Lecture 19: Learning -2

Victor Lesser

CMPSCI 683
Fall 2004

Today's Lecture

- **Neural Networks**
  - Representing functions using networks of simple arithmetic computing elements
  - Learning such representations from examples

Biological Inspiration Learning: The Brain

- Approximately 10^{11} neurons, 10^4 synapses (connections) per neuron.
- Neuron "fires" when its inputs exceed a threshold.
- Inputs are weighted and can have excitory or inhibitory effect.
- Individual firing is slow (≈ .001 second) but bandwidth is very high (≈ 10^{14} bits/sec).
- The brain performs many tasks much faster than a computer (Scene recognition time ≈ .1 second).
- Learning and graceful degradation.

What is Connectionist Computation?

Computational architectures and cognitive models that are neurally-inspired:

- **Faithful to coarse neural constraints — not neural models**
- **Large numbers of simple (neuron-like) processing units interconnected through weighted links**
- **They do not compute by transmitting symbolically coded messages**
  - Inhibitory and excitory signals
- ***"program" resides in the structure of the interconnections***
- **"massive parallelism" and no centralized control**

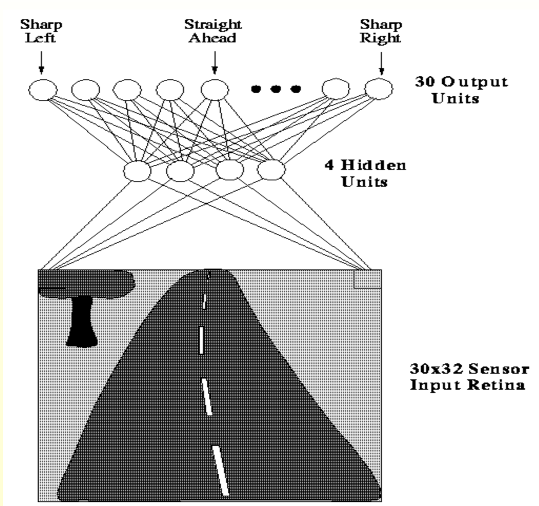
Some Properties of Connectionist Systems

- Ability to bring large numbers of interacting constraints to bear on problem solving (soft constraints)
- Noise resistance, error tolerance, graceful degradation
- Ability to do complex multi-layer recognition with a large number of inputs/outputs (quickly)
- Learning with generalization
- Biological plausibility
- Potential for speed of processing through fine-grained parallelism

Applications of neural networks

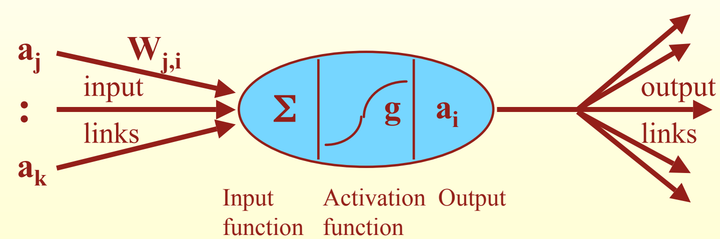
- Automobile automatic guidance systems
- Credit application evaluation, mortgage screening, real estate appraisal
- Object recognition (faces, characters)
- Speech recognition and voice synthesis
- Market forecasting, automatic bond trading
- Robot control, process control
- Breast cancer cell analysis
- Oil and gas exploration
- Image and data compression

ALVINN drives 70 mph on highways

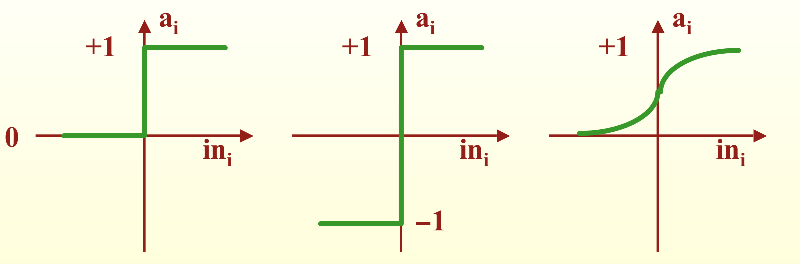


Artificial Neural Networks

- **Compose of nodes/units connected by links**
- **Each link has a numeric weight associated with it**
- **Processing units compute weighted sum of their inputs, and then apply a threshold function.**
  - Linear function combines inputs= sum ($w_{j,I}$• aj .... $w_{k,I}$• a_k)
  - Non-linear function transforms g combined input to activation value



Sample G's - activation functions



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

Can make each of these functions a threshold such that it outputs a 1 when the input is greater than threshold, can also through dummy link

Representation of Boolean Functions

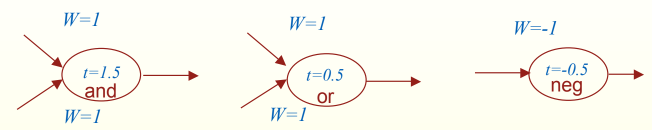
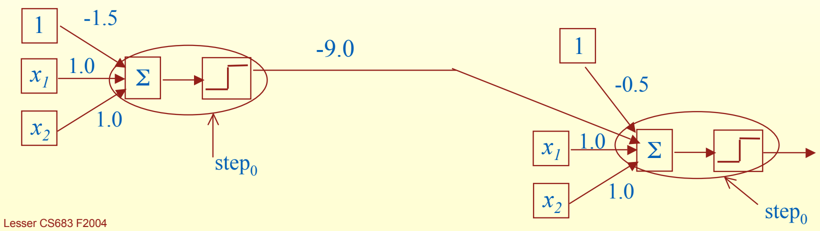


Figure 19.6 Units with a step function for the activation function can act as logic gates, given appropriate thresholds and weights.

> XOR requires multi-layer network



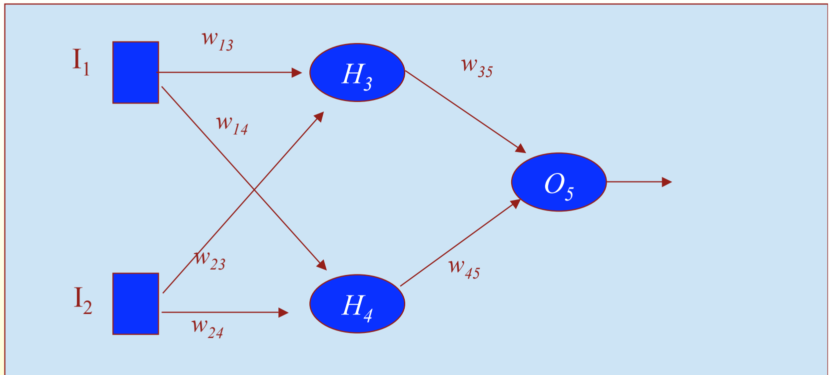
Neural Network Learning

- **Robust approach to approximating real-valued, discrete-value and vector-valued target functions**
- **Learning the Weights (and Connectivity)**
  - $w_{j,i}$ = 0 implies no connectivity among nodes a_j and a_i

Network Structure

- **Feed-Forward Networks: unidirectional links**
  - No cycles (DAG)
  - No internal state other than weights
  - **Layered feed-forward**
    - Each unit is linked only to units in the next layer
    - Synchronized movement of information from layer to layer
  - Relatively understood

Multi-Layer Network: Hidden Units



A very simple, two-layer, feed-forward network with two inputs, two hidden nodes, and one output node.

Network Structure cont.

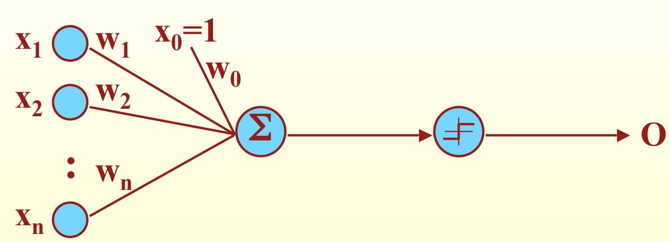
- **Recurrent Network: arbitrary links**
  - Activation is fed back to units that caused it
  - Internal state stored in activation levels
  - Can be unstable, oscillate etc.
  - Can represent more complex functions

Hopefield Networks

- Have bidirectional connections with symmetric weights
- All units are both input and output units
- Activation function is the sign function (can only be +1 or –1).
- Functions as an associative memory: a new example will cause the net to settle into a training pattern that most closely resembles the new example.
- Training set of photographs
  - **Each weight is a partial encoding of all photographs**
  - **New stimulus small piece of one of the trained photographs**
    - **Activation levels of neural units will reflect correct photograph**

Perceptron

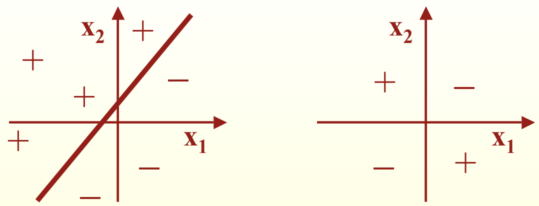
- **Single-layered feed-forward networks studied in the late 1950's.**



$$o(x_1,\ldots,x_n) = \begin{cases} 1 & \text{if } w_0 + w_1x_1 + \cdots + w_nx_n > 0 \\ -1 & \text{otherwise.} \end{cases}$$

Decision Surface of a Perceptron

Hyperplane in the input space



- Represents some useful functions
  - **linearly separable**
- But some functions not representable
  - **XOR**

Problem Encoding

- **Local encoding**
  - Each attributed single input value
  - Pick appropriate number of distinct values to correspond to distinct symbolic attributed value
- **Distributed encoding**
  - One input value for each value of the attribute
  - Value is one or zero whether value has that attribute
    - X between 0 and 3; 4 distinct inputs y1,y2,y3,y4;
    - X=3; y1=0,y2=0,y3=0,y4=1

Perceptron Learning

Perceptron learning rule:

$w_i \leftarrow w_i + \alpha(t-o)x_i$; *reduce difference between observed and predicted values in small increments*

where:

- t is the target value of training example
- o is the perceptron output
- α is a small constant (e.g., .1) called the **learning rate**
- x_i is either 1 or -1

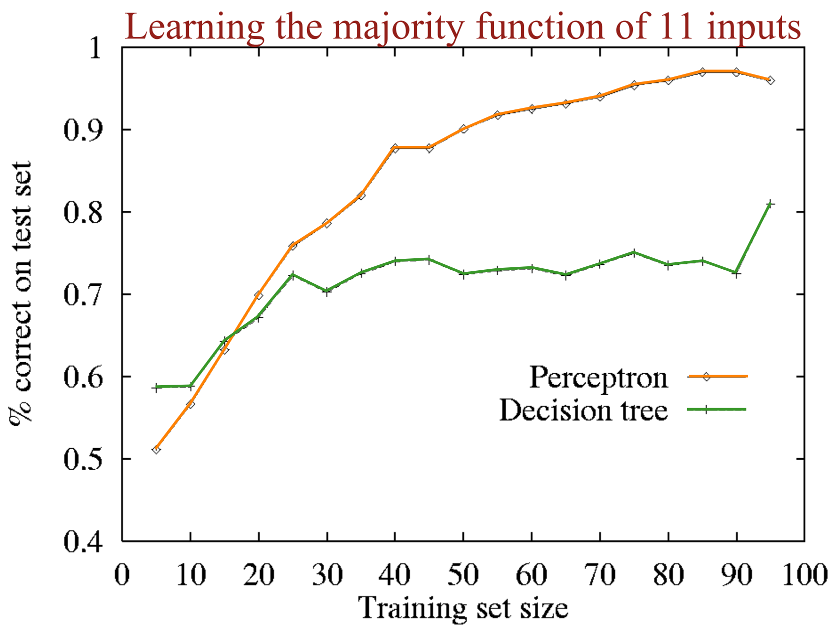
Start out with randomly assigned weights between [-0.5,0.5]

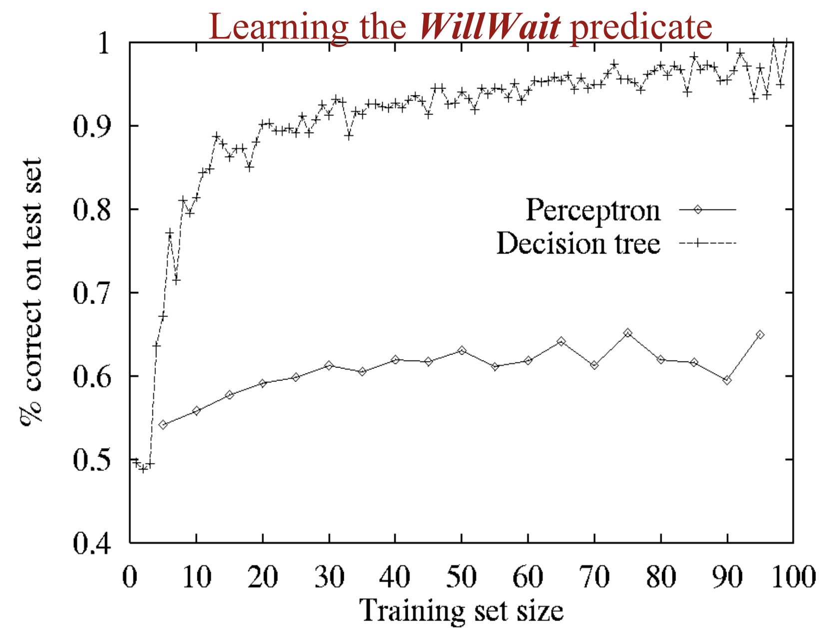
Perceptron Convergence Theorem

The perceptron learning rule will converge to a set of weights that correctly represents the examples, as long as the examples represent a “linearly separable” function and α is sufficiently small.

Why does it work? Perceptron is doing gradient descent in weight space that has no local minima.

- Optimization in the weight space based on sum of squared errors

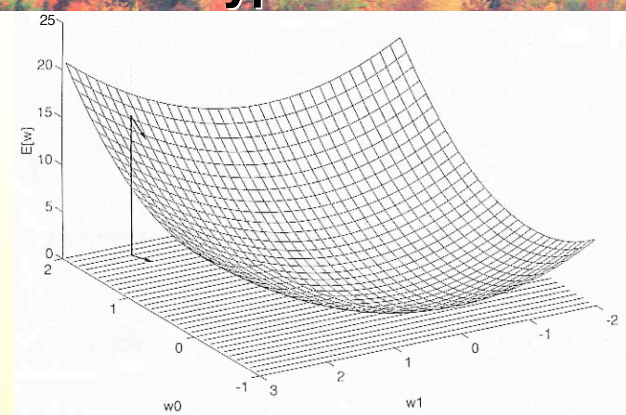




Gradient Descent and the Delta Rule

- **Delta Rule:** $w_i \leftarrow w_i + \alpha \Sigma_D (t_d - o_d) x_{id}$
  - D is set of entire training examples
  - If training examples are not linearly separable, Delta rule converges towards best-fit approximation to target concept
  - Use gradient descent search to search hypothesis space of possible weight vectors to find the weights that best fit the training example
    - Arbitrary initial weight vector
  - At each step, weight vector is altered in the direction that produces the steep descent along error surface until global minimum error is reached
    - ***least mean square error*** over ***all training examples***

Error Surface of Different Hypotheses



For a linear unit with two weights, the hypothesis space H is the w_0, w_1 plane. The vertical axis indicates the **error of the corresponding weight vector hypothesis**, relative to a **fixed set of training examples**. The arrow shows the negated gradient at one particular point, indicating the direction in the w_0, w_1 plane producing steepest descent along the error surface.

Delta Rule continued

- **Convergence guaranteed for perceptron since error surface contains only a single global minimum and learning rate sufficiently small**
  - large number of iterations
- **Larger learning rate**
  - Possibly overshoot minimum in the error surface
  - Can use larger learning rate if gradually reduce value of learning rate over time
    - Similar to simulated annealing

Stochastic Approximation to Gradient Descent

- **Incremental gradient descent by updating weights *per example***
  - $w_i \leftarrow w_i + \alpha(t - o')x_i$
- **Looks similar to perceptron rule**
  - O' not thresholded perceptron (no g) output rather thresholded linear combinations of inputs $w\,x$
- **Reduces cost of each update cycle**
- **Needs smaller learning rate**
  - More update cycles than gradient descent

Multilayer networks

- Problem with Perceptrons is coverage: many functions cannot be represented as a network.
  - **sum threshold function**
- But with one "hidden layer" and the sigmoid threshold function, can represent any continuous function.
  - **Choosing the right number of hidden units is still not well understood**
- With two hidden layers, can represent any discontinuous function.

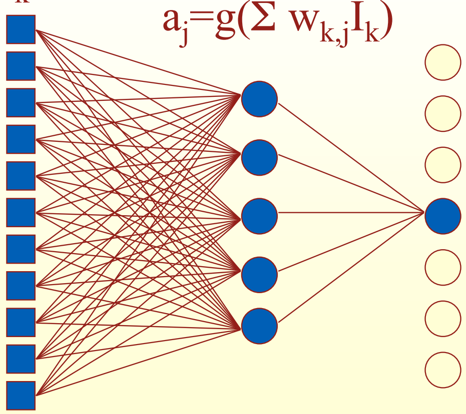
Learning in Multi-Layered Feed-Forward Networks

- **Back-Propagation Learning**
  - How to assess the blame for an error and divide it among the contributing weights at the same and different layers
  - Gradient descent over network weight vector

Hidden layers

I_k

$a_j = g(\Sigma\, w_{k,j} I_k)$



$O_i = g(\Sigma\, w_{j,i} a_j)$

2-Layer Stochastic Back-Propagation

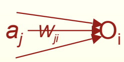
- Provides a way of dividing the calculation of gradient among the units, so that change in each weight can be calculated by the unit to which the weight is attached, using only local information
- Based on minimizing

$$E(W) = \frac{1}{2}\sum_i \left(T_i - O_i\right)^2 \quad ; i \text{ multiple output units}$$

Back-Propagation, cont.

- First level of Back propagation to hidden layer

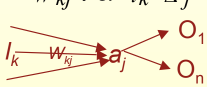
$$W_{ji} \leftarrow W_{ji} + \alpha \cdot a_j \cdot \Delta_i \qquad \Delta_i = \left(T_i - O_i\right) \cdot g'_i \sum_j \left(W_{ji} \cdot a_j\right)$$



Gradient of error (O_i) with respect to W_{ji}

- Second level of Back propagation to input layer

$$W_{kj} \leftarrow W_{kj} + \alpha \cdot I_k \cdot \Delta_j \qquad \Delta_j = g'_j \left(\sum_k W_{kj} I_k\right) \cdot \underline{\sum_i W_{ji} \Delta_i}$$



Gradient Affect of W_{kj} on o_i's

  - Summing the error terms for *each* output unit influence by w_{kj} thru a_j, weighting each by the w_{ji}; the degree to which hidden unit is "responsible for" error in output

Steps in Back-Propagation

- Compute the delta values for the output units using the observed error
- Starting with output layer, repeat the following for each layer in the network
  - Propagate delta values back to previous layer
  - Update the weights between the two layers

Back-Propagation, cont.

Typically use sigmoid function : $g(x) = \dfrac{1}{1+e^{-x}}$

Nice proprty : $\dfrac{dg(x)}{dx} = g(x)(1-g(x))$

Gradient of error E with respect to weight w_i :

$$\frac{\partial E}{\partial w_i} = -\sum_{d\in D}(t_d - o_d)o_d(1-o_d)x_{i,d}$$

Back-Propagation Algorithm

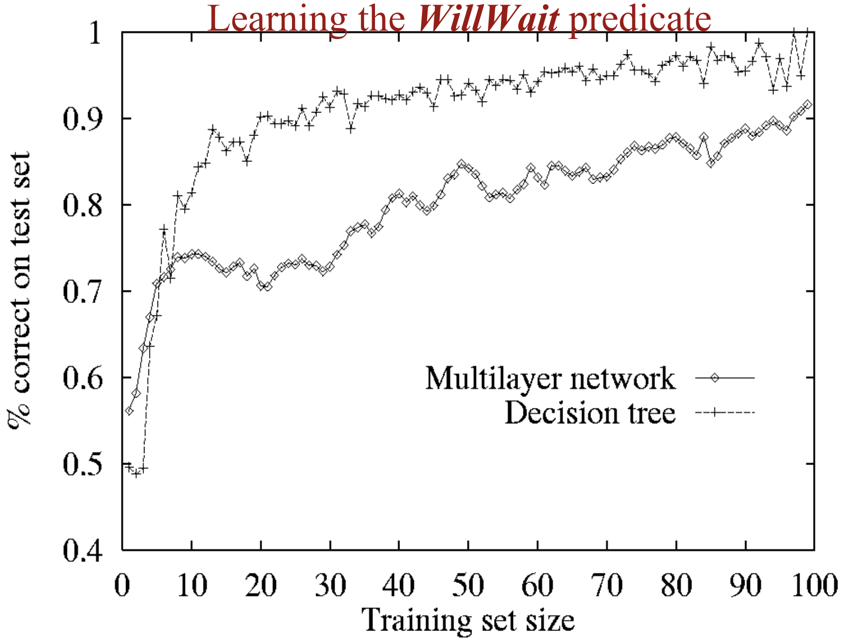
Initialize all weights to small random numbers

Repeat until satisfied: For each training example:

1. Compute the network outputs
2. For each output unit k : $\delta_k \leftarrow (t_k - o_k)o_k(1-o_k)$
3. For each hidden unit h : $\delta_h \leftarrow o_h(1-o_h)\sum_{k\in \text{outputs}} w_{h,k}\delta_k$
4. Update each weight: $w_{i,j} \leftarrow w_{i,j} + \Delta w_{i,j}$

   where $\Delta w_{i,j} = \alpha\delta_j x_i$

Learning the ***WillWait*** predicate



Hypothesis Space

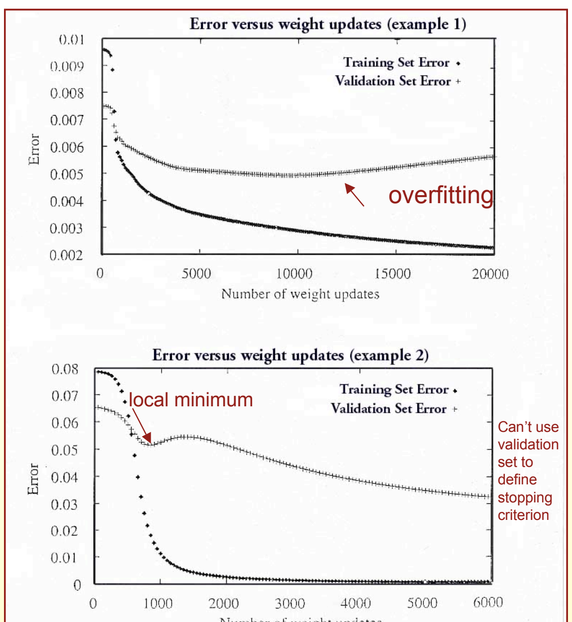
- **N-dimensional Euclidean Space of network weights**
- **Continuous**
  - Contrast with discrete space of decision tree
- **Error Measure is differentiable with respect to continous parameters**
  - Results in well-defined error gradient that provides a useful structure for organizing the search for the best hypothesis

Network Implicitly Generalizes

- **Smooth Interpolation between data points**
  - Smoothly varying decision regions
- **Tend to label points in between positive examples as positive examples if no negative examples**

Overfitting and Stopping Criteria

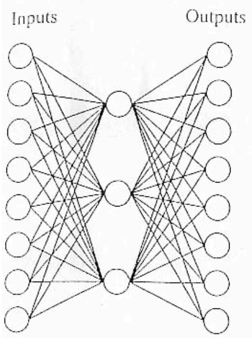
- **Backprop is susceptible to overfitting**
  - After initial learning weights are being tuned to fit idiosyncrasies of training examples and noise
  - Overly complex decision surfaces constructed
- **Weight Decay -- decrease weight by some small factor during each iteration thru data**
  - Keep weight values small to bias learning against complex decision surfaces
- **Exploit Validation Set**
  - Keep track of error in validation set during search
  - Use weight setting that minimizes error
  - Use as stopping criteria



Convergence

- **Error Surface can have multiple local minimum**
  - Guaranteed to converge only to local minimum
- **Momentum model**
  - Weight update partially dependent on the n-1 iteration
  - $\Delta w_{ji}(n)=\eta\ \delta_j x_{ji} + \alpha\Delta\ w_{ji}\ (n-1)$
  - Helps not to get stuck in local minimum
  - Gradually increasing the step size of the search in regions where the gradient is unchanging

Learned Hidden Layer Representation: New Features



Input		Hidden Values				Output
10000000	→	.89	.04	.08	→	10000000
01000000	→	.15	.99	.99	→	01000000
00100000	→	.01	.97	.27	→	00100000
00010000	→	.99	.97	.71	→	00010000
00001000	→	.03	.05	.02	→	00001000
00000100	→	.01	.11	.88	→	00000100
00000010	→	.80	.01	.98	→	00000010
00000001	→	.60	.94	.01	→	00000001

Figure 4.7 Learned Hidden Layer Representation. This 8 x 3 x 8 network was trained to learn the identity function, using the eight training examples shown. After 5000 training epochs, the three hidden unit values encode the eight district inputs using the encoding shown on the right. Notice if the encoded values are rounded zero or one, the result is the standard binary encoding for eight distinct values.

Back-propagation

- Gradient descent over network weight vector
- Easily generalizes to any directed graph
- Will find a local, not necessarily global error minimum
- Minimizes error over training examples — will it generalize well to subsequent examples?
- Training is slow — can take thousands of iterations.
- Using network after training is very fast

Applicability of Neural Networks

- Instances are represented by many attribute-value pairs
- The target function output may be discrete-valued, real-valued, or a vector of several real- or discrete-valued attributes
- The training examples may contain errors
- Long training times are acceptable
- Fast evaluation of the learned target function may be required
- The ability of humans to understand the learned target function is not important

Next Lecture

- **Reinforcement Learning**