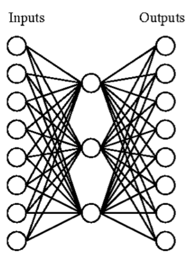
Artificial Neural Networks (ANNs)

Biologically inspired computational model:

(1) Simple computational units (neurons).
(2) Highly interconnected - connectionist view
(3) Vast parallel computation, consider:
- Human brain has ~10^{11} neurons
- Slow computational units, switching time ~10^{-3} sec (compared to the computer >10^{-10} sec)
- Yet, you can recognize a face in ~10^{-1} sec
- This implies only about 100 sequential, computational neuron steps - this seems too low for something as complicated as recognizing a face
- Parallel processing



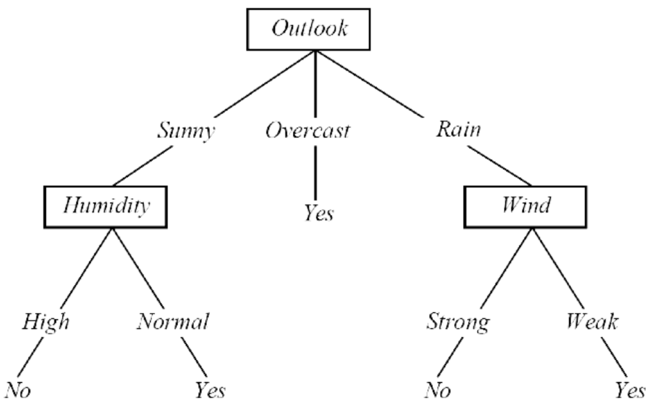
ANNs are naturally parallel - each neuron is a self-contained computational unit that depends only on its inputs.

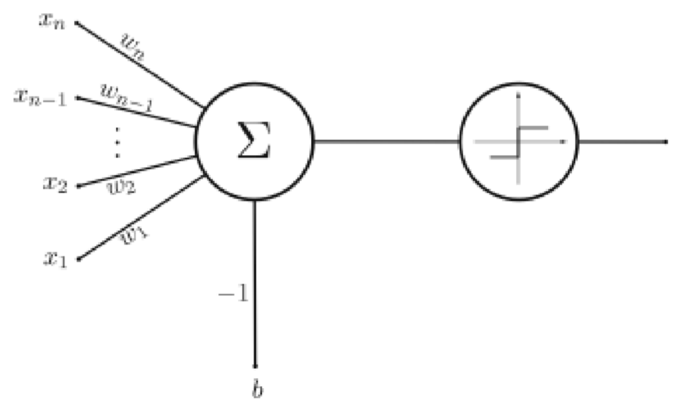
Learning

We have seen machine learning with different representations:

(1) Decision trees -- symbolic representation of various decision rules -- “disjunction of conjunctions”

(2) Perceptron -- learning of weights that represent alinear decision surface classifying a set of objects into two groups



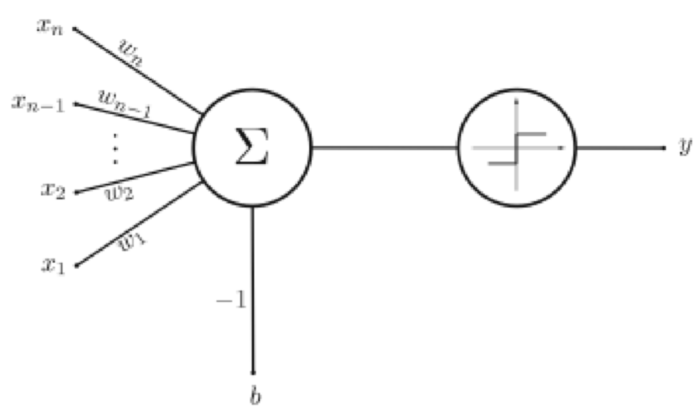


Different representations give rise to different <u>hypothesis</u> or <u>model spaces</u>.

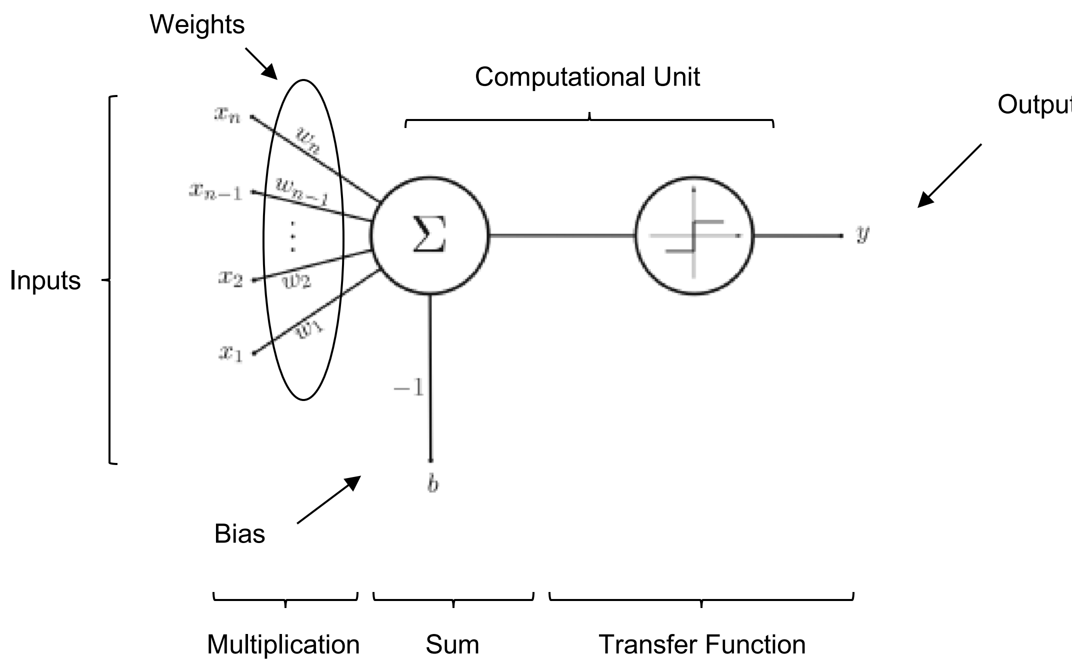
Machine <u>learning algorithms search</u> these model spaces for the <u>best fitting model</u>.

The Perceptron

- A simple, single layered neural “network” - only has a single neuron.
- However, even this simple neural network is already powerful enough to perform (linear) classification tasks.



The Architecture



Transfer Function:

$$\operatorname{sgn}(k) = \begin{cases} +1 \text{ if } k \geq 0 \\ -1 \text{ otherwise} \end{cases}$$

Perceptron Computation: $$y = \operatorname{sgn}\left(b + \sum_{i=1}^{m} w_i x_i \right)$$

Note: $y \in \{+1, -1\}$ ← Binary Classification

Computation

A perceptron computes the value,

$$y = \operatorname{sgn}\left(b + \sum_{i=1}^{m} w_i x_i\right)$$

Ignoring the activation function sgn and setting *m* = 1, we obtain,

$$y' = b + w_1 x_1$$

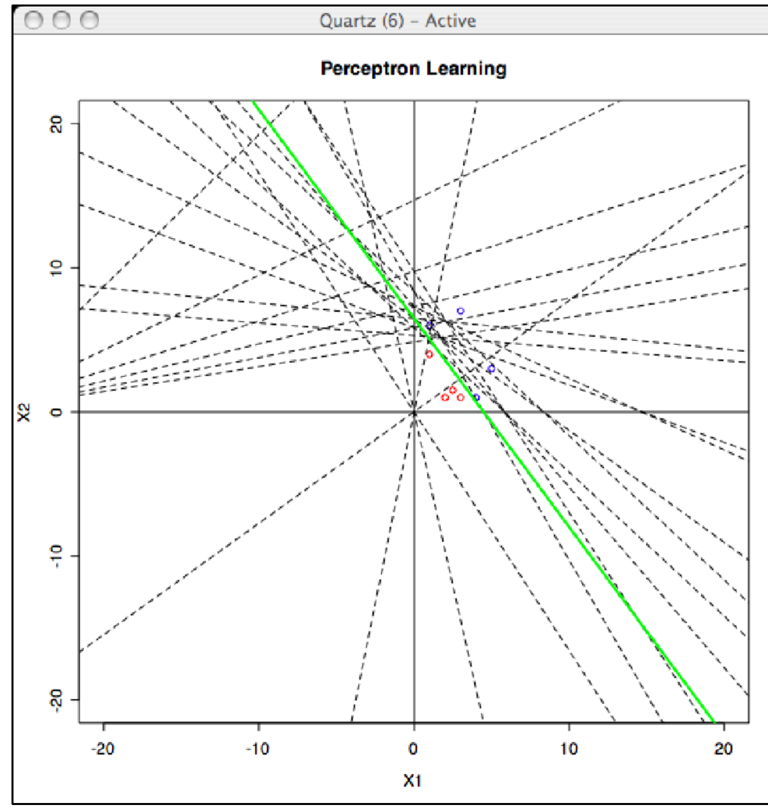
But this is the equation of a line with slope *w* and offset *b*.

Observation: For the general case the perceptron computes a hyperplane in order to accomplish its classification task,

$$y' = b + \sum_{i=1}^{m} w_i x_i = b + \vec{w} \bullet \vec{x}$$

Perceptron Learning Revisited

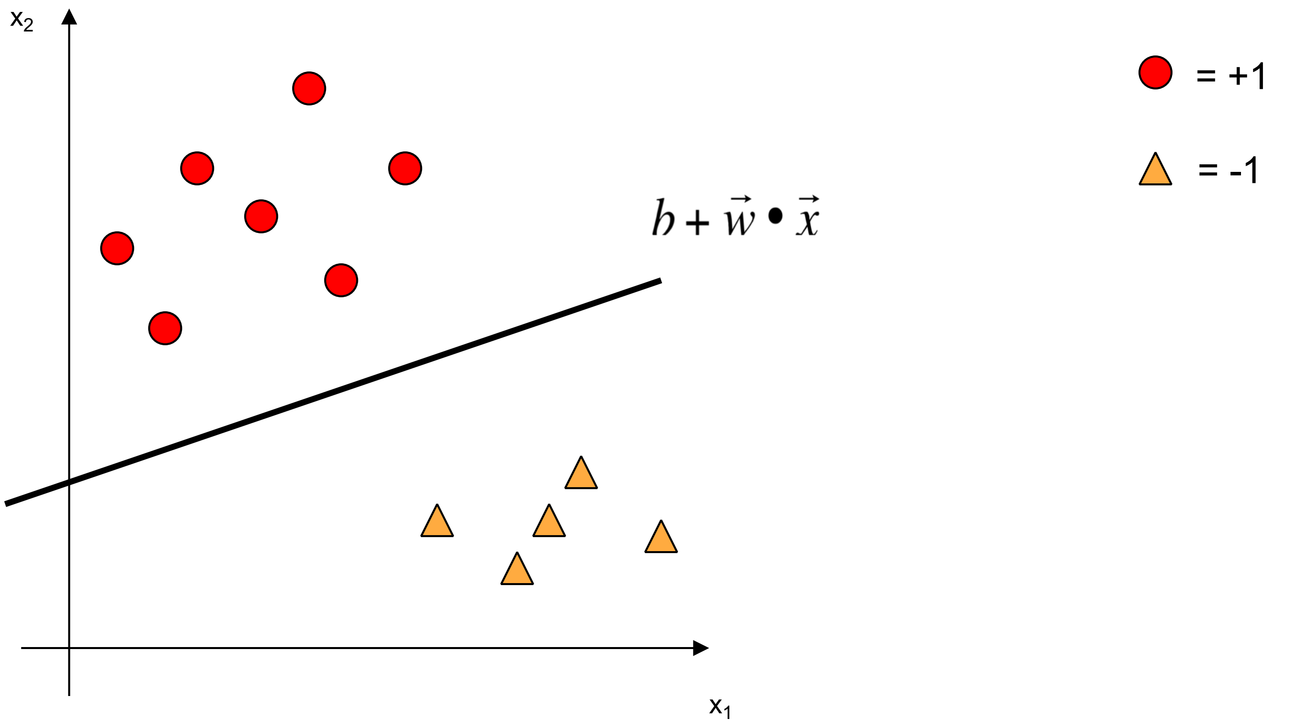
Initialize $\overline{w}$ and b to random values.
repeat
  for each $(\overline{x}_i, y_i) \in D$ **do**
    if $\hat{f}(\overline{x}_i) \neq y_i$ **then**
      Update $\overline{w}$ and b incrementally.
    end if
  end for
until D is perfectly classified.
return $\overline{w}$ and b



Constructs a line (hyperplane) as a classifier

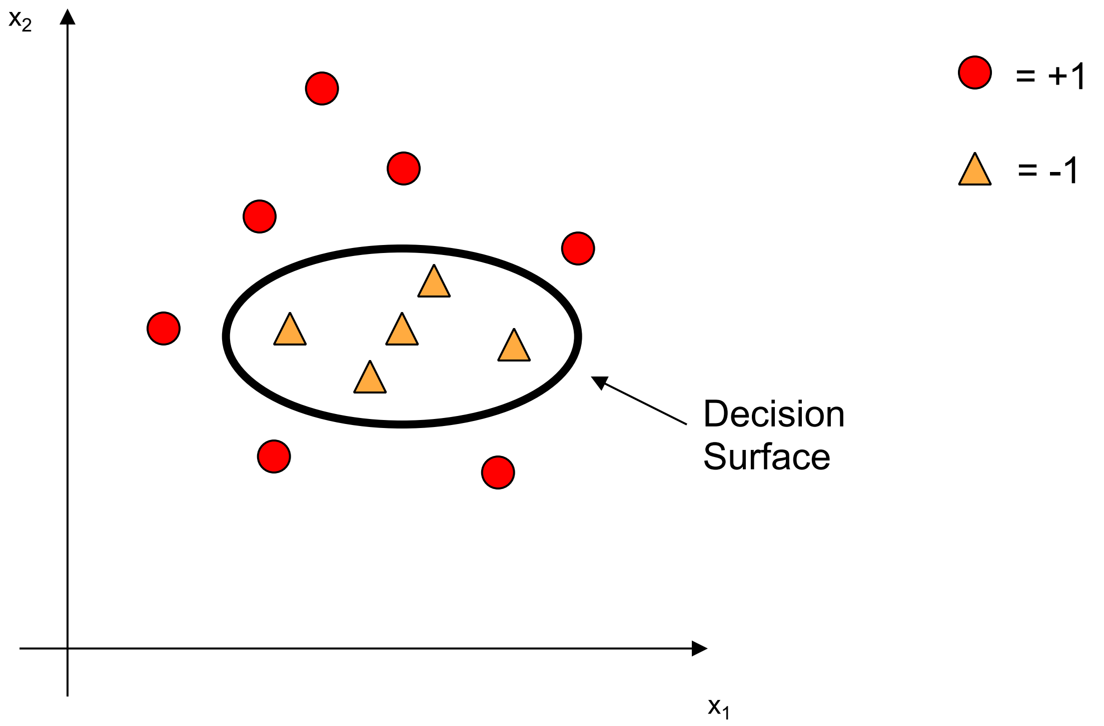
$$b + \vec{w} \bullet \vec{x}$$

Classification



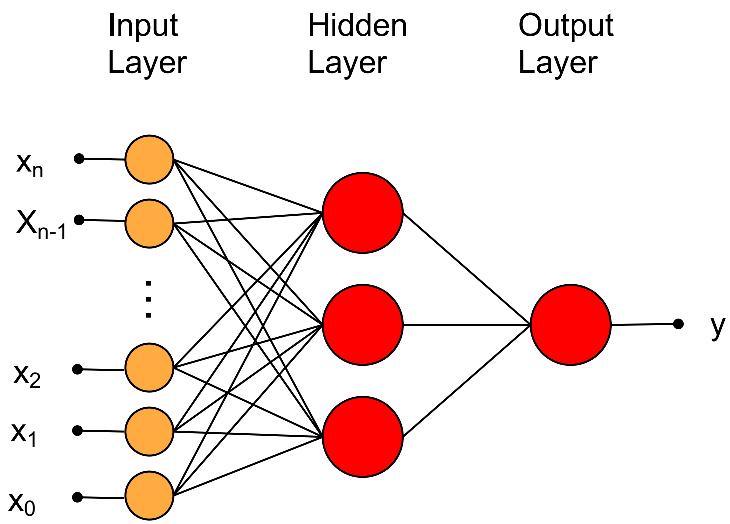
In order for the hyperplane to become a classifier we need to find *b* and *w* => learning!

What About Non-Linearity?

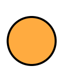


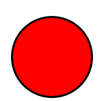
Can we learn this decision surface? …Yes! *Multi-Layer Perceptrons*

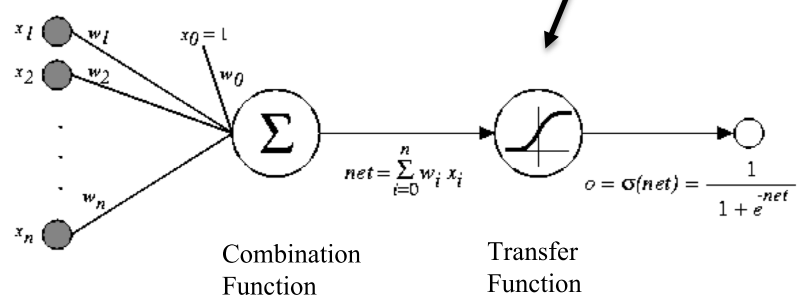
Multi-Layer Perceptrons (ANNs)



Notice the smooth Transfer function!


≡ Linear/Input Unit


≡



How do we train?

Perceptron was easy:

Initialize $\overline{w}$ and b to random values.
repeat
  for each $(\overline{x}_i, y_i) \in D$ **do**
    if $\hat{f}(\overline{x}_i) \neq y_i$ **then**
      Update $\overline{w}$ and b incrementally.
    end if
  end for
until D is perfectly classified.
return $\overline{w}$ and b

error

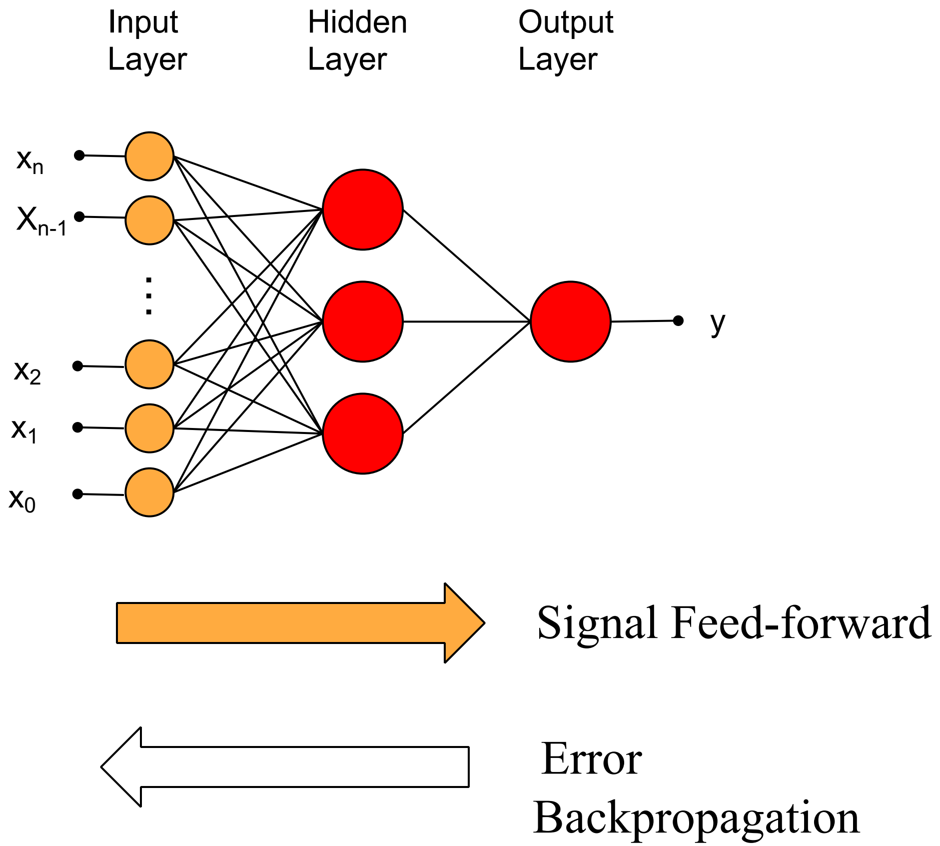
Every time we found an error of the predicted value $f(x_i)$ compared to the label in the training set y_i, we update w and b.

Not so easy in multi-layer neural networks – the error can occur deep in the network!

Artificial Neural Networks

Feed-forward with Backpropagation

We have to be a bit smarter in the case of ANNs: compute the signal (feed forward) and then use the error at the output to update all the weights by propagating the error back through the network.



Backpropagation

$$\delta_o = y(1-y)\Delta = \frac{\partial \Delta}{\partial \vec{w} \bullet \vec{x}} = \frac{\partial (t-y)^2}{\partial \vec{w} \bullet \vec{x}}$$

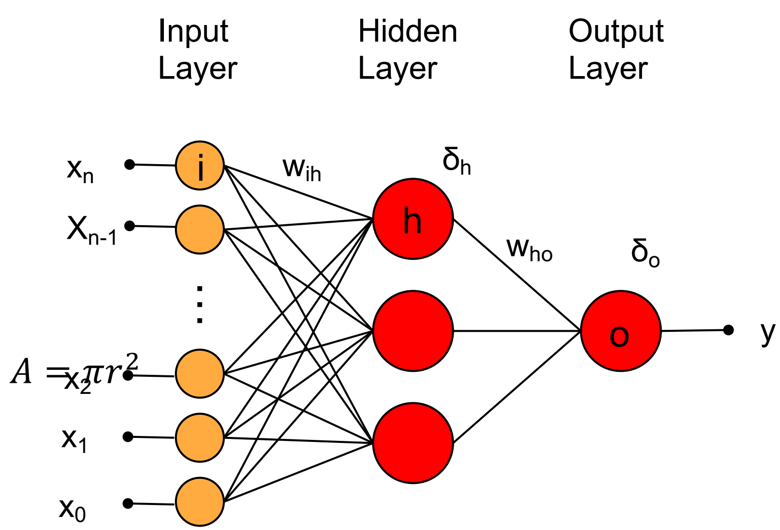
$E = (y' - y)^2$ (output error)

$\delta_0 = y(1-y)E$ (output node error)

$w_{ho} \leftarrow w_{ho} + \alpha\delta_o$ (weight update)

$\delta_h = y(1-y)w_{ho}\delta_o$ (hidden node error)

$w_{ih} \leftarrow w_{ih} + \alpha\delta_h$ (weight update)



For this to work transfer Function has to be smooth!!

This only works because

$$\delta_o = y(1-y)E = \frac{\partial E}{\partial \boldsymbol{w}\cdot\boldsymbol{x}} = \frac{\partial (y'-y)^2}{\partial \boldsymbol{w}\cdot\boldsymbol{x}} = 2(\mathrm{y}' - \mathrm{y})\left(\frac{\partial y\prime}{\partial \boldsymbol{w}\cdot\boldsymbol{x}} - \frac{\partial y}{\partial \boldsymbol{w}\cdot\boldsymbol{x}}\right)$$

and the output y is differentiable because the transfer function is differentiable. Also note, everything is based on the *rate of change* of the error…we are searching in the direction where the rate of change will minimize the output error.

Backpropagation Algorithm

Note: this algorithm is for a NN with a single output node o and a single hidden layer. It can easily be generalized.

```
Initialize the weights in the network (often randomly)
  Do
        For each example e in the training set
             // forward pass
             y = compute neural net output
             y' = label for e from training data
             Calculate error E = (y' - y)^2 at the output units
              // backward pass
             Compute error δ_o for weights from a hidden node h to the output node o using E
             Compute error δ_h for weights from an input node i to hidden node h using δ_o
             Update the weights in the network
  Until all examples classified correctly or stopping criterion satisfied
  Return the network
```

Source: http://en.wikipedia.org/wiki/Backpropagation

Neural Network Learning

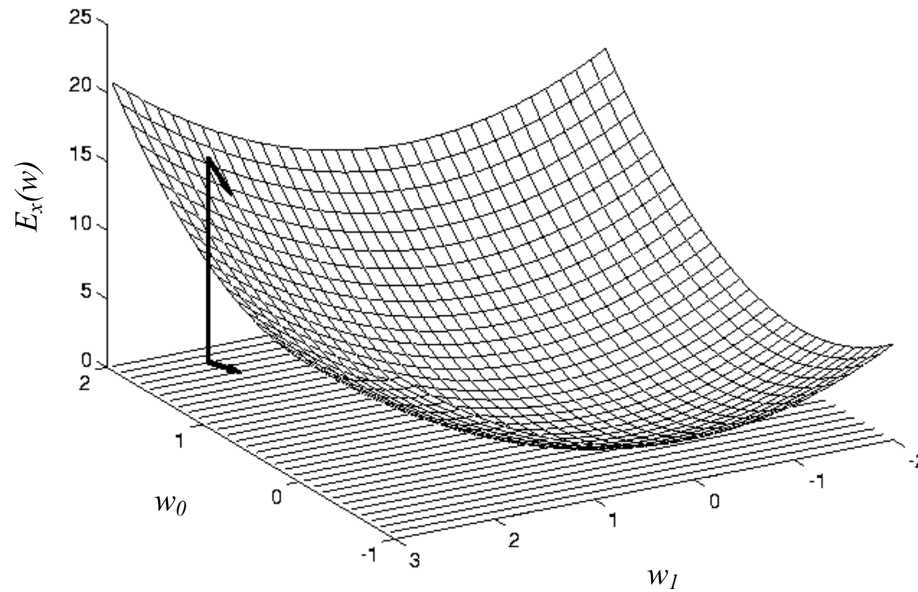
- Define the network error in terms of weights w as

$$E_x(w) = (y' - y)^2$$

for some training instance *x.*

- Use the gradient (slope) of the error surface to guide the search towards appropriate weights:

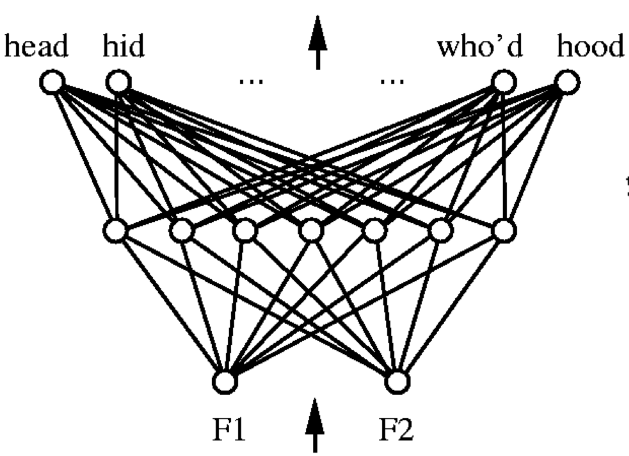
$$\Delta w_k = -\eta \frac{\partial E_x}{\partial w_k}$$

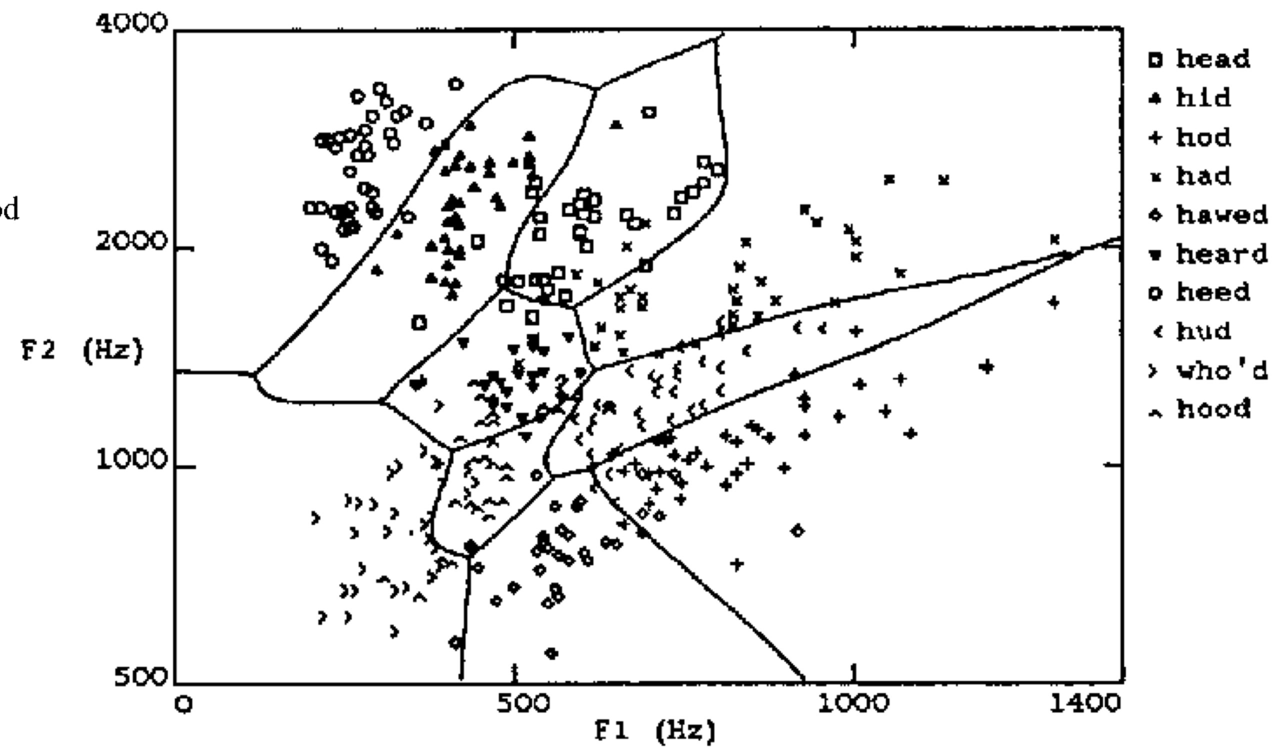


☞ Backpropagation can be understood as a <u>stochastic gradient search</u> on the error surface of the network.

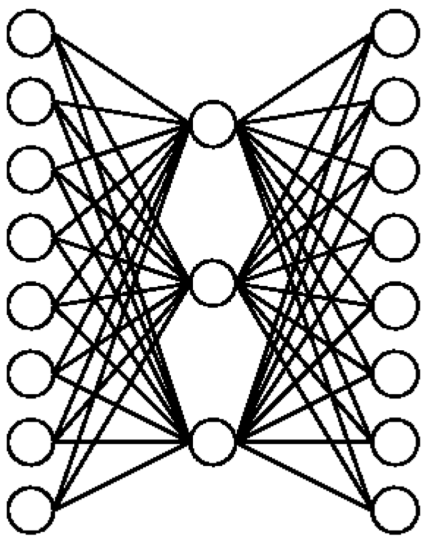
Representational Power

- Every bounded continuous function can be approximated with arbitrarily small error by a network with one hidden layer.
- Any function can be approximated to arbitrary accuracy by a network with two hidden layers.





Hidden Layer Representations

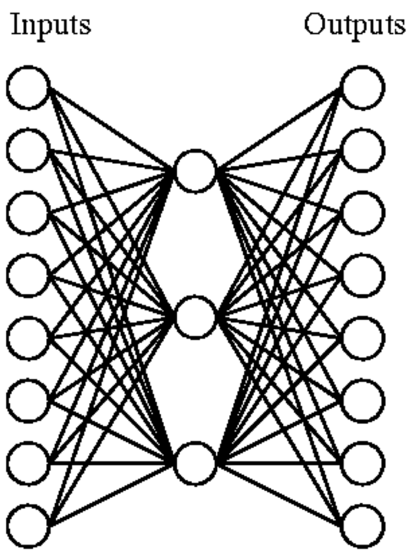


Target Function:

Input		Output
10000000	$\rightarrow$	10000000
01000000	$\rightarrow$	01000000
00100000	$\rightarrow$	00100000
00010000	$\rightarrow$	00010000
00001000	$\rightarrow$	00001000
00000100	$\rightarrow$	00000100
00000010	$\rightarrow$	00000010
00000001	$\rightarrow$	00000001

Can this be learned?

Hidden Layer Representations



Input		Hidden Values				Output
10000000	→	.89	.04	.08	→	10000000
01000000	→	.01	.11	.88	→	01000000
00100000	→	.01	.97	.27	→	00100000
00010000	→	.99	.97	.71	→	00010000
00001000	→	.03	.05	.02	→	00001000
00000100	→	.22	.99	.99	→	00000100
00000010	→	.80	.01	.98	→	00000010
00000001	→	.60	.94	.01	→	00000001

1	0	0
0	0	1
0	1	0
1	1	1
0	0	0
0	1	1
1	0	1
1	1	0

☞ This neural network architecture is sometimes also called autoencoder because of its ability to invent new representations of the input data and is a popular building block in deep-learning.