Connectionist Models of Learning

Neural Networks

Characterized by:

- A large number of very simple neuronlike processing elements.
- A large number of weighted connections between the elements.
- Highly parallel, distributed control.
- An emphasis on learning internal representations automatically.

Why Neural Nets?

Solving problems under the constraints similar to those of the brain may lead to solutions to AI problems that would otherwise be overlooked.

- Individual neurons operate very slowly.
  massively parallel algorithms
- Neurons are failure-prone devices.
  distributed representations
- Neurons promote approximate matching.
  less brittle

Neural Networks

Rich history, starting in the early forties.
(McCulloch and Pitts 1943)

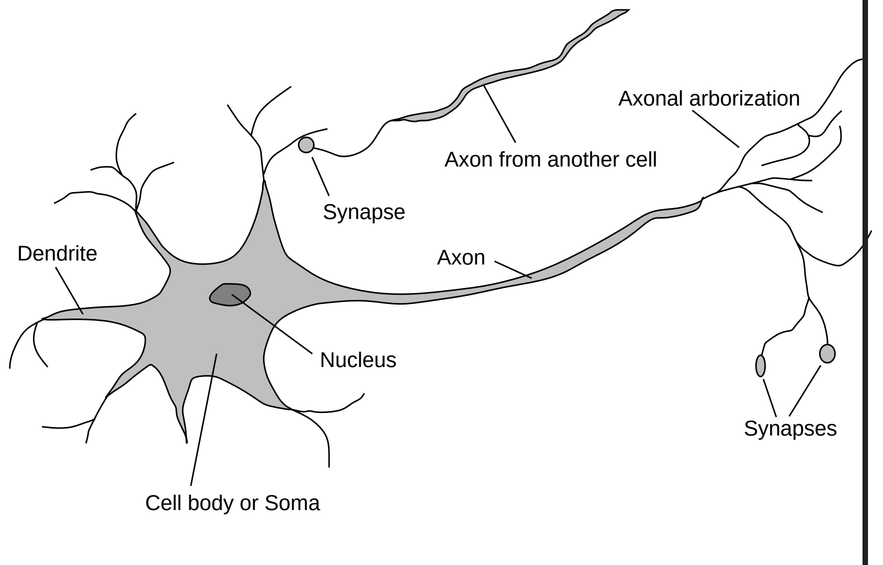
Two views:

- **Modeling the brain**.
- **"Just" representation of complex functions.**

(Continuous; contrast decision trees.)

Much progress on both fronts.

Drawn interest from: *Neuroscience, Cognitive science, AI, Physics, Statistics, and CS / EE.*



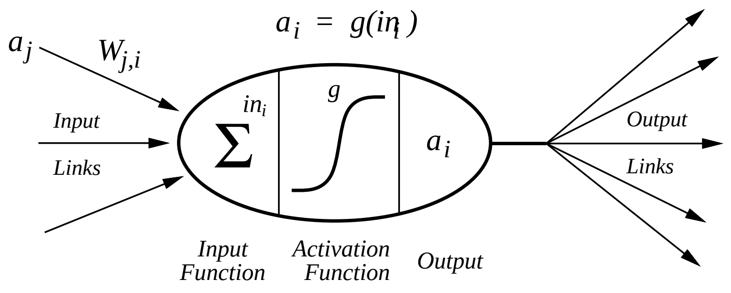
Learning, Massive Parallelism

There is evidence of **learning** — plasticity — at synapses.

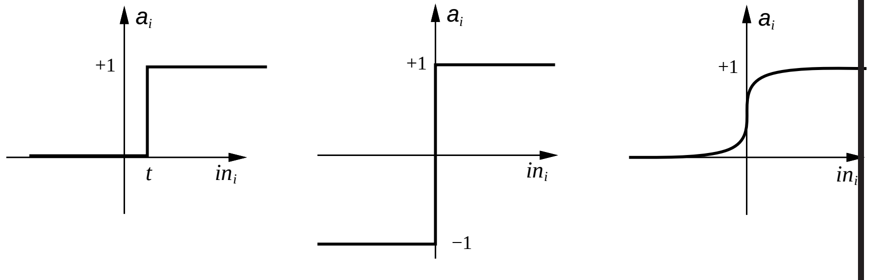
Complexity arises out of connectivity: Neurons perform highly parallel computation.

Idea: collection of simple cells leads to complex behavior: *thought, action, and consciousness ....*

Artificial Neural Networks

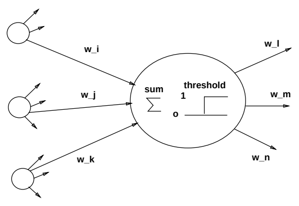


Activation Functions

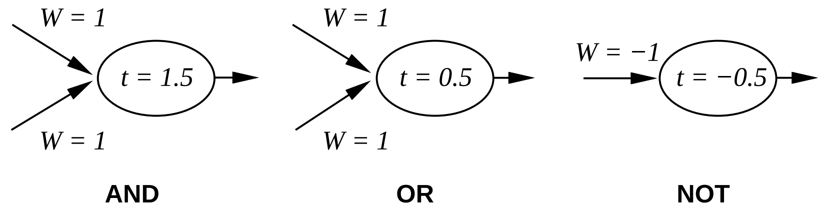


(a) Step function (b) Sign function (c) Sigmoid function

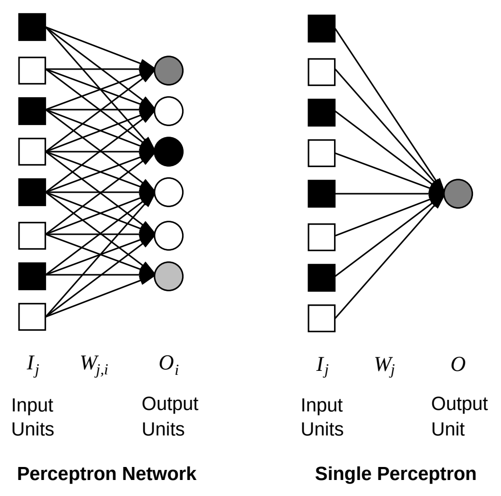
Example



Can Simulate Boolean Gates



Perceptrons



Perceptron Learning Algorithm

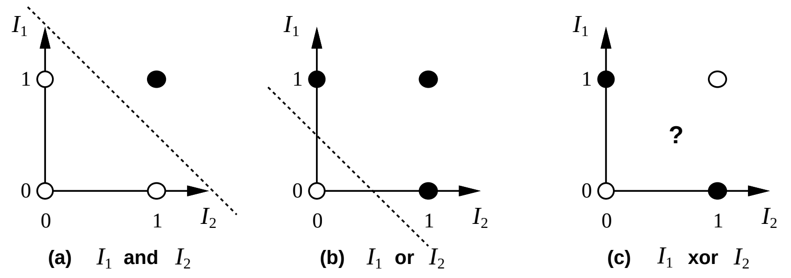
Remarkable learning algorithm: (Rosenblatt 1960)

If function can be represented by perceptron,
then learning algorithm is guaranteed to quickly converge
to the hidden function!

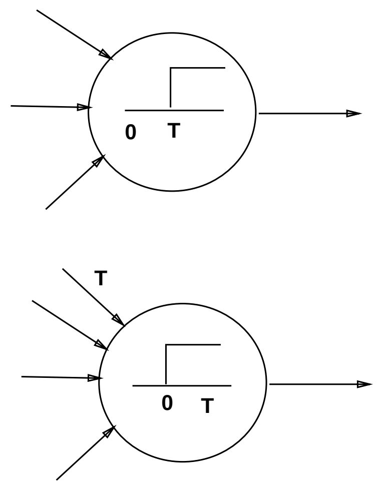
Enormous popularity in early to mid 60’s.

But analysis by Minsky and Papert (1969) showed certain simple functions cannot be represented (e.g. Boolean XOR). Killed the field! (and possibly Rosenblatt (rumored)).

Linearly Separable Functions Only



Learning Threshold Values



Perceptron Learning

A perceptron can learn any linearly separable function, given enough enough training examples.

Key idea: **adjust weights till all examples correct.**

Update weights repeatedly (epochs) for each example.

High Level Algorithm

```
function NEURAL-NETWORK-LEARNING(examples) returns network

  network ← a network with randomly assigned weights
  repeat
      for each e in examples do
          O ← NEURAL-NETWORK-OUTPUT(network, e)
          T ← the observed output values from e
          update the weights in network based on e, O, and T
      end
  until all examples correctly predicted or stopping criterion is reached
  return network
```

Weight Update Function

Single output O; target output for example T.

Define error: $Err = T - O$

Now, just move weights in right direction!

 If error is positive, then need to increase O.

 Each input unit j contributes $W_j \; I_j$ to total input.

  if I_j is positive, increasing W_j tends to increase O

  if I_j is negative, decreasing W_j tends to increase O

So, use: $W_j \leftarrow W_j + \alpha \times I_j \times Err$

Perceptron learning rule (Rosenblatt 1960). α is **learning rate**.

Rule is intuitively correct.

Gradient descent through weight space.

Surprise is **proof** of convergence.

Weight space has **no local minima.**

With enough examples, it will find the function.

(provided α not too large)

Explains early popularity.

Consider learning the logical "or" function.

Our examples are:

```
example      x_1    x_2    x_3     l
===================================
1             0      0     -1      0
2             0      1     -1      1
3             1      0     -1      1
4             1      1     -1      1
```

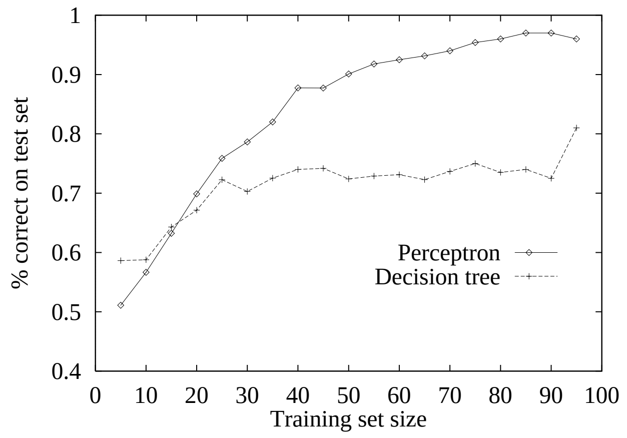
We'll use a single perceptron with three inputs (x_1, x_2, x_3) and single output (l).

Learning rate of 0.5.

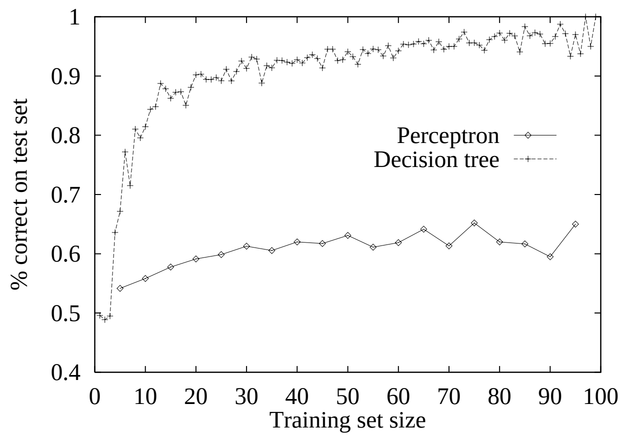
Note artificial input x_3 fixed at -1.

[Step-by-step walk-through is in separate handout.]

Learning Majority Function of 11 Inputs

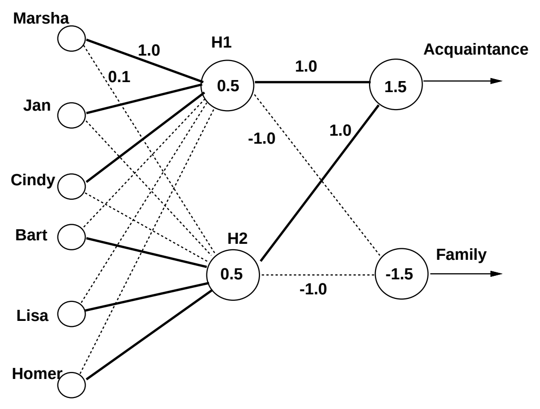


Restaurant Data Set



Complex Feedforward Nets for Classification

Feedforward, layered, fully connected

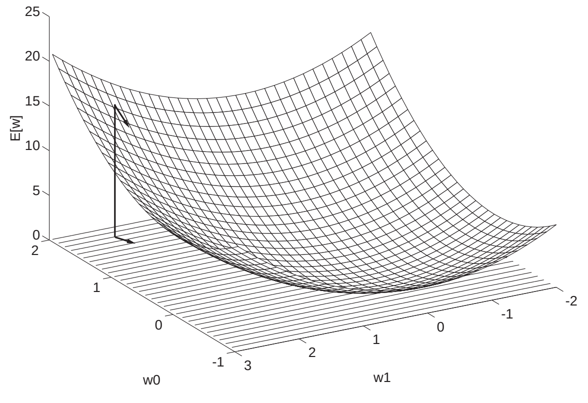


Backpropagation Procedure

Initialize weights. Until performance is satisfactory*,

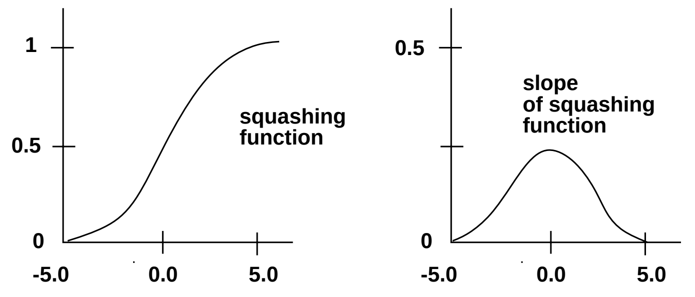
1. Present all training instances. For each one,
   (a) Calculate actual output. (forward pass)
   (b) Compute the weight changes. (backward pass)
      i. Calculate error at output nodes. Compute adjustment to weights from hidden layer to output layer accordingly.
      ii. Calculate error at hidden layer. Compute adjustment to weights from initial layer to hidden layer accordingly.
2. Add up weight changes and change the weights.

Gradient Descent Through Weight Space



Requires a Smooth Threshold Function

The error backpropagation procedure requires a differentiable activation function.



Slope of Sigmoid Function

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

Slope: $\frac{df(x)}{dx} = \frac{d}{dx}\left(\frac{1}{1+e^{-x}}\right)$
$= (1 + e^{-x})^{-2}\, e^{-x}$
$= \frac{e^{-x}}{(1+e^{-x})(1+e^{-x})}$
$= f(x)\frac{e^{-x}}{(1+e^{-x})}$
$= f(x)(1 - f(x))$

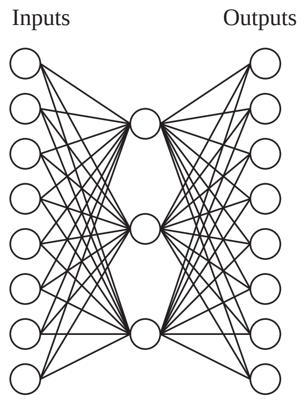
View in terms of output at node:
$= o_j(1 - o_j)$

Adjusting the Weights

Make a large change to a weight, w, if the change leads to a large reduction in the errors observed at the output nodes.

d = desired value at output nodes
o = actual value at output nodes
error $= d - o$

Example



Adjusting the Weights

Let change in $w_{i\to j}$ be proportional to

- the slope of the threshold function at j (i.e., $o_j\,(1 - o_j)$)
- the output at node i (i.e., o_i)
- degree of error at j *(benefit)*
  - output layer: $\beta_z = d_z - o_z$
  - hidden layers: $\beta_j = \sum_k w_{j\to k}\, o_k\,(1 - o_k)\,\beta_k$
- learning rate r

Change to $w_{i\to j}$ should be proportional to $o_i\, o_j\,(1 - o_j)\,\beta_j$.

The Backpropagation Procedure

Pick a rate parameter r.
Until performance is satisfactory,
  For each training instance,

- Compute the resulting output.
- Compute $\beta = d_z - o_z$ for nodes in the output layer.
- Compute $\beta = \sum_k w_{j\rightarrow k}\, o_k(1-o_k)\beta_k$ for all other nodes.
- Compute weight changes for all weights using

$$\Delta w_{i\rightarrow j} = r\ o_i\ o_j(1-o_j)\beta_j$$

Add up weight changes for all training instances, and change the weights.

Backpropagation Algorithm (Mitchell)

Initialize all weights to small random numbers. Until satisfied, do
 For each training example, do

- Input the training example to the network and compute the network outputs
- For each output unit k

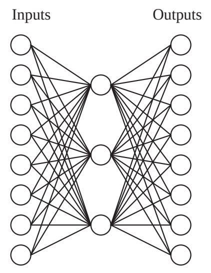
$$\delta_k \leftarrow o_k(1-o_k)(t_k-o_k)$$

- For each hidden unit h

$$\delta_h \leftarrow o_h(1-o_h)\sum_{k\in outputs} w_{h,k}\delta_k$$

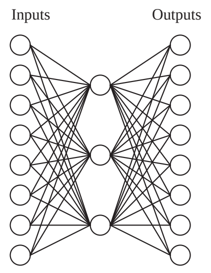
- Update each network weight $w_{i,j}$
  $w_{i,j} \leftarrow w_{i,j} + \Delta w_{i,j}$ where $\Delta w_{i,j} = \eta\delta_j x_{i,j}$

Learning Hidden Layer Representations



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

Learning Hidden Layer Representations



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

Hidden Units

- **Hidden units** are nodes that are situated between the input nodes and the output nodes.
- Hidden units allow a network to learn non-linear functions.
- Hidden units allow the network to represent combinations of the input features.
- Given too many hidden units, a neural net will simply memorize the input patterns.
- Given too few hidden units, the network may not be able to represent all of the necessary generalizations.

When to Consider Neural Networks

- Input is high-dimensional discrete or real-valued (e.g. raw sensor input)
- Output is discrete, real-valued, or a vector of values
- Possibly noisy data
- Form of target function is unknown
- Human readability of result is unimportant

More on Backpropagation

- Gradient descent over entire *network* weight vector
- Easily generalized to arbitrary directed graphs
- Will find a local, not necessarily global error minimum
  - In practice, often works well (can run multiple times)
- Minimizes error over *training* examples
  - Will it generalize well to subsequent examples?
- Training can take thousands of iterations $\rightarrow$ slow!
- Using network after training is very fast

Expressive Capabilities of ANNs

Boolean functions:

- Every boolean function can be represented by network with single hidden layer
- but might require exponential (in number of inputs) hidden units

Continuous functions:

- Every bounded continuous function can be approximated with arbitrarily small error, by network with one hidden layer [Cybenko 1989; Hornik et al. 1989]

Any function can be approximated to arbitrary accuracy by a network with two hidden layers [Cybenko 1988].

Momentum

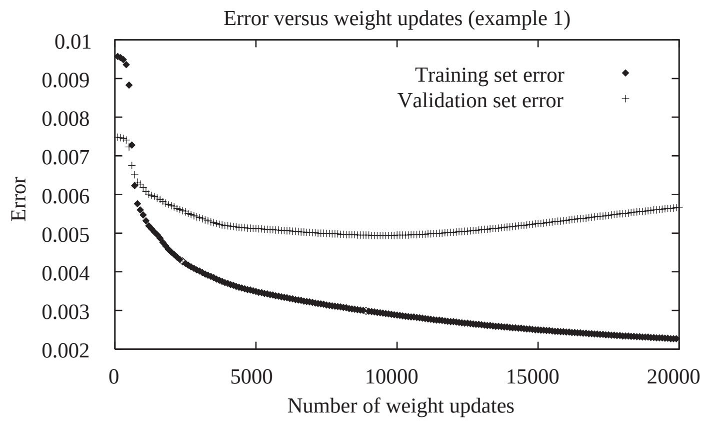
$$\Delta w_{ij}(t+1) = r\ o_i\ o_j(1-o_j)\beta_j + \alpha[w_{ij}(t) - w_{ij}(t-1)]$$

- A momentum factor, α, makes the n^{th} weight change partially dependent on the $(n-1)^{th}$ weight change. α ranges between 0 and 1.
- Momentum tends to keep the weight moving in the same direction, thereby improving convergence.
- Tends to increase the step size in regions where the gradient is unchanging, speeding convergence.
- Tends to avoid getting caught in small local minima and in oscillations about local minima.

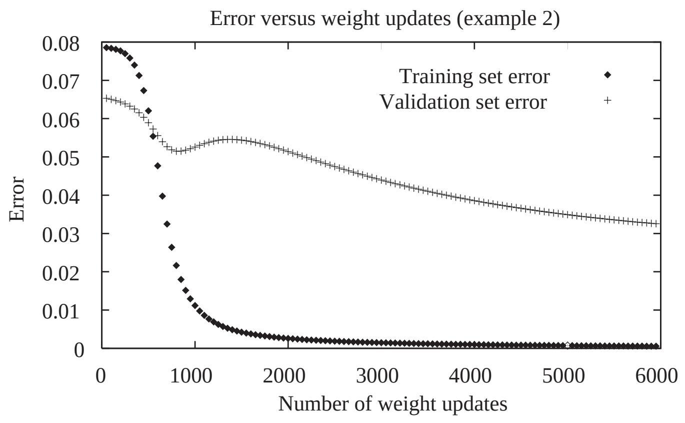
How long should you train the net?

- The goal is to achieve a balance between correct responses for the training patterns and correct responses for new patterns. (That is, a balance between memorization and generalization.)
- If you train the net for too long, then you run the risk of overfitting.
- In general, the network is trained until it reaches an acceptable error rate (e.g., 95%).

Overfitting in ANNs



Overfitting in ANNs



Implementing Backprop – Design Decisions

1. Choice of r
2. Stopping criterion – when should training stop?
3. Network architecture
   (a) How many hidden layers? how many hidden units per layer?
   (b) How should the units be connected? (Fully? Partial? Use domain knowledge?)