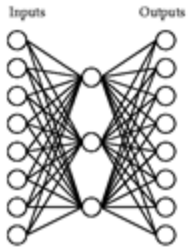
Artificial Neural Networks (ANNs)

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



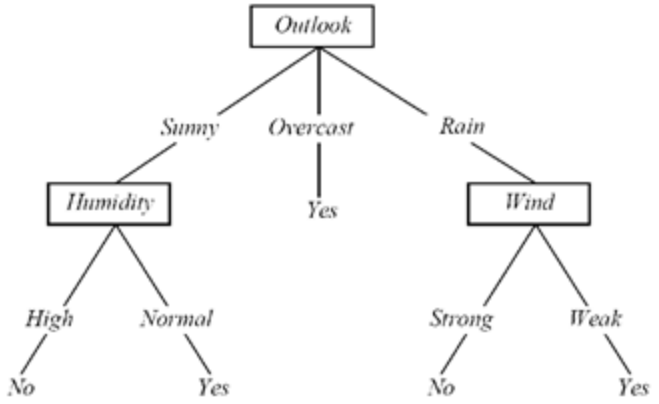
ANNs are naturally parallel - each neuron is a self-contained computational unit that depends only on its inputs.

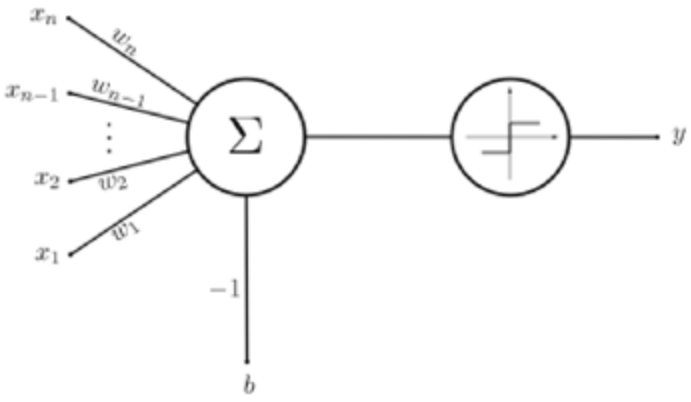
Learning

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



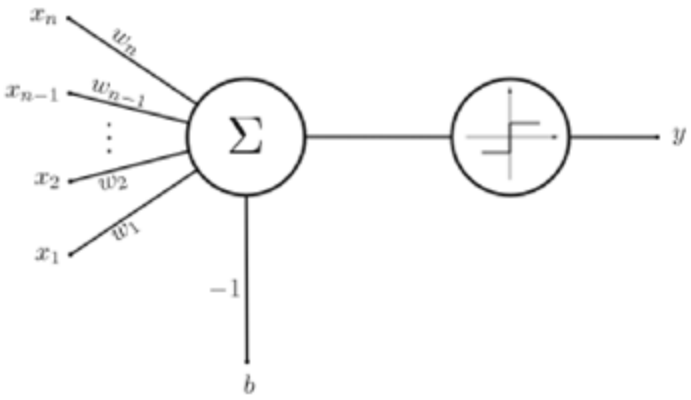


Different representations give rise to different hypothesis or model spaces.

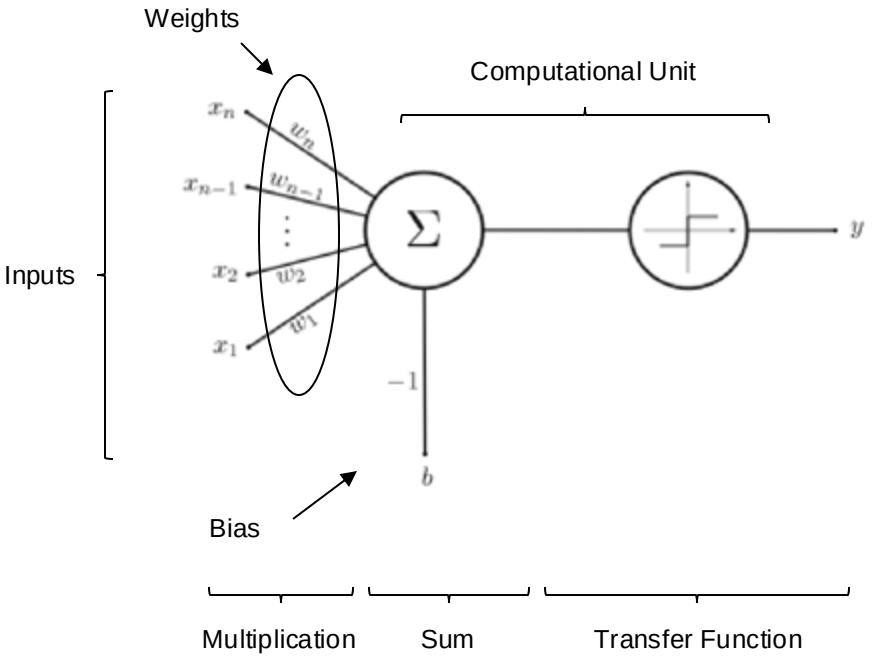
Machine learning algorithms search these model spaces for the best fitting model.

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:

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

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

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,

$$y' = b + w_1 x_1$$

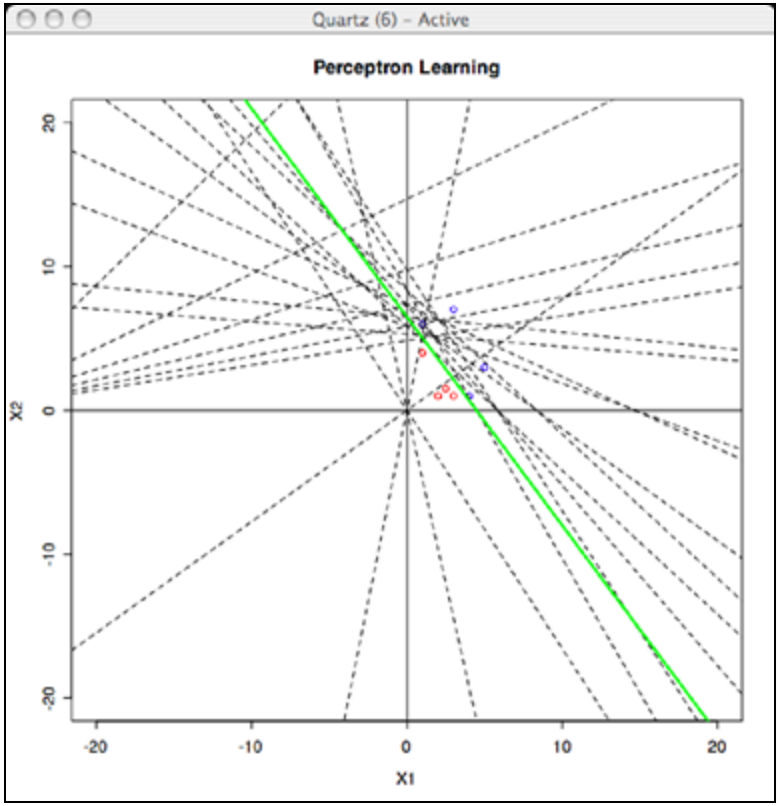
But this is the equation of a line with slope w and offset b.

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

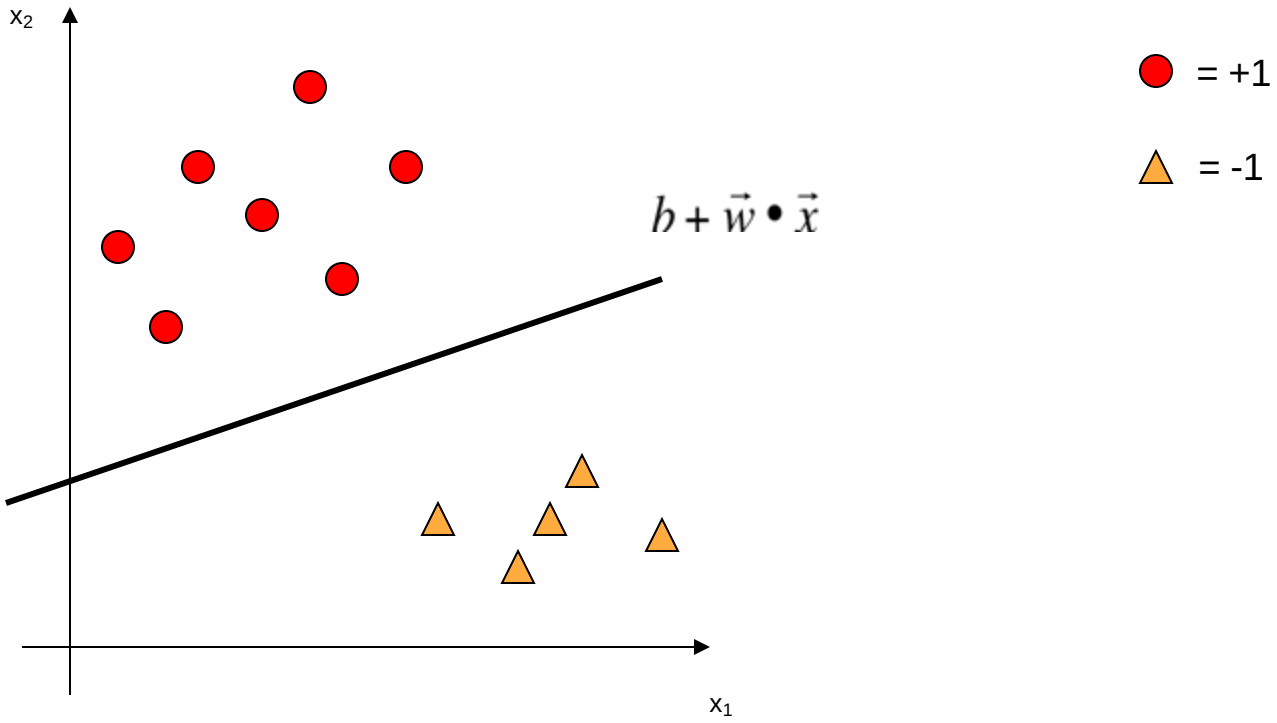
Initialize $\overline{w}$ and b to random values.
repeat
  for each $(\overline{x}_i, y_i) \in D$ **do**
    if $\hat{f}(\overline{x}_i) \neq y_i$ **then**
      Update $\overline{w}$ and b incrementally.
    end if
  end for
until D is perfectly classified.
return $\overline{w}$ and b



Constructs a line (hyperplane) as a classifier

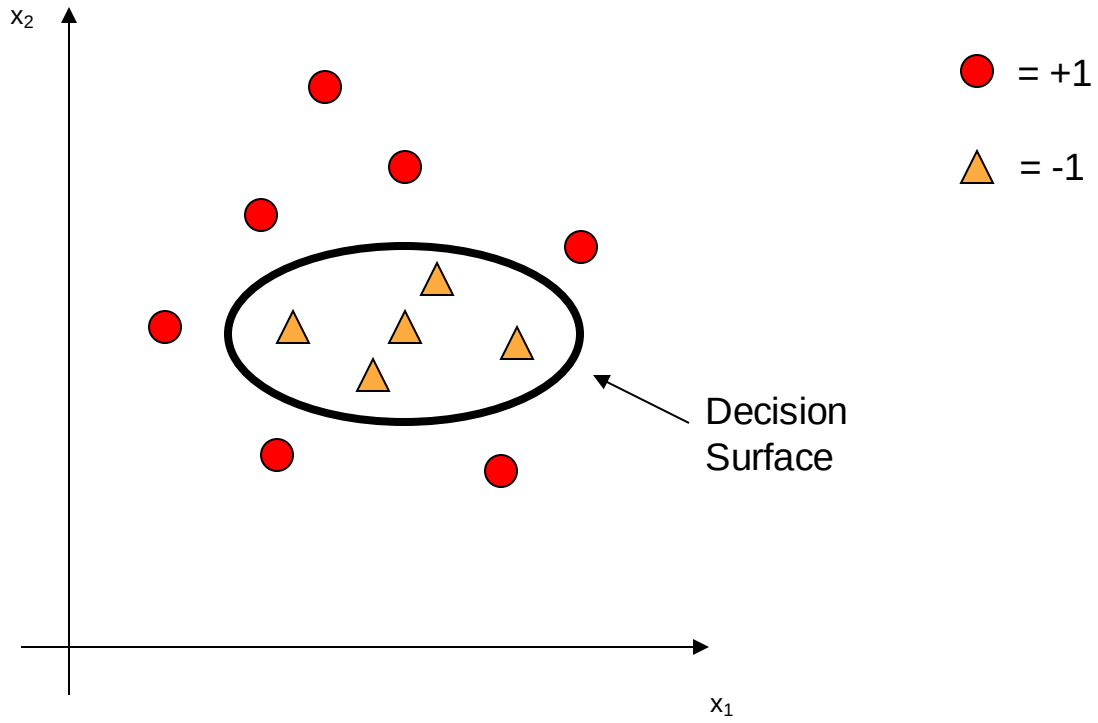
$$b + \vec{w} \bullet \vec{x}$$

Classification



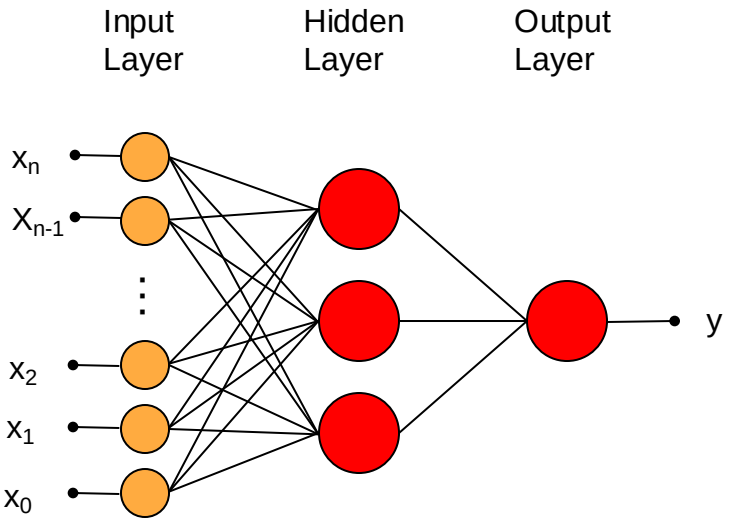
In order for the hyperplane to become a classifier we need to find *b* and *w* => learning!

What About Non-Linearity?

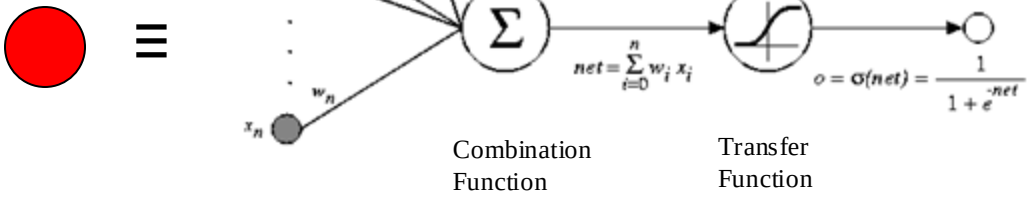


Can we learn this decision surface? …Yes! *Multi-Layer Perceptrons*

Multi-Layer Perceptrons (ANNs)

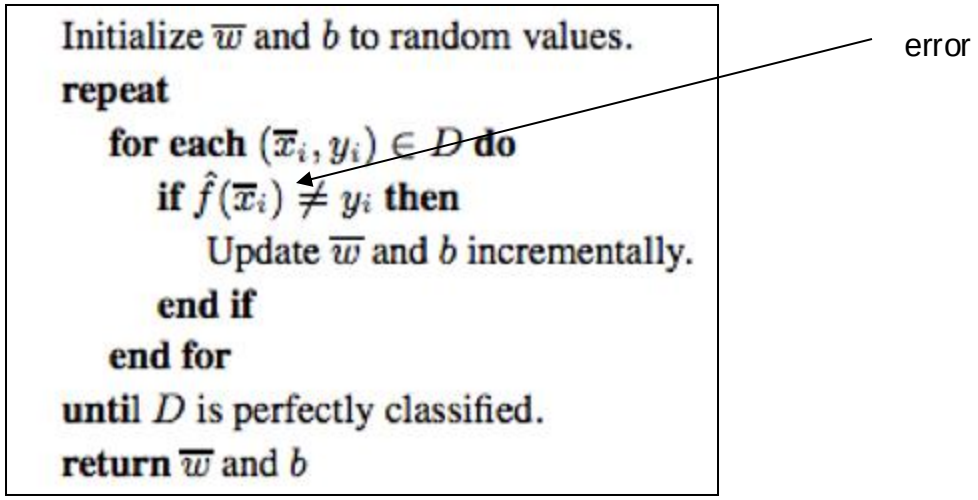


Notice the smooth Transfer function!



How do we train?

Perceptron was easy:



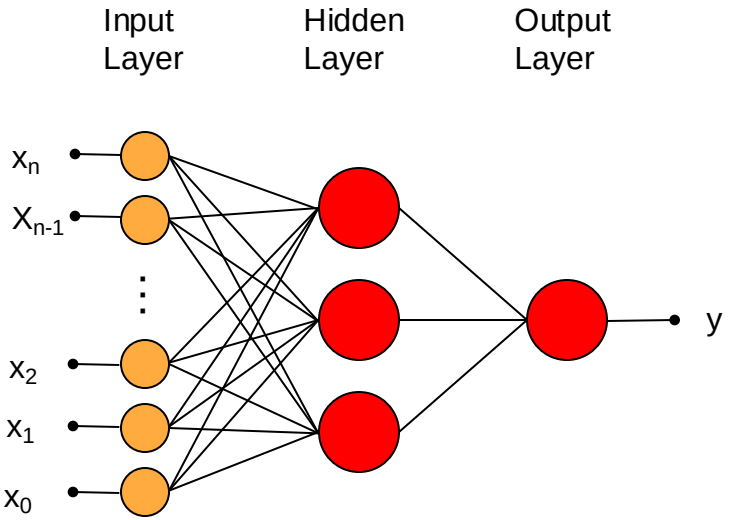
Every time we found an error of the predicted value $f(x_i)$ compared to the label in the training set y_i, we update w and b.

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

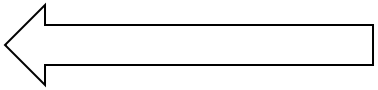
Artificial Neural Networks

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.



Signal Feed-forward



Error Backpropagation

$\alpha =$ learning rate

Backpropagation

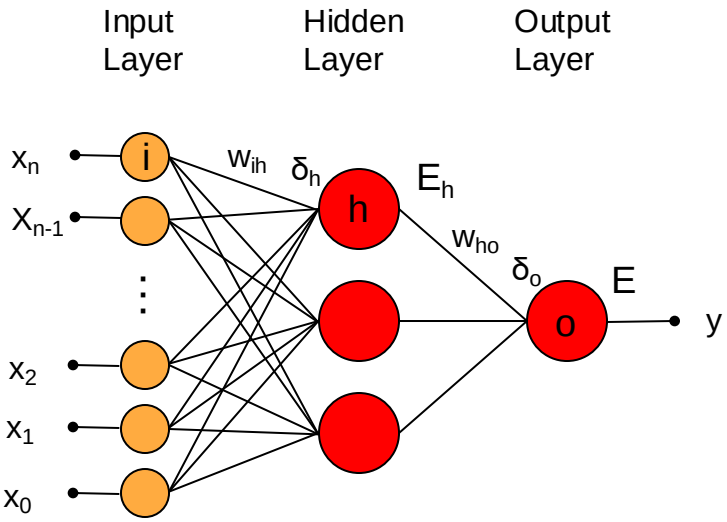
$E = (y' - y)^2$ (output error)

$\delta_0 = \frac{\partial E}{\partial w}$ (error at output node)

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

$\delta_h = \frac{\partial E_h}{\partial w} = \frac{\partial w_{ho}}{\partial w}\delta_o$ (error at hidden node)

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



Note: These computations only work if the transfer functions are differentiable.

Neural Network Learning

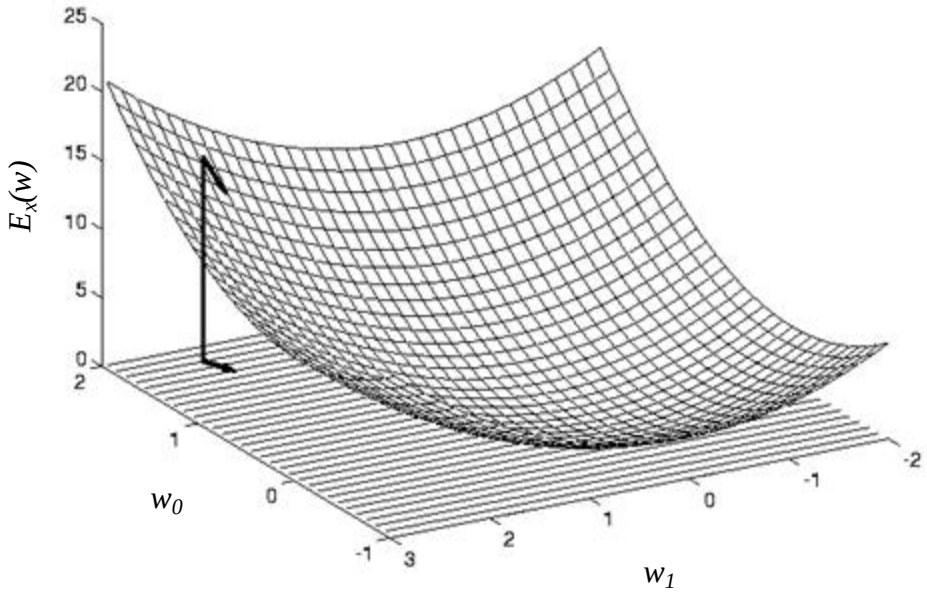
- Define the network error in terms of weights w as

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

for some training instance *x*.

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

$$\Delta w = -\frac{\partial E}{\partial w}$$



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

Backpropagation Algorithm

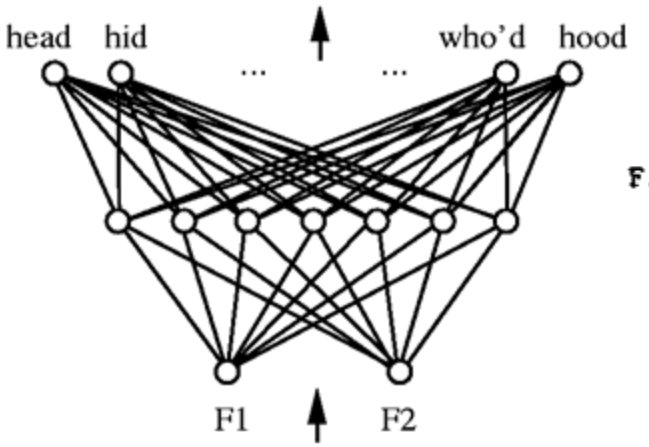
Note: this algorithm is for a NN with a single output node o and a single hidden layer. It can easily be generalized.

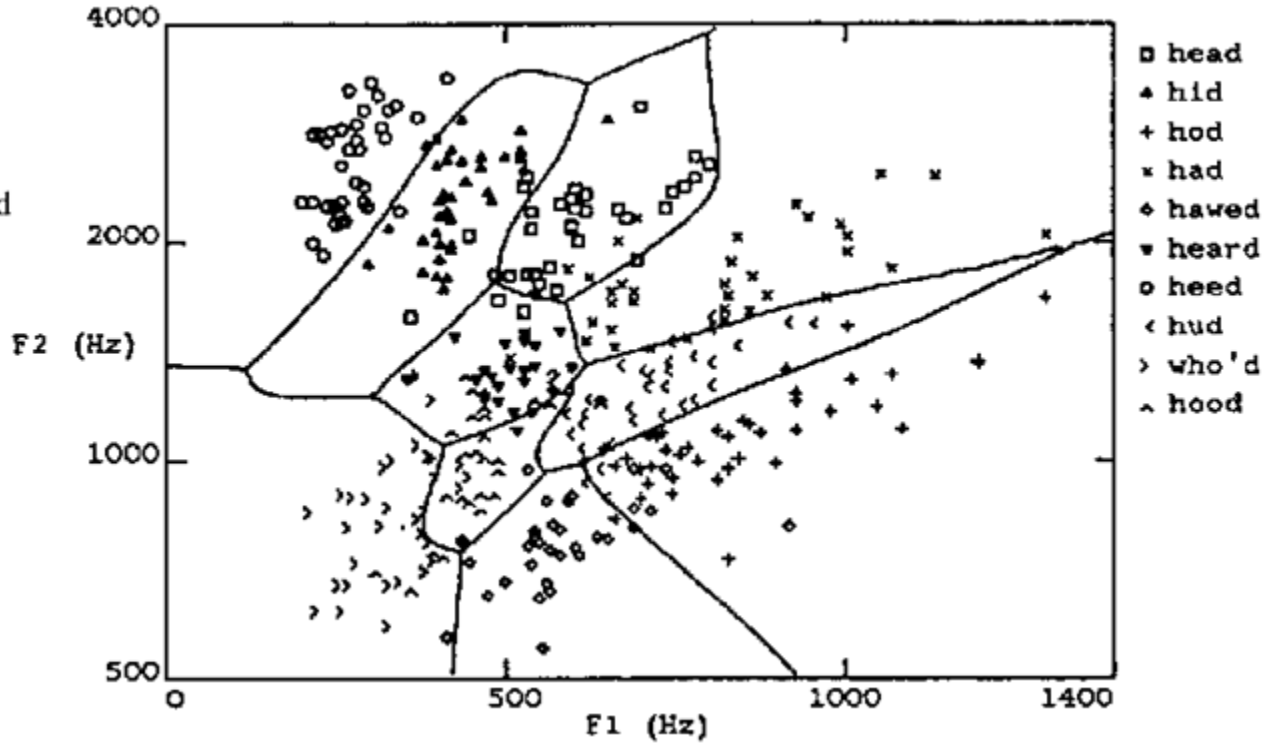
Initialize the weights in the network (often randomly)
Do
  For each example e in the training set
    // forward pass
    y = compute neural net output
    y' = label for e from training data
    Calculate error $E = (y' - y)^2$ at the output units
    // backward pass
    Compute error δ_o for weights 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

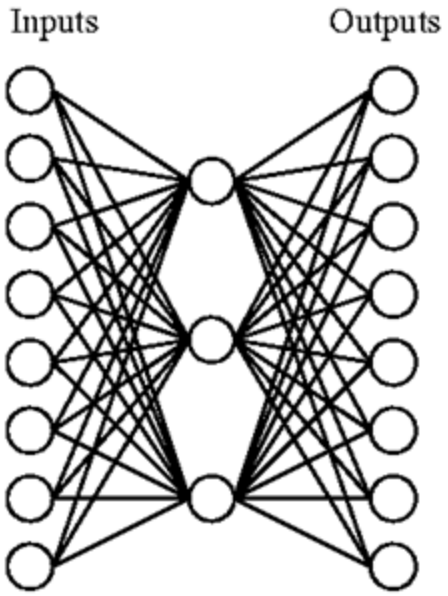
Representational Power

- 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

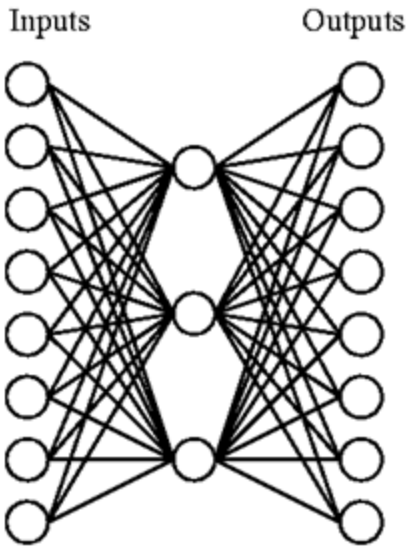


Target Function:

Input		Output
10000000	→	10000000
01000000	→	01000000
00100000	→	00100000
00010000	→	00010000
00001000	→	00001000
00000100	→	00000100
00000010	→	00000010
00000001	→	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.