

NEURAL NETWORK

SML

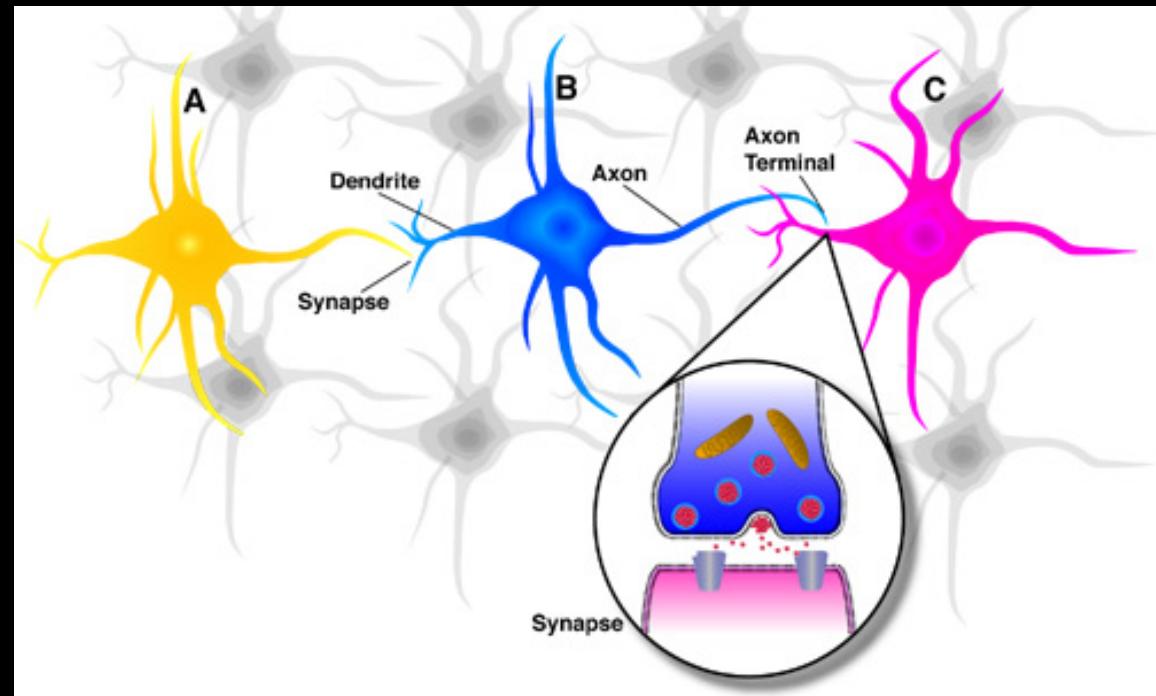
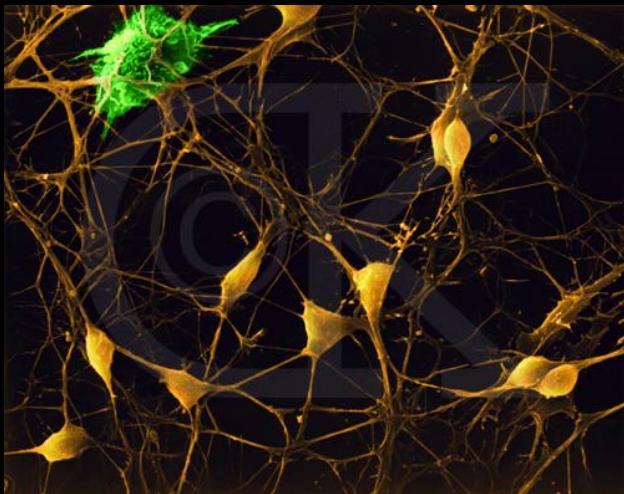
WHAT IS NEURAL NETWORK?

- Models of the **brain** and **nervous system**
- Highly parallel machine - process information much more like the brain than a serial computer
- Very simple principles
- Very complex behaviors

HISTORY

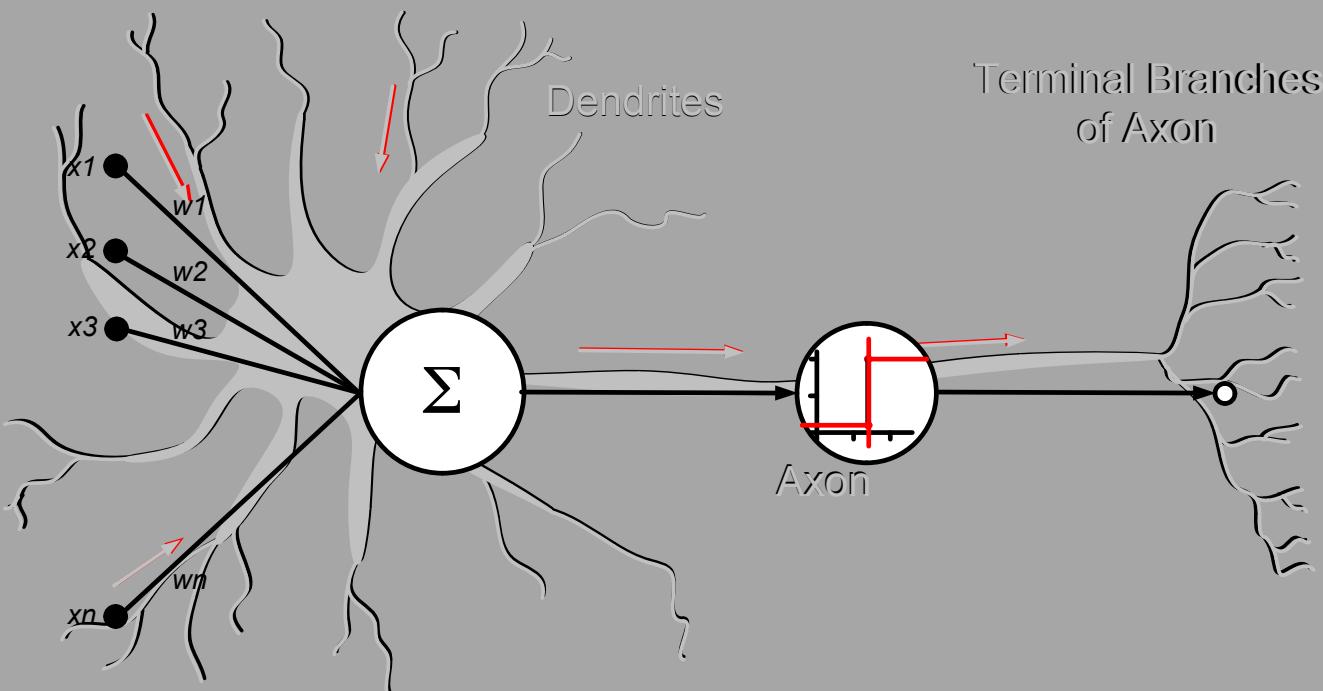
- McCulloch and Pitts introduced the Perceptron in 1943.
- Simplified model of a biological neuron
- Fell out of favor in the late 1960's
- (Minsky and Papert)
- Perceptron limitations
- Resurgence in the mid 1980's
- Nonlinear Neuron Functions
- Back-propagation training

NEURONS IN THE BRAIN



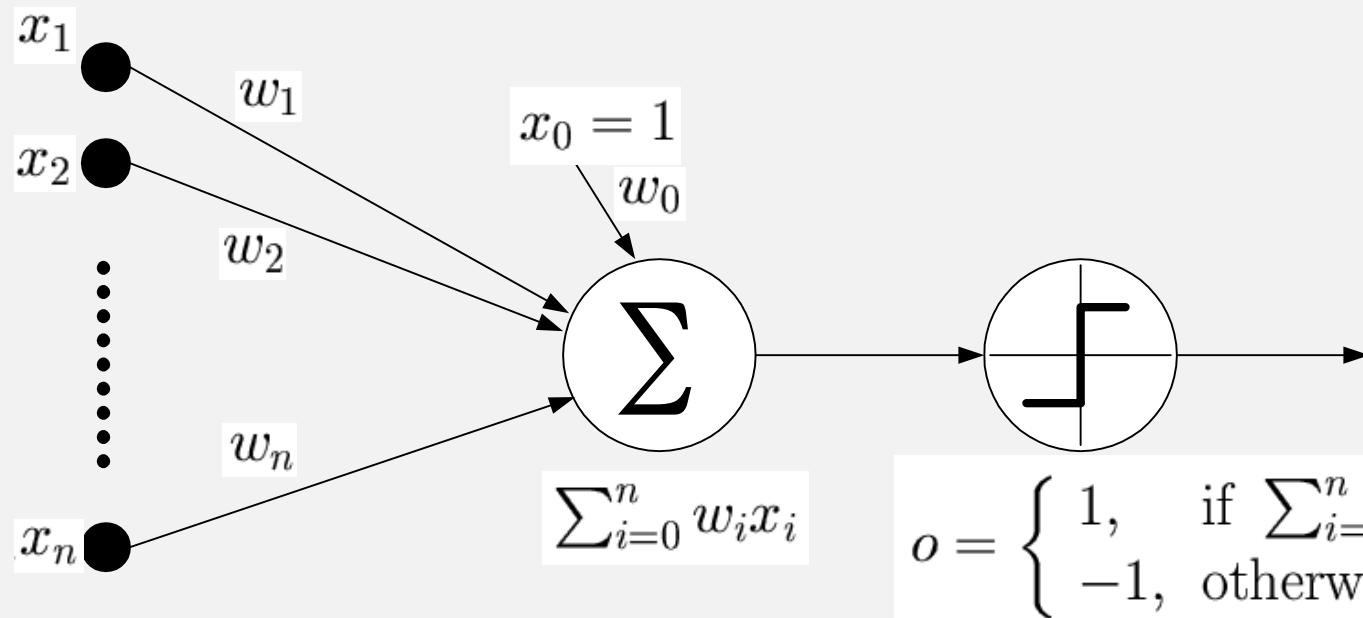
A SIMPLE ARTIFICIAL NEURON

- Perceptron

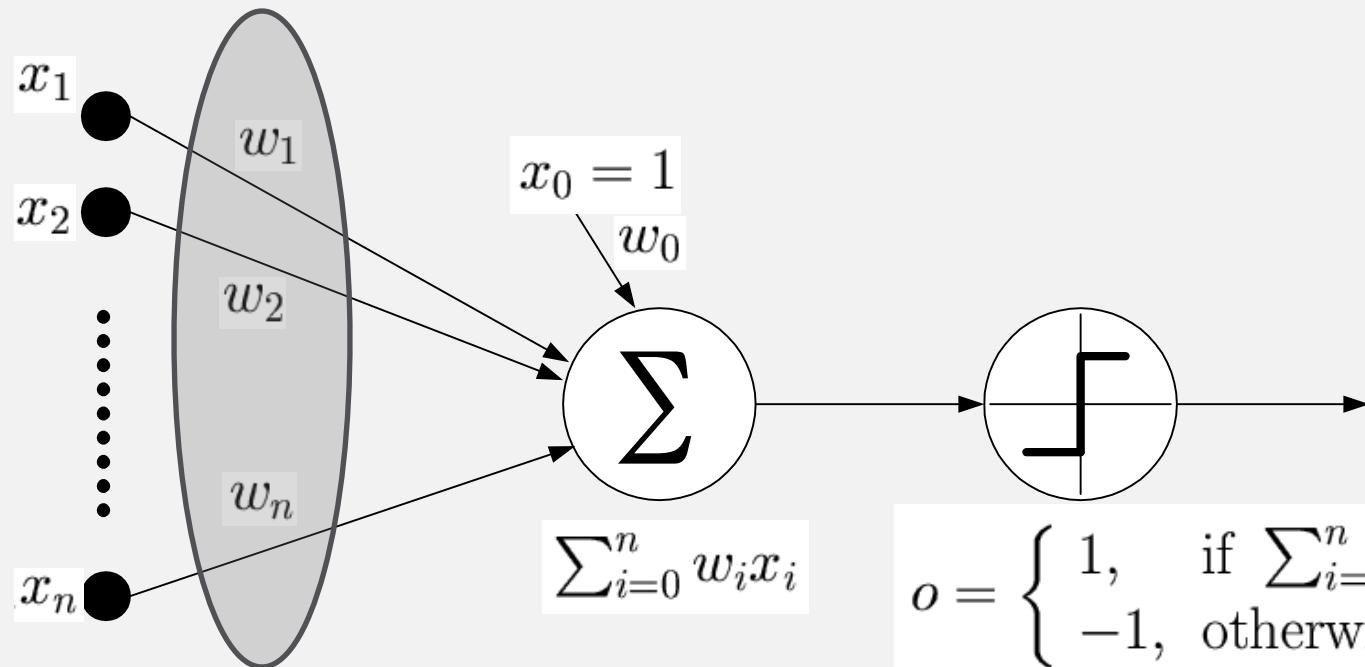


PERCEPTRON

- The neuron calculates a weighted sum of inputs and compares it to a threshold. If the sum is higher than the threshold, the output is set to 1, otherwise to -1.



WHAT IS HERE TO LEARN?



A MOTIVATING EXAMPLE

Each day you get lunch at the cafeteria.

Your diet consists of veg, chips, and coffee.

You get several portions of each

The cashier only tells you the total price of the meal

After several days, you should be able to figure out the price of each portion.

Each meal price gives a linear constraint on the prices of the portions:

$$price = x_{veg} w_{veg} + x_{chips} w_{chips} + x_{coffee} w_{coffee}$$

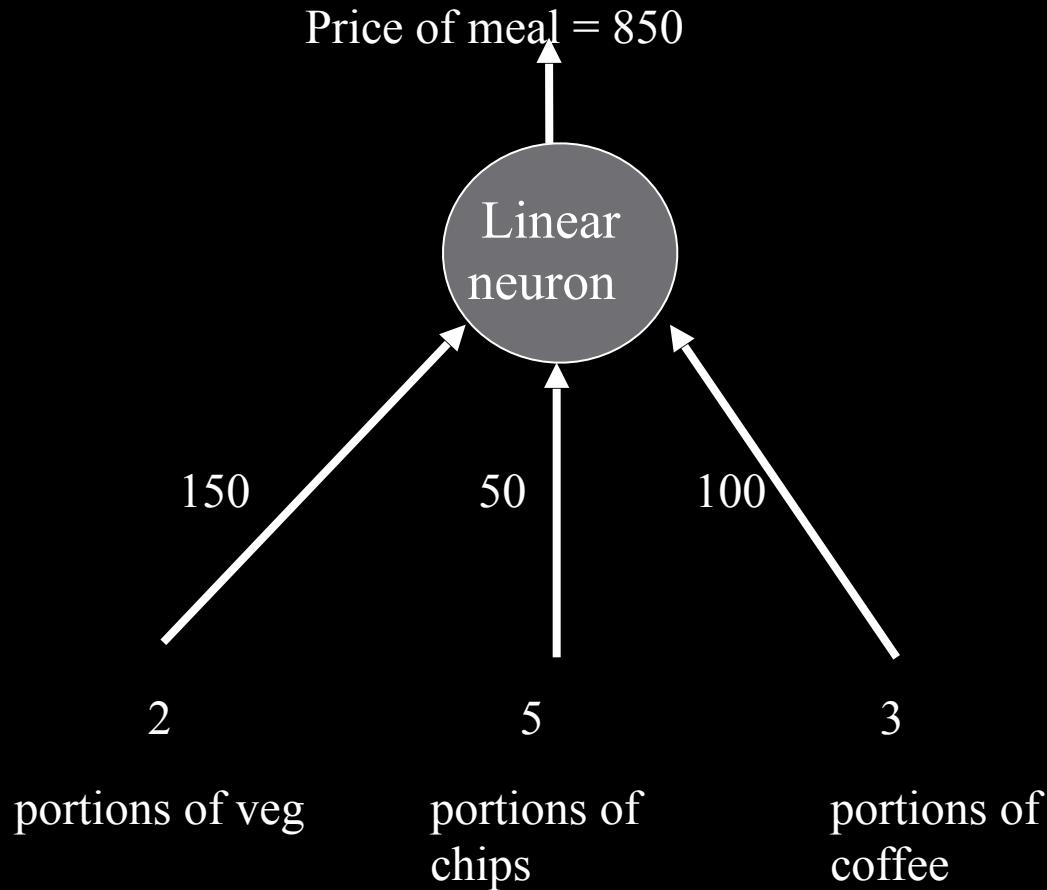
NNET WAY OF SOLVING THIS PROBLEM

The prices of the portions are like the weights in a linear neuron.

$$\mathbf{w} = (w_{veg}, w_{chips}, w_{coffee})$$

We will start with guesses for the weights and then adjust the guesses to give a better fit to the prices given by the cashier.

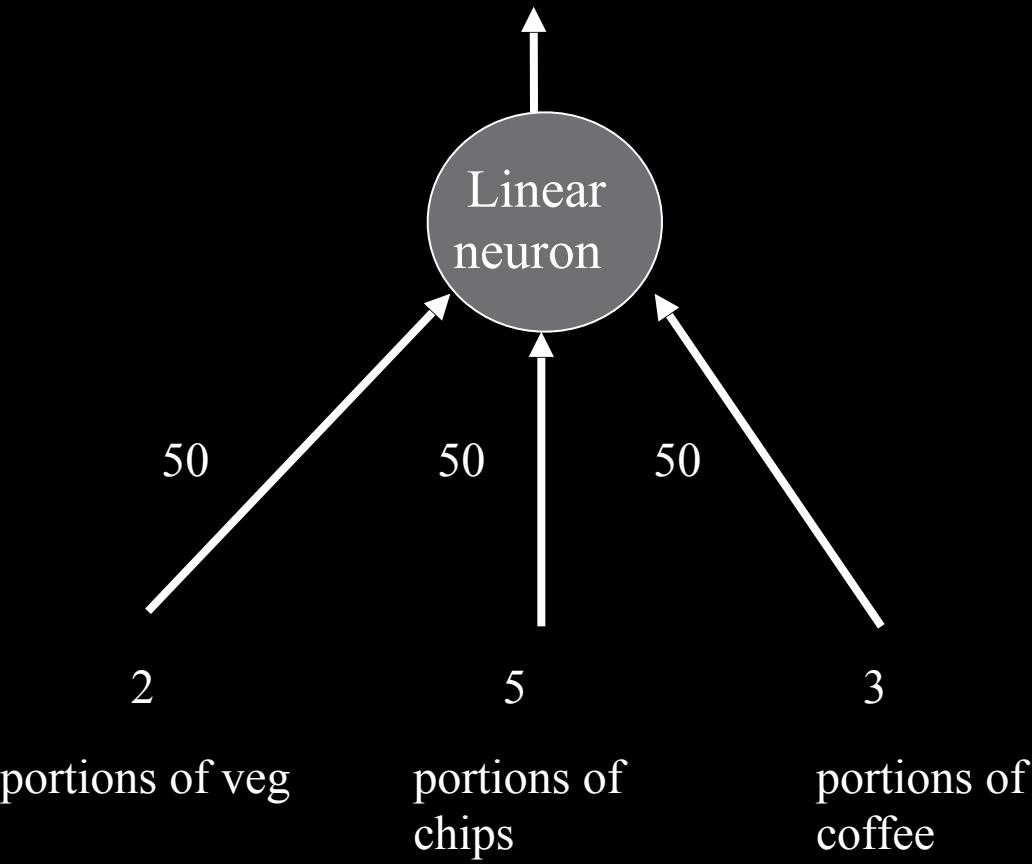
THE CASHIER'S BRAIN



A MODEL OF THE CASHIER'S BRAIN WITH ARBITRARY INITIAL WEIGHTS

Price of meal = 500

Residual error = 350

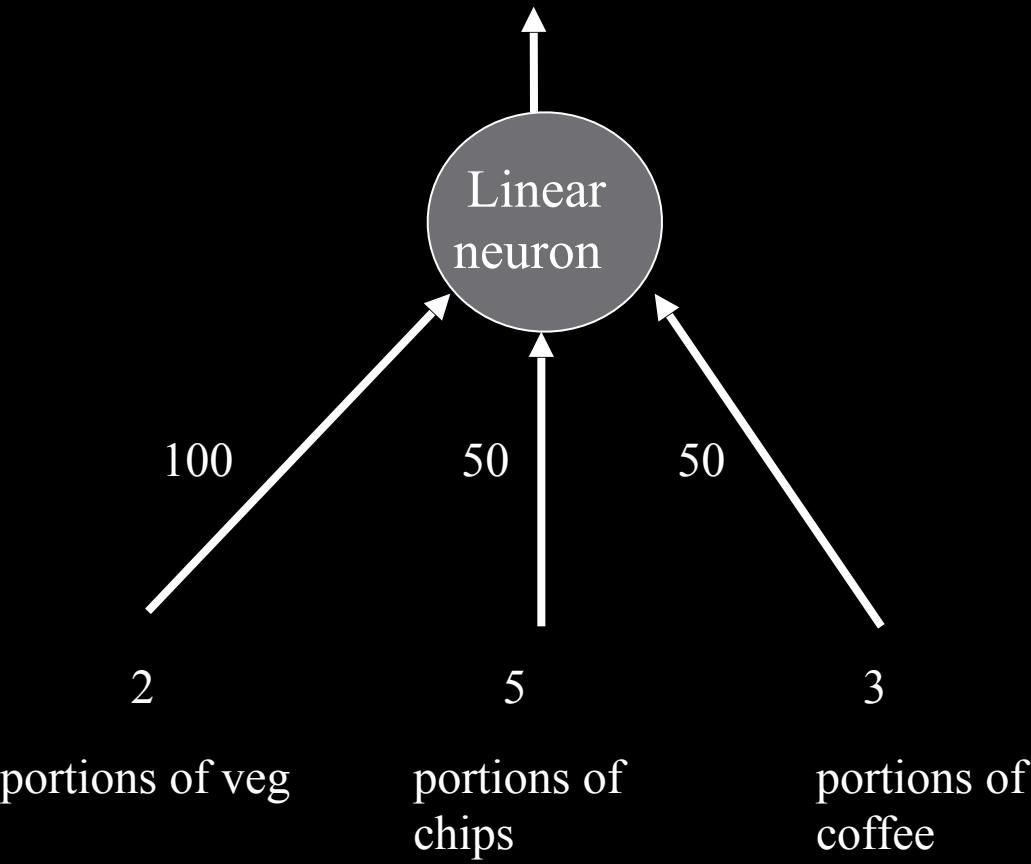


We can go back and start
adjusting the weights so that
residual error is reduced

A MODEL OF THE CASHIER'S BRAIN WITH ARBITRARY INITIAL WEIGHTS

Price of meal = 600

Residual error = 250

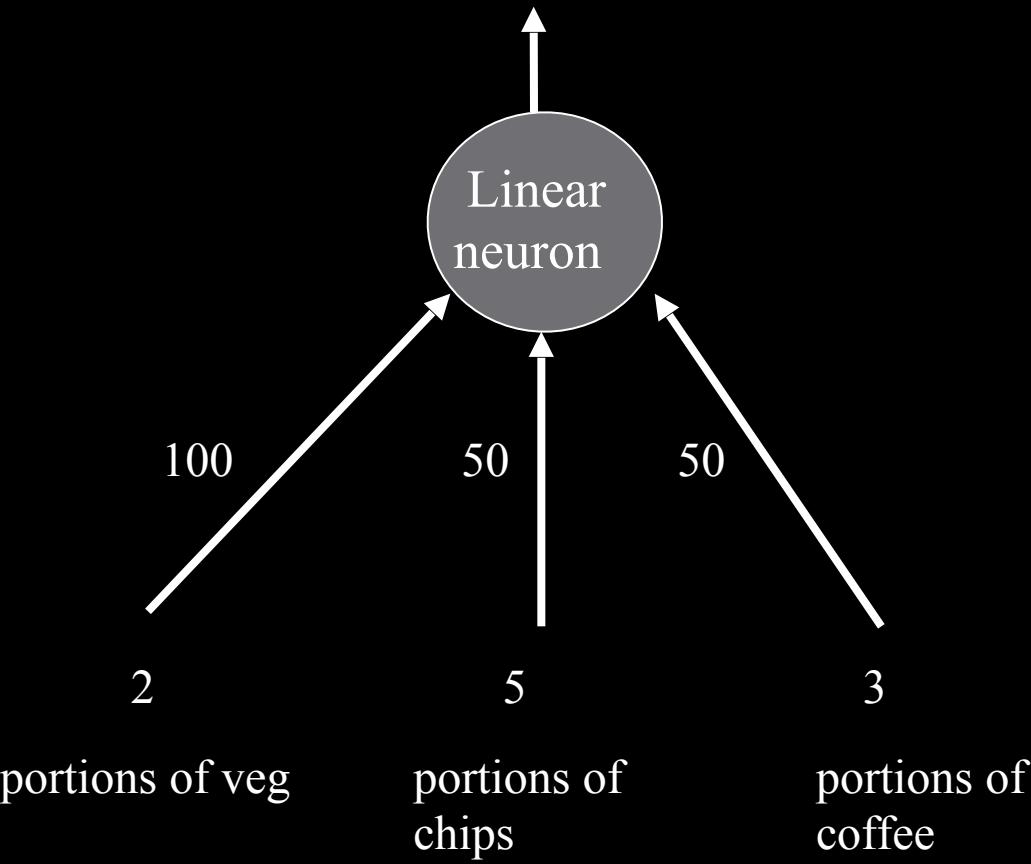


We can go back and start
adjusting the weights so that
residual error is reduced

A MODEL OF THE CASHIER'S BRAIN WITH ARBITRARY INITIAL WEIGHTS

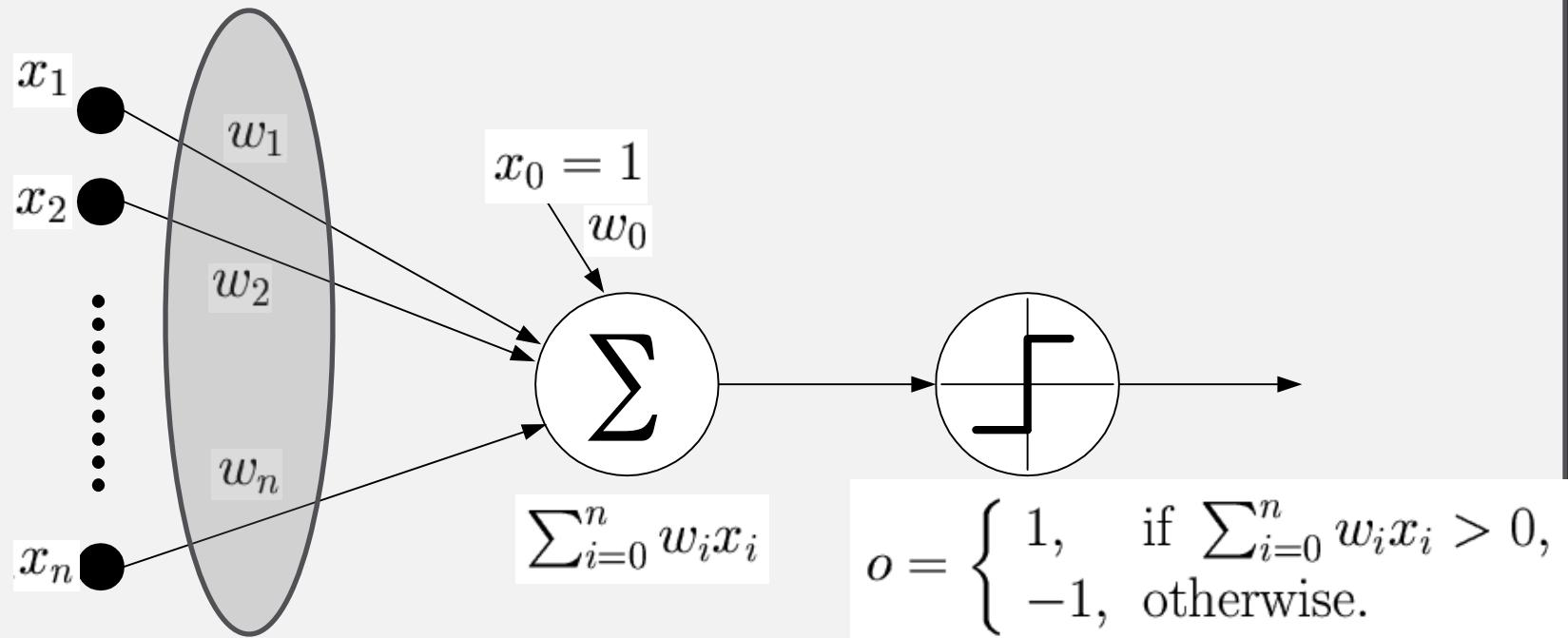
Price of meal = 600

Residual error = 250



We can go back and start
adjusting the weights so that
residual error is reduced

WHAT IS HERE TO LEARN?

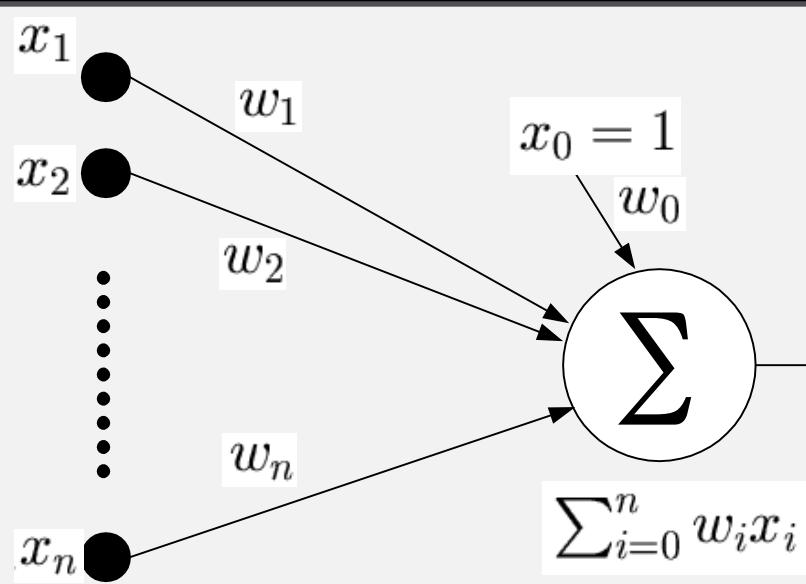


The Perceptron Training Rule

GRADIENT DESCENT AND DELTA RULE

- The delta training rule is best understood by considering the task of training an *unthresholded* perceptron; that is, a *linear unit* for which the output o is given by

$$o(\vec{x}) = \vec{w} \cdot \vec{x}$$



GRADIENT DESCENT AND DELTA RULE

In order to derive a weight learning rule for linear units, let us begin by specifying a measure for the training error of a hypothesis (weight vector), relative to the training examples.

$$E(\vec{w}) \equiv \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2$$

DERIVATION GDR

The vector derivative is called the *gradient* of E with respect to \vec{w}

$$\nabla E(\vec{w}) \equiv \left[\frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, \dots, \frac{\partial E}{\partial w_n} \right]$$

The gradient specifies the direction that produces the steepest increase in E . The negative of this vector therefore gives the direction of steepest decrease. The training rule for gradient descent is

$$\vec{w} \leftarrow \vec{w} + \Delta \vec{w}$$

where

$$\Delta \vec{w} = -\eta \nabla E(\vec{w})$$

DERIVATION OF GDR ...

The negative sign is presented because we want to move the weight vector in the direction that *decreases* E . This training rule can also written in its component form

$$w_i \leftarrow w_i + \Delta w_i$$

where

$$\Delta w_i = -\eta \frac{\partial E}{\partial w_i}$$

which makes it clear that steepest descent is achieved by altering each component w_i of \vec{w} in proportion to $\frac{\partial E}{\partial w_i}$.

DERIVATION OF GDR ...

The vector of $\frac{\partial E}{\partial w_i}$ derivatives that form the gradient can be obtained by differentiating E

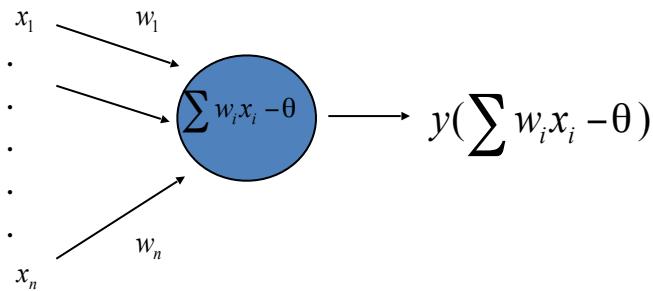
$$\begin{aligned}\frac{\partial E}{\partial w_i} &= \frac{\partial}{\partial w_i} \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2 \\ &= \frac{1}{2} \sum_{d \in D} \frac{\partial}{\partial w_i} (t_d - o_d)^2 \\ &= \frac{1}{2} \sum_{d \in D} 2(t_d - o_d) \frac{\partial}{\partial w_i} (t_d - o_d) \\ &= \sum_{d \in D} (t_d - o_d) \frac{\partial}{\partial w_i} (t_d - \vec{w} \cdot \vec{x}_d) \\ \frac{\partial E}{\partial w_i} &= \sum_{d \in D} (t_d - o_d) (-x_{id})\end{aligned}$$

The weight update rule for standard gradient descent can be summarized as

$$\Delta w_i = -\eta \frac{\partial E}{\partial w_i}$$

$$\Delta w_i = \eta \sum_{d \in D} (t_d - o_d) x_{id}$$

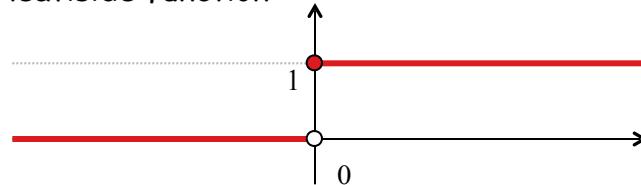
Activating function



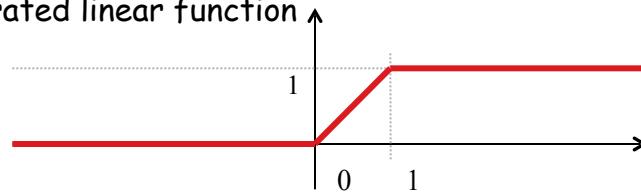
$$y(\xi) = \frac{1}{1 + e^{-\xi}}$$

$$y(\xi) = \frac{1 - e^{-\xi}}{1 + e^{-\xi}}$$

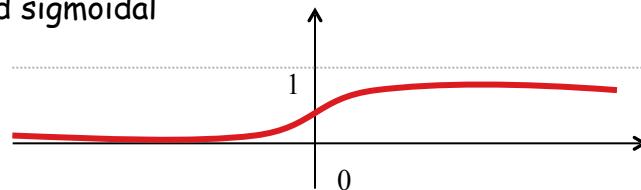
Heaviside function



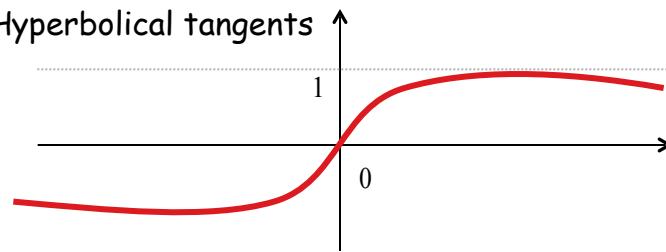
Saturated linear function



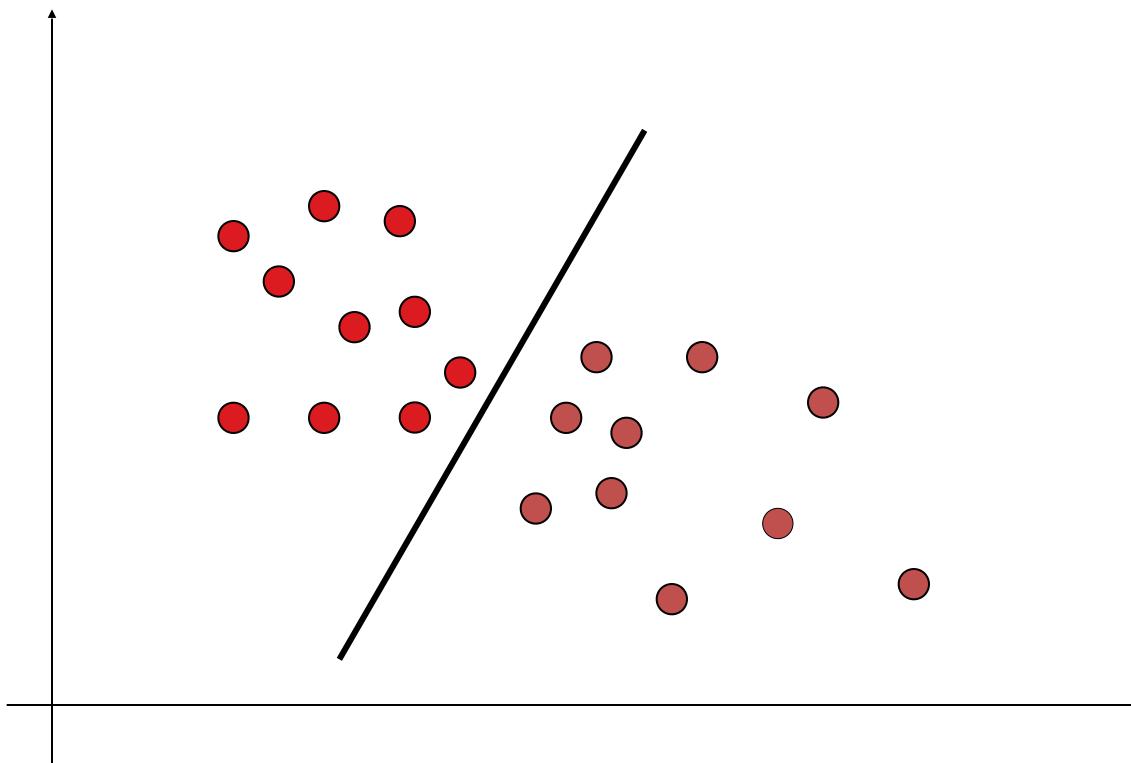
Standard sigmoidal function



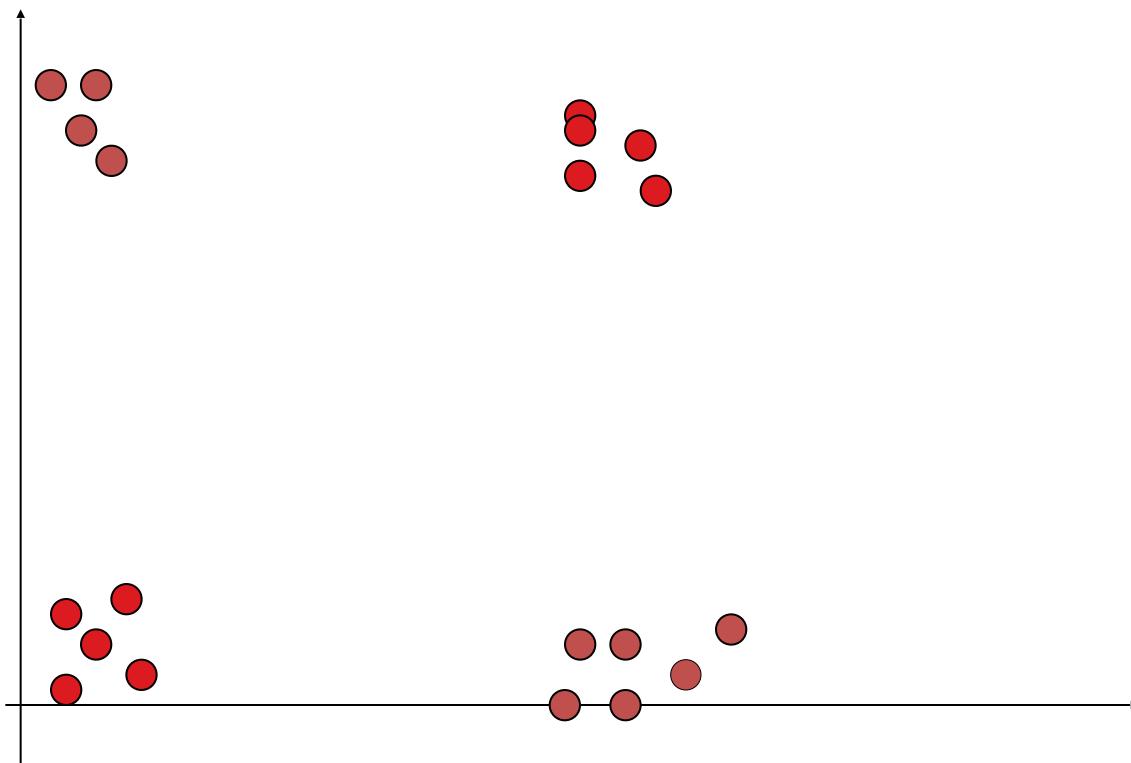
Hyperbolical tangents



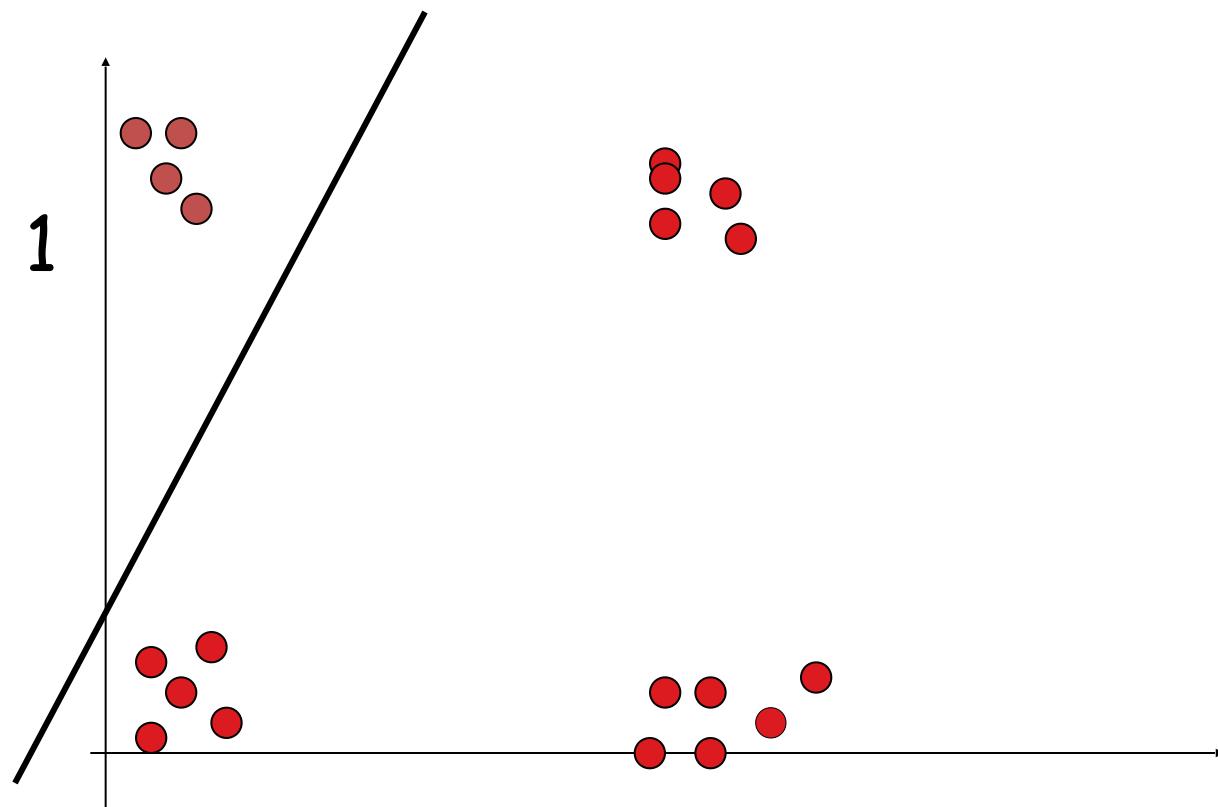
AB problem



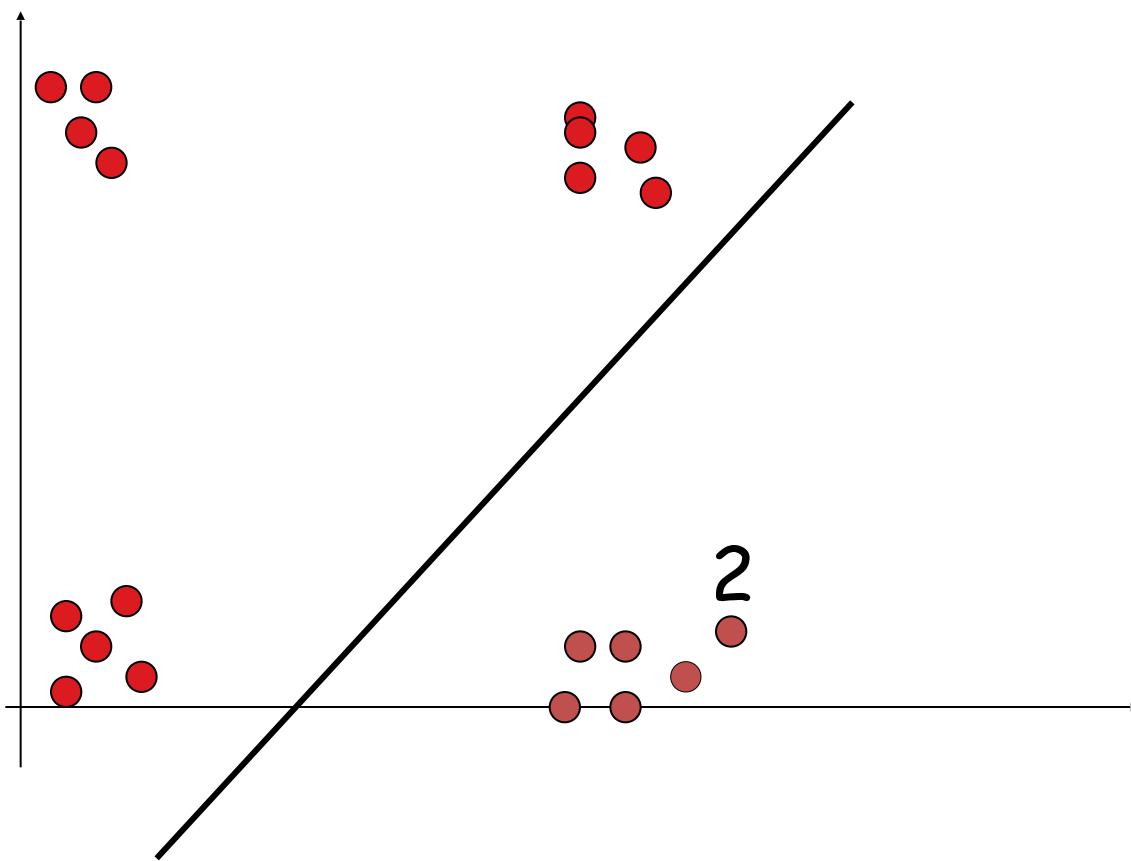
XOR problem



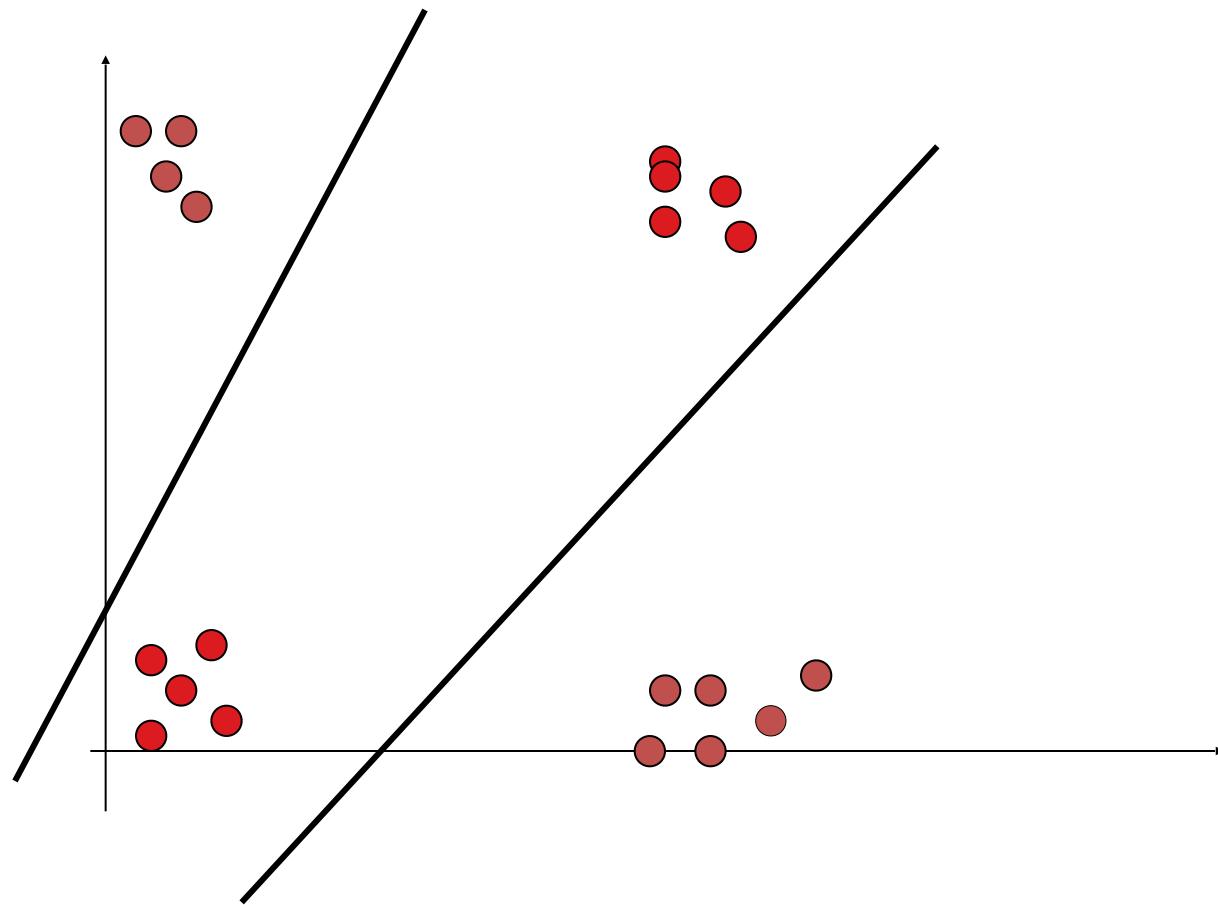
XOR problem



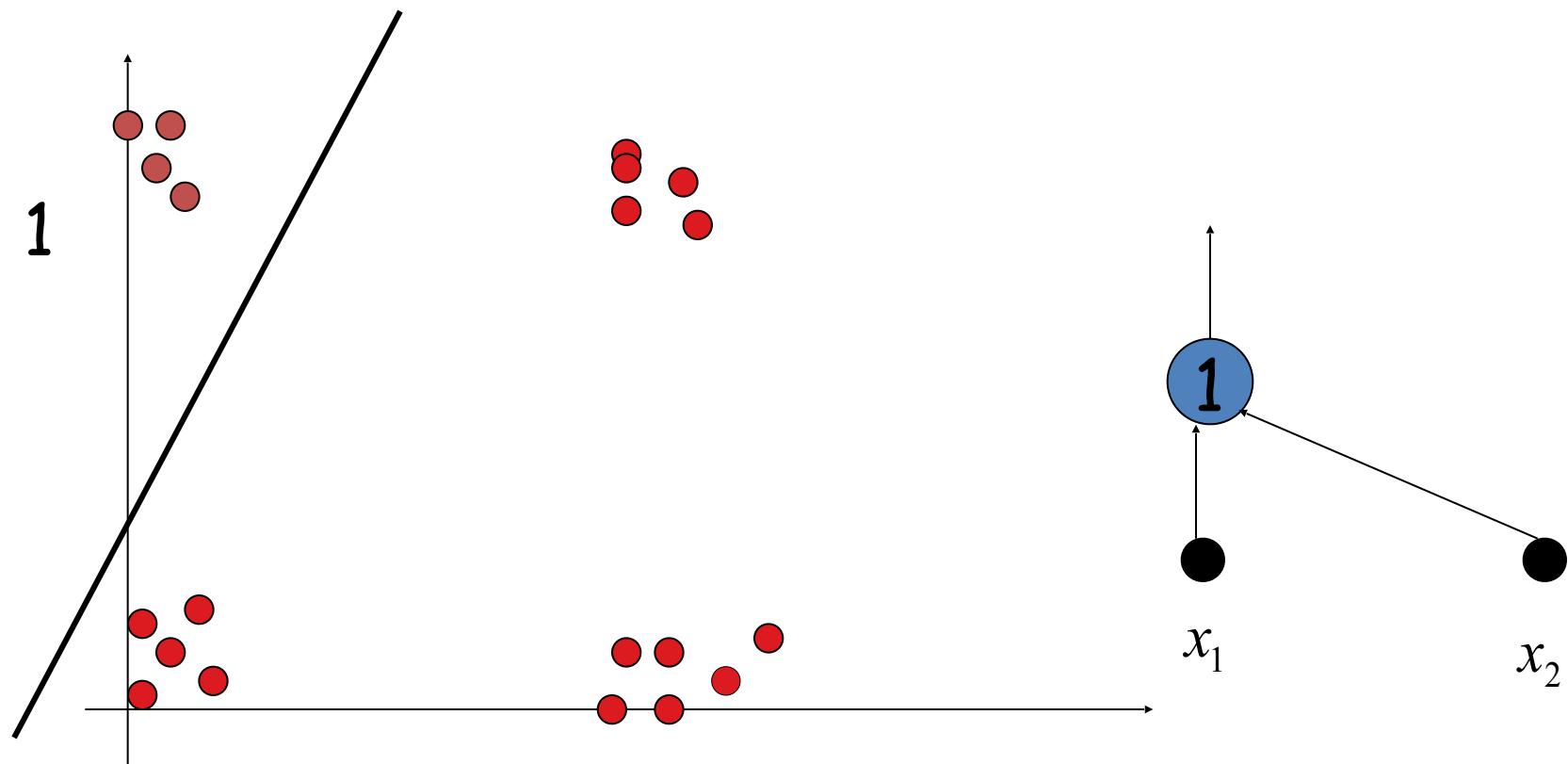
XOR problem



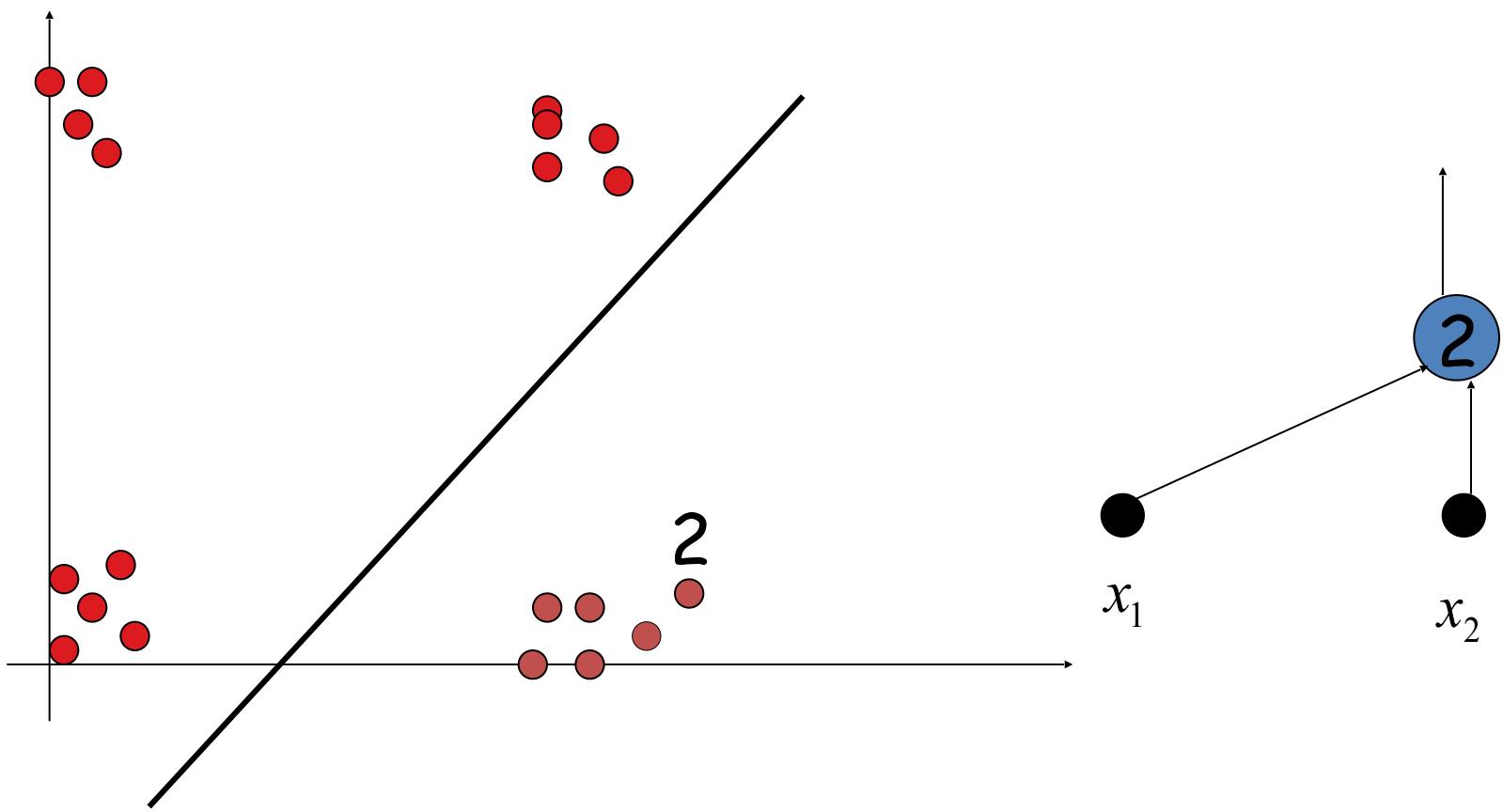
XOR problem



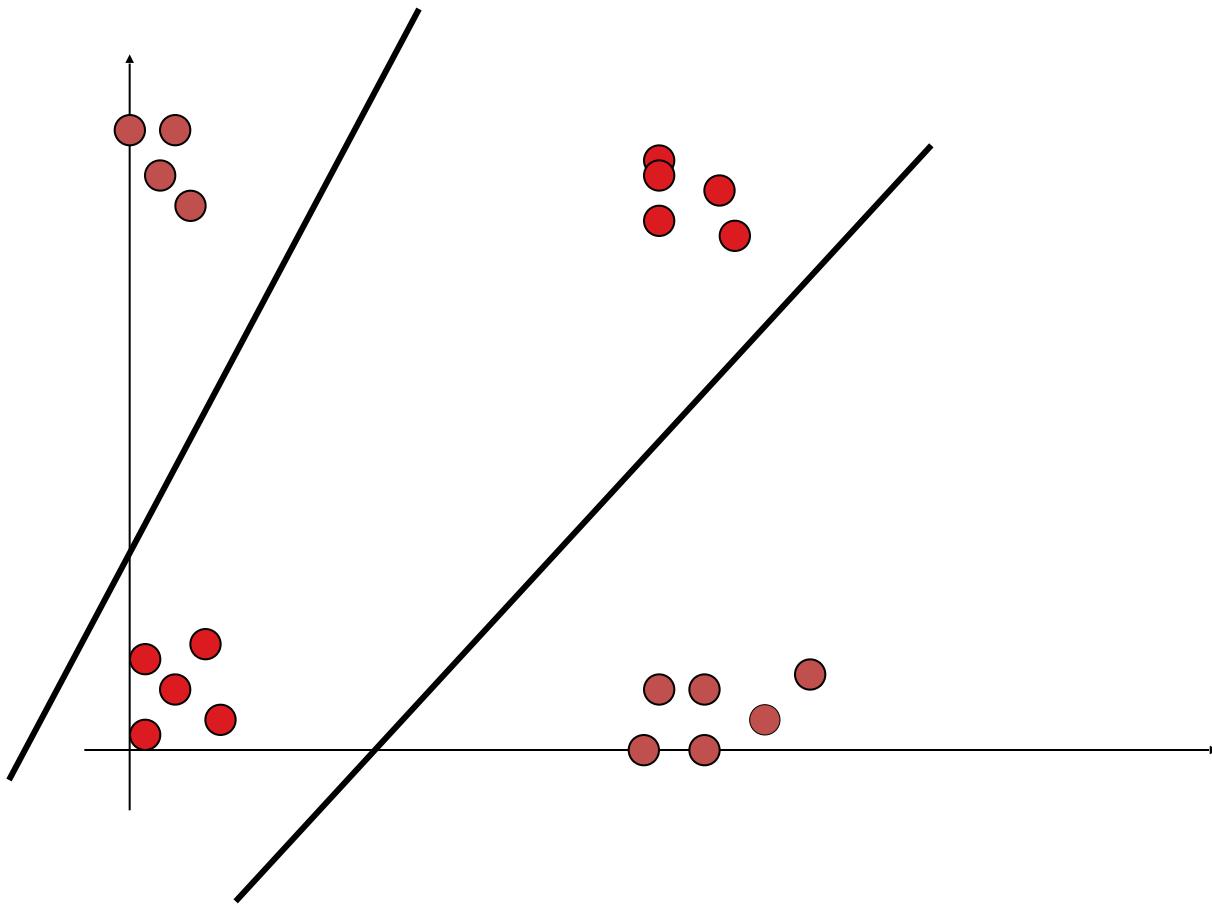
XOR problem



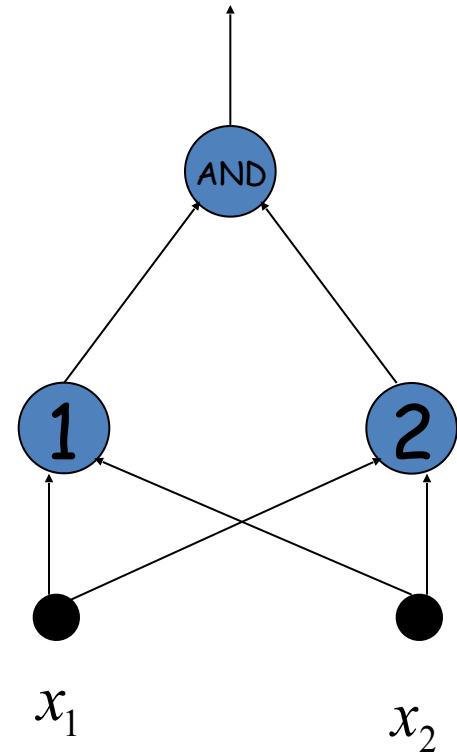
XOR problem



XOR problem

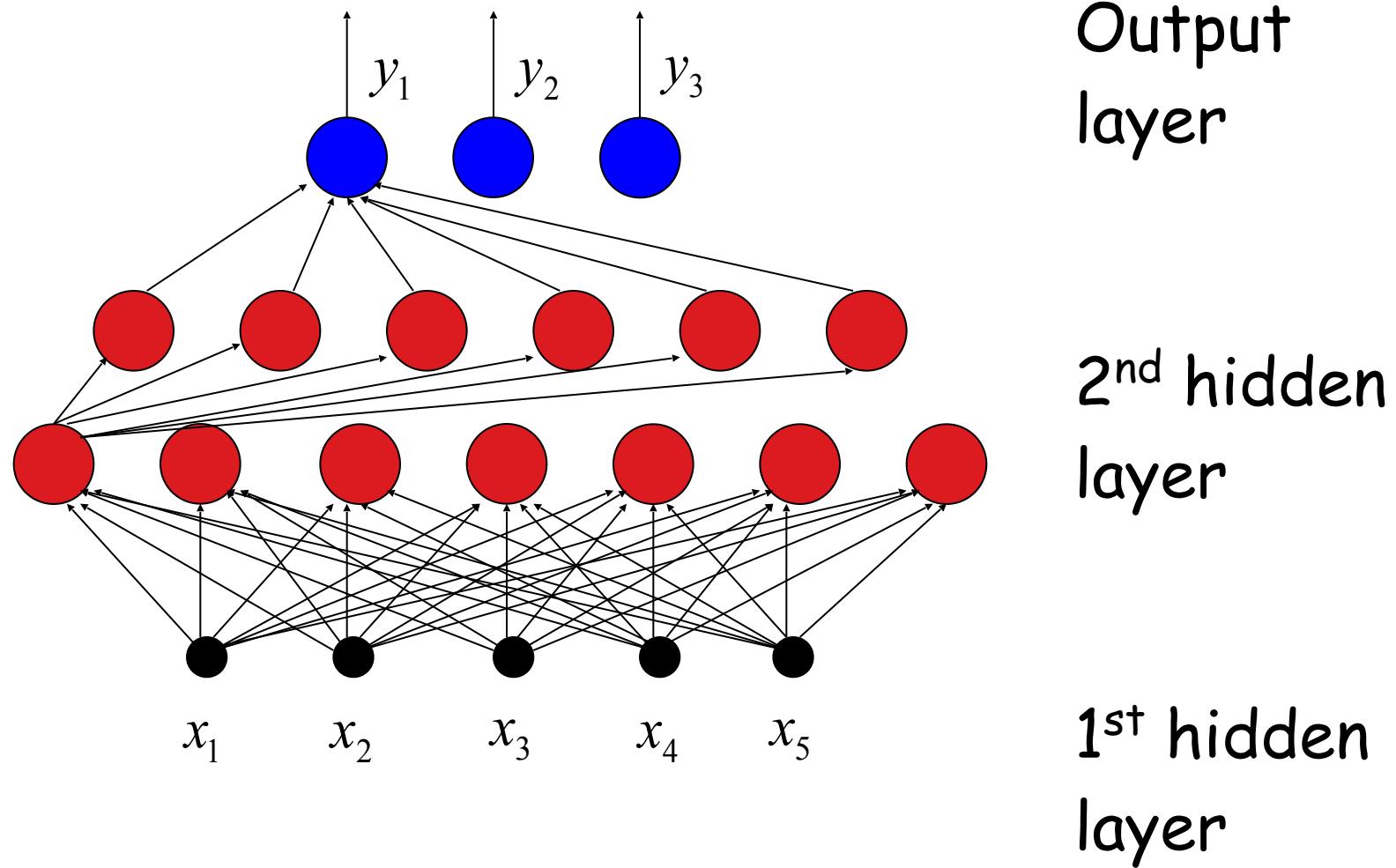


$$XOR(x_1, x_2)$$

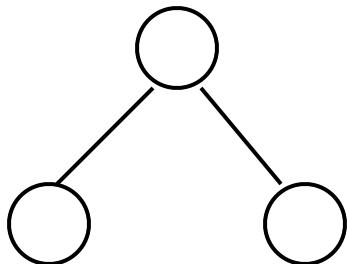
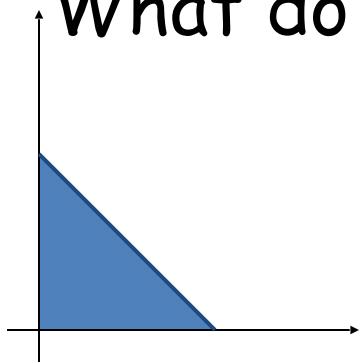


Feed-forward layered network

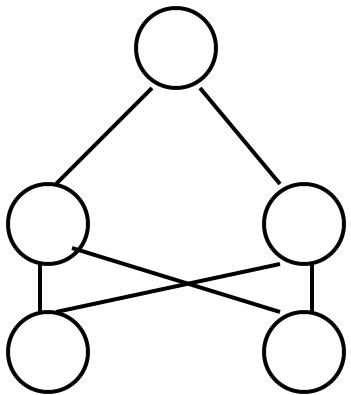
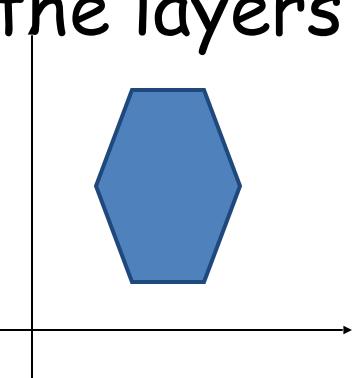
$$NN : X \rightarrow Y$$



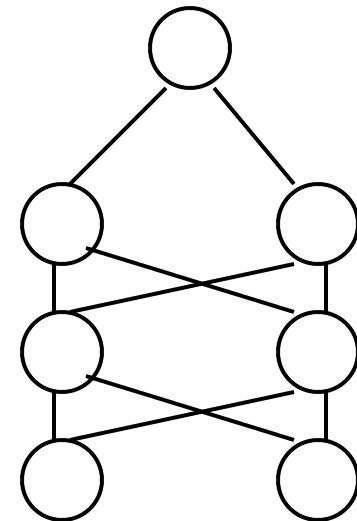
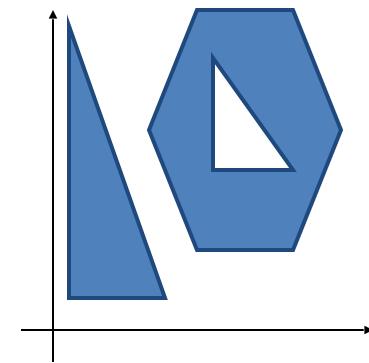
What do each of the layers do?



1st layer draws
linear boundaries

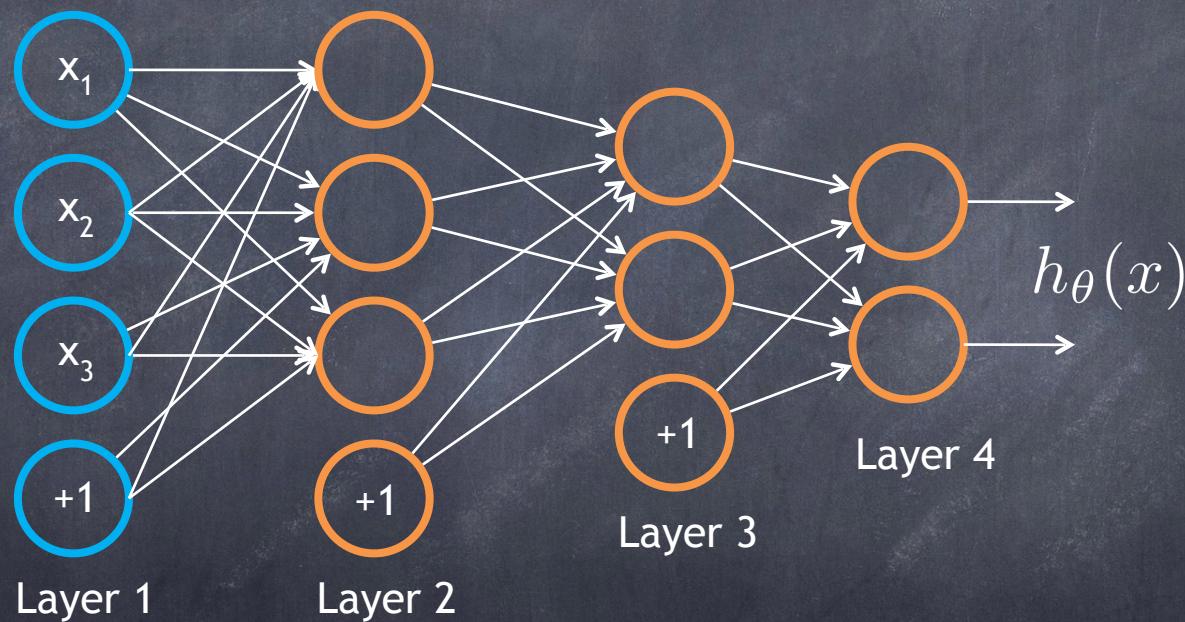


2nd layer combines
the boundaries

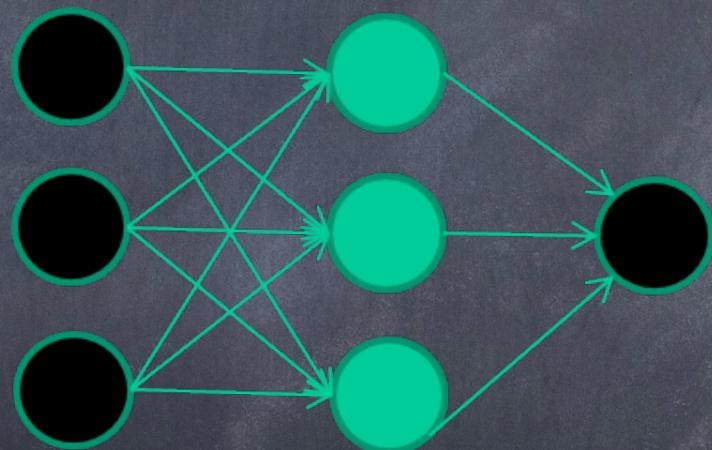


3rd layer can generate
arbitrarily complex
boundaries

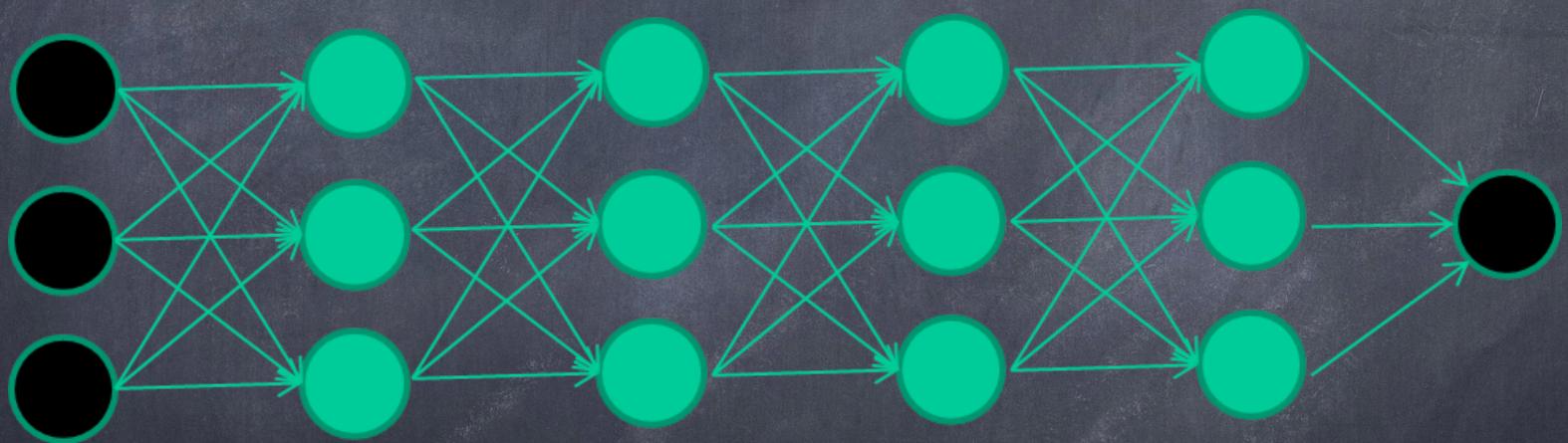
Neural Network



From Single Hidden Layer to Multiple Layers



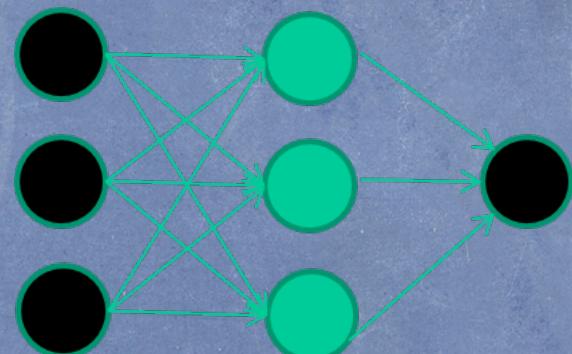
From Single Hidden Layer to Multiple Layers



Let Us Train A Simple Network

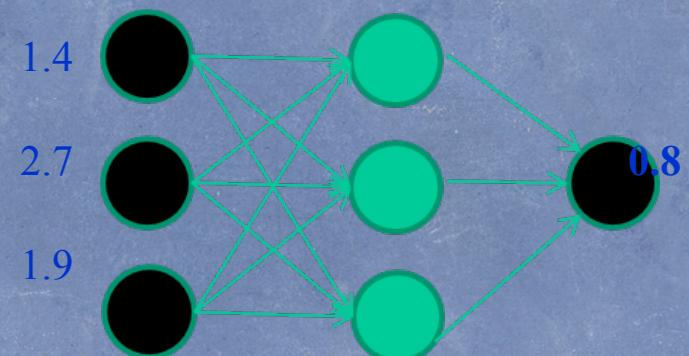
Training dataset

Fields	class		
1.4	2.7	1.9	0
3.8	3.4	3.2	0
6.4	2.8	1.7	1
4.1	0.1	0.2	0
etc ...			

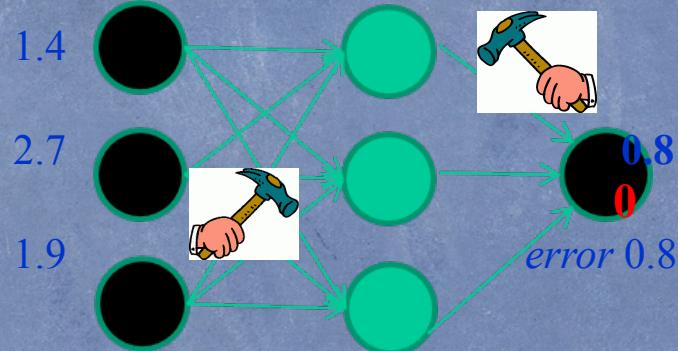


Let Us Train A Simple Network

Training data			
Fields	class		
1.4	2.7	1.9	0
3.8	3.4	3.2	0
6.4	2.8	1.7	1
4.1	0.1	0.2	0
etc ...			



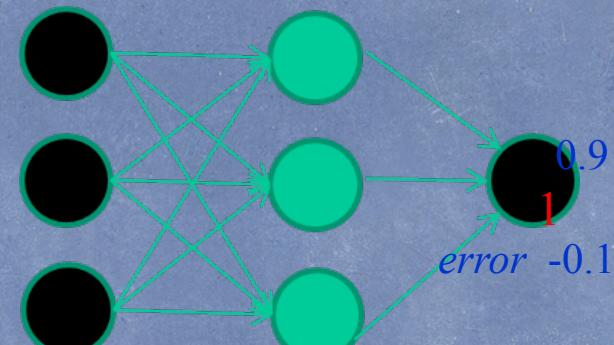
Let Us Train A Simple Network



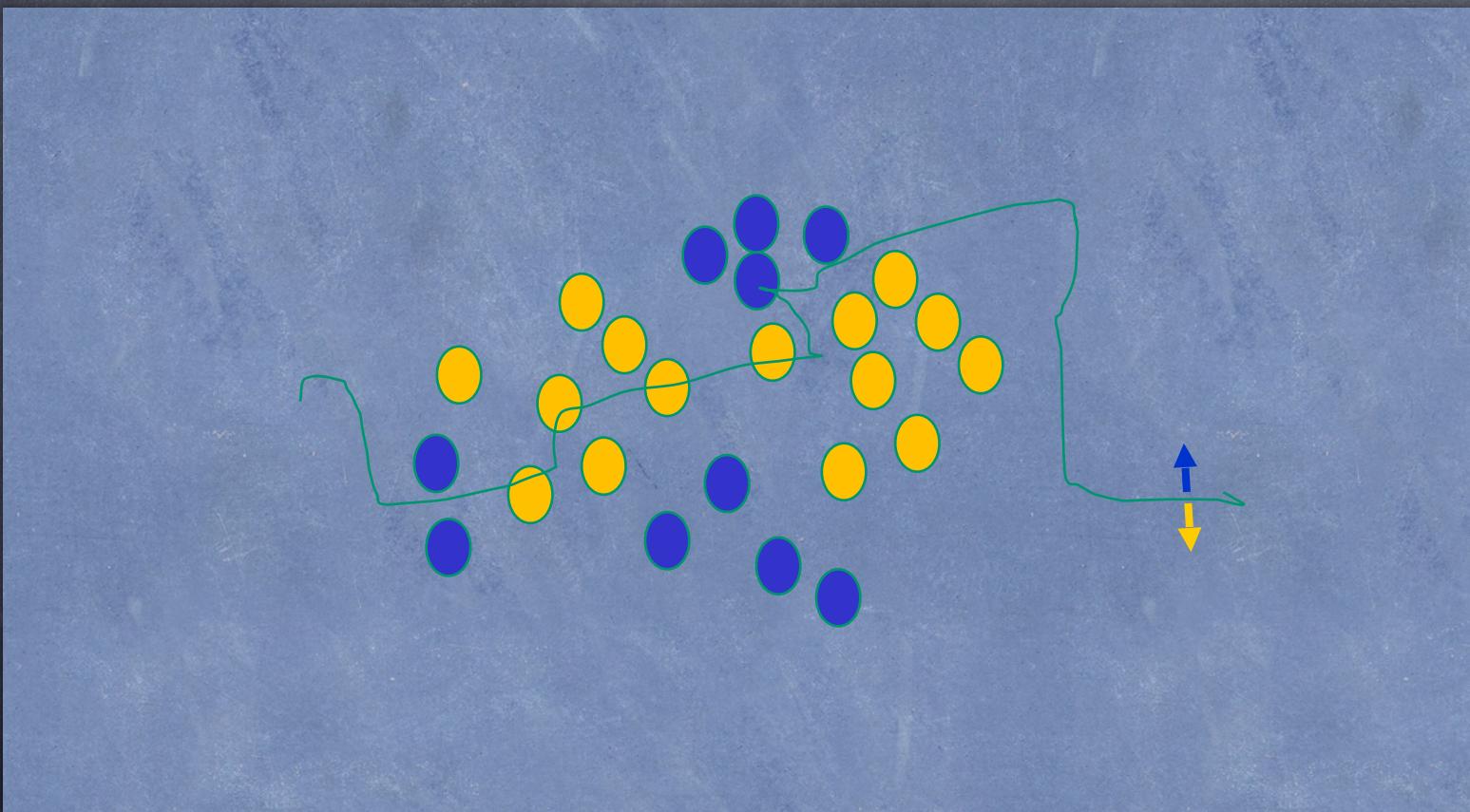
Let Us Train A Simple Network

Training data

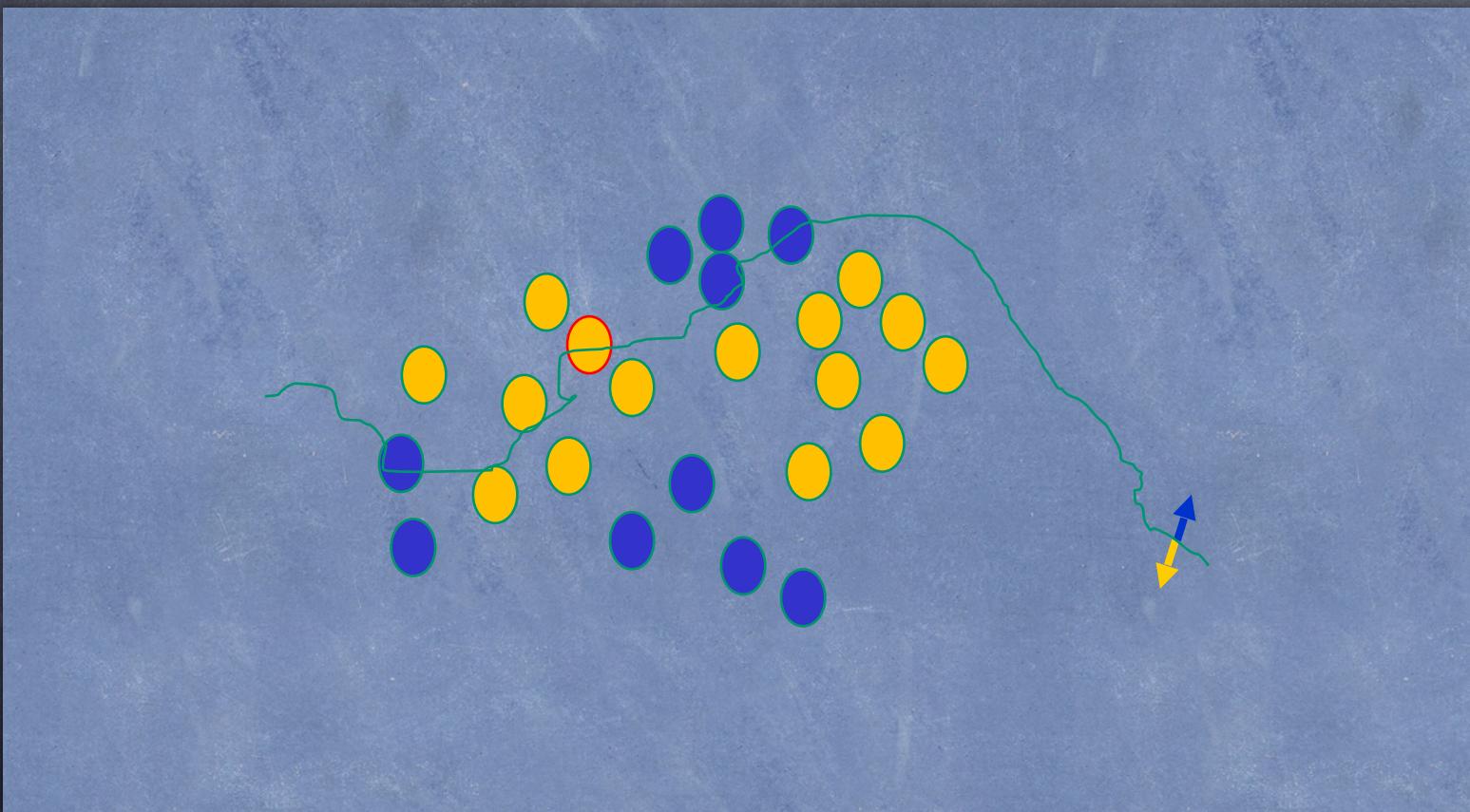
Fields	class		
1.4	2.7	1.9	0
3.8	3.4	3.2	0
6.4	2.8	1.7	1
4.1	0.1	0.2	0
etc ...			



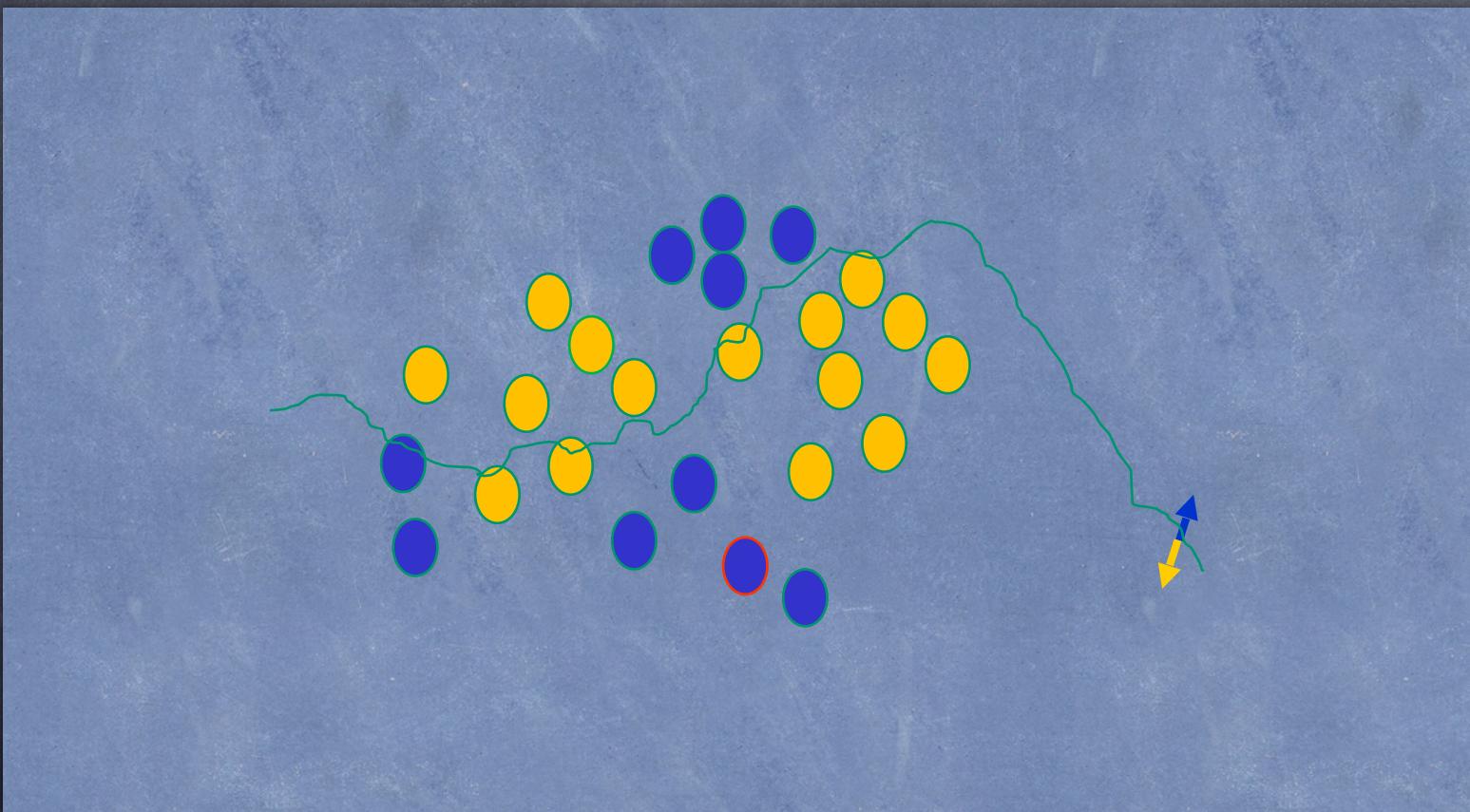
What is happening at the backend?



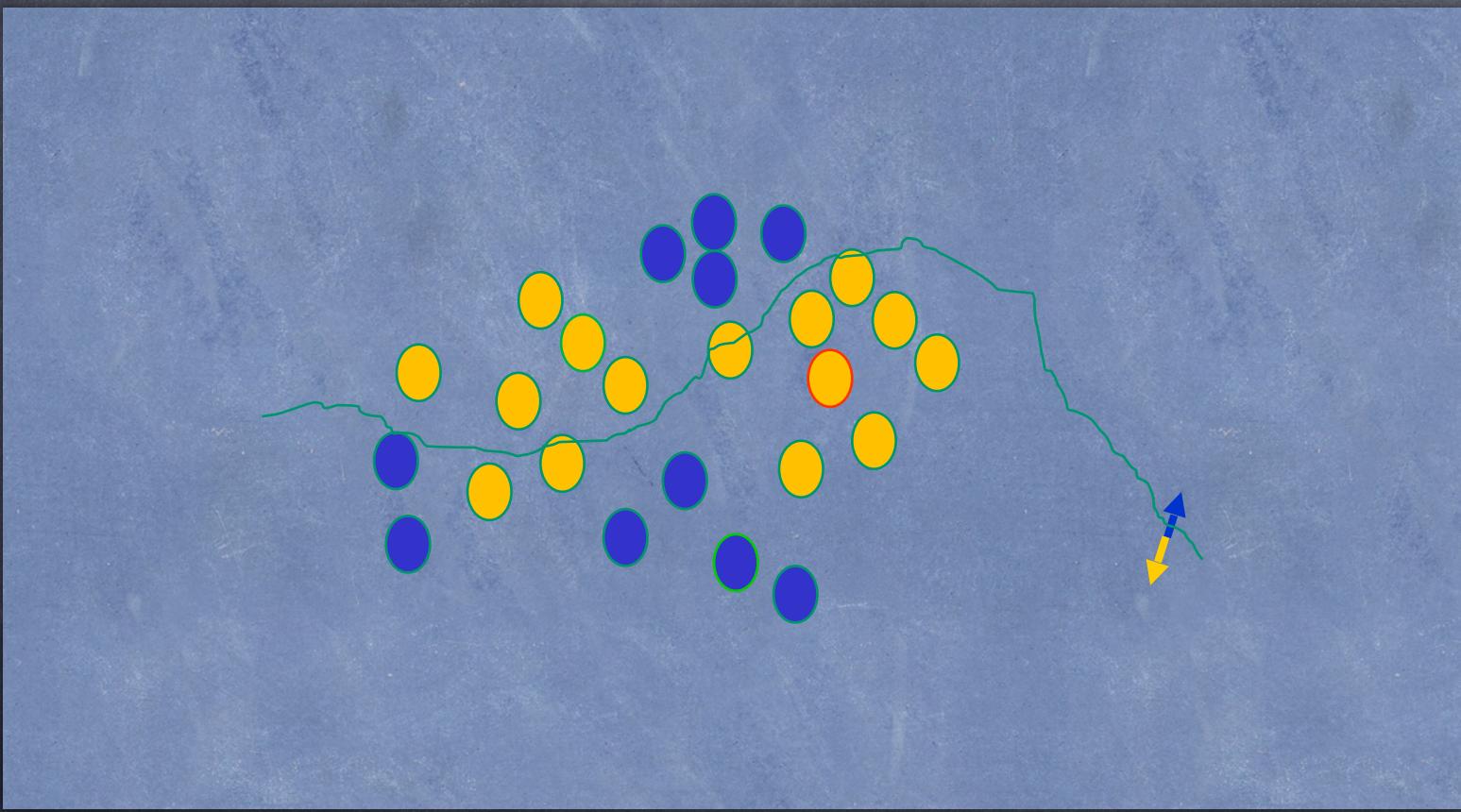
What is happening at the backend?



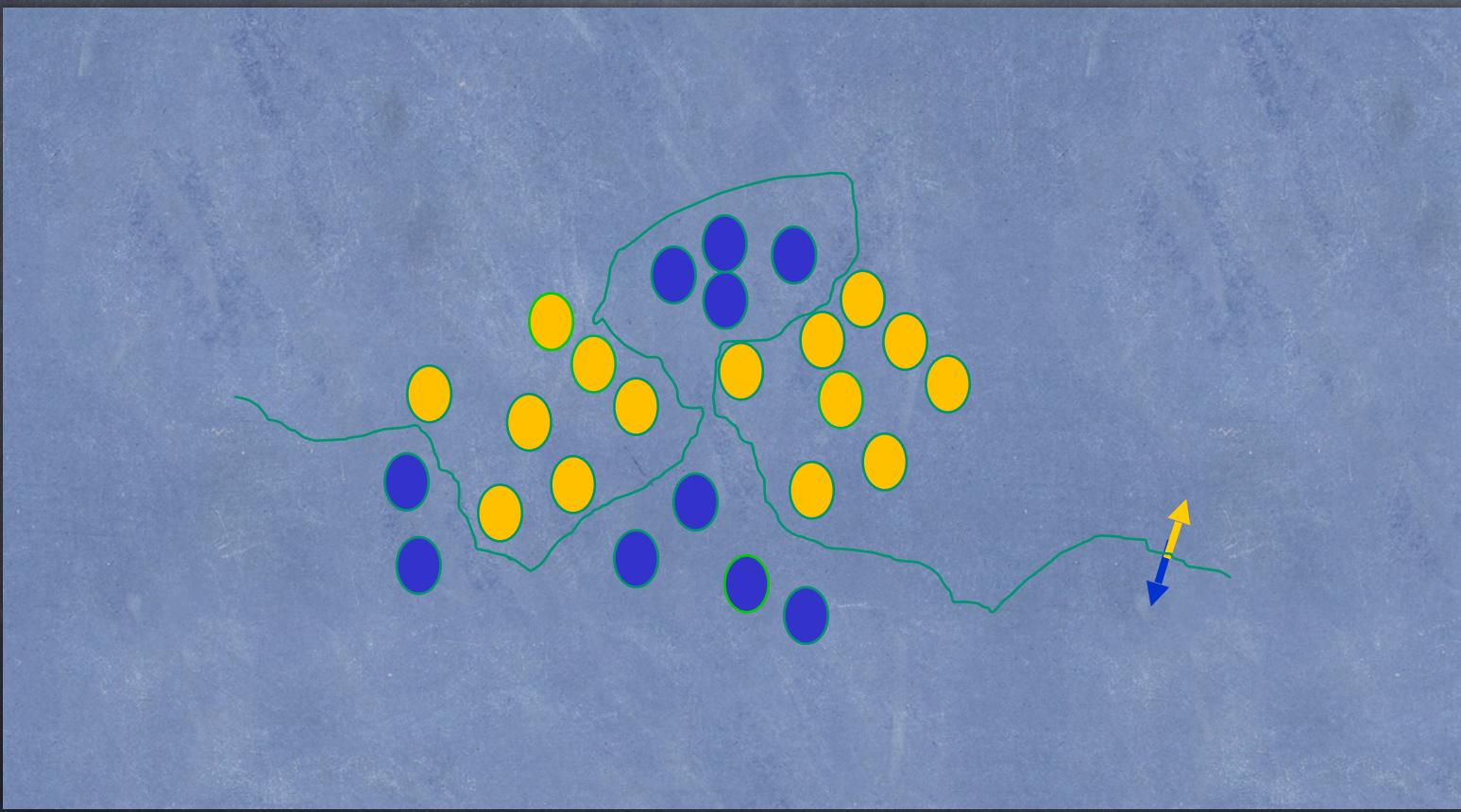
What is happening at the backend?



What is happening at the backend?



What is happening at the backend?

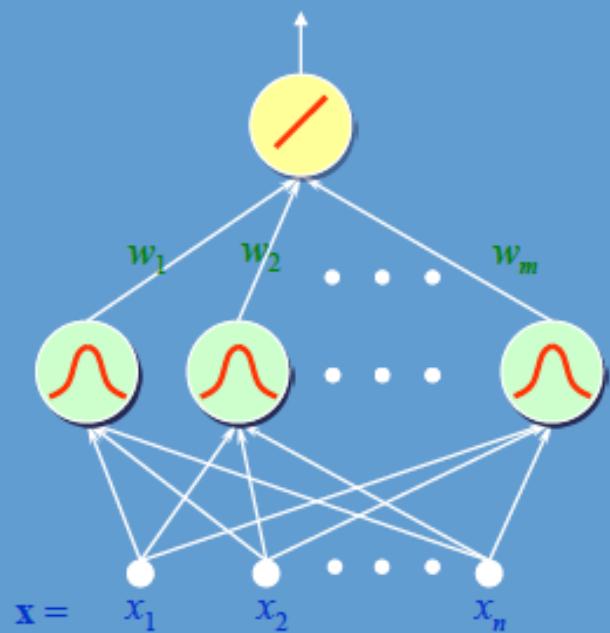


Training Neural Network

Training set $\mathcal{T} = \left\{ \left(\mathbf{x}^{(k)}, y^{(k)} \right) \right\}_{k=1}^p$

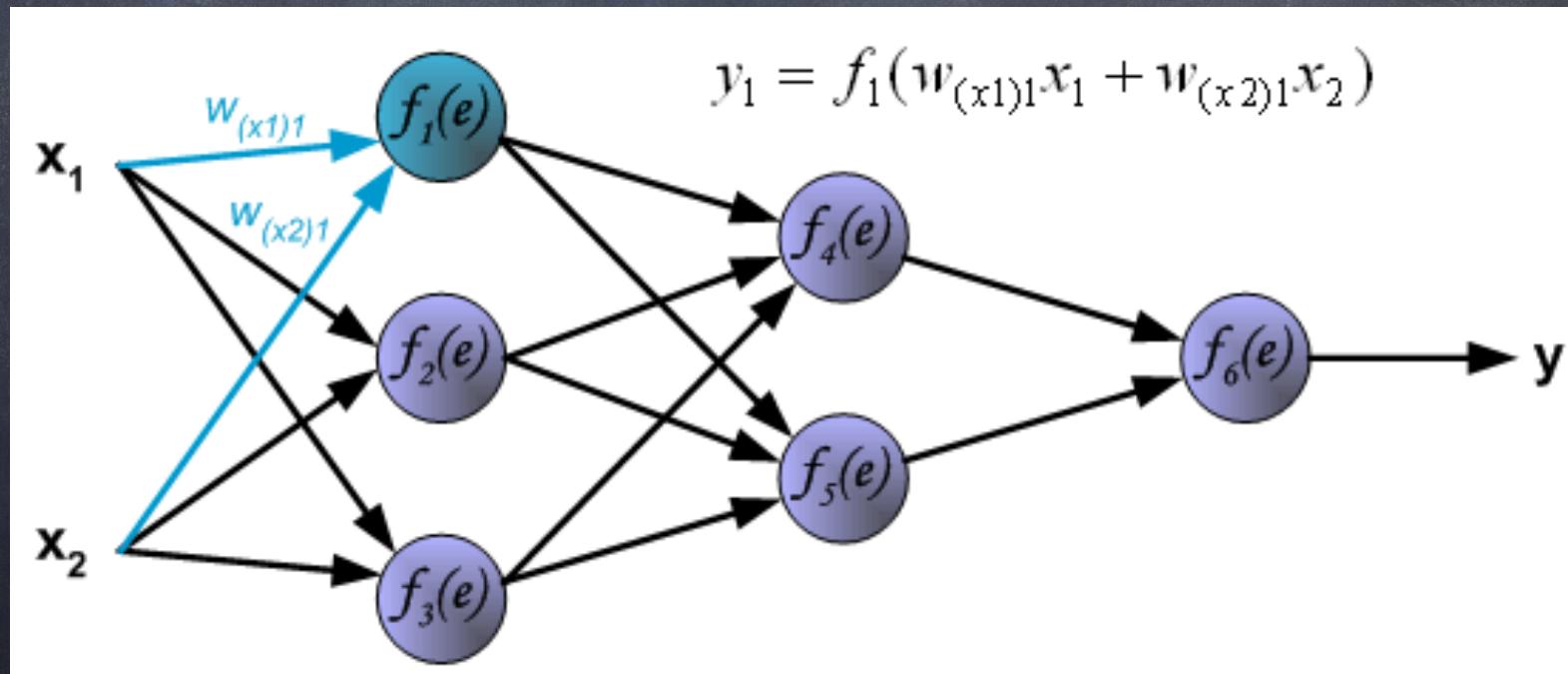
Goal $y^{(k)} \approx f(\mathbf{x}^{(k)})$ for all k

$$\min E = \frac{1}{2} \sum_{k=1}^p \left[y^{(k)} - f(\mathbf{x}^{(k)}) \right]^2$$

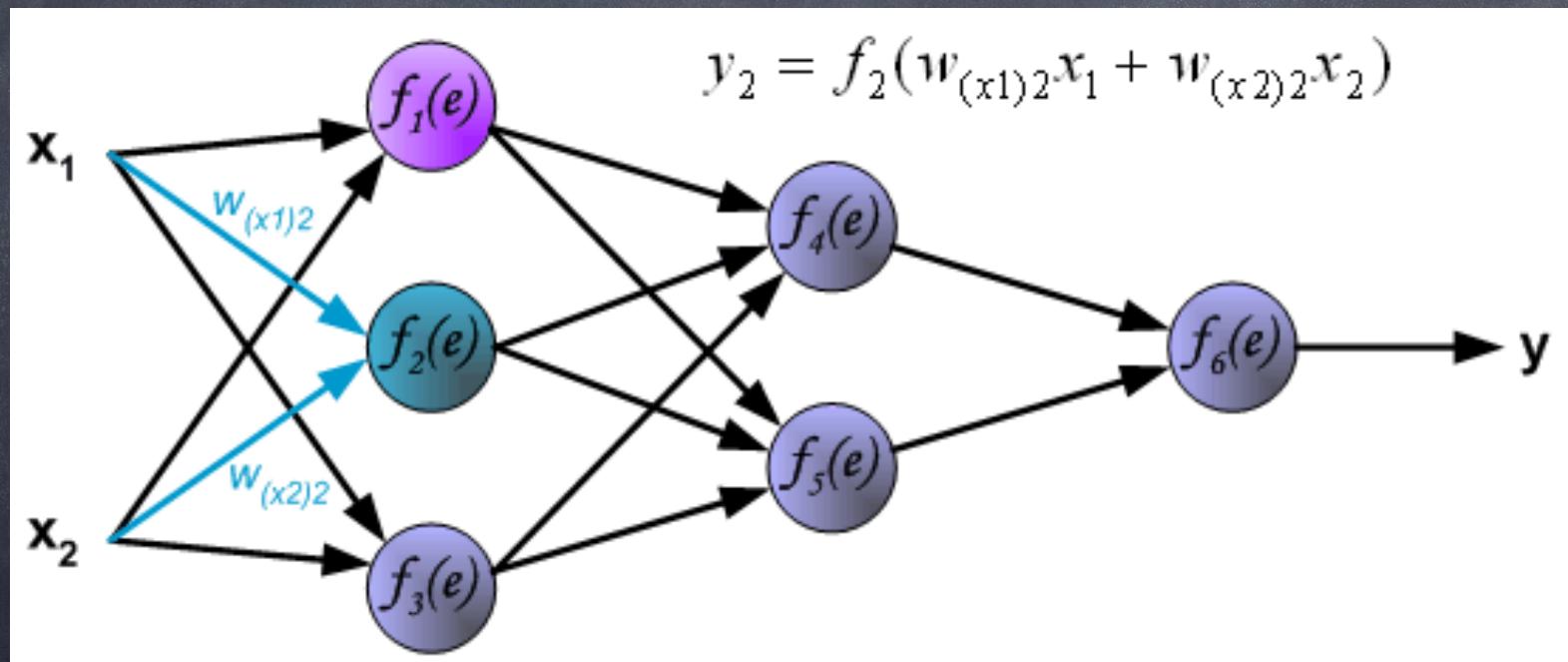


How to Train Neural Network

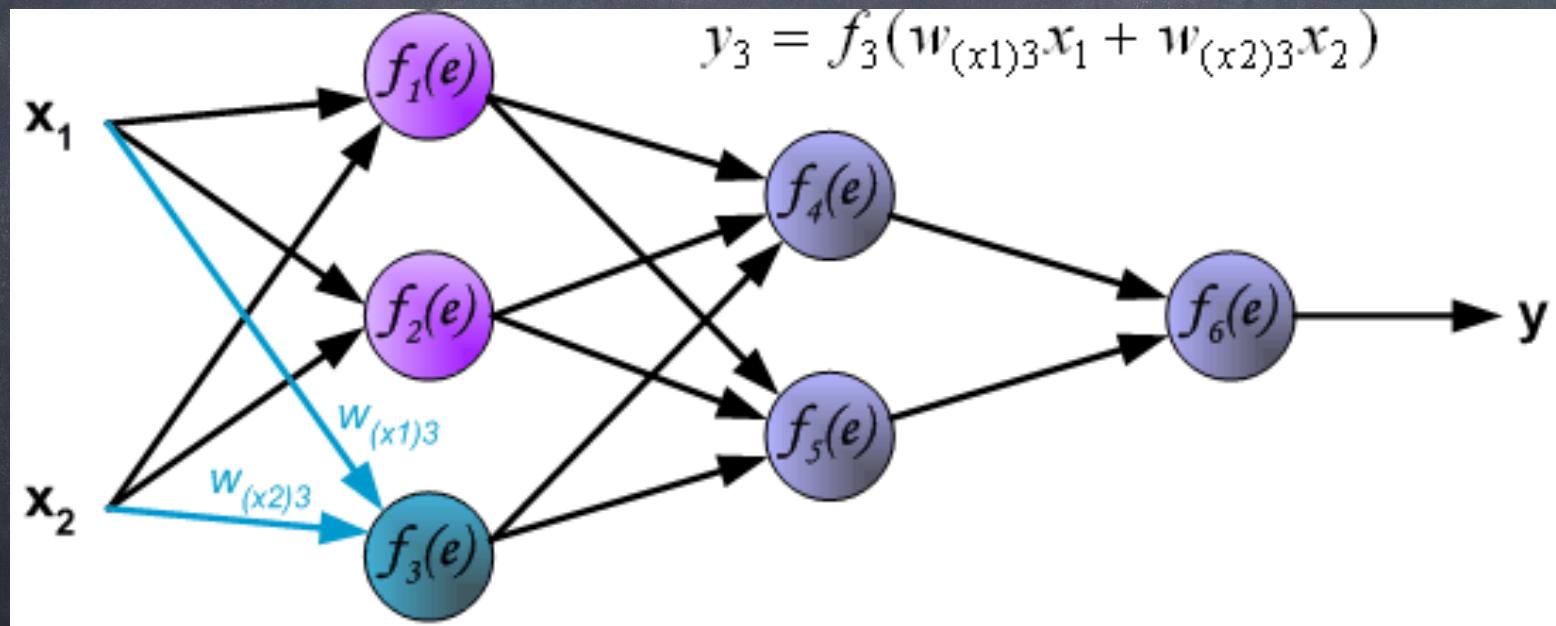
- Back-propagation approach



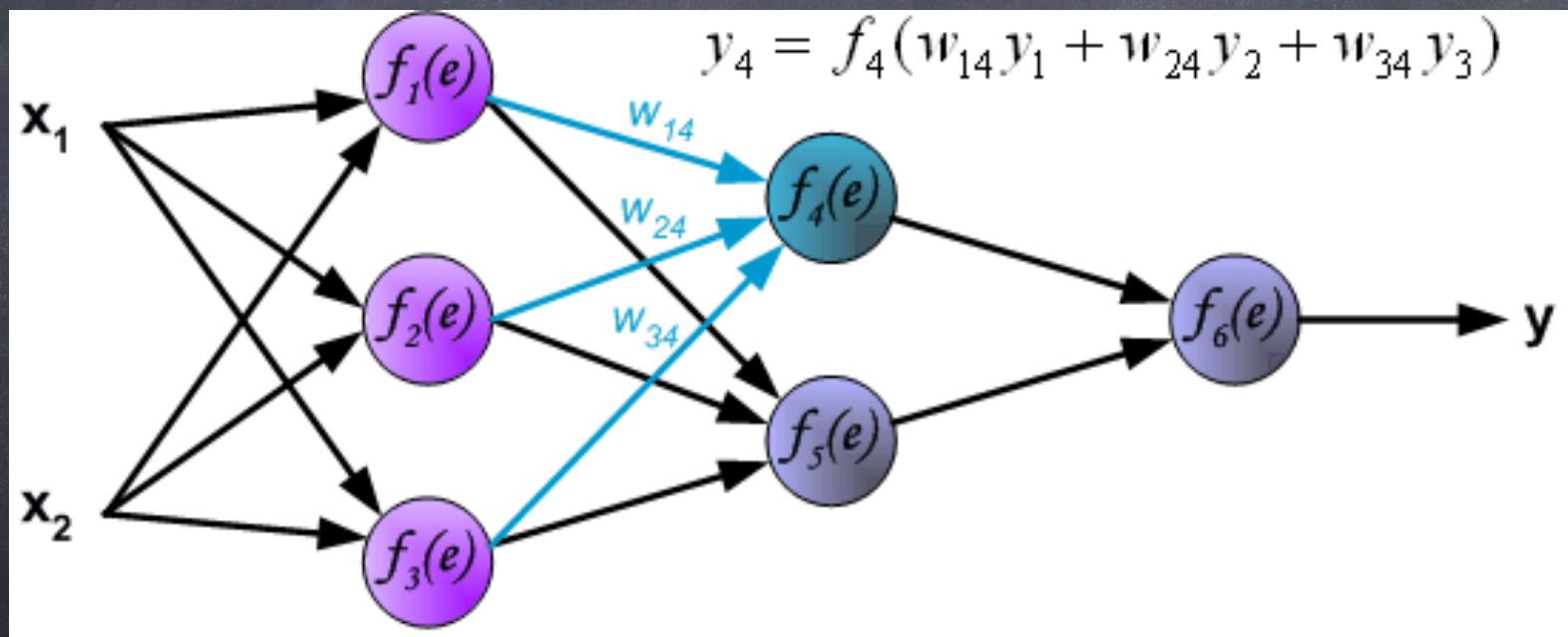
How to Train Neural Network



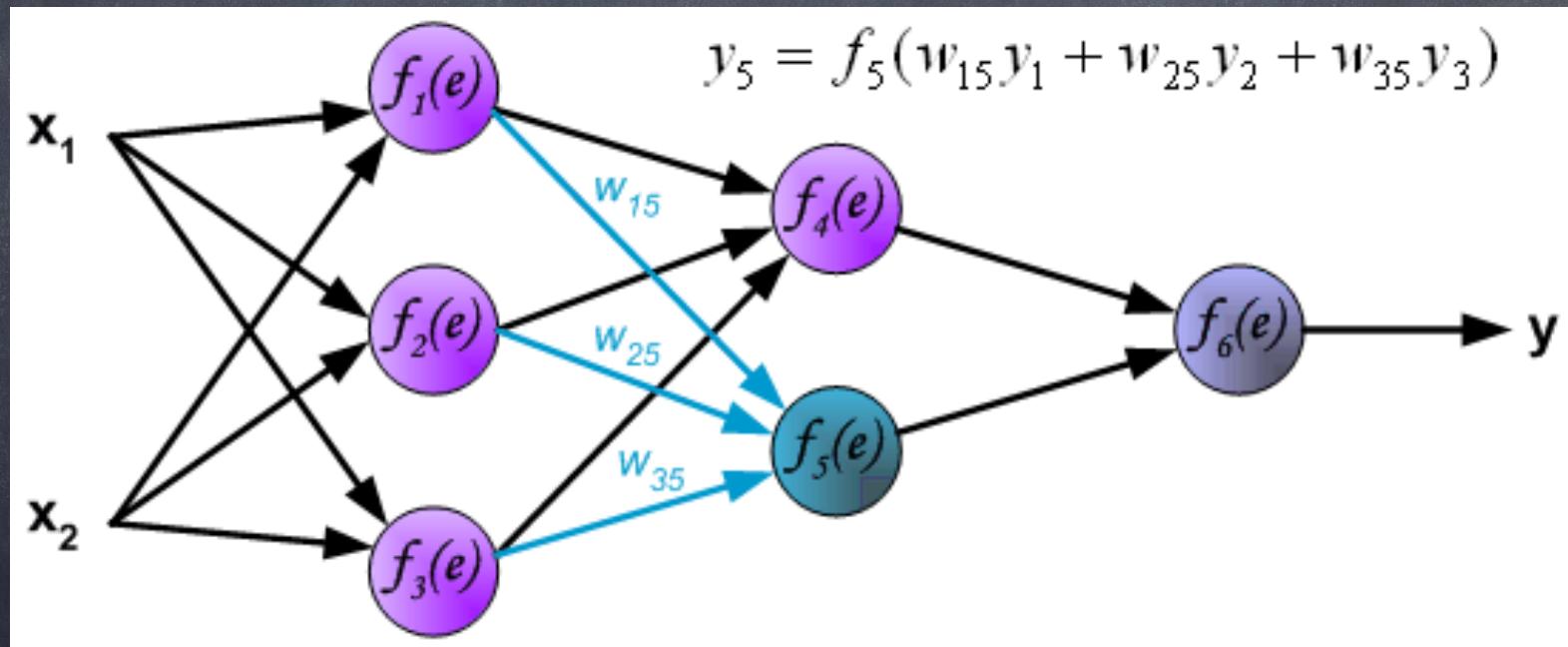
How to Train Neural Network



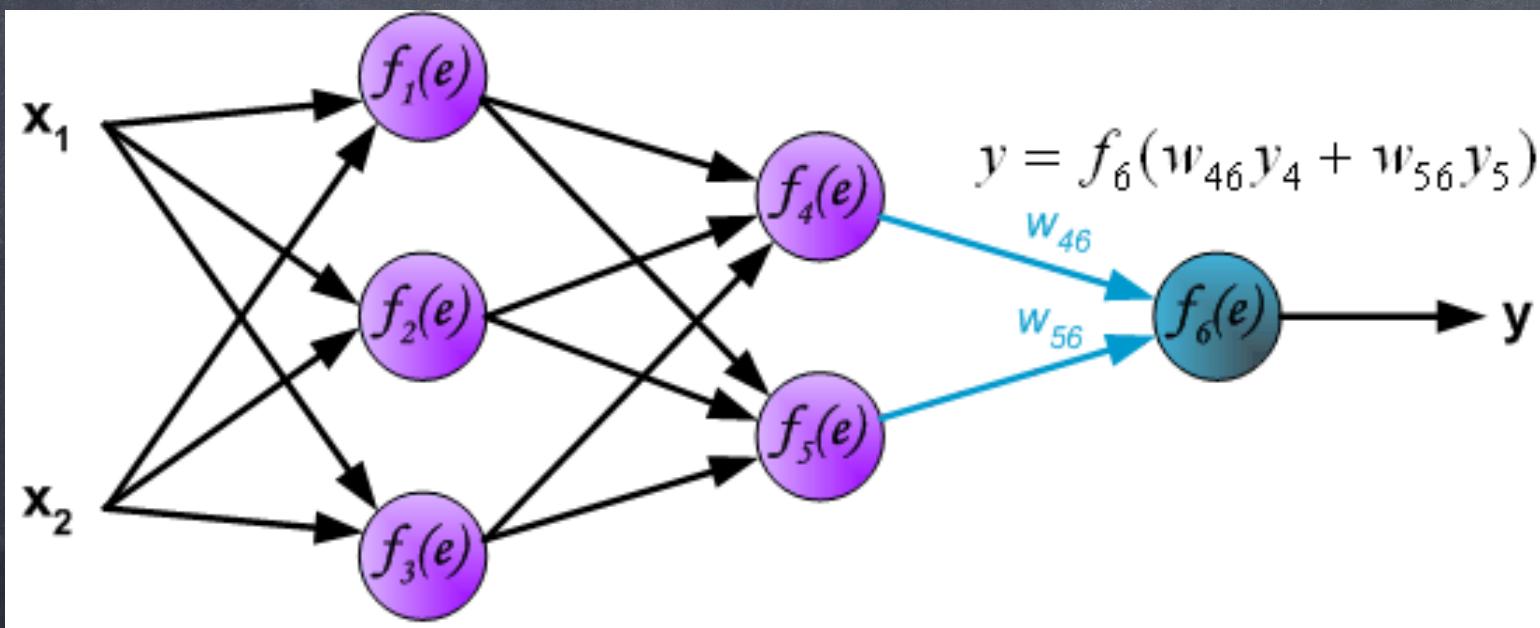
How to Train Neural Network



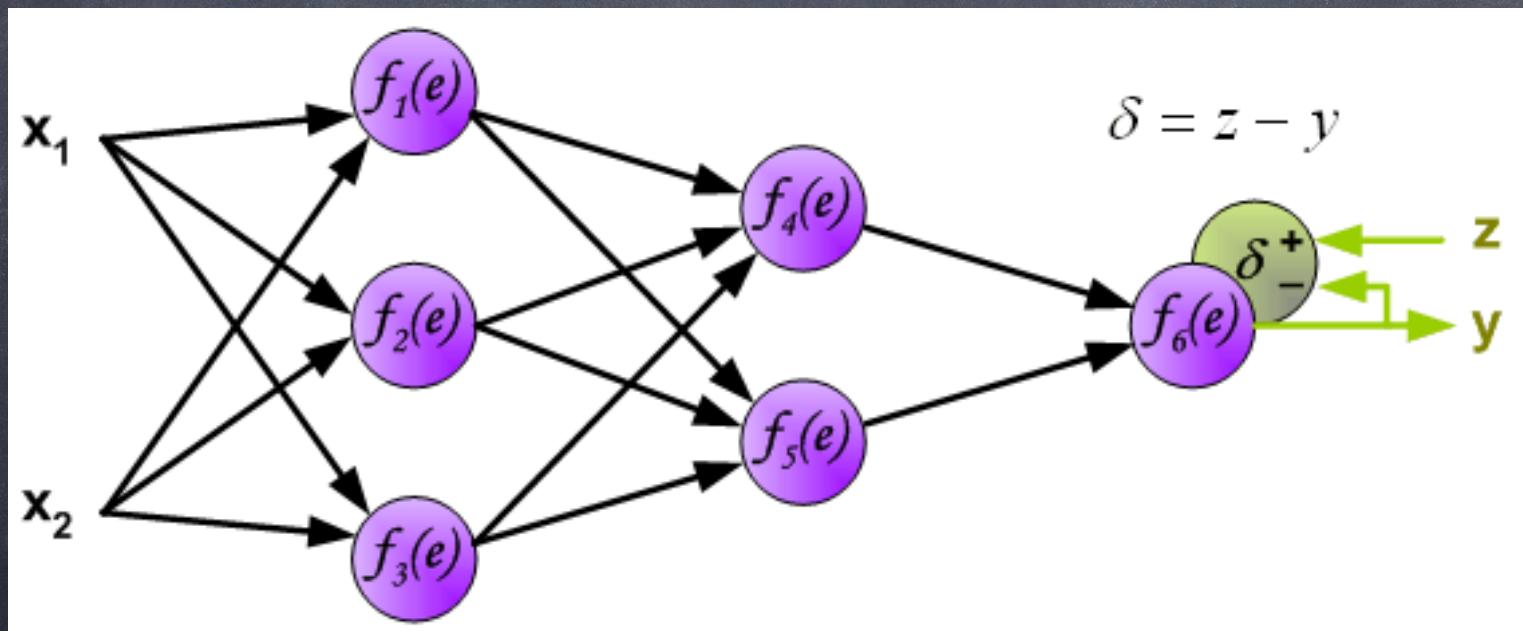
How to Train Neural Network



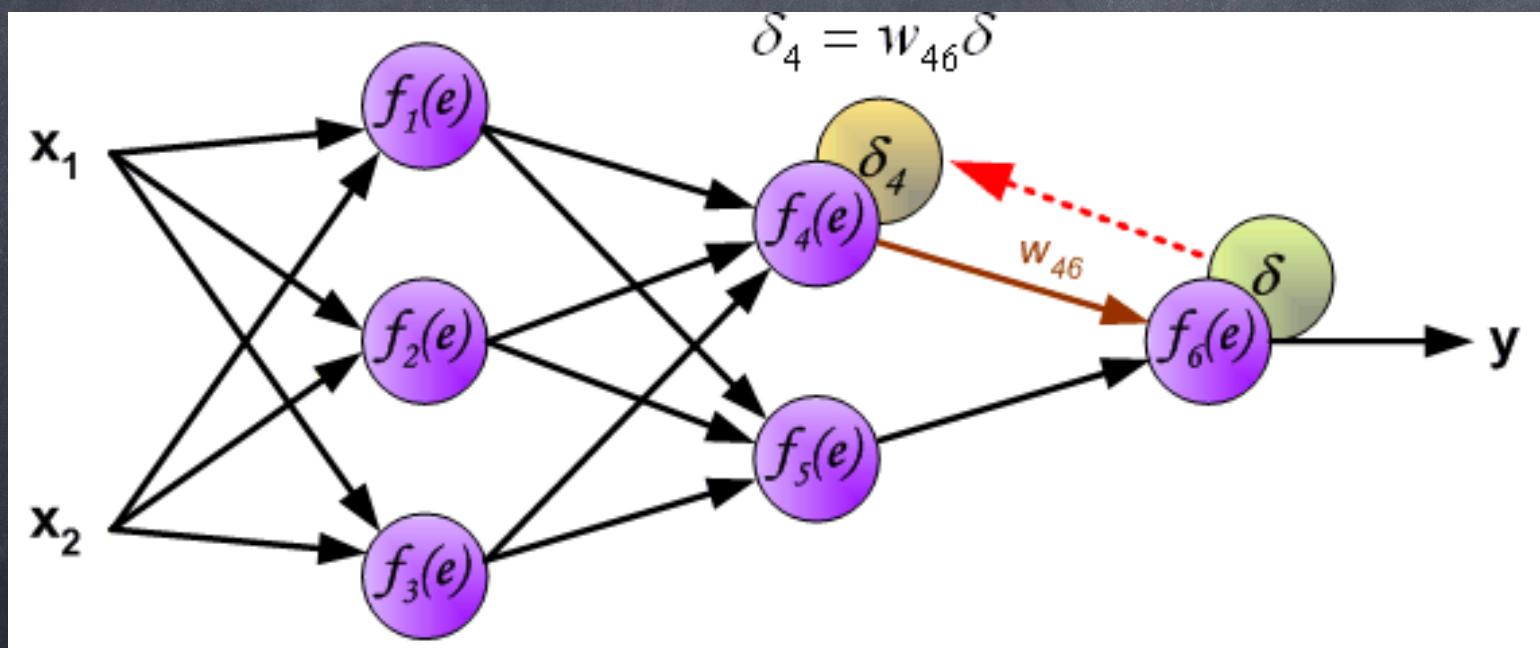
How to Train Neural Network



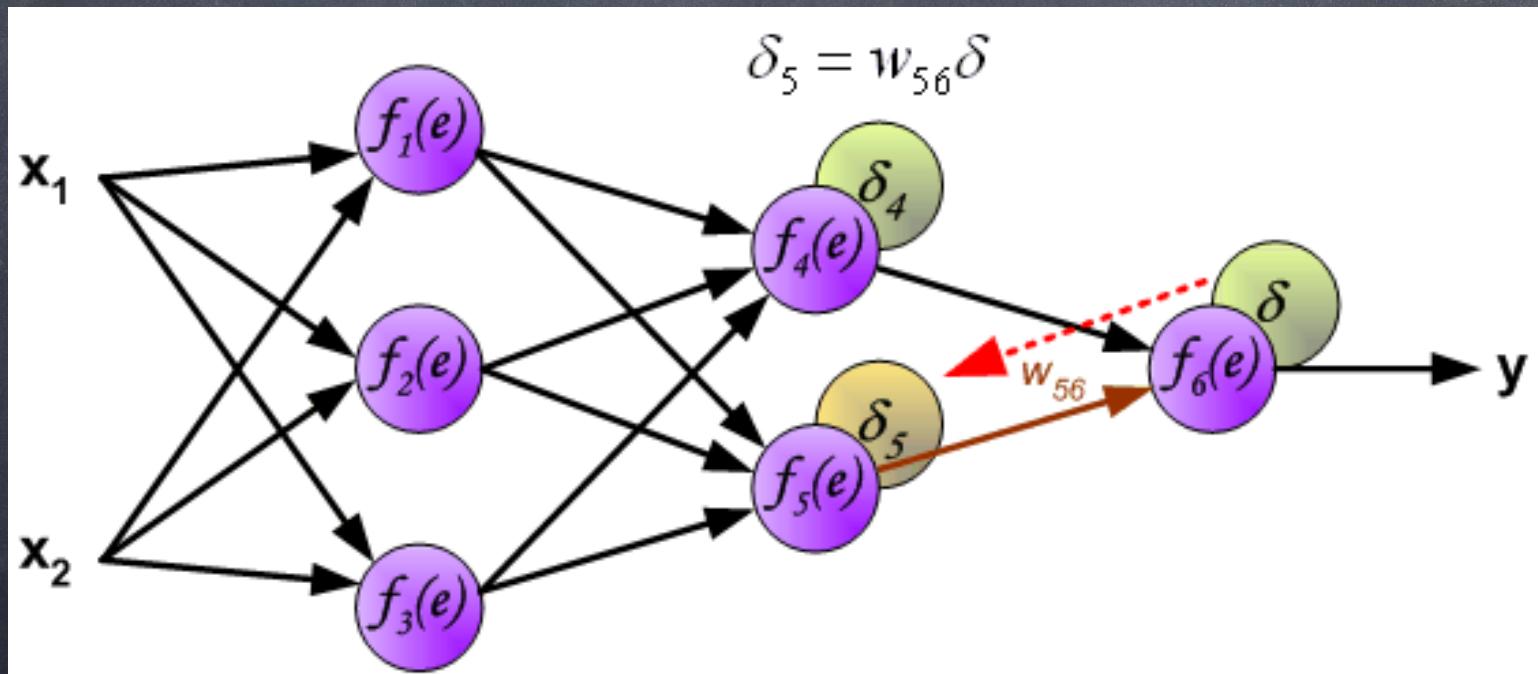
How to Train Neural Network



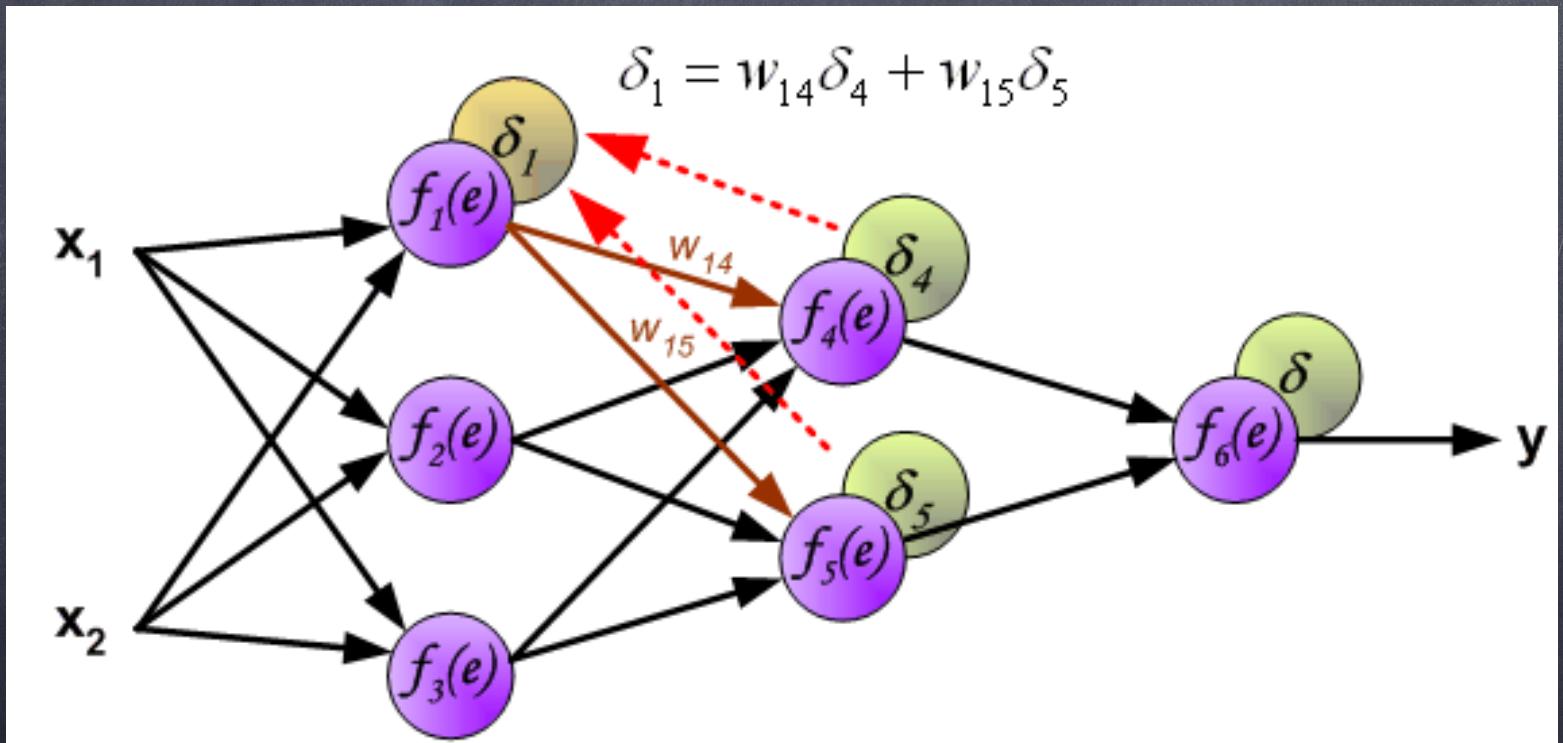
How to Train Neural Network



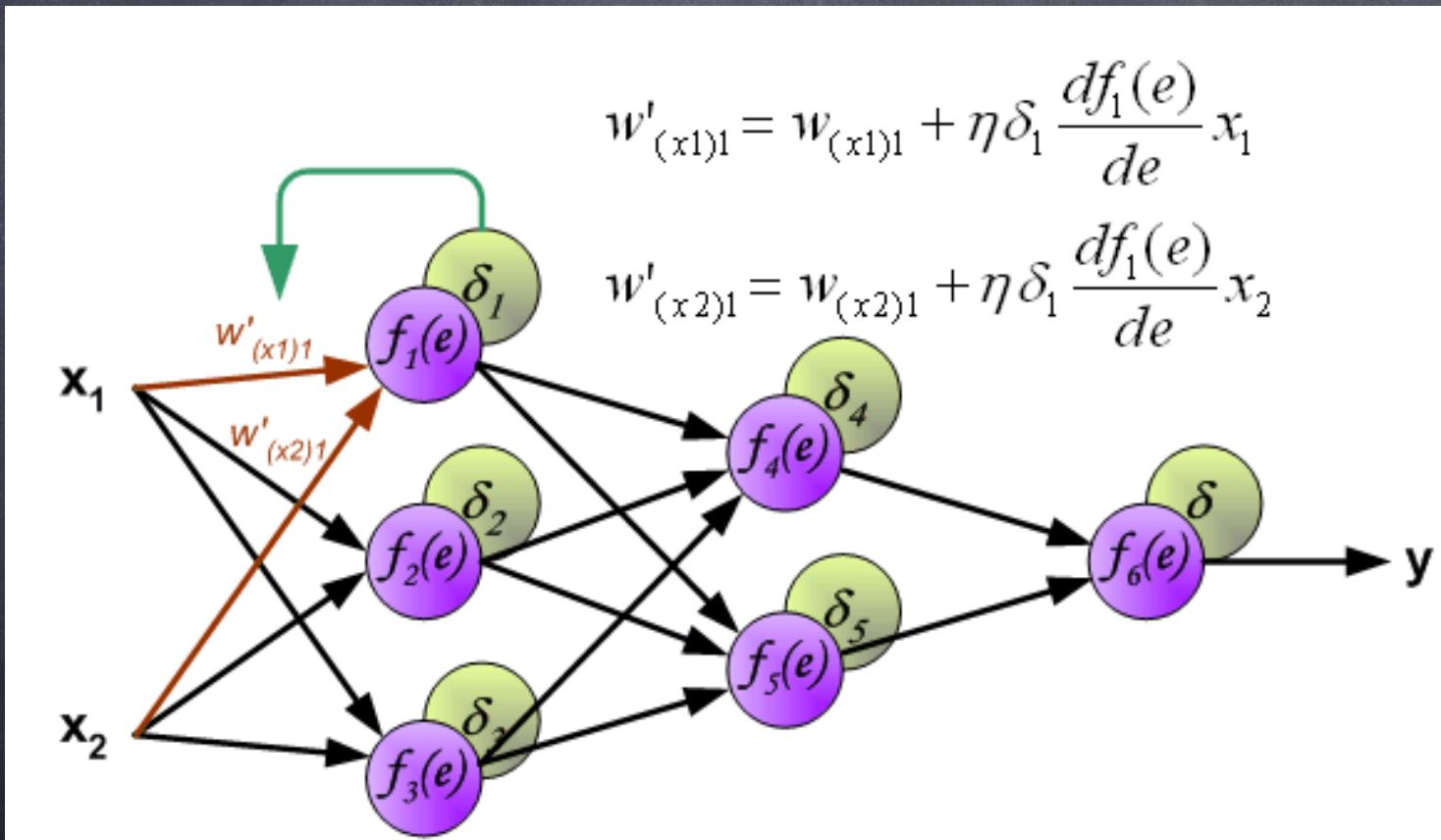
How to Train Neural Network



How to Train Neural Network



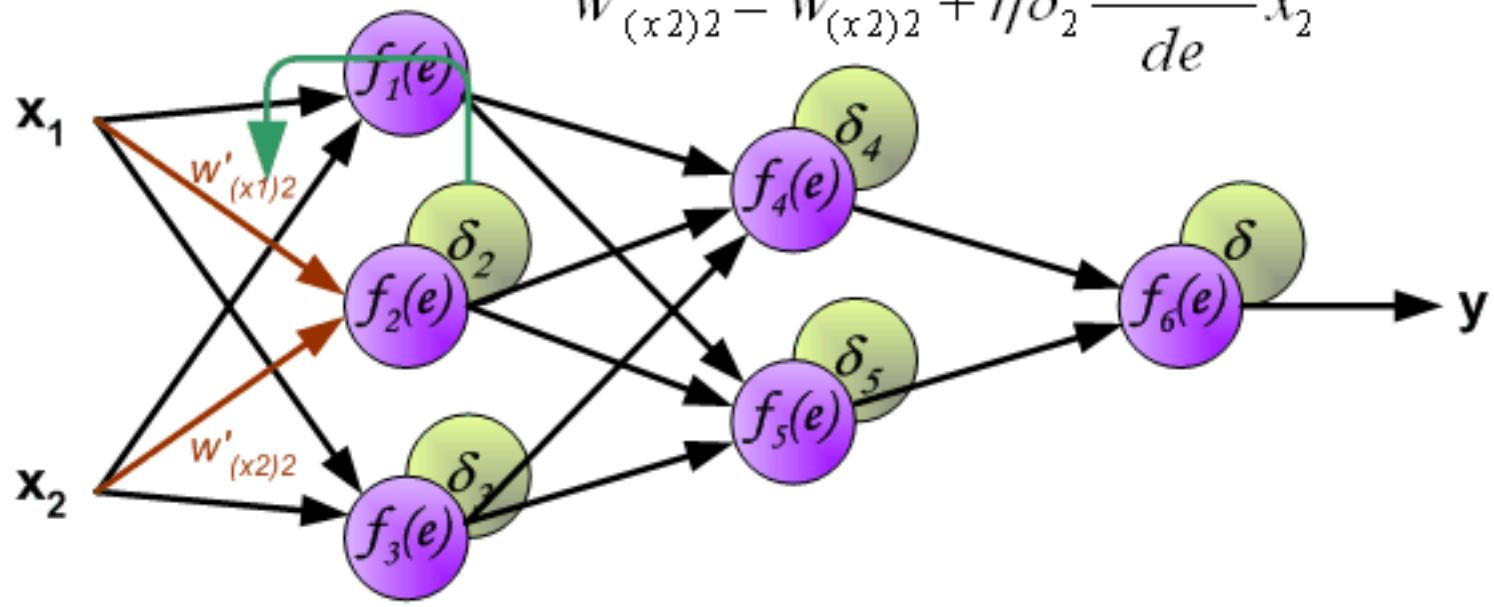
How to Train Neural Network



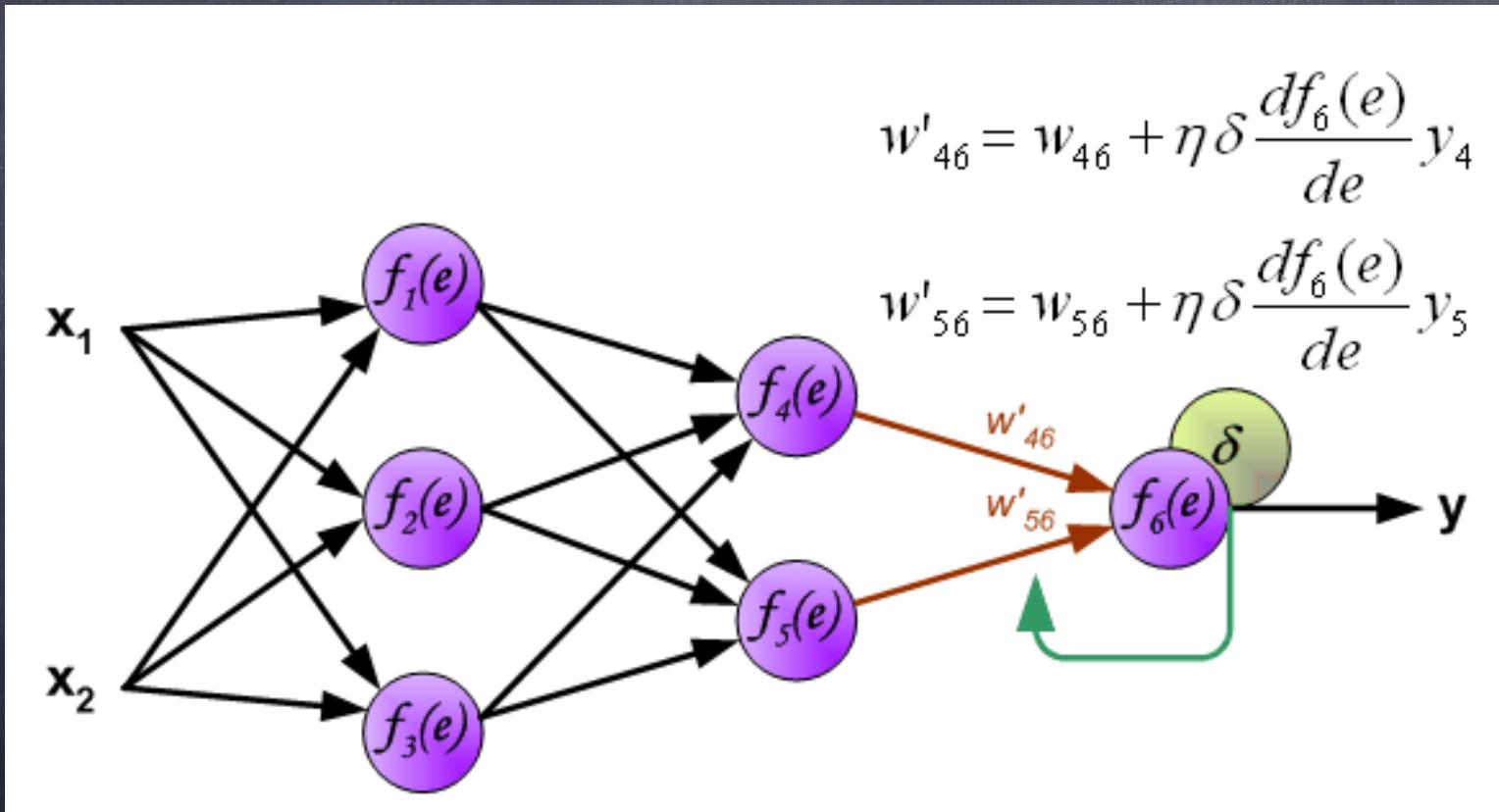
How to Train Neural Network

$$w'_{(x1)2} = w_{(x1)2} + \eta \delta_2 \frac{df_2(e)}{de} x_1$$

$$w'_{(x2)2} = w_{(x2)2} + \eta \delta_2 \frac{df_2(e)}{de} x_2$$

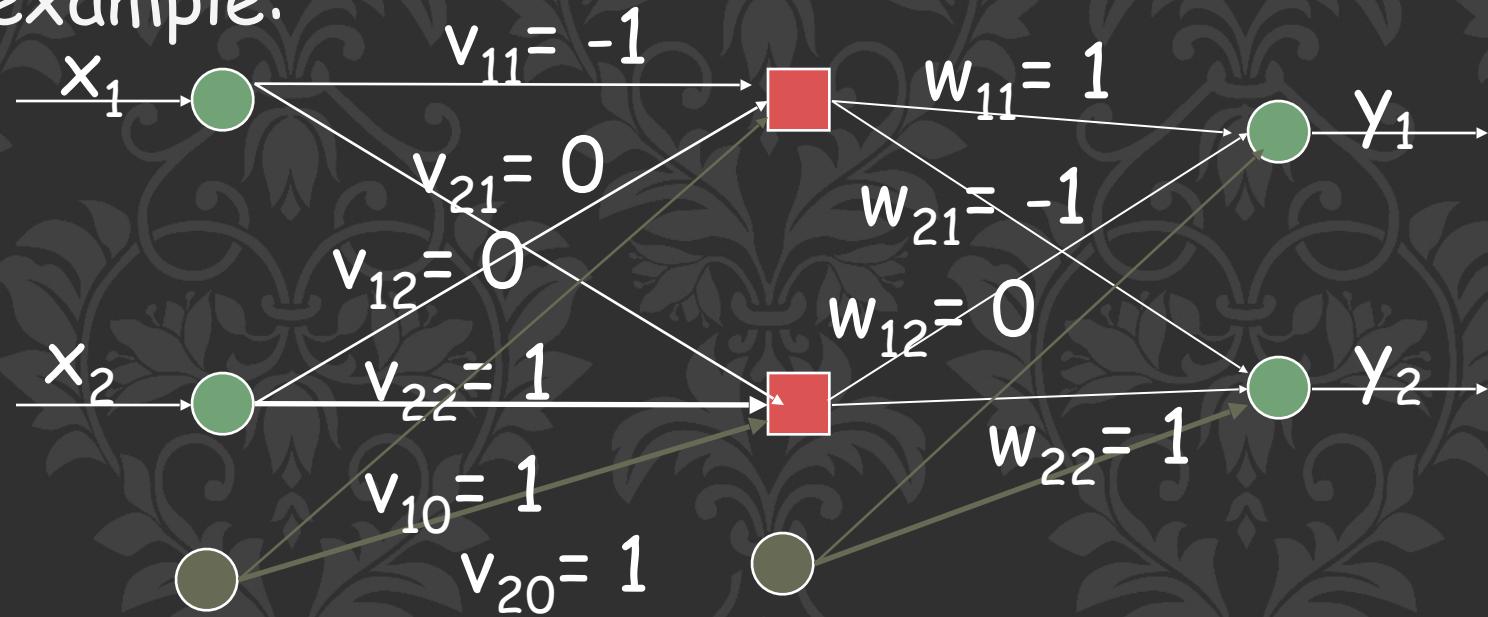


How to Train Neural Network



Once weight changes are computed for all units,
weights are updated
at the same time (bias included as weights here).

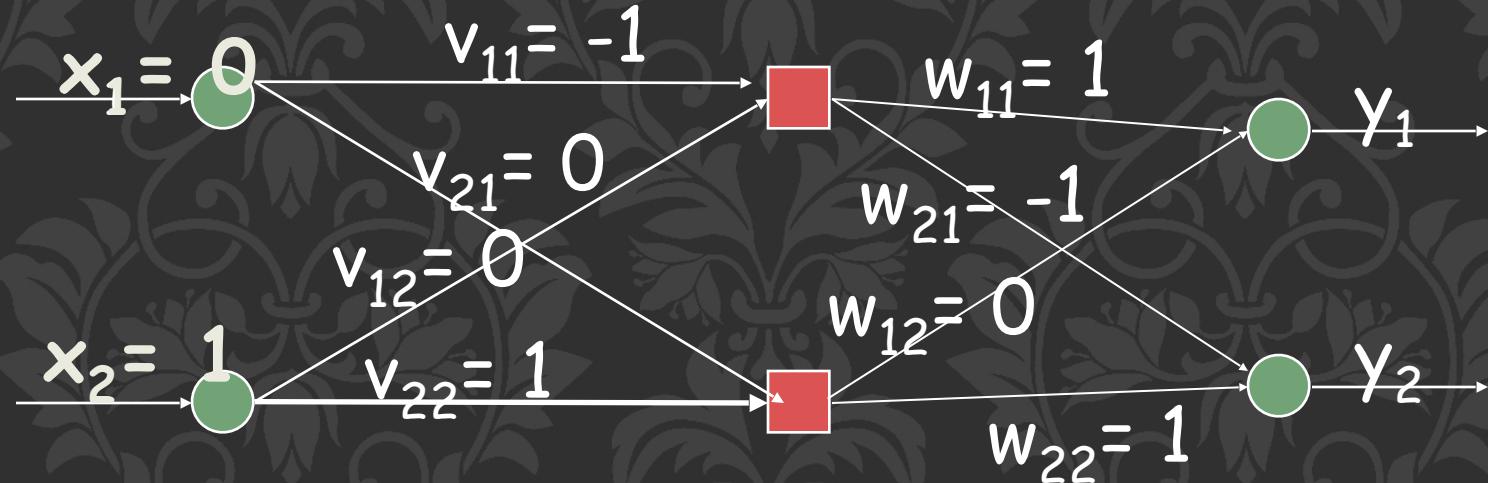
An example:



Use identity activation function (ie $g(a) = a$)

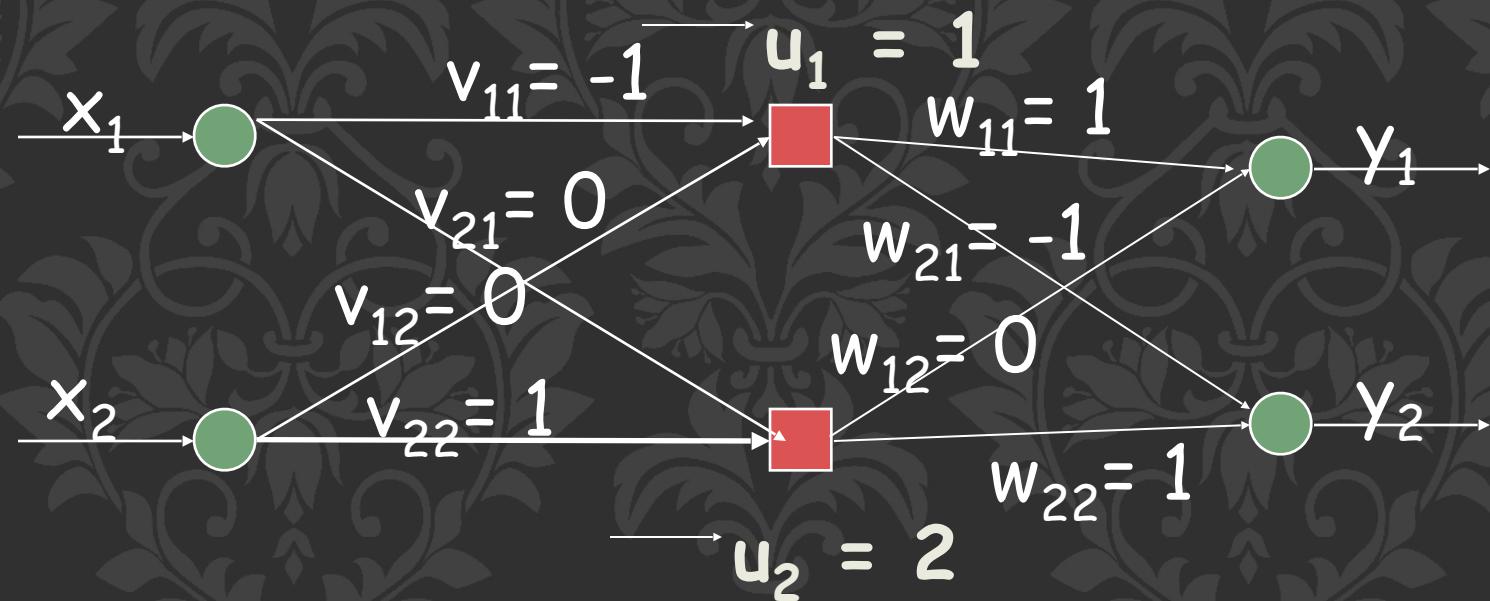
All biases set to 1. Will not draw them for clarity.

Learning rate $\eta = 0.1$



Have input $[0 \ 1]$ with target $[1 \ 0]$.

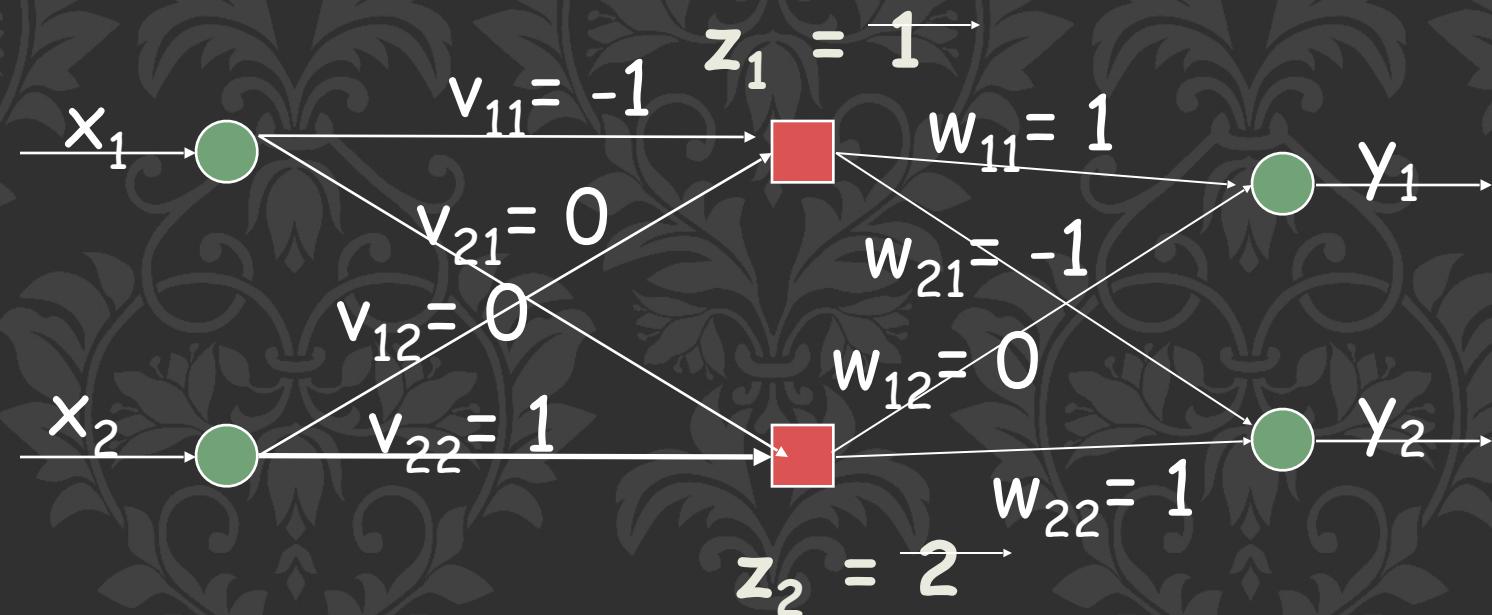
Forward pass. Calculate 1st layer activations:



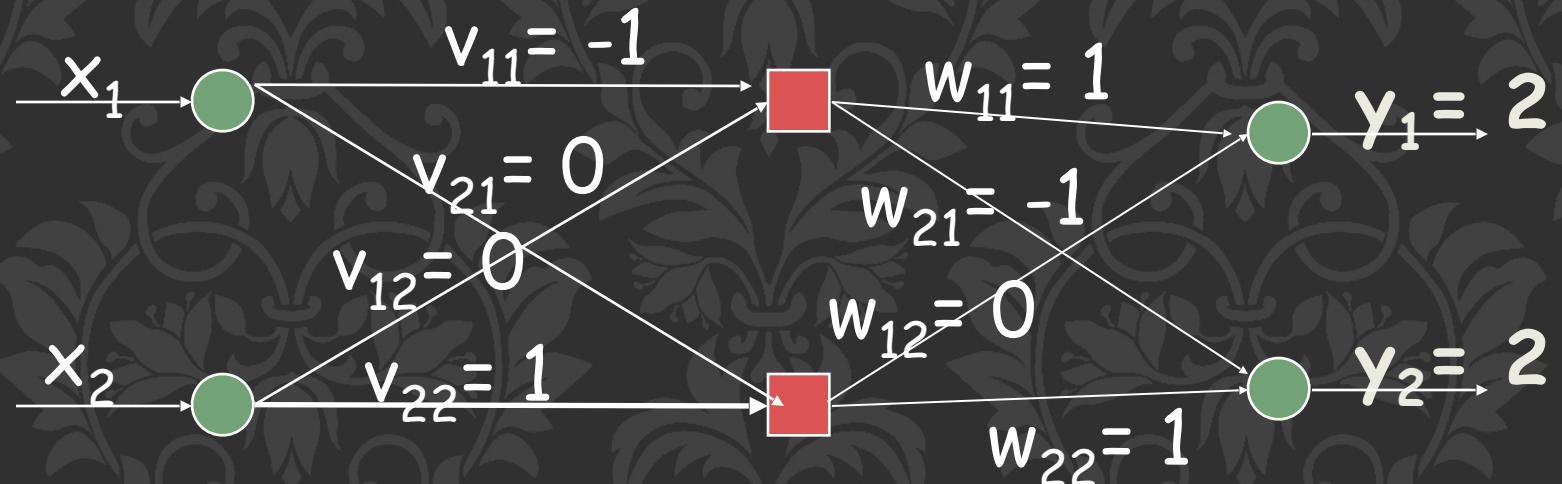
$$u_1 = -1 \times 0 + 0 \times 1 + 1 = 1$$

$$u_2 = 0 \times 0 + 1 \times 1 + 1 = 2$$

Calculate first layer outputs by passing activations thru activation functions



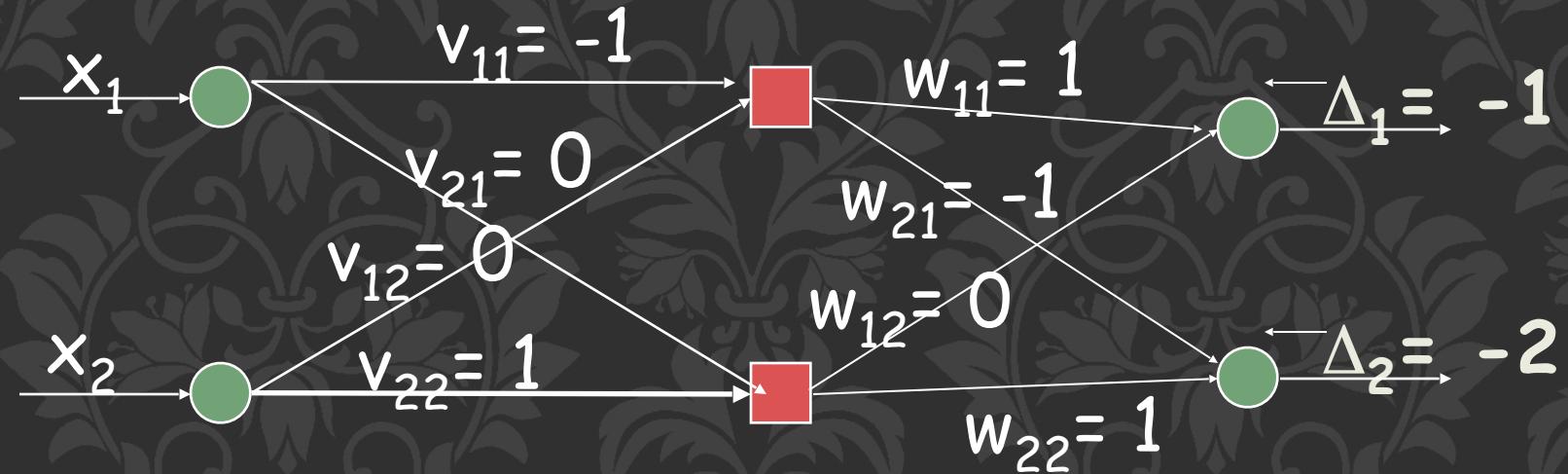
Calculate 2nd layer outputs (weighted sum thru activation functions):



$$y_1 = a_1 = 1 \times 1 + 0 \times 2 + 1 = 2$$

$$y_2 = a_2 = -1 \times 1 + 1 \times 2 + 1 = 2$$

Backward pass:



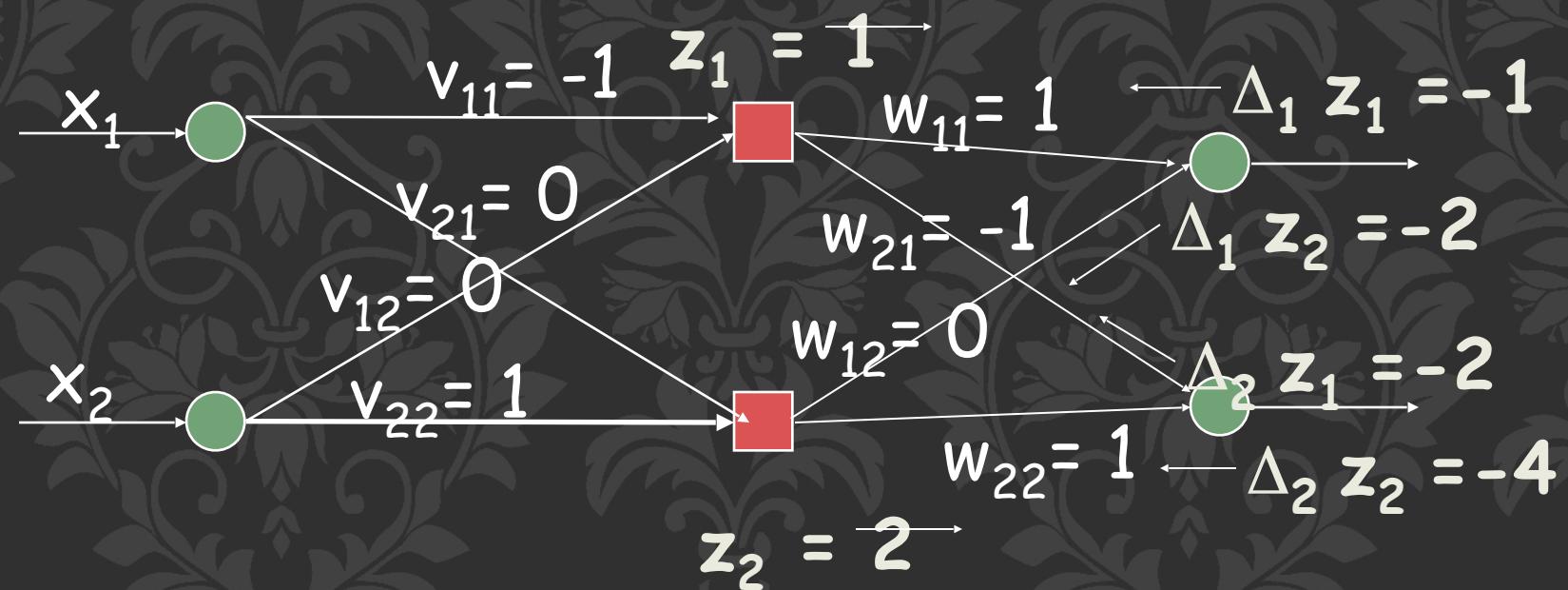
Target = [1, 0] so $d_1 = 1$ and $d_2 = 0$

So:

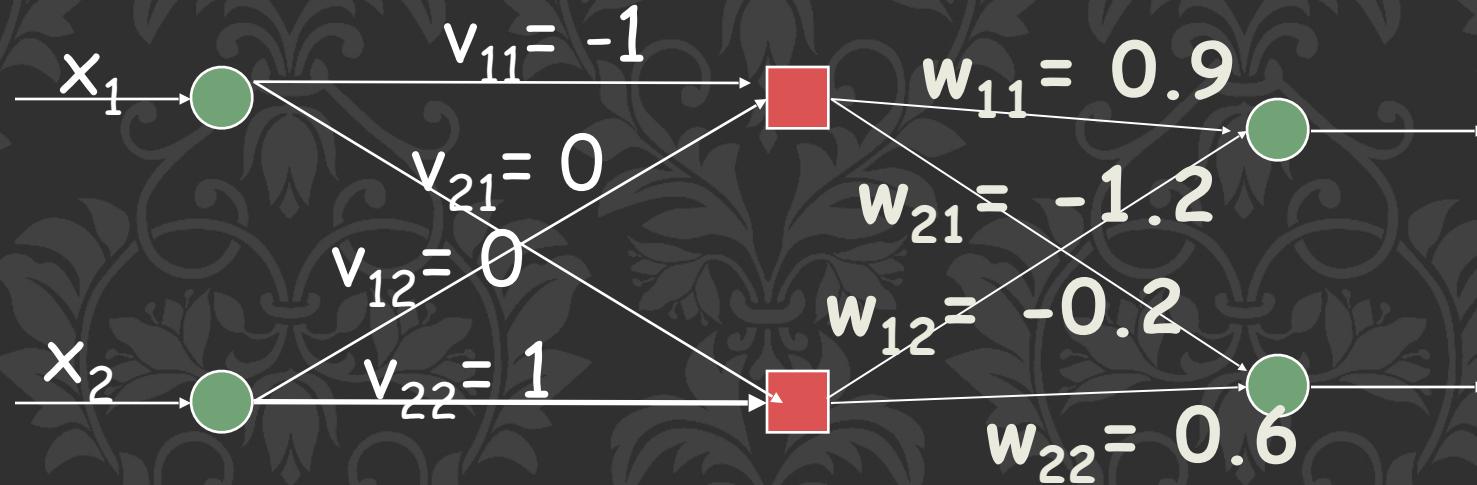
$$\Delta_1 = (d_1 - y_1) = 1 - 2 = -1$$

$$\Delta_2 = (d_2 - y_2) = 0 - 2 = -2$$

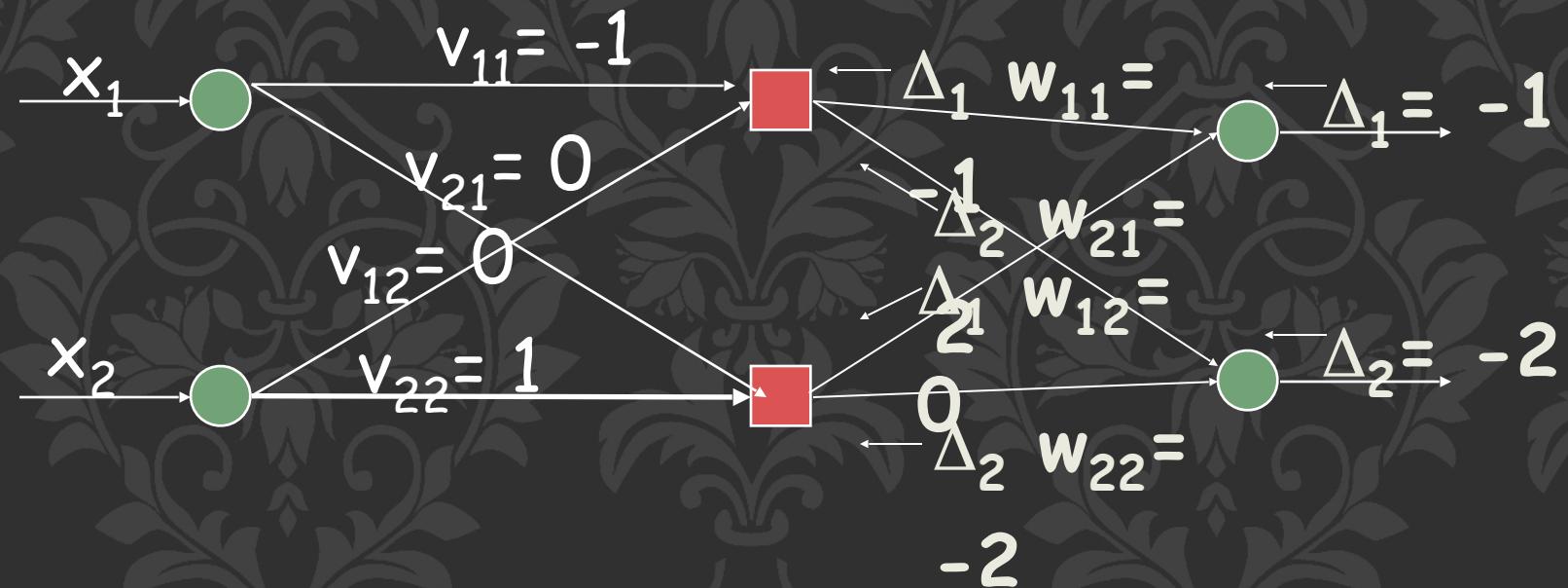
Calculate weight changes for 1st layer (cf perceptron learning):



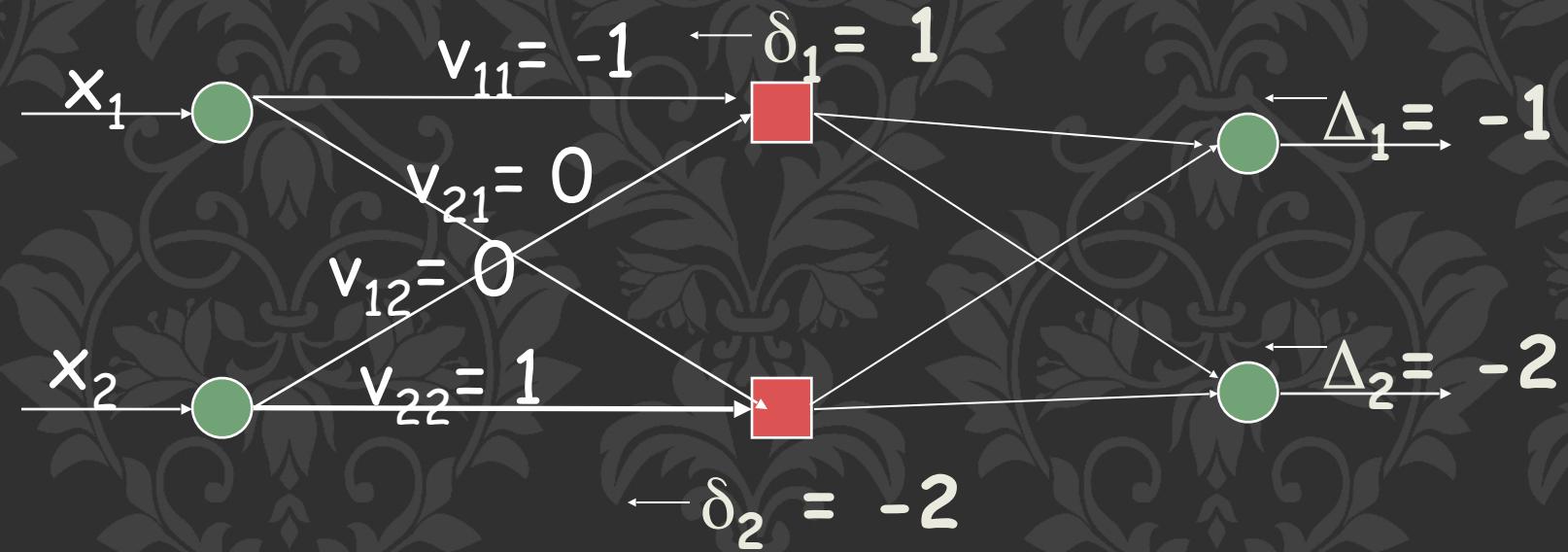
Weight changes will be:



But first must calculate δ 's:



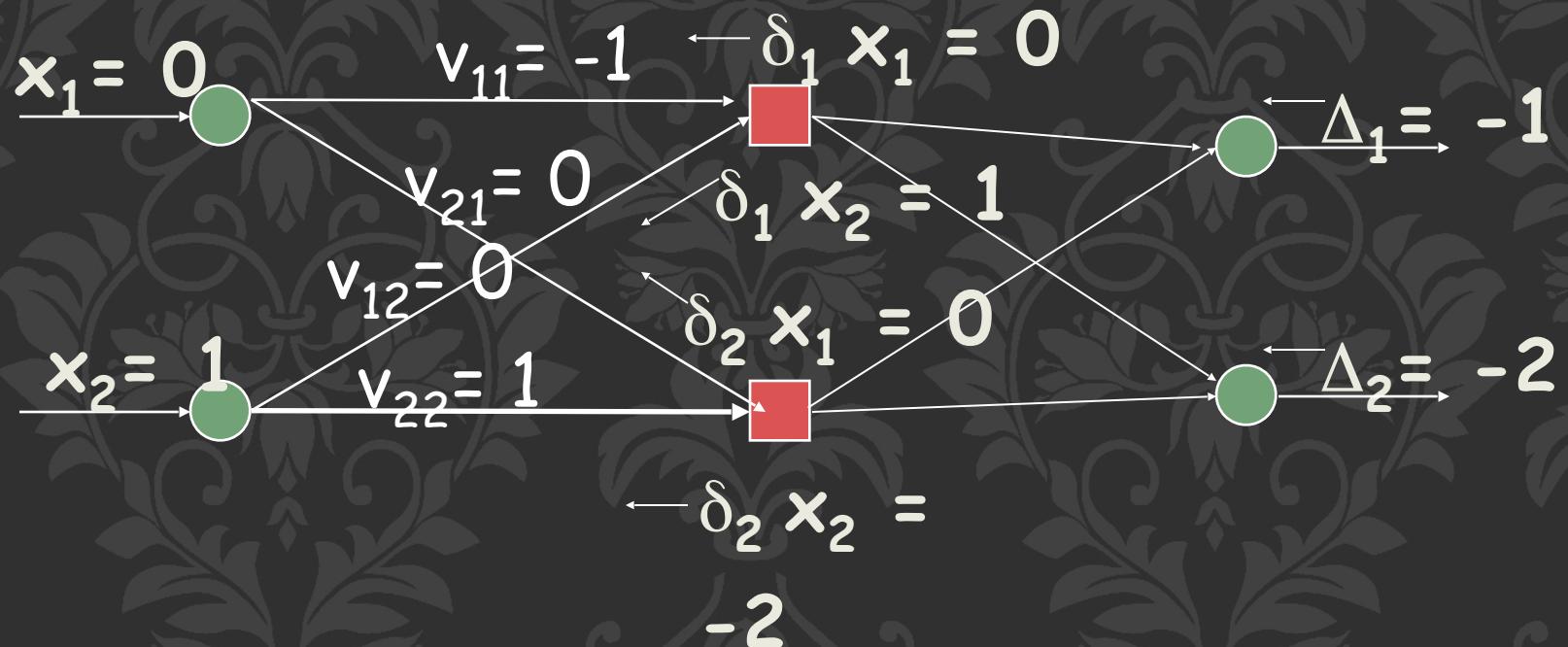
Δ 's propagate back:



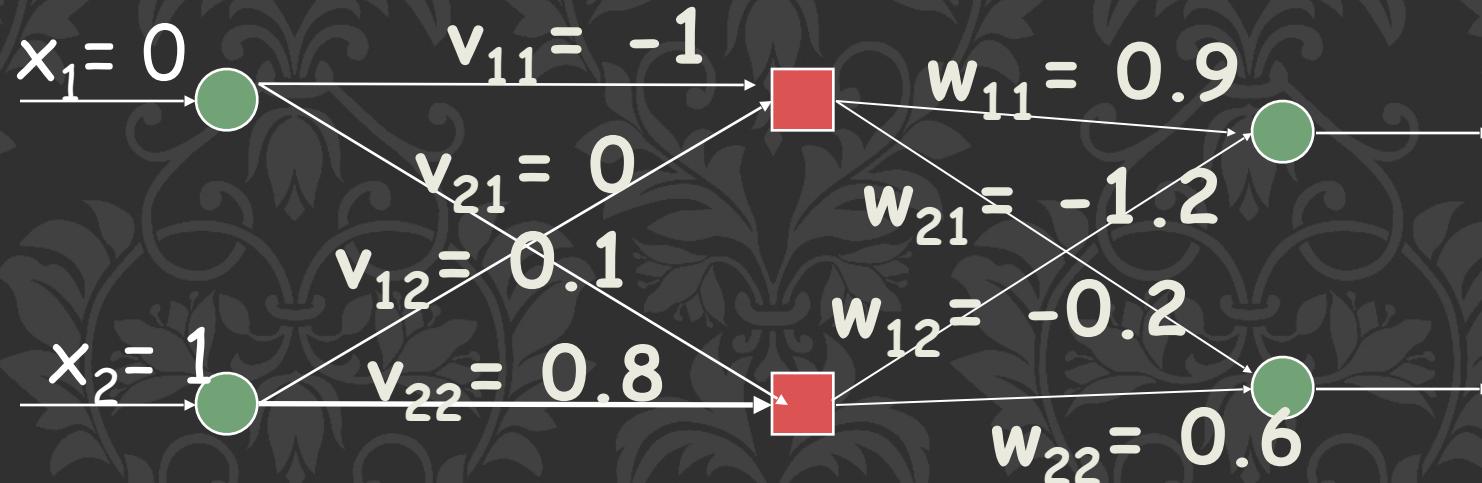
$$\delta_1 = -1 + 2 = 1$$

$$\delta_2 = 0 - 2 = -2$$

And are multiplied by inputs:

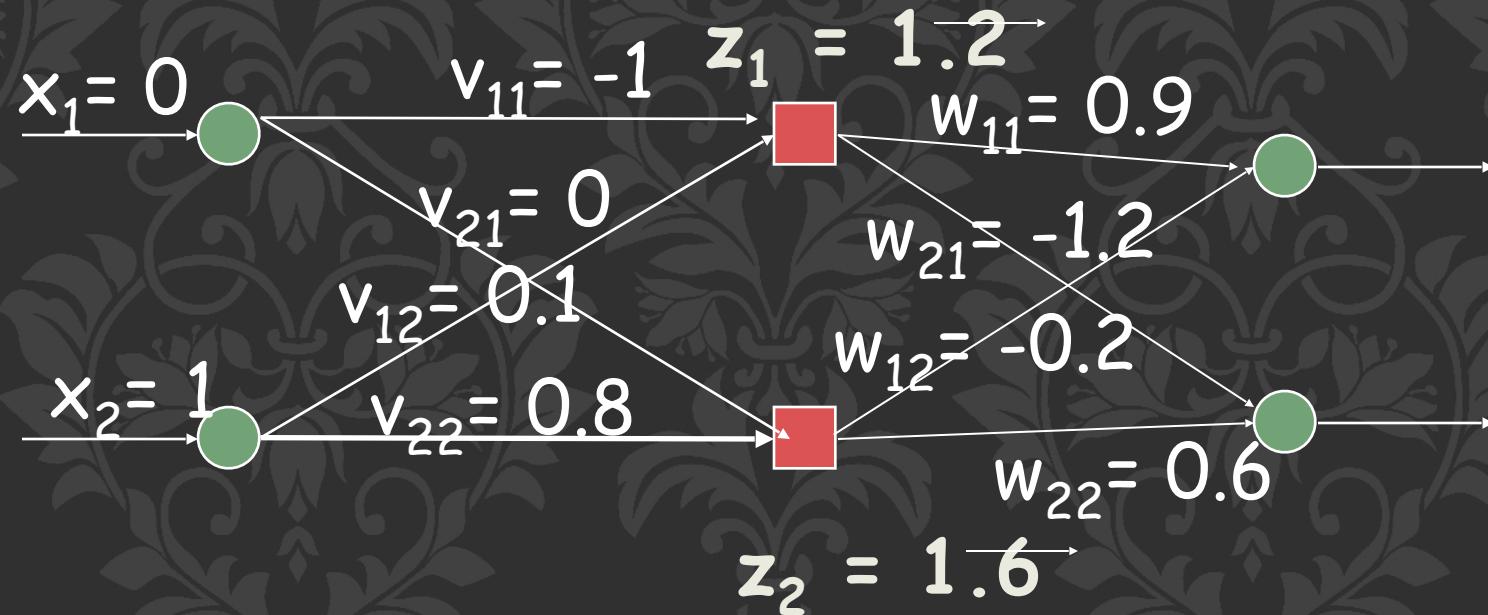


Finally change weights:

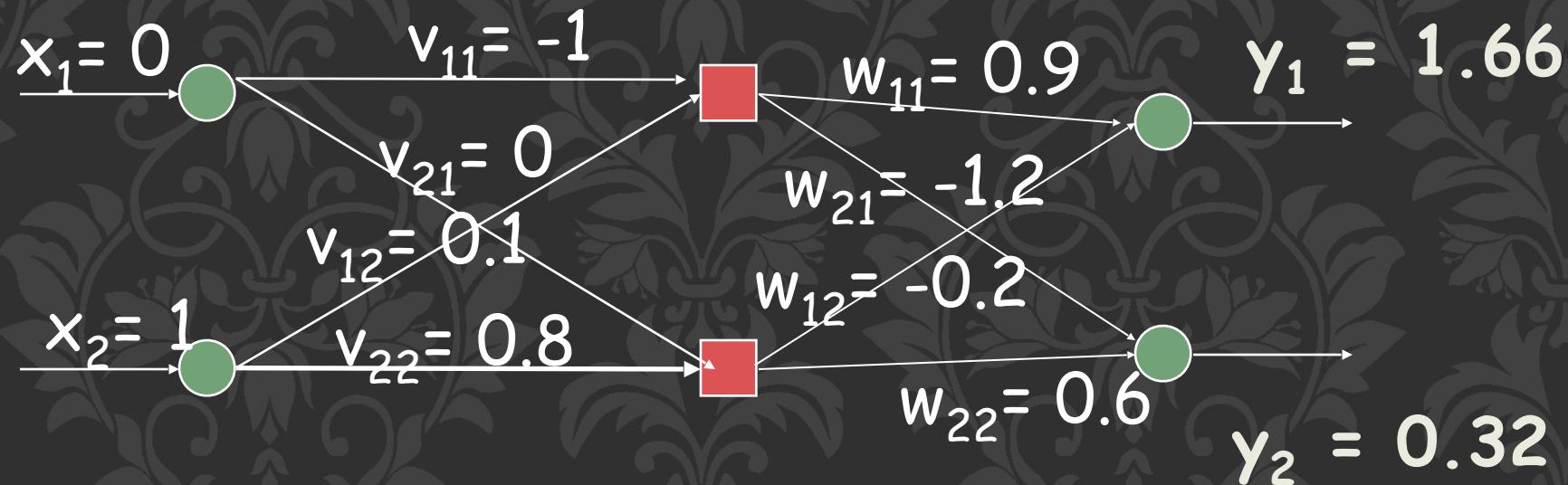


Note that the weights multiplied by the zero input are unchanged as they do not contribute to the error

Now go forward again (would normally use a new input vector):



Now go forward again (would normally use a new input vector):



Outputs now closer to target value [1, 0]