

COMP5212: Machine Learning

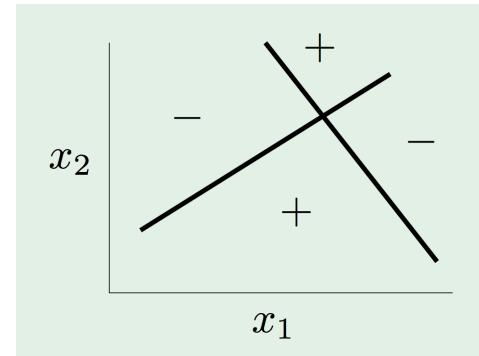
Lecture 12

Minhao Cheng

Neural networks

Another way to introduce nonlinearity

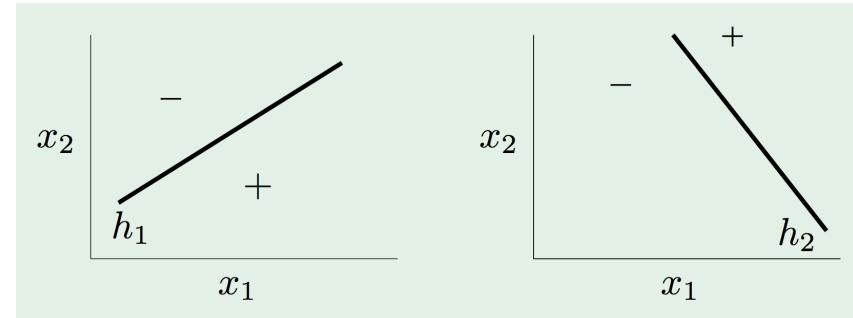
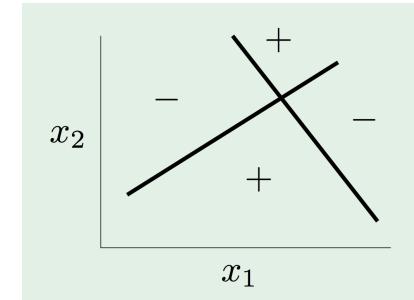
- How to generate this nonlinear hypothesis?



Neural network

Another way to introduce nonlinearity

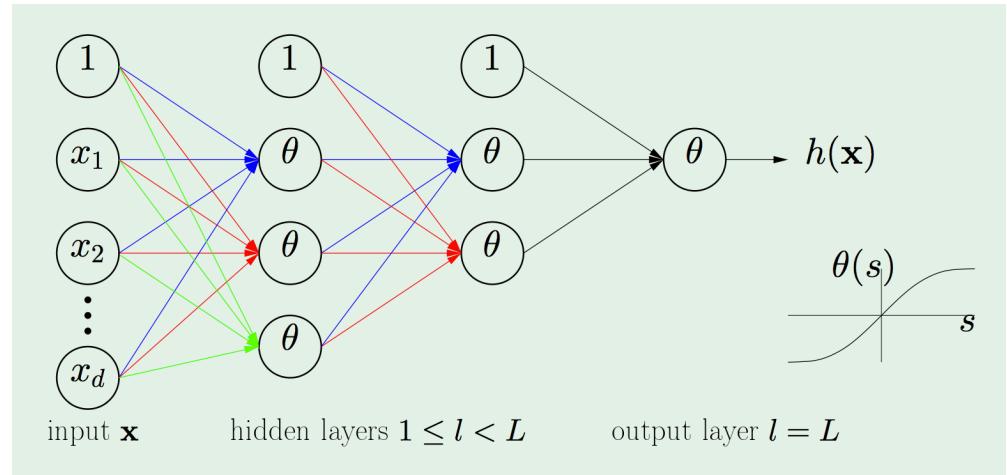
- How to generate this nonlinear hypothesis?
- Combining multiple linear hyperplanes to construct nonlinear hypothesis



Neural Network

Definition

- Input layer: d neurons (input features)
- Neurons from layer 1 to L : Linear combination of previous layers + activation function
 - $\theta(w^T x)$, θ : activation function
- Final layer: one neuron \Rightarrow prediction by $\text{sign}(h(x))$

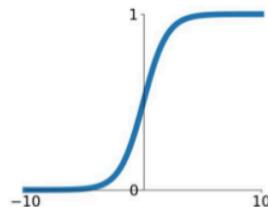


Neural network

Activation Function

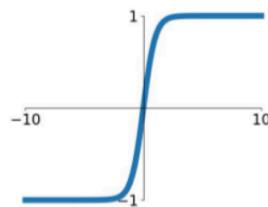
Sigmoid

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



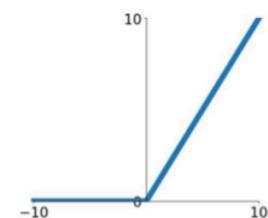
tanh

$$\tanh(x)$$



ReLU

$$\max(0, x)$$



Neural Network

Activation: Formal Definitions

Weight: $w_{\textcolor{red}{i}\textcolor{green}{j}}^{(\textcolor{blue}{l})}$ $\begin{cases} 1 \leq \textcolor{blue}{l} \leq L & : \text{layers} \\ 0 \leq \textcolor{red}{i} \leq d^{(l-1)} & : \text{inputs} \\ 1 \leq \textcolor{green}{j} \leq d^{(l)} & : \text{outputs} \end{cases}$

- bias: $b_j^{(l)}$: added to the j-th neuron in the l-th layer

Neural Network

Formal Definitions

- Weight: $w_{ij}^{(l)}$ $\begin{cases} 1 \leq l \leq L & : \text{layers} \\ 0 \leq i \leq d^{(l-1)} & : \text{inputs} \\ 1 \leq j \leq d^{(l)} & : \text{outputs} \end{cases}$
- bias: $b_j^{(l)}$: added to the j -th neuron in the l -th layer
- j -th neuron in the l -th layer:
 - $x_j^{(l)} = \theta(s_j^{(l)}) = \theta\left(\sum_{i=0}^{d^{(l-1)}} w_{ij}^{(l)} x_i^{(l-1)} + b_j^{(l)}\right)$

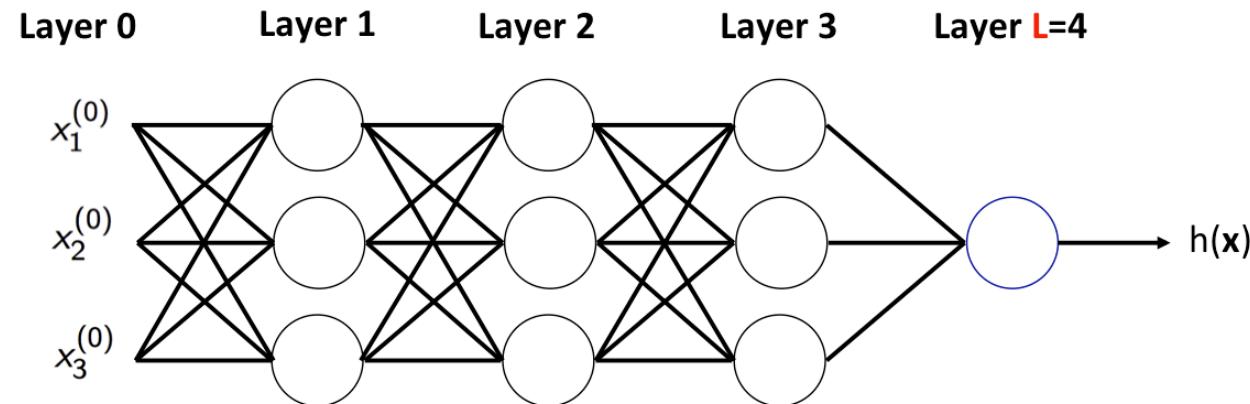
Neural Network

Formal Definitions

- Weight: $w_{ij}^{(l)}$ $\begin{cases} 1 \leq l \leq L & : \text{layers} \\ 0 \leq i \leq d^{(l-1)} & : \text{inputs} \\ 1 \leq j \leq d^{(l)} & : \text{outputs} \end{cases}$
- bias: $b_j^{(l)}$: added to the j-th neuron in the l-th layer
- j-th neuron in the l-the layer:
 - $x_j^{(l)} = \theta(s_j^{(l)}) = \theta\left(\sum_{i=0}^{d^{(l-1)}} w_{ij}^{(l)} x_i^{(l-1)} + b_j^{(l)}\right)$
- Output:
 - $h(x) = x_1^{(L)}$

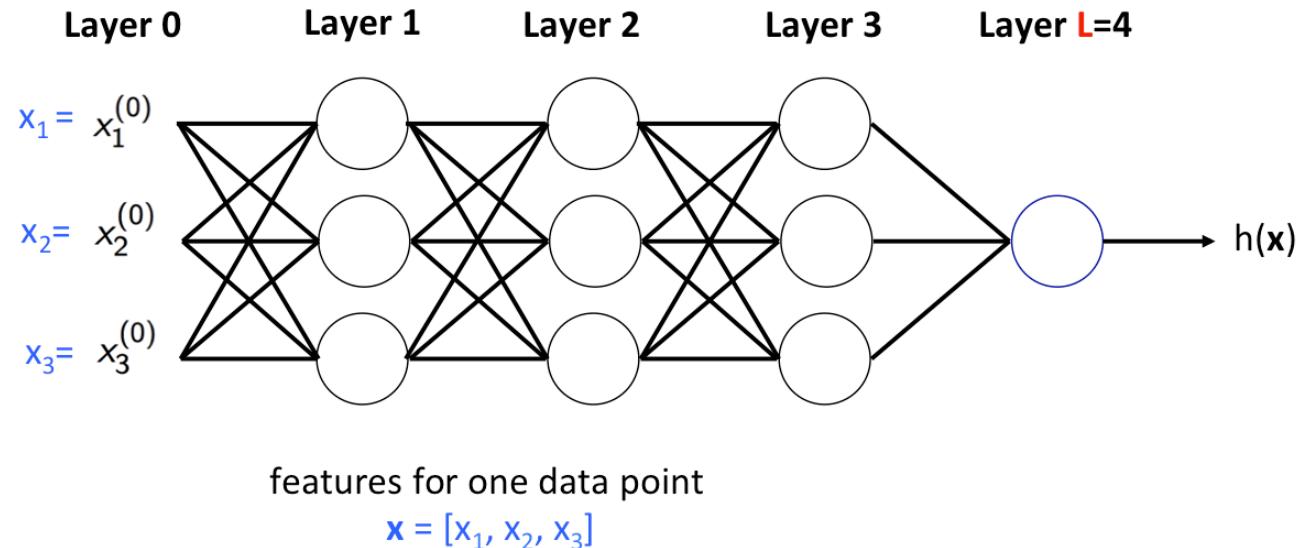
Neural Network

Forward propagation



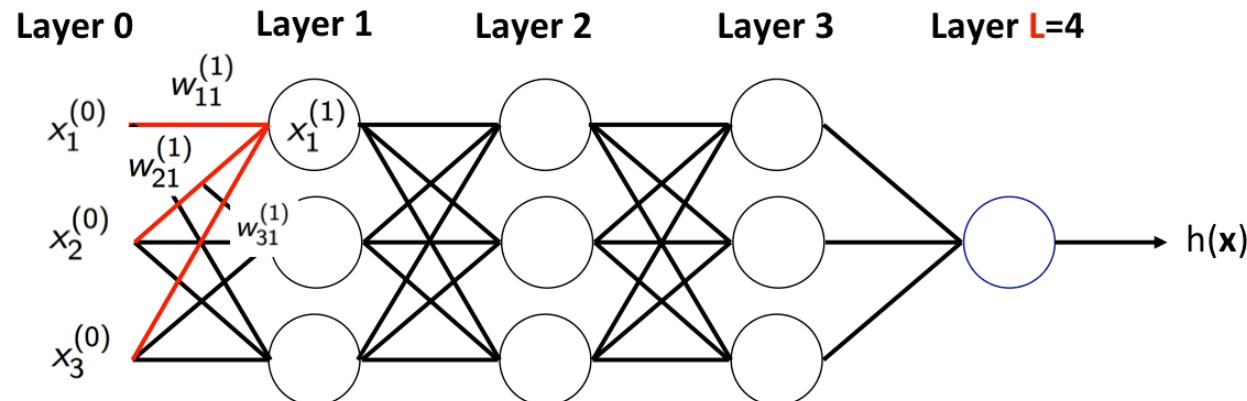
Neural Network

Forward propagation



Neural Network

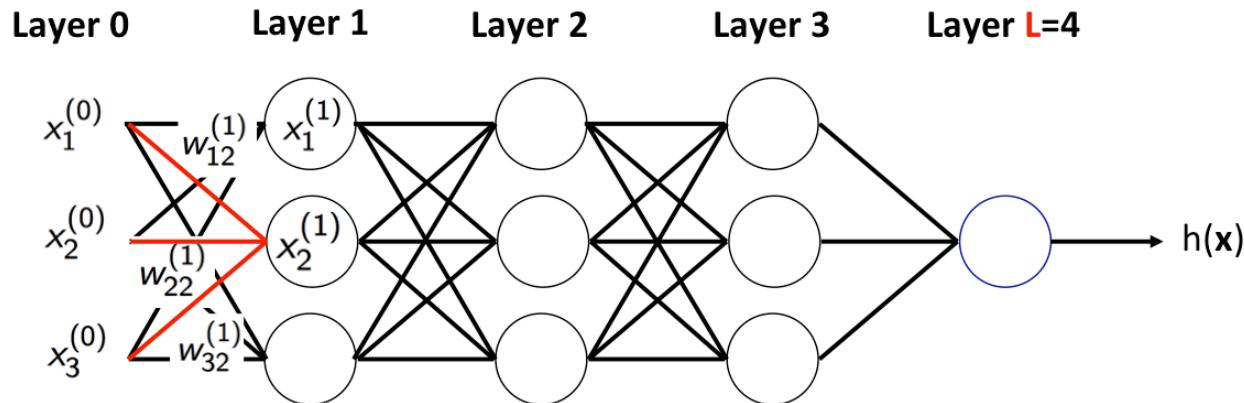
Forward propagation



$$x_1^{(1)} = \theta\left(\sum_{i=1}^3 w_{i1}^{(1)} x_i^{(0)}\right)$$

Neural Network

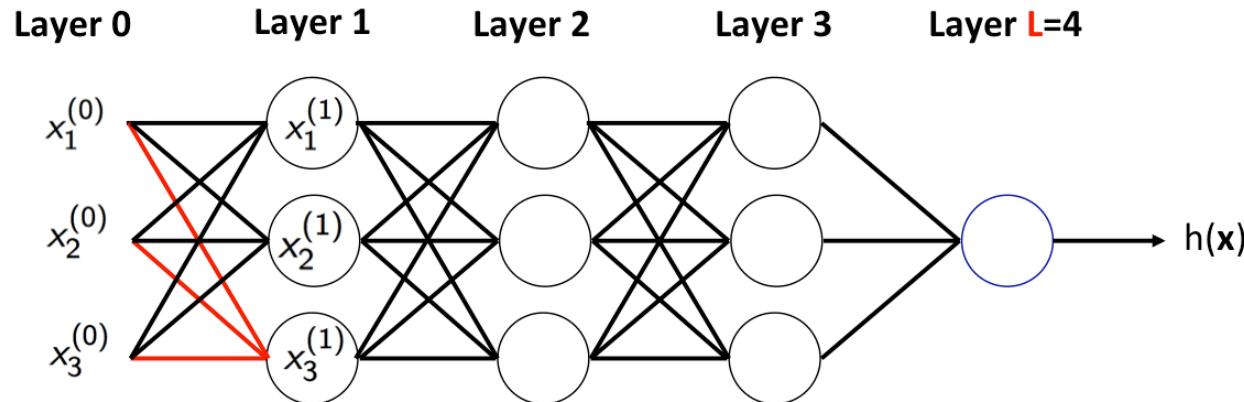
Forward propagation



$$x_2^{(1)} = \theta\left(\sum_{i=1}^3 w_{i2}^{(1)} x_i^{(0)}\right)$$

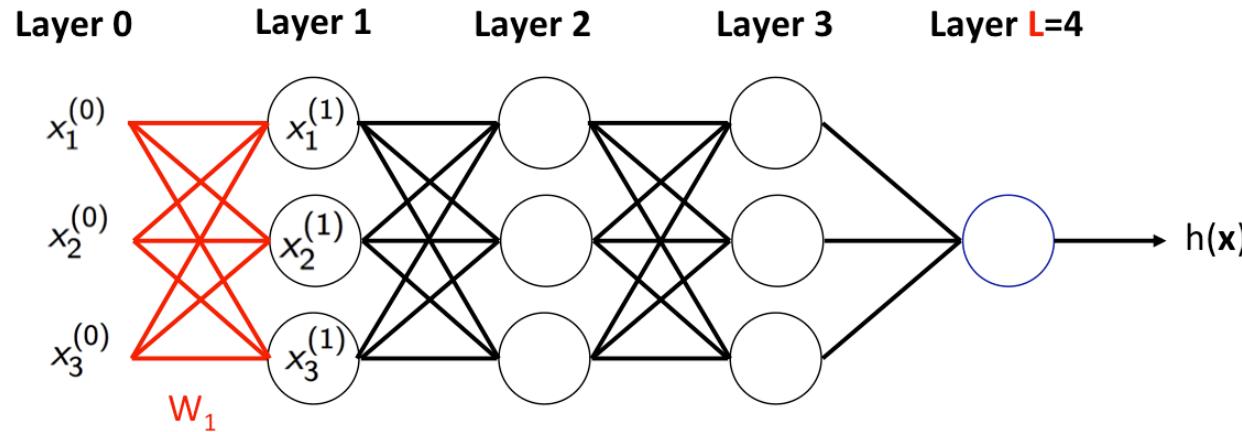
Neural Network

Forward propagation



Neural Network

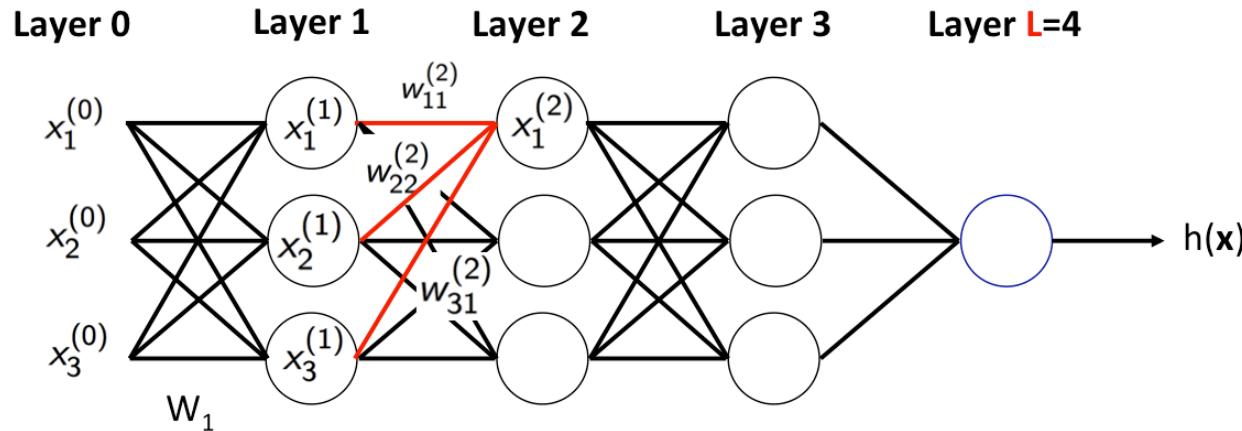
Forward propagation



$$\mathbf{x}^{(1)} = \begin{bmatrix} x_1^{(1)} \\ x_2^{(1)} \\ x_3^{(1)} \end{bmatrix} = \theta \left(\begin{bmatrix} w_{11}^{(1)} & w_{21}^{(1)} & w_{31}^{(1)} \\ w_{12}^{(1)} & w_{22}^{(1)} & w_{32}^{(1)} \\ w_{13}^{(1)} & w_{23}^{(1)} & w_{33}^{(1)} \end{bmatrix} \times \begin{bmatrix} x_1^{(0)} \\ x_2^{(0)} \\ x_3^{(0)} \end{bmatrix} \right) = \theta(\mathbf{W}_1 \mathbf{x}^{(0)})$$

Neural Network

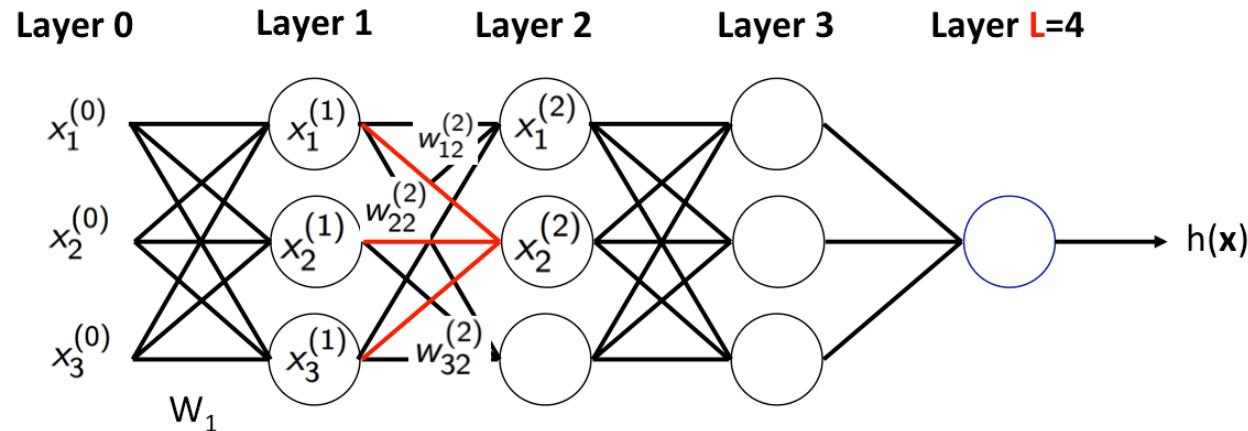
Forward propagation



$$x_1^{(2)} = \theta\left(\sum_{i=1}^3 w_{i1}^{(2)} x_i^{(1)}\right)$$

Neural Network

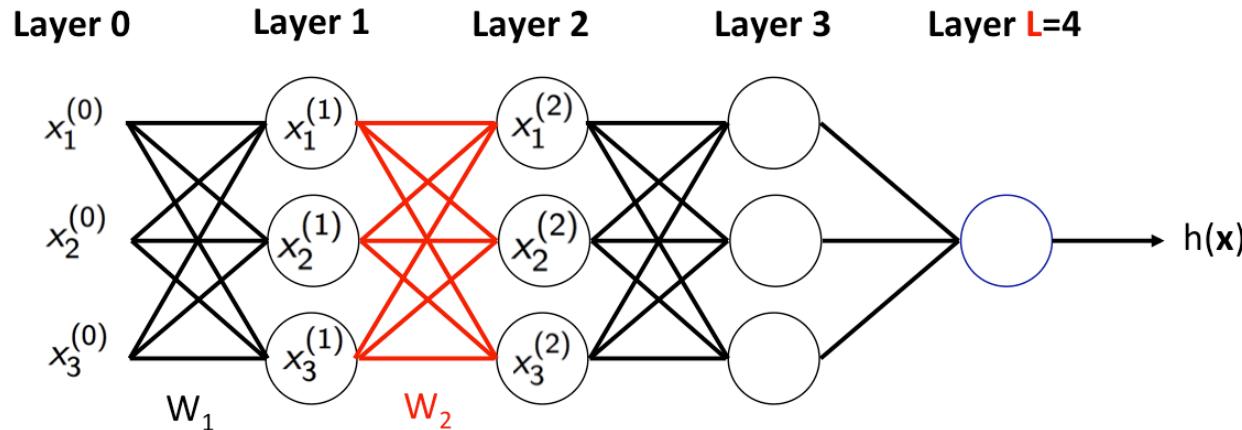
Forward propagation



$$x_2^{(2)} = \theta\left(\sum_{i=1}^3 w_{i2}^{(2)} x_i^{(1)}\right)$$

Neural Network

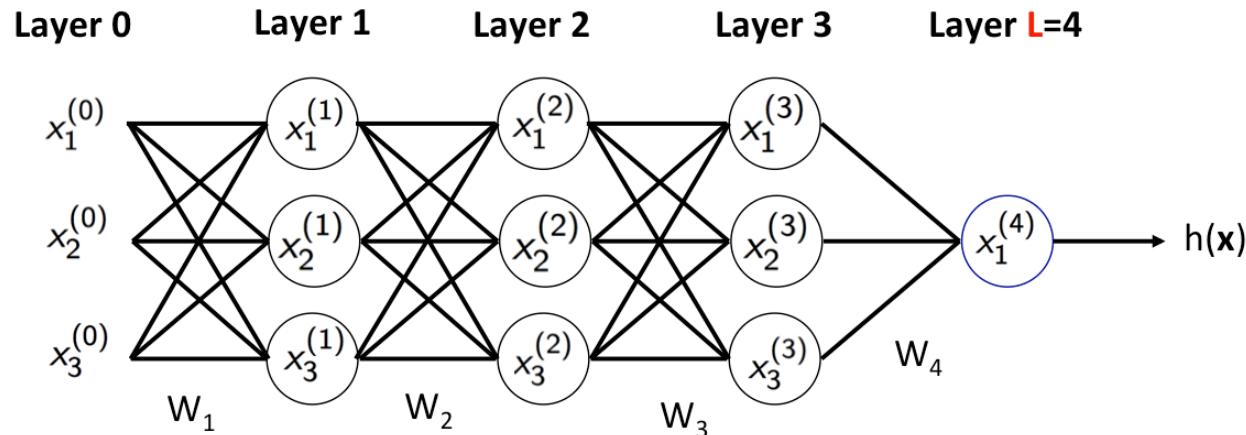
Forward propagation



$$\mathbf{x}^{(2)} = \begin{bmatrix} x_1^{(2)} \\ x_2^{(2)} \\ x_3^{(2)} \end{bmatrix} = \theta \left(\begin{bmatrix} w_{11}^{(2)} & w_{21}^{(2)} & w_{31}^{(2)} \\ w_{12}^{(2)} & w_{22}^{(2)} & w_{32}^{(2)} \\ w_{13}^{(2)} & w_{23}^{(2)} & w_{33}^{(2)} \end{bmatrix} \times \begin{bmatrix} x_1^{(1)} \\ x_2^{(1)} \\ x_3^{(1)} \end{bmatrix} \right) = \theta(W_2 \mathbf{x}^{(1)})$$

Neural Network

Forward propagation



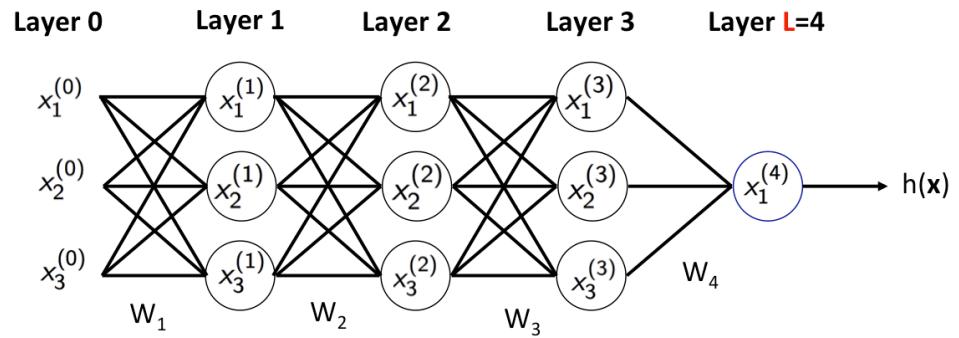
$$\begin{aligned} h(\mathbf{x}) &= x_1^{(4)} = \theta(W_4 \mathbf{x}^{(3)}) = \theta(W_4 \theta(W_3 \mathbf{x}^{(2)})) \\ &= \dots = \theta(W_4 \theta(W_3 \theta(W_2 \theta(W_1 \mathbf{x})))) \end{aligned}$$

Neural Network

Forward propagation

- With the bias term:

$$h(x) = \theta(W_4\theta(W_3\theta(W_2\theta(W_1x + b_1) + b_2) + b_3) + b_4)$$



$$\begin{aligned} h(\mathbf{x}) &= x_1^{(4)} = \theta(W_4 \mathbf{x}^{(3)}) = \theta(W_4 \theta(W_3 \mathbf{x}^{(2)})) \\ &= \dots = \theta(W_4 \theta(W_3 \theta(W_2 \theta(W_1 \mathbf{x})))) \end{aligned}$$

Neural Network

Capacity of neural networks

- Universal approximation theorem (Horink, 1991):
 - “A neural network with single hidden layer can approximate any continuous function arbitrarily well, given enough hidden units”
 - True for commonly used activations (ReLU, sigmoid, ...)

Neural Network

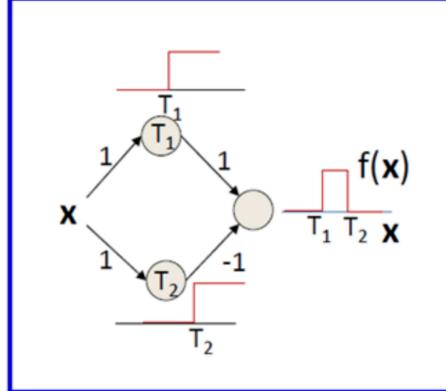
Capacity of neural networks

- Universal approximation theorem (Horink, 1991):
 - “A neural network with single hidden layer can approximate any continuous function arbitrarily well, given enough hidden units”
- True for commonly used activations (ReLU, sigmoid, ...)
- However,
 - Can’t have infinitely number of hidden units in practice
 - Only shows the ideal model exists, but we may not be able to find it

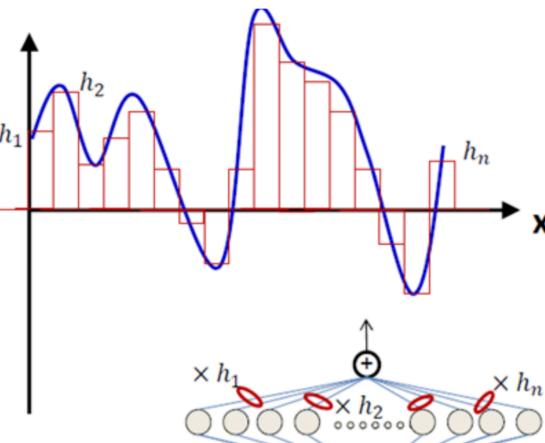
Neural Network

Universal approximation for step activation

- How to approximate an arbitrary function by single-layer NN with **step function** as activation:



2 hidden units to form a “rectangle”



any function can be approximated by rectangles

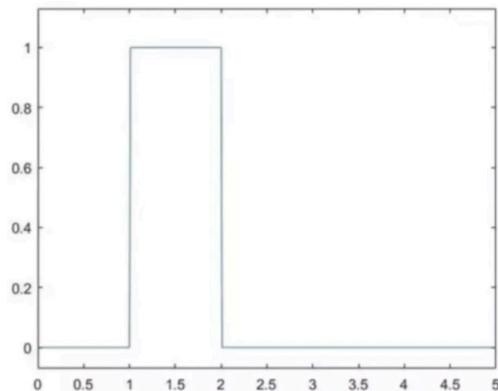
(figure from <https://medium.com/analytics-vidhya>)

Neural Network

Universal approximation for ReLU

- Use ReLU to approximate a rectangle:

- $\frac{1}{\epsilon}(\max(0,x - a) - \max(0,x - a - \epsilon) - \max(0,x - b) + \max(0,x - b - \epsilon))$



- Use (approximate) rectangles to approximate arbitrary function

Neural Network

Training

- Weights $W = \{W_1, \dots, W_L\}$ and bias $\{b_1, \dots, b_L\}$ determine $h(x)$
- Learning the weights: solve ERM with SGD
 - Loss on example (x_n, y_n) is Emperical Risk Minimization
 - $e(h(x_n), y_n) = e(W)$

Neural Network

Training

- Weights $W = \{W_1, \dots, W_L\}$ and bias $\{b_1, \dots, b_L\}$ determine $h(x)$
- Learning the weights: solve ERM with SGD
- Loss on example (x_n, y_n) is
 - $e(h(x_n), y_n) = e(W)$
- To implement SGD, we need the gradient
 - $\nabla e(W) : \left\{ \frac{\partial e(W)}{\partial w_{ij}^{(l)}} \right\}$ for all i, j, l (for simplicity we ignore bias in the derivations)

Neural Network

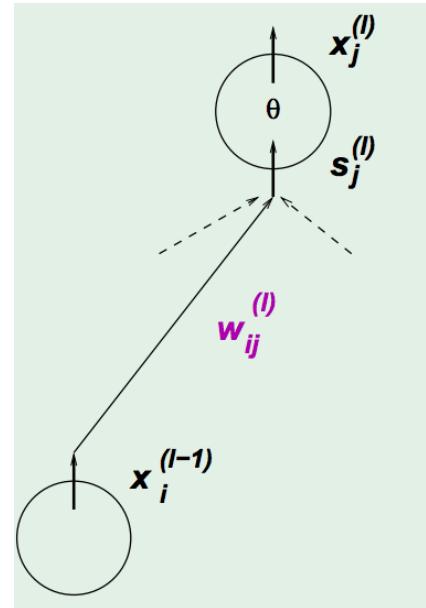
Computing Gradient $\frac{\partial e(W)}{\partial w_{ij}^{(l)}}$

- Use chain rule:

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

$$s_j^{(l)} = \sum_{i=1}^d x_i^{(l-1)} w_{ij}^{(l)}$$

$$\text{We have } \frac{\partial s_j^{(l)}}{\partial w_{ij}^{(l)}} = x_i^{(l-1)}$$



Neural Network

Computing Gradient $\frac{\partial e(W)}{\partial w_{ij}^{(l)}}$

- Define $\delta_j^{(l)} := \frac{\partial e(W)}{\partial s_j^{(l)}}$
- Compute by layer-by-layer:

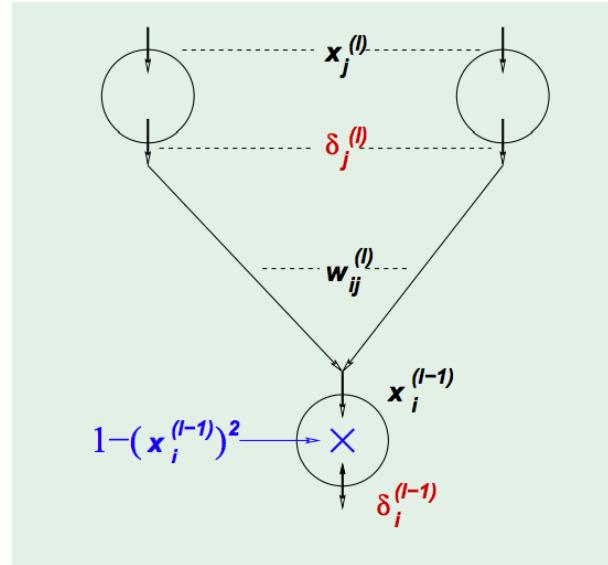
$$\delta_i^{(l-1)} = \frac{\partial e(W)}{\partial s_i^{(l-1)}}$$

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

$$= \sum_{j=1}^d \delta_j^{(l)} \times w_{ij}^{(l)} \times \theta'(s_i^{(l-1)}),$$

- where $\theta'(s) = 1 - \theta^2(s)$ for tanh $\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$

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



Neural Network

Final layer

- (Assume square loss)

- $e(W) = (x_1^{(L)} - y_n)^2$

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

- So,

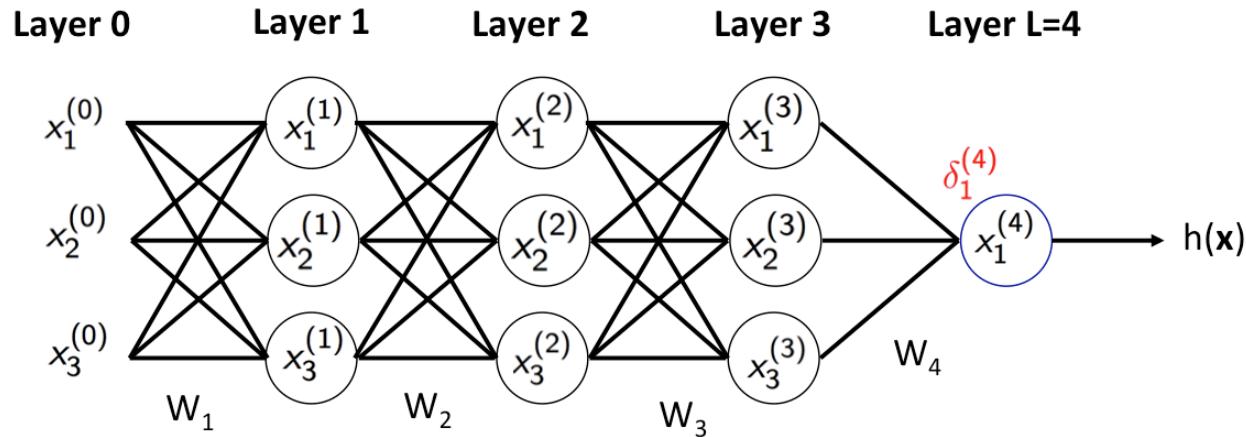
$$\delta_1^{(L)} = \frac{\partial e(W)}{\partial s_1^{(L)}}$$

- $$= \frac{\partial e(W)}{\partial x_1^{(L)}} \times \frac{\partial x_1^{(L)}}{\partial s_1^{(L)}}$$

- $$= 2(x_1^{(L)} - y_n) \times \theta'(s_1^{(L)})$$

Neural Network

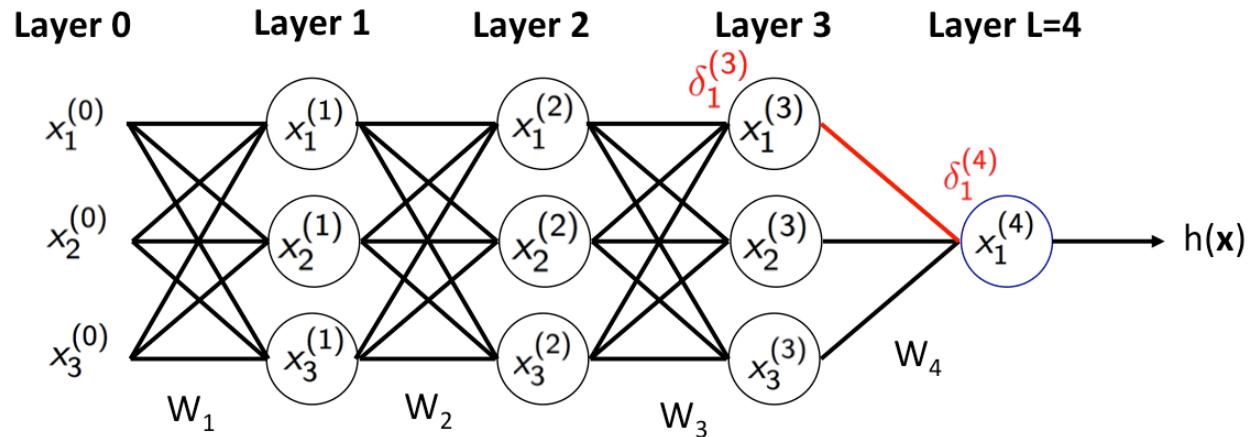
Backward propagation



$$\delta_1^{(4)} = 2(x_1^{(4)} - y_n) \times (1 - (x_1^{(4)})^2)$$

Neural Network

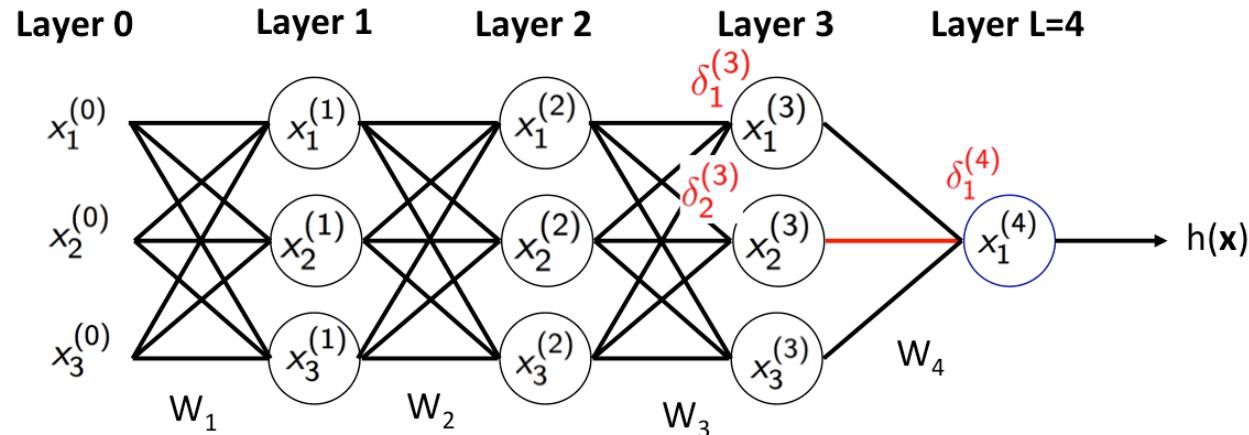
Backward propagation



$$\delta_1^{(3)} = (1 - (x_1^{(3)})^2) \times \delta_1^{(4)} \times w_{11}^{(4)}$$

Neural Network

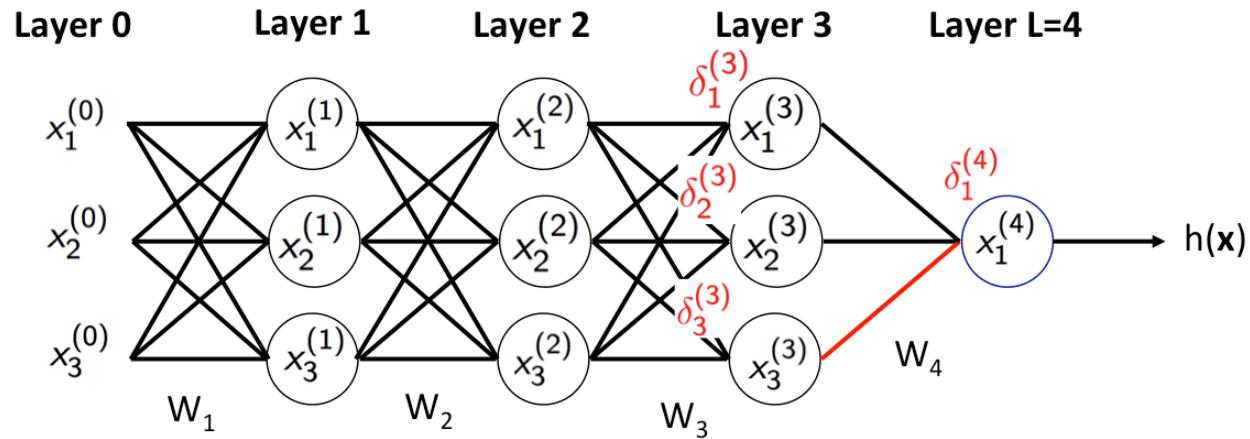
Backward propagation



$$\delta_2^{(3)} = (1 - (x_2^{(3)})^2) \times \delta_1^{(4)} \times w_{21}^{(4)}$$

Neural Network

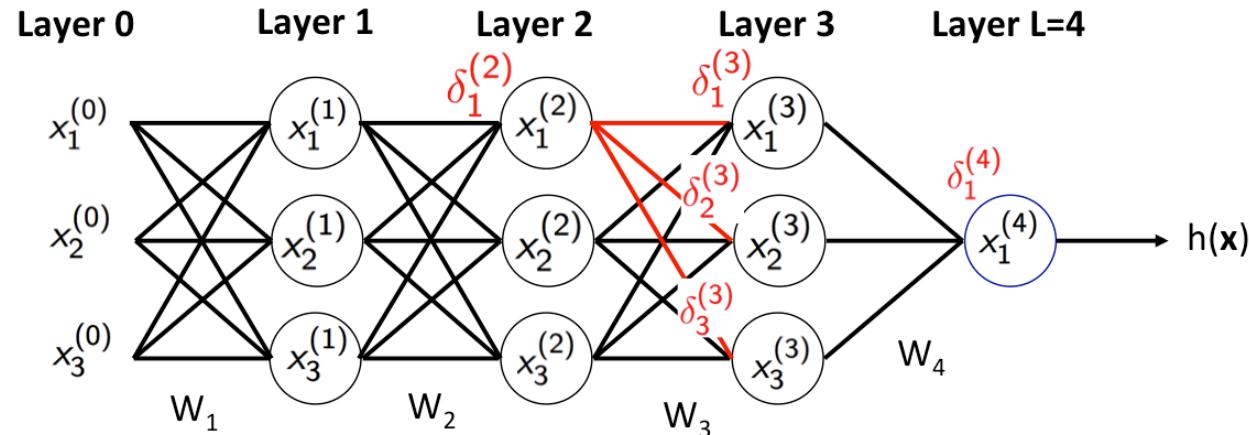
Backward propagation



$$\delta_3^{(3)} = (1 - (x_3^{(3)})^2) \times \delta_1^{(4)} \times w_{31}^{(4)}$$

Neural Network

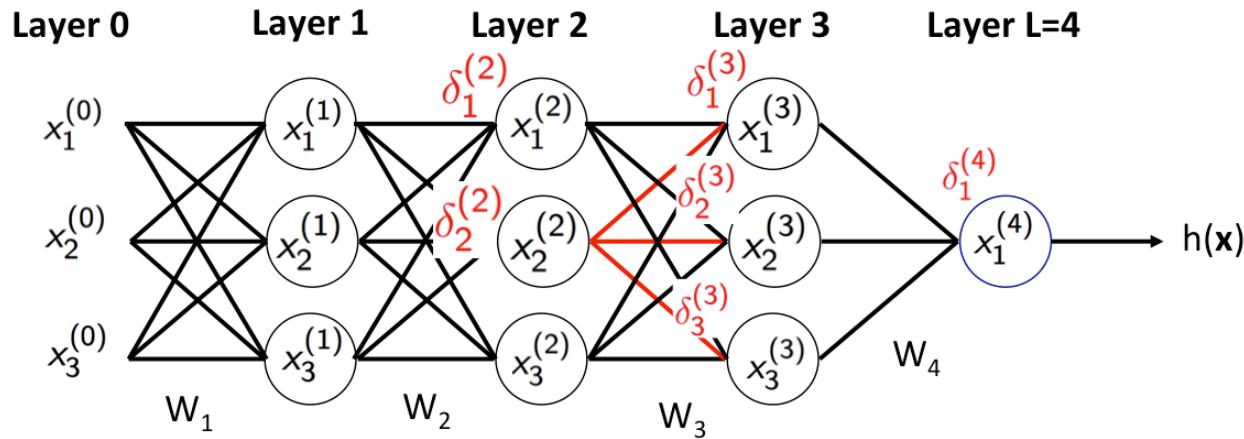
Backward propagation



$$\delta_1^{(2)} = (1 - (x_1^{(2)})^2) \sum_{j=1}^3 \delta_j^{(3)} w_{1j}^{(3)}$$

Neural Network

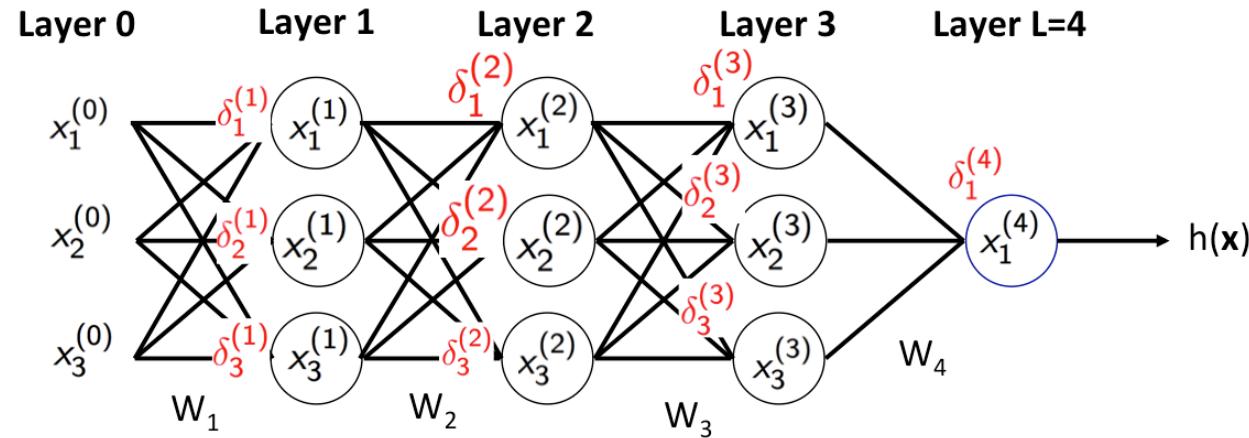
Backward propagation



$$\delta_2^{(2)} = (1 - (x_2^{(2)})^2) \sum_{j=1}^3 \delta_j^{(3)} w_{2j}^{(3)}$$

Neural Network

Backward propagation

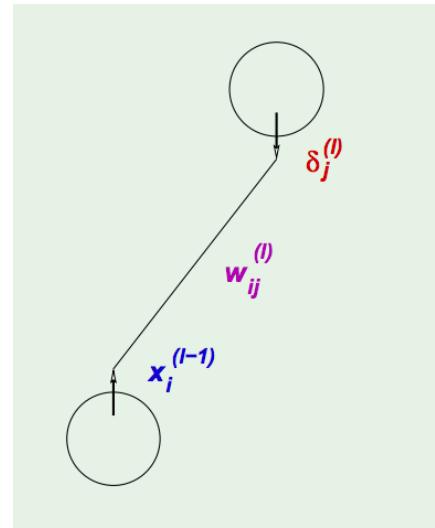


Neural Network

Backpropagation

SGD for neural networks

- Initialize all weights $w_{ij}^{(l)}$ at random
- For iter = 0, 1, 2, ...
 - Forward: Compute all $x_j^{(l)}$ from input to output
 - Backward: Compute all $\delta_j^{(l)}$ from output to input
 - Update all the weights $w_{ij}^l \leftarrow w_{ij}^{(l)} - \eta x_i^{(l-1)} \delta_j^{(l)}$



Neural Network

Backpropagation

- Just an automatic way to apply **chain rule** to compute gradient
- Auto-differentiation (AD) --- as long as we define derivative for **each basic** function, we can use AD to compute any of their compositions
- Implemented in most deep learning packages (e.g., pytorch, tensorflow)

Neural Network

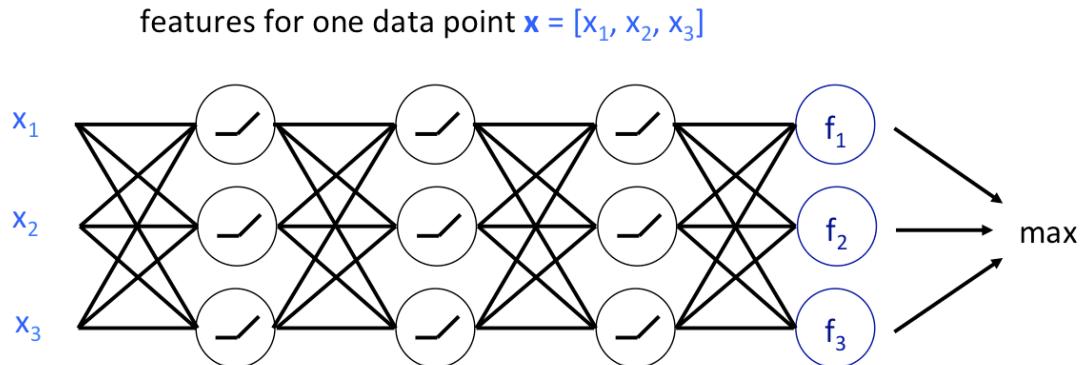
Backpropagation

- Just an automatic way to apply **chain rule** to compute gradient
- Auto-differentiation (AD) --- as long as we define derivative for **each basic** function, we can use AD to compute any of their compositions
- Implemented in most deep learning packages (e.g., pytorch, tensorflow)
- Auto-differentiation needs to store all the intermediate nodes of each sample
 - \Rightarrow Memory cost $>$ number of neurons \times batch size
 - \Rightarrow This poses a constraint on the batch size

Neural Network

Multiclass Classification

- K classes: K neurons in the final layer
- Output of each f_i is the score of class i
 - Taking $\arg \max_i f_i(x)$ as the prediction



Neural Network

Multiclass loss

- Softmax function: transform output to probability:

- $[f_1, \dots, f_K] \rightarrow [p_i, \dots, p_K]$

- Where $p_i = \frac{e^{f_i}}{\sum_{j=1}^K e^{f_j}}$

- Cross-entropy loss:

- $L = - \sum_{i=1}^K y_i \log(p_i)$

- Where y_i is the i -th label