Neural Networks

CS 446 Machine Learning

Dan is traveling

Administrative

Midterms were shown yesterday, and will be shown tomorrow

Once you leave with your midterms, we don’t take any regrade requests

Gradient

What is gradient of a function $f(x)$

$$\nabla f(x) = \frac{df}{dx}$$

Rate of change of function $f(x)$
with respect to input

What is gradient of multiple variable function

$$f(x_1, x_2, x_3, \dots, x_n)$$

$$\nabla f(x_1, x_2, \dots, x_n) = \left[\frac{\delta f}{\delta x_1}, \frac{\delta f}{\delta x_2}, \frac{\delta f}{\delta x_3}, \dots, \frac{\delta f}{\delta x_n}\right]$$

This is a vector. What is the magnitude ? What is the direction ?

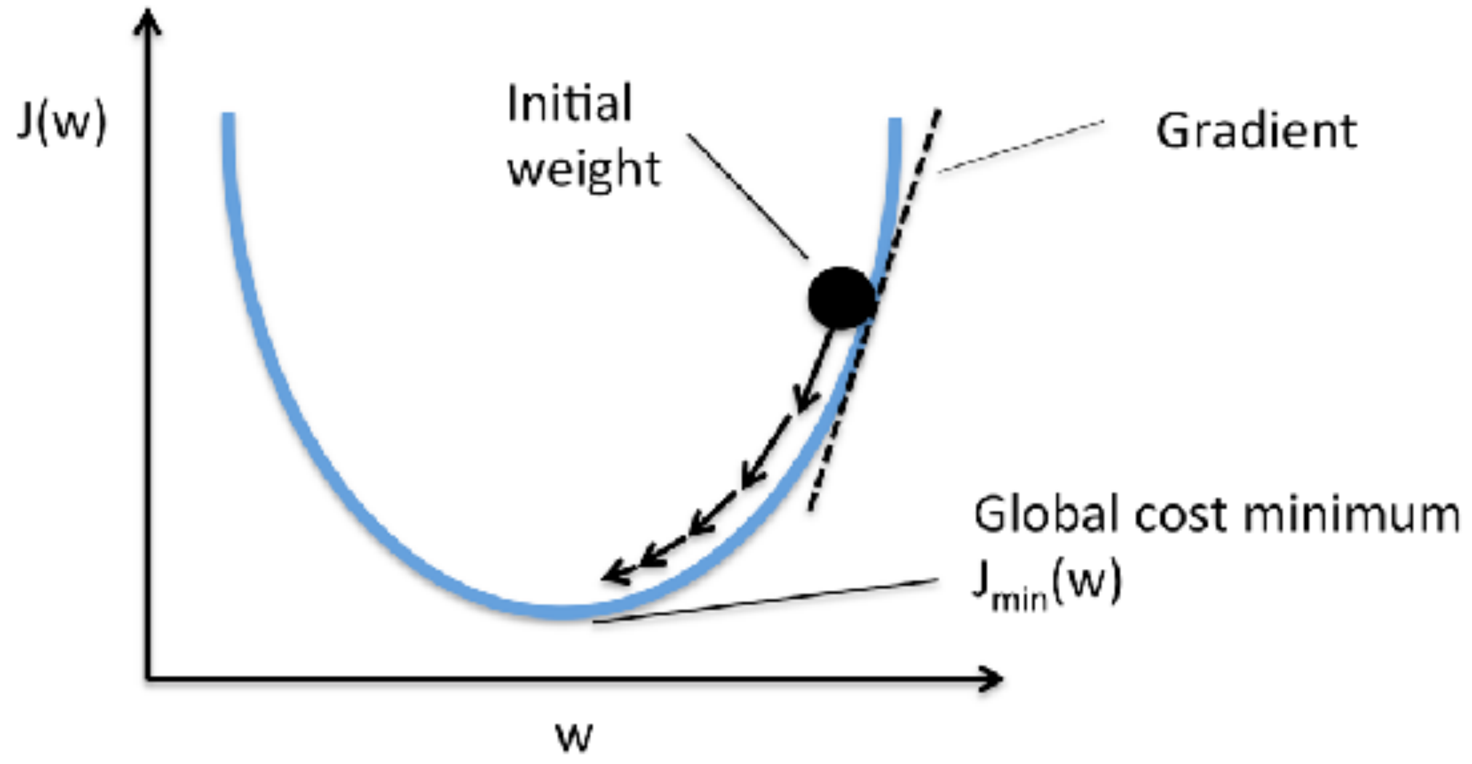
Direction : direction of steepest increase
Magnitude : rate of change in that direction

Gradient Descent

Since gradient vector gives me direction of steepest increase, we should go in the reverse direction to get lower function value

Update : $\boxed{\mathrm{x} \leftarrow \mathrm{x} - R\nabla f(\mathrm{x})}$ where $\mathrm{x} = [x_1, x_2, \ldots, x_n]$

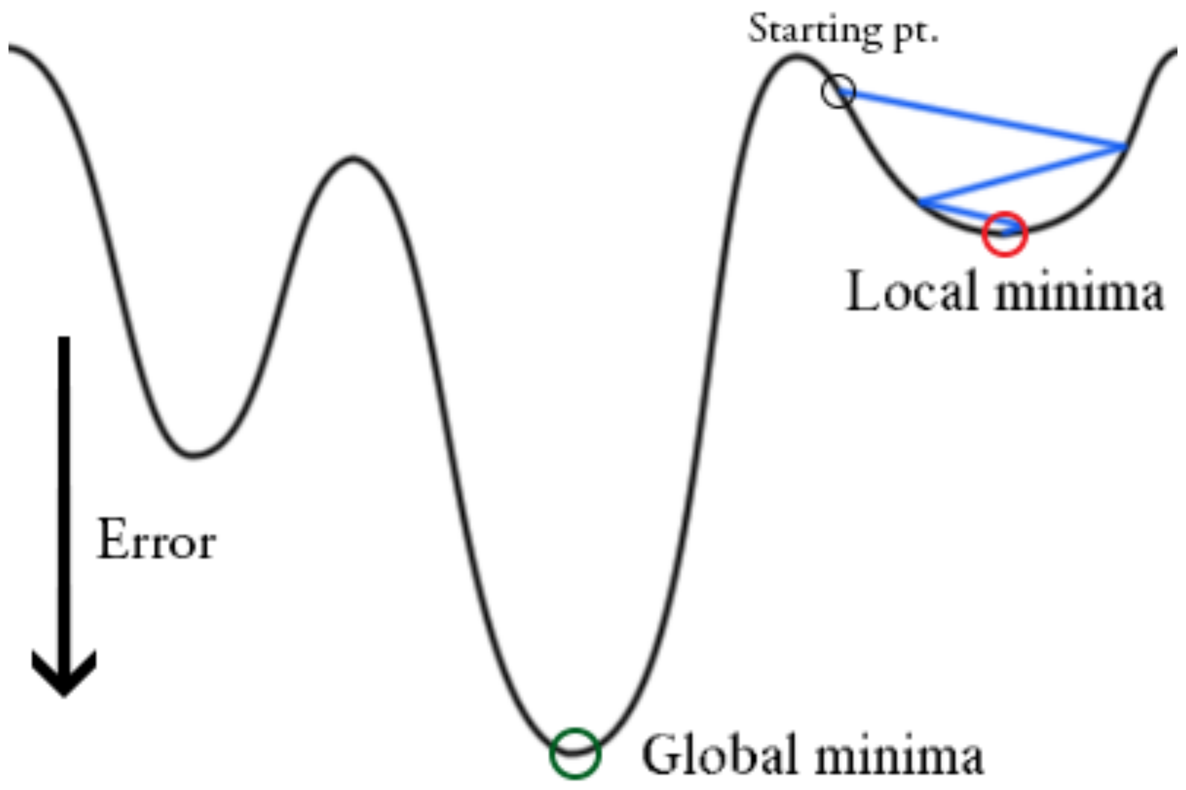
For each element, we have $\boxed{x_i \leftarrow x_i - R\frac{\delta f}{\delta x_i}}$



Works well when you have convex functions, guaranteed to reach global minima

Gradient Descent

What happens when function is non-convex ?



Ohh no ! Algorithm gets stuck in local minima

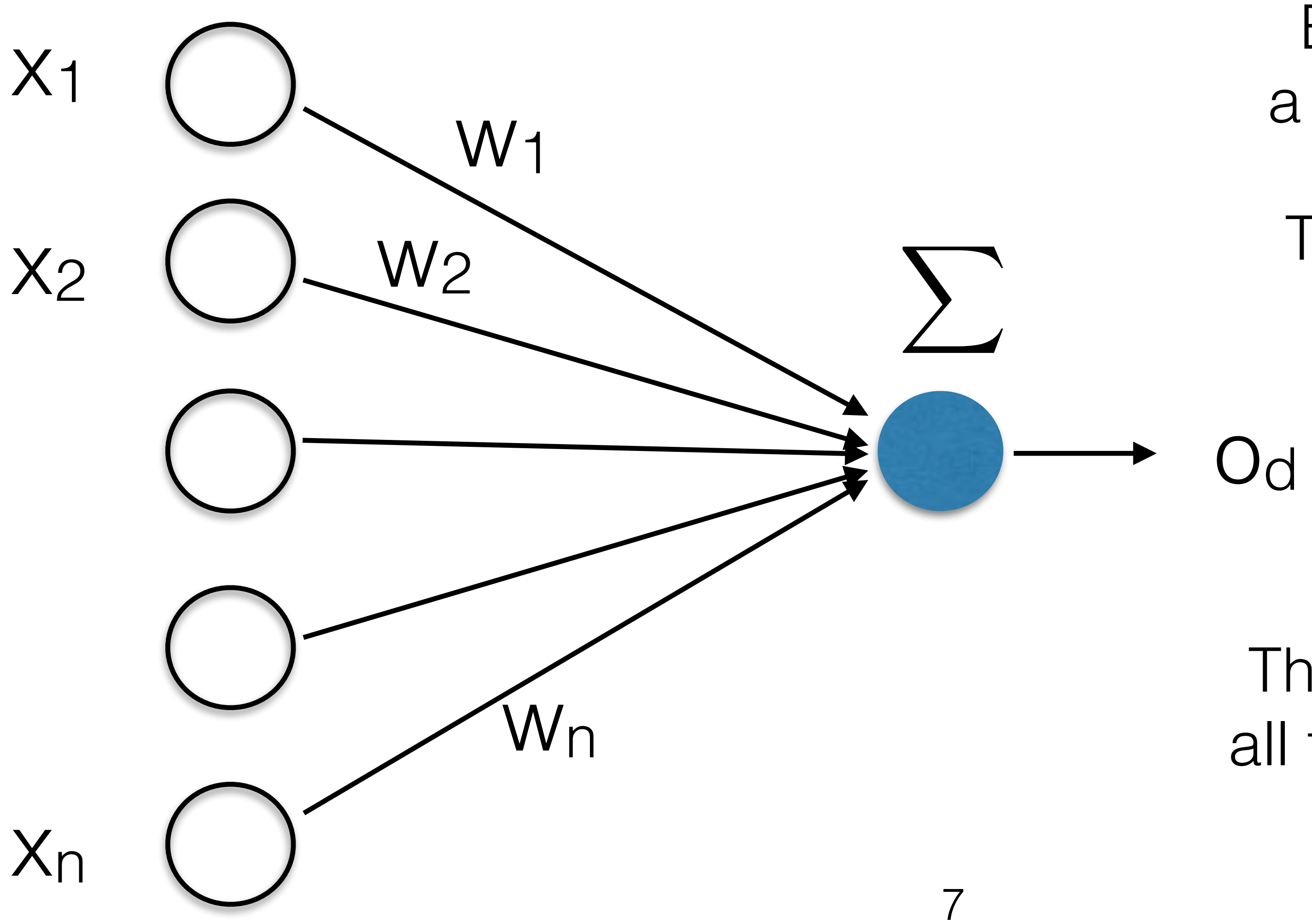
Till now, you have applied gradient descent on convex functions, today we will apply it on non-convex problem

A different view of LMS

For dth example $$o_d = \sum_i w_i x_i - T$$

Define error $$Err(\mathrm{w}) = \sum_d (o_d - t_d)^2$$

Perform gradient descent updates $$\mathrm{w} = \mathrm{w} - R\nabla Err(\mathrm{w})$$



Each input **x_i** is connected to a unit (blue circle) by weight **w_i**

Think of the input to unit from **ith** connection to be **w_i x_i**

The unit (blue circle) is summing all these inputs and outputting the sum

Expressivity by Depth

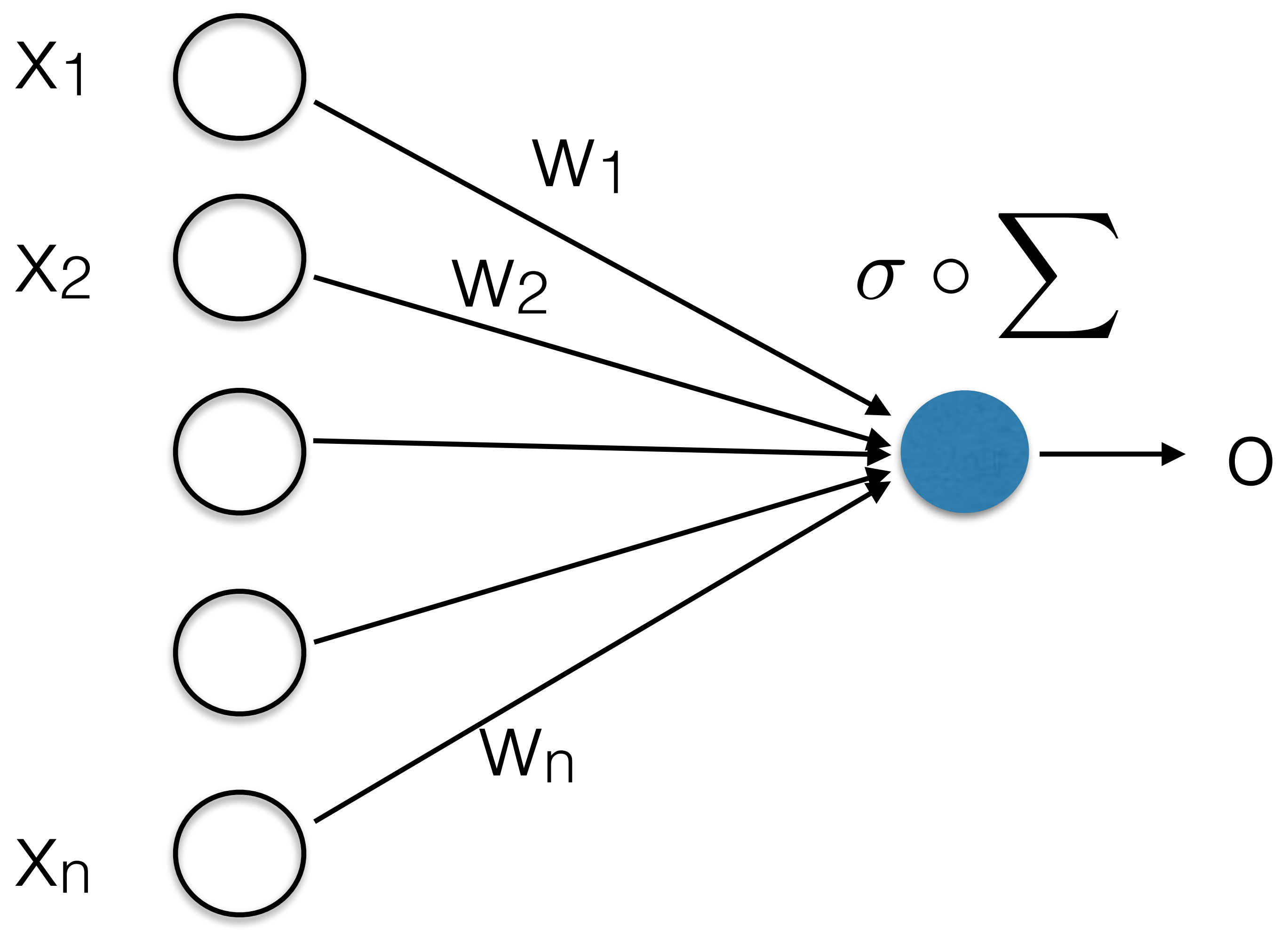
However linear threshold units like last slide can only deal with linear functions of input **x**

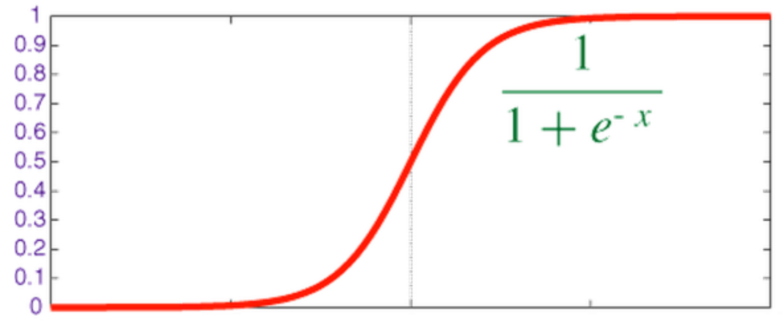
We now want to learn **non-linear functions** of the input. You already know some methods to learn non-linear functions of the input. Can you name them ?

Decision trees, kernels

Neural networks another way of doing that, **stack** several layers of threshold elements, each layer using the output of the previous layer as input

Basic Unit : Neuron





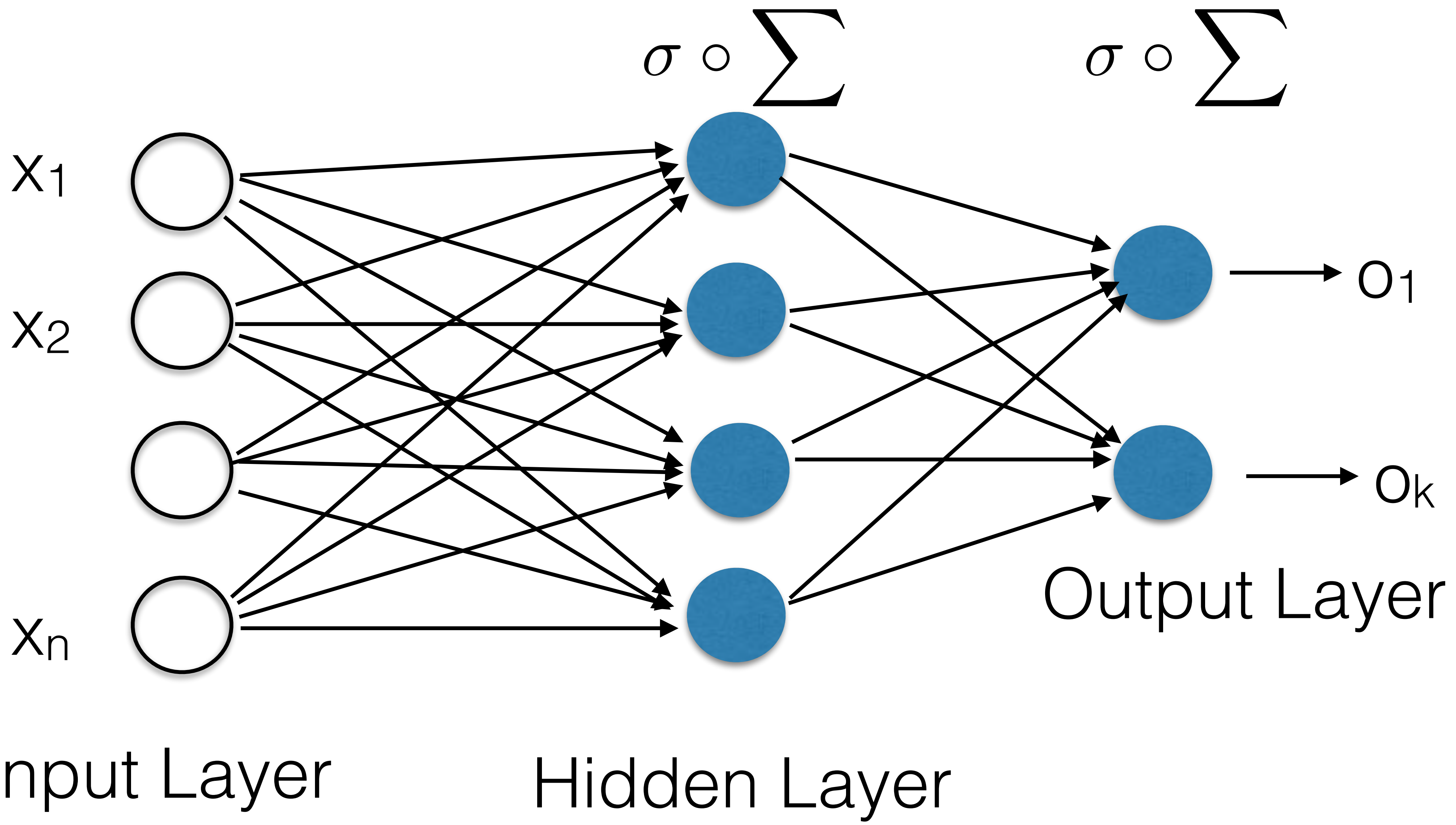
$$\sigma(x) = \frac{1}{1+e^{-x}}$$

$$o = \sigma(\sum_i w_i x_i - T)$$

Lets define a variable called **net** $$net = \sum_i w_i x_i$$

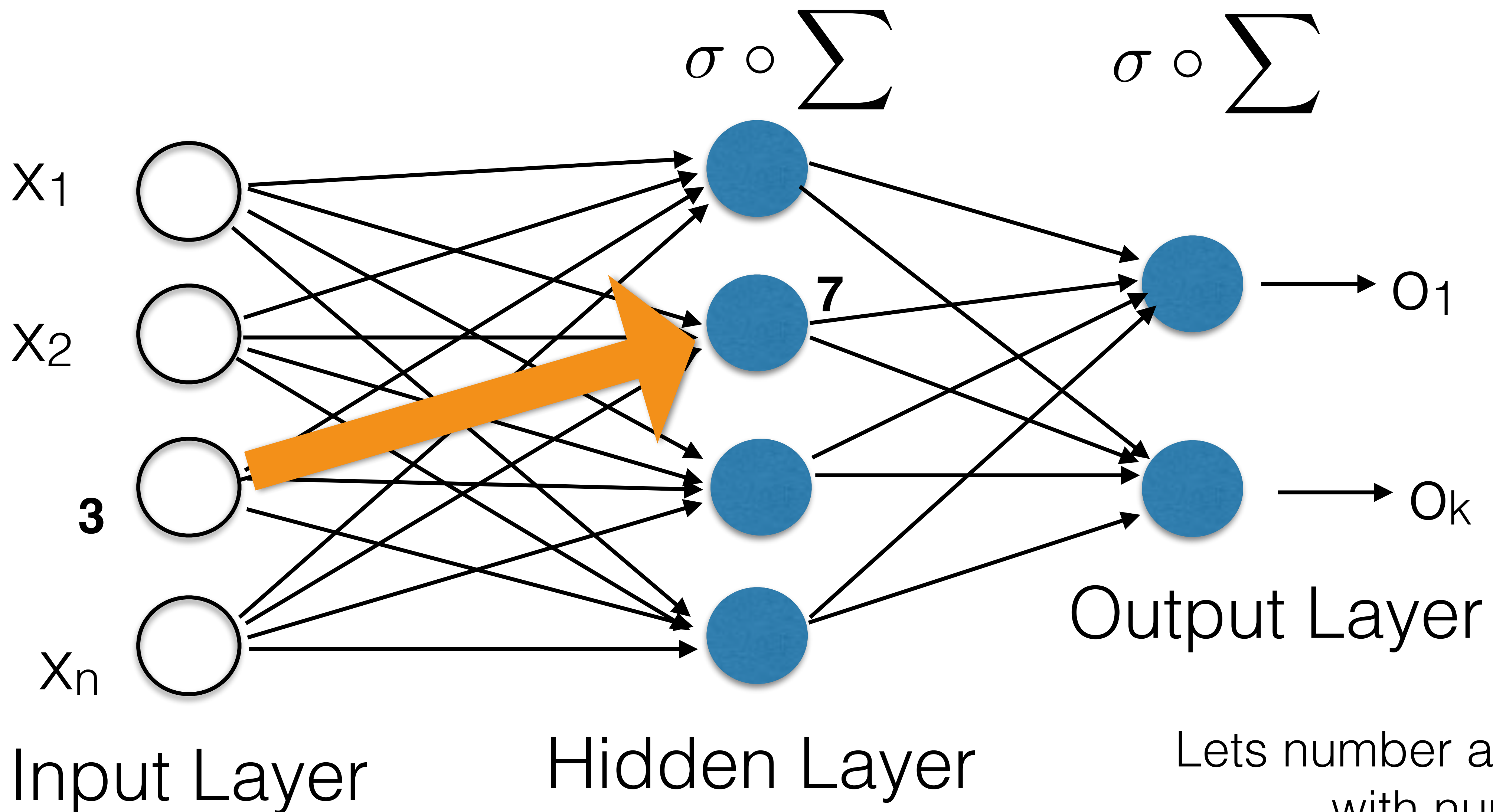
Then we have $$o = \sigma(net - T)$$

Stack them up



Feedforward Neural Network

Feedforward Neural Network

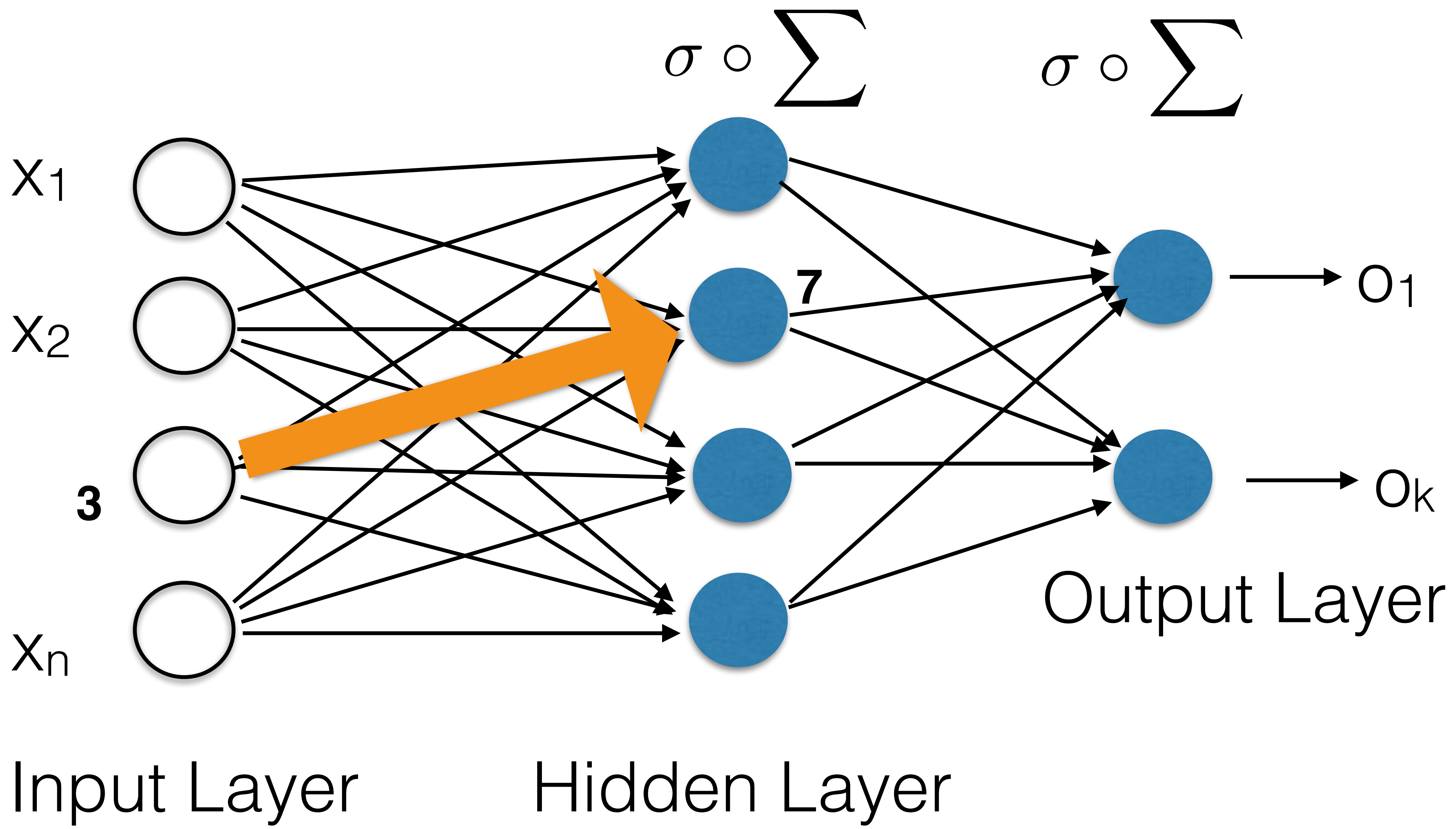


Lets number all circles / nodes with number ids.

Connection between node i and node j is denoted as $\mathbf{w}_{ij}$, net input to node j is $\mathbf{net}_j$, output from node j is $\mathbf{x}_j$ or $\mathbf{o}_j$

Connection (yellow arrow) between node 3 and node 7 is denoted by $\mathbf{w}_{37}$
Output of node 7 is $\mathbf{x}_7$ or $\mathbf{o}_7$
Net Input to node 7 is $\mathbf{net}_7$

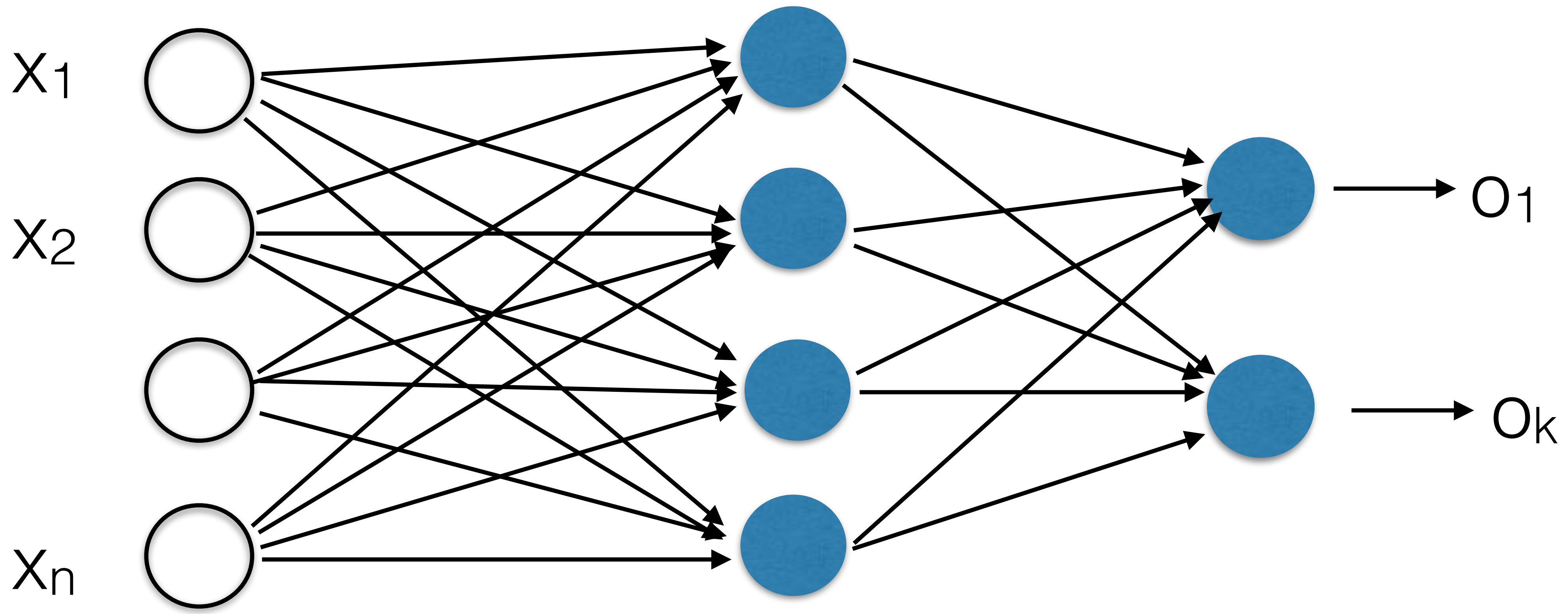
Feedforward Neural Network



For kth output node

$$o_k = \sigma\left(\sum_{j \in hidden} w_{jk} \quad \sigma\left(\sum_{i \in input} w_{ij} x_i - T_j \right) - T_k \right)$$

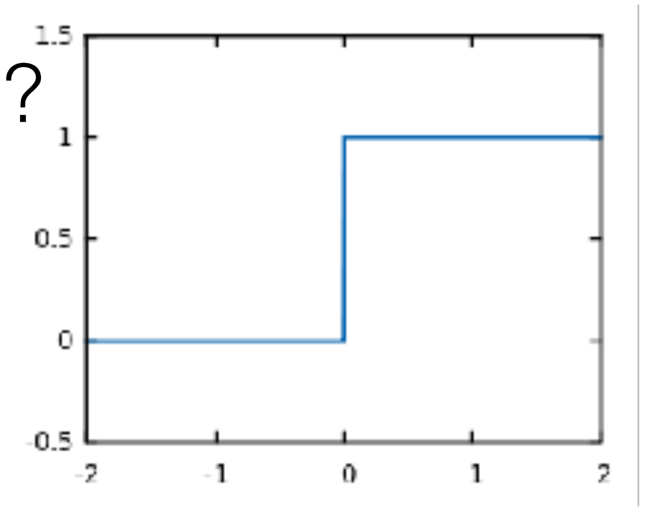
Alternatives



What if we use identity function (like LMS example) instead of sigmoid function for activation ?

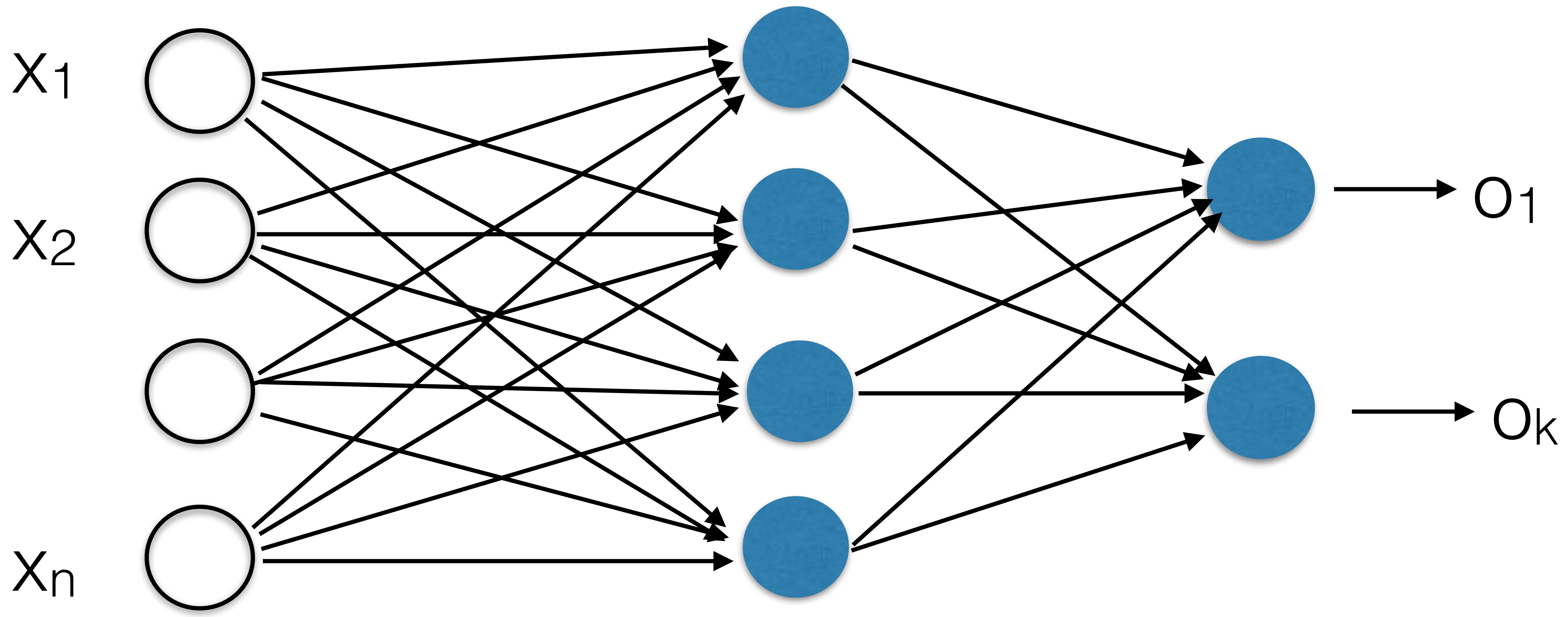
Becomes a Linear Model

What if we use step function (like Perceptron) then ?

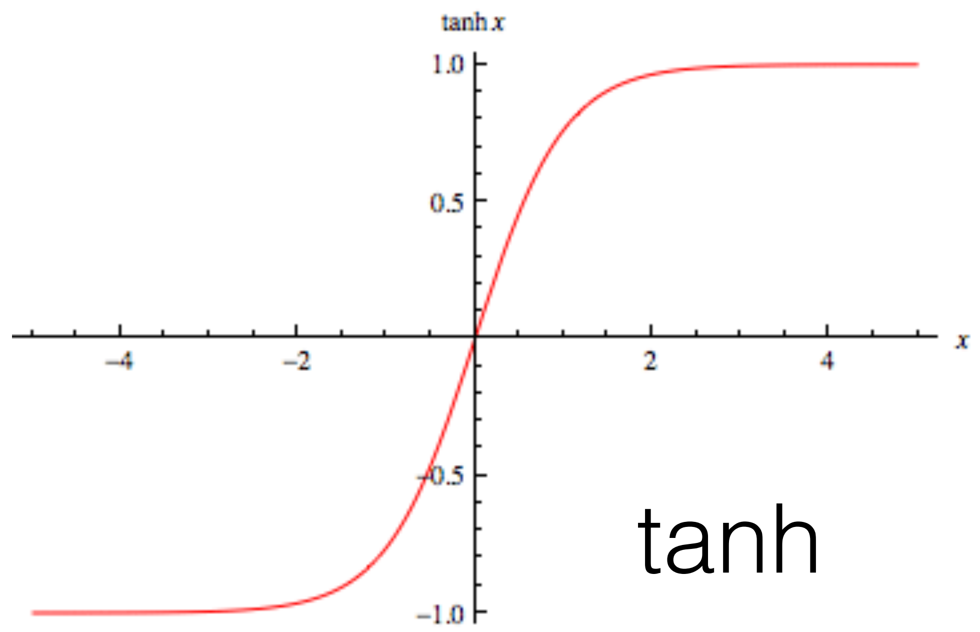


Non-differentiable, not suitable for gradient descent

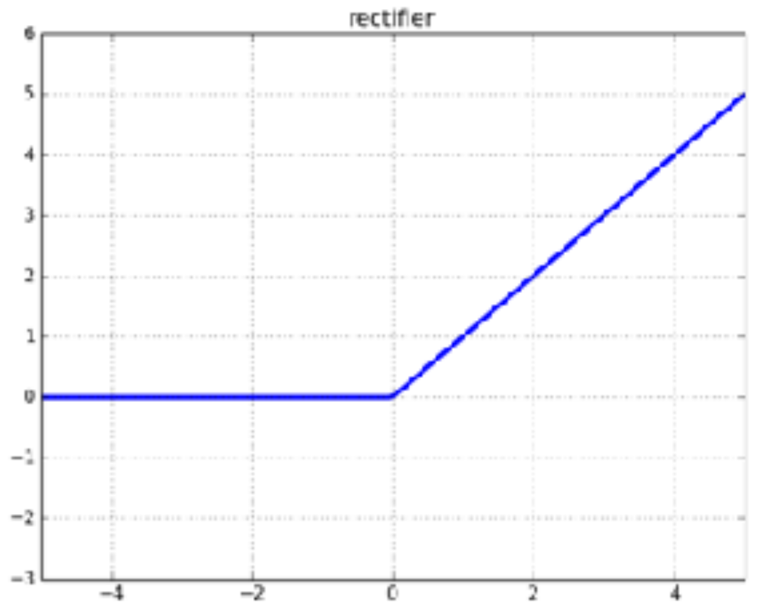
Alternatives



However there are other non-linear differentiable functions you can use

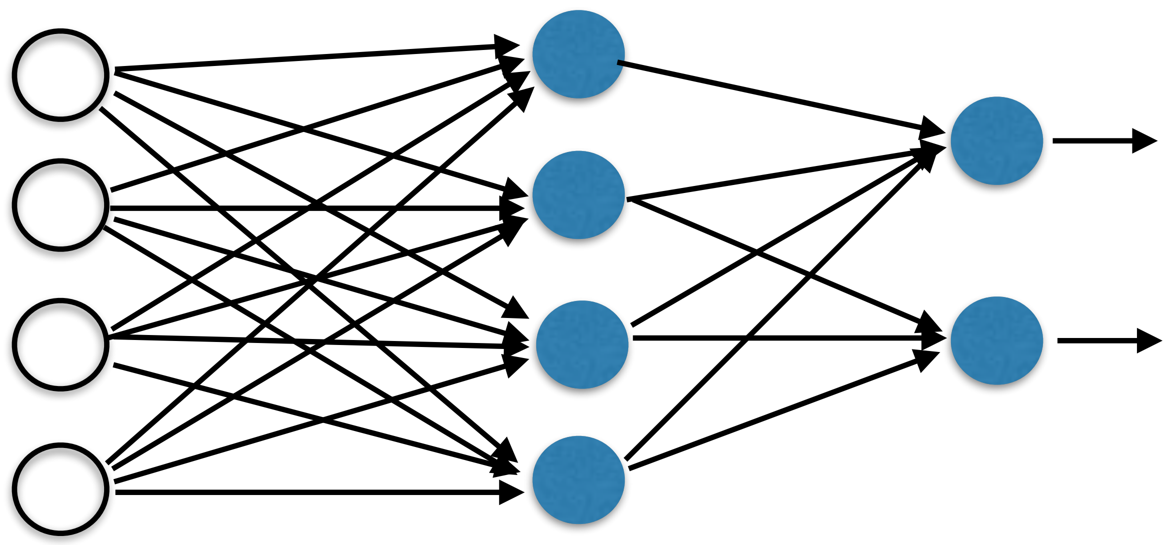


tanh



relu

Error

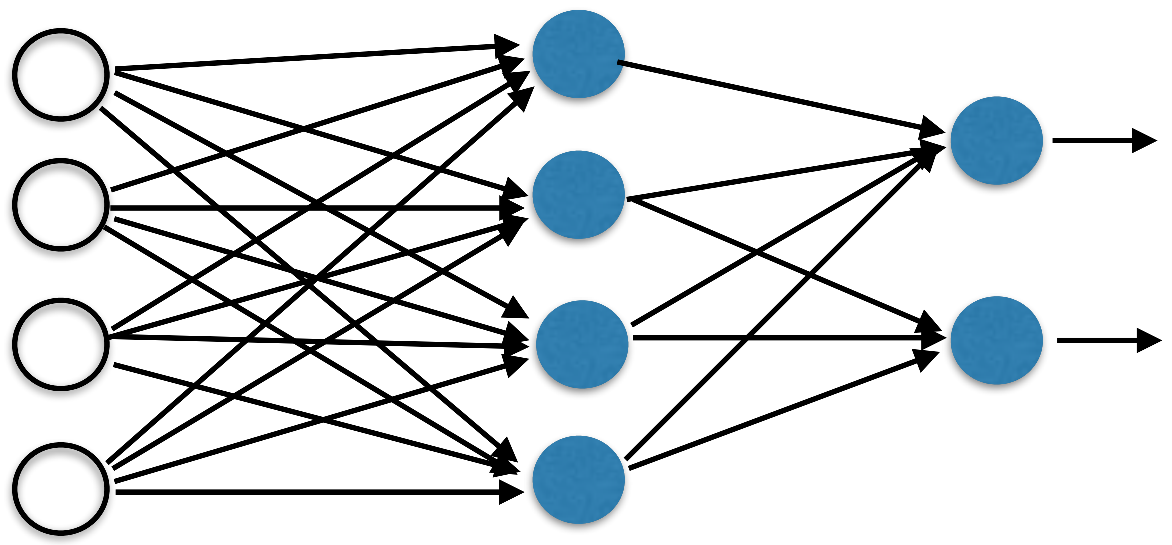


We will use squared error

$$Err(\mathrm{w}) = \frac{1}{2}\sum_{d\in D}\sum_{k\in K}(t_{kd} - o_{kd})^2$$

Here D is the set of training examples, and K is the set of output units

Remember Gradient Descent



$$\mathrm{w} = \mathrm{w} - R\nabla Err(\mathrm{w})$$

In order to compute $\nabla Err(\mathrm{w})$

You need to compute partial derivatives w.r.t each element in **w** above

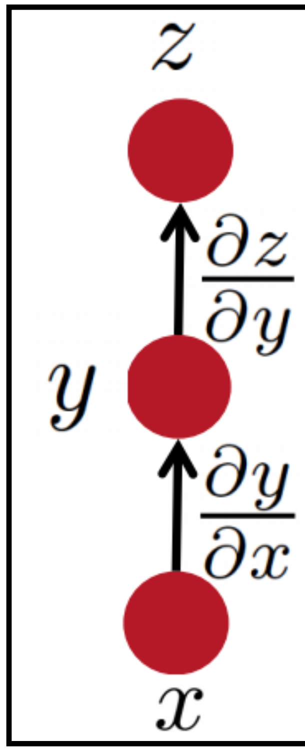
Whats **w** here ? What are the parameters of the model you want to train ?

$\boxed{\text{Connection weights } w_{ij}}$ $\boxed{\text{Thresholds at each unit } T_i}$

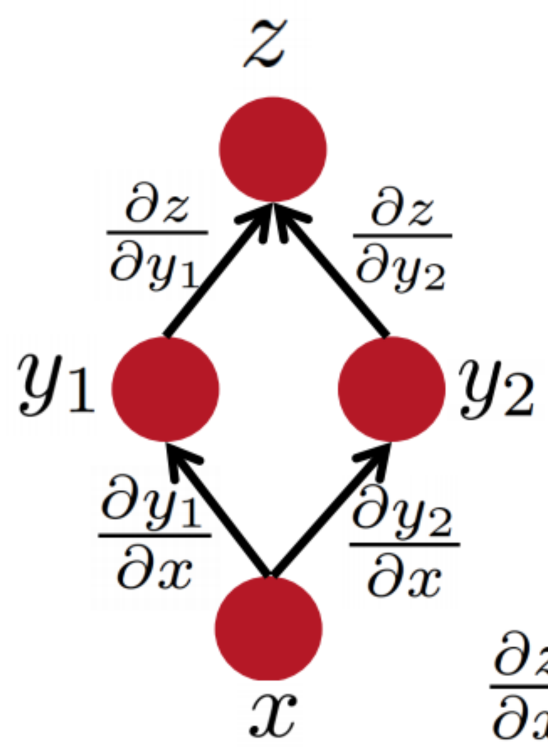
Do you remember derivatives ?

Function	Derivative
$y = \frac{1}{2}\sum_{k\in K}(c_k - x_k)^2$	$\frac{\partial y}{\partial x_i} = -(c_i - x_i)$
$y = \sum w_i . x_i$	$\frac{\partial y}{\partial w_i} = x_i$
$y = \frac{1}{1+\exp\{-(x-T)\}}$	$\frac{\partial y}{\partial x} = \frac{\exp\{-(x-T)\}}{(1+\exp\{-(x-T)\})^2} = y(1-y)$

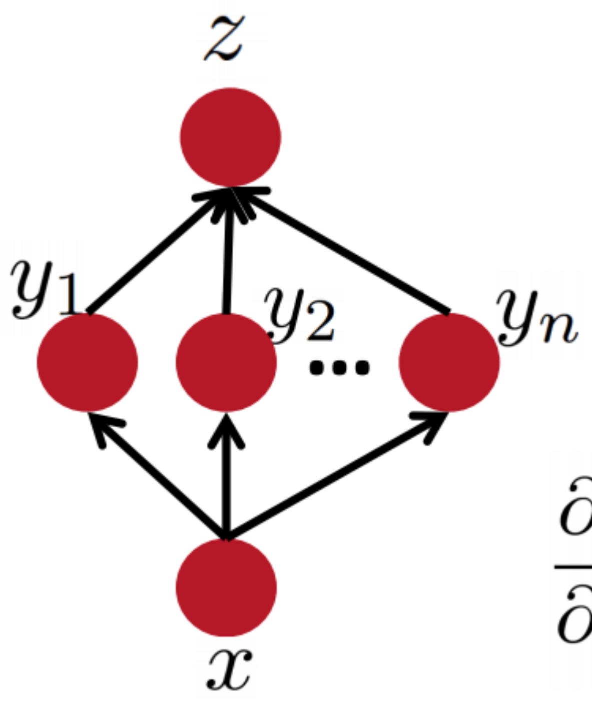
Some facts from real analysis


z
$\frac{\partial z}{\partial y}$
y
$\frac{\partial y}{\partial x}$
x

$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y}\frac{\partial y}{\partial x}$$

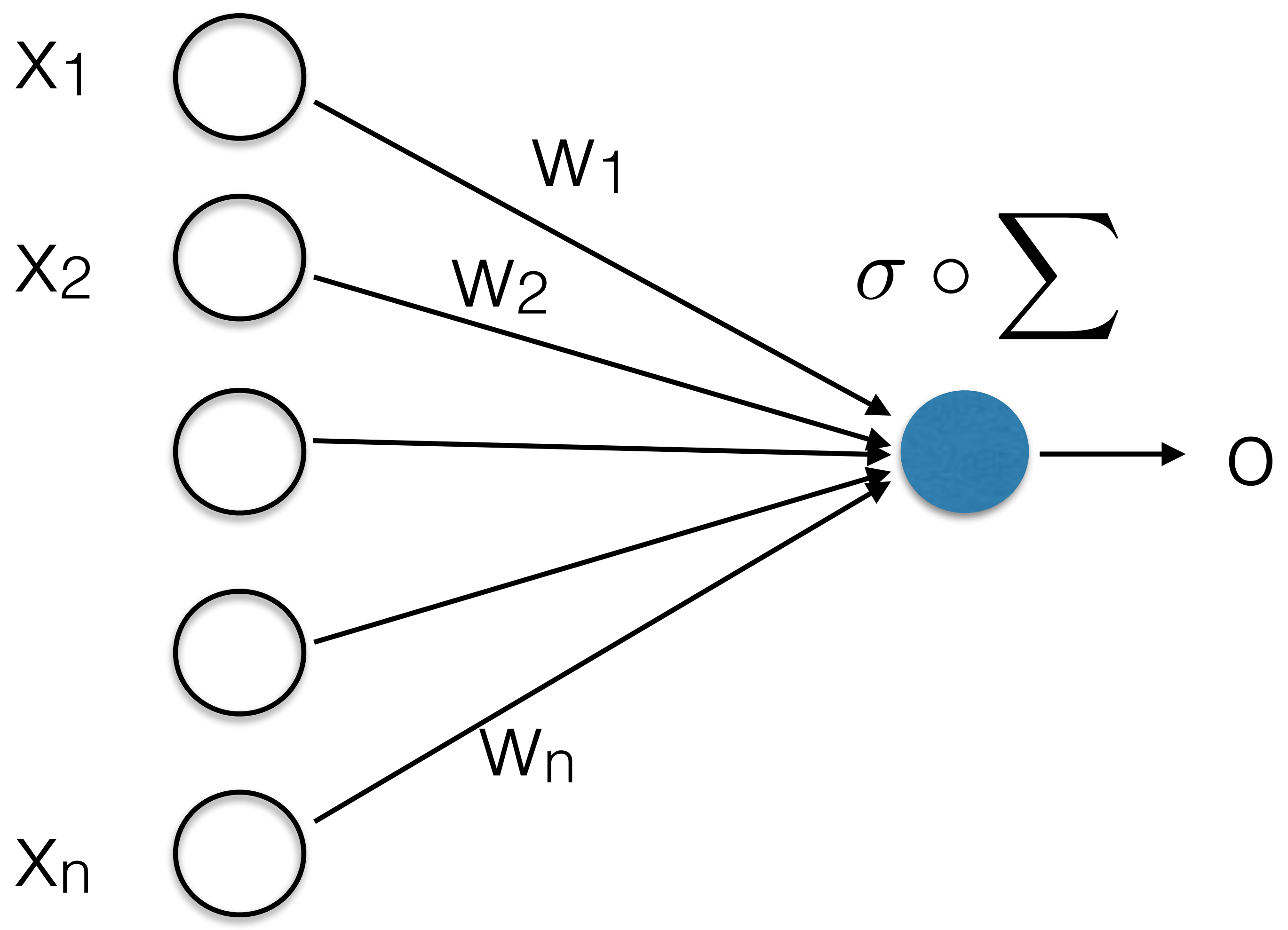

z
$\frac{\partial z}{\partial y_1}$
$\frac{\partial z}{\partial y_2}$
y_1
y_2
$\frac{\partial y_1}{\partial x}$
$\frac{\partial y_2}{\partial x}$
x

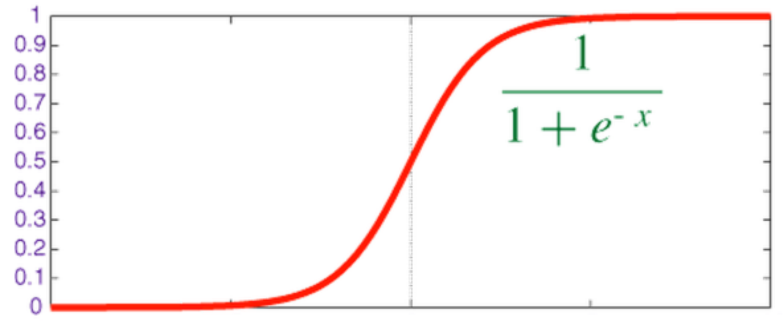
$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y_1}\frac{\partial y_1}{\partial x} + \frac{\partial z}{\partial y_2}\frac{\partial y_2}{\partial x}$$



$$\frac{\partial z}{\partial x} = \sum_{i=1}^{n} \frac{\partial z}{\partial y_i}\frac{\partial y_i}{\partial x}$$

Reminder : Neuron





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

$$o = \sigma(\sum_i w_i x_i - T)$$

Lets define a variable called **net** you can think of it as net input to a unit

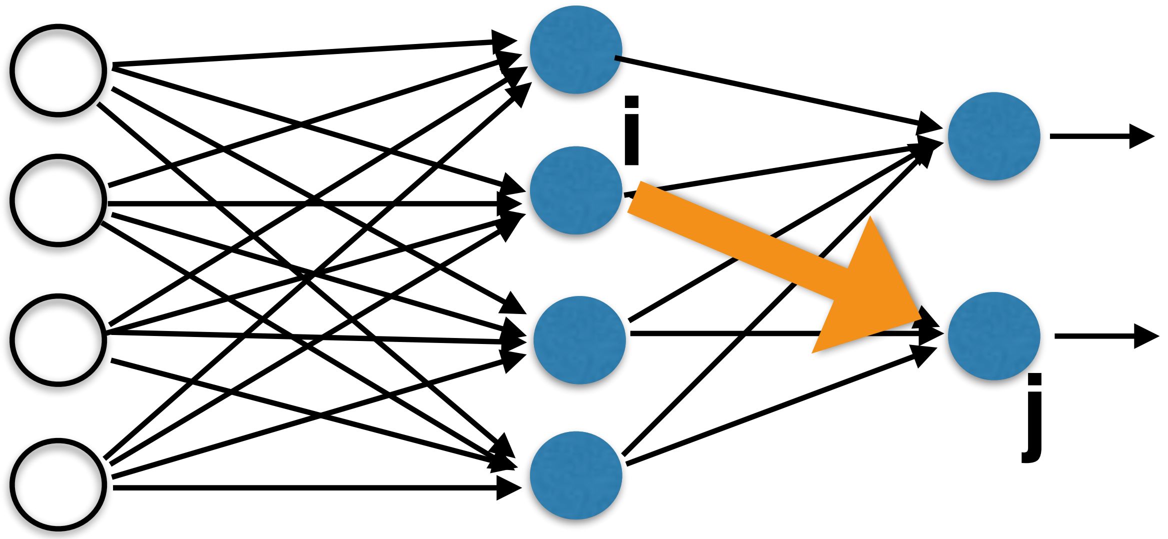
$$net = \sum_i w_i x_i$$

Then we have

$$o = \sigma(net - T)$$

ARE YOU READY
TO COMPUTE SOME PARTIAL DERIVATIVES ?
memegenerator.net

Derivation of Learning Rule



Connection $\mathbf{w_{ij}}$ between hidden and output layer (influences output only through $\mathbf{net_j}$)

$$\frac{\partial E_d}{\partial w_{ij}} = \frac{\partial E_d}{\partial \text{net}_j} \frac{\partial \text{net}_j}{\partial w_{ij}} = \frac{\partial E_d}{\partial o_j} \frac{\partial o_j}{\partial \text{net}_j} \frac{\partial \text{net}_j}{\partial w_{ij}}$$

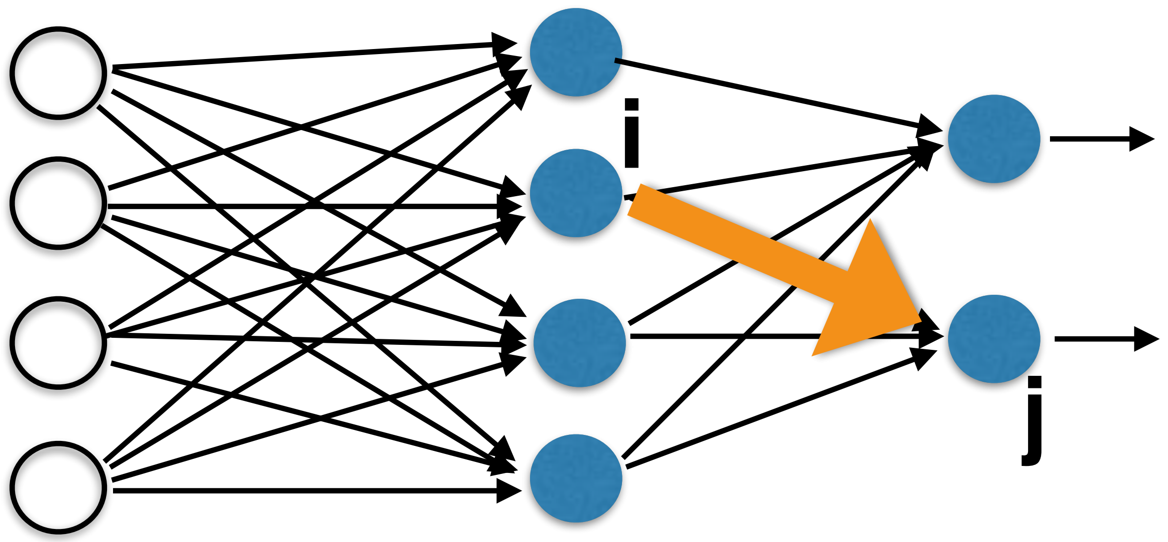
$$= -(t_j - o_j)\ o_j(1 - o_j)\ x_i$$

$$\boxed{Err_d(\vec{w}) = \frac{1}{2}\sum_{k \in K}(t_k - o_k)^2}$$

$$\boxed{\begin{gathered}\frac{\partial o_j}{\partial \text{net}_j} = o_j(1 - o_j) \\ o_j = \frac{1}{1 + \exp\{-(\text{net}_j - T_j)\}}\end{gathered}}$$

$$\boxed{net_j = \sum w_{ij} x_i}$$

Derivation of Learning Rule



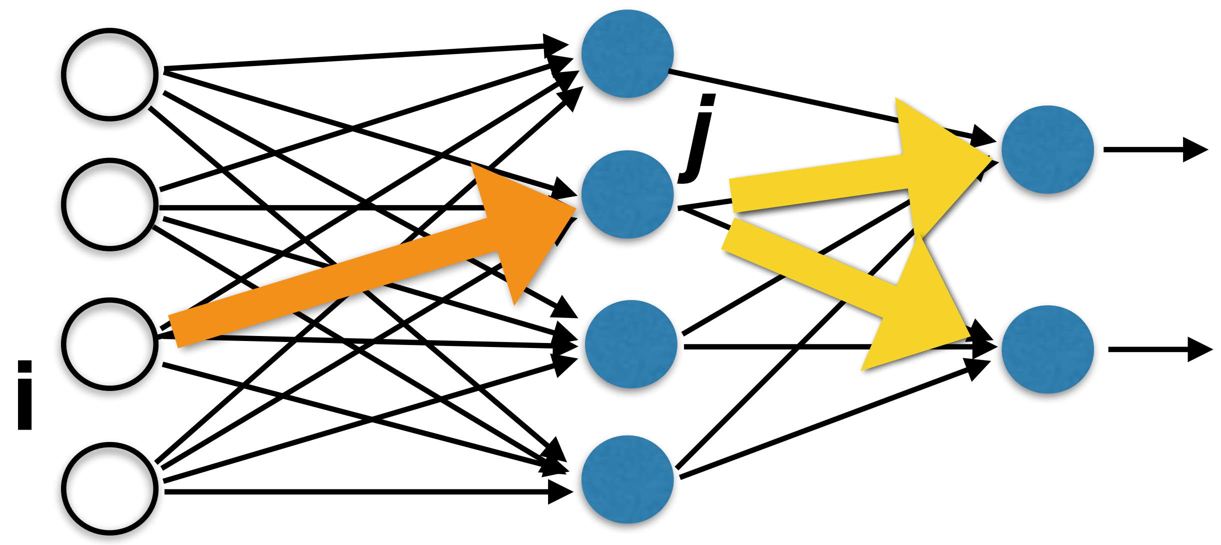
Connection **w$_{ij}$** between hidden and output layer is updated as

$$\Delta w_{ij} = R(t_j - o_j)o_j(1 - o_j)\, x_i$$
$$= R\delta_j\, x_i$$

where

$$\delta_j = (t_j - o_j)o_j(1 - o_j) \quad = -\frac{\delta E_d}{\delta net_j}$$

Derivation of Learning Rule

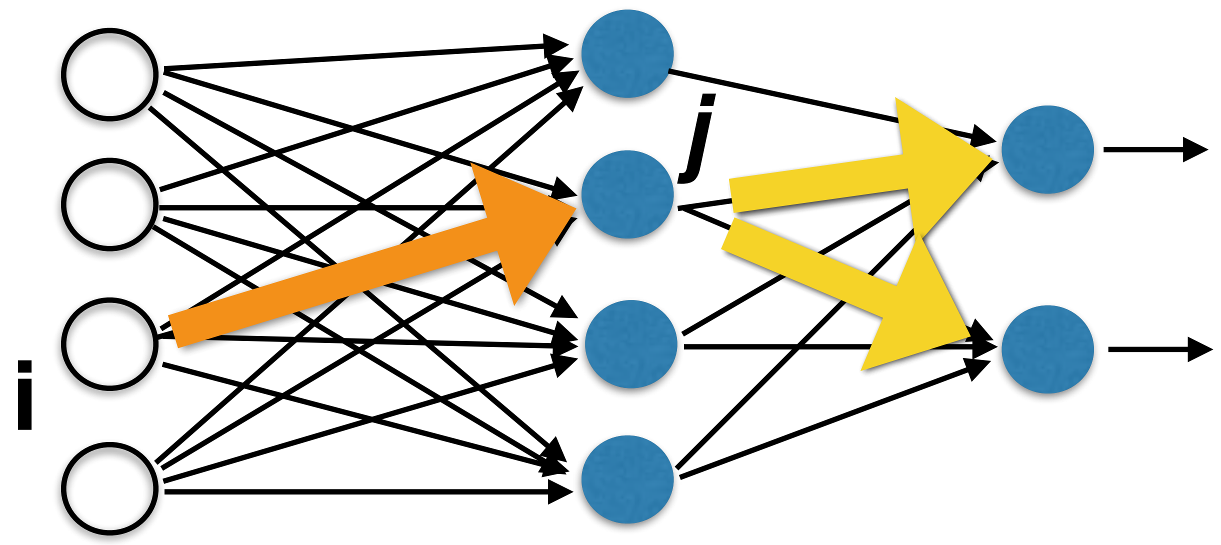


Connection $\mathbf{w_{ij}}$ between input and hidden layer (shown in orange)

Influences the output only through those output units which are connected to node ***j*** (shown connected by yellow arrows to ***j***)

$$\frac{\partial E_d}{\partial w_{ij}} = \frac{\partial E_d}{\partial \text{net}_j} \frac{\partial \text{net}_j}{\partial w_{ij}} = \sum_{k \in downstream(j)} \frac{\partial E_d}{\partial \text{net}_k} \frac{\partial \text{net}_k}{\partial \text{net}_j} \frac{\partial \text{net}_j}{\partial w_{ij}}$$

Derivation of Learning Rule



$$\frac{\partial E_d}{\partial w_{ij}} = \frac{\partial E_d}{\partial \text{net}_j}\frac{\partial \text{net}_j}{\partial w_{ij}} = \sum_{k \in downstream(j)} \frac{\partial E_d}{\partial \text{net}_k}\frac{\partial \text{net}_k}{\partial \text{net}_j}\frac{\partial \text{net}_j}{\partial w_{ij}}$$

$$= \sum_{k \in downstream(j)} -\delta_k \frac{\partial \text{net}_k}{\partial \text{net}_j} x_i$$

$$\boxed{\delta_j = -\frac{\delta E_d}{\delta net_j}}$$

$$\boxed{net_j = \sum w_{ij} x_i}$$

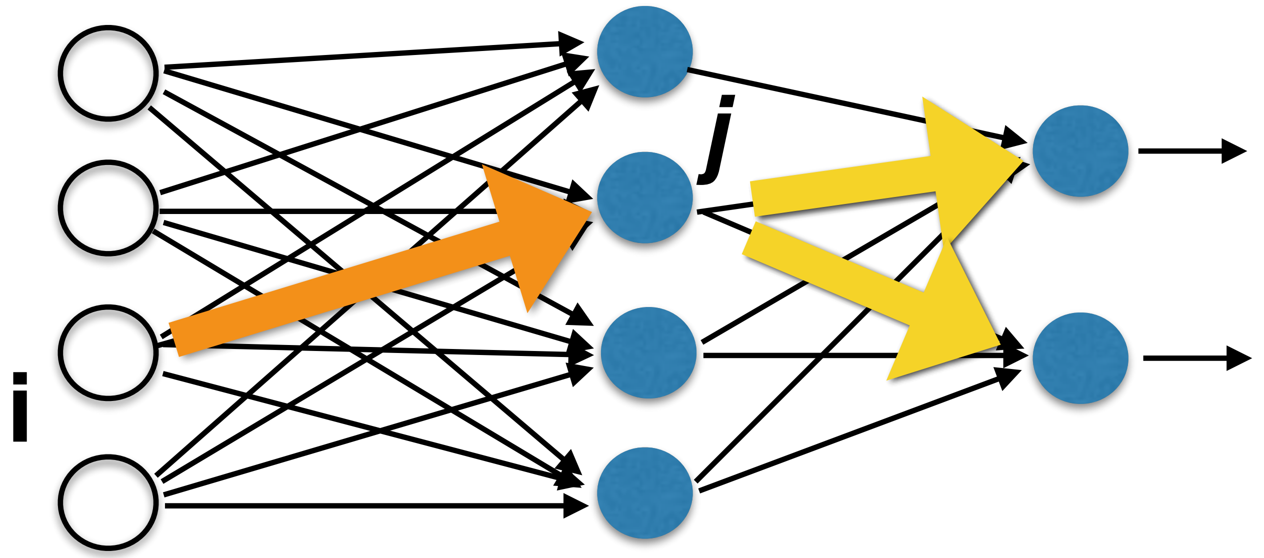
Derivation of Learning Rule

$$\frac{\partial E_d}{\partial w_{ij}} = \frac{\partial E_d}{\partial \text{net}_j} \frac{\partial \text{net}_j}{\partial w_{ij}} = \sum_{k \in downstream(j)} \frac{\partial E_d}{\partial \text{net}_k} \frac{\partial \text{net}_k}{\partial \text{net}_j} \frac{\partial \text{net}_j}{\partial w_{ij}}$$

$$= \sum_{k \in downstream(j)} -\delta_k \frac{\partial \text{net}_k}{\partial \text{net}_j} x_i$$

$$= \sum_{k \in downstream(j)} -\delta_k \frac{\partial \text{net}_k}{\partial o_j} \frac{\partial o_j}{\partial \text{net}_j} x_i$$

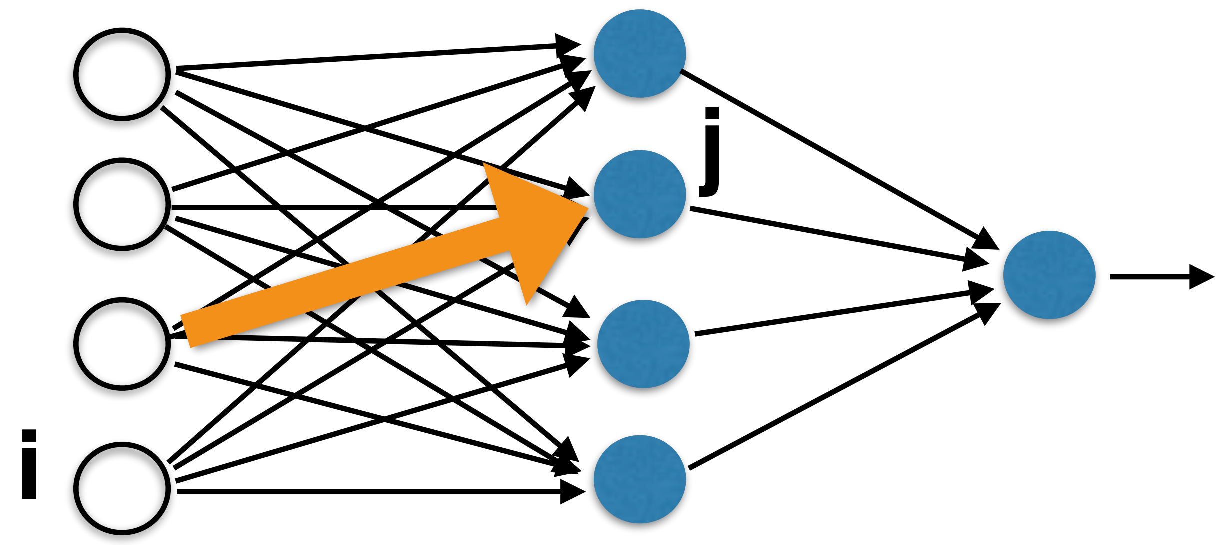
$$= \sum_{k \in downstream(j)} -\delta_k \; w_{jk} \; o_j(1 - o_j) \quad x_i$$



$$\boxed{net_j = \sum w_{ij} x_i}$$

$$\boxed{\begin{gathered}\frac{\partial o_j}{\partial \text{net}_j} = o_j(1 - o_j) \\ o_j = \frac{1}{1 + \exp\{-(\text{net}_j - T_j)\}}\end{gathered}}$$

Derivation of Learning Rule



Connection **w**

$$\Delta w_{ij} = R o_j (1 - o_j) \left(\sum_{k \in downstream(j)} \delta_k w_{jk} \right) x_i = R \delta_j x_i$$

where

$$\delta_j = o_j (1 - o_j) \left(\sum_{k \in downstream(j)} \delta_k w_{jk} \right)$$

First determine the error for the output units. Then, back propagate this error layer by layer through the network, changing weights appropriately in each layer

The Backpropagation Algorithm

For each example in the training set, do:

1. Compute the network output for this example
2. Compute the error term between he output and target values
3. For each output unit **j**, compute error term:

$$\delta_j = (t_j - o_j) o_j (1 - o_j)$$

4. For each hidden unit, compute error term

$$\delta_j = o_j (1 - o_j) \left(\sum_{k \in downstream(j)} \delta_k w_{jk} \right)$$

5. Update weights by $\Delta w_{ij} = R \delta_j x_i$

A few small things

We computed gradient for a single example. So, you can directly use it if you perform **stochastic gradient descent**

Do you know how to convert this information to gradient descent ?

We only showed updated for connection weights. Do you know how to derive updates for the thresholds **T** ?

Usually, for neural networks, people use stochastic gradient descent with mini-batches (and not with one example at a time)

Alternatives

You already saw that you can use a lot of other activation functions

You can also use any loss function. The example you saw uses squared loss

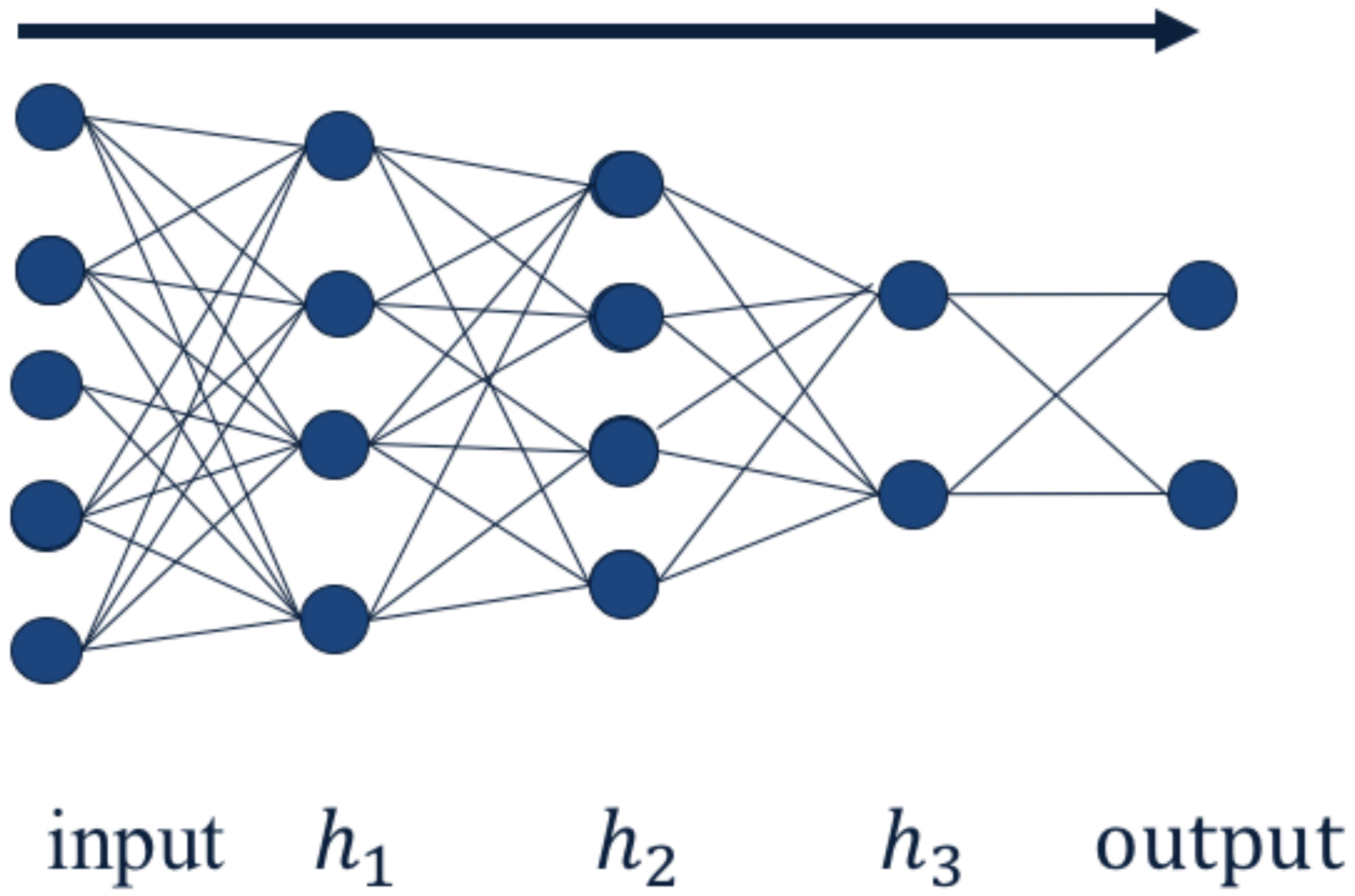
Squared loss is actually not that good for neural nets, due to training slowdown.

A better loss function is **cross entropy**

$$Err(\mathrm{w}) = -\sum_{i \in K} (t_i \log(o_i) + (1 - t_i) \log(1 - o_i))$$

Exercise : Try to compute its back propagation terms

More Hidden Layers



Same algorithm holds for more hidden layers

Deep Learning Libraries

Tensorflow

Torch

Theano

Pytorch

Dynet

They allow automatic differentiation. You can just write the network, get the gradients for free.

Lets look at **Tensorflow** demo

Comments on Training

- No guarantee of convergence, may oscillate or reach a local minima
- In practice, many large networks can be trained on large amounts of data for realistic problems
- Termination criteria : Number of epochs; Threshold on training set error; No decrease in error; Increase error on a validation set
- To avoid local minima : several trials with different random initial weights with majority or voting techniques

Over-fitting Prevention

- Running too many epochs may over-train the network and result in overfitting.

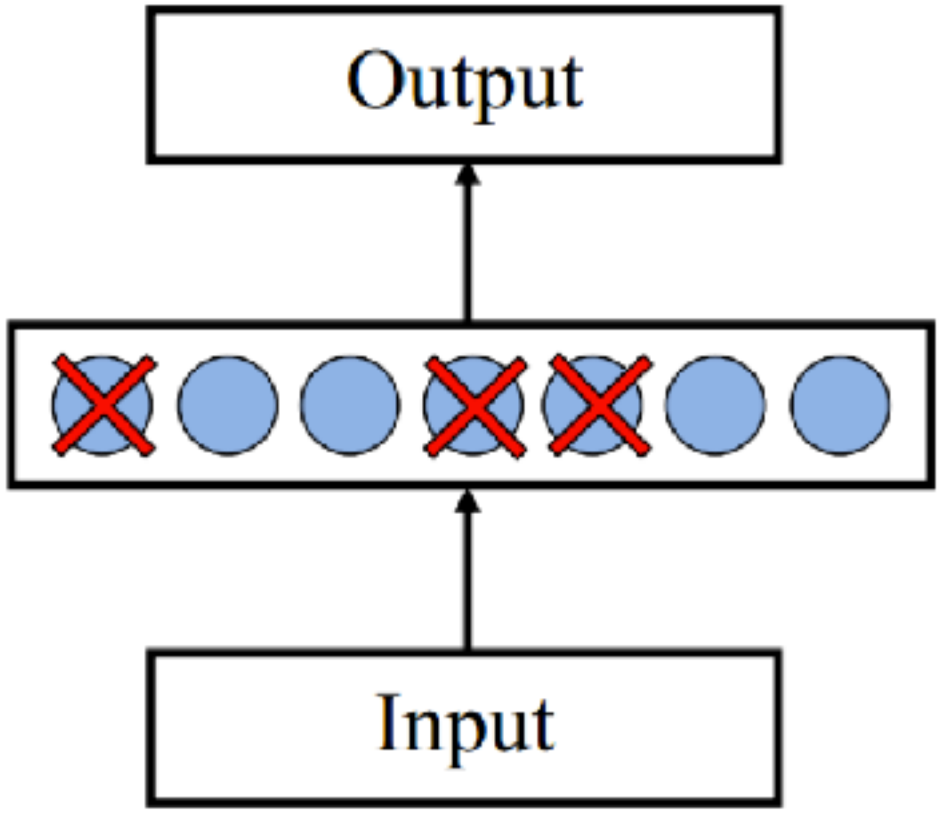
- Keep a held out validation set and test accuracy after each epoch, maintain weights for the best performing network on the validation set, and return it when performance decreases significantly beyond that.

- Too few hidden units can prevent the data from adequately fitting the data, too many hidden units lead to overfitting. You can tune to get the best number of hidden units for your task.

- Another approach to prevent overfitting : Change error function to include a term for the sum of squares of the weights in the network.

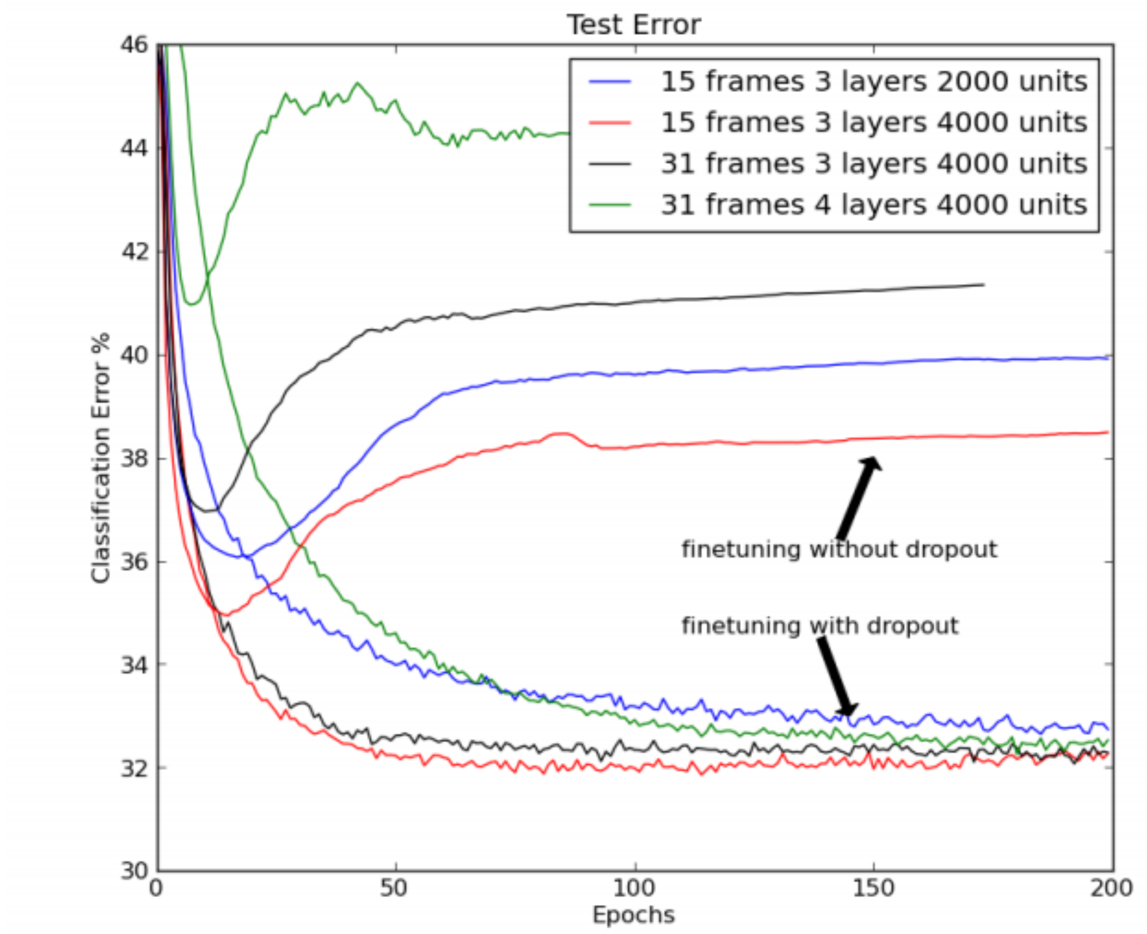
Dropout Training

Proposed by Hinton et al 2012



Each time, decide on whether to delete a hidden unit with some probability **p**

Dropout Training



Dropout of 50% of hidden units, and 20% of input units

Story Time

Some History

Inspired by biological systems, but don't take this seriously

Computation : McCollough and Pitts (1943) showed how linear threshold units can be used to compute logical functions

Learning Rules

Hebb (1949) suggested that if two units are both active (firing) then the weights between them should increase.

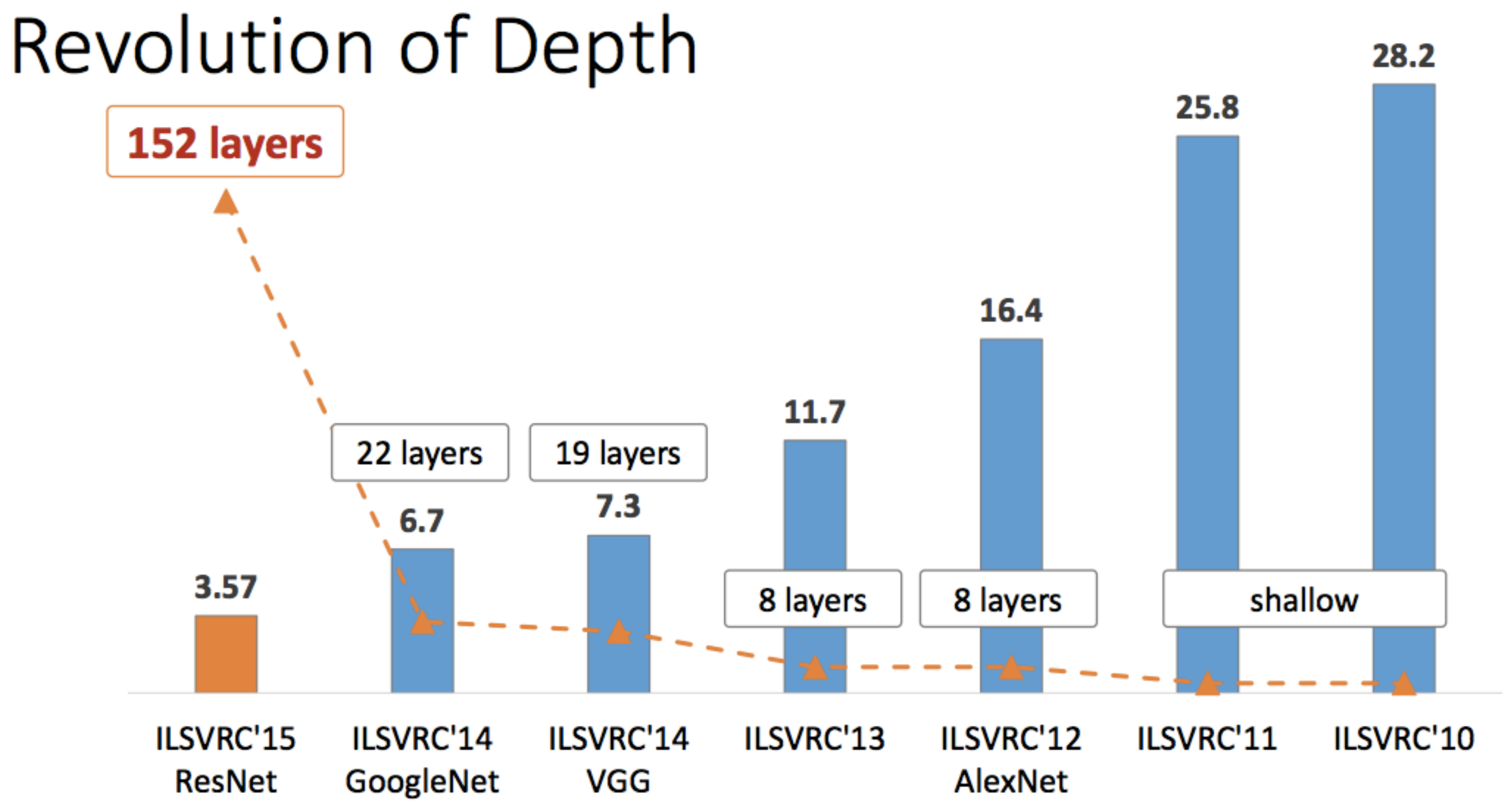
Rosenblatt (1959) suggested that when a target output value is provided for a single neuron with fixed input, it can incrementally change weights and learn to produce the output using the Perceptron learning rule..

Rise and Fall and Rise and Fall of Neural Nets

1959	Rosenblatt's Perceptron	👍
1969	Minsky and Papert's book	👎
1986	Backpropagation work	👍
1995	SVM's and deep nets not being trainable	👎
2006	Hinton's deep belief net paper	👍

See: http://people.idsia.ch/~juergen/who-invented-backpropagation.html

Deep Learning



Object Recognition Performance on Imagenet

Next Class

We will learn about some deep learning architectures