CS 60050
Machine Learning

Neural Networks

Gradient Descent – as we studied it

- GD minimizes

$$E_{\text{in}}(\mathbf{w}) = \frac{1}{N}\sum_{n=1}^{N} \underbrace{\mathrm{e}\left(h(\mathbf{x}_n), y_n\right)}$$

- Δparameter = - learning rate * gradient
- Gradient computed based on all training examples (x_n, y_n): "Batch" GD
- Epoch: using all training examples once

Stochastic Gradient Descent (SGD)

- Pick one (x_n, y_n) at a time, apply GD to $e(h(x_n), y_n)$
- When done over many training examples, many times, average direction of descent will be the same as the "ideal" direction
- Benefits
  - Cheaper computation
  - Randomization helps escape trivial local minima
  - But cannot guarantee reaching global minima in case of non-convex error functions

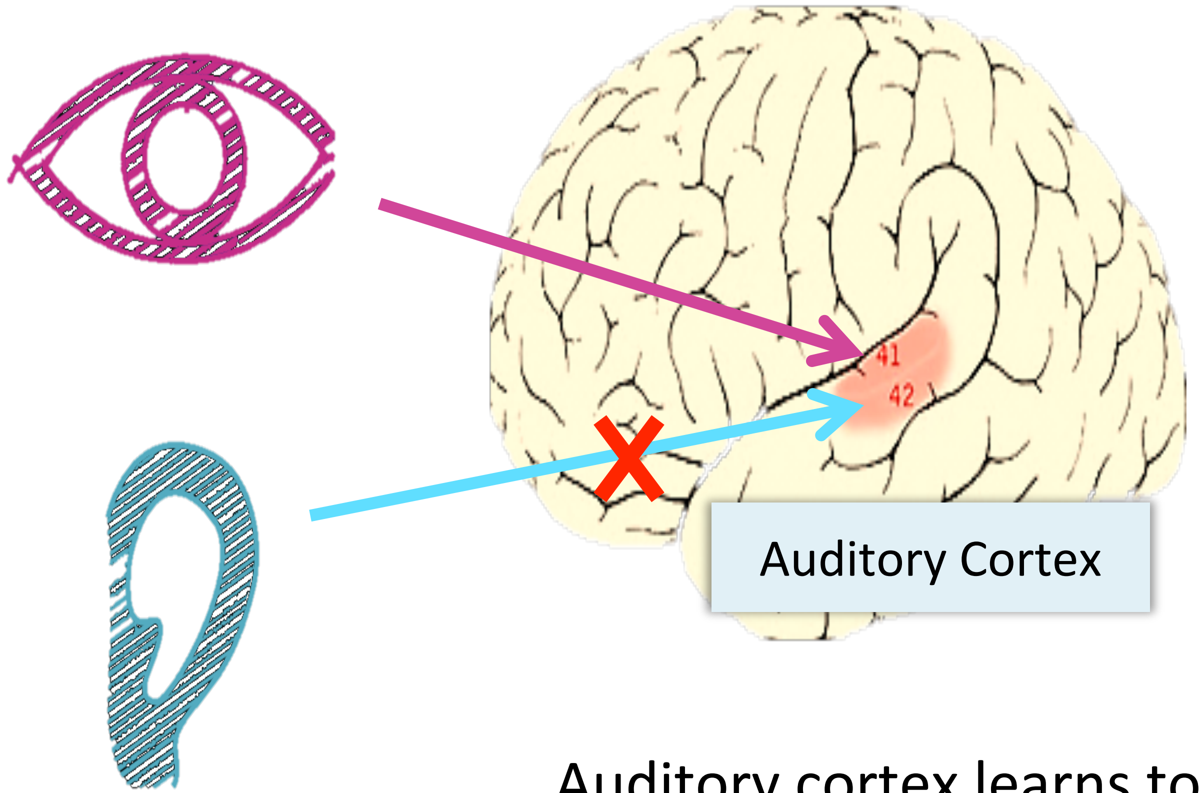
Limitations of linear models

- Linear models not sufficient for regression / classification of complex functions
- Non-linear combinations can be used, but not feasible as the number of features increases beyond few hundred (e.g., pixels in an image) – which non-linear combinations to use?

Neural Networks

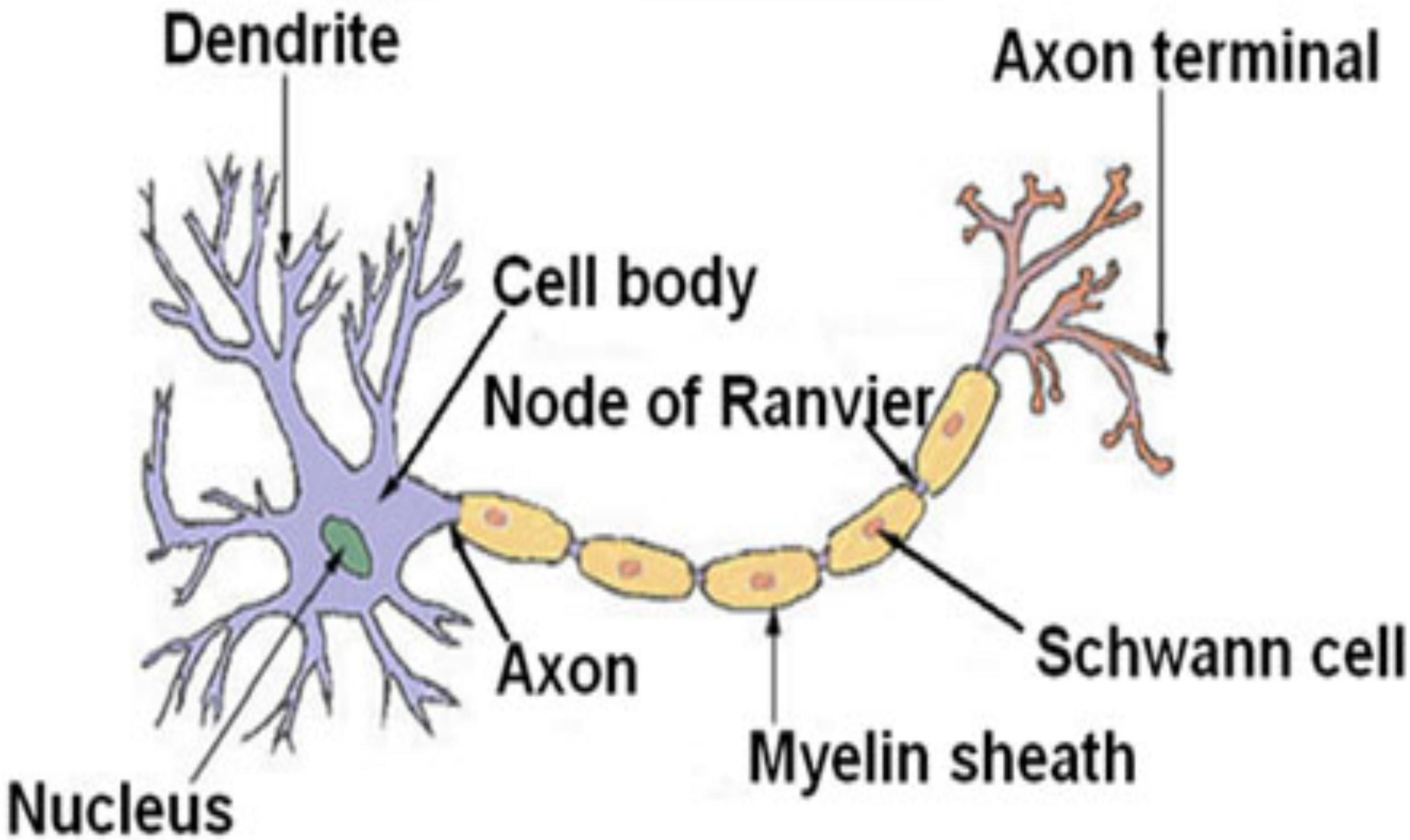
- Origins: Algorithms that try to mimic the brain.
- Was very widely used in 80s and early 90s; popularity diminished in late 90s.
- Recent resurgence: State-of-the-art technique for many applications

The “one learning algorithm” hypothesis

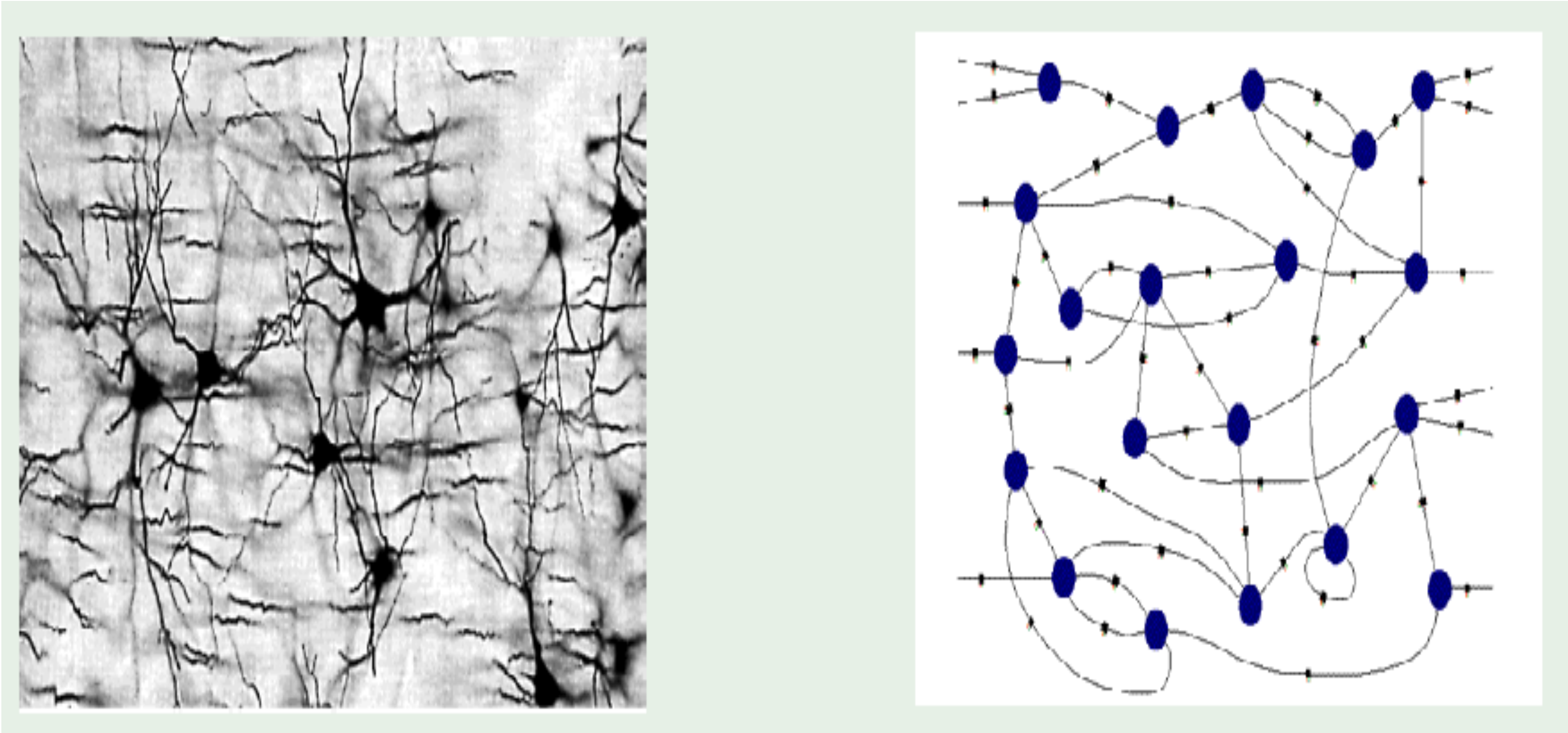


Auditory cortex learns to see

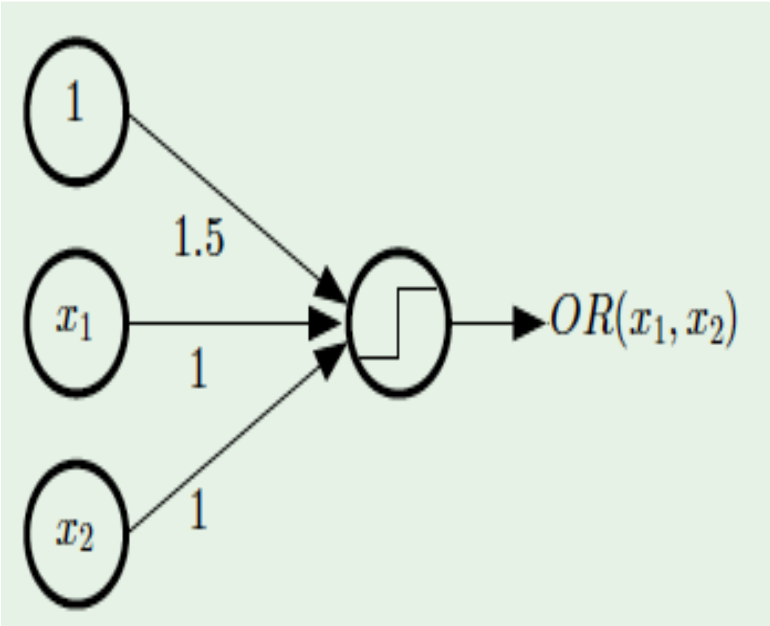
Neurons in the brain

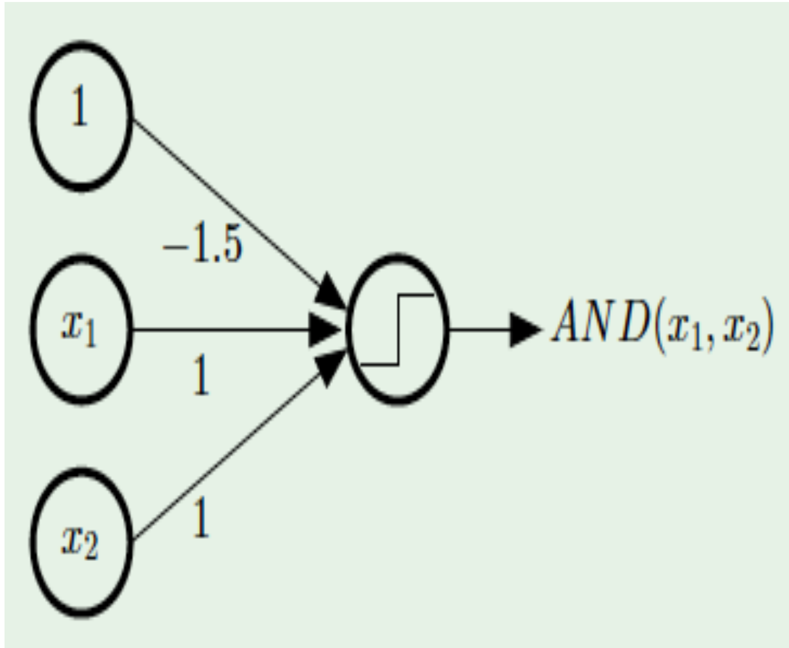


To mimic the biological function, mimic the biological structure



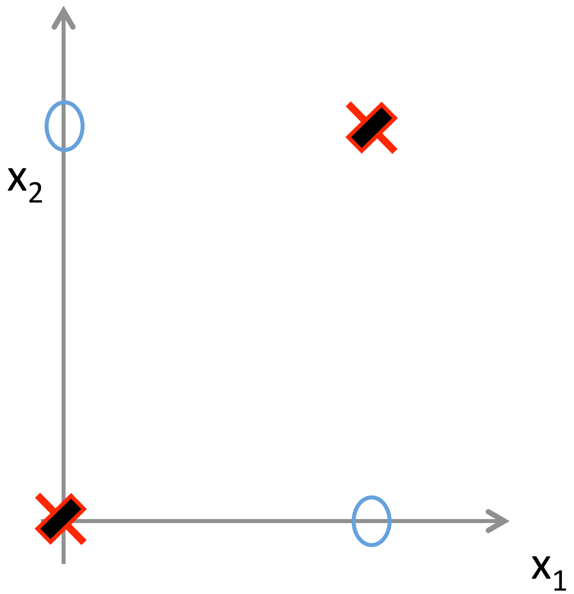
Logical unit: perceptron



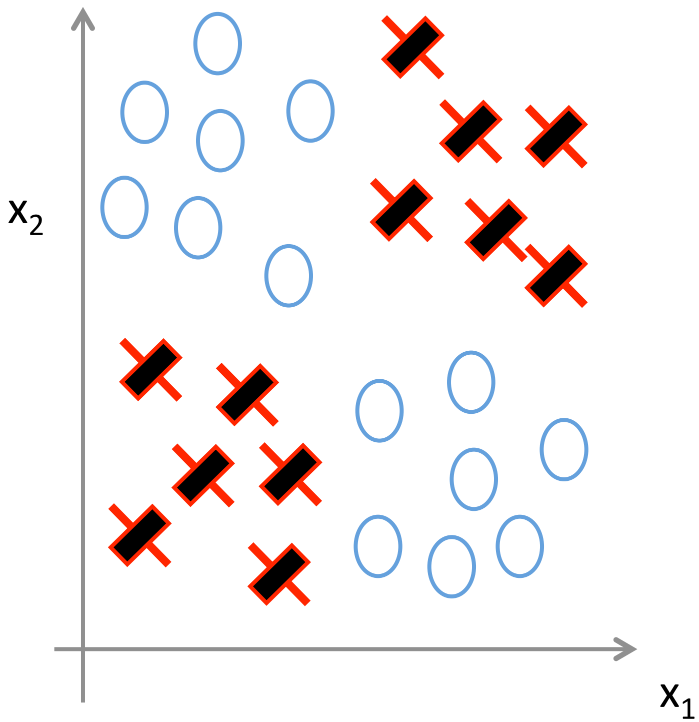


x_1, x_2 take values {-1, +1}

Non-linear classification example: XOR/XNOR

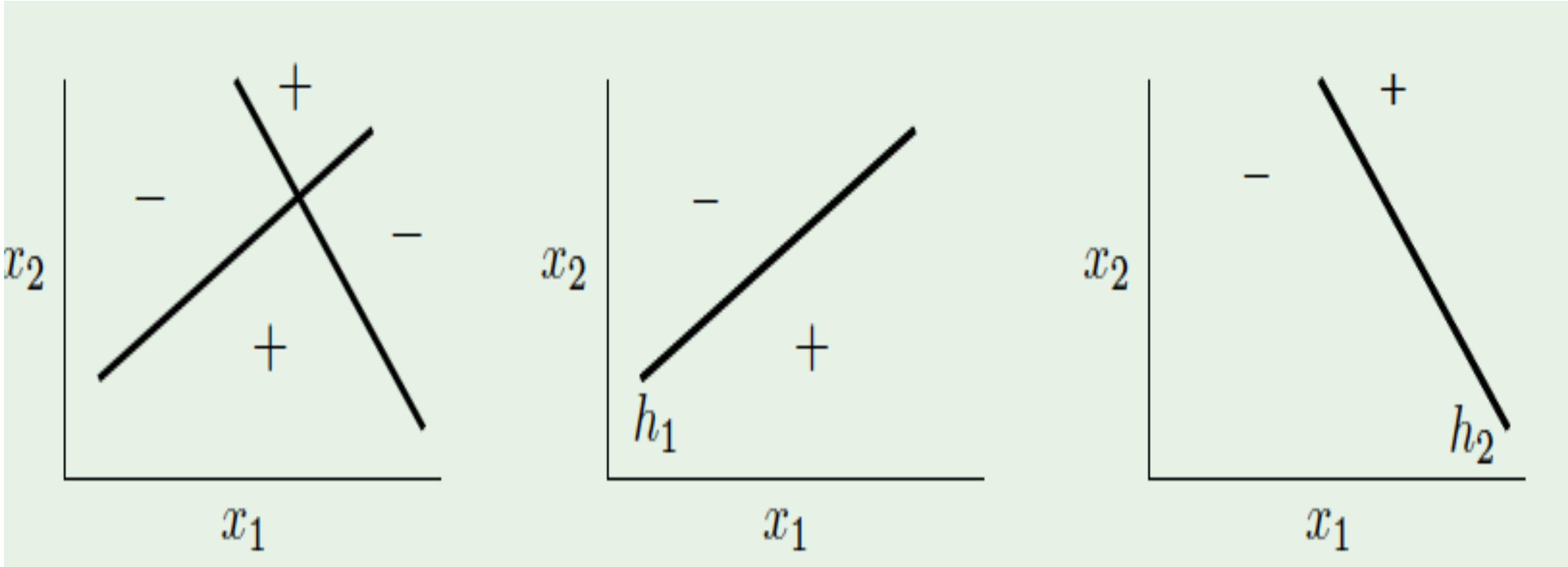


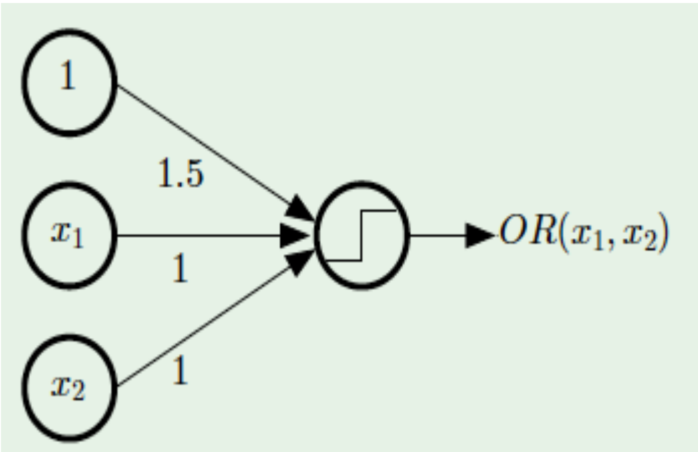
$$y = x_1 \text{ XOR } x_2$$
$$x_1 \text{ XNOR } x_2$$
$$\text{NOT } (x_1 \text{ XOR } x_2)$$

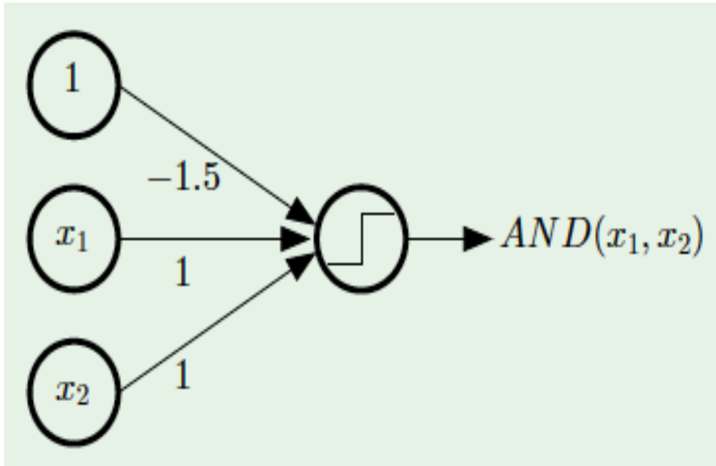


Cannot be separated using a perceptron or any linear classifier model

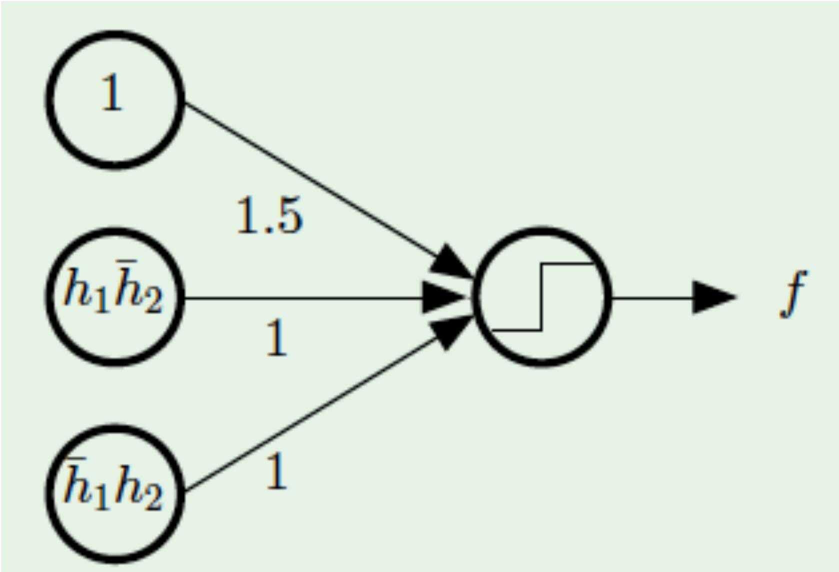
Combining multiple perceptrons



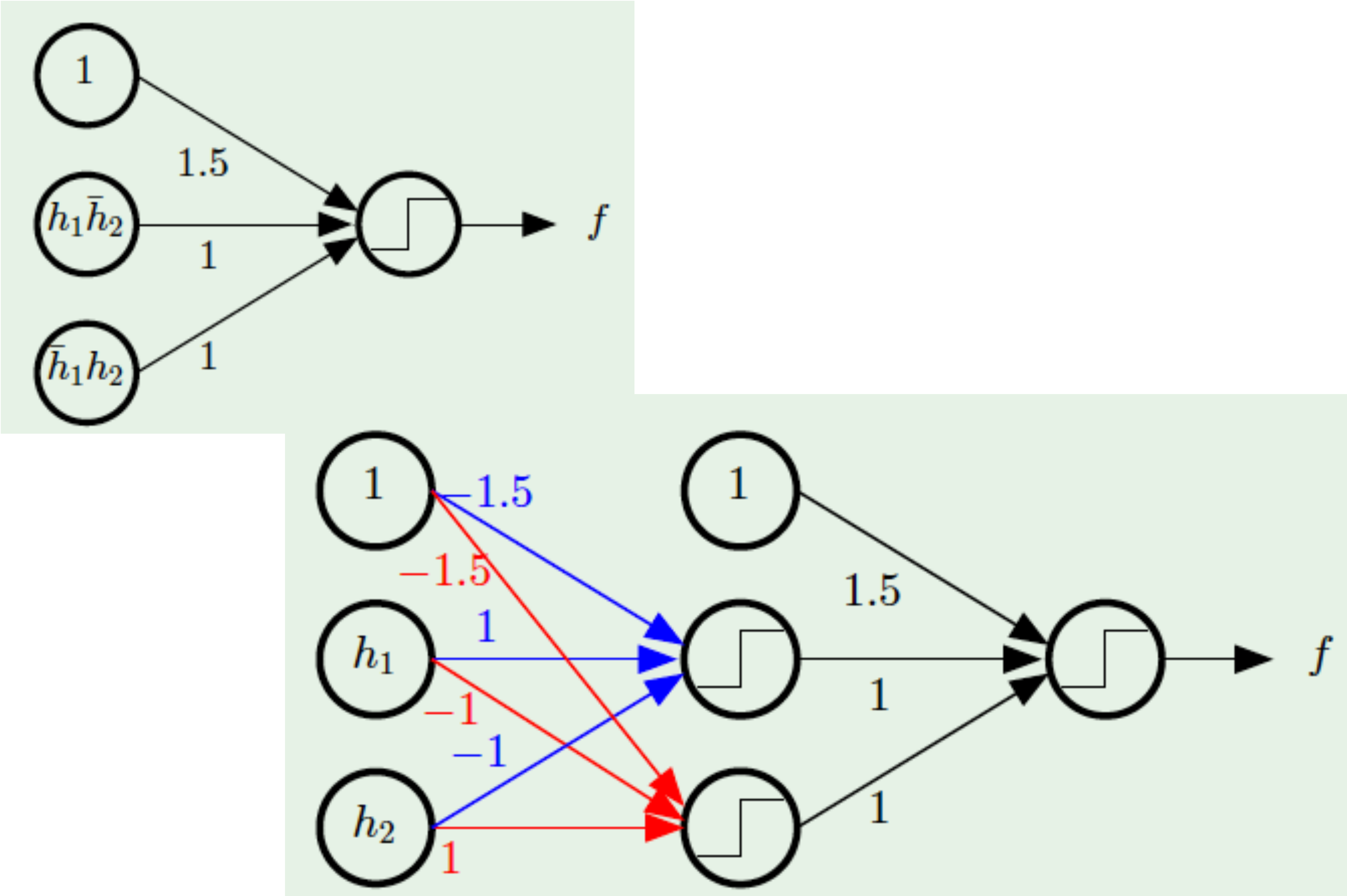




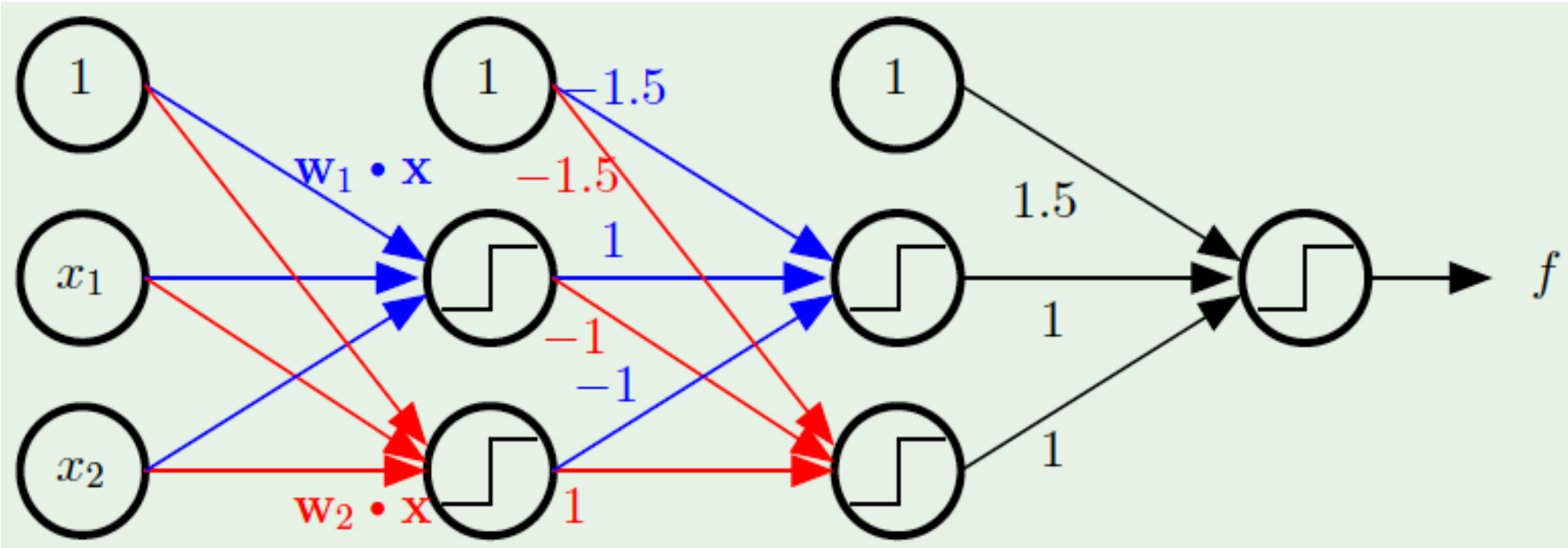
Creating layers



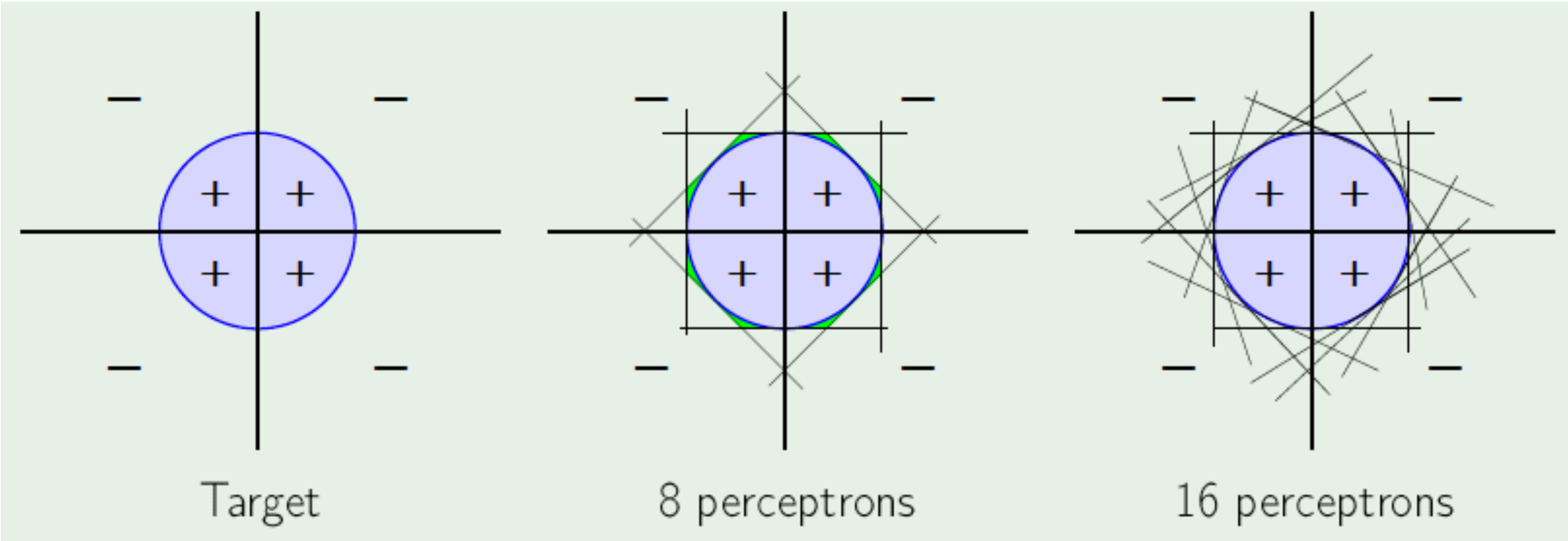
Creating layers



The multi-layer perceptron



A powerful model – can generate complex decision boundaries



From perceptron to a neuron implementing a non-linear function

- Desirable: a smooth function that is efficient to differentiate
- Possible functions
  - Range [0, 1]: logistic function
  - Range [-1, 1]: tanh function

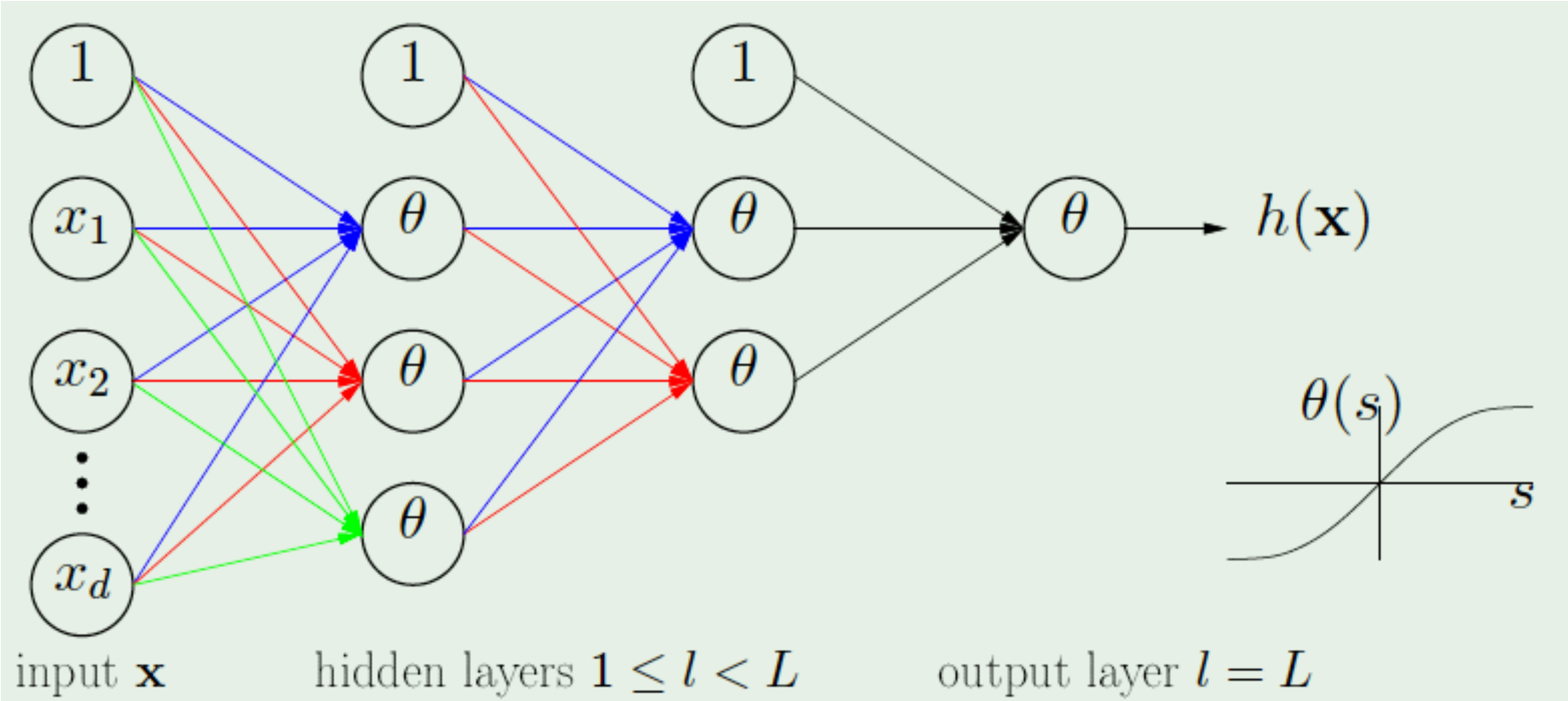
Logistic function

$$\Theta(z) = \frac{1}{1 + e^{-z}}$$

tanh function

$$\Theta(s) = \frac{e^{s} - e^{-s}}{e^{s} + e^{-s}}$$

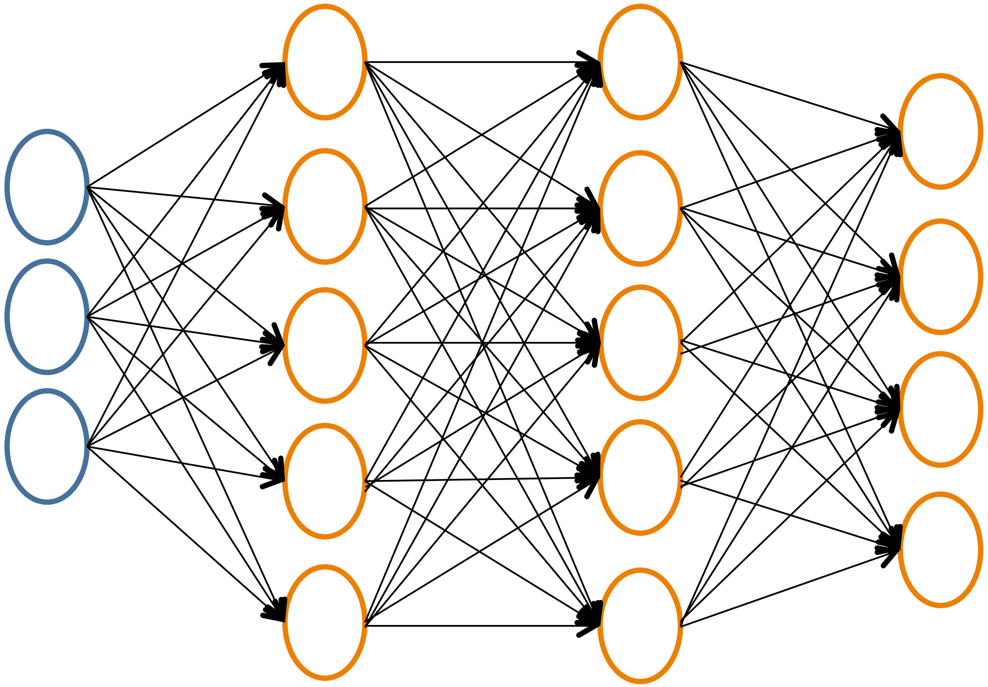
A neural network



Number of layers: L

Number of neurons in layer l: $d^{(l)}$

Different architectures possible for neural network. Example for a four-class classifier



For our discussion, we will consider a simple regression model with only one neuron in the output layer.

How the network operates

$$w_{ij}^{(l)} \quad \begin{cases} 1 \le l \le L & \text{layers} \\ 0 \le i \le d^{(l-1)} & \text{inputs} \\ 1 \le j \le d^{(l)} & \text{outputs} \end{cases}$$

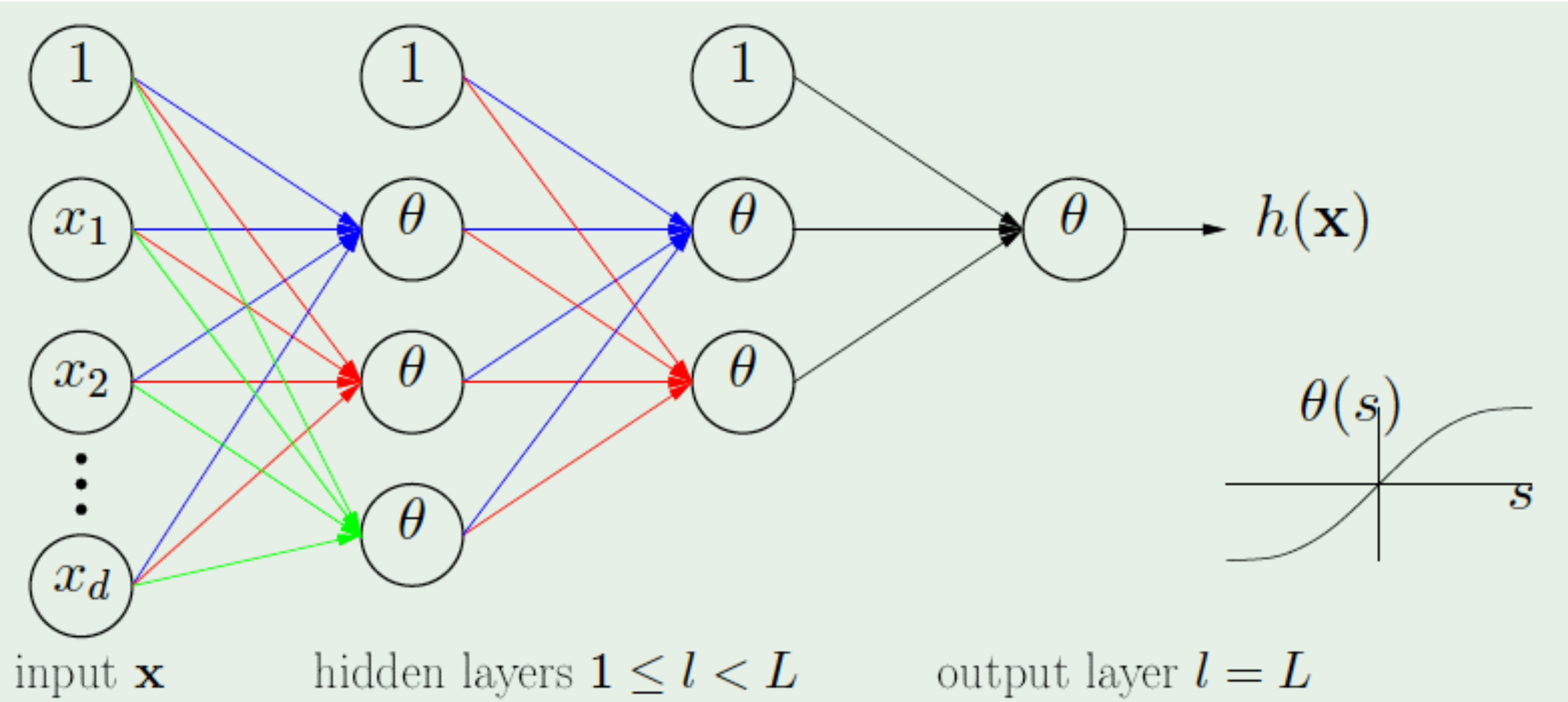
Weight of the link from i-th neuron in layer (l-1) to the j-th neuron in layer l

$$x_j^{(l)} = \theta(s_j^{(l)}) = \theta\left(\sum_{i=0}^{d^{(l-1)}} w_{ij}^{(l)}\, x_i^{(l-1)}\right)$$

Output of the j-th neuron in layer l

How the network operates

$$\text{Apply } \mathbf{x} \text{ to } x_1^{(0)} \cdots x_{d^{(0)}}^{(0)} \rightarrow \rightarrow x_1^{(L)} = h(\mathbf{x})$$



How to get the weights?

- As ML practitioners, our job is to automatically learn the weights from training data

- Learning the weights efficiently: Backpropagation algorithm

Applying SGD

- All the weights $w = \{ w_{ij}^{(l)} \}$ determine the hypothesis h
- Error on example (x_n, y_n) is $e(h(x_n), y_n) = e(w)$
- To implement SGD, we need the gradient

$$\nabla \mathrm{e}(\mathbf{w}): \quad \frac{\partial\ \mathrm{e}(\mathbf{w})}{\partial\ w_{ij}^{(l)}} \quad \text{for all} \quad i, j, l$$

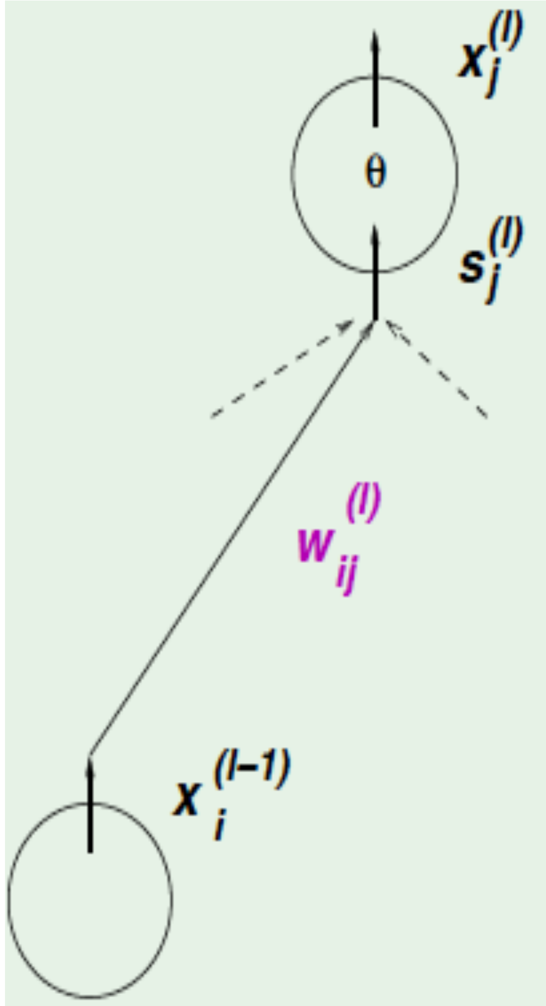
- Can compute the differentials one by one, analytically or numerically, but it will be very inefficient

Computing $\dfrac{\partial \ \mathrm{e}(\mathbf{w})}{\partial \ w_{ij}^{(l)}}$

A trick for efficient computation:

$$\frac{\partial \, \mathrm{e}(\mathbf{w})}{\partial \, w_{ij}^{(l)}} = \frac{\partial \, \mathrm{e}(\mathbf{w})}{\partial \, s_j^{(l)}} \times \frac{\partial \, s_j^{(l)}}{\partial \, w_{ij}^{(l)}}$$

We have $\dfrac{\partial \, s_j^{(l)}}{\partial \, w_{ij}^{(l)}} = x_i^{(l-1)}$ We only need: $\dfrac{\partial \, \mathrm{e}(\mathbf{w})}{\partial \, s_j^{(l)}} = \delta_j^{(l)}$



δ for the final (output) layer

$$\delta_j^{(l)} = \frac{\partial\ \mathrm{e}(\mathbf{w})}{\partial\ s_j^{(l)}}$$

For the final layer $l = L$ and $j = 1$:

$$\delta_1^{(L)} = \frac{\partial\ \mathrm{e}(\mathbf{w})}{\partial\ s_1^{(L)}}$$

$$\mathrm{e}(\mathbf{w}) = (\ x_1^{(L)} - y_n)^2$$

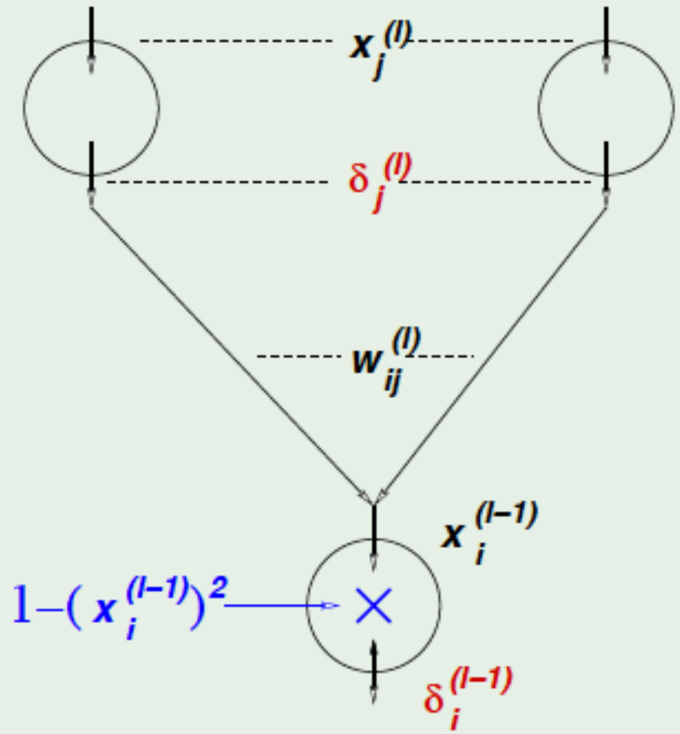
$$x_1^{(L)} = \theta(s_1^{(L)})$$

$$\theta'(s) = 1 - \theta^2(s)$$ for the tanh

Back propagation of δ

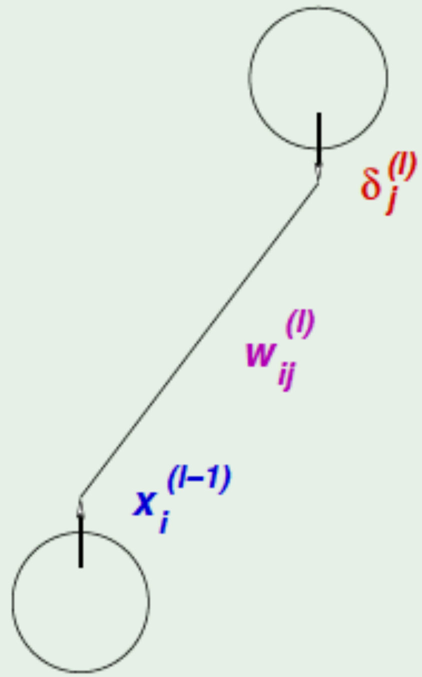
$$
\begin{aligned}
\delta_i^{(l-1)} &= \frac{\partial\ \mathrm{e}(\mathbf{w})}{\partial\ s_i^{(l-1)}} \\
&= \sum_{j=1}^{d^{(l)}} \frac{\partial\ \mathrm{e}(\mathbf{w})}{\partial\ s_j^{(l)}} \times \frac{\partial\ s_j^{(l)}}{\partial\ x_i^{(l-1)}} \times \frac{\partial\ x_i^{(l-1)}}{\partial\ s_i^{(l-1)}} \\
&= \sum_{j=1}^{d^{(l)}} \delta_j^{(l)} \times w_{ij}^{(l)} \times \theta'(s_i^{(l-1)})
\end{aligned}
$$

$$
\delta_i^{(l-1)} = (1 - (x_i^{(l-1)})^2) \sum_{j=1}^{d^{(l)}} w_{ij}^{(l)}\ \delta_j^{(l)}
$$



Back propagation algorithm

Initialize all weights $w_{ij}^{(l)}$ at random
for $t = 0, 1, 2, \ldots$ **do**
  Pick $n \in \{1, 2, \cdots, N\}$
  Forward: Compute all $x_j^{(l)}$
  Backward: Compute all $\delta_j^{(l)}$
  Update the weights: $w_{ij}^{(l)} \leftarrow w_{ij}^{(l)} - \eta \, x_i^{(l-1)} \delta_j^{(l)}$
  Iterate to the next step until it is time to stop
Return the final weights $w_{ij}^{(l)}$



Note: weights have to be initialized at random, or with some intelligent values. Zero initialization will not work.

What are the hidden layers doing?
Learning non-linear transforms