HFT: Ch 11

Artificial Neural Networks

R Greiner

Cmput 466 / 551

Thanks: T Dietterich, R Parr, J Shewchuk

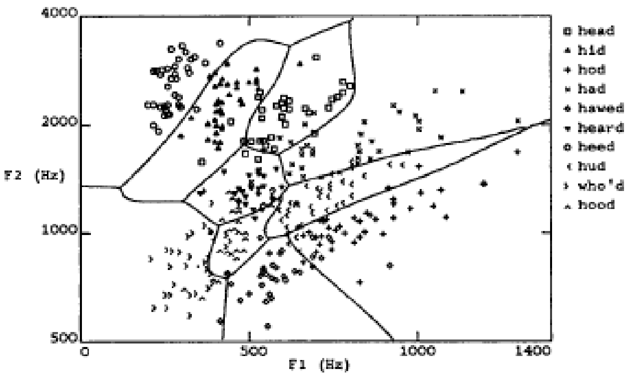
"An Intro to Conjugate Gradient Method without Agonizing Pain"

Outline

- Introduction
  - Historical Motivation, non-LTU, Objective
  - Types of Structures
- Multi-layer Feed-Forward Networks
  - Sigmoid Unit
  - Backpropagation
- Tricks
  - Line Search
  - Conjugate Gradient
  - Alternative Error Functions
- Hidden layer representations
  - Example: Face Recognition
- Recurrent Networks

Motivation for non-Linear Classifiers

- Linear methods are “weak”
  - ☐ Make strong assumptions
  - ☐ Can only express relatively simple functions of inputs



- Need to learn more-expressive classifiers, that can do more!
  - ☐ What does the space of hypotheses look like?
  - ☐ How do we navigate in this space?

Skip

Non-Linear ⇒ Neural Nets

- Linear separability depends on FEATURES!!

  A function can be
  - not-linearly-separable with one set of features,
  - but linearly separable in another

- Have system to produce features,
  that make function linearly-separatable

- … neural nets …

Why “Neural Network”

- Brains – network of neurons – are only known example of actual intelligence
- Individual neurons are slow, boring
- Brains succeed by using massive parallelism
- Idea: **Use for building approximators!**

- Raises many issues:
  - Is the computational metaphor suited to the computational hardware?
  - How to copy the important part?
  - Are we aiming too low?

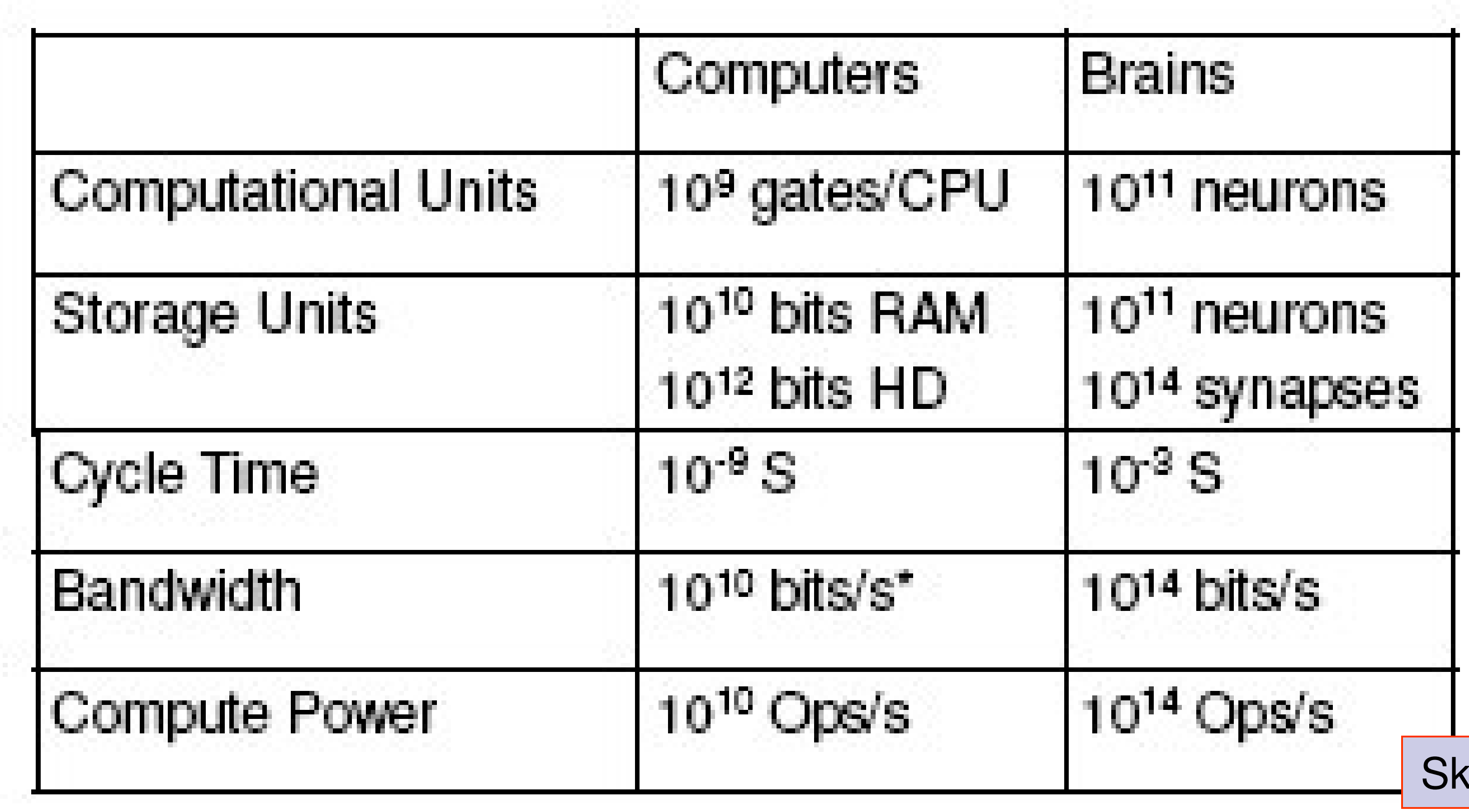
Artificial Neural Networks

- Develop abstraction of function of actual neurons
- Simulate large, massively parallel artificial neural networks on conventional computers
- Some have tried to build the hardware too
- Try to approximate human learning, robustness to noise, robustness to damage, etc.

Skip

Comparison…

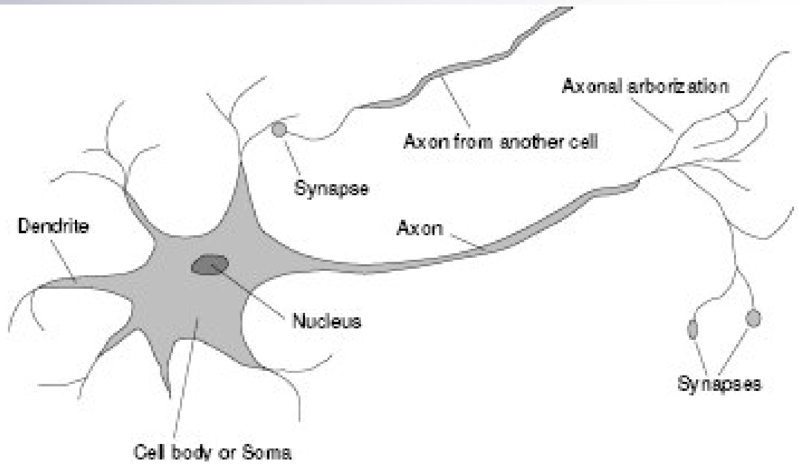
Maybe computers should be more brain-like:



	Computers	Brains
Computational Units	10^9 gates/CPU	10^{11} neurons
Storage Units	10^{10} bits RAM 10^{12} bits HD	10^{11} neurons 10^{14} synapses
Cycle Time	10^{-9} S	10^{-3} S
Bandwidth	10^{10} bits/s*	10^{14} bits/s
Compute Power	10^{10} Ops/s	10^{14} Ops/s

Skip

Natural Neurons

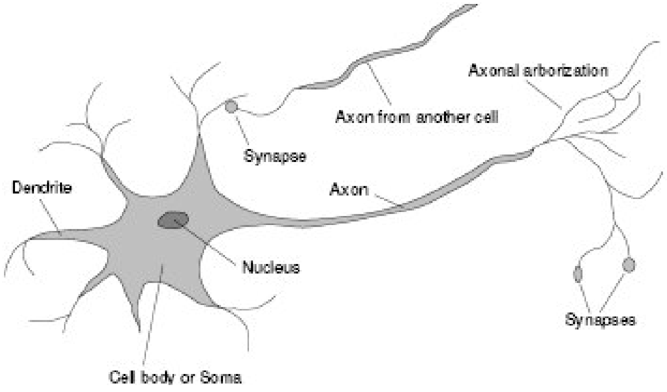


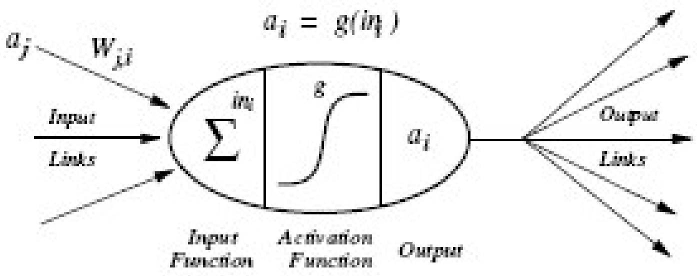
- Neuron switching time ≈0.001 second
- Number of neurons ≈10^{11}
- Connections per neuron ≈ 10^{4-5}
- Scene recognition time ≈0.1 second
- Only time for ≈100 inference steps
  - not enough if only 1 operation/time

⇒ much parallel computation

Skip

Natural, vs Artificial, Neurons



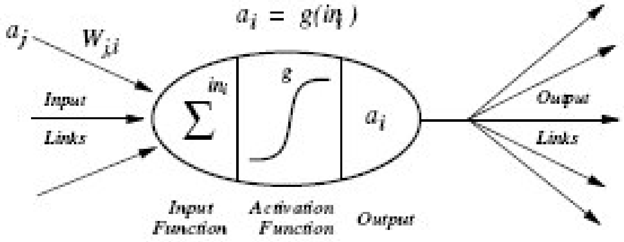


Properties of artificial neural nets (ANN's):

- Many neuron-like threshold switching units
- Many weighted interconnections among units
- Highly parallel, distributed process
- Emphasis on tuning weights automatically

Artificial Neural Networks

- Mathematical abstraction!
- **Units**, connected by **links**; with **weight** $\in \Re$
- Each unit has
  + set of inputs links from other units
  + set of output links to other units
  . . . computes activation at next time step



- Lots of simple computational unit

  ⇒ massively parallel implementation
- Non-Linear function approximation

  One of the most widely-used learning methods

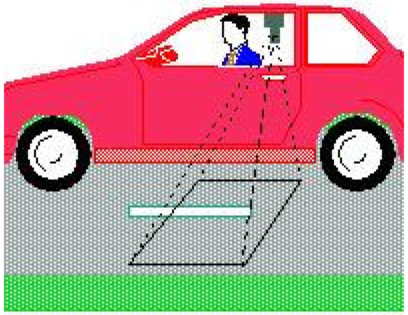
> "… neural nets are the second best thing for learning anything!" J Denker

Artificial Neural Networks

- Rich history, starting in early forties (McCulloch/Pitts 1943)
- Two views:
  - ☐ Modeling the brain
  - ☐ “Just” rep'n of complex functions
- Much progress on both fronts
- Interests from:

  Neuro-science, Cognitive science, Physics, Statistics, Engineering, CS / EE, ... and AI

Uses of Artificial Neural Nets



- Trained to drive
  - No-hands across America (Pomerleau)
  - ARPA Challenge (Thrun)
- Trained to pronounce English (NETtalk)
  - Training set: Sliding window over text, sounds
  - 95% accuracy on training set
  - 78% accuracy on test set
- Trained to recognize handwritten digits
  - >99% accuracy

Applications of Neural Nets

Learn to. . .

- Control
  - drive cars
  - control plants
  - pronunciation: NETtalk … mapping text to phonemes
  - . . .
- Recognize/Classify
  - handwritten characters
  - spoken words
  - images (eg, faces)
  - credit risks
  - . . .
- Predict
  - Market forecasting
  - Trend analysis
  - ...

Skip

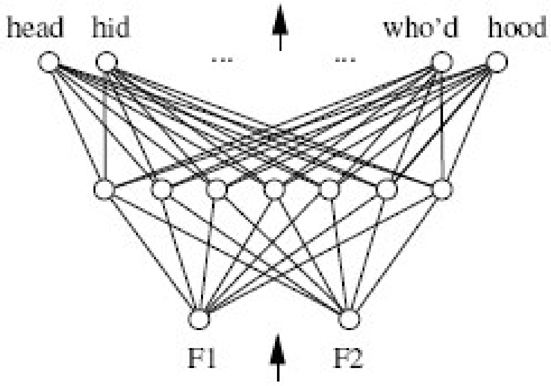
Neural Network Lore

- Neural nets have been adopted with an almost religious fervor within the AI community
  … several times
- Often ascribed near magical powers by people…
  - usually people who know the *least* about computation or brains ☺
- For most AI people, magic is gone…
  but neural nets remain extremely interesting and useful mathematical objects

Skip

When to Consider Neural Networks

- Input is
  - high-dimensional (attribute-value pairs)
  - discrete or real-valued
  - possibly noisy [training, testing]
  - complete
  - (eg, raw sensor input)
- Output is
  - vector of values
  - discrete or real valued
  - “linear ordering"



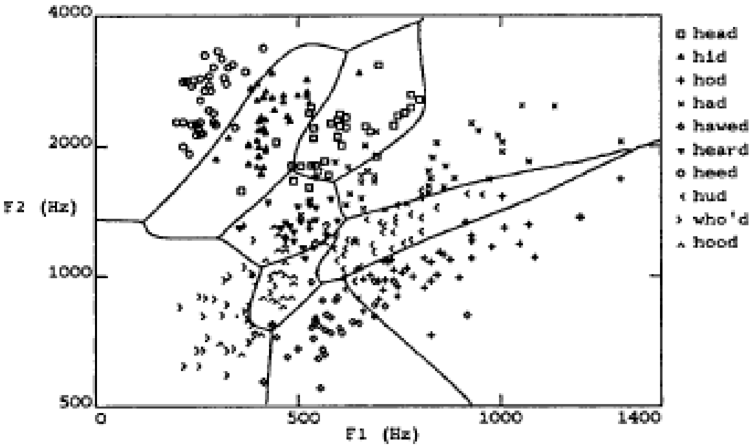
$\Rightarrow \Re^n \to \Re$

- . . . have LOTS OF TIME to train (performance is fast)
- Form of target function is unknown

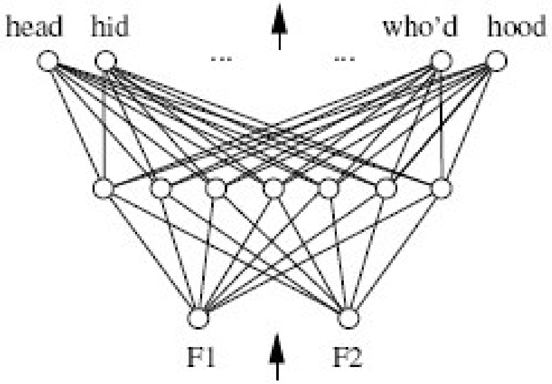
- Human readability / Explanability is NOT important

Multi-Layer Networks

- Perceptrons GREAT if want SINGLE STRAIGHT SURFACE
- What about . . .



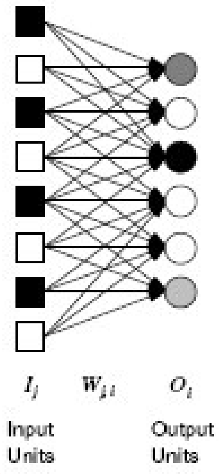
- Need NETWORK of nodes. . .



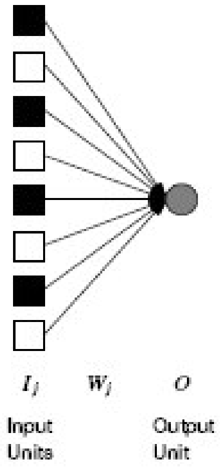
Skip

Types of Network Structures

- Single layer:
    - Linear Threshold Units
    - Linear Units, Sigmoid Units

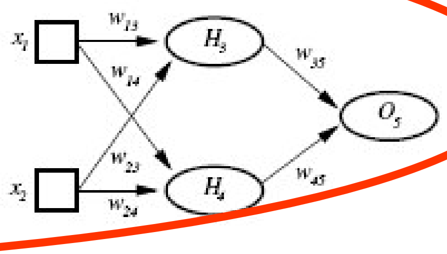


Perceptron Network



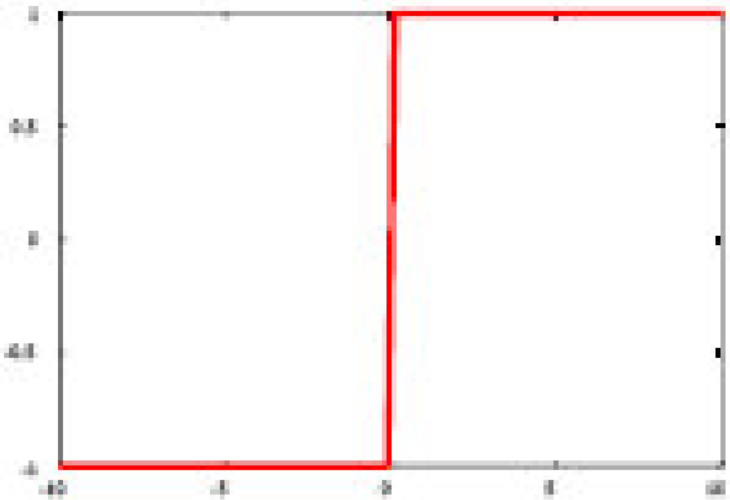
Single Perceptron

- General multi-layered feed-forward:
    - input / hidden units / output

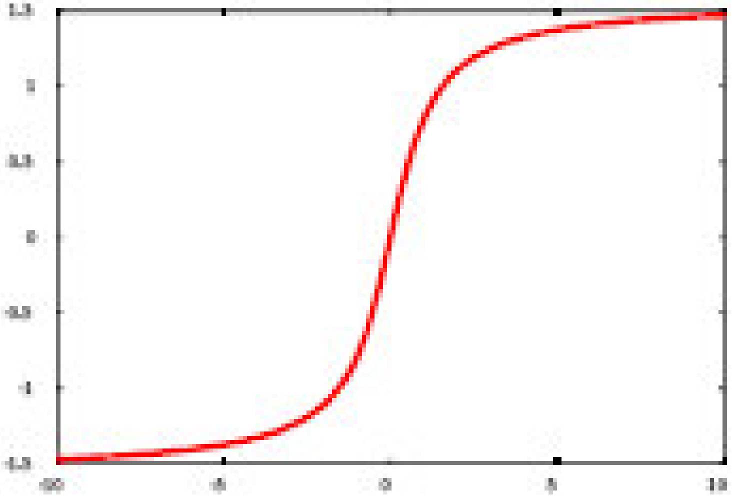


- Recurrent + Cycles, to allow “state”
    - Hopfield networks (used for associative memory), Boltzmann machines, . . .

Threshold Functions

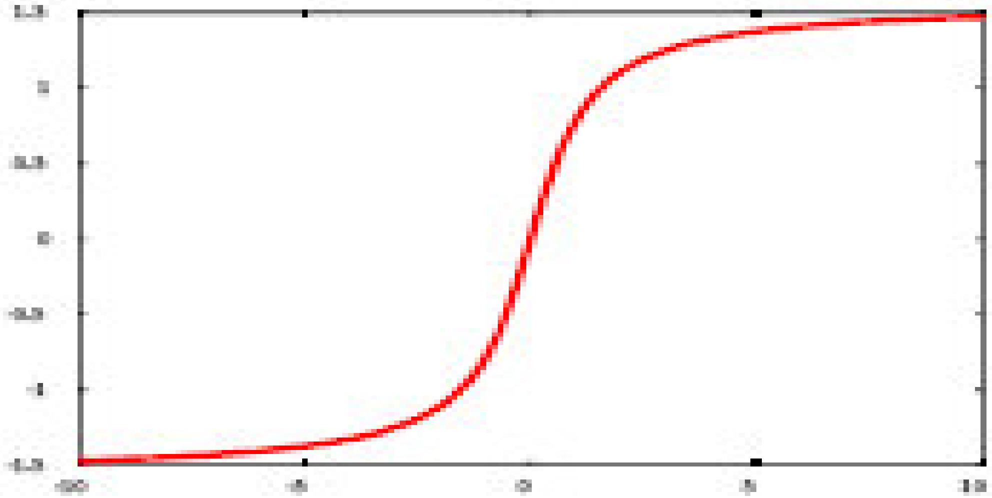


g(x) = sign(x)
(perceptron)



g(x)=tanh(x) or 1/(1+exp(-x))
(logistic regression; sigmoid)

Sigmoid Unit

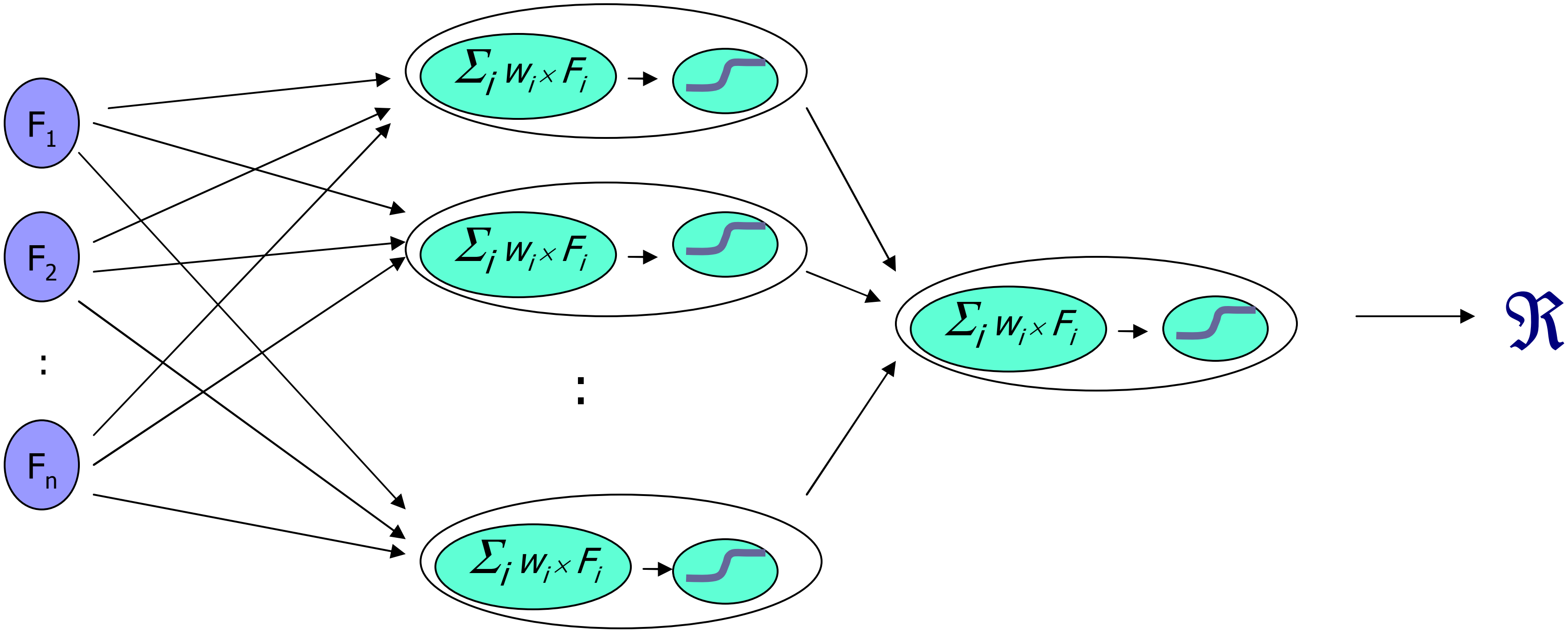


- Sigmoid Function: $\sigma(x) = \frac{1}{1+e^{-x}}$

- Useful properties:
  - $\sigma : \Re \rightarrow [0, 1]$
  - $\frac{\partial \sigma(x)}{\partial x} = \sigma(x)\ (1 - \sigma(x))$
  - If $x \approx \frac{1}{2}$, then $\sigma(x) \approx x$

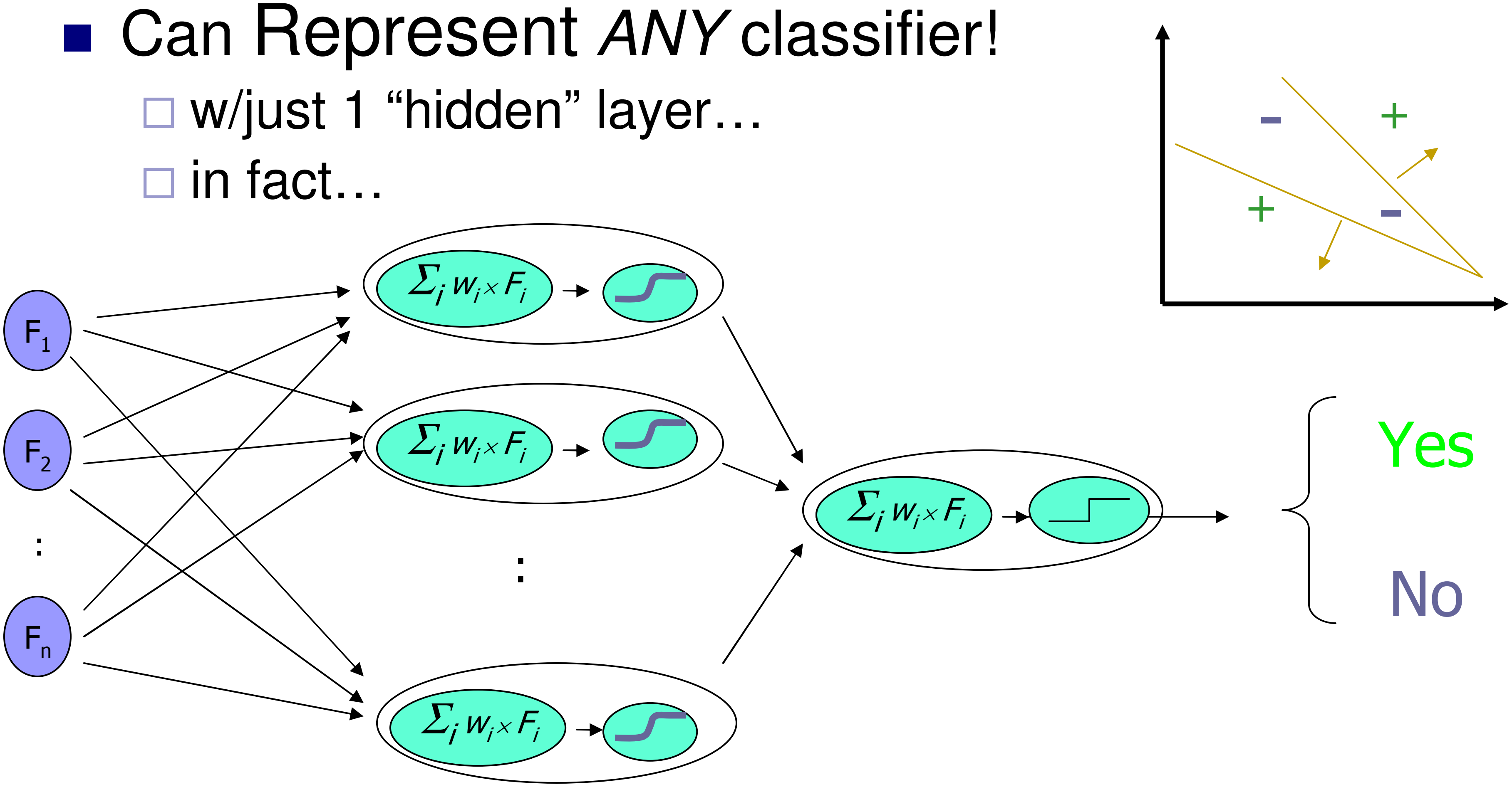
Feed Forward Neural Nets

- *SET* of **connected Sigmoid Functions**



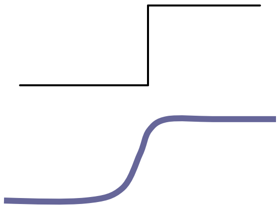
Artificial Neural Nets

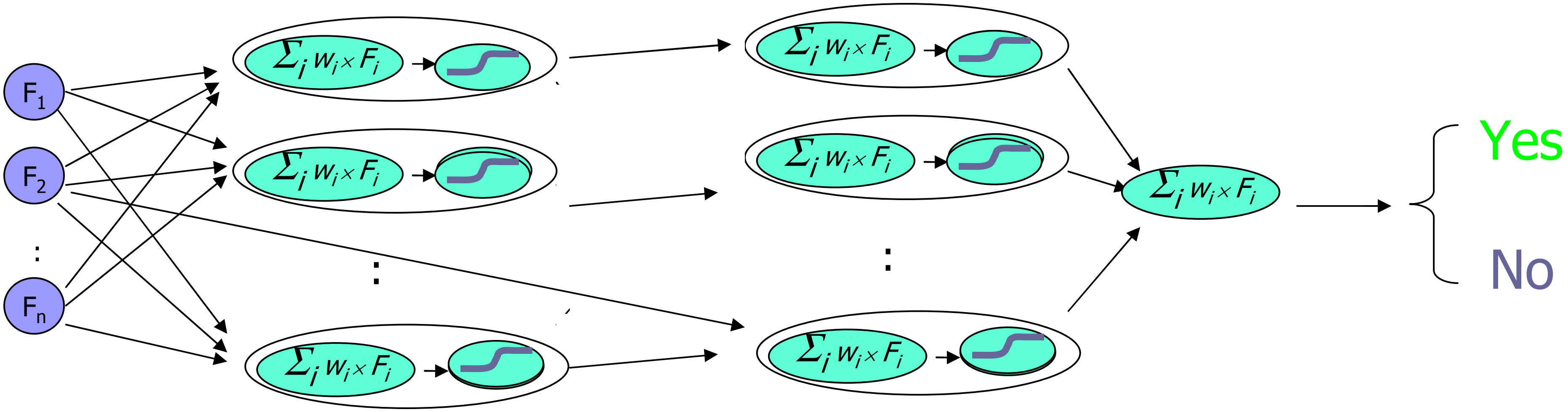
- Can Represent *ANY* classifier!
  - ☐ w/just 1 “hidden” layer…
  - ☐ in fact…



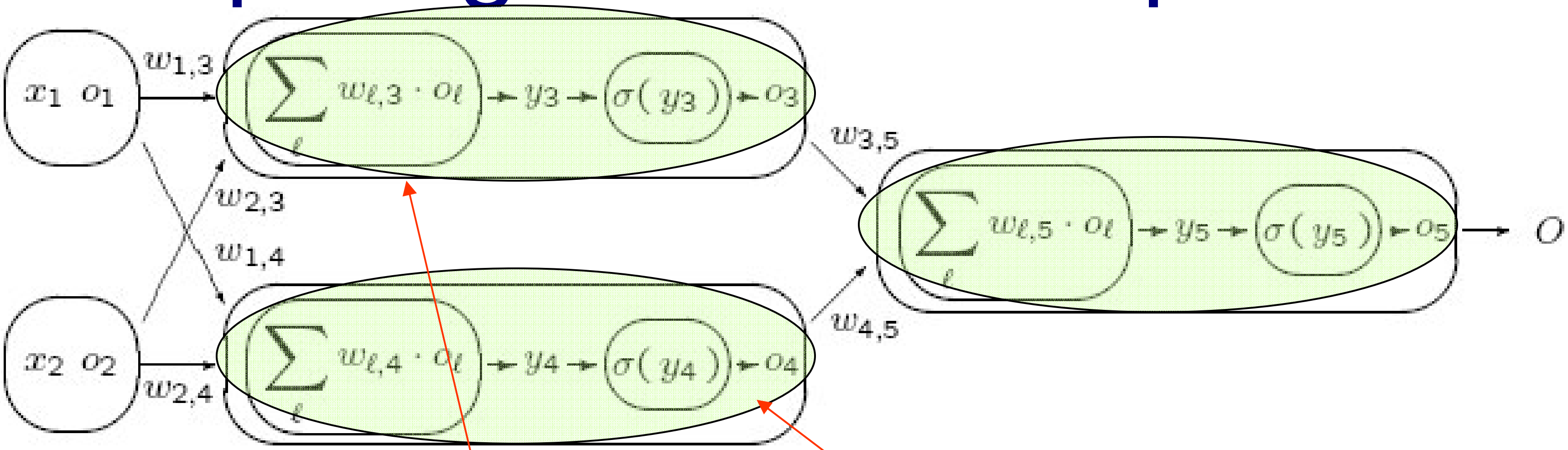
ANNs: Architecture

- Different # of layers
- Different structures
  - what's connected to what..
- Different "squashing function"





Computing Network Output



- Two (non-input) layers: 2 input units + 2 hidden units + 1 output unit
- “Activation” passed from input to output:

$$o = \sigma\left(\textstyle\sum_r w_{r,5} \cdot o_r \right) = \sigma\left(w_{3,5} \cdot o_3 + w_{4,5} \cdot o_4 \right)$$

$$= \sigma\left(w_{3,5} \cdot \boxed{\sigma\left(\textstyle\sum_s w_{s,3} \cdot o_s \right)} + w_{4,5} \cdot \boxed{\sigma\left(\textstyle\sum_t w_{t,4} \cdot o_t \right) \right)}$$

$$= \sigma\left(w_{3,5} \cdot \sigma\left(w_{1,3} \cdot o_1 + w_{2,3} \cdot o_2 \right) + w_{4,5} \cdot \sigma\left(w_{1,4} \cdot o_1 + w_{2,4} \cdot o_2 \right) \right)$$

Node #0 set to “1” is input to each node (using $w_{0,t}$)
Final unit (here “#5”) typically NOT $\sigma(\cdot)$

Representational Power

- **Any Boolean Formula**
  - Consider formula in DNF: $(x_1 \& \neg x_2) \vee (x_2 \& x_4) \vee (\neg x_3 \& x_5)$
  - Represent each AND by hidden unit; the OR by output unit.
  - ... but may need exponentially-many hidden units!

- **Bounded functions**
  - Can approximate any bounded continuous function to arbitrary accuracy with 1 hidden sigmoid layer

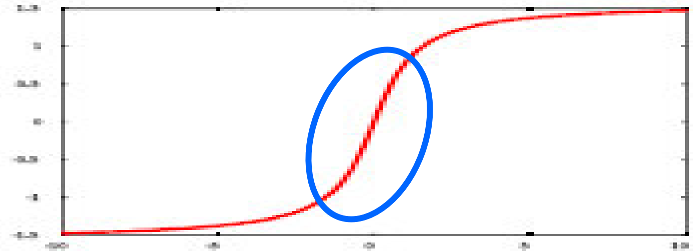
    \+ linear output unit
  - ... given enough hidden units.

  (Output unit "linear" ⇒ computes $\hat{y} = W_4 \cdot A$)

- **Arbitrary Functions**
  - Can approximate any function to arbitrary accuracy with 2 hidden sigmoid layers + linear output unit

Fixed versus Variable Size

- Network w/fixed # of hidden unit represents fixed hypothesis space
- But iterative training process
- More steps ⇒ can “reach" more functions
- So... view networks as having a *variable* hypothesis space



If all $w_{i,\ell} \approx 0$,
then $y = y_i \approx: \sum_\ell w_{i,\ell}\, o_\ell$

If $|y| < \epsilon$ then $\sigma(y) \approx y$

$\Rightarrow$ ≈ LINEAR!

$$\Rightarrow \quad \sum_i w_i \cdot \sigma(y_i) \approx \sum_i w_i \cdot y_i$$
$$= \sum_i w_i \sum_j w_j x_j \equiv \sum_j w'_j x_j$$

for new constant w'_j

Skip

Learning Neural Networks

Neural Networks Can Represent Complex Decision Boundaries

- ≈Stratified:

  More "gradient descent" steps ⇒ reach more functions
- **Deterministic**
- **Continuous Parameters**

Learning algorithms for neural networks

- **Local Search**:

  same algorithm as for sigmoid threshold units
- **Eager**
- **Batch** (typically)

Skip

MultiLayerNetwork Learning Task

- Want to minimize error on training ex's
  [not quite. . . why?]

⇒ function minimization problem.

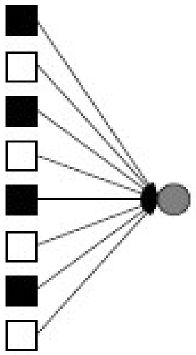
$$\boxed{Err(D, \vec{w}) \;=\; \frac{1}{2} \sum_{\langle \vec{x}, y\rangle \in D} (\, y - o_{\vec{w}}(\vec{x})\,)^2}$$

- *Err* on outputs, for given input,
  is function of weights *{ w_{ij} }*
- Minimize:
  - ☐ gradient descent in weight space:

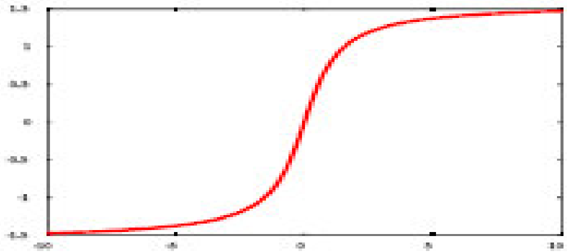
  ⇒ backpropagation algorithm (aka “chain rule”)

Backpropagation

- Perceptron learning relied on direct connection between input value x_j, weight w_j, output value
  ⇒ could localize contribution & determine change

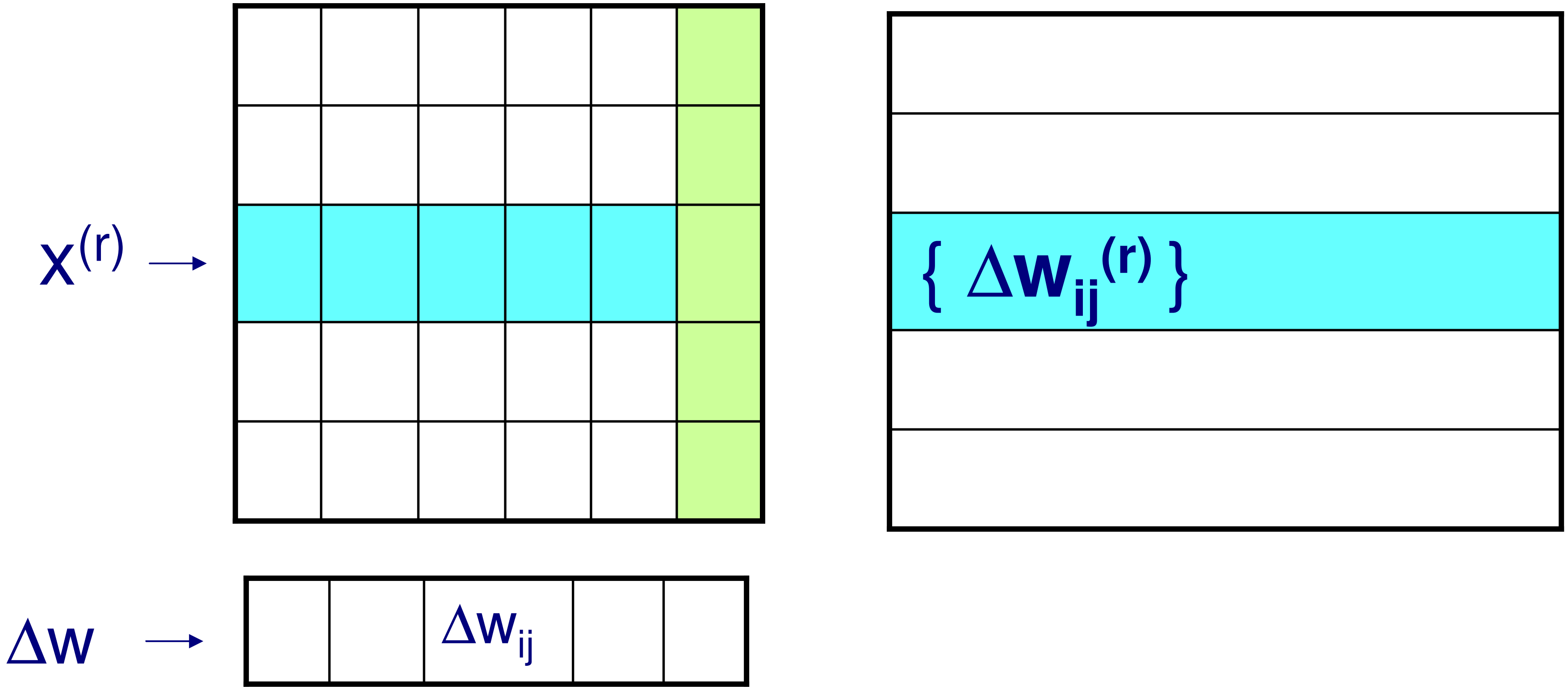


- Not true for multilayer network!
- Still, can estimate effect of each weight
  ... and make small changes accordingly
  Use derivative of error, wrt weight w_{ij} !
  Propagate backward (up net) using chain rule
- But no guarantees here... ∃ many local minima!
- Need to take DERIVATIVE
  ⇒ use “sigmoid” squashing function. . .

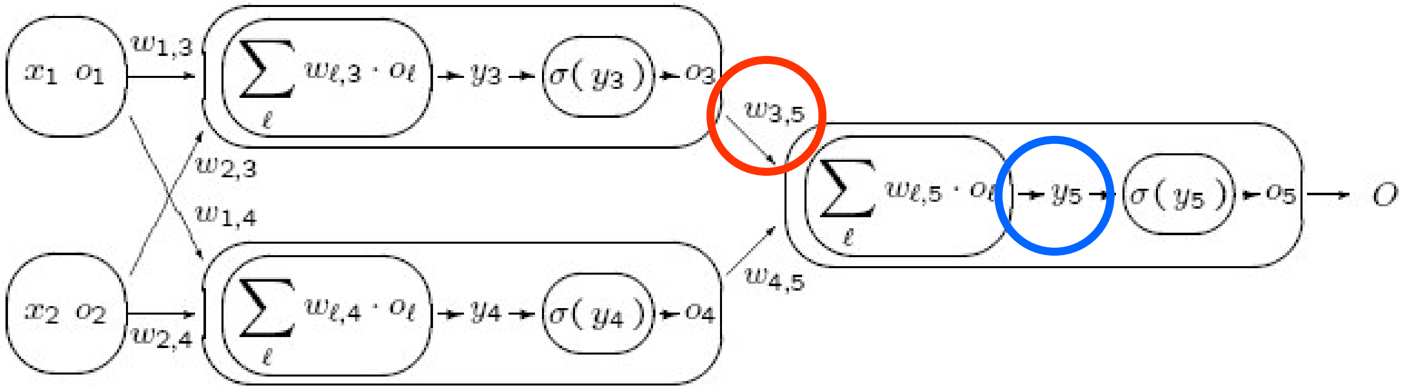


```
0. New $\mathbf{w}$
    $\Delta \mathbf{w} := 0$
1. For each instance r, compute
    $\Delta \mathbf{w}_{ij}^{(r)} = \ldots$

    $\Delta \mathbf{w}_{ij} \mathrel{+}= \Delta \mathbf{w}_{ij}^{(r)}$
2. Increment $w \mathrel{+}= \eta \, \Delta \mathbf{w}$
```



Error Gradient for Network



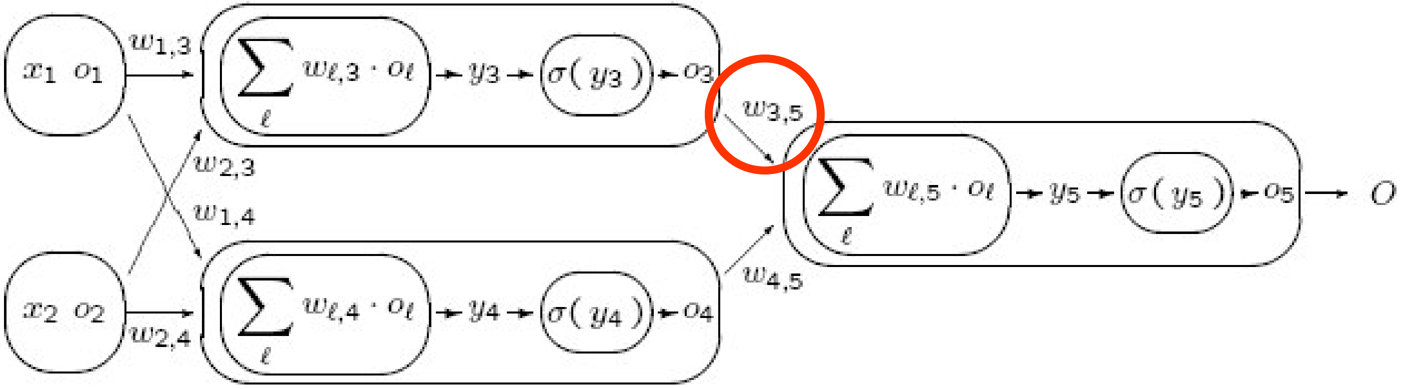
- $E \;=\; E(\,[\mathbf{x};\, t]\,) = \frac{1}{2}\,(O_{\mathbf{w}}(\mathbf{x}) - t\,)^2$

Let $\delta_i \triangleq \frac{\partial E}{\partial y_i}$

- $$\frac{\partial E(\langle \vec{x}, \vec{t} \rangle)}{\partial w_{3,5}} = \frac{\partial E}{\partial y_5}\frac{\partial y_5}{\partial w_{3,5}} = \delta_5 \frac{\partial y_5}{\partial w_{3,5}}$$
- $$\frac{\partial y_5}{\partial w_{3,5}} = \frac{\partial(\sum_\ell w_{\ell,5} \cdot o_\ell)}{\partial w_{3,5}} = \frac{\partial(w_{3,5} \cdot o_3 \; + \; w_{4,5} \cdot o_4)}{\partial w_{3,5}} = o_3$$

$$\Rightarrow \quad \boxed{\frac{\partial E(\langle \vec{x}, \vec{t} \rangle)}{\partial w_{3,5}} = \delta_5 \; o_3}$$

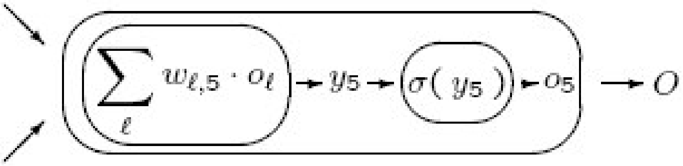
Factoring Derivative



- Here: $$\boxed{\frac{\partial E(\langle \vec{x}, \vec{t} \rangle)}{\partial w_{3,5}} = \delta_5 \; o_3}$$
- In General $\frac{\partial E(\langle \vec{x}, \vec{t} \rangle)}{\partial w_{i,j}} = \frac{\partial E}{\partial y_j} \frac{\partial y_j}{\partial w_{i,j}} = \delta_j \; o_i$
$$\boxed{\frac{\partial E(\langle \vec{x}, \vec{t} \rangle)}{\partial w_{i,j}} = \delta_j \; o_i}$$
- Compute each o_i during $\boxed{\text{FORWARD sweep}}$

  Compute each δ_j during $\boxed{\text{BACKWARD sweep}}$!

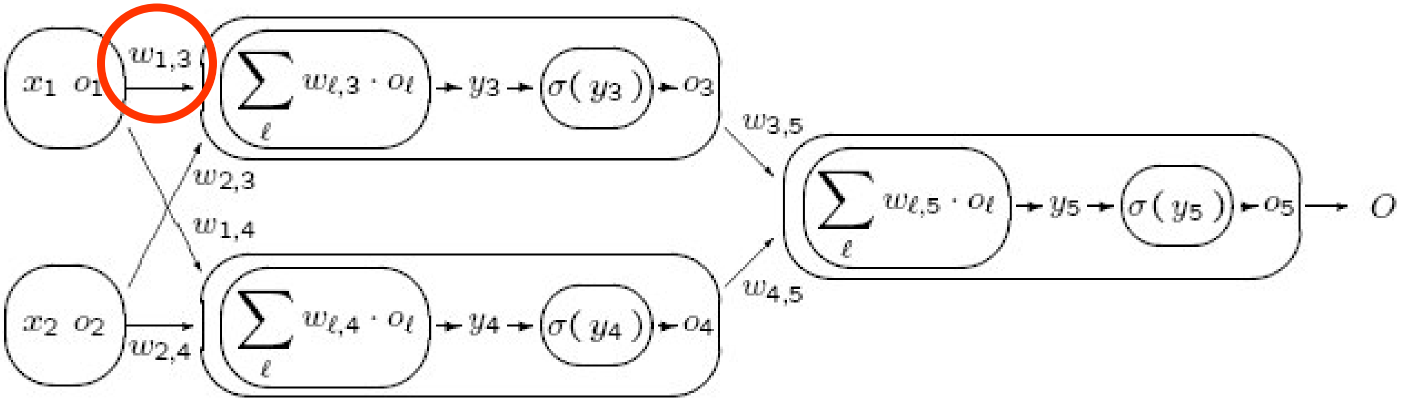
Computing “Terminal” δ_is



- $\delta_5 = \frac{\partial E}{\partial y_5} = \frac{\partial E}{\partial o_5} \frac{\partial o_5}{\partial y_5}$
- $\frac{\partial E(\langle \vec{x}, t \rangle)}{\partial \, o_5} = \frac{\partial}{\partial o_5} \left[\frac{1}{2} (o_5 - t)^2 \right] = (o_5 - t) \cdot \frac{\partial}{\partial o_5} (o_5 - t) = (o_5 - t)$
- $\frac{\partial \, o_5}{\partial y_5} = \frac{\partial \sigma(y_5)}{\partial y_5} = \sigma(y_5) \, (1 - \sigma(y_5)) = o_5 \, (1 - o_5)$

$$\Rightarrow \boxed{\delta_5 = (o_5 - t) \;\; o_5 \, (1 - o_5)}$$

Computing Non-Terminal δ_is

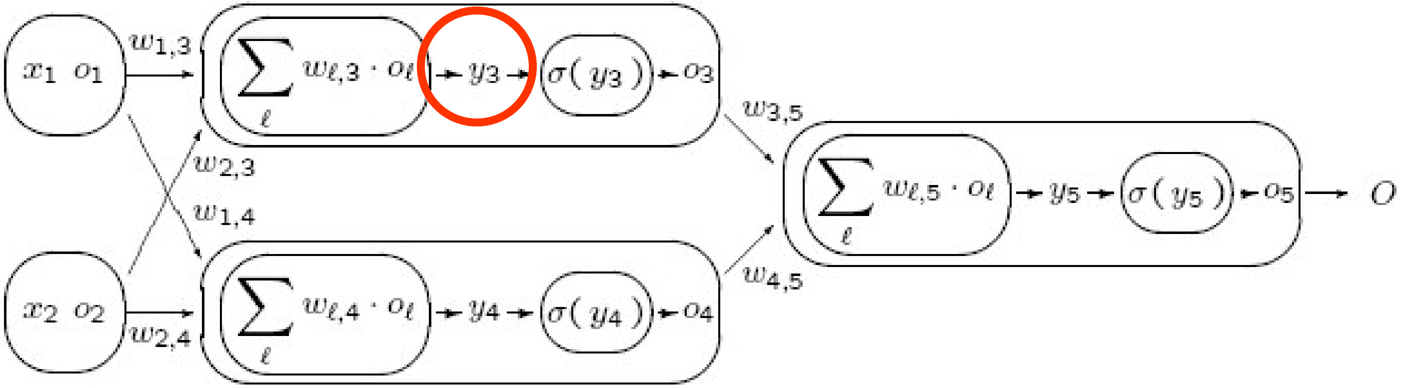


As $\frac{\partial E(\langle \vec{x},t \rangle)}{\partial w_{1,3}}$ depends only on o_3, and hence y_3

$$\Rightarrow \quad \boxed{\frac{\partial E(\langle \vec{x}, t \rangle)}{\partial w_{1,3}} = \frac{\partial E}{\partial y_3} \frac{\partial y_3}{\partial w_{1,3}} = \delta_3 \; o_1}$$

- $\frac{\partial y_3}{\partial w_{1,3}} = \frac{\partial (\sum_\ell w_{\ell,3} o_\ell)}{\partial w_{1,3}} = o_1$
- $\delta_3 = \frac{\partial E}{\partial y_3} = \frac{\partial E}{\partial o_3} \frac{\partial o_3}{\partial y_3}$

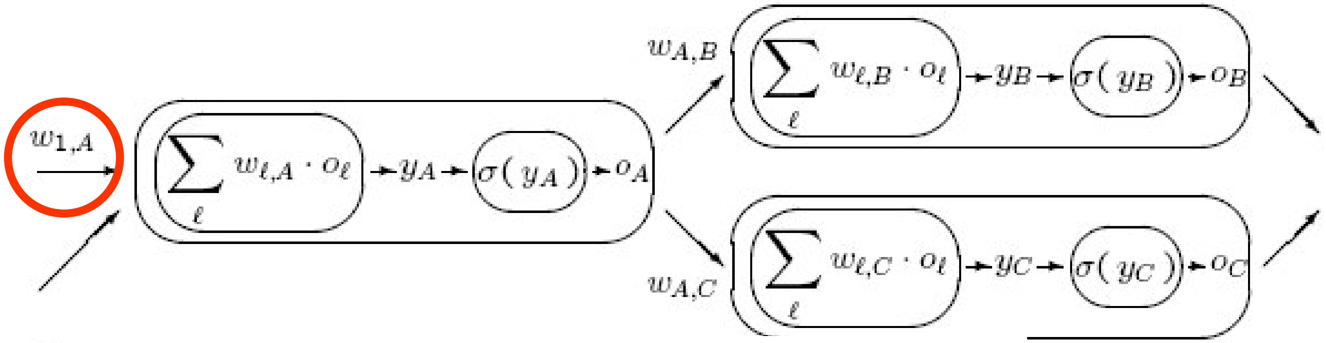
Computing δ_3



- $\delta_3 = \frac{\partial E}{\partial y_3} = \frac{\partial E}{\partial o_3} \frac{\partial o_3}{\partial y_3}$
- $\frac{\partial E}{\partial o_3} = \frac{\partial E}{\partial y_5} \frac{\partial y_5}{\partial o_3} = \delta_5 \frac{\partial(\sum_\ell w_{\ell,5} \cdot o_\ell)}{\partial o_3} = \delta_5 \cdot w_{3,5}$
- $\frac{\partial o_3}{\partial y_3} = \frac{\partial \sigma(y_3)}{\partial y_3} = \sigma(y_3)\,(1 - \sigma(y_3)) = o_3\,(1 - o_3)$

$$\Rightarrow \quad \boxed{\delta_3 = \left[\delta_5\, w_{3,5}\right] o_3\,(1 - o_3)}$$

What if Many Children?



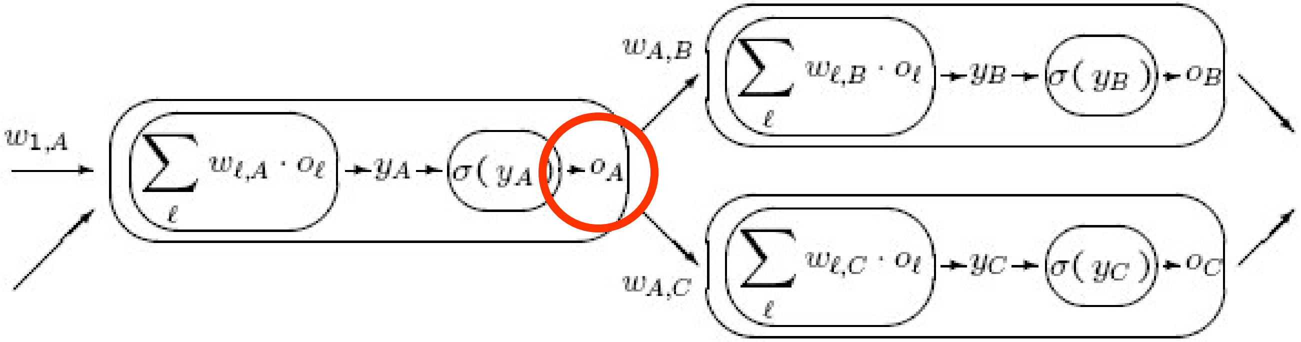
- As before. . .

$$\frac{\partial E}{\partial w_{1,A}} = \frac{\partial E}{\partial y_A}\frac{\partial y_A}{\partial w_{1,A}} = \delta_A\, o_1$$

$$\delta_A = \frac{\partial E}{\partial y_A} = \frac{\partial E}{\partial o_A}\frac{\partial o_A}{\partial y_A} = \frac{\partial E}{\partial o_A}[o_A(1-o_A)]$$

- Notice $\frac{\partial E}{\partial o_A}$ depends only on BOTH
  - $\star$ B (via y_B)
  - $\star$ C (via y_C)

Multiple Children (con't)



- $$\frac{\partial E}{\partial o_A} = \frac{\partial E}{\partial y_B}\frac{\partial y_B}{\partial o_A} + \frac{\partial E}{\partial y_C}\frac{\partial y_C}{\partial o_A} = \sum_{k\in child(A)} \frac{\partial E}{\partial y_k}\frac{\partial y_k}{\partial o_A}$$

$$= \sum_{k\in child(A)} \delta_k \frac{\partial\left(\sum_\ell w_{\ell,k}\cdot o_\ell\right)}{\partial o_A} = \sum_{k\in child(A)} \delta_k w_{A,k}$$

Here: $\delta_A = o_A(1-o_A)\left[\delta_B w_{A,B} + \delta_C w_{A,C}\right]$

- In general: $$\boxed{\delta_\ell = o_\ell(1-o_\ell) \sum_{k\in child(\ell)} \delta_k w_{\ell,k}}$$

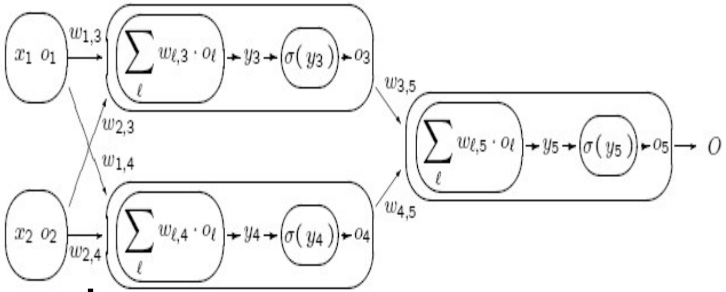
Basic Computations

- 1. Sweep FORWARD, from input to output
  - For each node *n*, compute “output” o_n
- 2. Sweep BACKWARD, from output to input
  - For each node *n*, compute

$$\begin{aligned}
\delta_n &= \frac{\partial E}{\partial y_n} \\
&= o_n\,(1-o_n) \begin{cases} (t-o) & \text{if terminal} \\ \displaystyle\sum_{k \in child(n)} \delta_k\, w_{n,k} & \text{otherwise} \end{cases} \\
\frac{\partial E}{\partial w_{\ell,n}} &= \delta_n\, o_\ell
\end{aligned}$$

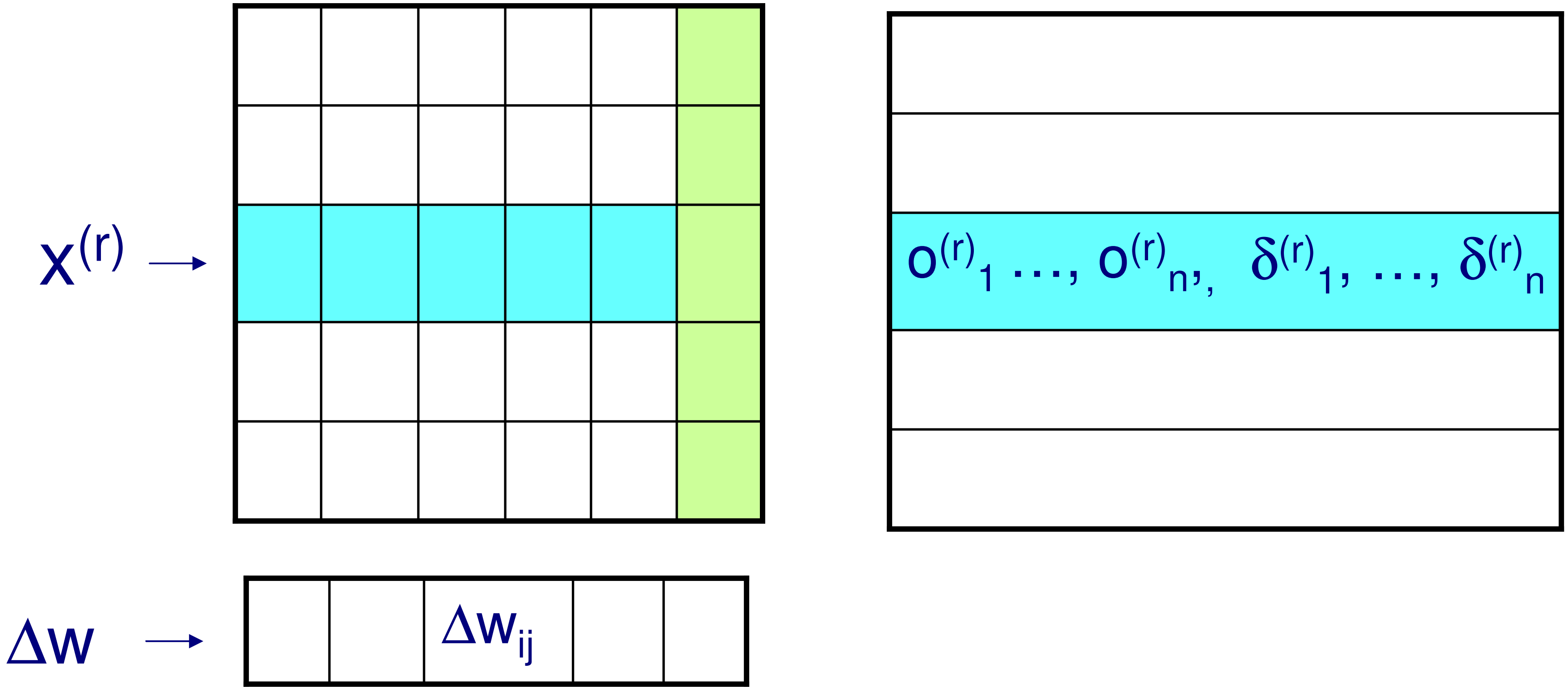
- Notice everything is trivial to compute!

Backpropagation Alg

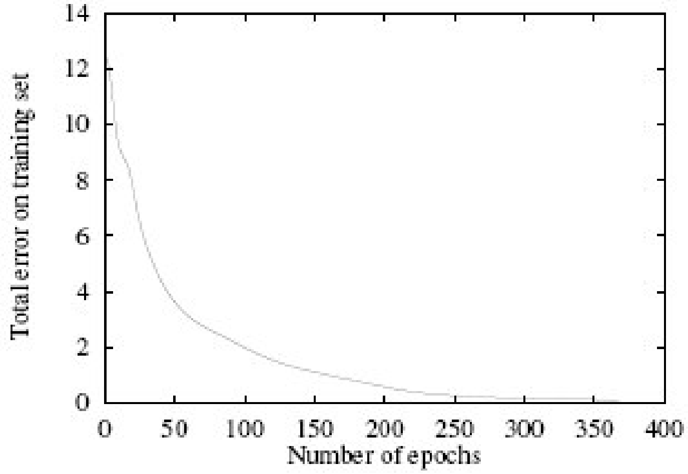


```
Initialize all weights to small random numbers
Until satisfied, do
  ■ For each training example [x, t] , do
    1. Sweep forward
       Compute network outputs $o_k$ for x for each hidden/output node
    2. Sweep backward
       For each output unit k
         $\delta_k \leftarrow o_k (1 - o_k) (t_k - o_k)$
       For each hidden unit h
         $\delta_h \leftarrow o_h (1 - o_h) \sum_{k \in \text{child}(h)} w_{h,k} \delta_k$
    3. Update each network weight
         $w_{i,j} \leftarrow w_{i,j} + \eta \delta_j o_i$
```

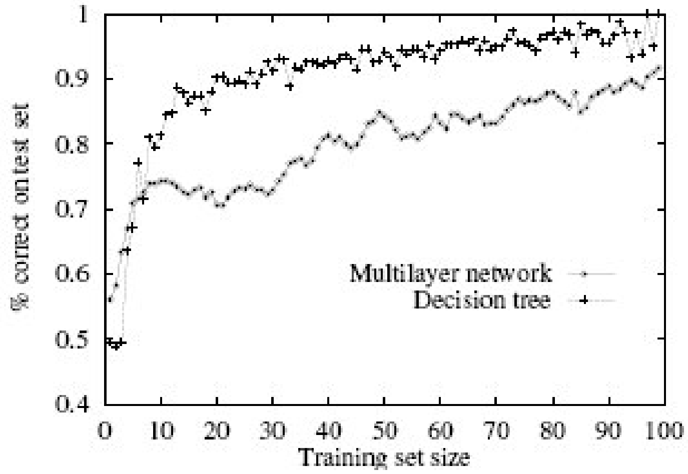
0. New **w**
   $\Delta\mathbf{w} := 0$
1. For each instance r, compute
   a. Forward: $o^{(r)}_i := \sigma\left(\sum_j w_{ji}\, o^{(r)}_j \right)$
   b. Backward: $\delta_i^{(r)} = \frac{\partial E^{(r)}}{\partial y^{(r)}}$
   c. $\Delta \mathbf{w}_{ij}$ += $\delta^{(r)}_j\, o^{(r)}_i$
2. Increment w += $\eta\, \Delta\mathbf{w}$



Empirical Results (MultiLayer Net)



“Restaurant Domain”



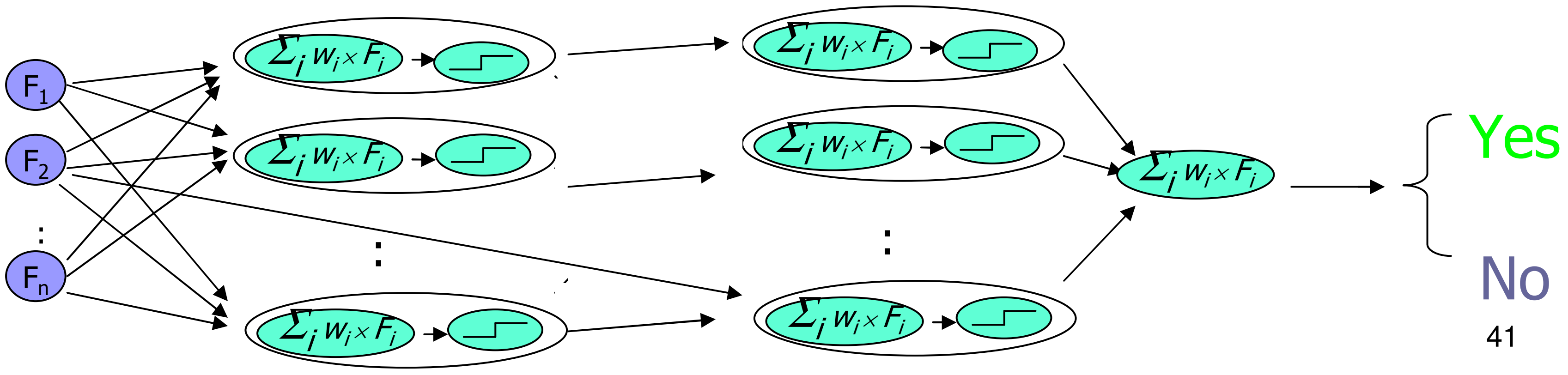
More on Backpropagation

- Gradient descent over entire network weight vector $\{ w_{ij} \}$
- Can be either: “Incremental Mode” Gradient Descent or “Batch Mode”:

$$\frac{\partial E}{\partial w_i} = \sum_{d \in D} \frac{\partial E^{(d)}}{\partial w_i}$$

- Easily generalized to arbitrary directed graphs
  - ☐ If have > 1 OUTPUTs: Just add them up!
  - ☐ Can have arbitrary connections

    Not just “everything on level 3 to everything on level 4”



Issues

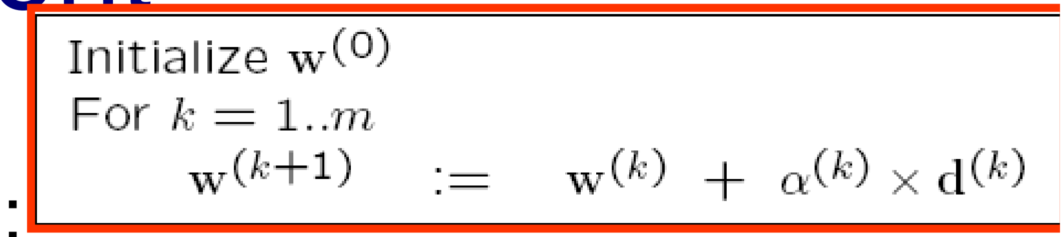
Backprop will (at best)…

- … slowly …
  - Faster? Line search, Conjugate gradient, …
- … converge to LOCAL Opt …
  - Multiple restart, simulated annealing, …
- … wrt Training Data
  - Early stopping, regularization

Outline

- Introduction
  - Historical Motivation, non-LTU, Objective
  - Types of Structures
- Multi-layer Feed-Forward Networks
  - Sigmoid Unit
  - Backpropagation
- Tricks for Effectiveness
  - Efficiency: Line Search, Conjugate Gradient
  - Generalization: Alternative Error Functions
- Hidden layer representations
  - Example: Face Recognition
- Recurrent Networks

Gradient Descent



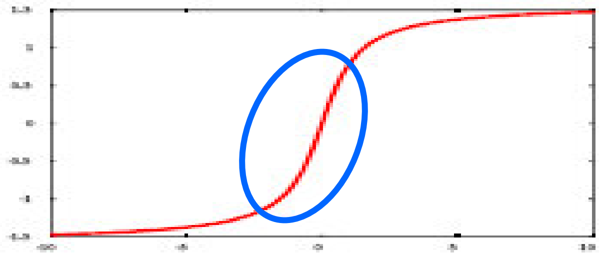
```
Initialize $\mathbf{w}^{(0)}$
For $k = 1..m$
    $\mathbf{w}^{(k+1)} := \mathbf{w}^{(k)} + \alpha^{(k)} \times \mathbf{d}^{(k)}$
```

- General description:
  Want w* that minimizes function J(w)
- So far. . .
  - $w^{(0)}$ is random
  - $\alpha^{(k)} = 0.05$
  - $d^{(k)} = \nabla J = \left\langle \frac{\partial J(\mathbf{w}^{(k)})}{\partial w_i^{(k)}} \right\rangle_i$ is derivative
  - *m* = until bored...
- Alternatively...
  1. Use *small* random values for $w^{(0)}$
  2. Use *line search* for distance $\alpha^{(k)}$
  3. Use *conjugate gradient* for direction $d^{(k)}$
  4. Use “cross tuning” for stopping criteria m … overfitting

1. Proper Initialization (variables)

- Put all of the variables on same scale
- Standardize all feature values
    - Mean = 0, Standard Deviation = 1
    - (ie, subtract mean, divide by std.dev.)

1. Proper Initialization (w)



- Start in “linear regions”
  - Keep all weights near 0,

    ⇒ sigmoid units in linear regions.

    ⇒ whole net one linear threshold unit (very simple function)
- Break symmetry
  - Ensure each unit has different input weights (so hidden units move in different directions)
  - Set weight to random number in range

$$[-1,\ +1] \times \frac{1}{\sqrt{\text{fan-in}}}$$

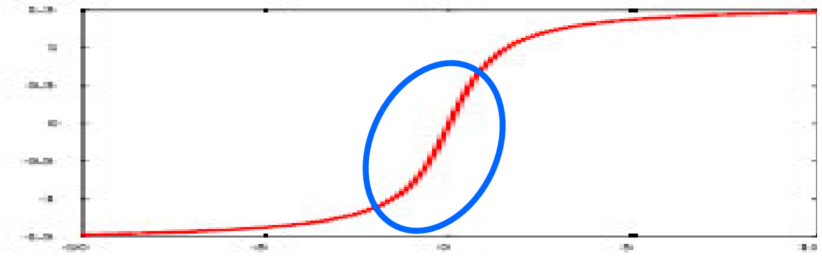
Why BackProp tends to Work?

- Only guaranteed to converge
    - EVENTUALLY
    - to a LOCAL opt
- Why does it work so well in practice?

As start w/ $w_{ij} \approx 0$,

network ≈ linear in weights…

so moves quickly



... until in “correct region”

Efficiency

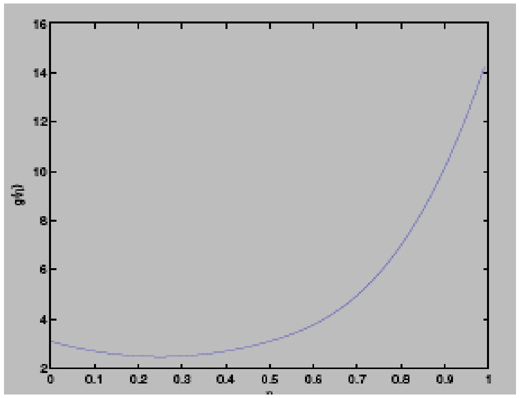
- **Number of Iterations**: Very important!
  - If too small: high error
  - If too large: overfitting ⇒ high gen'l error
- **Learning**: Intractable in general
  - Training can take thousands of iterations .. slow!
  - Learning net w/ single hidden unit is NP-hard
  - In practice: backprop is very useful.
- Use: Using network (after training) is very fast

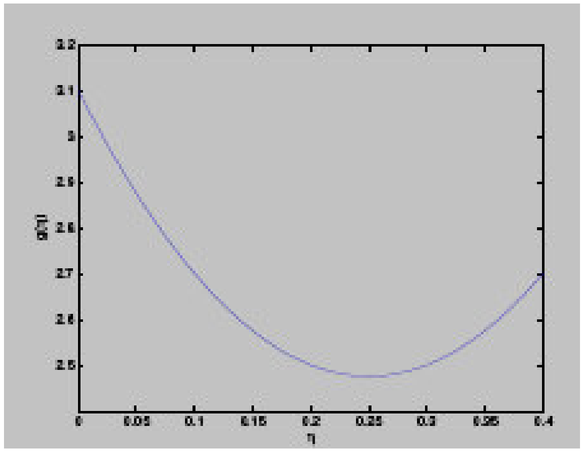
2. Line Search

- **Task**: Seek **w** that minimize $J(\mathbf{w})$
- Approach: Given direction $d \in \Re^n$
  - ☐ New value $\mathbf{w}^{(r+1)} := \mathbf{w}^{(r)} + \eta\, \mathbf{d}$
  - ☐ But what value of η?
- Good news: $\eta \in \Re \Rightarrow$ 1 dim search!
- Let $e(\eta) = J(\, \mathbf{w} + \eta \cdot \mathbf{d}\,)$

  Want $\eta^* = \text{argmin } e(\eta)$
- **Line Search**:

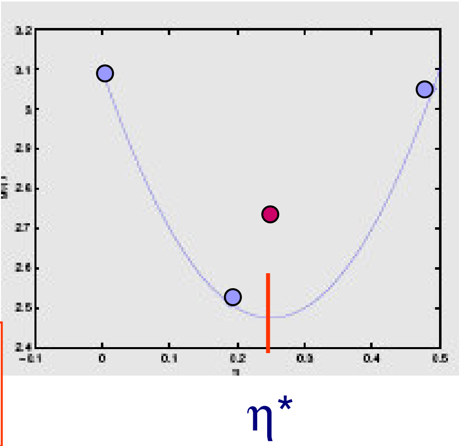
  Near 0, $e(\eta) \approx$ quadratic





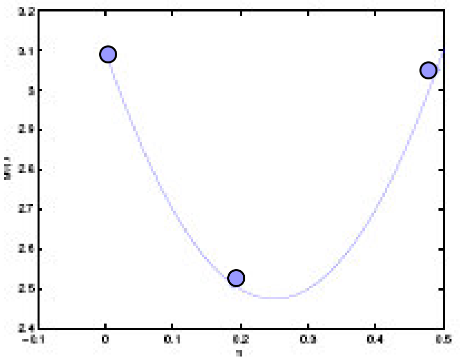
Line Search, con't

- Set $\eta_A = 0$, and guess 2 other values:
  Eg, $\eta_B = 0.2 \qquad \eta_C = 0.5$
  s.t. $e(\eta_A), e(\eta_C) > e(\eta_B)$



- Fit 2-D poly $\boxed{h(\eta) = r\,\eta^2 + s\,\eta + t}$
  to $[\eta_A, e(\eta_A)]$, $[\eta_B, e(\eta_B)]$, $[\eta_C, e(\eta_C)]$
- Take min of this poly… the new η^*
- Compute $e(\eta^*)$

Line Search, III



- Let $\eta^* = \text{argmin}_\eta\ h(\eta)$

  Iteration $\langle \eta'_A, \eta'_B, \eta'_C \rangle$:=

  $\langle \eta^*, \eta_B, \eta_C \rangle$ if $\eta^* < \eta_B$ & $e(\eta^*) > e(\eta_B)$
  $\langle \eta_A, \eta^*, \eta_C \rangle$ if $\eta^* < \eta_B$ & $e(\eta^*) < e(\eta_B)$
  $\langle \eta_B, \eta^*, \eta_C \rangle$ if $\eta^* > \eta_B$ & $e(\eta^*) < e(\eta_B)$
  $\langle \eta_A, \eta_B, \eta^* \rangle$ if $\eta^* > \eta_B$ & $e(\eta^*) > e(\eta_B)$

- ... for ONE ITERATION of general search

  Search can involve m iterations,

  Each iteration may involve 10's of eval's to get η^*

- Issues:
  - ☐ How to find first 3 values?
  - ☐ Many other tricks... (Brent's Method)
  - ☐ Given assumptions, ANALYTIC form

3. Conjugate Gradient

- At step *r*, searching along gradient $\mathbf{d}^{(r)}$

  ... using $q(\eta) = J(\mathbf{w}^{(r)} + \eta \cdot \mathbf{d}^{(r)})$

At minimum η^*: $\boxed{\frac{\partial}{\partial \eta} J(w^{(r)} + \eta\, d^{(r)}) = 0}$

Let $\mathbf{w}^{(r+1)} = \mathbf{w}^{(r)} + \eta^* \cdot \mathbf{d}^{(r)}$

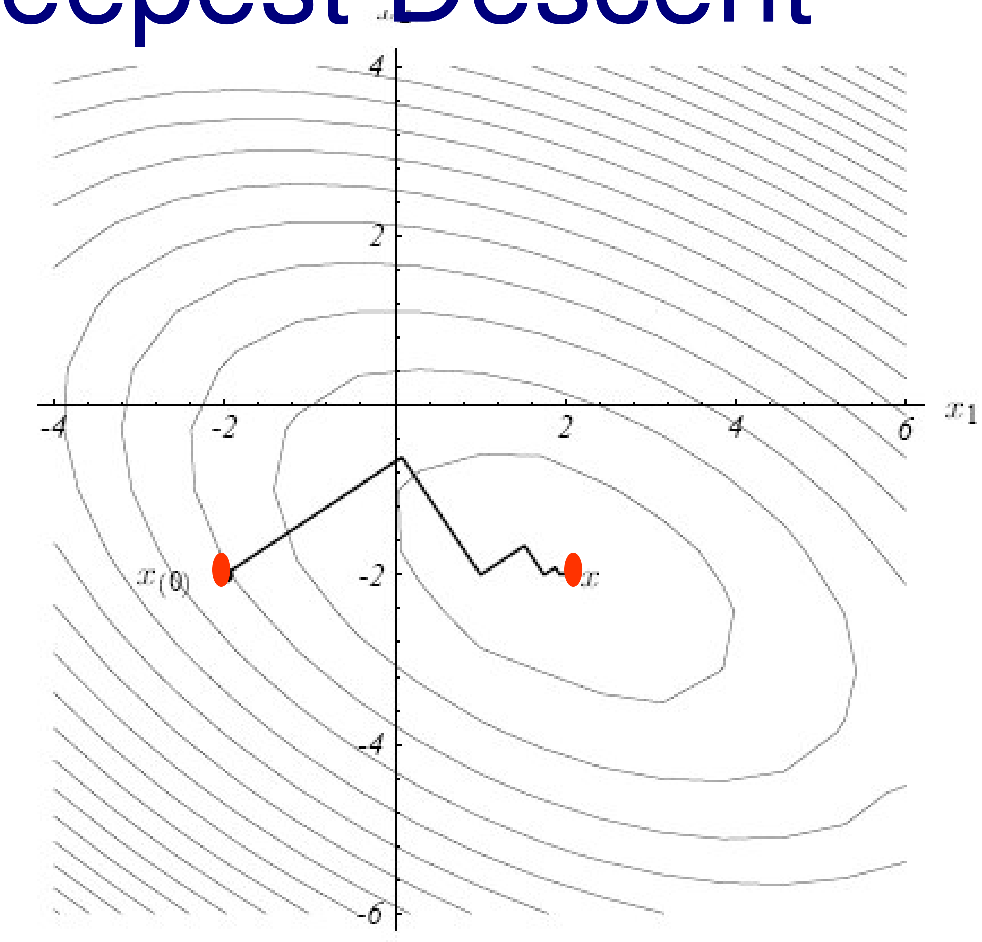
$\Rightarrow \quad \nabla J(\mathbf{w}^{(r+1)})^{\top} \; \mathbf{d}^{(r)} = 0$

- Gradient $\nabla J(\mathbf{w}^{(r+1)})$ at r +1st step is ORTHOGONAL to previous search direction $\mathbf{d}^{(r)}$!

- Is this the best direction??

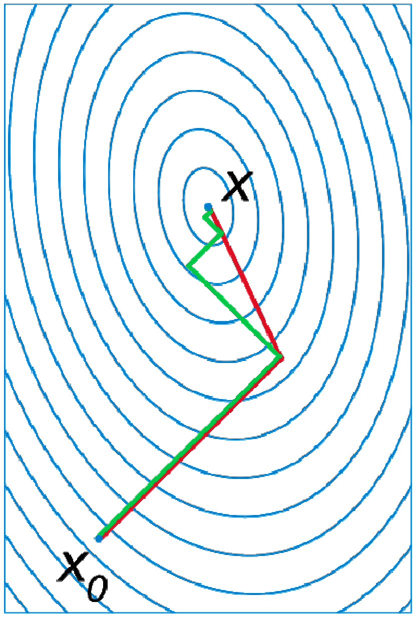
Problem with Steepest Descent

- Steepest Descent…
  from $[-2,-2]^T$ to $[2,-2]^T$

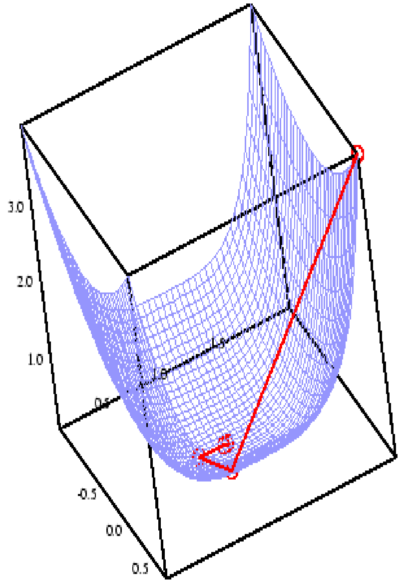


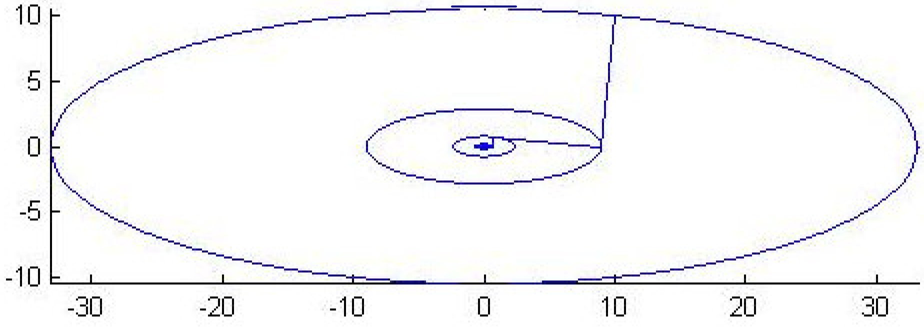
- Path "zigzag"s as each gradient is orthogonal to the previous gradient

Does Gradient always work??



- Each green line is gradient…
- Problematic when going down narrow canyon
- Red is better…





Better…

- Problem: Gradients { $\mathbf{g}_i$ } are NOT orthogonal to each other
  - so can "repeat" same directions
- Suppose directions { $\mathbf{d}_i$ } were Conjugate
  - Spanning
  - "Orthogonal" (wrt matrix)
- Then after n moves (dim of space), must be at optimum!!

Make Descent Directions Orthogonal

- At step *r*, searching along gradient $\mathbf{d}_r$

  … using $g(\eta) = J(\mathbf{w}_r + \eta \cdot \mathbf{d}_r)$

At minimum: $\boxed{\frac{\partial}{\partial \eta} J(w_r + \eta^* d_r) = 0}$

Let $\mathbf{w}_{r+1} = \mathbf{w}_r + \eta^* \cdot \mathbf{d}_r$

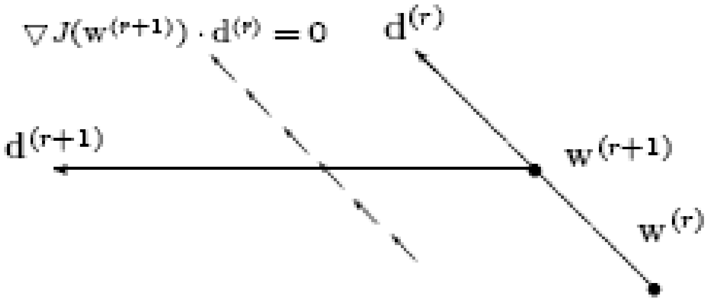
$\Rightarrow \nabla J(\mathbf{w}_{r+1})^T \mathbf{d}_r = 0$

- Gradient $\nabla J(\mathbf{w}_{r+1})$ at r +1st step is ORTHOGONAL to previous search direction $\mathbf{d}_r$!

Direction $\mathbf{d}_{r+1}$ is conjugate to direction d_r

if component of gradient parallel to d_r

remains 0 as move along d_{r+1}



Conjugate Gradient, IIa

$g = \nabla J = \left\langle \frac{\partial J}{\partial w_1}, \ldots, \frac{\partial J}{\partial w_n} \right\rangle$ Later. . . $\mathbf{g}_r = \nabla J(\mathbf{w}^{(r)})$ on r^{th} iteration

- Let **d** be DIRECTION of change.

  Could have $\mathbf{d} = \mathbf{g}$ but . . .

- At time *r*, require $g(\mathbf{w}_{r+1})^T \mathbf{d}_r = 0$

  Want this to be true for next direction as well:

  $g(\mathbf{w}_{r+2})^T \mathbf{d}_r = 0$

... want d_{r+1} s.t.

$$\mathbf{w}_{r+2} := \mathbf{w}_{r+1} + \lambda\, \mathbf{d}_{r+1}$$

$$g(\mathbf{w}_{r+1} + \lambda\, \mathbf{d}_{r+1})^T \mathbf{d}_r = 0$$

Conjugate Gradient, IIb

- First order Taylor expansion:

$$0 = g(\,\mathbf{w}_{r+1} + \lambda\, \mathbf{d}_{r+1}\,)^T$$
$$= g(\mathbf{w}_{r+1})^T + \lambda \mathbf{d}_{r+1}{}^T\, g'(\,\mathbf{w}_{r+1} + \gamma\, \mathbf{d}_{r+1}\,)$$

for some $\gamma \in (0, \lambda)$

- Post-Multiply by $\mathbf{d}_r$ & use $g(\mathbf{w}_{r+1})^T\, \mathbf{d}_r = 0$ to get

$$\lambda \mathbf{d}_{r+1}{}^T\, g'(\mathbf{w}_{r+1} + \gamma\, \mathbf{d}_{r+1})\, \mathbf{d}_r = 0$$

- Let $\mathcal{H}(\mathbf{w}_r) = g'(\mathbf{w}_r) = \nabla(\nabla J(\mathbf{w}_r)\ )$

Hessian Matrix (Second Derivatives)

- Consider $J(x, y) = x^2 + 3xy - 5x$

- $g(x,y) = \nabla J = \left\langle \frac{\partial J(x,y)}{\partial x}, \frac{\partial J(x,y)}{\partial y} \right\rangle = \langle 2x + 3y - 5,\ 3x \rangle$

- $\mathcal{H} = \nabla\nabla J = \begin{bmatrix} \frac{\partial}{\partial x}\frac{\partial J(x,y)}{\partial x} & \frac{\partial}{\partial y}\frac{\partial J(x,y)}{\partial x} \\ \frac{\partial}{\partial x}\frac{\partial J(x,y)}{\partial y} & \frac{\partial}{\partial y}\frac{\partial J(x,y)}{\partial y} \end{bmatrix}$

$$= \begin{bmatrix} \frac{\partial}{\partial x}(2x + 3y - 5) & \frac{\partial}{\partial y}(2x + 3y - 5) \\ \frac{\partial}{\partial x}(3x) & \frac{\partial}{\partial y}(3x) \end{bmatrix} = \begin{bmatrix} 2 & 3 \\ 3 & 0 \end{bmatrix}$$

- As $J(x, y)$ is quadratic, $\mathcal{H}$ is constant

  If $J(x, y) = x^3y^2 + \ldots$, then is function of args

$$\boxed{\lambda \mathbf{d}_{r+1}{}^{T}\ g'(\mathbf{w}_{r+1} + \gamma\, \mathbf{d}_{r+1})\ \mathbf{d}_r = 0}$$

- Using $\mathcal{H}(\mathbf{w}_r) = g'(\mathbf{w}_r) = \nabla(\ \nabla J(\mathbf{w}_r)\ )$

  $0 = \mathbf{d}_{r+1}{}^{T}\ g'(\mathbf{w}_{r+1} + \gamma\, \mathbf{d}_{r+1})\ \mathbf{d}_r$

  $= \mathbf{d}_{r+1}{}^{T}\, \mathcal{H}(\mathbf{w}_{r+1} + \gamma\, \mathbf{d}_{r+1})\ \mathbf{d}_r$

  $\approx \mathbf{d}_{r+1}{}^{T} \qquad \mathcal{H} \qquad \mathbf{d}_r$
- Challenge: How to find such $\mathbf{d}_r$ vectors?
- Assuming $J(\mathbf{w}) = J_0 + b^T\mathbf{w} + \frac{1}{2}\, \mathbf{w}^T \mathcal{H}\, \mathbf{w}$

  then $\mathbf{g}(\mathbf{w}) = \nabla J(\mathbf{w}) = b + \mathcal{H}\, \mathbf{w}$
- J is min at $\mathbf{w}^*$ s.t. $g(\mathbf{w}^*) = b + \mathcal{H}\, \mathbf{w}^* = 0$

Conjugate Gradient, IV

- Spse ∃ k vectors "mutually conjugate wrt $\mathcal{H}$"

  $$\mathbf{d}_j^T \mathcal{H} \mathbf{d}_i = 0 \quad j \neq i$$

  Then $\{ \mathbf{d}_i \}$ linearly independent (if $\mathcal{H}$ pos def)
- Starting from $\mathbf{w}_1$; want minimum $\mathbf{w}^*$

  As $\{ \mathbf{d}_i \}$ spanning, $\mathbf{w}^* - \mathbf{w}_1 = \sum_{i=1}^{k} \alpha_i \mathbf{d}_i$
- Let $\mathbf{w}_j = \mathbf{w}_1 + \sum_{i=1}^{j-1} \alpha_i \mathbf{d}_i$

  $\Rightarrow \quad \mathbf{w}_{j+1} = \mathbf{w}_j + \alpha_j \mathbf{d}_j$
- Series of steps, each parallel some conjugate direction, of magnitude $\alpha_j \in \Re$
- Earlier: computed optimal α_j by line search.

  But given above assumptions...

To find α_j

- To find value for α_j :
  - ☐ multiply $\mathbf{w}^* - \mathbf{w}_1 = \sum_{i=1}^{k} \alpha_i \mathbf{d}_i$
  - ☐ by $\mathbf{d}_j^T \mathcal{H}$:

$$\mathbf{d}_j^T(-\mathbf{b} - \mathcal{H}\,\mathbf{w}_1) = \sum_{i=1}^{k} \alpha_i\, \mathbf{d}_j^T \mathcal{H}\, \mathbf{d}_i = \alpha_j\, \mathbf{d}_j^T \mathcal{H}\, \mathbf{d}_j$$

As $\mathbf{w}^*$ is optimum, $0 = g(\mathbf{w}^*) = \mathcal{H}(\mathbf{w}^*) + b$

As $\mathbf{d}_j^T \mathcal{H}\, \mathbf{d}_i = 0$ unless i = j

$$\boxed{\alpha_j = -\frac{\mathbf{d}_j^T(\mathbf{b} + \mathbf{H}\mathbf{w}_{\boxed{1}})}{\mathbf{d}_j^T \mathbf{H}\mathbf{d}_j} = -\frac{\mathbf{d}_j^T(\mathbf{b} + \mathbf{H}\mathbf{w}_{\boxed{j}})}{\mathbf{d}_j^T \mathbf{H}\mathbf{d}_j} = -\frac{\mathbf{d}_j^T \mathbf{g}_j}{\mathbf{d}_j^T \mathbf{H}\mathbf{d}_j}}$$

$$\left(\begin{aligned} \mathbf{d}_j^T \mathcal{H}\, \mathbf{w}_j &= \mathbf{d}_j^T \mathcal{H}\, [\mathbf{w}_1 + \textstyle\sum_{i=1}^{(j-1)} \alpha_i\, \mathbf{d}_i] \\ &= \mathbf{d}_j^T \mathcal{H}\, \mathbf{w}_1 + \textstyle\sum_{i=1}^{(j-1)} \alpha_i\, \mathbf{d}_j^T \mathcal{H}\, \mathbf{d}_i = \mathbf{d}_j^T \mathcal{H}\, \mathbf{w}_1 \end{aligned}\right)$$

Obtaining $\mathbf{d}_i$ from $\mathbf{g}_i$

- Given gradient $\mathbf{g}_{j+1}$
  let $\mathbf{d}_{j+1} := -\mathbf{g}_{j+1} + \beta_j \mathbf{d}_j$

- Find β_j such that: $\mathbf{d}_{j+1}{}^T \mathcal{H}\, \mathbf{d}_j = 0$

$$\Rightarrow \quad \mathbf{g}_{j+1}{}^T \mathcal{H}\, \mathbf{d}_j = \beta_j\, \mathbf{d}_j{}^T \mathcal{H}\, \mathbf{d}_j$$

$$\Rightarrow \quad \boxed{\beta_j = \frac{g_{j+1}^T H d_j}{d_j^T H d_j}}$$

Simpler version of $\beta_j = \frac{g_{j+1}^T H d_j}{d_j^T H d_j}$

- Observe

  $\mathbf{g}_{j+1} - \mathbf{g}_j = [\mathcal{H}\, \mathbf{w}_{j+1} + b] - [\mathcal{H}\, \mathbf{w}_j + b]$

  $= \mathcal{H}\, [\mathbf{w}_{j+1} - \mathbf{w}_j\,] \;=\; \mathcal{H}\, [\, \alpha_j\, \mathbf{d}_j] \;=\; \alpha_j\, \mathcal{H}\, \mathbf{d}_j$

- So… $\mathcal{H}\, \mathbf{d}_j \;= [\mathbf{g}_{j+1} - \mathbf{g}_j]/\, \alpha_j$

$$\beta_j = \frac{g_{j+1}^T H d_j}{d_j^T H d_j} = \frac{g_{j+1}^T [g_{j+1} - g_j]/\alpha_j}{d_j^T [g_{j+1} - g_j]/\alpha_j} = \frac{g_{j+1}^T [g_{j+1} - g_j]}{d_j^T [g_{j+1} - g_j]}$$

- Note $\mathbf{d}_j^T\, \mathbf{g}_k = 0 \quad \forall j < k$

Computing Actual Direction **d**

- $\mathbf{d}_{j+1} := -\mathbf{g}_{j+1} + \beta_j \mathbf{d}_j$ where $\beta_j = \dfrac{g_{j+1}^T H d_j}{d_j^T H d_j}$
- Assuming **J** is quadratic...
  - ☐ Hestenes-Stiefel: $$\boxed{\beta_j = \frac{g_{j+1}^T [g_{j+1} - g_j]}{d_j^T [g_{j+1} - g_j]}}$$
  - ☐ Polak-Ribiere: $$\boxed{\beta_j = \frac{g_{j+1}^T [g_{j+1} - g_j]}{g_j^T g_j}}$$
  - ☐ Fletcher-Reeves: $$\boxed{\beta_j = \frac{g_{j+1}^T g_{j+1}}{g_j^T g_j}}$$
- If **J** is NOT quadratic, Polak-Ribiere seems best
  [If gradients similar, $\beta \approx 0$, so $\approx$restarting!]

Conjugate Gradient Algorithm

- Update parameters: $\mathbf{w}_{j+1} := \mathbf{w}_j + \alpha_j\, \mathbf{d}_j$
  - ☐ To get DIRECTION $\mathbf{d}_j$
    - $\mathbf{d}_1 := -\mathbf{g}_1$
    - $\mathbf{d}_{j+1} := -\mathbf{g}_{j+1} + \beta_j\, \mathbf{d}_j$
  - ☐ To find appropriate distance

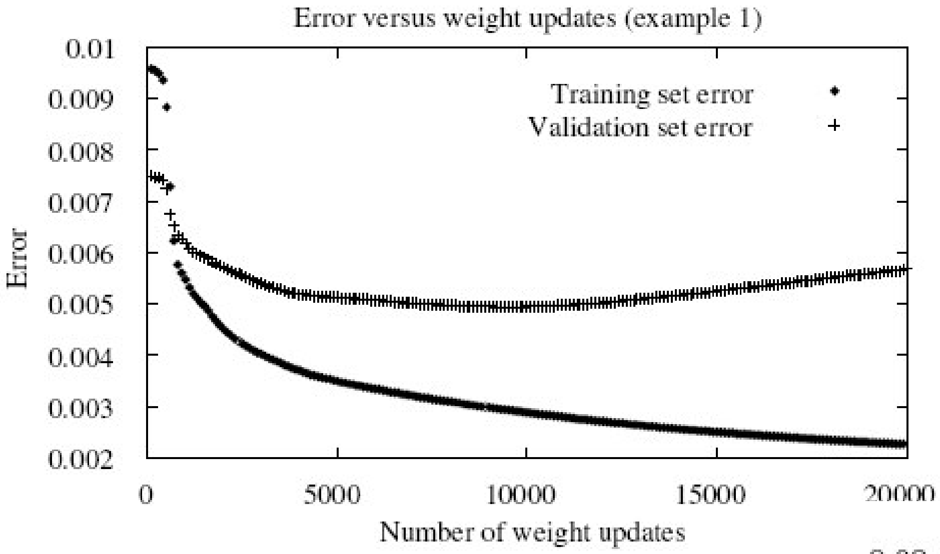
$$\boxed{\beta_j = \frac{g_{j+1}^T [g_{j+1} - g_j]}{g_j^T g_j}}$$

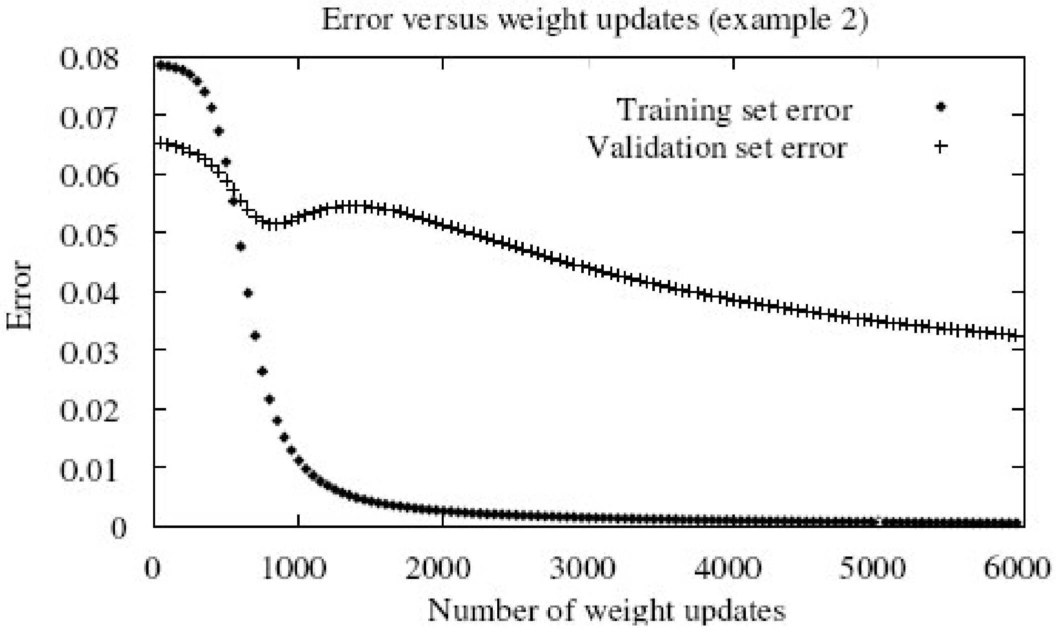
$$\boxed{\alpha_j = -\frac{\mathbf{d}_j^T \mathbf{g}_j}{\mathbf{d}_j^T \mathbf{H} \mathbf{d}_j}}$$

- If $\mathbf{J}$ quadratic, converge in n steps!

  If not… sometimes reset: $\mathbf{d}_t := -\mathbf{g}_t$
- Do not need to compute Hessian $\mathcal{H}$ for β

Overfitting in ANNs





Local ≠ Global Optimum

- Techniques so far: Seek **LOCAL** minimal
- For Linear Separators: PERFECT

  ∃ 1 minimum

  ... if everything nearby looks “bad” ⇒ Done!
- Not true in general!

- Simulated Annealing

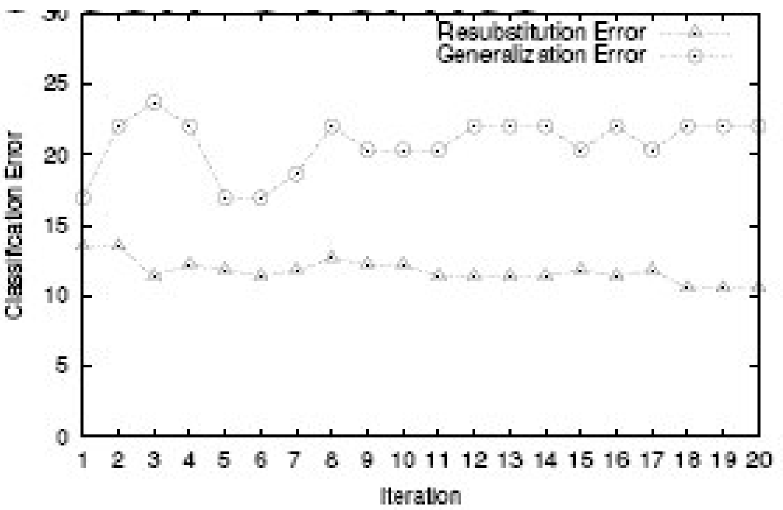
  Go wrong-way sometimes ...

  with diminishing probabilities

4. Stopping Criteria

- After N iterations? (for fixed N)
- When resubstitution error is suff. small?

  BAD: often overfits



- Use “validation data set”
  1. Do many iterations,
     then use weights from high-water mark
  2. Cross validation:
     Plot # iterations vs error → opt = r_i
     Let $\underline{r}$ =median(r_i)
     Use all data, for $\underline{r}$ iterations

Regularized Error Functions

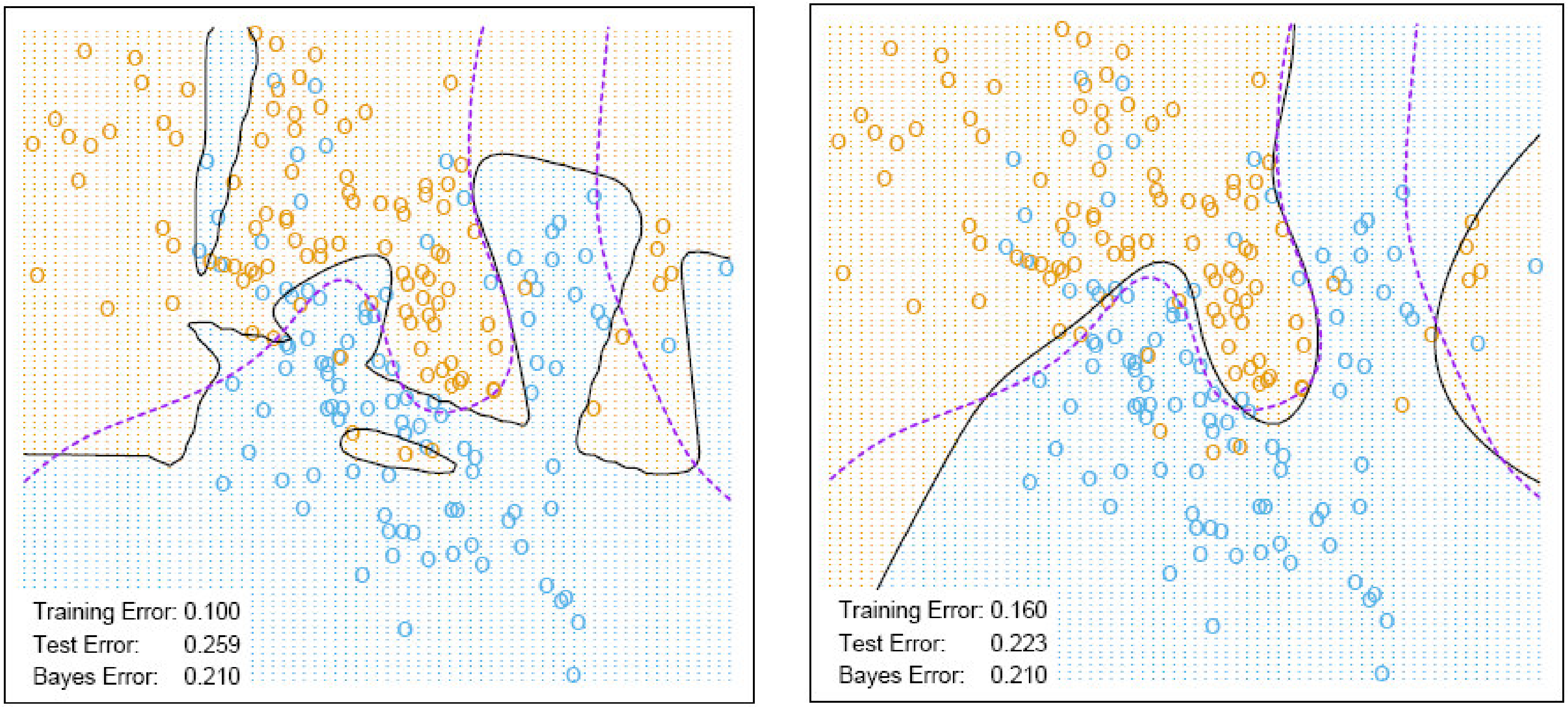
- Penalize large weights: “Regularizing” … “weight decay”

$$E(\vec{w}) \equiv \frac{1}{2}\sum_{d\in D}\sum_{k\in outputs}(t_{kd}-o_{kd})^2 + \gamma\sum_{i,j} w_{ij}^2$$

$$E(\vec{w}) \equiv \frac{1}{2}\sum_{d\in D}\sum_{k\in outputs}(t_{kd}-o_{kd})^2 + \gamma\sum_{i,j} \frac{w_{ij}^2}{1+w_{ij}^2}$$

- ≈ ridge regression

Example



No Weight Decay

Weight Decay=0.02

Neural Network - 10 Units

Other Ideas

- Train on target slopes as well as values:

  (more constraints…)

$$E(\vec{w}) \equiv \frac{1}{2}\sum_{d\in D}\sum_{k\in outputs}\left[(t_{kd}-o_{kd})^2+\mu\sum_{j\in inputs}\left(\frac{\partial t_{kd}}{\partial x_d^j}-\frac{\partial o_{kd}}{\partial x_d^j}\right)^2\right]$$

- Tie together weights:

  – eg, in phoneme recognition network

  (Fewer weights, …)
- Multiple restarts
- Change structure

Dynamically Modifying Network Structure

- So far, assume structure FIXED..
  … only learning values of WEIGHTS
- Why not modify structure as well?

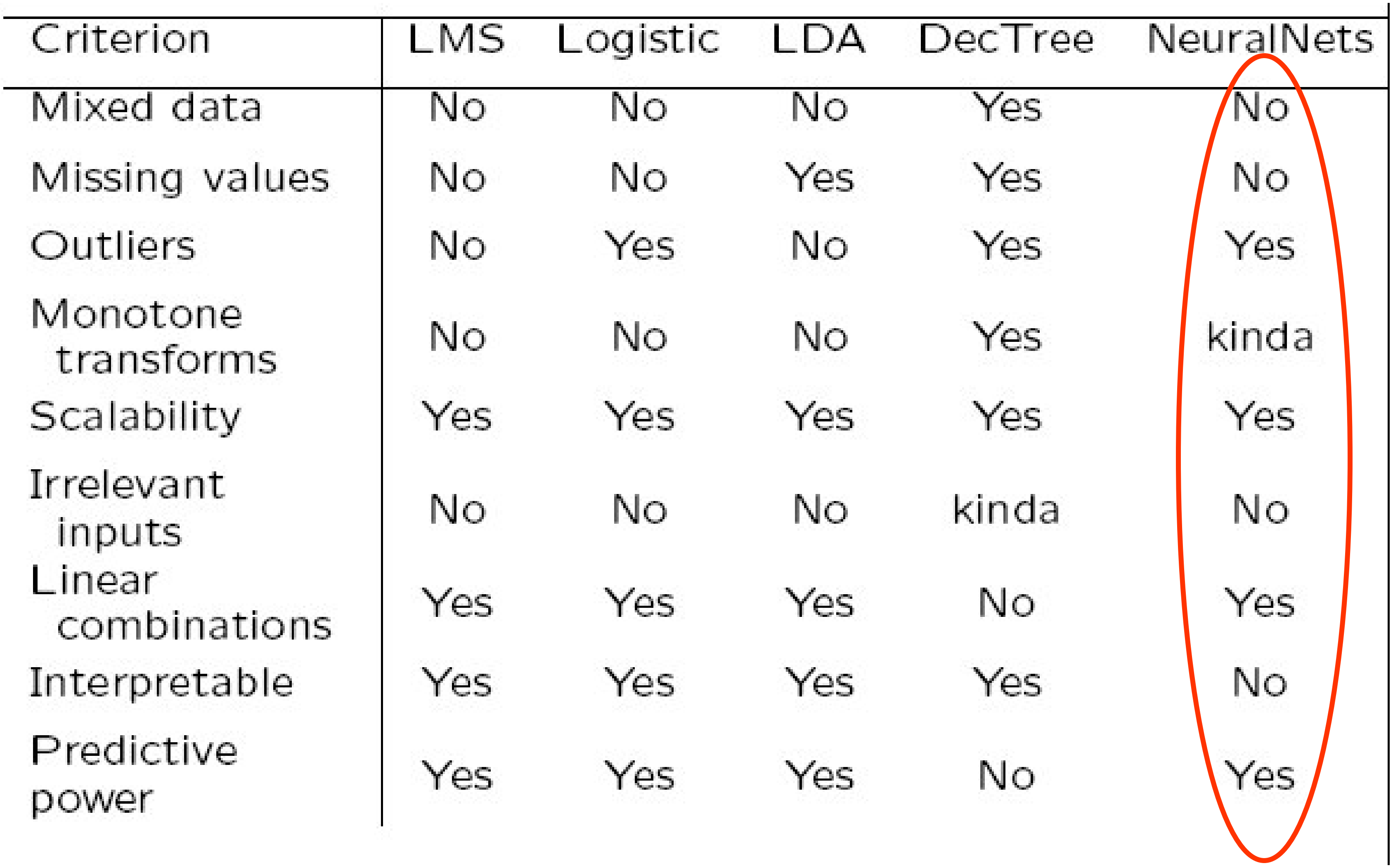
"Cascade Correlation"

1. Initially: NO hidden units
   . . . just direct connections from input-output
2. Find best weights for this structure
3. If good fit: STOP.
   Otherwise. . . if significant residual error:
4. Produce new hidden unit
   from previous units, and to all output units
   w/weights CORRELATED to residual error
   Goto 2

"Optimal Brain Damage"

start w/ complex network,
prune "inessential" connections
Inessential if $w_i \approx 0$
. . . or $dE/d\, w_i \approx 0$

Neural Network Evaluation



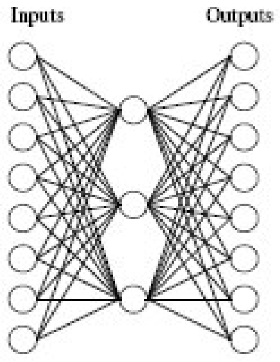
Criterion	LMS	Logistic	LDA	DecTree	NeuralNets
Mixed data	No	No	No	Yes	No
Missing values	No	No	Yes	Yes	No
Outliers	No	Yes	No	Yes	Yes
Monotone transforms	No	No	No	Yes	kinda
Scalability	Yes	Yes	Yes	Yes	Yes
Irrelevant inputs	No	No	No	kinda	No
Linear combinations	Yes	Yes	Yes	No	Yes
Interpretable	Yes	Yes	Yes	Yes	No
Predictive power	Yes	Yes	Yes	No	Yes

Outline

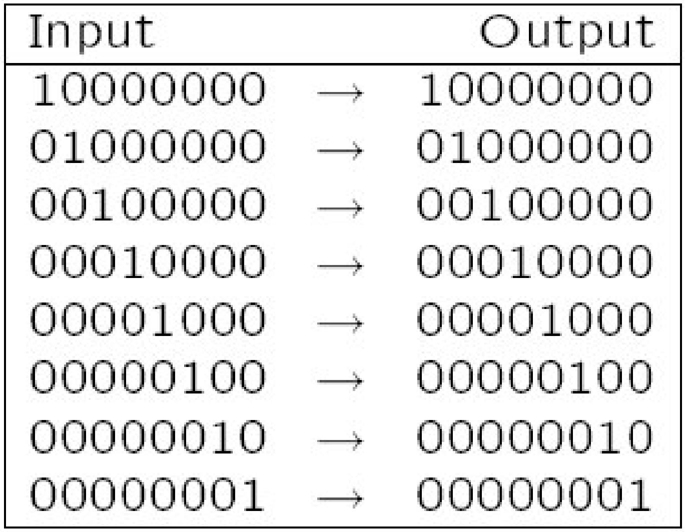
- Introduction
  - Historical Motivation, non-LTU, Objective
  - Types of Structures
- Multi-layer Feed-Forward Networks
  - Sigmoid Unit
  - Backpropagation
- Tricks
  - Line Search
  - Conjugate Gradient
  - Alternative Error Functions
- Hidden layer representations
  - Example: Face Recognition
- Recurrent Networks

Learning Hidden Layer Repr'n

- Auto-encoder:



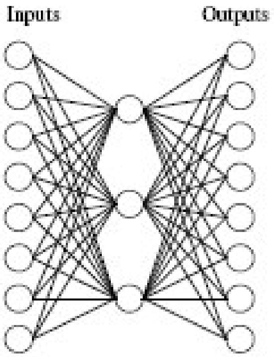
- Goal: Learn

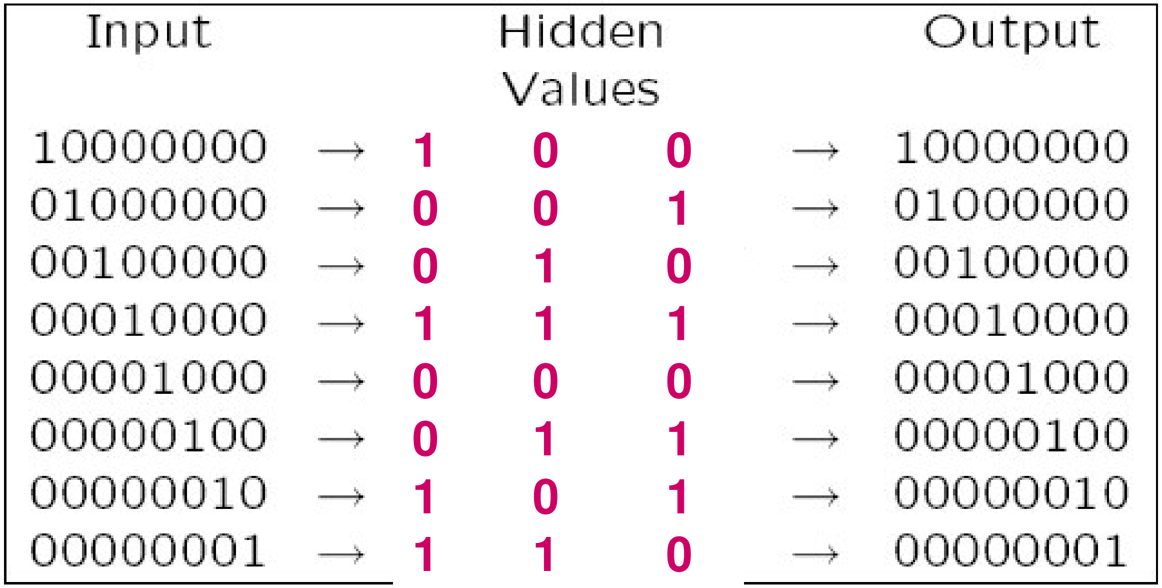


Input		Output
10000000	→	10000000
01000000	→	01000000
00100000	→	00100000
00010000	→	00010000
00001000	→	00001000
00000100	→	00000100
00000010	→	00000010
00000001	→	00000001

Hidden Layer Representations

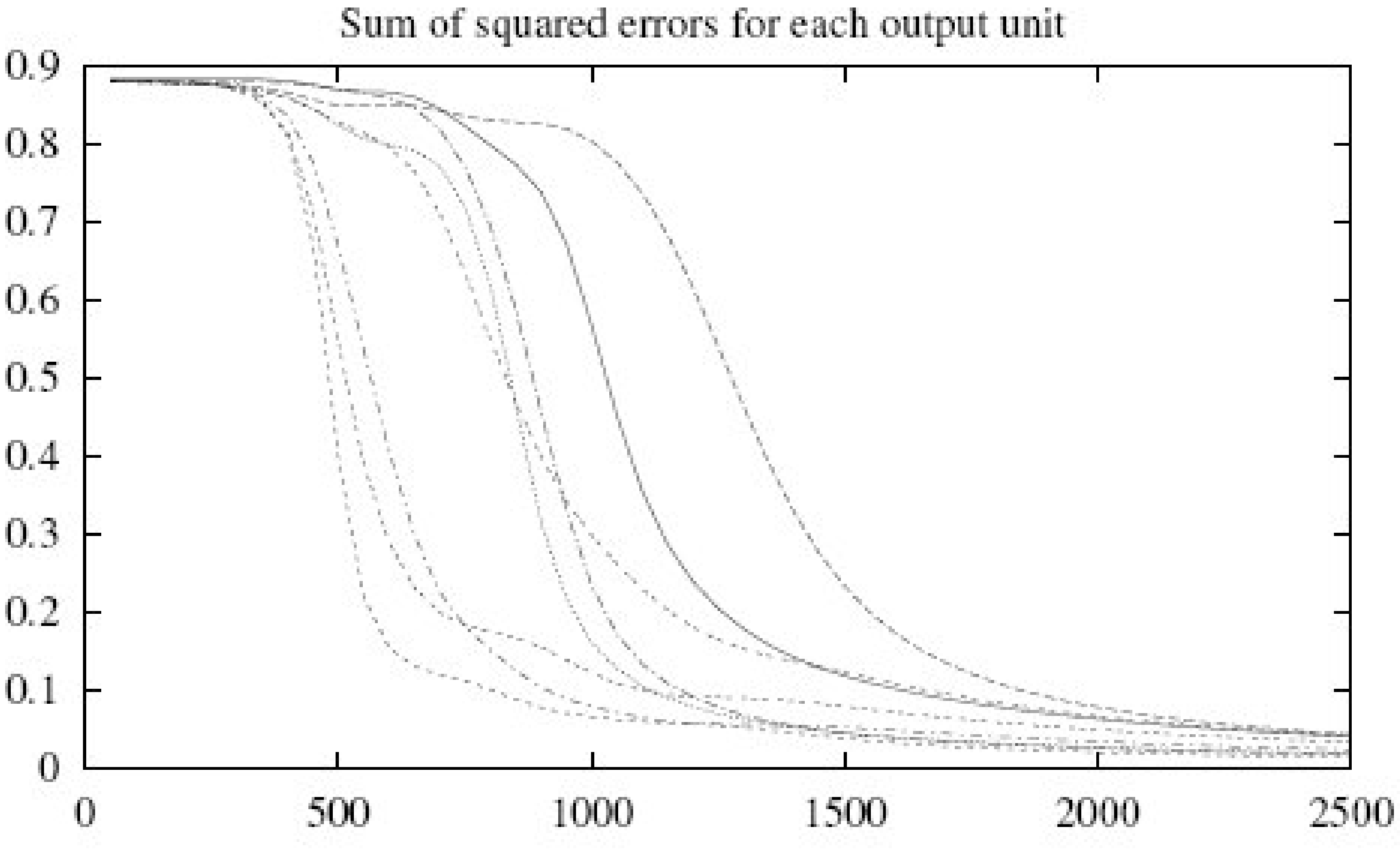
- Learned hidden layer representation:



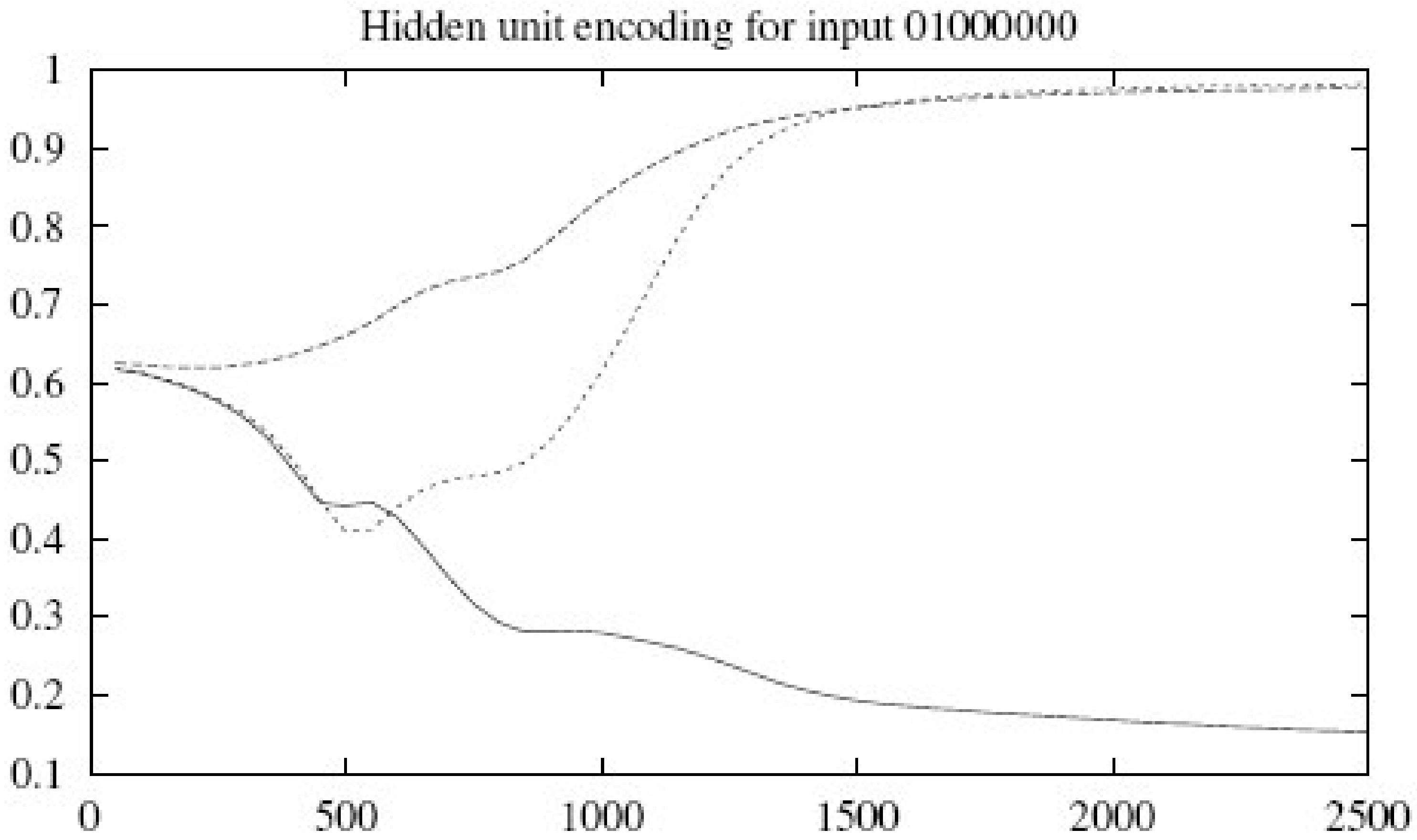


Input		Hidden Values				Output
10000000	→	1	0	0	→	10000000
01000000	→	0	0	1	→	01000000
00100000	→	0	1	0	→	00100000
00010000	→	1	1	1	→	00010000
00001000	→	0	0	0	→	00001000
00000100	→	0	1	1	→	00000100
00000010	→	1	0	1	→	00000010
00000001	→	1	1	0	→	00000001

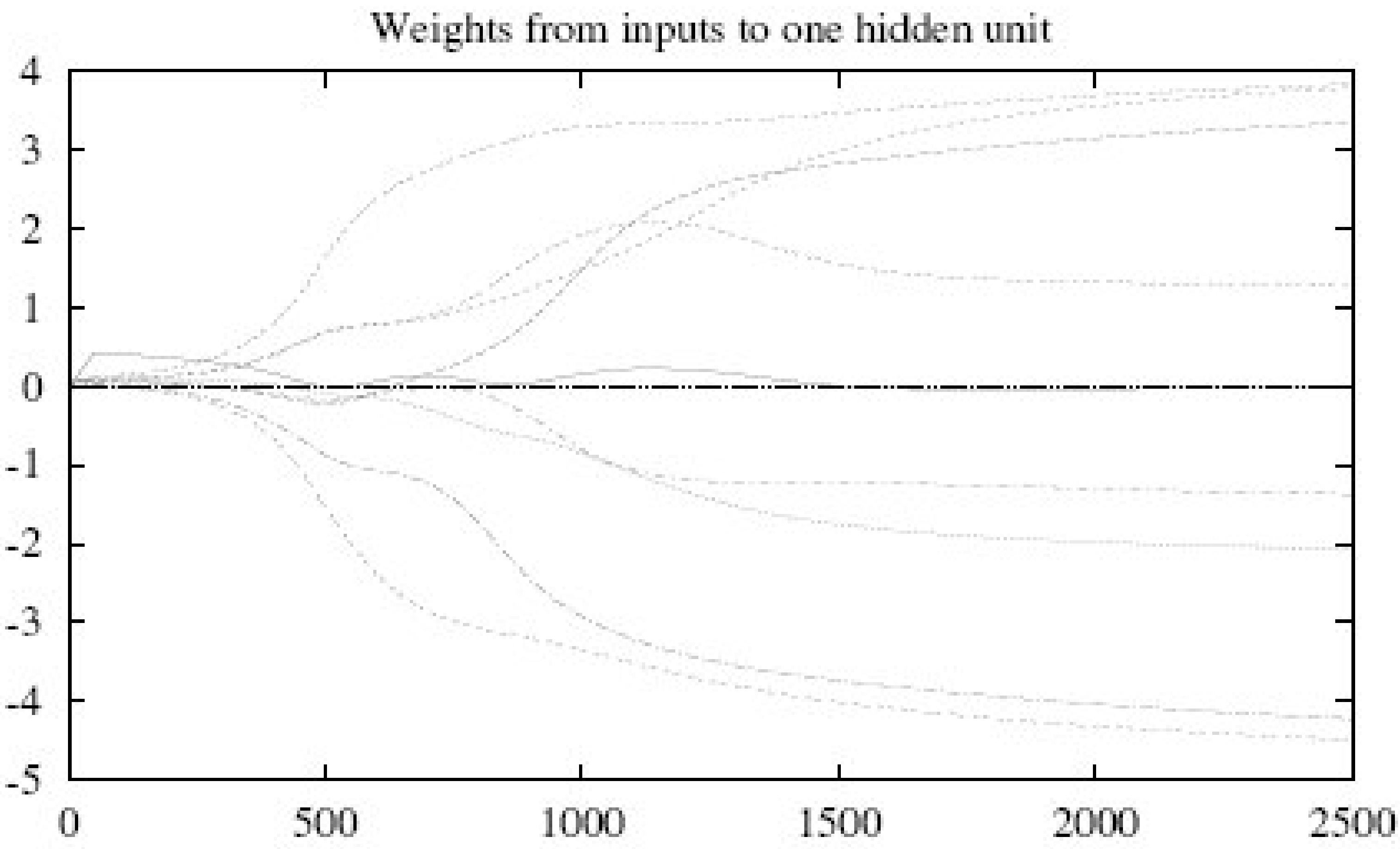
Training Curve #1



Training Curve #2



Training Curve #3



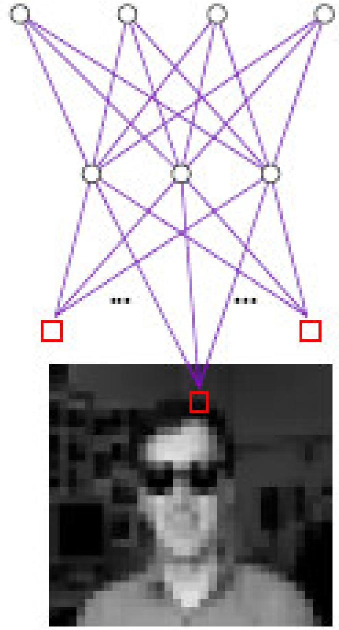
Neural Nets for Face Recognition

- **Performance Task:** Recognize DIRECTION of face
- **Framework**: Different people, poses, “glasses”, different background, . . .
- **Design Decisions:**
  - **Input Encoding:**
    - Just pixels? (subsampled? averaged?)
    - or perhaps lines/edges?
  - **Output Encoding:**
    - Single output ([0, 1/n] = #1, . . .)
    - Set of n-output (take highest value)
  - **Network structure**: # of layers
    - Connections (training time vs accuracy)
  - **Learning Parameters:** Stochastic?
    - Initial values of weights?
    - Learning rate η, Momentum α, . . .
    - Size of Validation Set, . . .

Neural Nets Used

left strt rght up

Typical input images

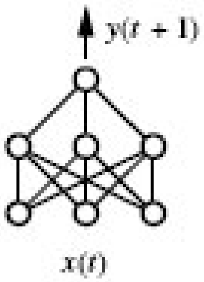


90% accurate learning head pose,
and recognizing 1-of-20 faces

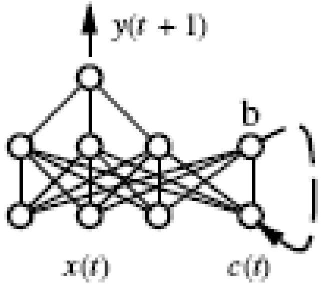
Recurrent Networks

- Brain needs short-term memory, . . .
  ⇒ feedforward network not sufficient.
- Brain has many feed-back connections.
  ⇒ brain is recurrent network, with Cycles!
- Recurrent nets:
  - Can capture internal state.
    (activation keeps going around)
  - More complex agents
  - Much harder to analyze.
    … Unstable, Oscillate, Chaotic
- Main types:
  - Iterative model
  - Hopfield networks
  - Boltzmann machines

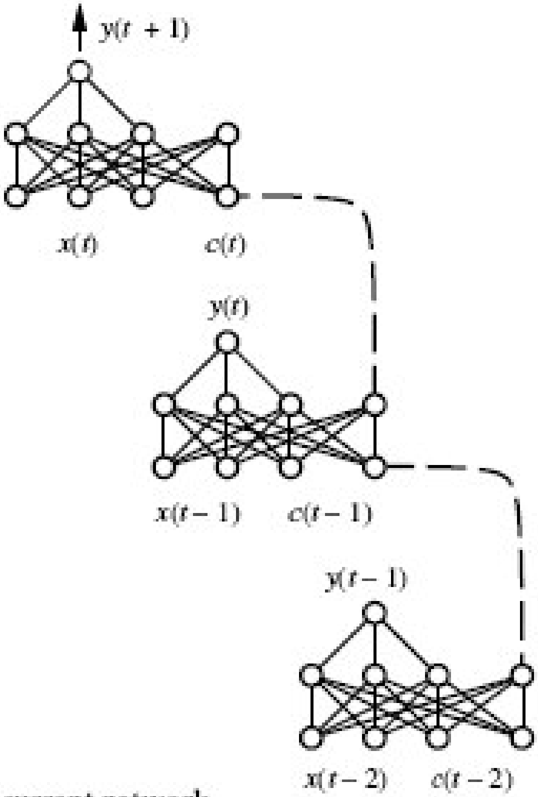
Iterative Recurrent Network



(a) Feedforward network



(b) Recurrent network



(c) Recurrent network
unfolded in time

Hopfield Networks

- Symmetric connections ($W_{i,j} = W_{j,i}$)
  - Activation only $\{+1, -1\}$
  - $\sigma(.)$ is sign-function
- Train weights to obtain associative memory
  - eg, store patterns
- After learning, can "retrieve" patterns:
  - Set some node values,
  - other nodes settle to best pattern match

- **Theorem**:
  An N-unit Hopfield net can store up to
  0.138N patterns reliably.

- Note: No explicit storage; all in the weights!

Boltzmann Machines

- Symmetric connections ($W_{i,j} = W_{j,i}$)
- Activation only {+1, -1 }, but stochastic
- $P(n_i = 1)$ depends on inputs
  - ☐ Network in constant motion,
    computing average output value of each node

  . . . like simulated annealing
- Has nice (but slow) learning algorithm.
- Related to probabilistic reasoning

  … belief networks!

Other Topics

- Architecture
- Initialization
    - Incorporating Background Knowledge
    - KBANN, ...
- Better statistical models
    - When to use which system?
    - Other training techniques
    - Regularizing
- Other “internal” functions
    - Sigmoid
    - Radial Basis Function

What to Remember

- Neural Nets can represent arbitrarily complex functions
- It can be challenging to LEARN the parameters, as multiple local optima
  - … gradient descent … using backpropagation
- Many tricks to make gradient descent work!
  - Line search
  - Conjugate gradient

… useful for ANY optimization (not just NN)