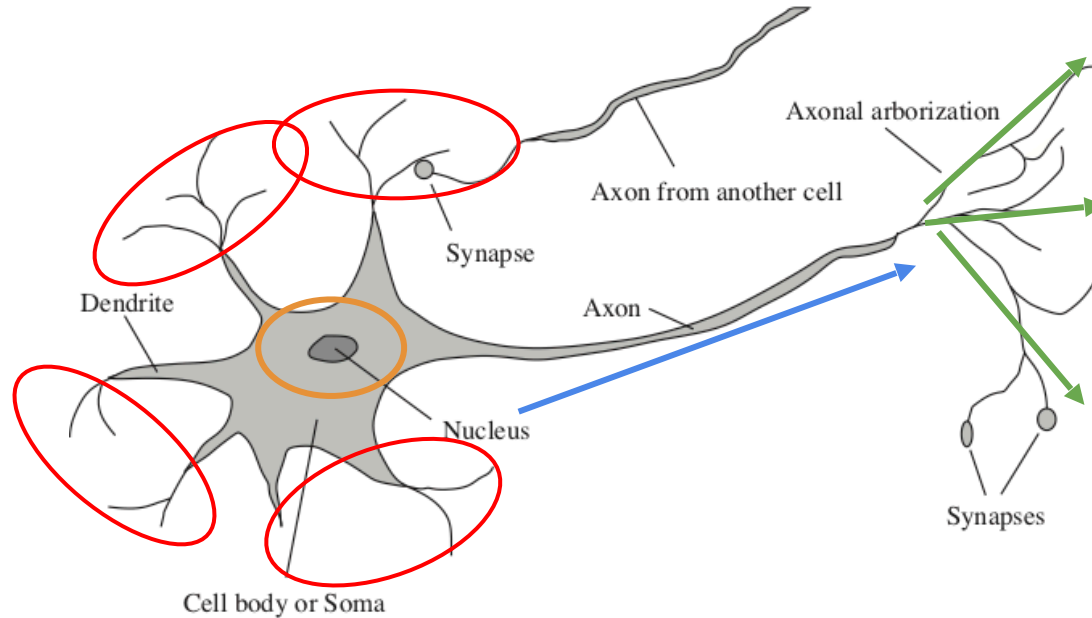


# Neural Networks

CS 480

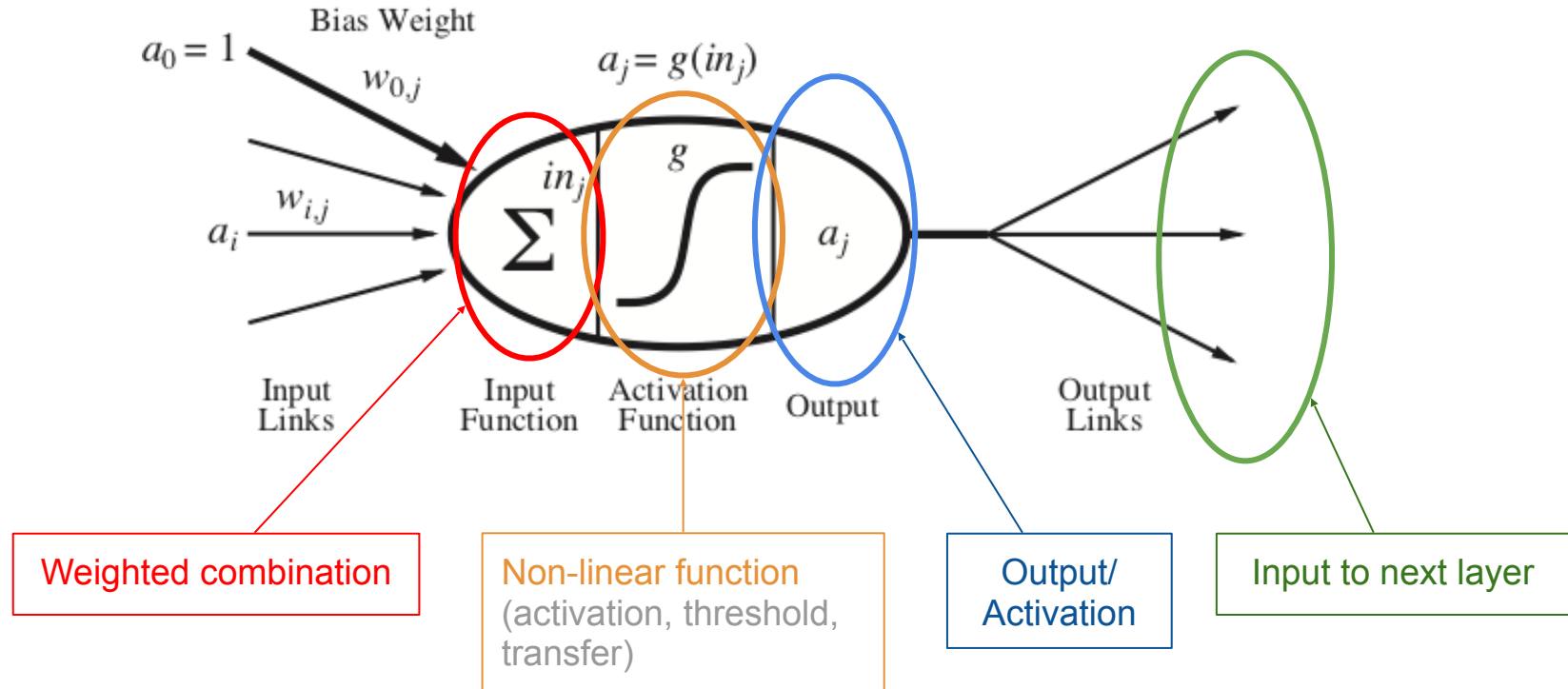
Intro to Artificial Intelligence

# A simplified diagram of a neuron

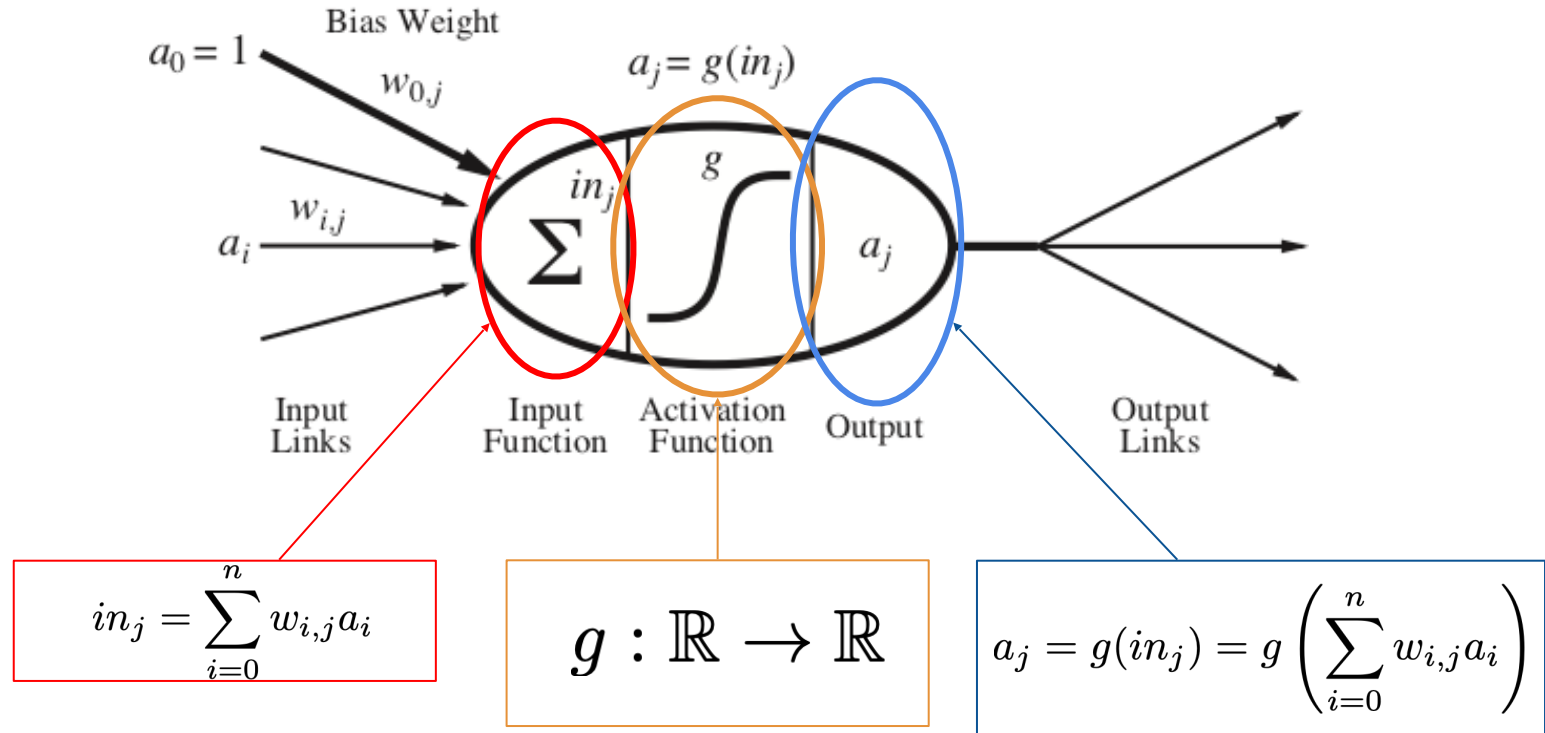


The neuron **fires** after the **input** exceeds some **threshold**, propagating signal to the **next layer** of neurons

# A computational “model” of a neuron (1)



## A computational “model” of a neuron (2)

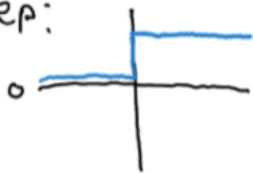


# Activation functions

- Introduce some non-linearity (otherwise, same as linear regression)
- Some popular choices:

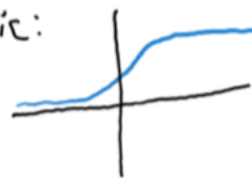
Logic  
circuits

Step:



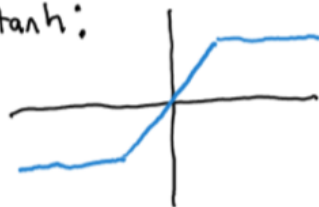
$$g(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{else} \end{cases}$$

logistic:



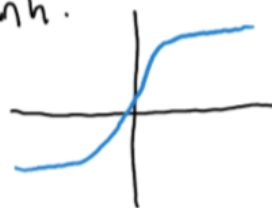
$$g(x) = \frac{1}{1 + e^{-x}}$$

hard tanh:



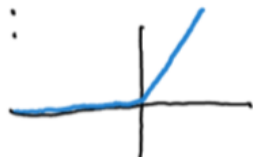
$$g(x) = \begin{cases} 1 & \text{if } x > 1 \\ x & \text{if } -1 < x < 1 \\ -1 & \text{if } x < -1 \end{cases}$$

tanh:



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

ReLU:



$$g(x) = \begin{cases} x & \text{if } x > 0 \\ 0 & \text{else} \end{cases}$$

Softplus



$$g(x) = \log(1 + e^x)$$

# As “logic circuits” (1)

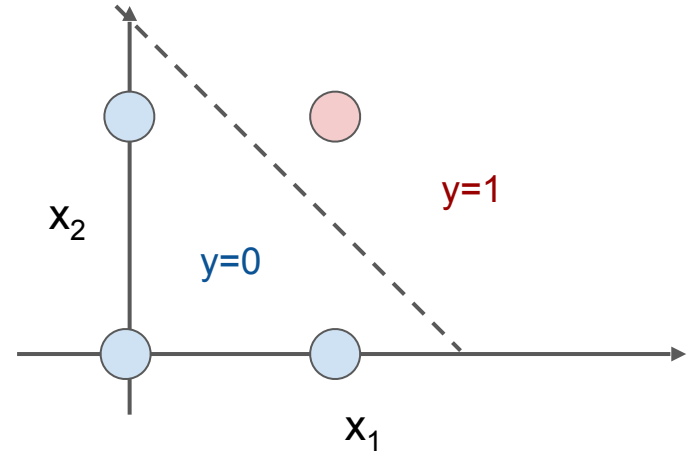
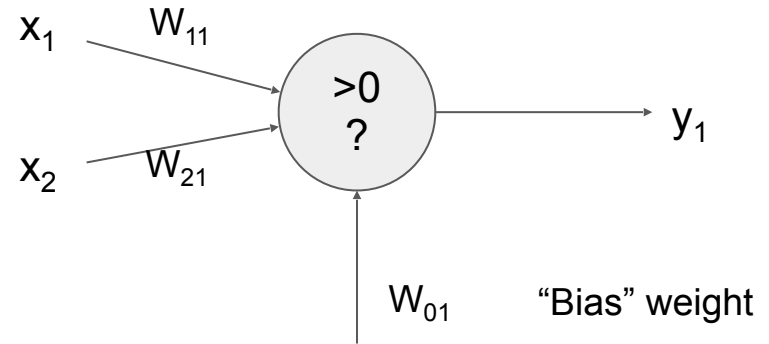
Let  $X=\{0,1\}^2$ ,  $Y=\{0,1\}$ ,  $g(a) = 1$  if  $a>0$  else 0

AND:

$$W_{01} = -0.5$$

$$W_{11} = 0.3$$

$$W_{21} = 0.3$$



## As “logic circuits” (2)

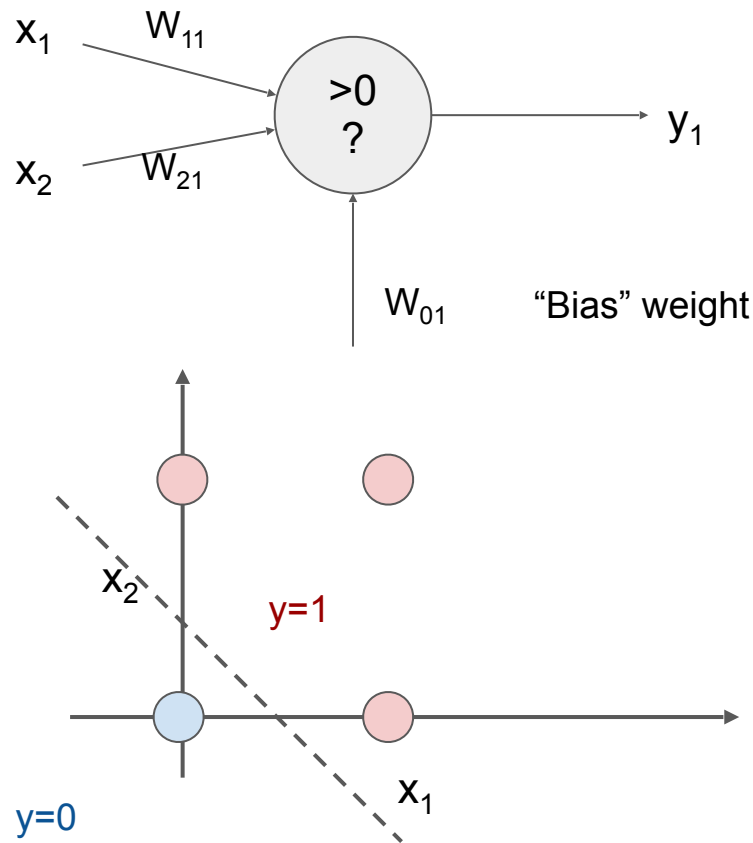
Let  $X=\{0,1\}^2$ ,  $Y=\{0,1\}$ ,  $g(a) = 1$  if  $a>0$  else 0

OR:

$$W_{01} = -0.3$$

$$W_{11} = 0.5$$

$$W_{21} = 0.5$$



# As “logic circuits” (3)

Let  $X=\{0,1\}^2$ ,  $Y=\{0,1\}$ ,  $g(a) = 1$  if  $a>0$  else 0

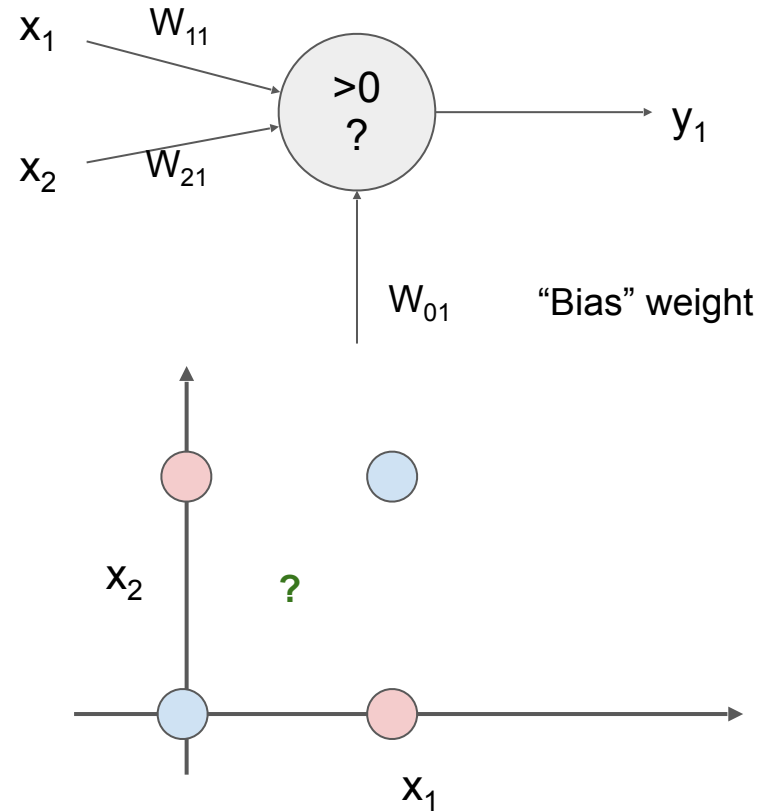
XOR:

$$W_{01} =$$

$$W_{11} =$$

$$W_{21} =$$

A single neuron is equivalent to picking a hyperplane.  
Can't correctly label data that is not **linearly separable**.





# As “logic circuits” (4)

Let  $X=\{0,1\}^2$ ,  $Y=\{0,1\}$ ,  $g(a) = 1$  if  $a>0$  else 0

XOR:

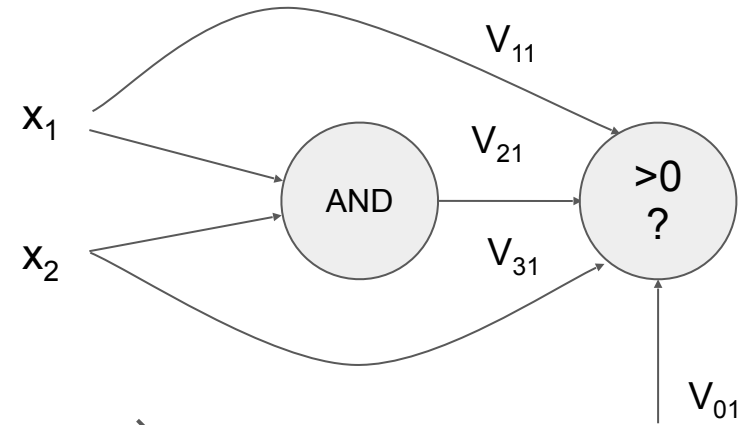
$$V_{01} = -0.25$$

$$V_{11} = 0.5$$

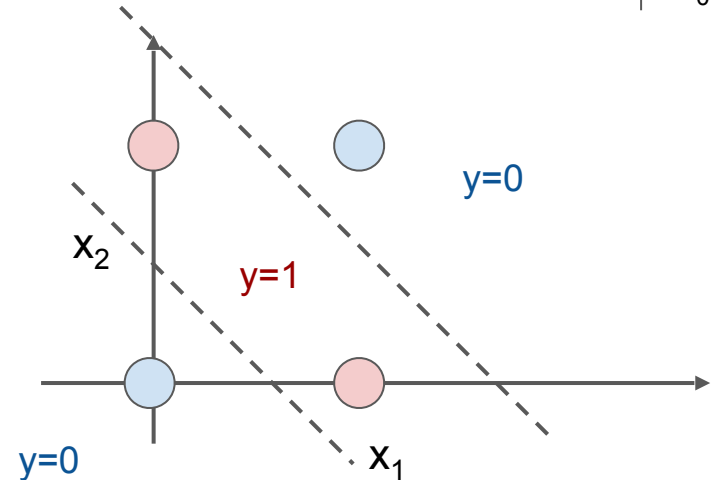
$$V_{21} = -1.25$$

$$V_{31} = 0.5$$

In general, we can build **any** logic circuit with a **network** of neurons, as long as we have enough units in each **layer**.



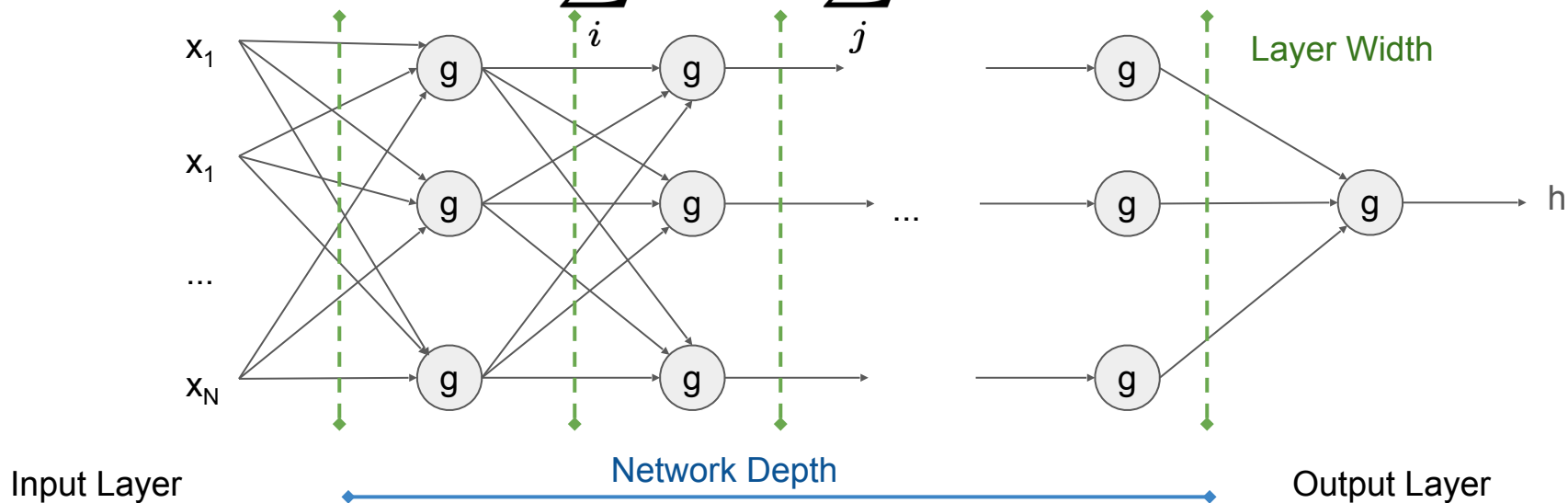
$y_1$



# Feed-forward networks

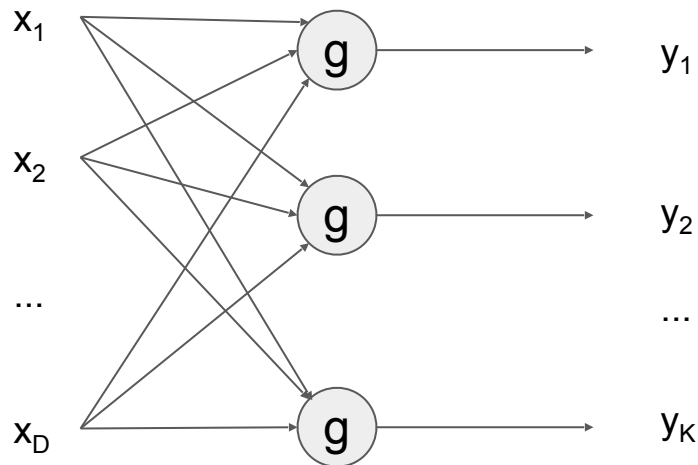
- A composition of multiple activation functions
- No “internal state”, just an input->output mapping

$$h(\mathbf{x}) = g\left(\sum \mathbf{w}^{(l)} \cdot g\left(\sum \mathbf{w}^{(l-1)} \cdot g(\dots)\right)\right)$$



# The perceptron

- A single “layer”, one unit per output



## Simplified problem domain

Let

$$X = \{0, 1\}^D$$

$$Y = \{0, 1\}$$

$$g(a) = 1 \text{ if } a > 0 \text{ else } 0$$

How can we learn the weights?

# The perceptron learning rule

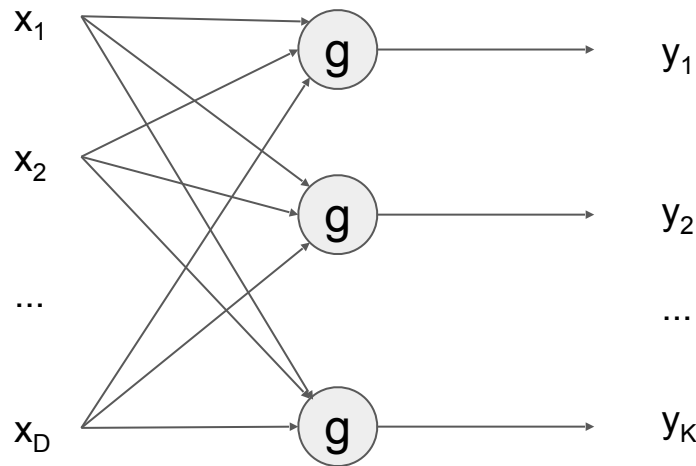
Loop:

Pick  $(\mathbf{x}^{(i)}, y^{(i)})$  from the training data

Foreach  $w_{j,k}$ :

$$w_{j,k} \leftarrow w_{j,k} + \alpha \cdot (y_k^{(i)} - h(\mathbf{x}^{(i)})) \cdot x_j^{(i)}$$

If the weights stop changing, done!



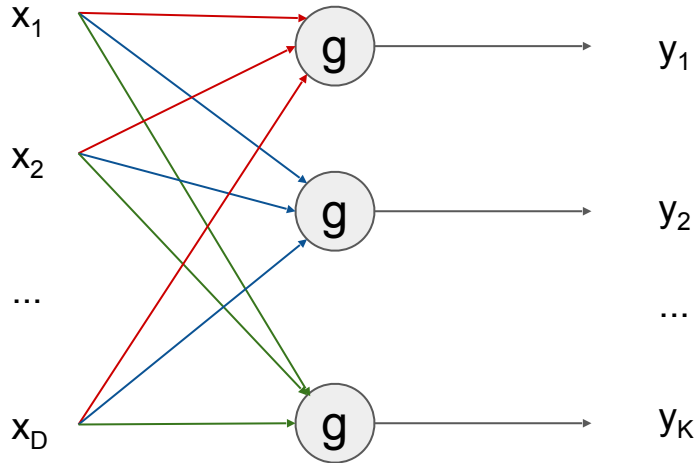
$$w_{j,k} \leftarrow w_{j,k} + \alpha \cdot (y_k^{(i)} - h(\mathbf{x}^{(i)})) \cdot x_j^{(i)}$$

# The perceptron learning rule - notes

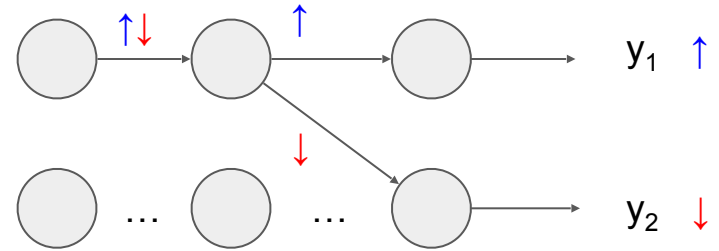
- If the predicted output and the actual output agree, weight doesn't change  
(  $(y-h(x)) = 0 \Rightarrow$  don't change weight )
- If predicted output is 0 and actual output is 1, weight wasn't large enough to get past threshold: increase weight  
(  $(y-h(x)) > 0 \Rightarrow$  increase weight )
- If predicted output is 1 and actual output is 0, weight was too large, decrease  
(  $(y-h(x)) < 0 \Rightarrow$  decrease weight )
- Learning rate  $\alpha$  controls how much to change weight based on a single example (if training data contains noise, make small changes)
- If dataset is **linearly separable**, the perceptron rule is guaranteed to fit the training data in a finite number of steps

# More complex networks

- Notice, that for a single layer, the weights for different outputs do not interact: training can happen in parallel



- For different activation functions, use Stochastic Gradient Descent
- For multiple layers, weights do interact...



- New algorithm: **Backprop**

# Stochastic Gradient Descent for NNs

Like Gradient Descent, except we only use a single point instead of an entire dataset

Loop:

Pick  $(\mathbf{x}^{(i)}, y^{(i)})$  from the training data

Foreach  $w_{j,k}$ :

$$w_{j,k} \leftarrow w_{j,k} + (-1)\alpha \frac{\partial}{\partial w_{j,k}} E(w, \mathbf{x}^{(i)}, y^{(i)})$$

If the weights stop changing, done!

Where

$$E(w, \mathbf{x}, y) = \frac{1}{2}(y - h(\mathbf{x}))^2$$

Notes:

- We want to move to a **minimum** of the error, so we move **down** the gradient with the  $(-1)$
- Gradient Descent, like hill-climbing can get stuck in **local optima**. Empirically, Stochastic Gradient Descent seems to be able to avoid getting stuck.

# Gradient of the error function, single layer (1)

$$\frac{\partial E}{\partial w_{j,k}} = \frac{\partial}{\partial w_{j,k}} \frac{1}{2} (y_k - h(\mathbf{x}))^2$$

Chain rule

$$= (y_k - h(\mathbf{x})) \frac{\partial}{\partial w_{j,k}} (y_k - h(\mathbf{x}))$$

Single layer

$$= (y_k - h(\mathbf{x})) \frac{\partial}{\partial w_{j,k}} (y_k - a_k)$$

Definition of  $a_k$

$$= (y_k - h(\mathbf{x})) \frac{\partial}{\partial w_{j,k}} (y_k - g(in_k))$$



## Gradient of the error function, single layer (2)

$$\frac{\partial E}{\partial w_{j,k}} = (y_k - h(\mathbf{x})) \frac{\partial}{\partial w_{j,k}} (y_k - g(in_k))$$

Chain rule

$$= (y_k - h(\mathbf{x}))(-1)g'(in_k) \frac{\partial}{\partial w_{j,k}} in_k$$

$g'()$  = deriv. of  $g()$

Definition of  $in_k$

$$= (y_k - h(\mathbf{x}))(-1)g'(in_k) \frac{\partial}{\partial w_{j,k}} \sum_i w_{i,k} \cdot x_i$$

$$= (y_k - h(\mathbf{x}))(-1)g'(in_k) \cdot x_j$$

# Comparing Perceptron rule with SGD (single layer)

## Perceptron rule

$$w_{j,k} \leftarrow w_{j,k} + \alpha \cdot (y_k^{(i)} - h(\mathbf{x}^{(i)})) \cdot x_j^{(i)}$$

- Guaranteed global convergence (linearly separable data)
- Requires step threshold function
- Perceptrons (single layers) only

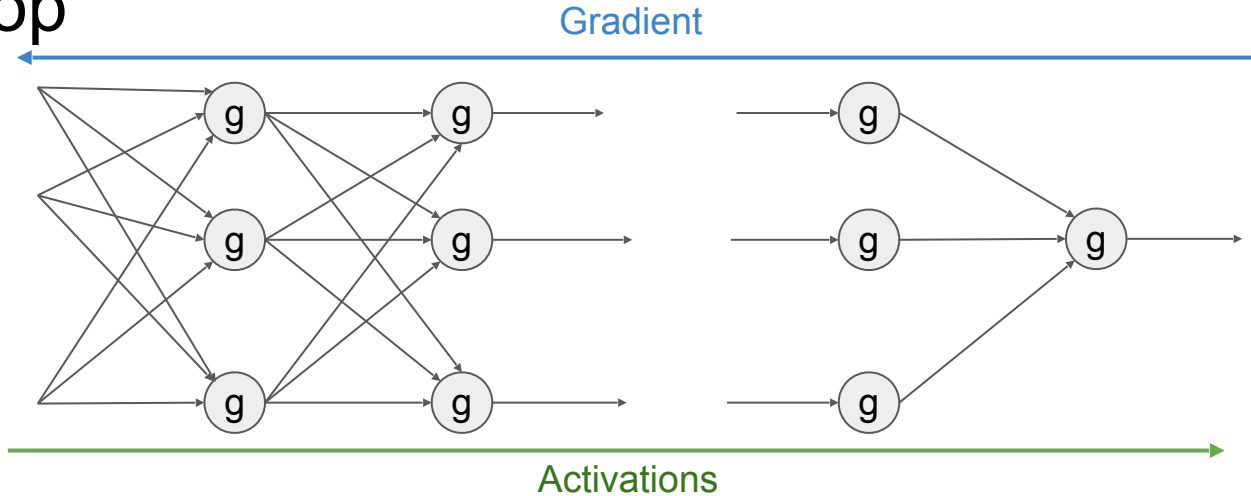
## SGD (single layer)

$$w_{j,k} \leftarrow w_{j,k} + \alpha \cdot (y_k^{(i)} - a_k) g'(in_k) x_j^{(i)}$$

- Probabilistic convergence in the limit
- Works with any  $g()$  that has a derivative
- Generalizes to more than one layer

How can we generalize this to more than one layer?

# Backprop



The derivative of the error function for a weight can be written in terms of

- The derivative of the threshold function ( $g'$ )
- The input from the previous layer (activations)
- The error from the following layer (error)

# Weight updates for multilayer networks

$$w_{j,k} \leftarrow w_{j,k} + \alpha \cdot a_j \cdot \Delta_k$$

where for the output layer

$$\Delta_k = (y_k - a_k) \cdot g'(in_k)$$

and for hidden layers

$$\Delta_j = g'(in_j) \cdot \sum_k (w_{j,k} \cdot \Delta_k)$$

Delta from following layer

Weight from j to k

# Summary and preview

## Wrapping up

- Perceptrons: easy to train for linearly separable functions
- Stochastic Gradient Descent: a variant of GD using a single training datapoint at a time
- Backprop for training multi-layer feed-forward neural networks, has two stages layer by layer:
  - Feed signal forward, store activations
  - Propagate error backwards, update weights

Next time: Deep Learning