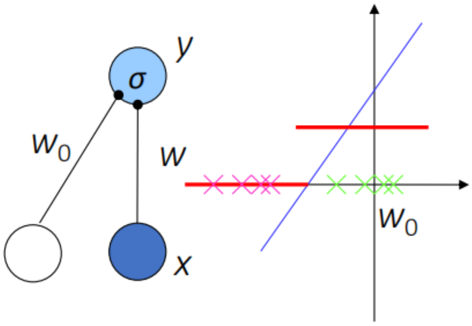
激活函数：是否感兴趣

- 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.

Perceptron用途（P7）

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





- 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:

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

Training a Perceptron（P8）

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

- 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

limitations and solutions(P11-14)

Limitation1: perceptron cannot learn data that are not linearly separable

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

Limitation2: thresholding functions are discrete and non-differentiable.

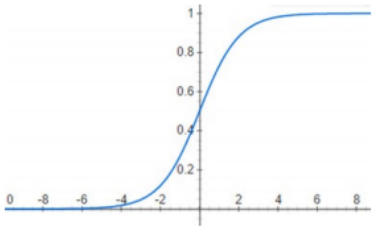
Continuous Thresholding(activation functions)(P14)

- 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):

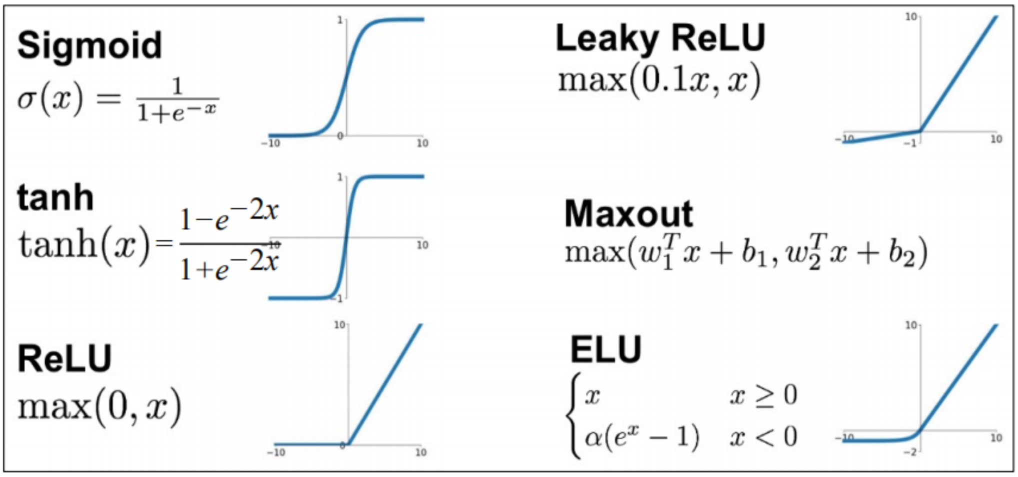
$$y = \text{sigmoid}(\mathbf{w}^T \mathbf{x})$$



- 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.

- other **Activation Functions**

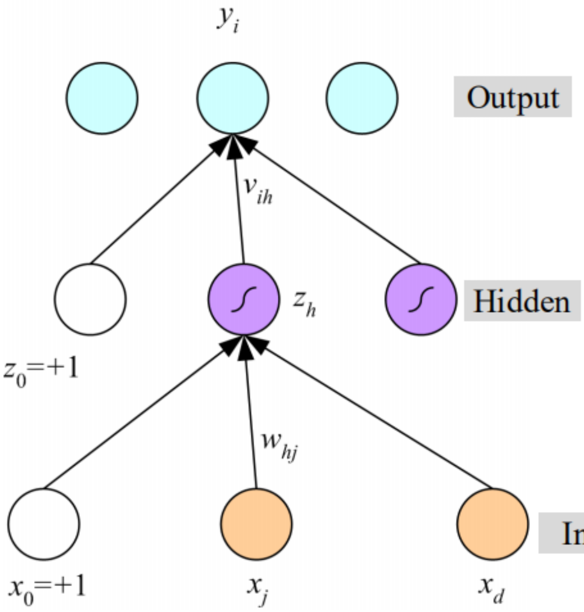
- In general, we can apply many other continuous thresholding functions, called **activation functions**, to introduce nonlinearity in perceptrons.



Multi-layer Perceptrons

definition(P17)

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



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.

What an MLP does(P21-22)

train an MLP

Optimizing MLP using GD (P24-31)

Network Parameters: $\theta = \{\underline{w_0, w_1, \ldots, w_n}\}$ 参数量很大

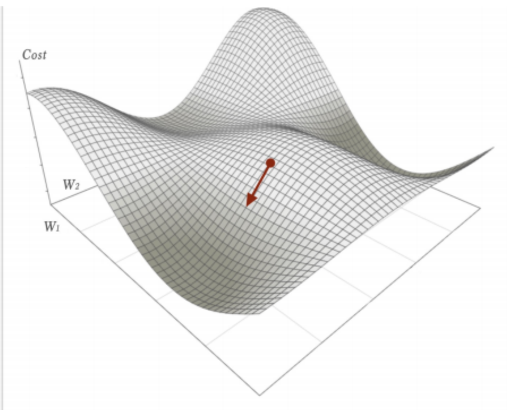
$\theta^0 \rightarrow \theta^1 \rightarrow \theta^2 \rightarrow \ldots$

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

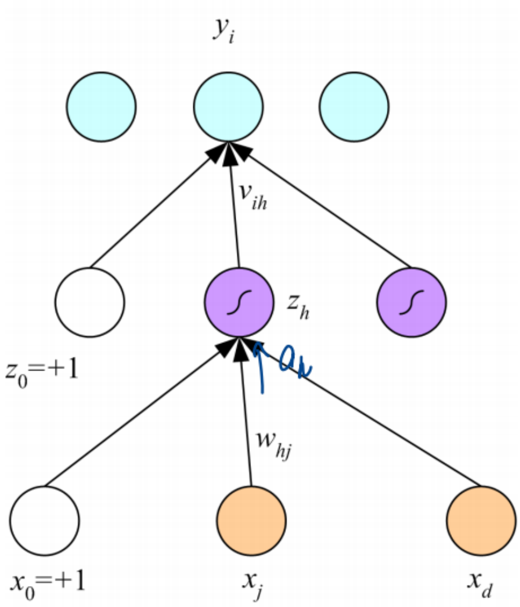
$$\theta^2 = \theta^1 - \eta \nabla L(\theta^1)$$

$$\vdots$$

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



Gradient Descend for 2-Layer MLP



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

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

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

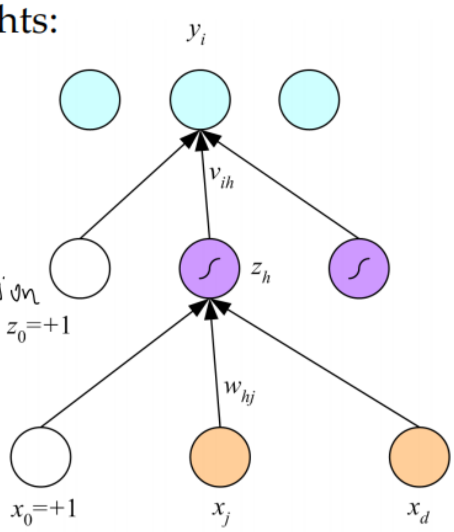
a_h表示神经元h拿到的分数

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

损失函数

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

- Update rule for second-layer weights:



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

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

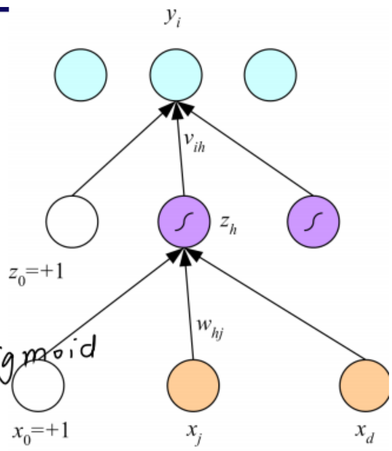
$$= \eta \sum_\ell \sum_k \frac{r_k^{(\ell)}}{y_k^{(\ell)}} \frac{\partial y_k^{(\ell)}}{\partial o_i^{(\ell)}} \frac{\partial o_i^{(\ell)}}{\partial v_{ih}}$$

在sigmoid regression的结果

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

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

- Update rule for first-layer weights:



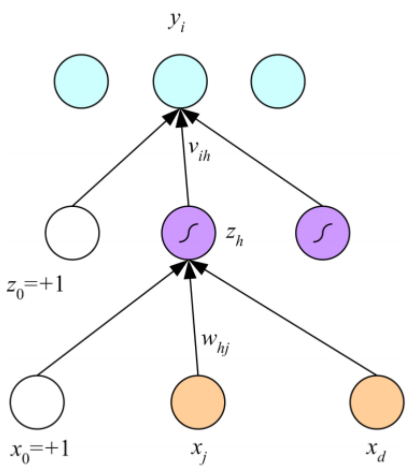
$$
\begin{aligned}
\Delta w_{hj} &= -\eta \frac{\partial L}{\partial w_{hj}} \\
&= \eta \sum_{\ell} \sum_{k} \frac{r_k^{(\ell)}}{y_k^{(\ell)}} \frac{\partial y_k^{(\ell)}}{\partial z_h^{(\ell)}} \frac{\partial z_h^{(\ell)}}{\partial w_{hj}} \\
&= \eta \sum_{\ell} \sum_{k} \sum_{i} \frac{r_k^{(\ell)}}{y_k^{(\ell)}} \frac{\partial y_k^{(\ell)}}{\partial o_i^{(\ell)}} \frac{\partial o_k^{(\ell)}}{\partial z_h^{(\ell)}} \frac{\partial z_h^{(\ell)}}{\partial a_h^{(\ell)}} \frac{\partial a_h^{(\ell)}}{\partial w_{hj}} \\
&= \eta \sum_{\ell} \sum_{k} \sum_{i} \frac{r_k^{(\ell)}}{y_k^{(\ell)}} y_k^{(\ell)} \left(\delta_{ki} - y_k^{(\ell)}\right) v_{ih} \underline{z_h^{(\ell)} \left(1 - z_h^{(\ell)}\right)} x_j^{(\ell)} \\
&= \eta \sum_{\ell} \left[\sum_{i} \left(r_i^{(\ell)} - y_i^{(\ell)}\right) v_{ih} \right] z_h^{(\ell)} \left(1 - z_h^{(\ell)}\right) x_j^{(\ell)}
\end{aligned}
$$

sigmoid

- **Algorithm**

Gradient Descend for 2-Layer MLP

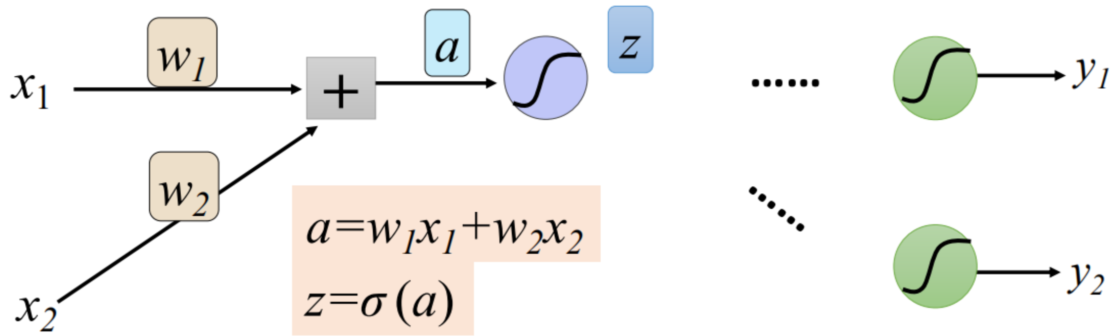
Initialize all v_{ih} and w_{hj} to rand$(-0.01, 0.01)$
Repeat
    For all $(\boldsymbol{x}^t, r^t) \in \mathcal{X}$ in random order
        For $h = 1, \ldots, H$
            $z_h \leftarrow \text{sigmoid}(\boldsymbol{w}_h^T \boldsymbol{x}^t)$
        For $i = 1, \ldots, K$
            $y_i = \boldsymbol{v}_i^T \boldsymbol{z}$
        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



- **Problem:** Straightforward derivation of gradients in MLP is tedious, inflexible and often infeasible, due to the dependences between gradients(由于梯度之间的依赖性，在MLP中直接推导梯度是繁琐的、不灵活的，而且往往是不可行的)

Backpropagation

- overview(P34)



前项　反项

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

(Chain rule)

Forward pass:

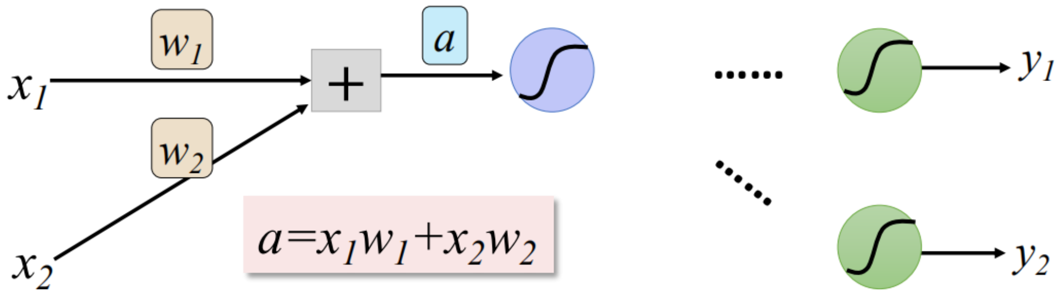
Compute $\partial a / \partial w$ for all parameters $\boldsymbol{w}$

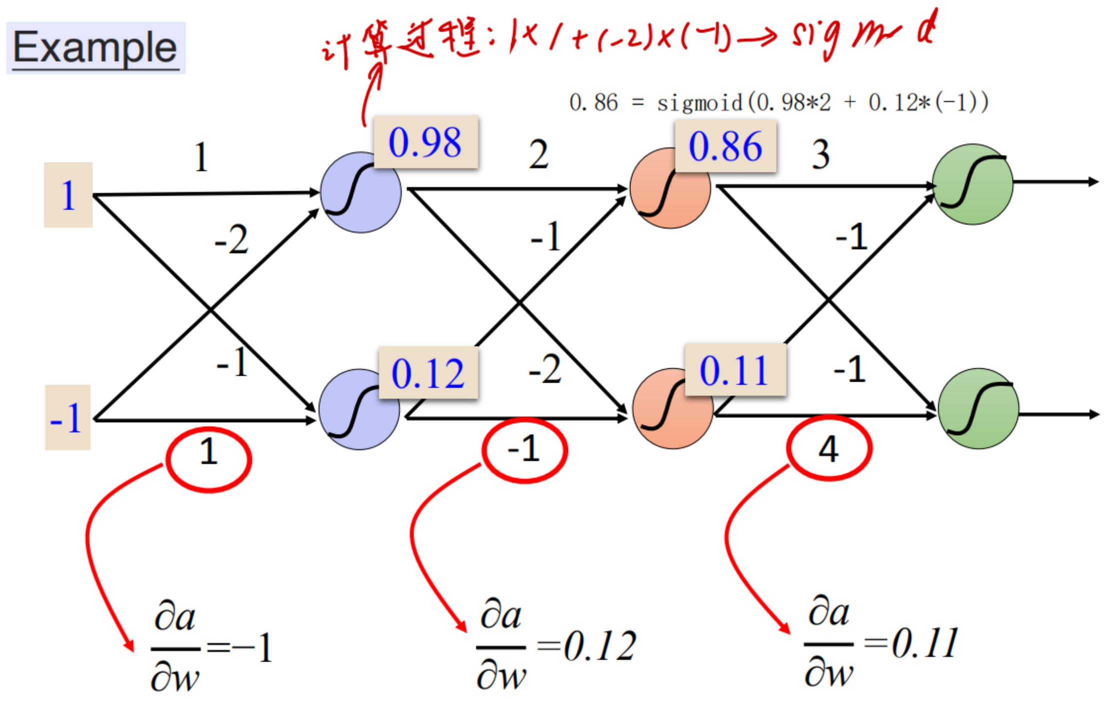
Backward pass:

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

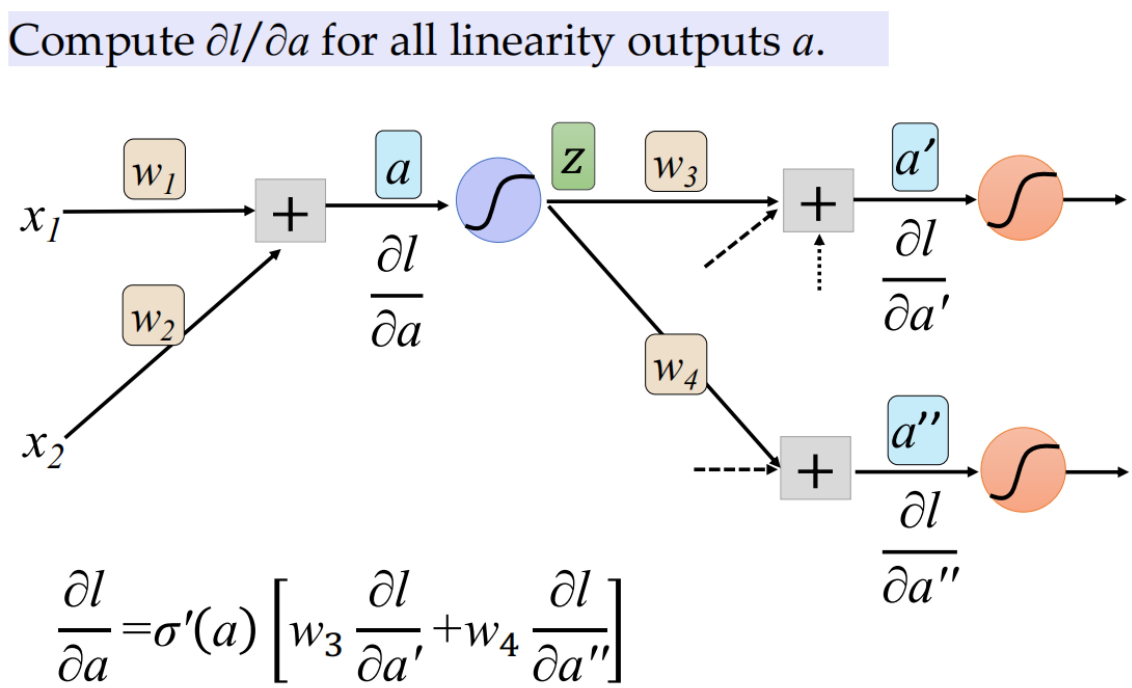
- **Forward pass(P35-36)**

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





- **Backward Pass（P37-46）**

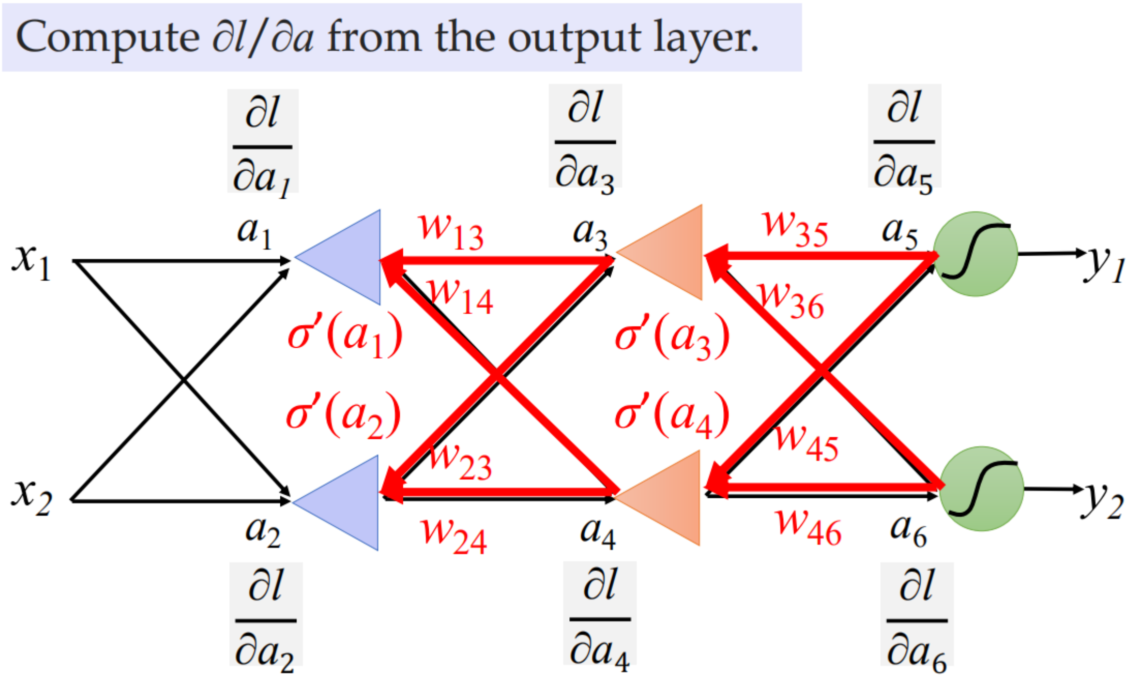


- 两种情况：
    - output layer:直接结束

Case 1. Output Layer → 到输出层

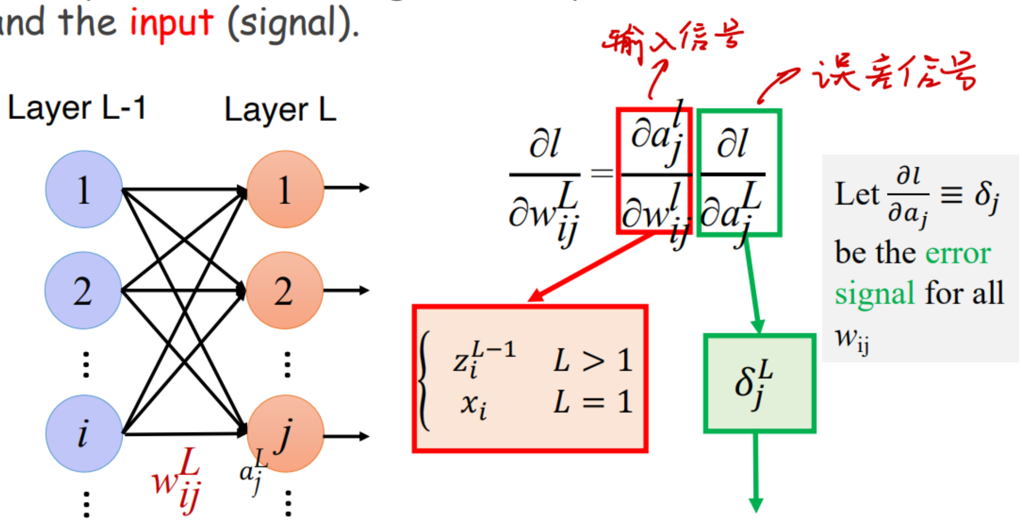
$$\frac{\partial l}{\partial a'} = \frac{\partial y_1}{\partial a'}\frac{\partial l}{\partial y_1} \qquad \frac{\partial l}{\partial a''} = \frac{\partial y_2}{\partial a''}\frac{\partial l}{\partial y_2}$$

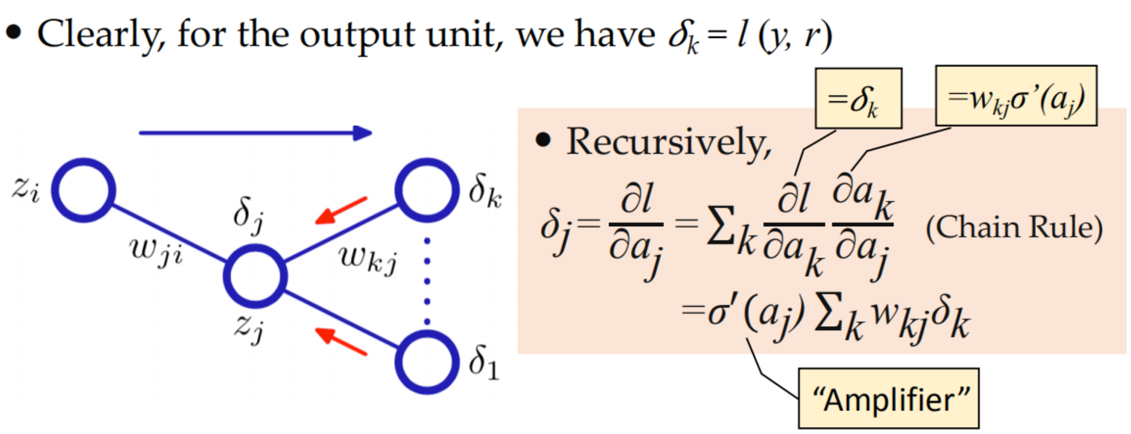
- **Case 2. Not Output Layer：Dynamic Programming**

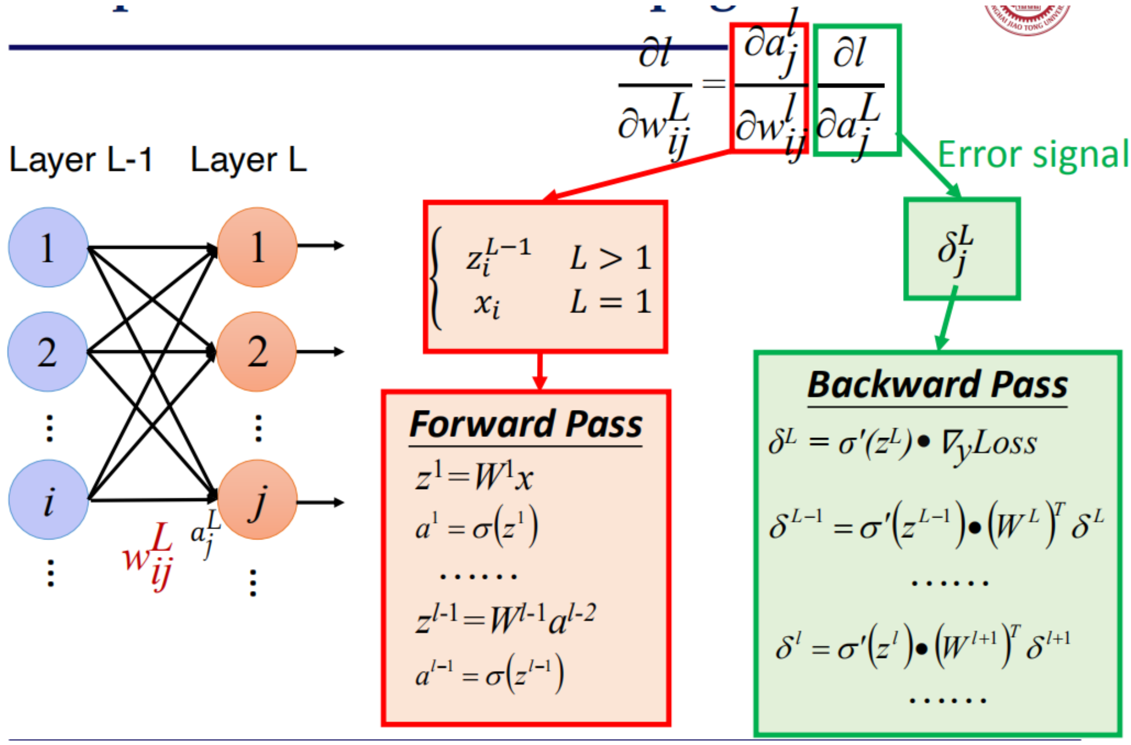


Perspective from Error-Propagation(P52-56)

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







Algorithm(P57)

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.

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

η :学习率
δ： 后续神经元的误差
σ： 放大信号
z_i:第i个神经元的输出

BP Example(P58-77)