CS 60050
Machine Learning

Neural Networks

Some slides taken from course materials of Abu Mostafa, Andrew NG

This lecture

- Linear models
- Perceptron – a linear model
- Non-linear models
- Multi Layer Perceptron
- From perceptron to neuron
- Neural networks
- Learning using neural networks: the Backpropagation algorithm

LINEAR MODELS

Linear models

- The hypothesis function (used for prediction) is a linear function
- E.g., for linear regression:

$$h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_n x_n$$

Linear models: a clarification

- The hypothesis function (used for prediction) is a linear function in what?
  - Features / variables? Or
  - Coefficients of the model?

$$h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_n x_n$$

Linear in terms of both model coefficients and features/variables

$$h_\theta(x) = \theta_0 + \theta_1 (size) + \theta_2 \sqrt{(size)}$$

Linear in terms of model coefficients, but not in terms of features / variables

Both definitions are used by different ML practitioners

What we are considering

- Linear model is one that is linear in terms of the features/variables
  - A line in 2-d feature space
  - A plane in 3-d feature space
  - A hyperplane in n-d feature space

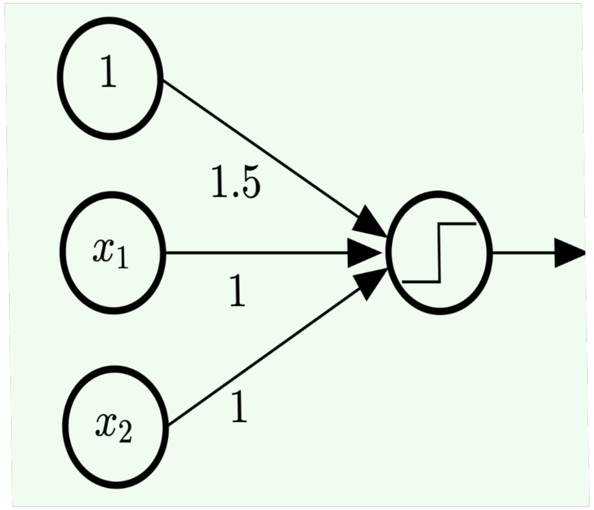
- Examples
  - Linear regression
  - A perceptron

PERCEPTRON

Perceptron

- Inputs x_1, x_2, …
- One input is a constant (called a bias)
- Each input x_i has a weight w_i
- Output: weighted sum of inputs = $\sum w_i x_i$

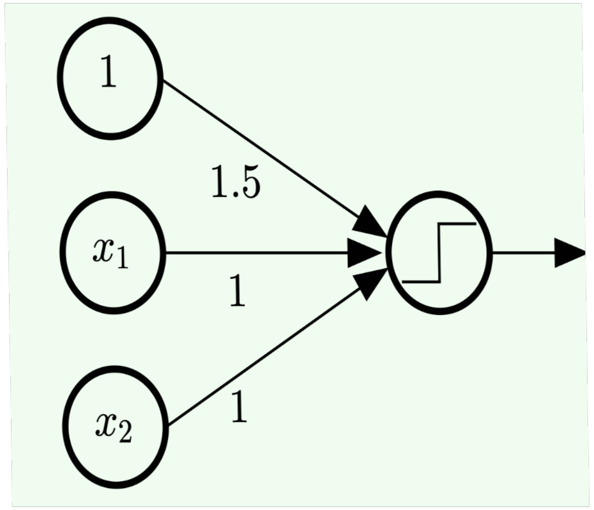
- We assume the convention:
  - For both inputs and output, -ve means logical 0, +ve means logical 1
  - Each input takes values in {-1, +1}



Perceptron – another notation

- Inputs x_1, x_2, …
- Each input x_i has a weight w_i
- The constant input multiplied by its weight is considered a single constant b that is called bias
- Output: $\sum w_i x_i + b$

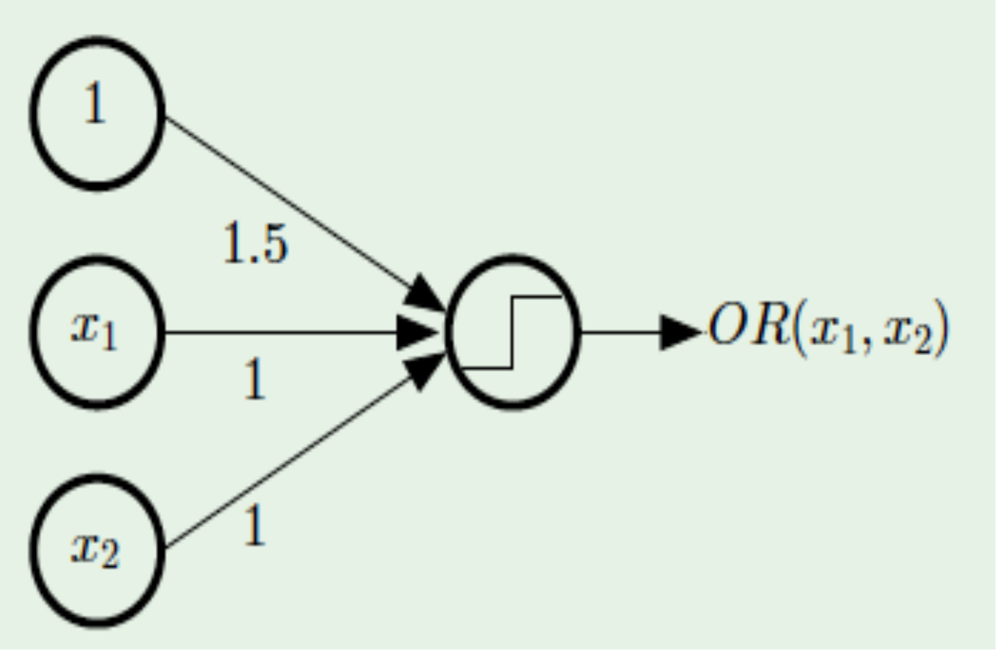
- We assume the convention:
  - For both inputs and output, -ve means logical 0, +ve means logical 1
  - Each input takes values in {-1, +1}



Using perceptron for logical operation (OR)

Inputs x_1, x_2, … each take values {-1, +1}
Output: weighted sum of inputs = $\sum w_i x_i$

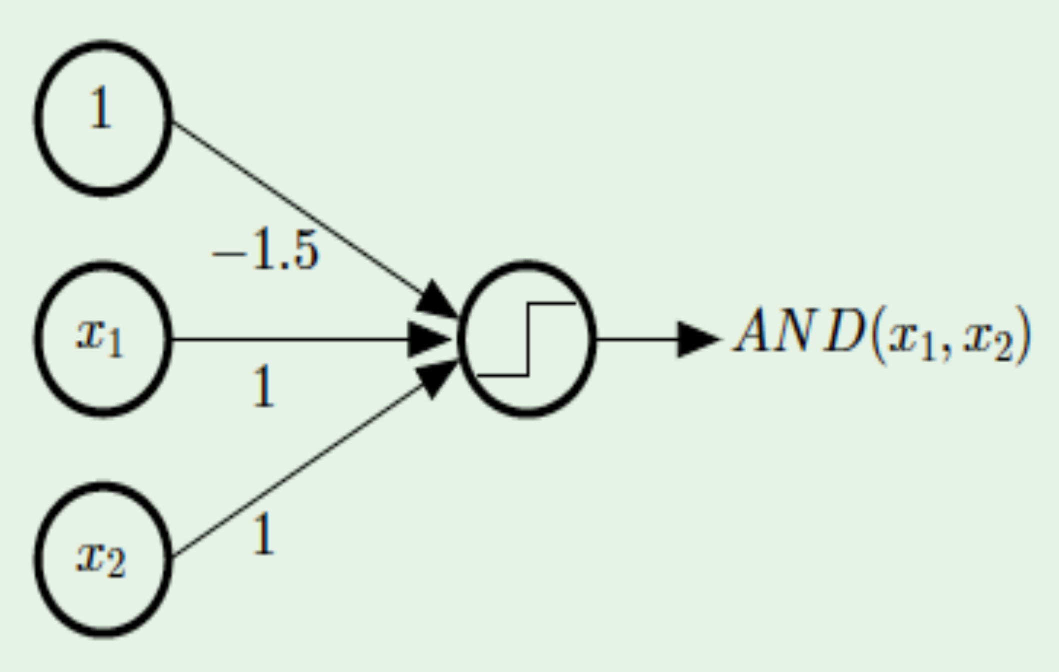
Convention for both inputs and output: negative means logical 0, positive means logical 1



Using perceptron for logical operation (AND)

Inputs x_1, x_2, … each take values {-1, +1}
Output: weighted sum of inputs = $\sum w_i x_i$

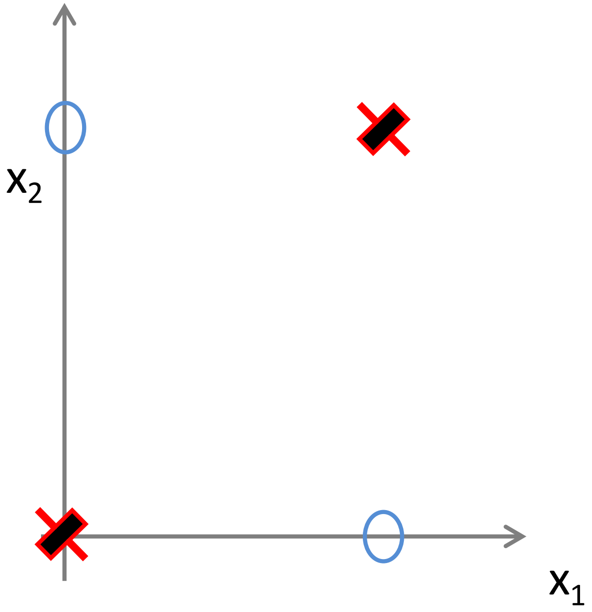
Convention for both inputs and output: negative means logical 0, positive means logical 1



Are linear models sufficient?

- Linear models not sufficient for regression / classification of complex functions
- We are not making this statement based on performance over some selected datasets
- We can theoretically show that linear models are not sufficient to model some functions

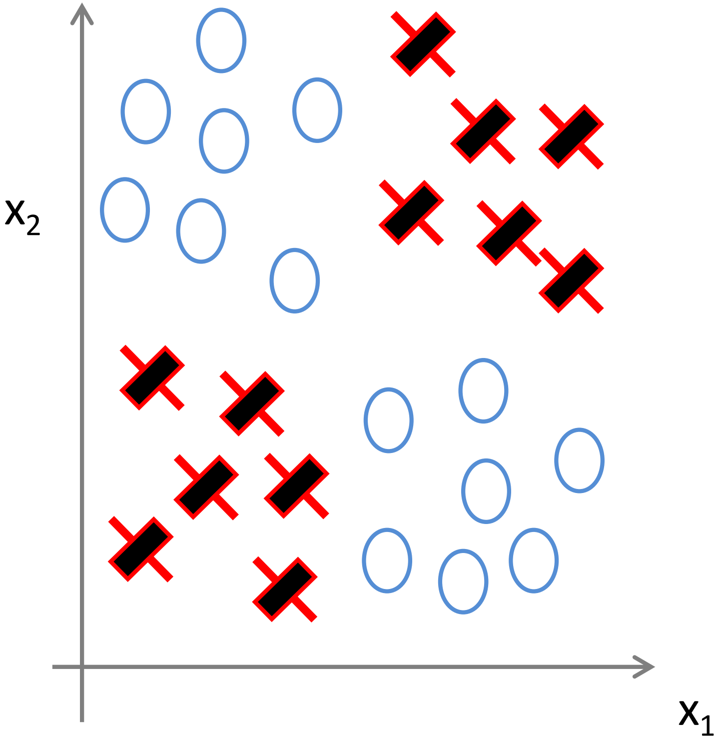
A function for which linear models are not sufficient (assuming two features)



$$y = x_1 \text{ XOR } x_2$$

$$x_1 \text{ XNOR } x_2$$

$$\text{NOT } (x_1 \text{ XOR } x_2)$$



Cannot be separated using a perceptron or any linear classifier model

How to address the limitations of linear models?

- We have seen that non-linear combinations of features can be used with linear models
- But not feasible as the number of features increases beyond few hundred (e.g., pixels in an image) – which non-linear combinations to use?
- Need for non-linear models

NON-LINEAR MODELS

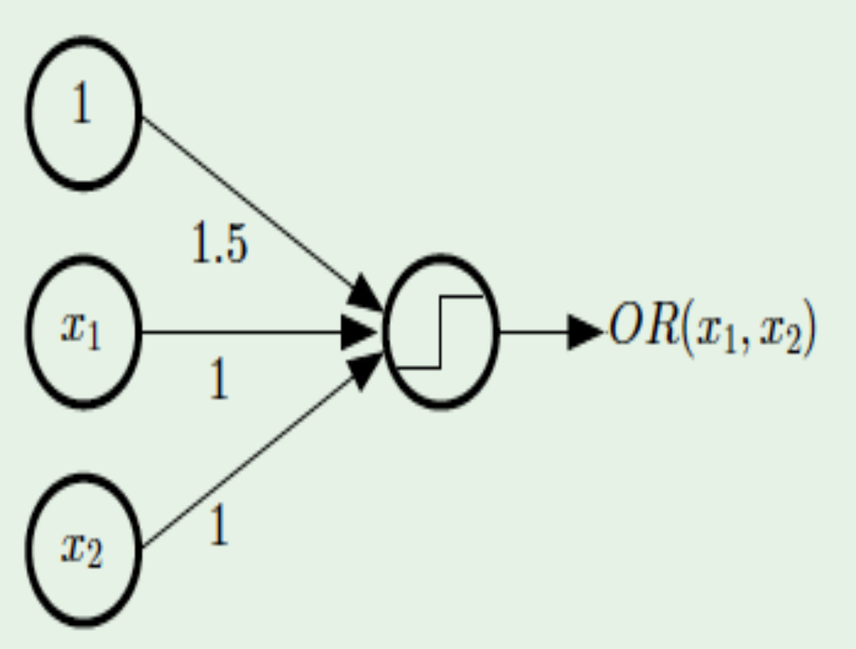
Can we model non-linear functions using multiple linear models?

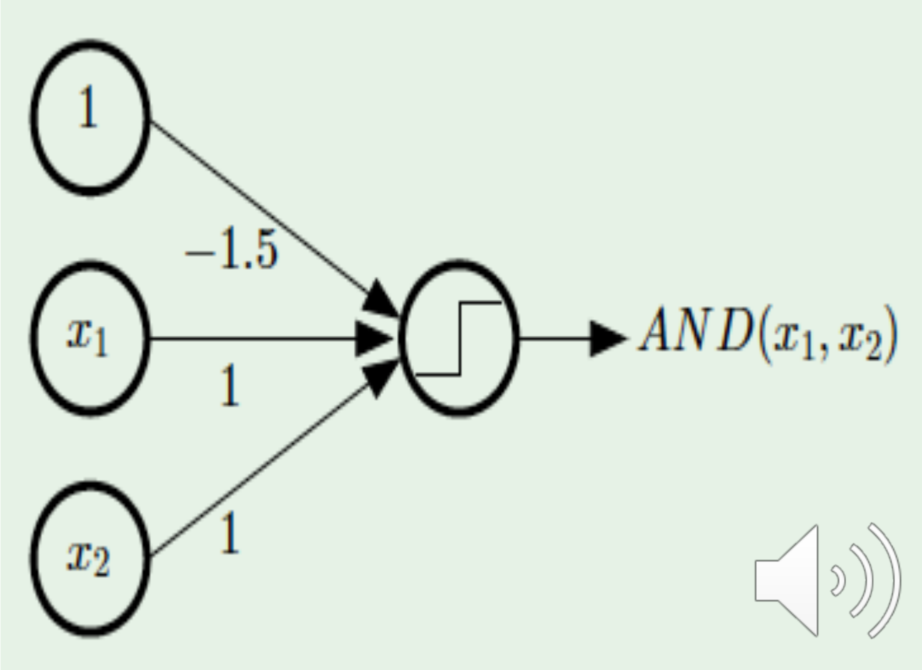
A specific version of the question:

Can we model the XOR function using multiple perceptrons?

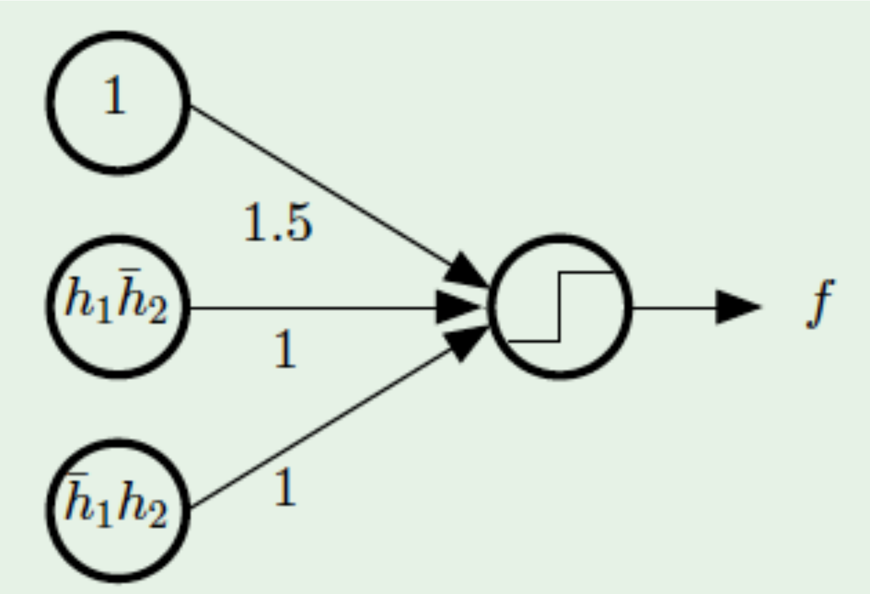
Recall from previous slides

We used a single perceptron to model the OR function and the AND function

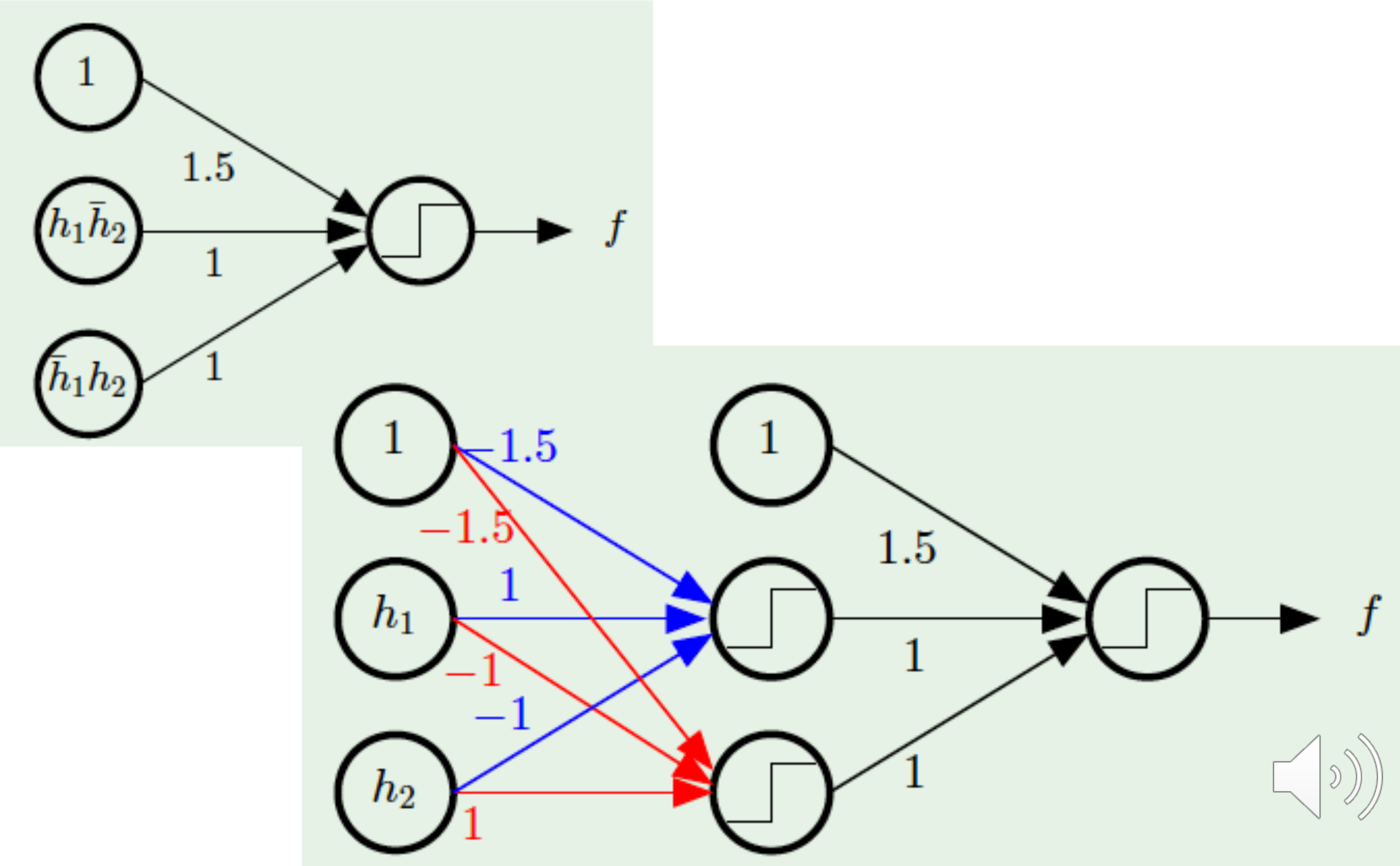




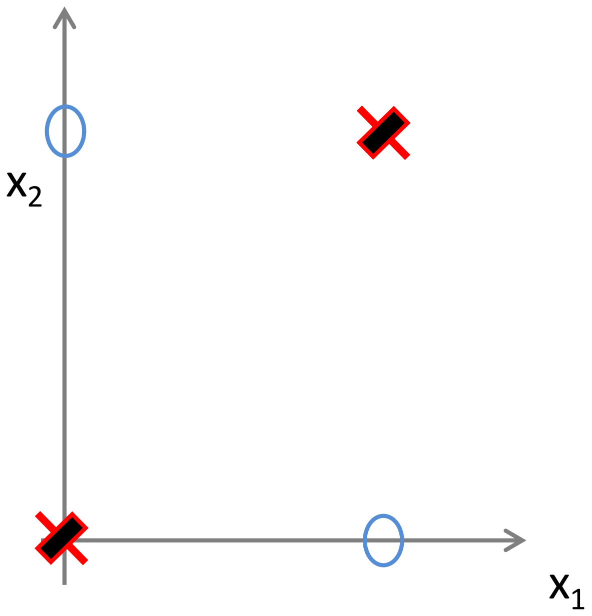
Creating layers of perceptrons to implement more complex functions (XOR)



Creating layers of perceptrons to implement more complex functions (XOR)



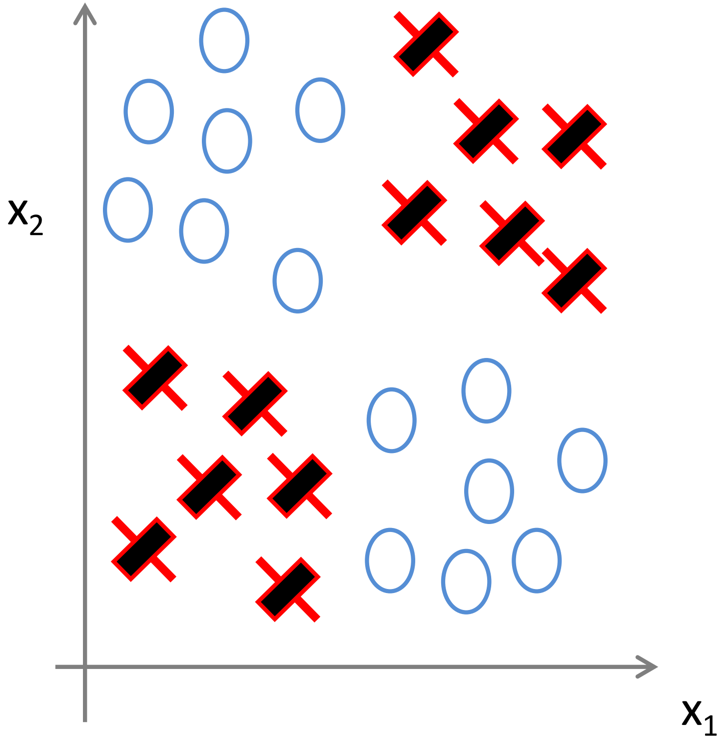
Non-linear classification using perceptrons



$$y = x_1 \text{ XOR } x_2$$

$$x_1 \text{ XNOR } x_2$$

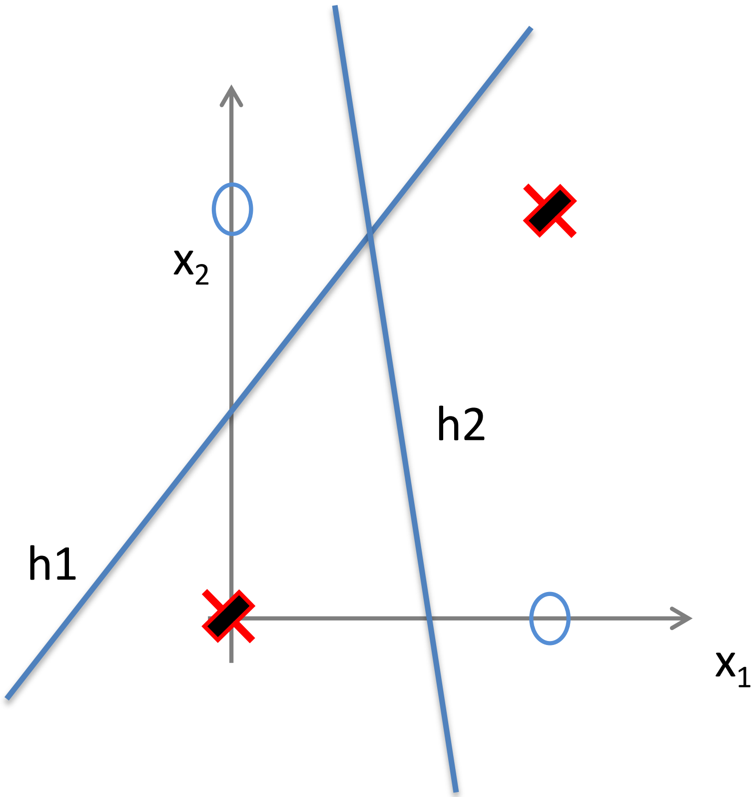
$$\text{NOT } (x_1 \text{ XOR } x_2)$$



Can be separated using a

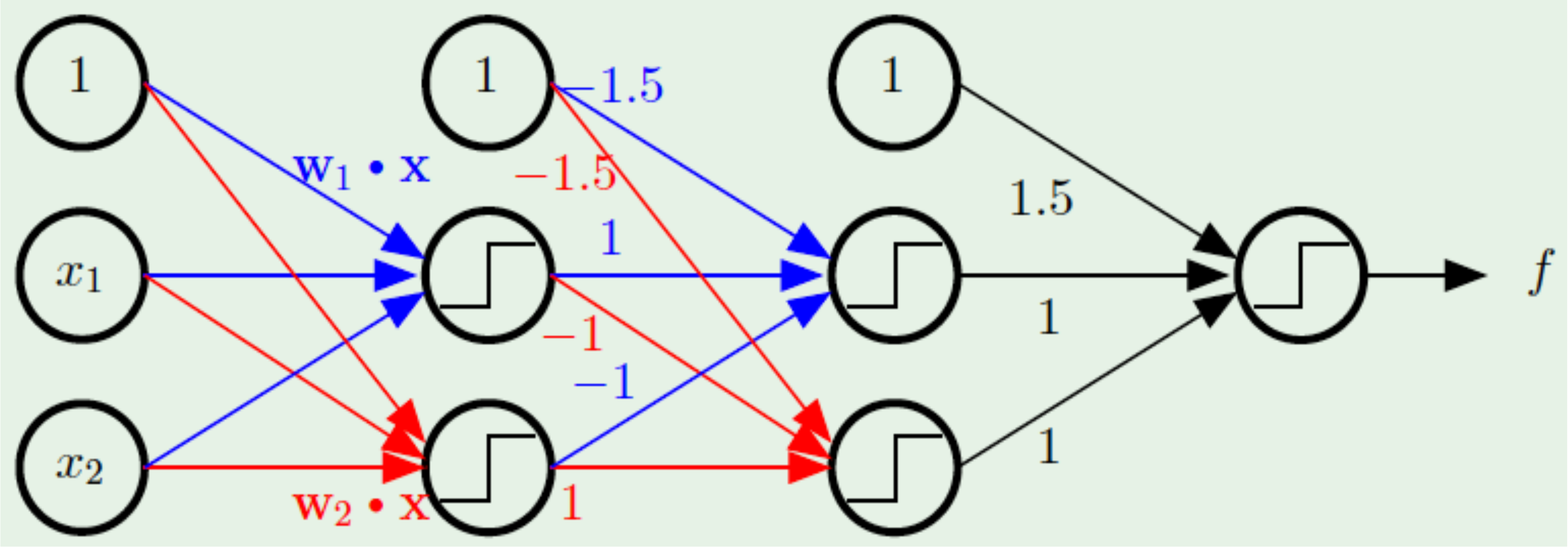
Multi Layer Perceptron (MLP)

We need to combine multiple perceptrons suitably



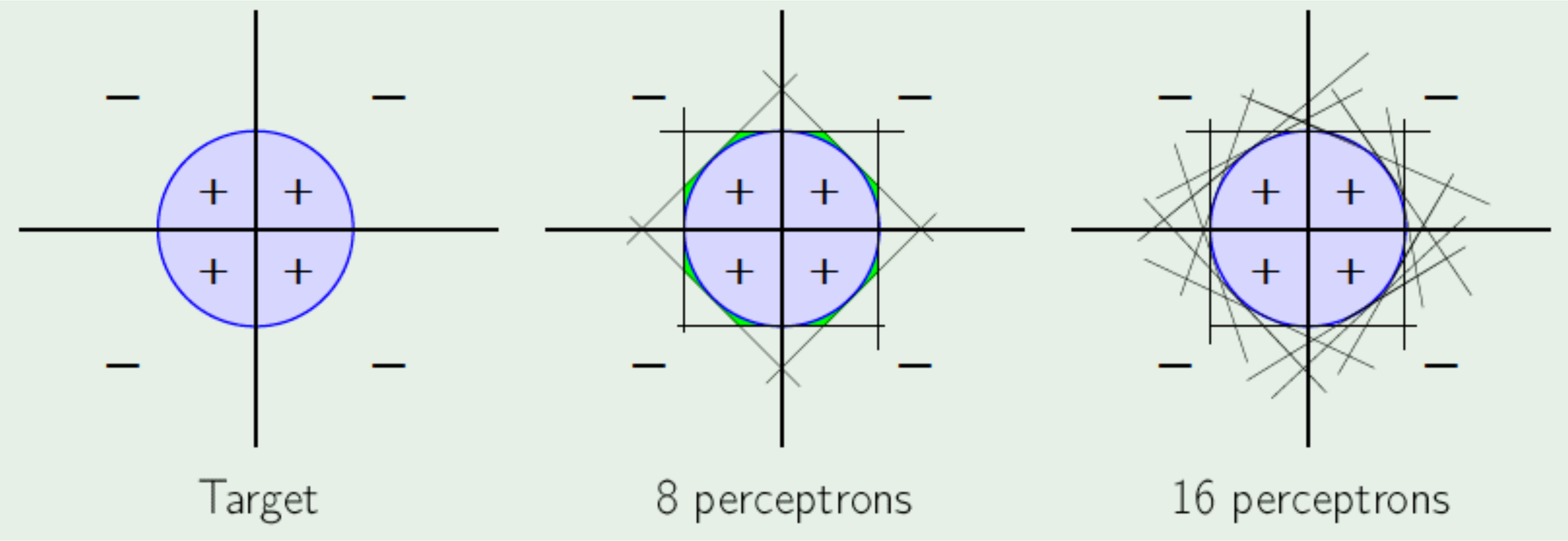
Weights or parameters of each perceptron to be tuned based on actual points

A multi-layer perceptron for general non-linear classification



Suitable values need to be fixed for the weights w_1 and w_2 (model parameters), based on the data points

A powerful model – can generate complex decision boundaries by combining many linear classifiers



Multilayer perceptrons, suitably combined, can generate almost all functions / decision boundaries

FROM PERCEPTRON TO NEURON

A problem with perceptron

- What we considered for a perceptron:
  - Output of perceptron: $\sum w_i x_i$
  - For both inputs and output, -ve means logical 0, +ve means logical 1
  - Basically, a hard threshold decides the output (logical 0 or 1)
- Optimization becomes difficult with many perceptrons
- We would like to change the input a little and see how the output changes (iterative methods)

From perceptron to neuron

- Desirable: instead of a hard threshold, a smooth function that is efficient to differentiate

- So that we can change the inputs a little, observe the corresponding small change in the output, hence compute gradient, etc.

- A perceptron with a smooth non-linear function is called a neuron

From perceptron to a neuron

- Desirable: a smooth function that is efficient to differentiate
- Possible functions
  - Logistic (sigmoid) function: range [0,1]
  - tanh function: range [-1, 1]
  - Other functions also used in Deep NNs, e.g., RelU

Logistic function

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

tanh function

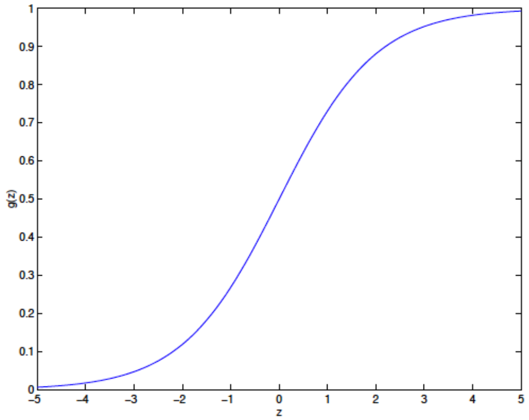
$$\Theta(s) = \frac{e^{s} - e^{-s}}{e^{s} + e^{-s}}$$

Task: verify that these functions are easy to differentiate

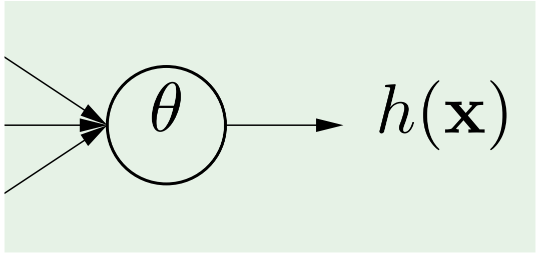
One neuron with logistic function

Logistic function

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



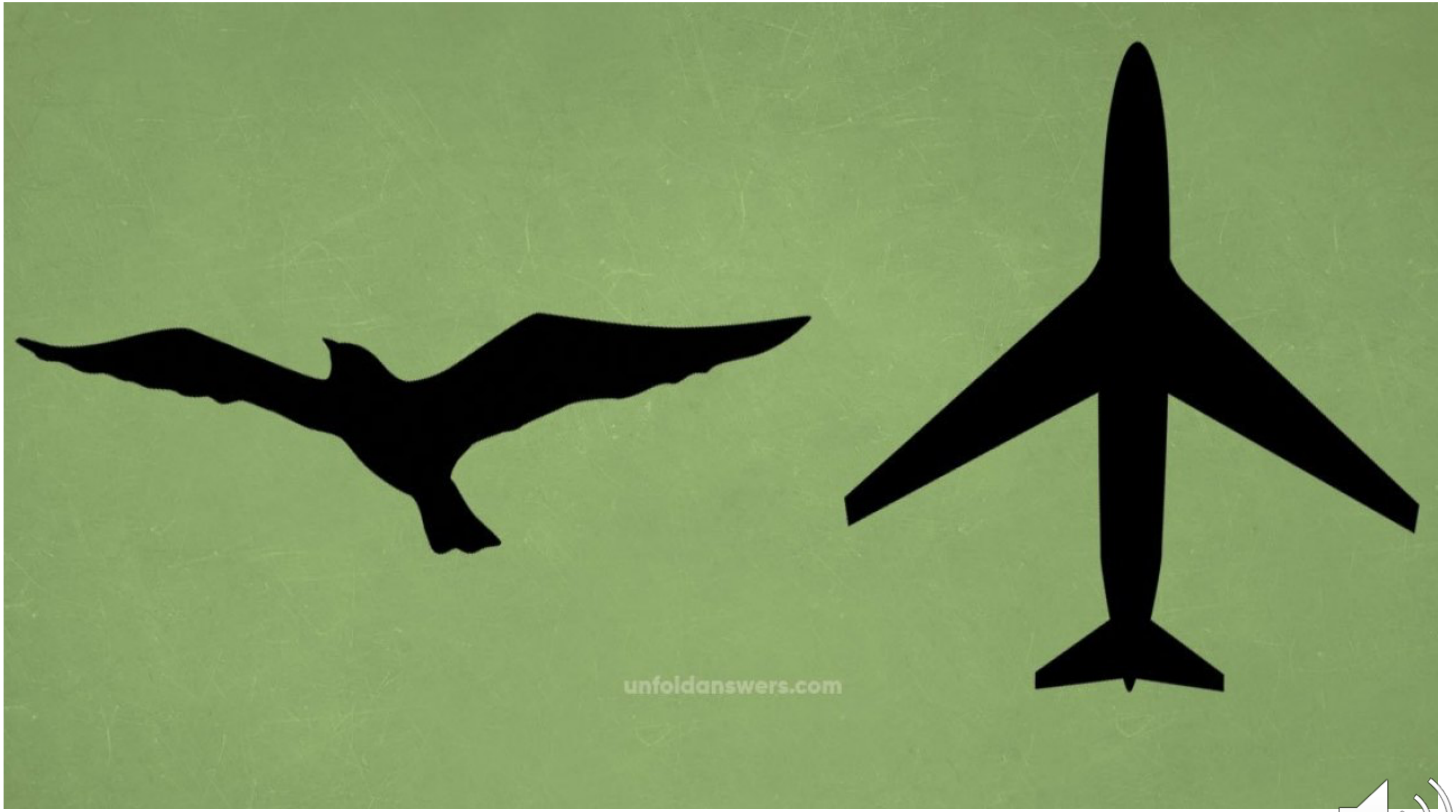
- Where $z = \sum w_i x_i$
- Essentially implementing a logistic regression classifier over the input features x_i



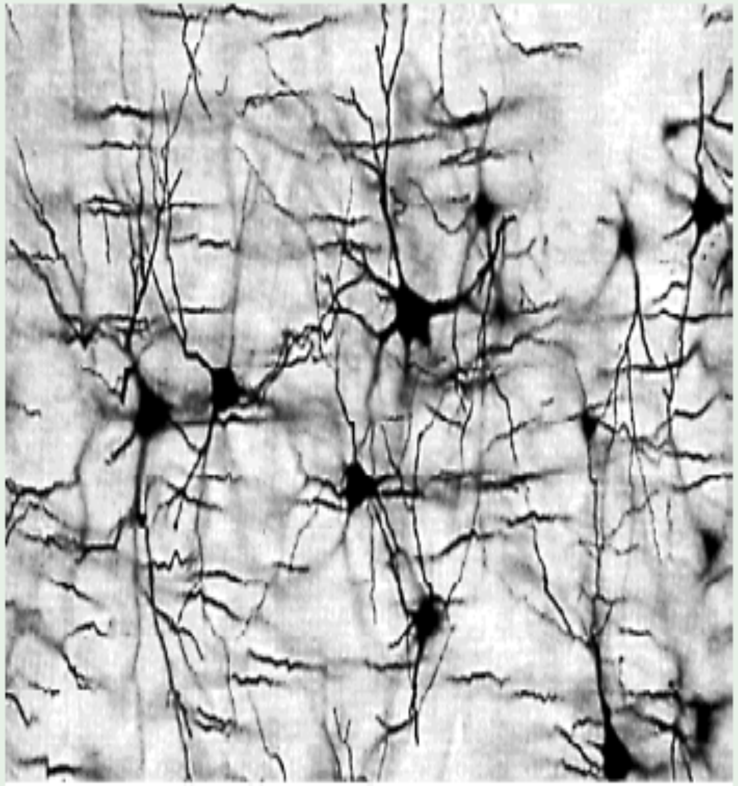
NEURAL NETWORKS

Algorithms that try to mimic the brain

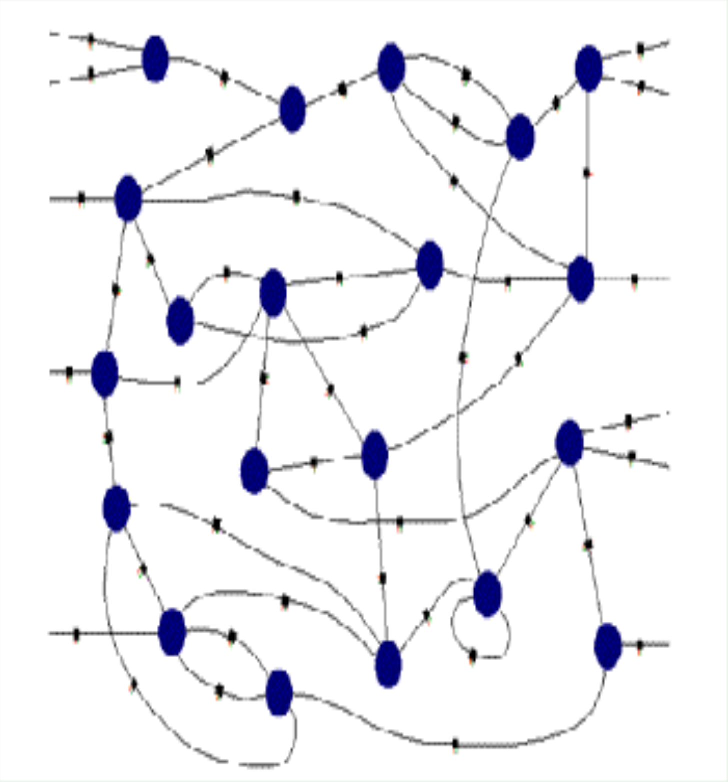
Idea: To mimic the biological function, first mimic the biological structure



Idea: To mimic the biological function, first mimic the biological structure



Brain has network of biological neurons



Network of artificial neurons arranged in layers (similar to M, but using neurons)

Brief history of neural networks

- ~1943: a highly simplified model of an artificial neuron (already discussed) proposed by McCulloh and Pitts

- ~1957: Rosenblatt coined the term “perceptron” as a very promising model for AI

SPRING OF AI

- ~1965: First generation multilayer perceptron developed by Ivakhnenko et al.

Brief history of neural networks

- 1969: In their famous book “Perceptrons”, Minsky and Papert showed that perceptrons cannot even learn some very simple functions (e.g., XOR)

WINTER OF AI

- This led to severe criticism of AI and reduced interest, till around 1986

- Interestingly, Minsky and Papert themselves said that MLPs can implement such functions; but this fact was overlooked

Brief history of neural networks

- ~1986: Backpropagation algorithm developed, that allows efficient training of a neural network
  - Will be discussed in detail

REGENERATION OF INTEREST IN AI

- 1989: The Universal Approximation Theorem: A multilayer network of neurons with a single hidden layer can be used to approximate any continuous function to any desired precision

Brief history of neural networks

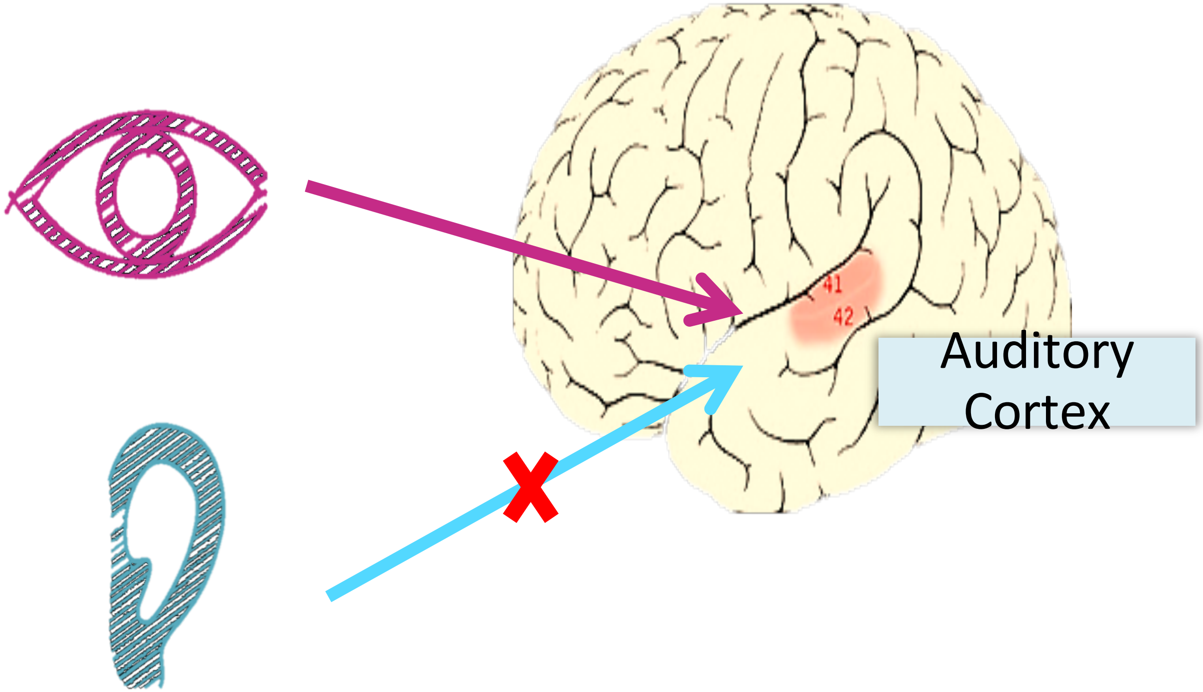
- In spite of algorithmic advances (Backpropagation) still there were practical difficulties in training really deep NNs (with many layers)

- 2006: An efficient way to train Deep NNs in practice developed by Hinton et al

DEEP LEARNING ERA

- Also better hardware infrastructure (e.g., GPUs)

The “one learning algorithm” hypothesis

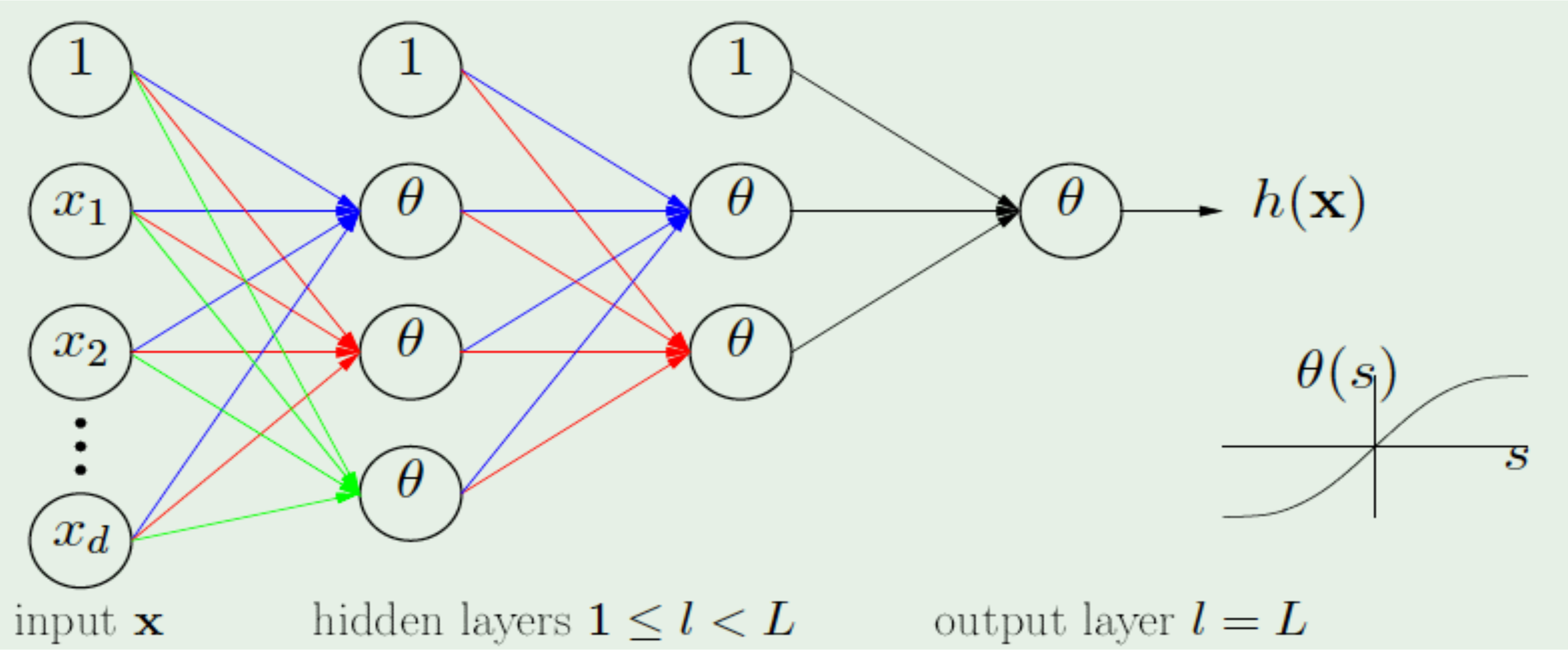


Visual projections routed to the auditory pathway in ferrets: receptive fields of visual neurons in primary auditory cortex, Roe et al, 1992

Auditory cortex, when connected to the eyes, learns to see

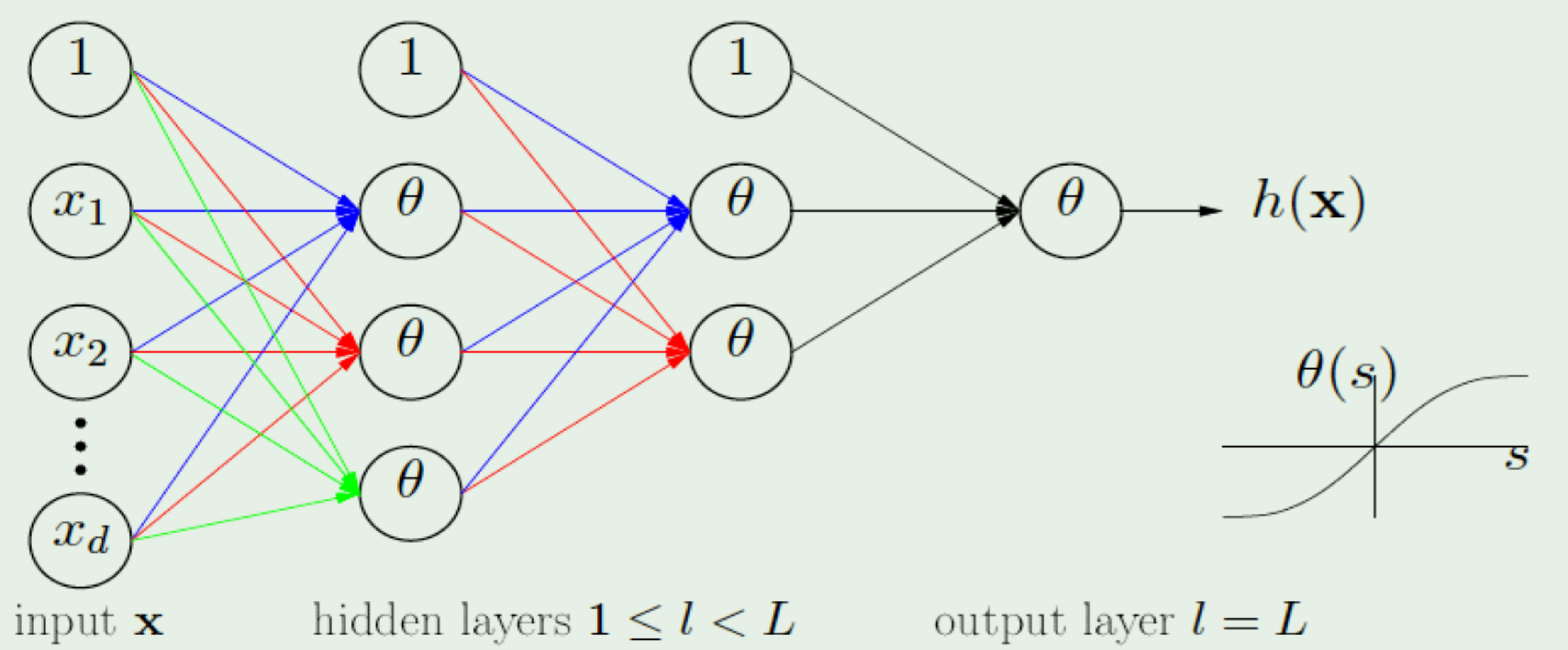
=> A neural network can learn various functionalities

A neural network



- Multi-layer network
- Each unit is a neuron, implementing a non-linear function (e.g., sigmoid) over the weighted inputs
- Input layer, hidden layer(s), output layer

A neural network



- Number of layers: L
- Number of neurons in layer l: $d^{(l)}$
- Number of neurons in input layer = $d^{(0)}$ = number of features in input

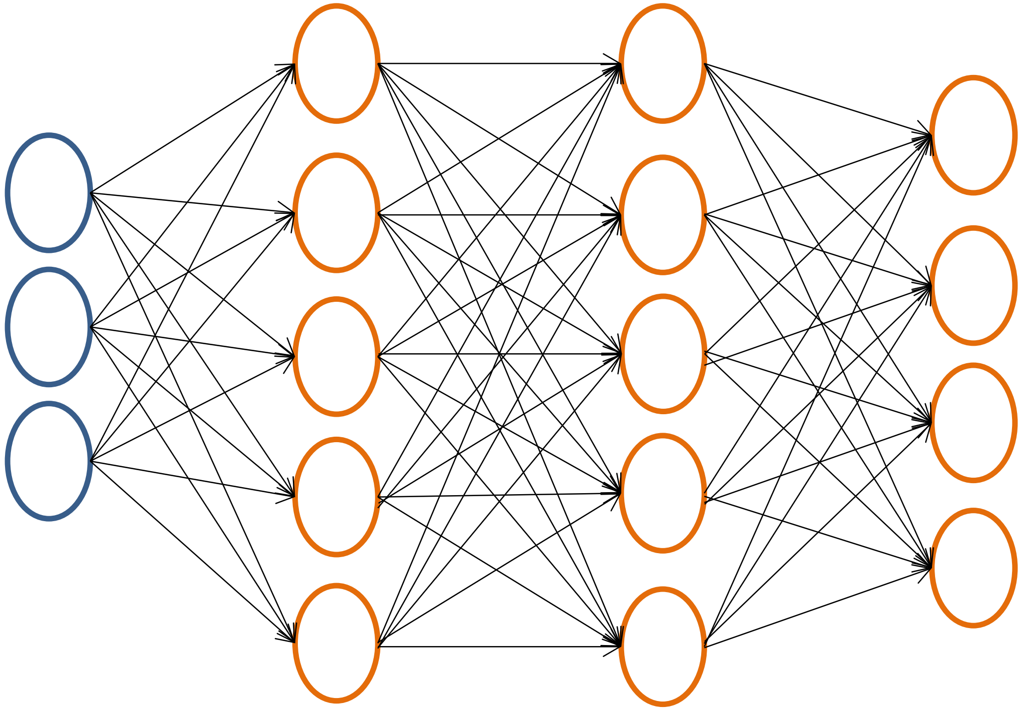
Our simplified situation

- The neural network architecture we are considering is called a “fully connected” (FC) architecture
  - Many other architectures are possible

- We consider all neurons to implement the same non-linear function
  - Non-linearity in different neurons can be different

- We consider a simple regression model with only one neuron in the output layer
  - Multiple neurons in output layer are possible

Example neural network for a four-class classifier



Each output neuron conceptually outputs the probability of the data point being in each of the 4 classes

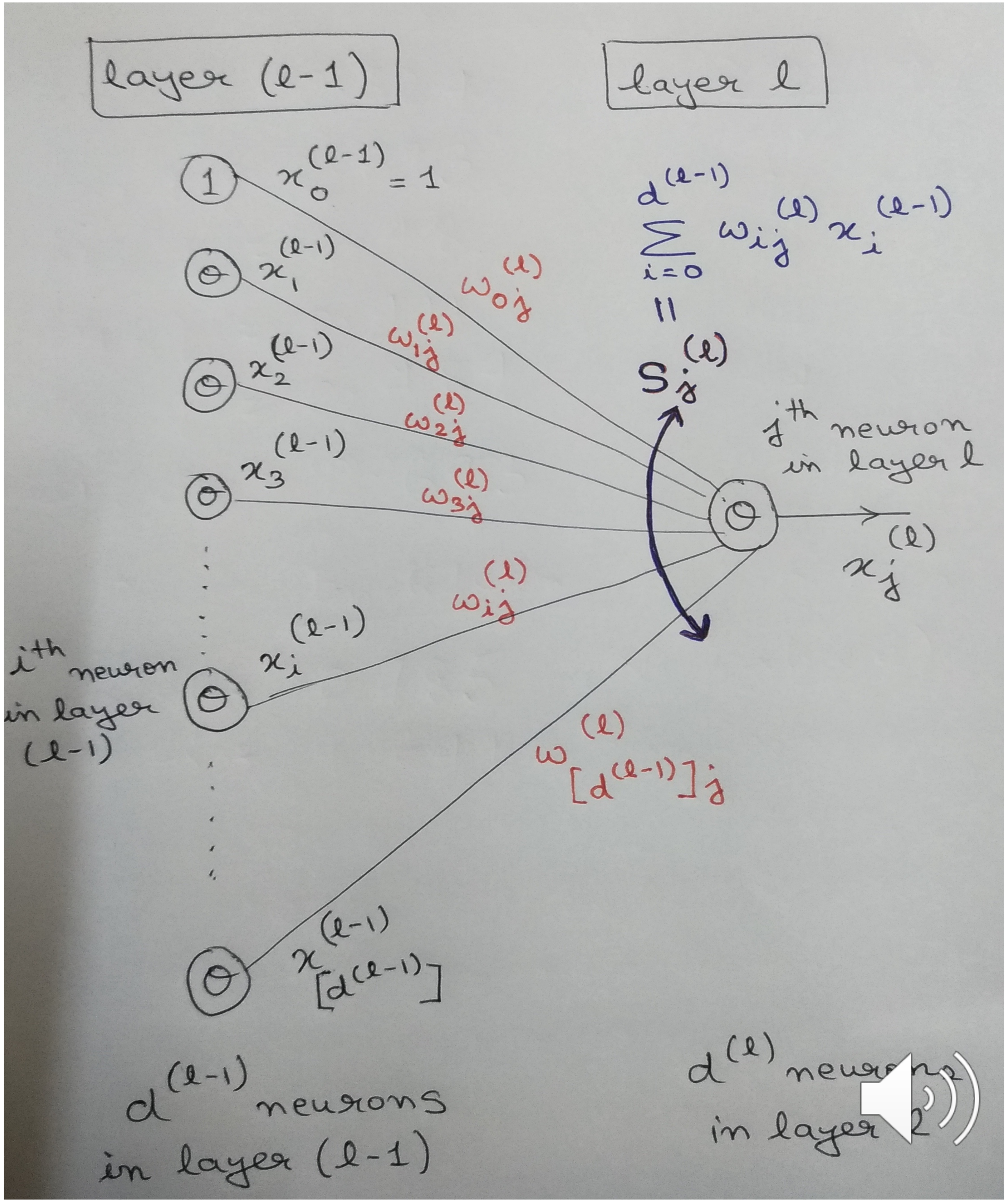
Note: number of neurons in different layers depends on the exact application

Every link in the neural network has a weight

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

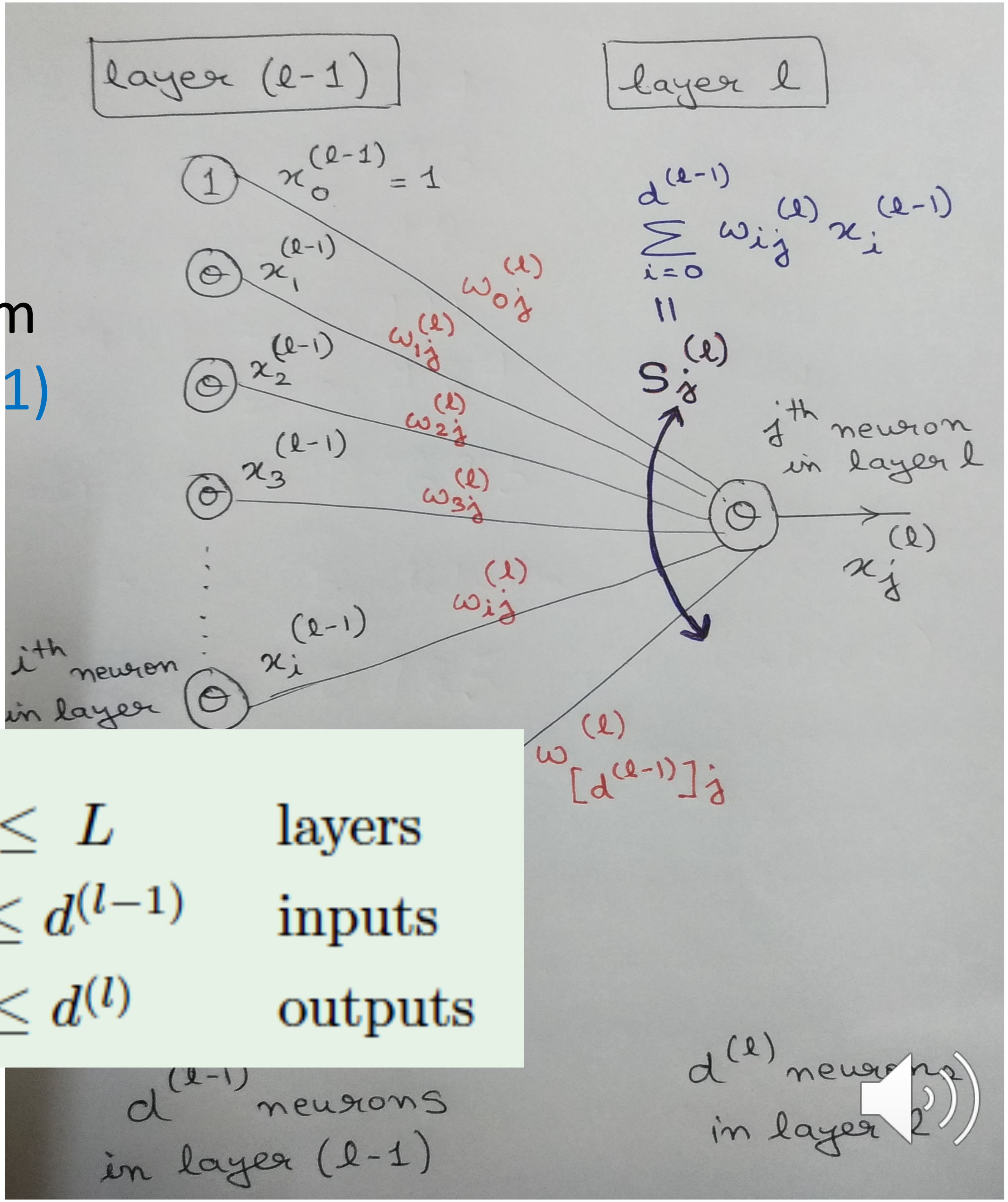
Weight of the link
from i-th neuron in layer (l-1)
to j-th neuron in layer l

Focusing on two layers

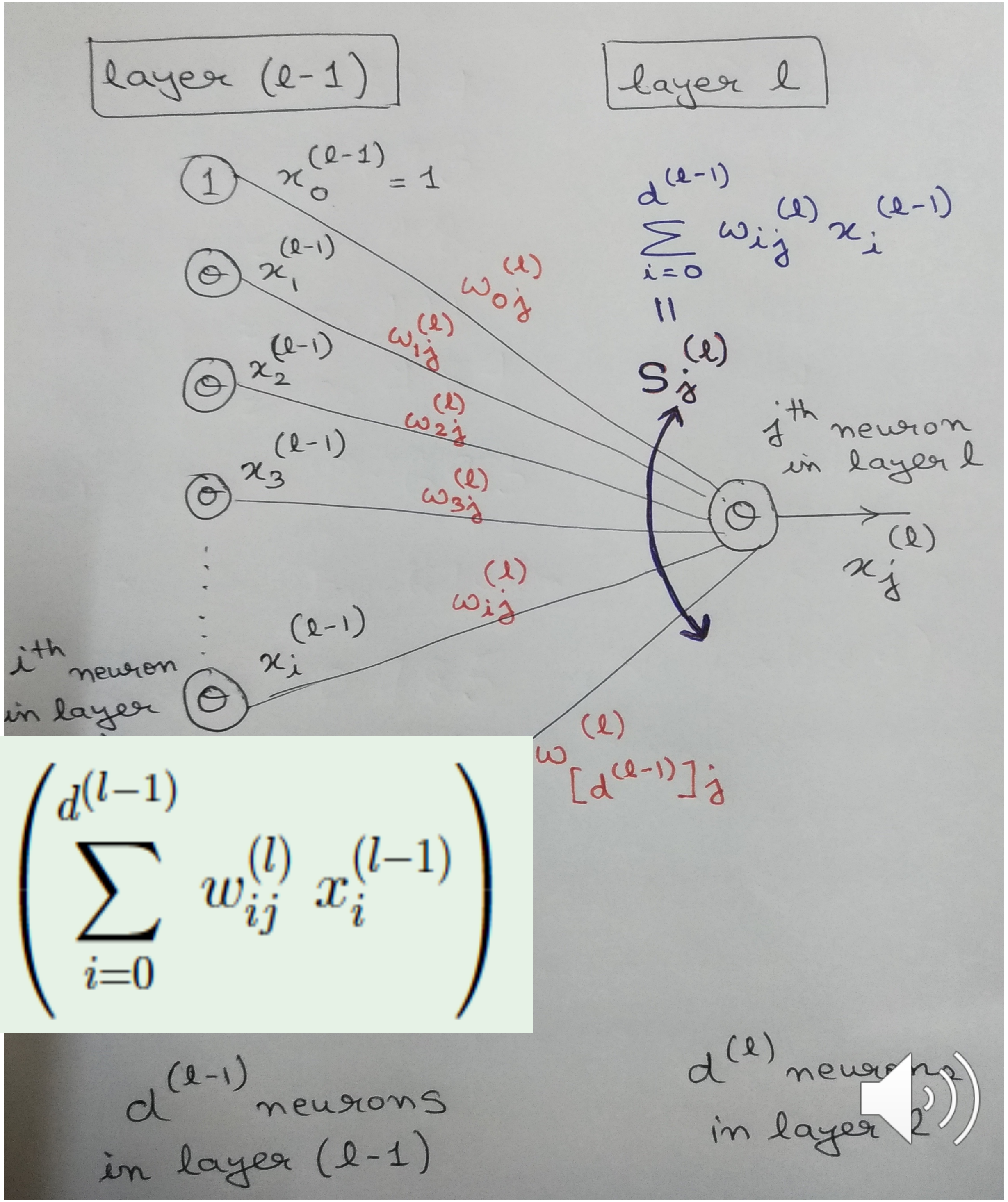


Weight of the link from
i-th neuron in layer (l-1)
to
j-th neuron in layer l

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



Output of the j-th neuron in layer l

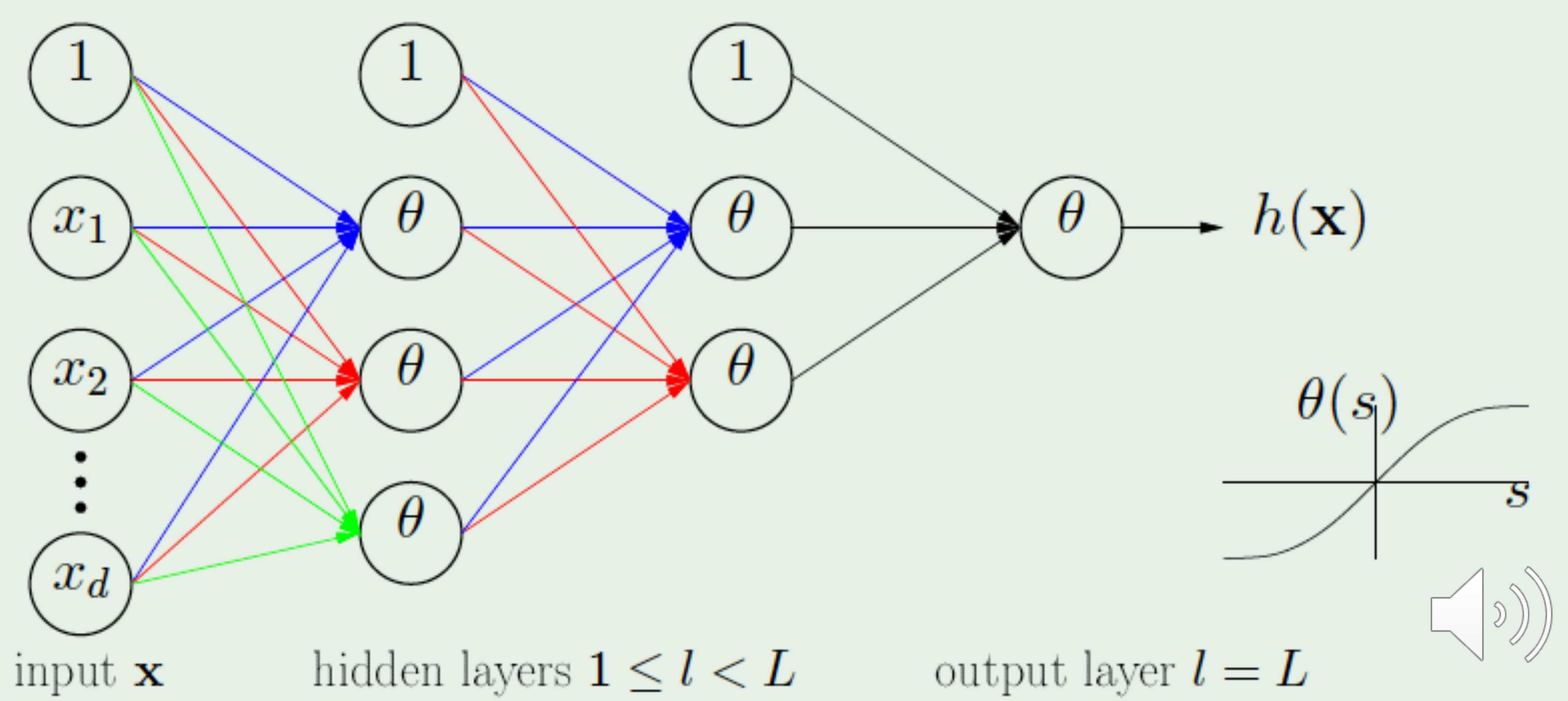


$$x_j^{(l)} = \theta(s_j^{(l)}) = \theta\left(\sum_{i=0}^{d^{(l-1)}} w_{ij}^{(l)}\ x_i^{(l-1)}\right)$$

How the network operates

Assuming all weights are known

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



How to get the weights?

- Till now what we have discussed – if the weights are known, how the neural network operates
- As ML practitioners, our job is to automatically learn the weights from training data
- Learning the weights efficiently: Backpropagation algorithm

BACKPROPAGATION ALGORITHM

Gradient Descent

- General setup (should be familiar)
  - Apply input x_n (with known output y_n) to the input layer
  - All weights $w = \{ w_{ij}^{(l)} \}$ determine the hypothesis $h(x_n)$
  - Compute error $e(h(x_n), y_n)$
  - Adjust the parameters in w --> compute a gradient for each parameter with the error: $\Delta w_{ij}^{(l)}$ = - learning rate * gradient
- What we studied earlier
  - Gradient computed based on all training examples (x_n, y_n): "Batch" GD
  - Epoch: using all training examples once to compute gradient
  - Inefficient for large datasets

Stochastic Gradient Descent (SGD)

- Pick one (x_n, y_n) at a time, apply GD to e(h(x_n), y_n)
- Idea: When done many times, over many training examples, average direction of descent will be the same as the "ideal" direction
- Benefits
  - Cheaper computation especially for large training sets used with neural networks
  - Randomization helps escape trivial local minima
- Like batch GD, cannot guarantee reaching global minima for non-convex error functions

Applying SGD

- All weights $w = \{ w_{ij}^{(l)} \}$ determine the hypothesis h(x)
- Error on example (x_n, y_n) is $e(h(x_n), y_n) = e(w)$ which can be squared error or logistic error or others
- To implement SGD, we need the gradient

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

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

The solution

- Backpropagation algorithm

- Idea (similar to recursion / induction)
  - Start with finding the gradients for the weights in the last layer l = L (output layer)
  - Assuming all gradients have been computed for layer l, devise a mechanism for computing gradients in layer (l-1)

- Gradients flow backwards in the network, giving the algorithm its name

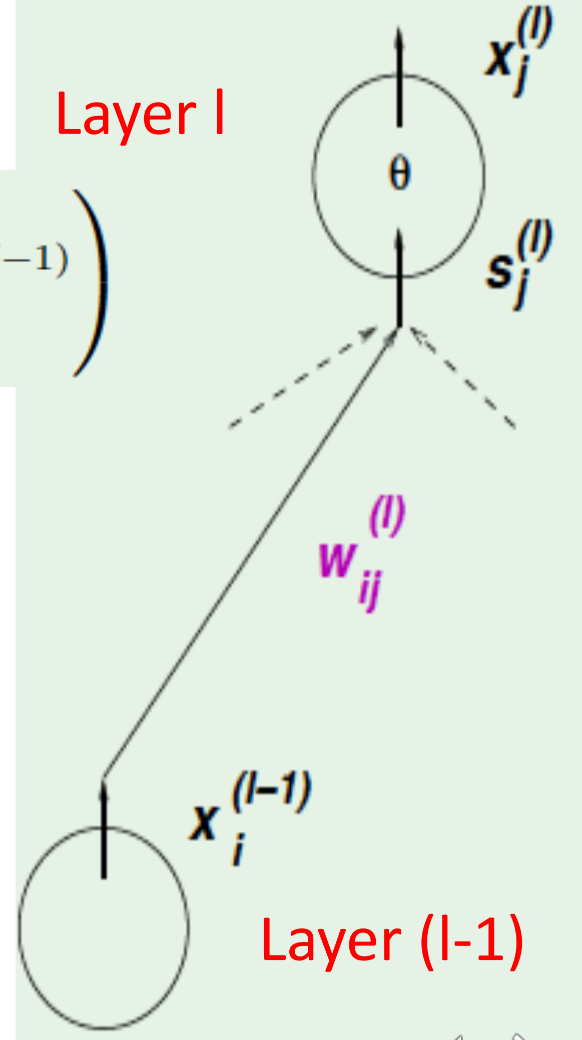
Computing $\frac{\partial \ \mathrm{e}(\mathbf{w})}{\partial \ w_{ij}^{(l)}}$

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

A trick for efficient computation:

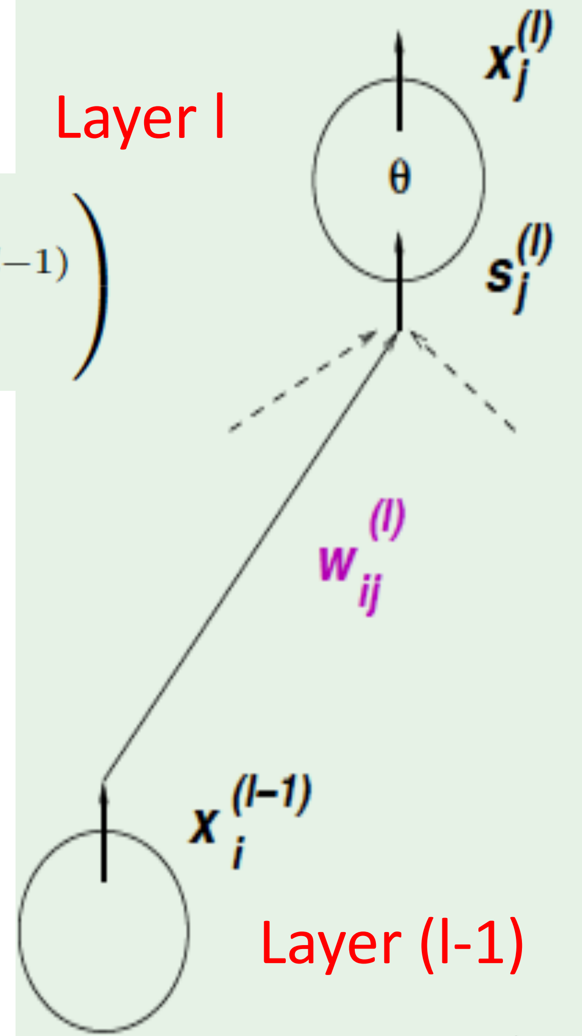
$$\frac{\partial \ \mathrm{e}(\mathbf{w})}{\partial \ w_{ij}^{(l)}} = \frac{\partial \ \mathrm{e}(\mathbf{w})}{\partial \ s_j^{(l)}} \times \frac{\partial \ s_j^{(l)}}{\partial \ w_{ij}^{(l)}}$$

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



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

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



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

Compute this recursively, starting from the last layer backwards

δ for the last (output) layer

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

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

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

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

Assuming squared error function

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

$$\theta'(s) = 1 - \theta^2(s) \quad \text{for the tanh}$$

$x_1^{(L)}$ = output of the only neuron in the last layer = $h(x_n)$

y_n = known output for the input x_n

θ = the non-linear function (e.g., tanh)

tanh function:

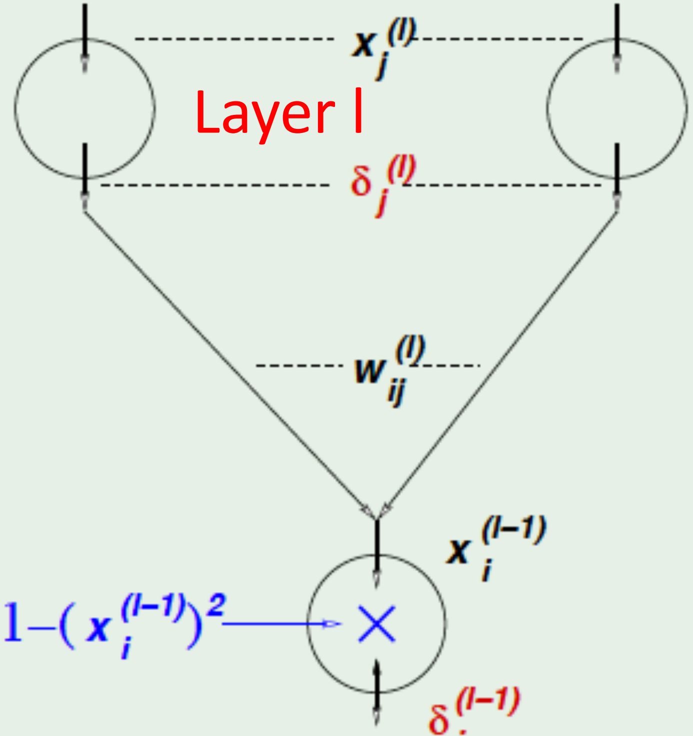
$$\frac{e^{s} - e^{-s}}{e^{s} + e^{-s}}$$

Back propagation of δ - Assuming all δ values of layer l have been computed already, how to compute δ for the i-th neuron (for any i) in layer (l-1)?

Back propagation of δ - Assuming all δ values of layer l have been computed already, how to compute δ for the i-th neuron (for any i) in layer (l-1)?

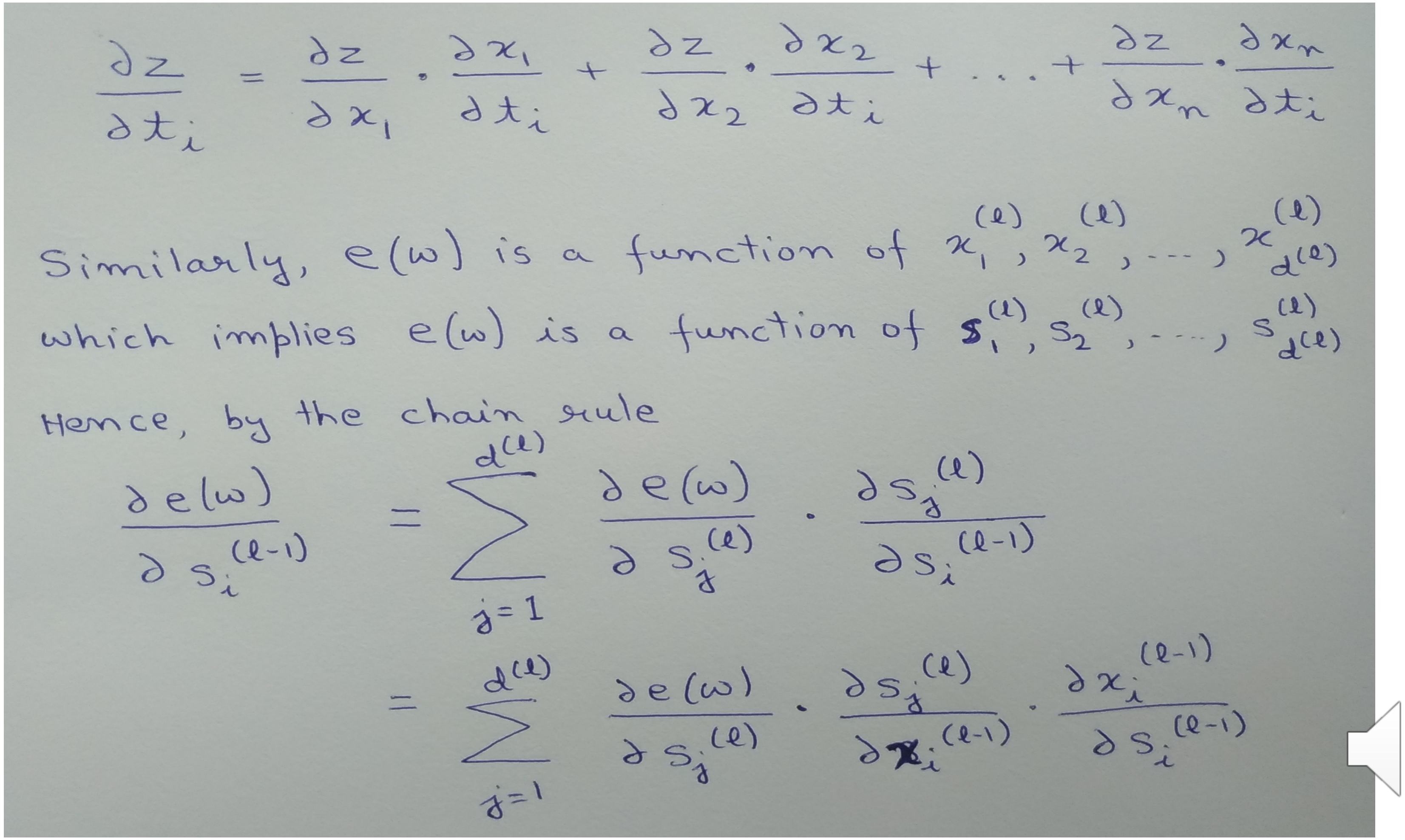
$$\begin{aligned}
\delta_i^{(l-1)} &= \frac{\partial\, \mathrm{e}(\mathbf{w})}{\partial\, s_i^{(l-1)}} \\
&= \sum_{j=1}^{d^{(l)}} \frac{\partial\, \mathrm{e}(\mathbf{w})}{\partial\, s_j^{(l)}} \times \frac{\partial\, s_j^{(l)}}{\partial\, x_i^{(l-1)}} \times \frac{\partial\, x_i^{(l-1)}}{\partial\, s_i^{(l-1)}}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

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



Recap of chain rule for partial derivatives

- Suppose z is a function of n variables $x_1, x_2, \ldots, x_n$ and each x_i is in turn a function of m variables $t_1, t_2, \ldots, t_m$
- Then for any variable t_i, i=1, 2, …, m, we have:



$$\frac{\partial z}{\partial t_i} = \frac{\partial z}{\partial x_1} \cdot \frac{\partial x_1}{\partial t_i} + \frac{\partial z}{\partial x_2} \cdot \frac{\partial x_2}{\partial t_i} + \ldots + \frac{\partial z}{\partial x_n} \cdot \frac{\partial x_n}{\partial t_i}$$

Similarly, $e(w)$ is a function of $x_1^{(l)}, x_2^{(l)}, \ldots, x_{d^{(l)}}^{(l)}$

which implies $e(w)$ is a function of $s_1^{(l)}, s_2^{(l)}, \ldots, s_{d^{(l)}}^{(l)}$

Hence, by the chain rule

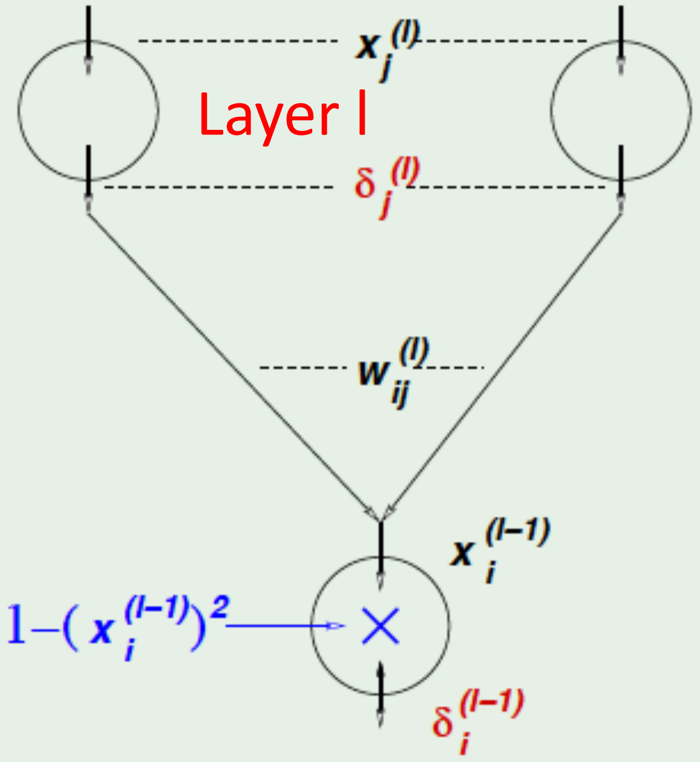
$$\frac{\partial e(w)}{\partial s_i^{(l-1)}} = \sum_{j=1}^{d^{(l)}} \frac{\partial e(w)}{\partial s_j^{(l)}} \cdot \frac{\partial s_j^{(l)}}{\partial s_i^{(l-1)}}$$

$$= \sum_{j=1}^{d^{(l)}} \frac{\partial e(w)}{\partial s_j^{(l)}} \cdot \frac{\partial s_j^{(l)}}{\partial x_i^{(l-1)}} \cdot \frac{\partial x_i^{(l-1)}}{\partial s_i^{(l-1)}}$$

Back propagation of δ - Assuming all δ values of layer l have been computed already, how to compute δ for the i-th neuron (for any i) in layer (l-1)?

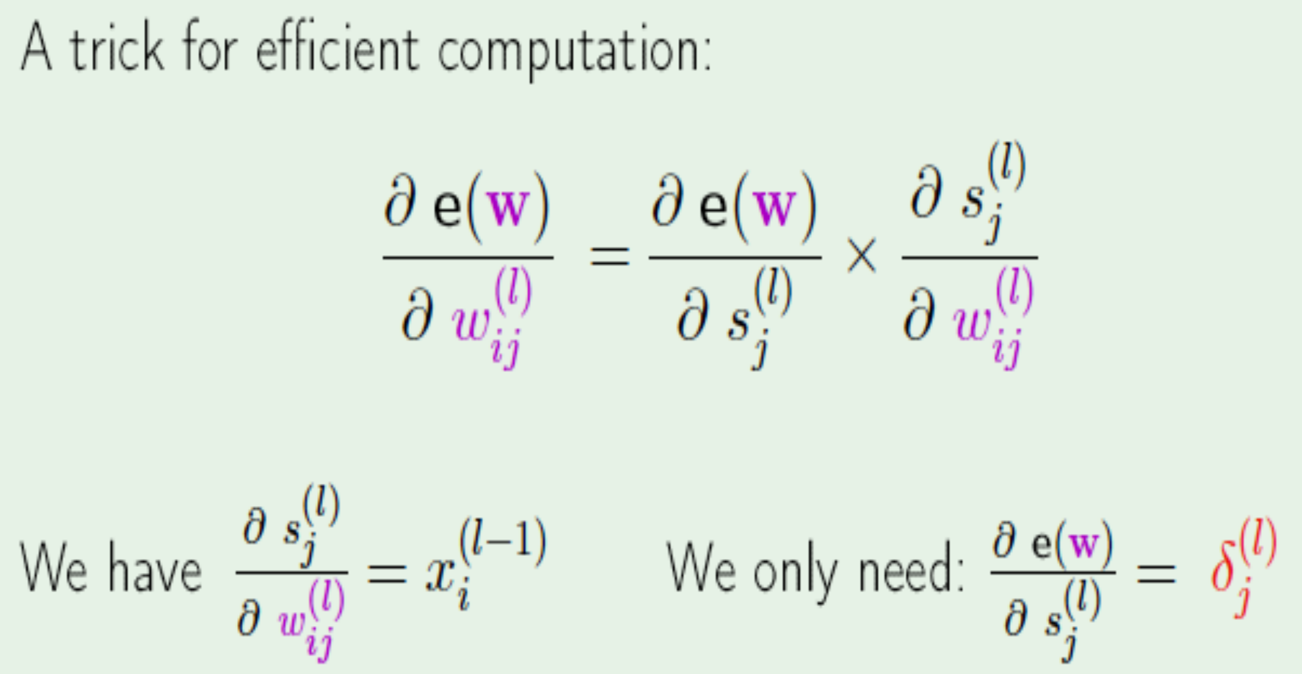
$$
\begin{aligned}
\delta_i^{(l-1)} &= \frac{\partial\, e(\mathbf{w})}{\partial\, s_i^{(l-1)}} \\
&= \sum_{j=1}^{d^{(l)}} \frac{\partial\, e(\mathbf{w})}{\partial\, s_j^{(l)}} \times \frac{\partial\, s_j^{(l)}}{\partial\, x_i^{(l-1)}} \times \frac{\partial\, x_i^{(l-1)}}{\partial\, s_i^{(l-1)}} \\
&=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

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



Since we assume θ to be tanh function, the derivative is computed as shown

Backpropagation algorithm: summary



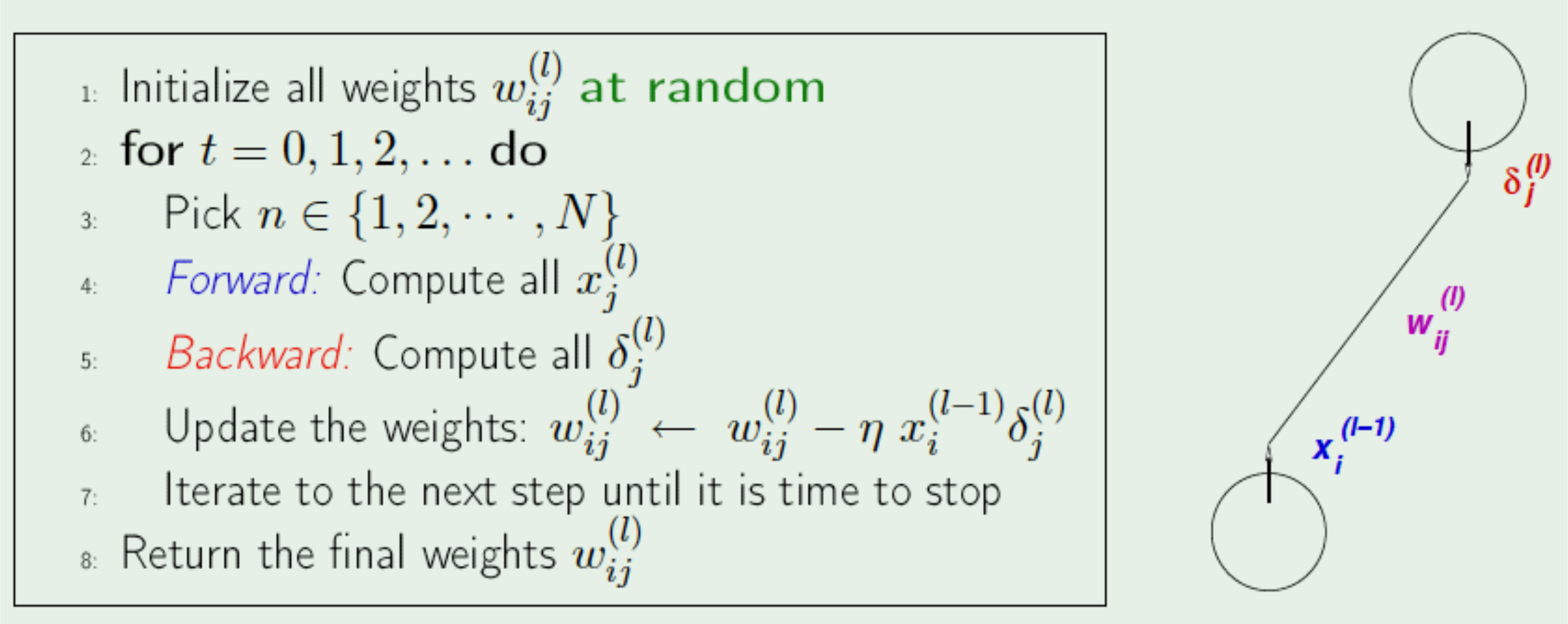
A trick for efficient computation:

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

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

- We devised a recursive way of computing δ values
  - First we compute δ values for the output layer l = L
  - δ values of layer (l-1) are computed based on the δ values of layer l
- So the δ values (and hence the gradient values) propagate backwards through the network

Backpropagation 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: Each iteration uses only one training sample: SGD

Not guaranteed to reach global minima; will reach a local minima depending on initialization, which sample chosen in which iteration, etc.

Discussion

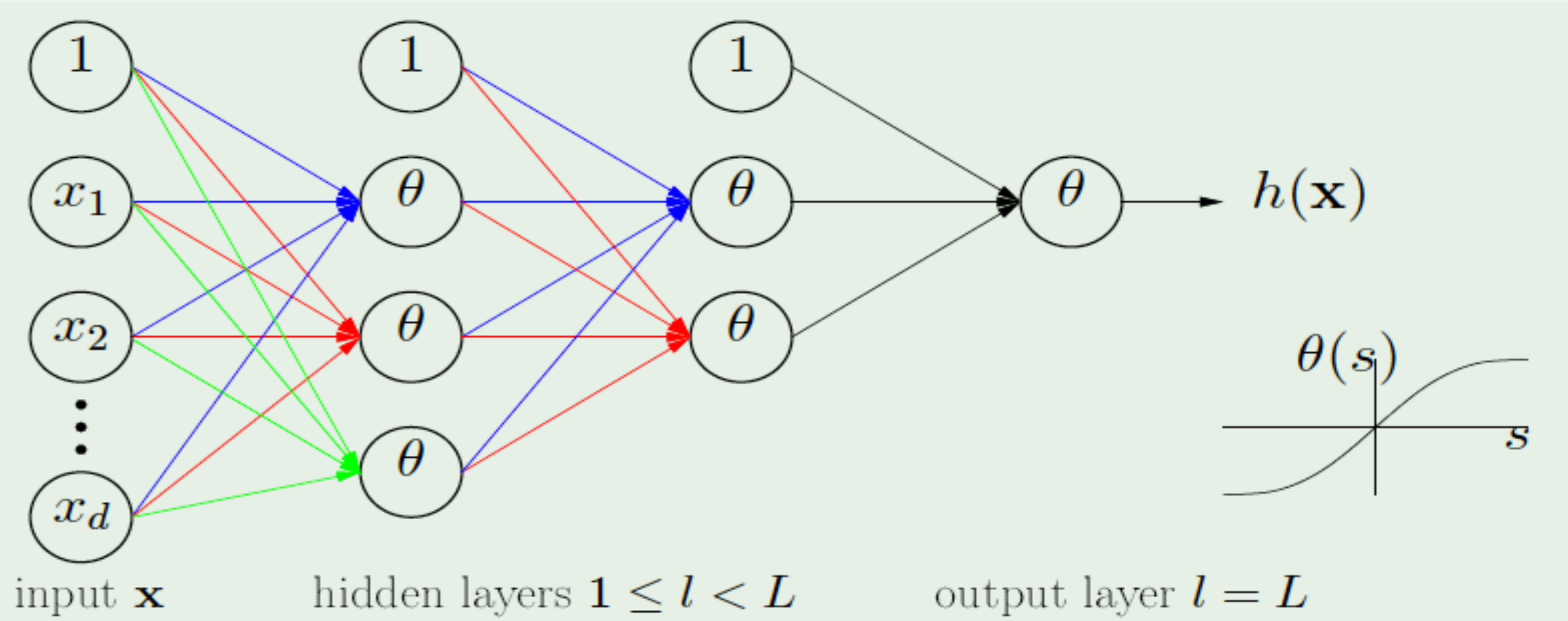
- Zero initialization will not work
  - If all weights initialized to zero, either all x's or all δ's will be zero; hence weights would not be adjusted
  - Weights have to be initialized randomly

- Intelligent ways of initializing weights can be used to ensure faster convergence and better weight values
  - Based on models used earlier for similar tasks
  - Called pre-trained models

Discussion

- Many things to decide
  - How many layers? How many neurons in each layer?
  - What error function? What non-linear function?
  - What learning rate? …
- All these are hyperparameters
  - Decide from experience, or
  - Use validation set to determine what values perform well
- Size of network decides the number of parameters (weights) – should be decided based on available training data

What are the hidden layers doing?



Hidden layers are learning higher level non-linear transforms of the input features

Advantage: we do not need to decide what non-linear transforms to learn; the network figures that out

Disadvantage: Interpretability of output is difficult – may not have a clear idea of what the hidden layer is learning

THANK YOU

Questions can be mailed to Dr. S. Ghosh (saptarshi@cse.iitkgp.ac.in)