Neural Networks

What is Connectionism?

- Neurally-Inspired approach to cognitive modeling.
  - Units are like neurons, which compute simply
  - Connections are like synapses with weights
  - Input function combines outputs from other units
  - Output function is function of input and state
  - Learning function adjusts weights over time

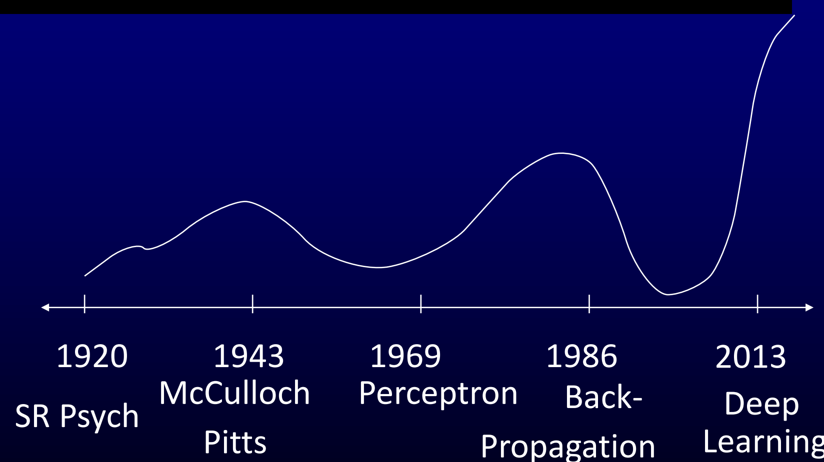
Alternatives Models of Mind

- Mind as Steam Engine
- Mind as Switching Network
- Mind as Computer program
- Mind as automobile
- Mind as Ecology/economy
- Mind as Immune System
- *Mind as Brain*
- Mind as Weather

The Connectionist Aesthetic

- Units and connections are not dynamically created or destroyed quickly
- It should only take 100 cycles to accomplish interesting tasks
  - Neurons are slow (1-10 ms)
  - Intelligence is fast (100ms-1s)
- What does this buy?
  - Parallel hardware plausibility
  - Linkage to neuroscience

History of Connectionism



High Hopes & Hubris 1943-1969

– Networks were part of the establishment of computational and information theories. Brain was imagined as complex automata. Von Neuman, Shannon, Wiener, Minsky, all worked on Neural Networks.

Symbolic Seventies 1969-1981

- Very low interest from AI and computer science researchers. Very little funding. Continued work in Cybernetics (europe), mathematical biology, mathematical psychology

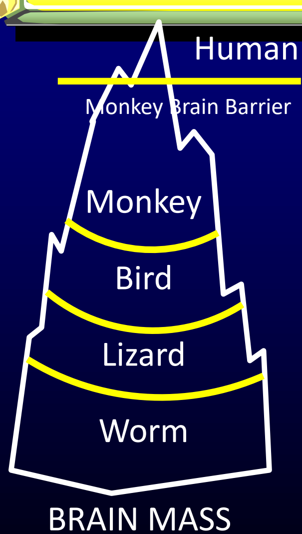
Connectionism is a Moving Target

- Justifications change over time
  - Parallel Processing, Learning, Neuro-realism, cognitive modeling, cheap GPU hardware, old AI stinks, etc.
- Styles of neural networks change over time
  - Parallel versus serial, hand-programmed versus machine-learned, theoretical networks versus practical networks, etc.
- Whats going on?

New View of Connectionism

- The search for a new EMBODIMENT OF MIND Which is not dependent on sequential rule-application in a recursive framework.
- As models are explored and their limits discovered, and as new neuroscientific data is established, we continually throw out our partial successes.

Why study Neural-like un-symbolic processing?



Conscious Deliberative Intelligence is New!

First 97% of evolution created

Sapiens who could survive.

Only Last 3% is Symbolic

Minsky & Papert 69, 72, 88...

- General Principle for Machine Learning?

"No machine can learn to recognize X unless it possesses, at least potentially, some scheme for representing X"

"Perceptrons had no way to represent knowledge"

Born Again Connectionism 1980-1990's

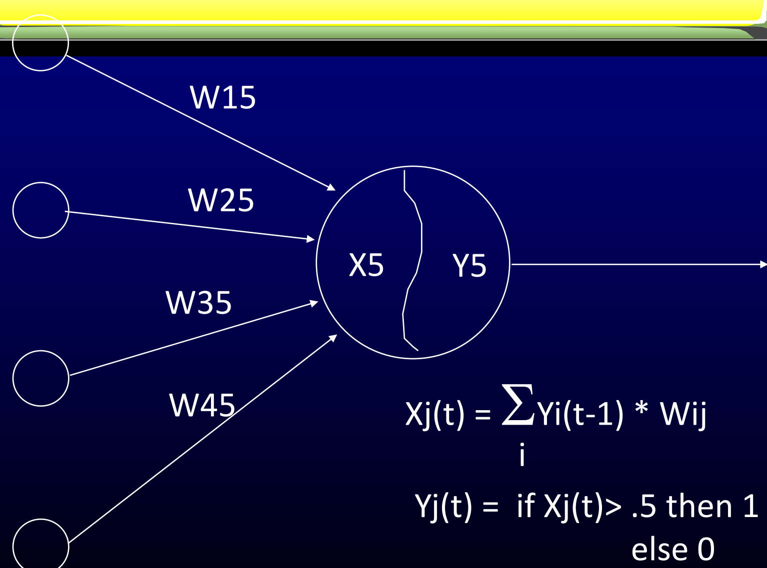
- Stimulated by curious results, politically powerful physicists, disaffected AI'ers, neuroscientific results, Connectionism boomed in the late 80's with interests from biology, physics, EE, CS, Psychology, and Medicine.

- By 2000 a range of neural techniques are part of everyone's AI or "data science" toolkit, but we did not seeing significant scaling up until Deep Learning in 2012

Neural Network Framework

- Set of simple processing units each with some "state" (a number)
- Weighted connections
- simple combination input function
- simple computation output function
  - –although this is inspired by the brain, no one has any idea of how "mind" emerges...

Example networked unit



$$Xj(t) = \sum_{i} Yi(t-1) * Wij$$

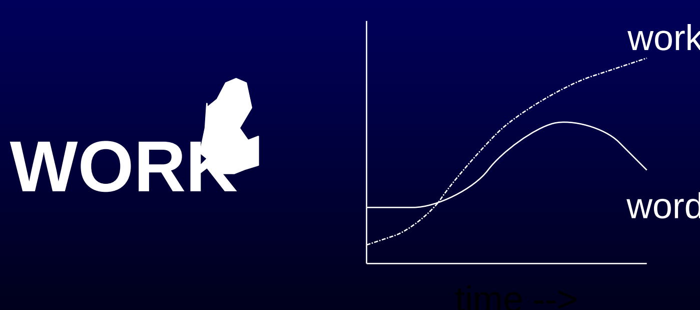
Yj(t) = if Xj(t)> .5 then 1
else 0

NN Models in two Categories

- Recurrent/dynamic systems
  - Systems with "state"
  - Move from state to state driven by input or by time
    - good learning algorithms for recurrent networks are rare
- Feedforward associationist/classification systems
  - Systems have no state, transform inputs to outputs
  - Learning algorithms abound, with different characteristics
  - Today we will focus on feed-forward networks

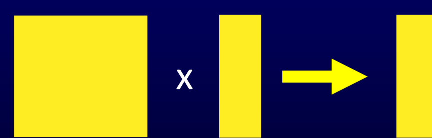
McClelland & Rumelhart 1981 Interactive Activation Model

- Static Network with fixed weights
- Bounded Linear combinations of inputs
- Proportional Updating (non-linear)



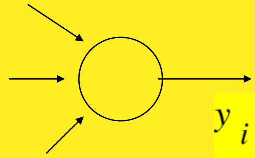
Matrix Model of network

- weights represented as matrix
- State represented as Vector
- Add nonlinear filter (squash, limit, etc) before recurrence



Perceptrons:
Threshold Logic that Learns

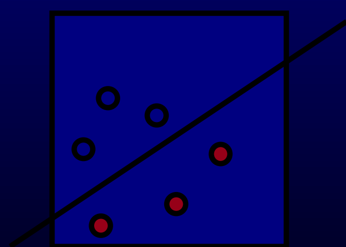
- Binary Units
- Compares lin. comb. of inputs to threshold
- proven rule for adjusting weights
- only one layer of weights



$$y_i = 1 \text{ if } T_i < \sum_j W_{ji} y_j$$

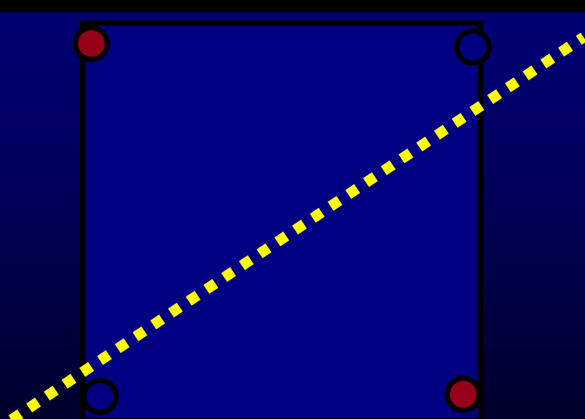
Perceptrons can do Inductive Classification (limited)

- If a testcase puts out a 0 when it should put out a 1, increase weights
- If it puts out 1 when it should put out 0, decrease weights



unfortunately, the set of learnable problems by perceptrons is very small, and it doesnt generalize to multiple levels

Perceptron can do 14/16 boolean functions of 2 inputs!



Exclusive Or is not Linearly Separable,
and neither are most higher order predicates!

Back Propagation of Error Signals

–Rumelhart/Hinton/Williams

–Other claimants: Parker/Lecun/Werbos

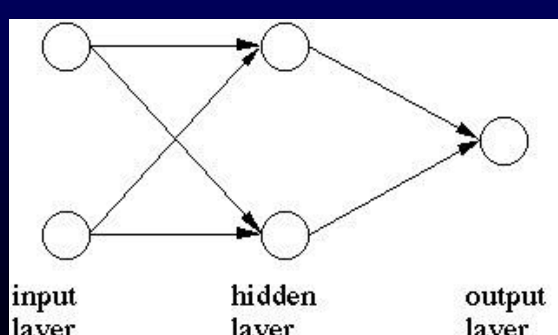
–Relaxed Perceptron Binary Threshold to Continuous "sigmoid" function

–Showed how learning could be stable in multiple layers of perceptrons

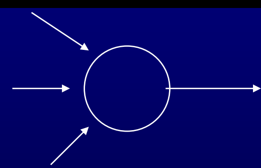
–Large demonstrations (NetTalk) on TV

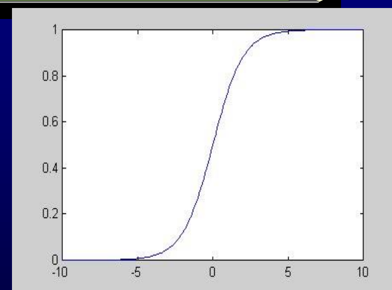
Breakdown of discrete MLP

- A small change in a weight in level 1
- Triggers a change in output
- Invalidates the weights in layer 2.



Back Propagation





-
- Computes lin. comb. of inputs
- Output is sigmoid of input
  1/ 1 + e ^ -x
-

Back prop

- Intuitive Idea: Distribute *Blame* for Error to Previous Layers
- Algorithm *Train-by-Backprop* (*D*, *r*)
  - Each training example is a pair of the form $\langle x, t(x) \rangle$, where x is the vector of input values and $t(x)$ is the output value. r is the learning rate (e.g., 0.05)
  - Initialize all weights w_i to (small) random values
  - UNTIL the termination condition is met, DO

    FOR each $\langle x, t(x) \rangle$ in D, DO

    Input the instance x to the unit and compute the output $o(x) = \sigma(net(x))$

    FOR each output unit k, DO

    $$\delta_k = o_k(x)(1 - o_k(x))(t_k(x) - o_k(x))$$

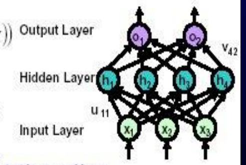
    FOR each hidden unit j, DO

    $$\delta_j = h_j(x)(1 - h_j(x)) \sum_{k \in outputs} v_{j,k}\delta_j$$

    Update each $w = u_{i,j}$ ($a = h_j$) or $w = v_{j,k}$ ($a = o_k$)

    $$w_{start\text{-}layer,\ end\text{-}layer} \leftarrow w_{start\text{-}layer,\ end\text{-}layer} + \Delta w_{start\text{-}layer,\ end\text{-}layer}$$

    $$\Delta w_{start\text{-}layer,\ end\text{-}layer} \leftarrow r\,\delta_{end\text{-}layer}\, a_{end\text{-}layer}$$
  - RETURN final u, v



KSU

Back Propagation

- Basic method for "learning" weights in multi-layer networks
- Can develop representations in "hidden" units
- A form of hill-climbing or gradient descent
  - Why does it work so well? nobody knows, but its related to many statistical methods
- Many applications
  - curve-fitting
  - logical function inference
  - inductive classification
- Many variants and improvements

Back Propagation of Errors

- Input
  - matched set of input/desired-output cases
  - conforming multiple-layer network randomly initialized
- Process
  - FORWARD PASS: Run each input to generate output
  - BACKWARD PASS: ERROR= (desired output - actual output)^2
    - partial derivative for each weight and sum over all cases
  - UPDATE WEIGHTS: Change weights to decrease error

Details

- two learning parameters MU and ALPHA
  - control velocity and acceleration of changes
- bias node
  - hidden and output nodes have a link from a unit which is always 1
  - This acts like a threshold in perceptrons
- Epoch versus Immediate training
  - update after all training cases or after each case
  - Modern DL uses Stochastic Gradient Descent
    - random sub-batches before update

Things to do with BP nets

- logical function
- classification
- compress data
- Function Interpolation

BACKPROP in LISP

```
(defstruct bp-node
  in-list
  (input 0.0 :type float)
  (output 0.0 :type float)
  (delta-in 0.0 :type float)
  (delta-out 0.0 :type float))

(defstruct bp-link
  from-node
  (weight 0.0 :type float)
  (delta 0.0 :type float)
  (previous-delta 0.0 :type float))
```

setup a network

```
(defun setuplayers (layers &aux *levels* *links* *biasnode*)
 (setf *biasnode* (make-bp-node :output 1.0))
 (setf *levels* (loop for i in layers collect
                 (loop for j from 1 to i collect (make-bp-node))))
 (loop for level on *levels* do
     (and (cdr level)
          (loop for y in (cadr level) do
                (push (make-bp-link :from-node *biasnode*) *links*)
                (push (car *links*) (bp-node-in-list y))
                (loop for x in (car level) do
                   (push (make-bp-link :from-node x) *links*)
                   (push (car *links*) (bp-node-in-list y))))))
   (noise (cons *levels* *links*) .5)
 (cons *levels* *links*))
```

set links to random values

- (defun noise (network n)
- "Small ± weights."
- (loop for link in (cdr network) do
- (setf (bp-link-weight link)
- (/ (- (random (* 200 n)) (* 100.0 n)) 100.0))))

The Forward Pass

- (defun forward-pass (network in-list &optional flag)
- ;; set inputs
- (loop for node in (caar network) as i in in-list do
- (setf (bp-node-output node) (float i)))
- ;; do loop forward pass
- (loop for level in (cdar network) do
- (loop for node in level do
- (getinput node)
- (setoutput node)))
- ;;flag to generate output values
- (and flag (loop for node in (car (last (car network))) collect
- (bp-node-output node))))

Forward Pass routines

```
(defun getinput (node)
  (setf (bp-node-input node) 0.0)
  (loop for link in (bp-node-in-list node) do
       (incf (bp-node-input node)
            (* (bp-link-weight link)
                 (bp-node-output (bp-link-from-node link))))))

;;subroutine for forward pass
(defun setoutput (node)
  (setf (bp-node-output node) (sigmoid (bp-node-input node))))
```

Backward Pass

```
(defun backward-pass (network desired tol edge-tol &optional addflag
                               &aux (error nil))
  "Given a desired set of values and a cutoff, perloop forms a backward pass"
  ;; clear input values
  (loop for level in  (car network) do
      (loop for node in level do
           (setf (bp-node-delta-out node) 0.0)))
  ;; set inputs
  (loop for node in (car (last (car network))) as d in desired do
      (setf error (or (< (or (and (floatp d) tol) edge-tol)
                           (abs (setf (bp-node-delta-out node)
                                      (- (bp-node-output node) (or d (bp-node-
output node))))))
                     error))
```

Backward Pass (Continued)

```
•       (setdeltain node)
•       (errorprop node addflag))
•    ;; unless the output is good enough, do the backward pass
•    (and *indiv-trace-flag* (write-char (or (and error #\*) #\.)))
•    (cond
•     ((or error *always-bp-flag*)
•      ;; do backward pass
•      (bpfromtail (car network) addflag)))
•    (cond (error
•           ;; compute error
•           (loop for node in (car (last (car network))) sum
•               (* (bp-node-delta-out node) (bp-node-delta-out node))))
•          (t 0.0)))
```

Backward Pass routines

```
•  (defun bpfromtail (levels addflag)
•    (and (cddr levels) (bpfromtail (cdr levels) addflag))
•    (loop for node in (car levels) do
•        (setdeltain node)
•        (errorprop node addflag)))

•  ;;subroutine for bpfromtail
•  (defun setdeltain (node)
•    (setf (bp-node-delta-in node)
•         (* (bp-node-delta-out node)
•            (bp-node-output node)
•            (- 1.0 (bp-node-output node)))))
```

Backward Pass Routines

```
(defun errorprop (node &optional addflag)
  (loop for link in (bp-node-in-list node) do
     (cond (addflag (incf (bp-link-delta link)
                          (* (bp-node-delta-in node)
                             (bp-node-output (bp-link-from-node link)))))
        (t (setf (bp-link-delta link)
                 (* (bp-node-delta-in node)
                    (bp-node-output (bp-link-from-node link))))))
     (incf (bp-node-delta-out (bp-link-from-node link))
        (* (bp-link-weight link)(bp-node-delta-in node)))))
```

main routine

```
(defun learn (network in-list out-list
              &optional (mu 0.3) (alpha 0.9) (min-dif 0.2) (limit 1000)
              &aux (error -1))
  "Uses in-list and out-list to train the network."
  ;; initial value loop for momentum
  (loop for link in (cdr network) do
     (setf (bp-link-previous-delta link) 0.0))
  ;; now do the iterations
  (format t "~%")
  (loop for cycle from 1 to limit until (zerop error) do
     (loop for link in (cdr network) do
         (setf (bp-link-delta link) 0.0))
```

Main Routine (continued)

```
;;now do the forward and backward passes
    (setq error
        (loop for x in in-list as y in out-list
              sum (prog2
                      (forward-pass network x)
                      (backward-pass network y min-dif min-dif t))))
    (format t "Cycle: ~d  Error: ~d~%" cycle error)
    ;;now update weights
    (loop for link in (cdr network) do
       (incf (bp-link-weight link)
             (- (setf (bp-link-previous-delta link)
                      (+ (* mu (bp-link-delta link))
                         (* alpha (bp-link-previous-delta link))))))))
```