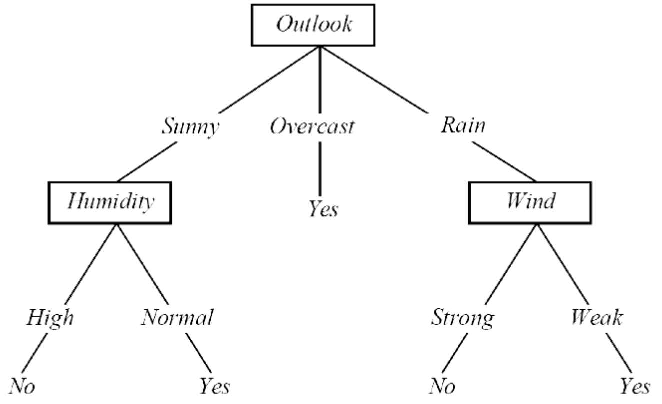
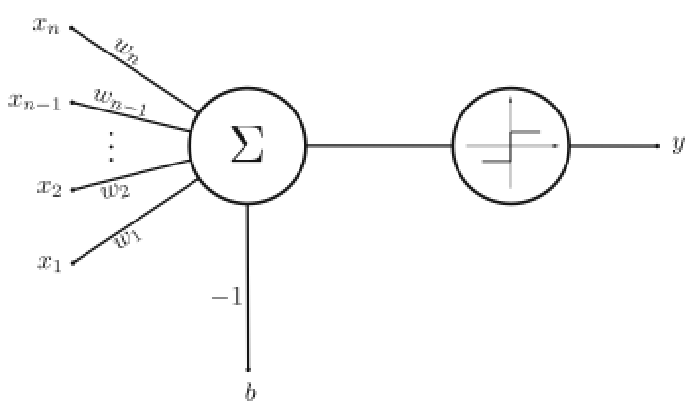
Learning

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



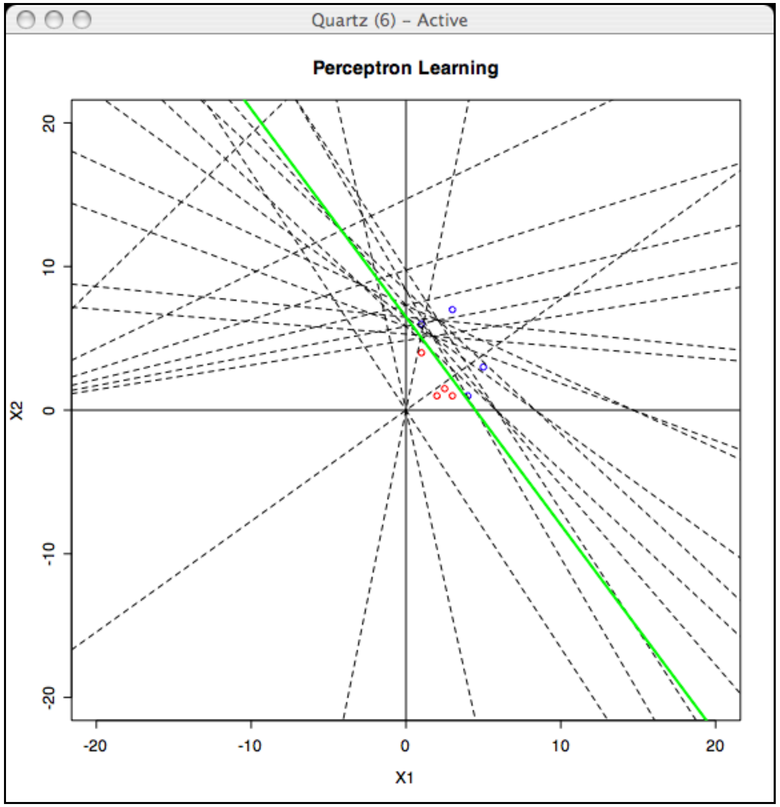


Different representations give rise to different <u>hypothesis</u> or <u>model spaces</u>.

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

Perceptron Learning Revisited

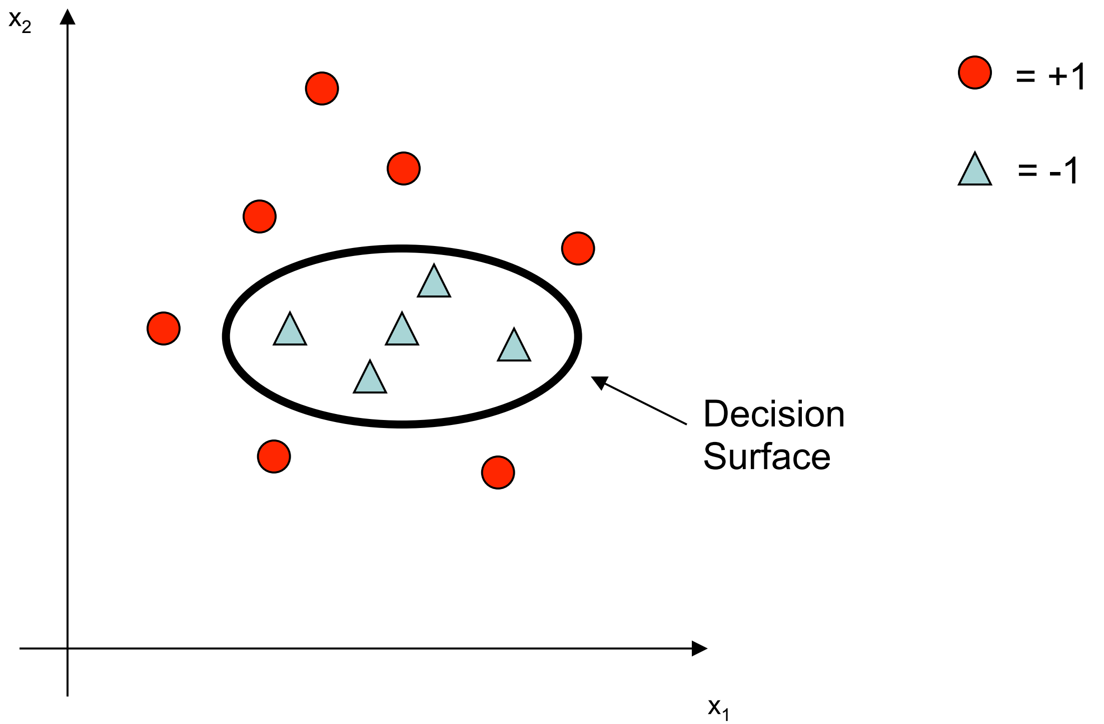
Initialize $\overline{w}$ and b to random values.
repeat
  for each $(\overline{x}_i, y_i) \in D$ **do**
    if $\hat{f}(\overline{x}_i) \neq y_i$ **then**
      Update $\overline{w}$ and b incrementally.
    end if
  end for
until D is perfectly classified.
return $\overline{w}$ and b



Constructs a line (hyperplane) as a classifier.

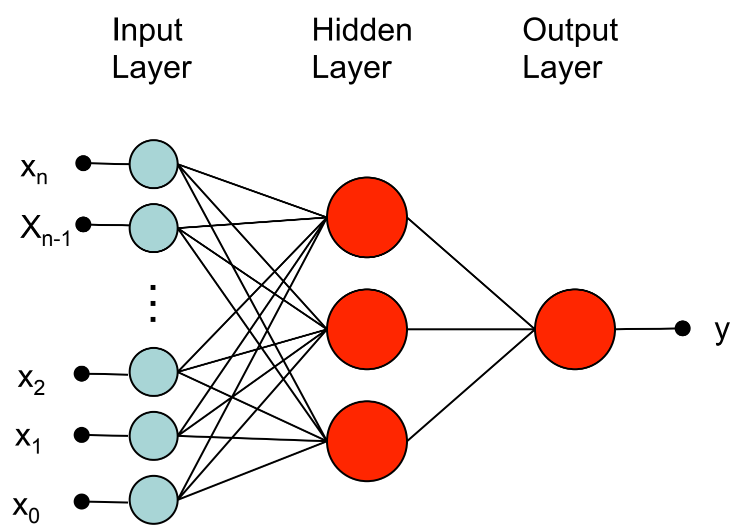
R Demo

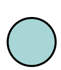
What About Non-Linearity?

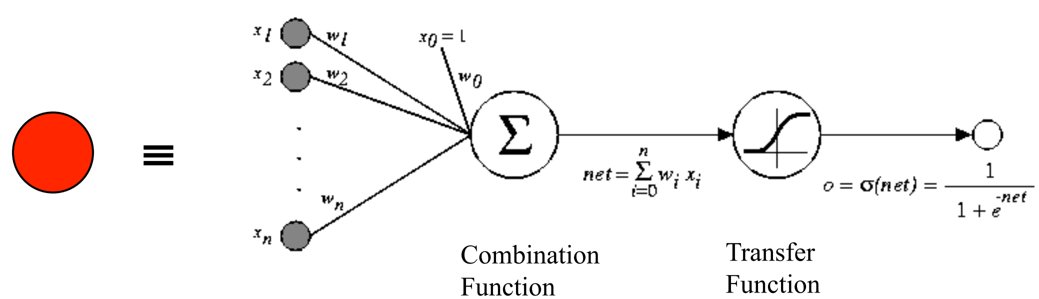


Can we learn this decision surface? …Yes! Multi-Layer Perceptronsl.

Multi-Layer Perceptrons



 ≡ Linear 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

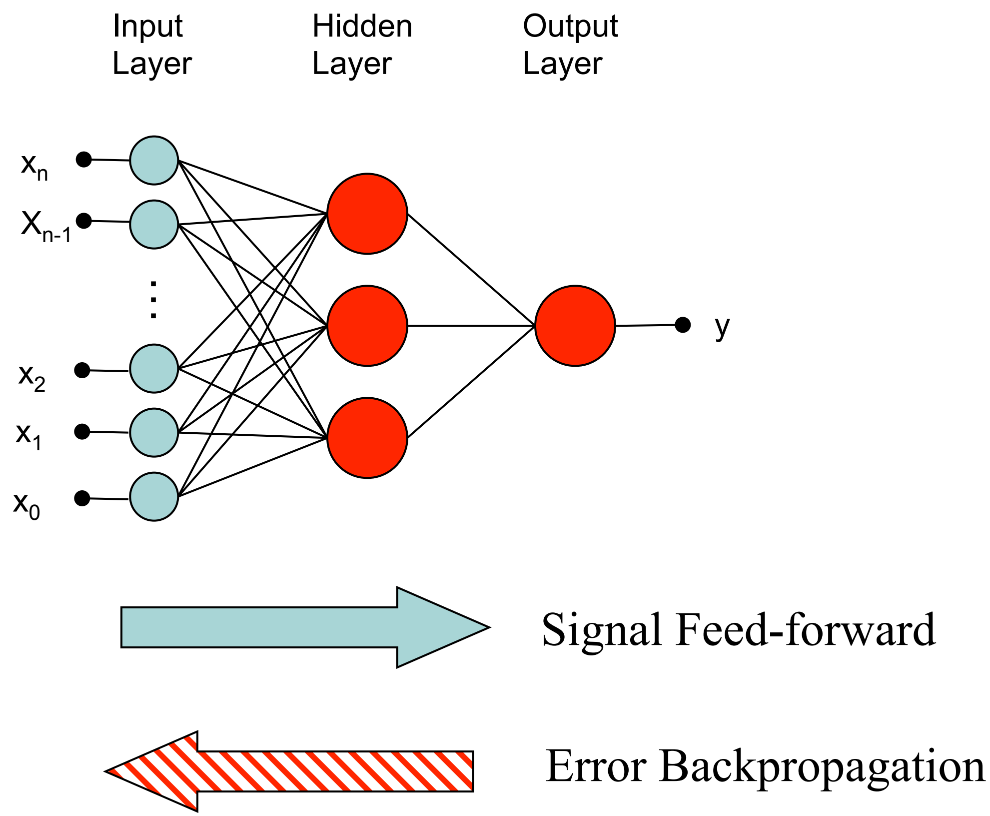
error

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.

Artificial Neural Networks

Feed-forward with Backpropagation

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



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
              t = label for e from training data
              Calculate error Δ = (t - 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 Δ
              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
```

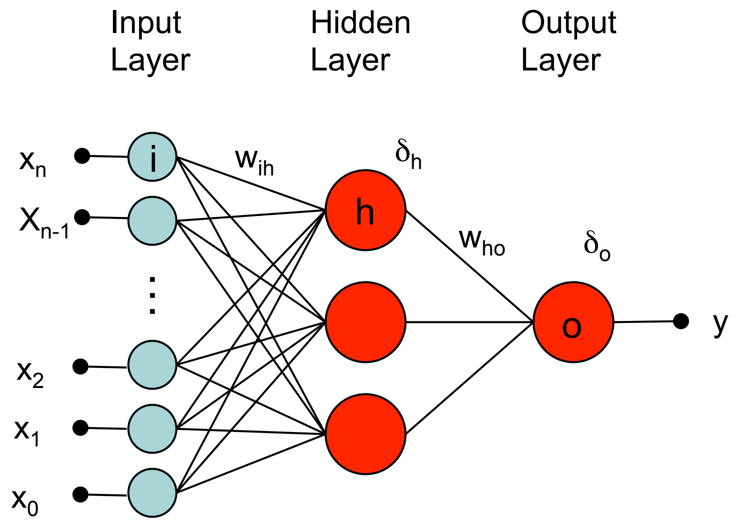
Source: http://en.wikipedia.org/wiki/Backpropagation

Backpropagation

$\Delta = (t - y)^2$

$$\delta_o = y(1-y)\Delta$$
$$w_{ho} \leftarrow w_{ho} + \alpha_o \delta_o$$

$$\delta_h = y(1-y) w_{ho} \delta_o$$
$$w_{ih} \leftarrow w_{ih} + \alpha_h \delta_h$$



This only works because

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

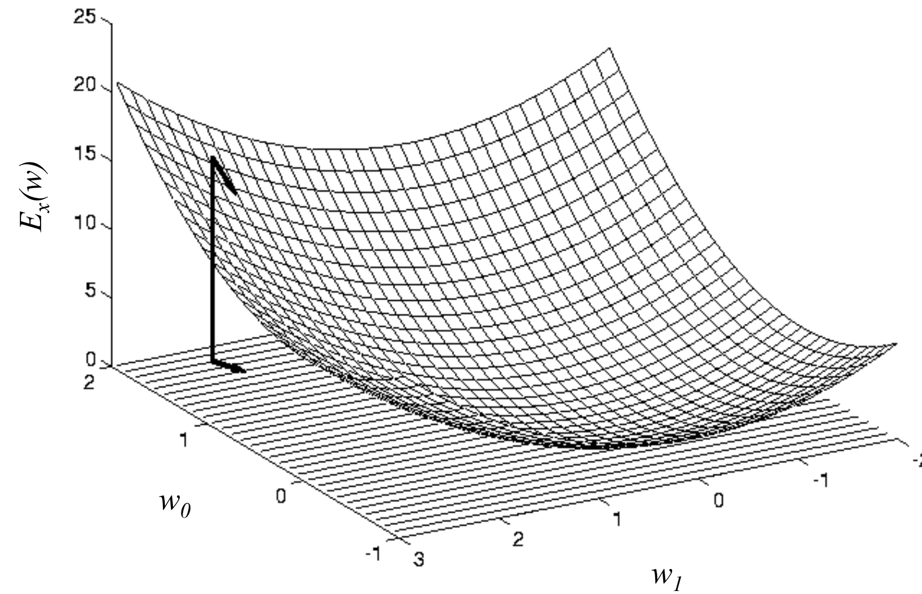
and the output y is differentiable because the transfer function is differentiable. Also note, everything is based on the *rate of change* of the error…we are searching in the direction where the rate of change will minimize the output error.

Neural Network Learning

- Define the network error as

$$\Delta_x = (t - y)^2$$

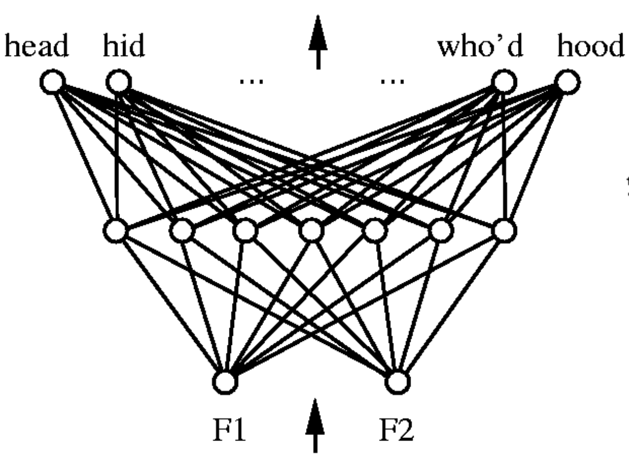
  for some $x \in X$, where i is an index over the output units.
- Let $\Delta_x(w)$ be the error E_x as a function of the weights w.
- Use the gradient (slope) of the error surface to guide the search towards appropriate weights:

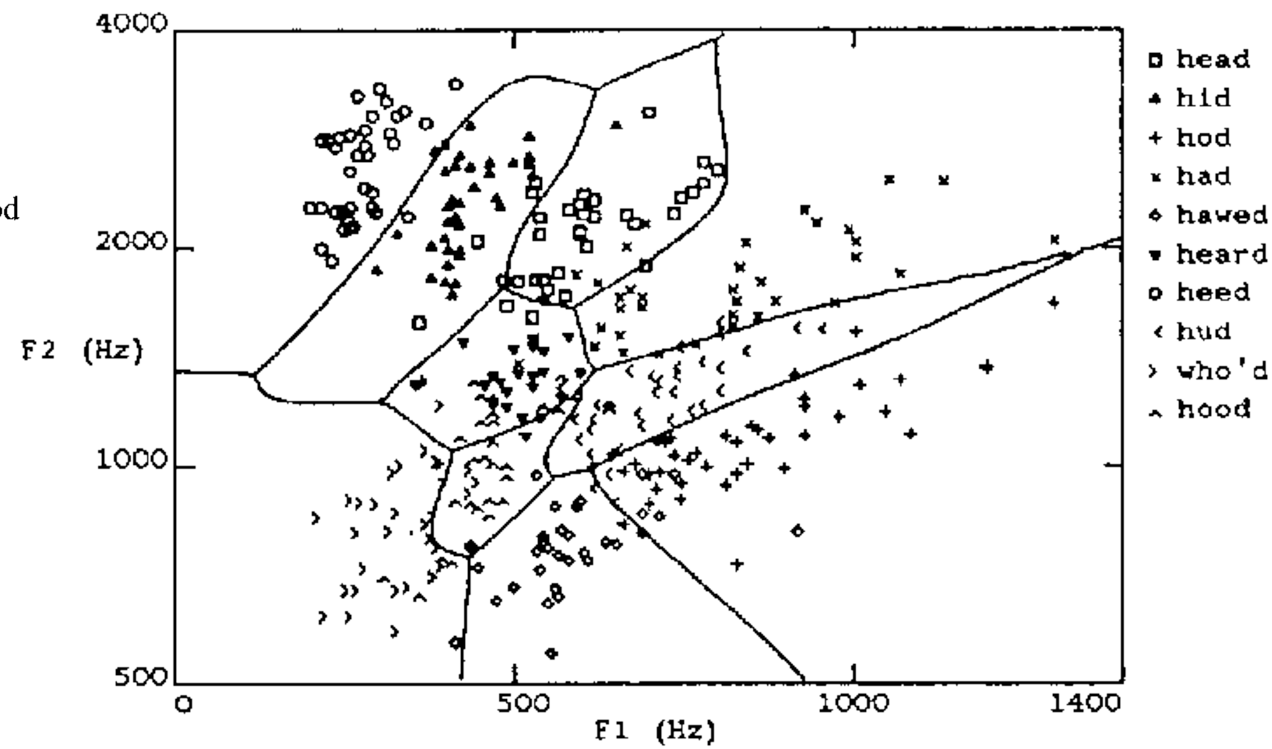


$$\Delta w_k = -\eta \frac{\partial \Delta_x}{\partial w_k}$$

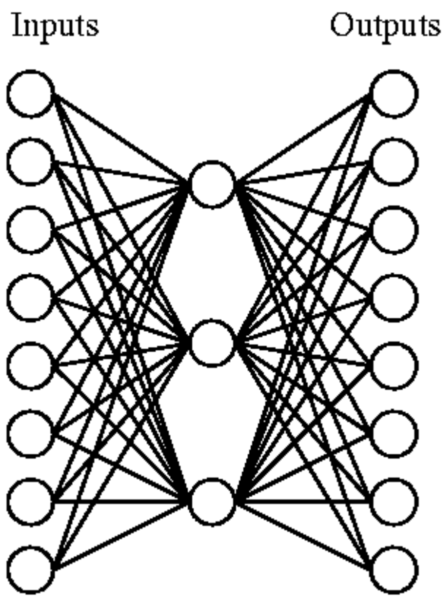
Representational Power

- 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

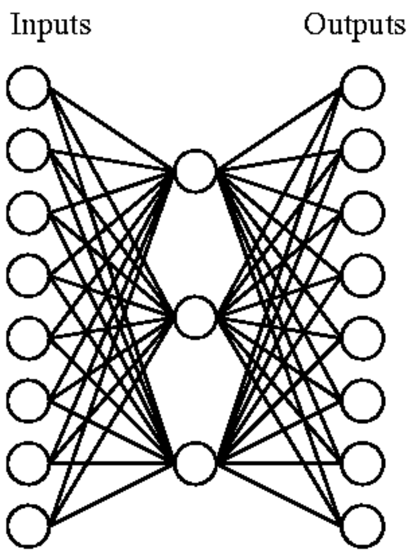


Target Function:

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

Hidden layers allow a network to invent appropriate internal representations.

WEKA Machine Learning

http://www.cs.waikato.ac.nz/ml/weka/