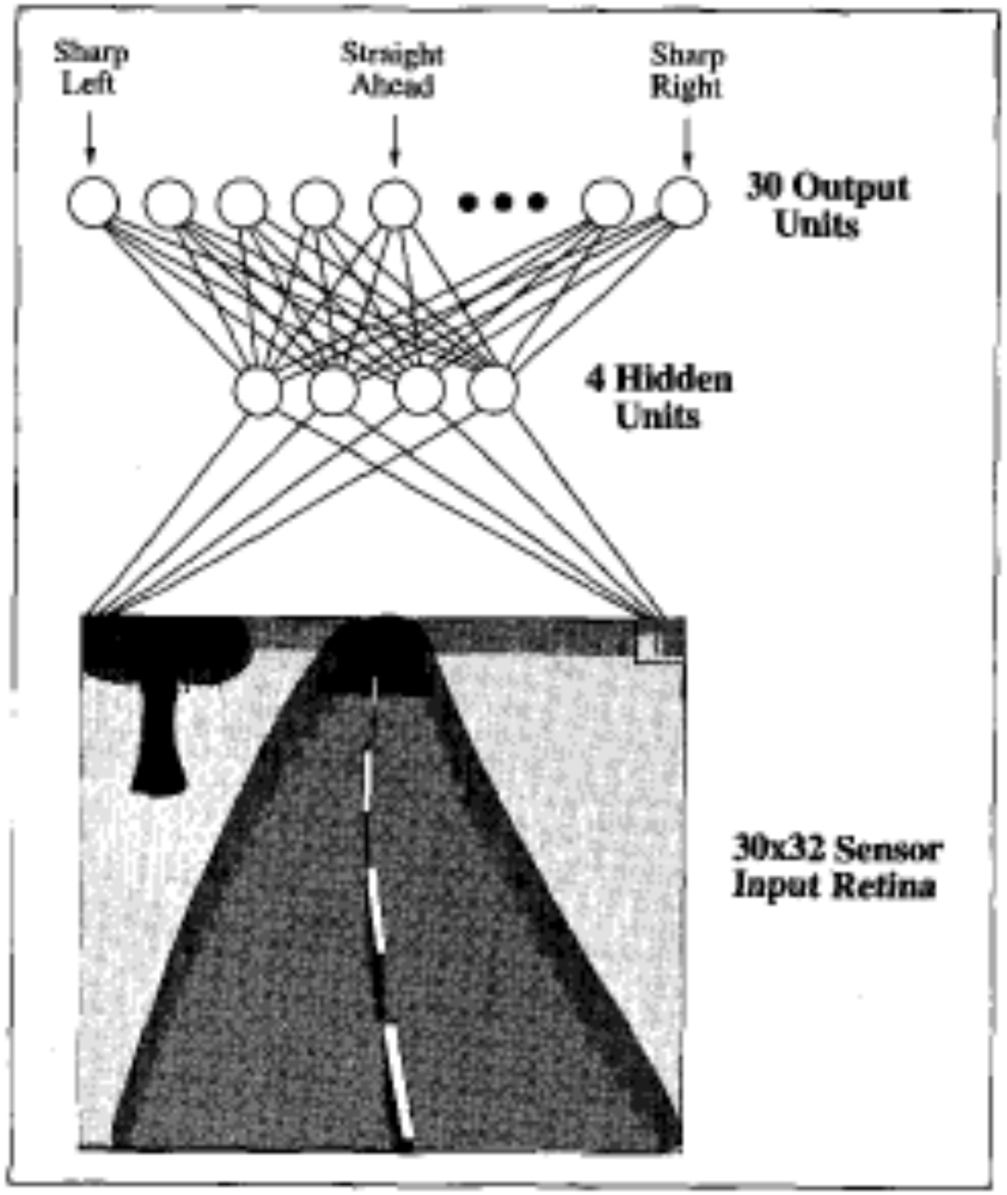
Artificial Neural Networks

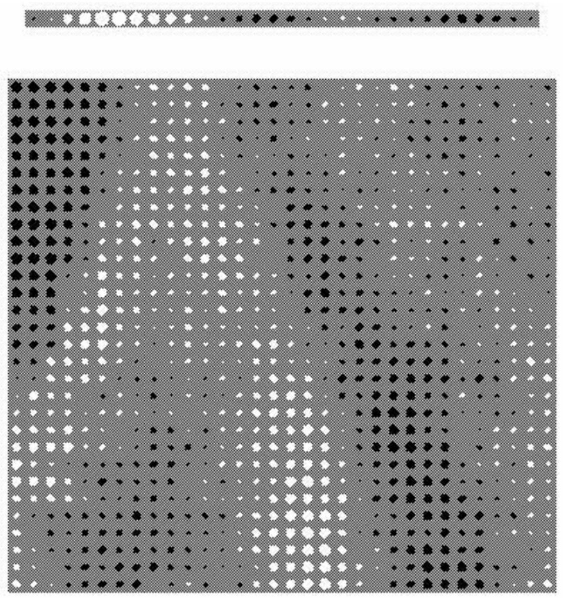
- Artificial neural networks (ANNs) provide a general, practical method for learning real-valued, discrete-valued, and vector-valued functions from examples.
- Algorithms such as BACKPROPAGATION gradient descent to tune network parameters to best fit a training set of input-output pairs.
- ANN learning is robust to errors in the training data and has been successfully applied to problems such as interpreting visual scenes, speech recognition, and learning robot control strategies.

Biological Motivation

- The study of artificial neural networks (ANNs) has been inspired in part by the observation that biological learning systems are built of very complex webs of interconnected neurons.
- Artificial neural networks are built out of a densely interconnected set of simple units, where each unit takes a number of real-valued inputs (possibly the outputs of other units) and produces a single real-valued output (which may become the input to many other units).
- The human brain is estimated to contain a densely interconnected network of approximately 10^{11} neurons, each connected, on average, to 10^4 others.
  - Neuron activity is typically inhibited through connections to other neurons.

ALVINN – Neural Network Learning To Steer An Autonomous Vehicle





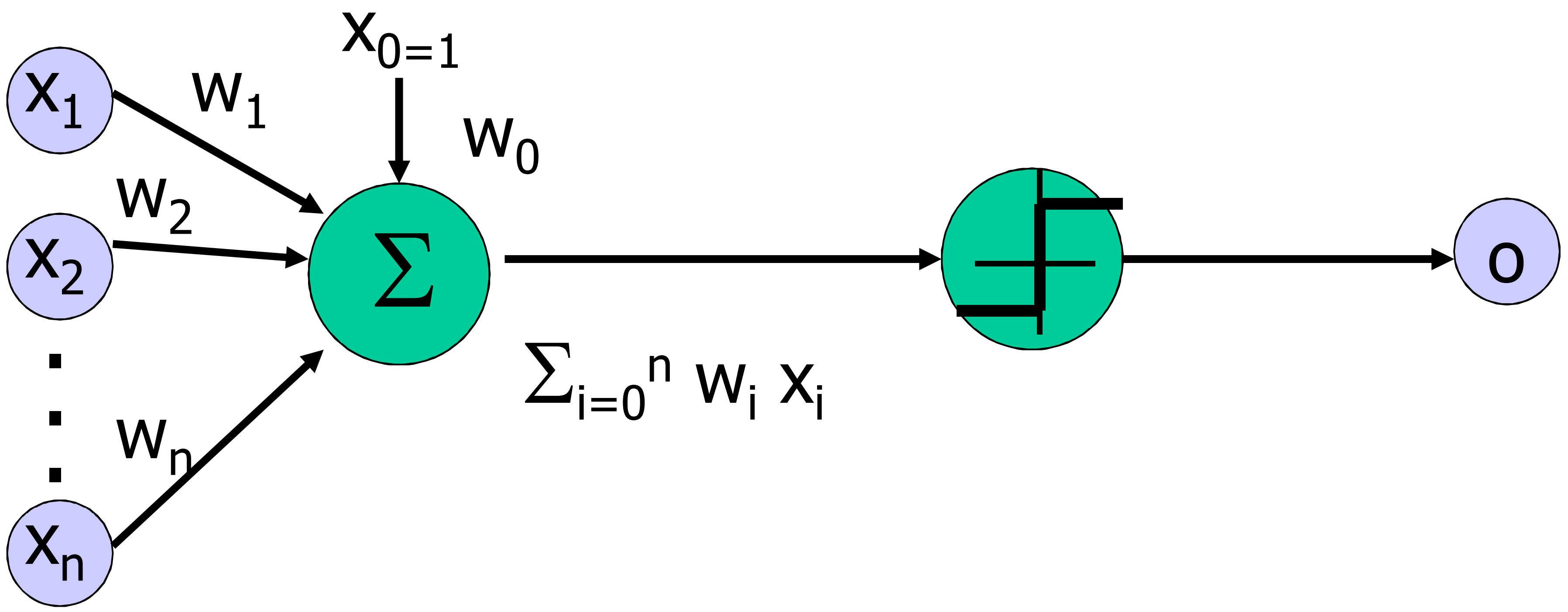
Properties of Artificial Neural Networks

- A large number of very simple, neuron-like processing elements called **units**,
- A large number of weighted, directed connections between pairs of units
  - Weights may be positive or negative real values
- Local processing in that each unit computes a function based on the outputs of a limited number of other units in the network
- Each unit computes a simple function of its input values, which are the weighted outputs from other units.
  - If there are n inputs to a unit, then the unit's output, or activation is defined by a = g((w1 * x1) + (w2 * x2) + ... + (wn * xn)).
  - Each unit computes a (simple) function g of the linear combination of its inputs.
- Learning by tuning the connection weights

Appropriate Problems for NN Learning

- Instances are represented by many attribute-value pairs.
- The target function output may be discrete-valued, real-valued, or a vector of several real-valued 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.

Perceptron



$$o(x_0, \ldots, x_n) = \begin{cases} 1 & \text{if } \Sigma_{i=0}^{n} w_i x_i > 0 \\ -1 & \text{otherwise} \end{cases}$$

Perceptron

- Perceptron is a Linear Threshold Unit (LTU).
- A perceptron takes a vector of real-valued inputs, calculates a linear combination of these inputs, then outputs 1 if the result is greater than some threshold and -1 otherwise.
- Given inputs x_l through x_n, the output $o(x_1, \dots, x_n)$ computed by the perceptron is:

$$o(x_1, \ldots, x_n) = \begin{cases} 1 \text{ if } w_0 + w_1x_1 + w_2x_2 + \cdots + w_nx_n > 0 \\ -1 \text{ otherwise} \end{cases}$$

  each w_i is a real-valued constant, or weight, that determines the contribution of input x_i to the perceptron output.
- The quantity $(-w_0)$ is a threshold that the weighted combination of inputs must surpass in order for the perceptron to output 1.
  - To simplify notation, we imagine an additional constant input $x_0 = 1$

Perceptron - Learning

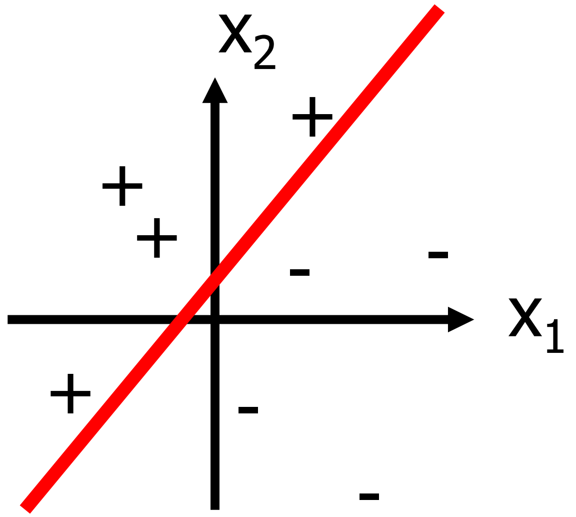
- Learning a perceptron involves choosing values for weights w_0, …,w_n.
- The space H of candidate hypotheses considered in perceptron learning is the set of all possible real-valued weight vectors

$$H = \{\vec{w} \mid \vec{w} \in \Re^{(n+1)}\}$$

Representational Power of Perceptrons

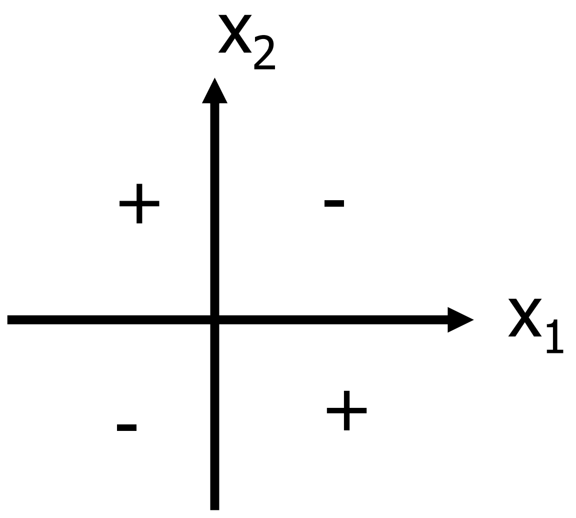
- A perceptron represents a hyperplane decision surface in the n-dimensional space of instances.
- The perceptron outputs 1 for instances lying on one side of the hyperplane and outputs -1 for instances lying on the other side.
- The equation for this decision hyperplane is $\vec{w} \cdot \vec{x} = 0$
- Some sets of positive and negative examples cannot be separated by any hyperplane. Those that can be separated are called **linearly separable** sets of examples.
- A single perceptron can be used to represent many boolean functions.
  - AND, OR, NAND, NOR are representable by a perceptron
  - XOR cannot be representable by a perceptron.

Representational Power of Perceptrons



Representable by a perceptron

XOR is not representable by a perceptron



NOT representable by a perceptron

Perceptron Training Rule

- To learn an acceptable weight vector is to begin with random weights, then iteratively apply the perceptron to each training example, modifying the perceptron weights whenever it misclassifies an example.
  - If the training example classifies correctly, weights are not updated.
- This process is repeated, iterating through the training examples as many times as needed until the perceptron classifies all training examples correctly.
  - Each pass through all of the training examples is called one **epoch**
- Weights are modified at each step according to **perceptron training rule**

Perceptron Training Rule

$$\mathbf{w_i = w_i + \Delta w_i}$$
$$\mathbf{\Delta w_i = \eta\ (t - o)\ x_i}$$

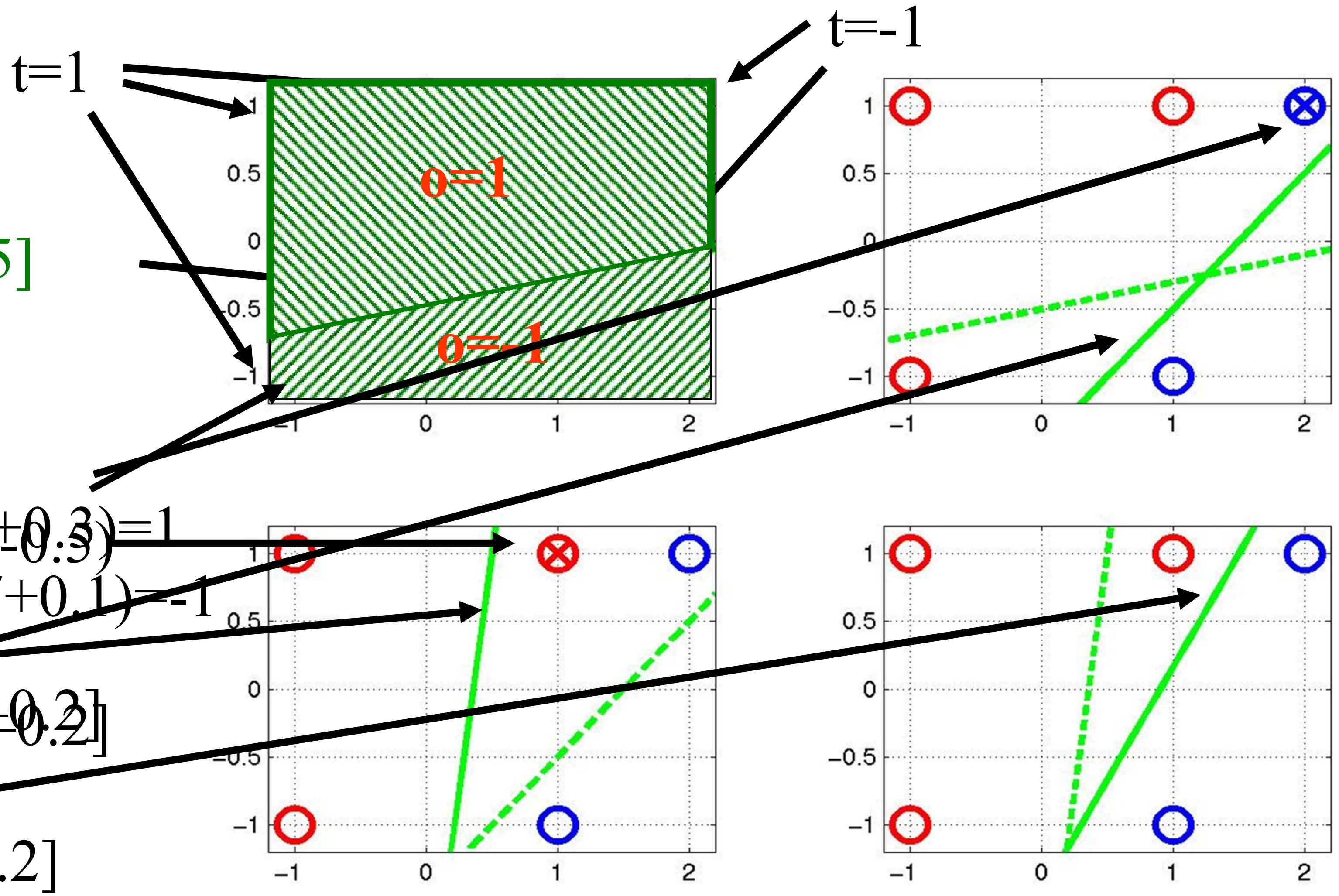
t is the target value

o is the perceptron output

η is a small constant (e.g. 0.1) called ***learning rate***

- If the output is correct (t=o) the weights w_i are not changed
- If the output is incorrect ($t \neq o$) the weights w_i are changed such that the output of the perceptron for the new weights is *closer* to t.
- The algorithm converges to the correct classification
  - if the training data is linearly separable
  - and η is sufficiently small

Perceptron Learning Rule



w=[0.25,–0.1,0.5]

$x_2 = 0.2\ x_1 - 0.5$

(x,t)=([2,1],-1)

(x,t)=([-1,-1],1)

o=sgn(0.45-0.6+0.3)=1

(x,t)=([1,1],1)

o=sgn(0.25+0.1-0.5)

=-1

o=sgn(0.25-0.7+0.1)=-1

Δw=[0.2,–0.2,–0.2]

Δw=[-0.2,–0.4,–0.2]

Δw=[0.2, 0.2, 0.2]

Perceptron Learning Rule – Learning OR

x1	x2	T	O	Δw1	w1	Δw2	w2	Δw0	w0
-	-	-	-	-	.1	-	.5	-	-.8
0	0	0	0	0	.1	0	.5	0	-.8
0	1	1	0	0	.1	.2	.7	.2	-.6
1	0	1	0	.2	.3	0	.7	.2	-.4
1	1	1	1	0	.3	0	.7	0	-.4
0	0	0	0	0	.3	0	.7	0	-.4
0	1	1	1	0	.3	0	.7	0	-.4
1	0	1	0	.2	.5	0	.7	.2	-.2
1	1	1	1	0	.5	0	.7	0	-.2
0	0	0	0	0	.5	0	.7	0	-.2
0	1	1	1	0	.5	0	.7	0	-.2
1	0	1	1	0	.5	0	.7	0	-.2
1	1	1	1	0	.5	0	.7	0	-.2

Learning rate parameter is 0.2

The result of executing the learning algorithm for 3 epochs.

Perceptron Learning Rule – Learning AND

x1	x2	T	O	Δw1	w1	Δw2	w2	Δw0	w0
-	-	-	-	-	.1	-	.5	-	-.8
0	0	0	0	0	.1	0	.5	0	-.8
0	1	0	0	0	.1	0	.5	0	-.8
1	0	0	0	0	.1	0	.5	0	-.8
1	**1**	**1**	**0**	**.2**	**.3**	**.2**	**.7**	**.2**	**-.6**
0	0	0	0	0	.3	0	.7	0	-.6
0	**1**	**0**	**1**	**0**	**.3**	**-.2**	**.5**	**-.2**	**-.8**
1	0	0	0	0	.3	0	.5	0	-.8
1	**1**	**1**	**0**	**.2**	**.5**	**.2**	**.7**	**.2**	**-.6**
0	0	0	0	0	.5	0	.7	0	-.6
0	**1**	**0**	**1**	**0**	**.5**	**-.2**	**.5**	**-.2**	**-.8**
1	0	0	0	0	.5	0	.5	0	-.8
1	1	1	1	0	.5	0	.5	0	-.8

Learning rate parameter is 0.2

The result of executing the learning algorithm for 4 epochs.

1 epcoch not shown in the table.

Gradient Descent and the Delta Rule

- The **perceptron rule** finds a successful weight vector when the training examples are linearly separable, it can fail to converge if the examples are not linearly separable.
- The **delta rule** overcomes this difficulty.
- If the training examples are not linearly separable, the delta rule converges toward a best-fit approximation to the target concept.
- The key idea behind the delta rule is to use **gradient descent** to search the hypothesis space of possible weight vectors to find the weights that best fit the training examples.
- The delta rule is important because gradient descent provides the basis for the BACKPROPAGATION Algorithm, which can learn networks with many interconnected units.
  - The gradient descent can serve as the basis for learning algorithms that must search through hypothesis spaces containing many different types of continuously parameterized hypotheses

Gradient Descent

- Consider linear unit without threshold and continuous output o (not just –1,1)

  $o = w_0 + w_1 x_1 + \ldots + w_n x_n$

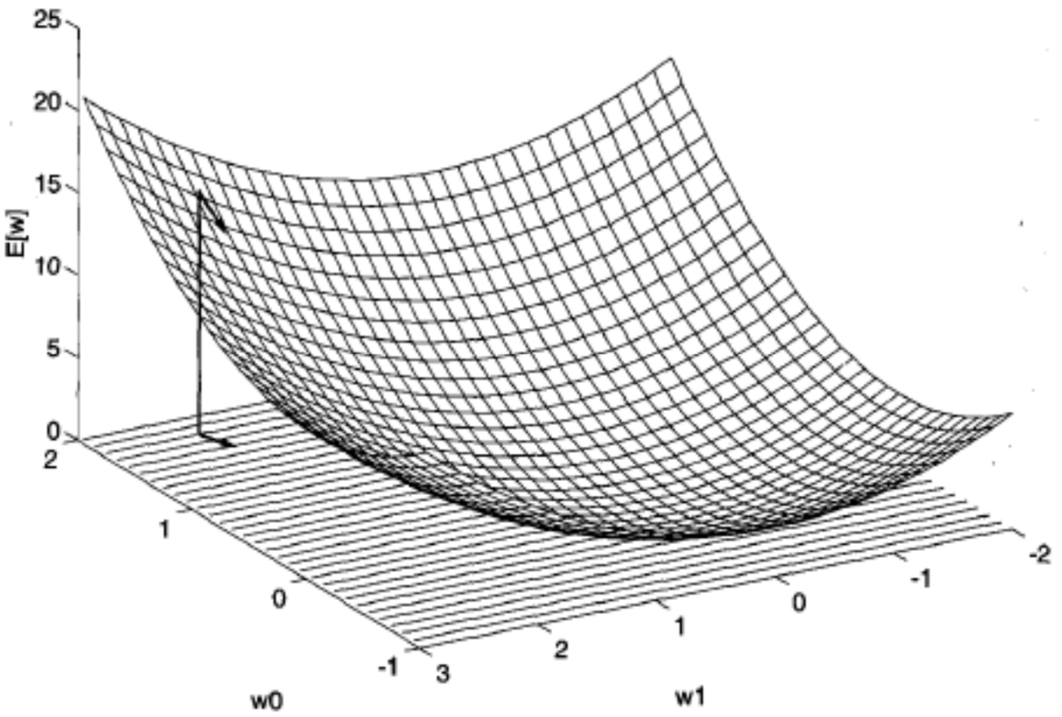
- Train the w_i's such that they minimize the squared error

  $E[w_0,\ldots,w_n] = \frac{1}{2} \Sigma_{d \in D} (t_d - o_d)^2$

  Here (cost/loss ftn): SSE.
  Other: MAE, MSE, Cross Entropy, etc.

  where D is the set of training examples t_d is the target output for training example d, and o_d is the output of the linear unit for training example d.

Gradient Descent



• The ***wo, wl*** plane represents the entire hypothesis space.
• The vertical axis indicates the error E relative to some fixed set of training examples.
• The error surface summarizes the desirability of every weight vector in the hypothesis space (we desire a hypothesis with minimum error).
• The arrow shows the negated ***gradient*** at one particular point, indicating the direction in the wo, wl plane producing steepest descent along the error surface.

• Gradient descent search determines a weight vector that minimizes ***E*** by starting with an arbitrary initial weight vector, then repeatedly modifying it in small steps.
• At each step, the weight vector is altered in the direction that produces the steepest descent along the error surface.
• This process continues until the global minimum error is reached.

Gradient Descent

- How can we calculate the direction of steepest descent along the error surface?
- This direction can be found by computing the derivative of ***E*** with respect to each component of the vector w0,..,wn.
- This vector derivative is called the ***gradient*** of ***E*** with respect to w1,..,wn**,** written ∇E(w0,…,wn)
- **Gradient:**

  ∇E[w0,…,wn] = [∂E/∂w0,… ∂E/∂wn]
- When the gradient is interpreted as a vector in weight space, the gradient specifies the direction that produces the steepest increase in ***E*****.**
- The negative of this vector (-∇E[w0,…,wn]) gives the direction of steepest decrease.

Training Rule for Gradient Descent

- For each weight wi
  - **$wi = wi + \Delta wi$**
  - **$\Delta wi = - \eta$** $\nabla E[wi]$
- η is a small constant called ***learning rate***

$$\Delta wi = - \eta \nabla E[wi]$$
$$= - \eta (\partial E/\partial wi)$$

.

.

$$\begin{aligned}\partial E/\partial wi &= \partial/\partial wi \ \tfrac{1}{2} \Sigma_{d\in D} (t_d - o_d)^2 \\ &= \tfrac{1}{2} \Sigma_{d\in D} \partial/\partial wi \ (t_d - o_d)^2 \\ &= \tfrac{1}{2} \Sigma_{d\in D} 2(t_d - o_d) \ \partial/\partial wi \ (t_d - o_d) \\ &= \Sigma_{d\in D} (t_d - o_d) \ \partial/\partial wi \ (t_d - (w_0 x_{0d} + \ldots + w_n x_{nd}))\end{aligned}$$

$$\boldsymbol{\partial E/\partial wi = \Sigma_{d\in D} (t_d - o_d) (-x_{id})}$$

$$\boldsymbol{\Delta wi = - \eta \Sigma_{d\in D} (t_d - o_d) (-x_{id})}$$

Gradient Descent

```
Gradient-Descent(training_examples, $\eta$)
    Each training example is a pair of the form <$(x_1, \dots x_n)$,t> where $(x_1, \dots, x_n)$ is the vector
    of input values, and t is the target output value, $\eta$ is the learning rate (e.g. 0.1)
•   Initialize each $w_i$ to some small random value
•   Until the termination condition is met, Do
        –  Initialize each $\Delta w_i$ to zero
        –  For each <$(x_1, \dots x_n)$,t> in training_examples, Do
               •  Input the instance $(x_1, \dots, x_n)$ to the linear unit and compute the output o
               •  For each linear unit weight $w_i$ Do
                      –  $\Delta w_i = \Delta w_i + \eta\,(t-o)\,x_i$
        –  For each linear unit weight wi, Do
               •  $w_i = w_i + \Delta w_i$
```

Incremental (Stochastic) Gradient Descent

Gradient descent is a strategy for searching through a large or infinite hypothesis space that can be applied whenever

- the hypothesis space contains continuously parameterized hypotheses (e.g., the weights in a linear unit), and
- the error can be differentiated with respect to these hypothesis parameters.

The key practical difficulties in applying gradient descent are

- converging to a local minimum can sometimes be quite slow (i.e.,it can require many thousands of gradient descent steps), and
- if there are multiple local minima in the error surface, then there is no guarantee that the procedure will find the global minimum.

Incremental (stochastic) gradient descent tries to solve these problems.

Incremental (Stochastic) Gradient Descent

- Batch mode : gradient descent

  $w=w - \eta \nabla E_D[w]$ over the entire data D

  $E_D[w]=1/2\Sigma_d(t_d-o_d)^2$

- Incremental mode: gradient descent

  $w=w - \eta \nabla E_d[w]$ over individual training examples d

  $E_d[w]=1/2 (t_d-o_d)^2$

Incremental Gradient Descent can approximate Batch Gradient Descent arbitrarily closely if η is small enough

Incremental (Stochastic) Gradient Descent

```
Incremental (Stochastic) Gradient-Descent(*training_examples*, $\eta$)
    *Each training example is a pair of the form $<(x_1,...x_n),t>$ where $(x_1,...,x_n)$ is the vector
    of input values, and t is the target output value, $\eta$ is the learning rate (e.g. 0.1)*
•   Initialize each $w_i$ to some small random value
•   Until the termination condition is met, Do
        –  ~~Initialize each $\Delta w_i$ to zero~~
        –  For each $<(x_1,\ldots x_n),t>$ in *training_examples*, Do
              •  Input the instance $(x_1,\ldots,x_n)$ to the linear unit and compute the output o
              •  For each linear unit weight $w_i$ Do
                   ~~– $\Delta w_i = \Delta w_i + \eta (t-o) x_i$~~        $w_i = w_i + \eta\ (t\text{-}o)\ x_i$
        –  ~~For each linear unit weight wi, Do~~
              ~~•  $w_i = w_i + \Delta w_i$~~
```

Incremental (Stochastic) Gradient Descent

- In standard gradient descent, the error is summed over all examples before updating weights, whereas in stochastic gradient descent weights are updated upon examining each training example.
- Summing over multiple examples in standard gradient descent requires more computation per weight update step. On the other hand, because it uses the true gradient, standard gradient descent is often used with a larger step size per weight update than stochastic gradient descent.
- In cases where there are multiple local minima with respect to E(w0,…,wn), stochastic gradient descent can sometimes avoid falling into these local minima because it uses the various ∇E_d(w0,…,wn) rather than ∇E (w0,…,wn) to guide its search.

Comparison Perceptron and Gradient Descent Rule

Perceptron learning rule guaranteed to succeed if

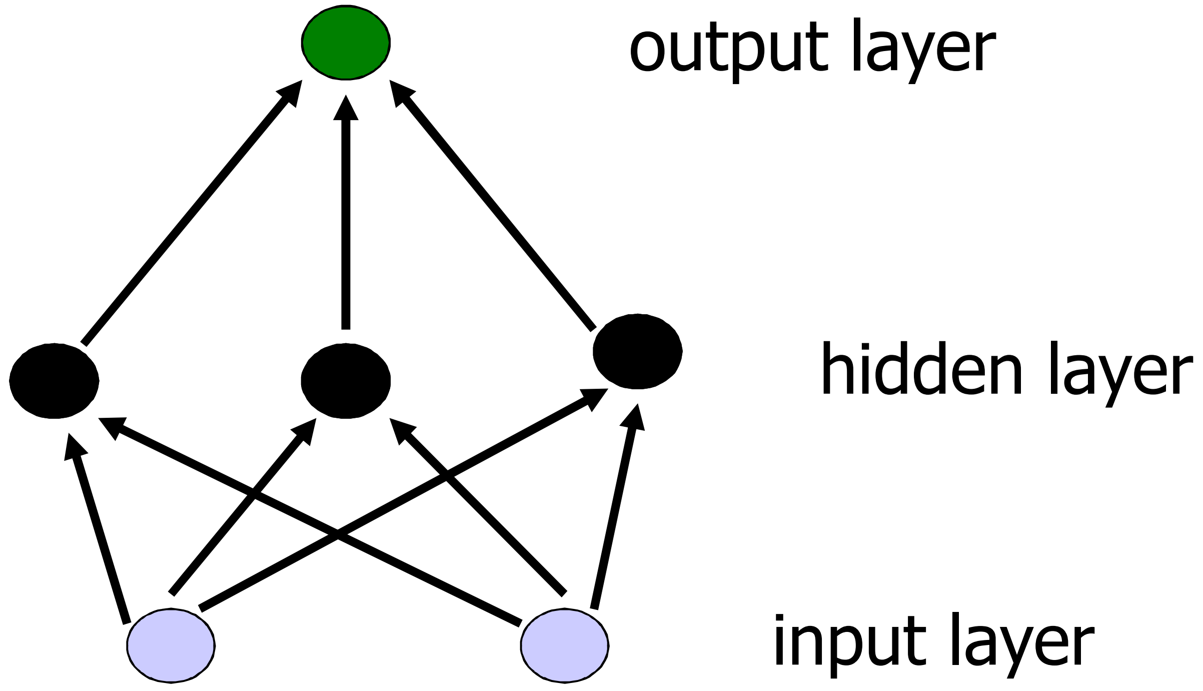
- Training examples are linearly separable
- Sufficiently small learning rate η

Linear unit training rules uses gradient descent

- Guaranteed to converge to hypothesis with minimum squared error
- Given sufficiently small learning rate η
- Even when training data contains noise
- Even when training data not separable by H

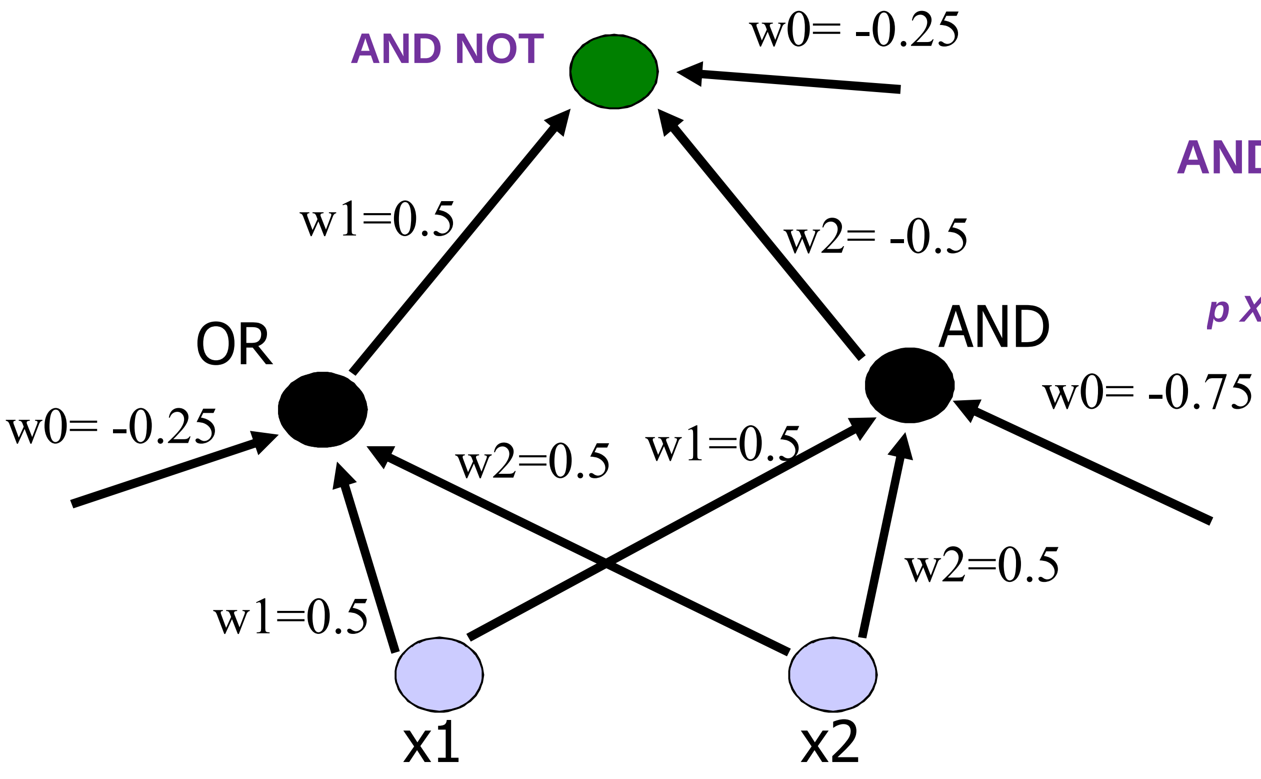
Multi-Layer Networks

- Single perceptrons can only express linear decision surfaces.
- Multilayer networks are capable of expressing a rich variety of nonlinear decision surfaces.



Multi-Layer Networks with Linear Units
Ex. XOR

- Multiple layers of cascaded linear units still produce only linear functions.



OR: $0.5*x1 + 0.5*x2 - 0.25 \quad > 0$

AND: $0.5*x1 + 0.5*x2 - 0.75 \quad > 0$

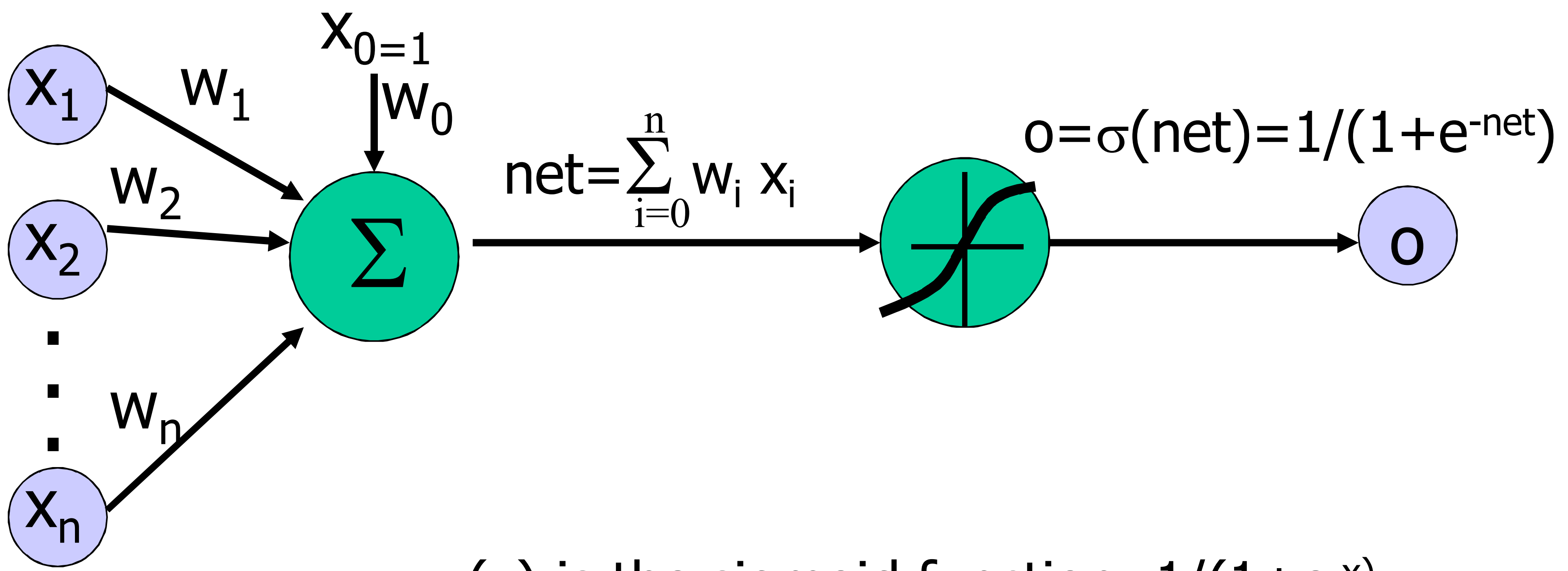
AND NOT
~~XOR~~: $0.5*x1 - 0.5*x2 - 0.25 \quad > 0$

p XOR q = (p OR q) AND NOT (p AND q)

Multi-Layer Networks with Non-Linear Units

- Multiple layers of cascaded linear units still produce only linear functions.
- We prefer networks capable of representing highly nonlinear functions.
- What we need is a unit whose output is a nonlinear function of its inputs, but whose output is also a differentiable function of its inputs.
- One solution is the **sigmoid unit**, a unit very much like a perceptron, but based on a smoothed, differentiable threshold function.

Sigmoid Unit



x_1
x_2
x_n
w_1
w_2
w_n
$x_{0=1}$
w_0
Σ
$net = \sum_{i=0}^{n} w_i x_i$
$o = \sigma(net) = 1/(1+e^{-net})$
o

$\sigma(x)$ is the sigmoid function: $1/(1+e^{-x})$

$d\sigma(x)/dx = \sigma(x)\,(1-\sigma(x))$ ← differentiable function

Sigmoid Unit

Derive gradient descent rules to train:

- one sigmoid function

  $$\partial E/\partial w_i = -\Sigma_d (t_d - o_d)\, o_d\, (1 - o_d)\, x_{id}$$

- Multilayer networks of sigmoid units $\rightarrow$ backpropagation

MLP Backpropagation Algorithm

- Create a feed-forward network with n_i inputs, n_{hidden} hidden units, and n_{out} output units.
- Initialize each w_i to some small random value (e.g., between -.05 and .05).
- Until the termination condition is met, Do
  - For each training example $<(x_1,\ldots x_n),t>$, Do

    // Propagate the input forward through the network:

    1. Input the instance $(x_1,\ldots,x_n)$ to the network and compute the network outputs o_k for every unit

    // Propagate the errors backward through the network:

    2. For each output unit k, calculate its error term δ_k

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

    3. For each hidden unit h, calculate its error term δ_h

       $\delta_h=o_h(1-o_h)\,\Sigma_k\, w_{h,k}\,\delta_k$

    4. For each network weight $w_{i,j}$, Do

       $w_{i,j}=w_{i,j}+\Delta w_{i,j}$ where $\Delta w_{i,j}= \eta\, \delta_j\, \overset{Oi}{\mathbf{Oi}\,x_{i,j}}$

Read the detail of derivatives in the textbook (T. Mitchell) 4.5.3

Stochastic gradient descent version of the Backpropagation Algorithm for Feedforward networks (hidden and output) Sigmoid units

Lab: Implementation in Python

For testing if the output is > 0.5*(bcz of Sigmoid ftn)*, then return 1, else return 0.

In case of batch gradient descent the change in weight for the total batch will be added. Note, here, the cost ftn is SSE. In case of MSE the mean of the change in weight for the total batch will be added.

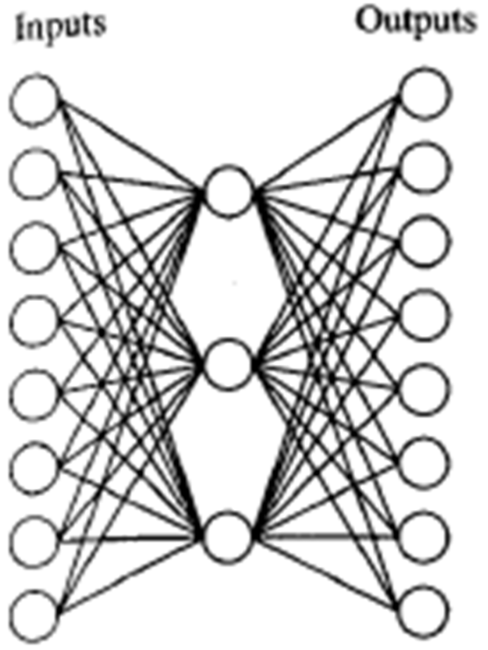
Backpropagation

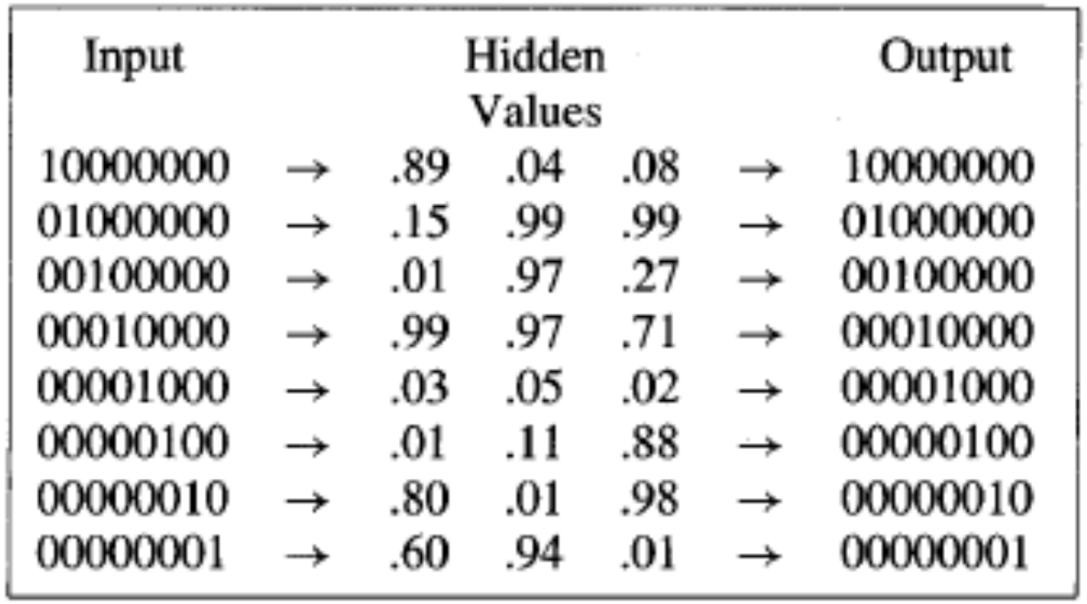
- Gradient descent over entire *network* weight vector
- Easily generalized to arbitrary directed graphs
- Will find a local, not necessarily global error minimum

  -in practice often works well (can be invoked multiple times with different initial weights)
- Often include weight *momentum* term

  $$\Delta w_{i,j}(n) = \eta\ \delta_j\ x_{i,j} + \alpha\ \Delta w_{i,j}\ (n-1)$$
- Minimizes error training examples
- Training can be slow typical 1000-10000 iterations
- Using network after training is fast

8-3-8 Binary Encoder -Decoder

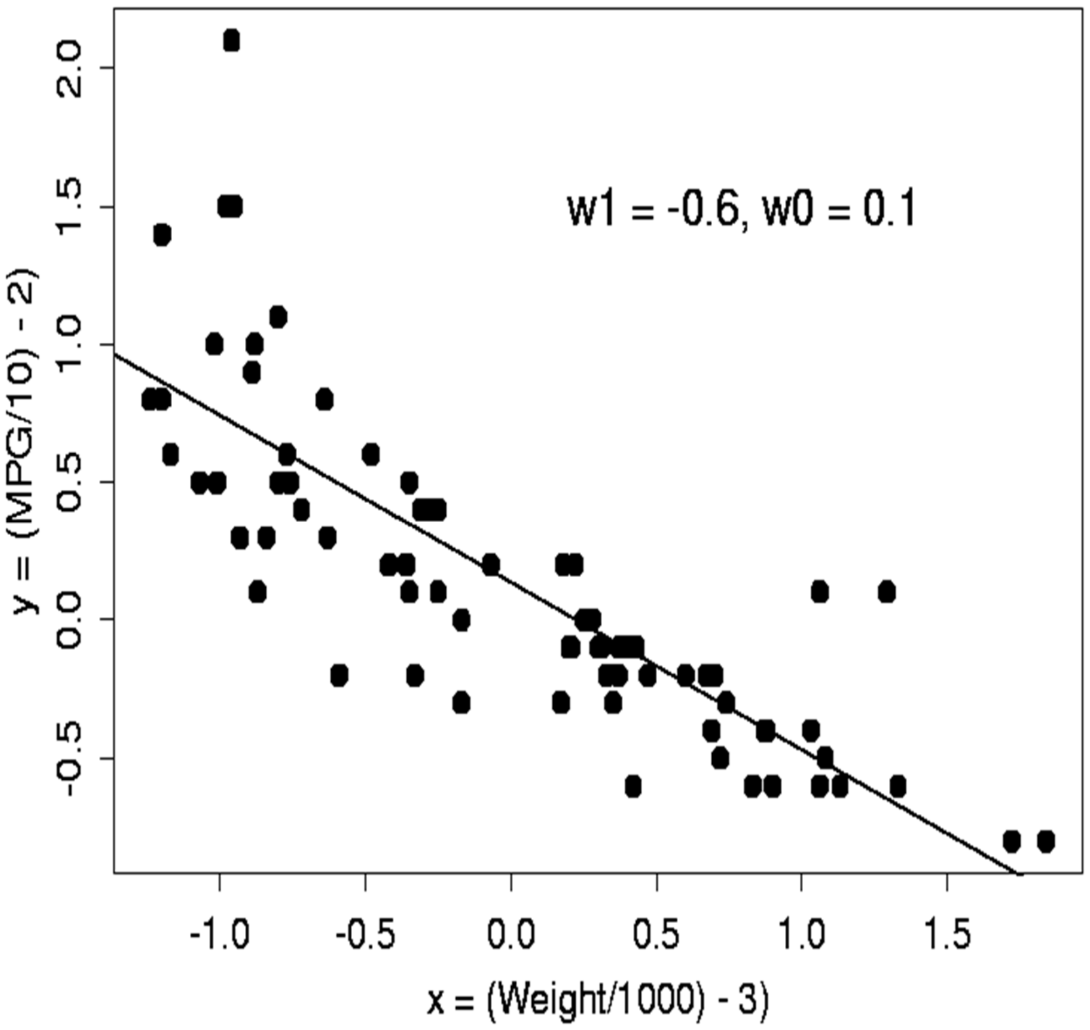




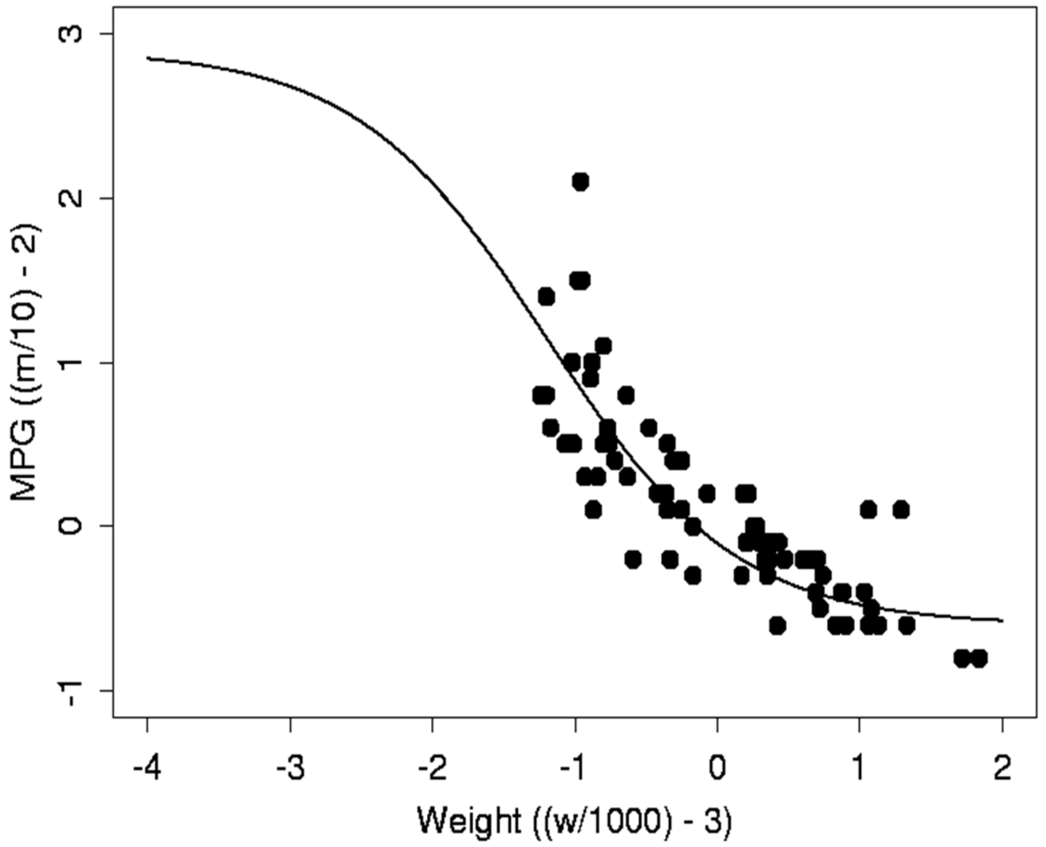
Input		Hidden Values				Output
10000000	→	.89	.04	.08	→	10000000
01000000	→	.15	.99	.99	→	01000000
00100000	→	.01	.97	.27	→	00100000
00010000	→	.99	.97	.71	→	00010000
00001000	→	.03	.05	.02	→	00001000
00000100	→	.01	.11	.88	→	00000100
00000010	→	.80	.01	.98	→	00000010
00000001	→	.60	.94	.01	→	00000001

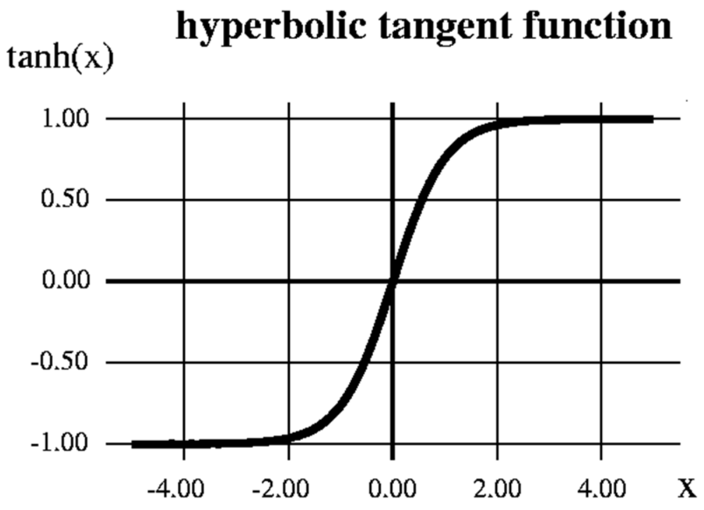
- 8 x 3 x 8 network was trained to learn the identity function, using the eight training examples shown.
- After 5000 training epochs, the three hidden unit values encode the eight distinct inputs using the encoding shown on the right.
- If the encoded values are rounded to zero or one, the result is the standard binary encoding for eight distinct values.

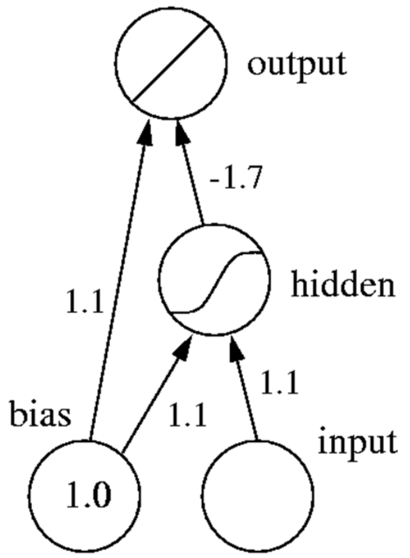
Single Layer – Linear Problem



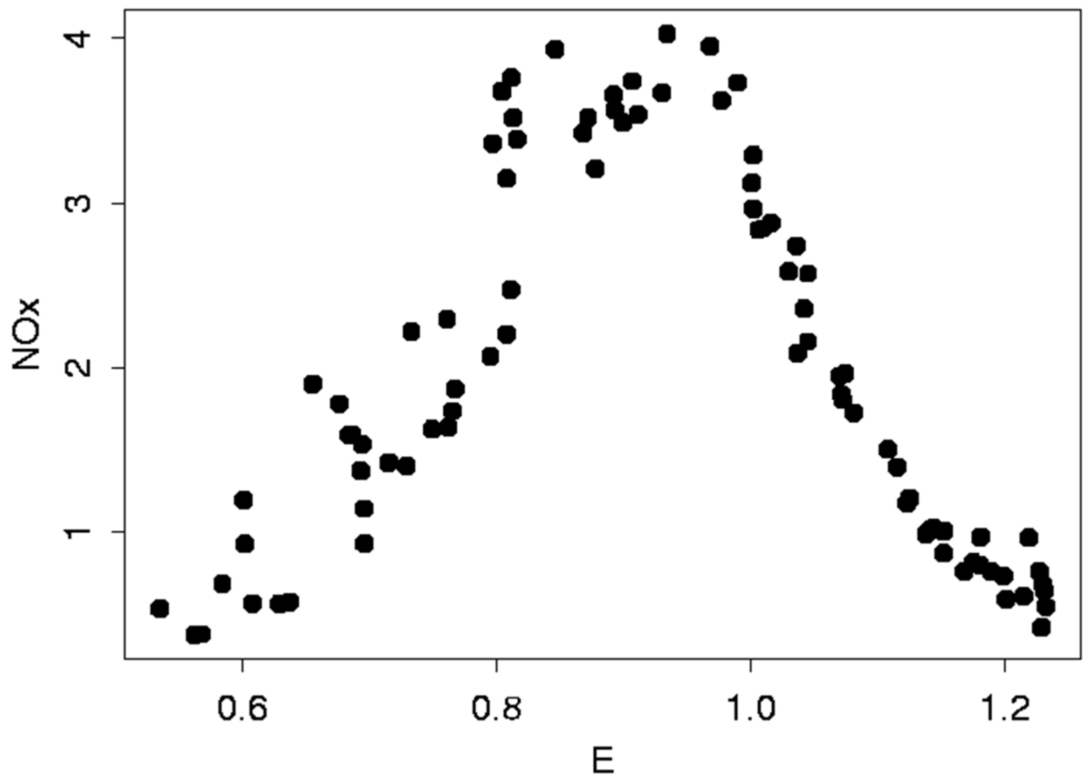
Multi-Layer Network – NonLinear Problem

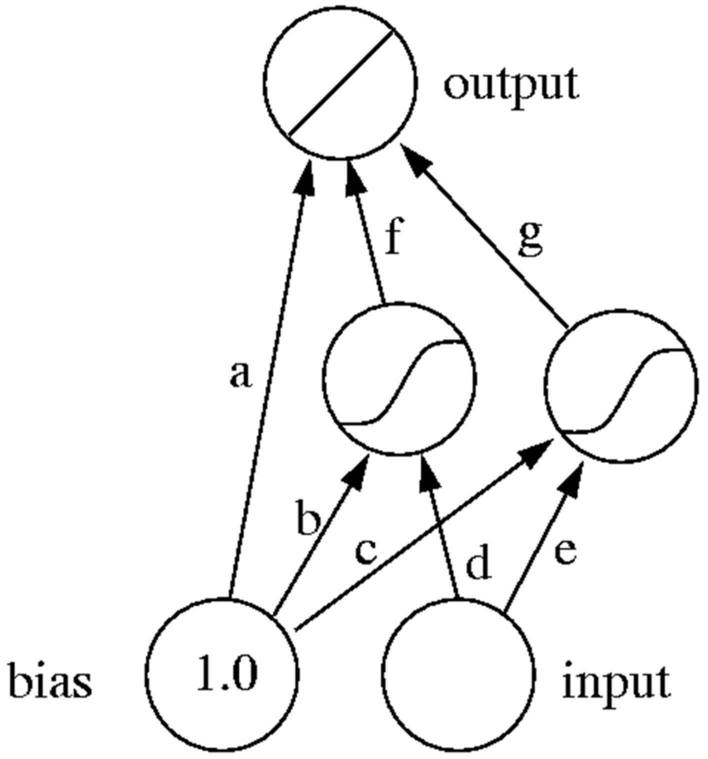






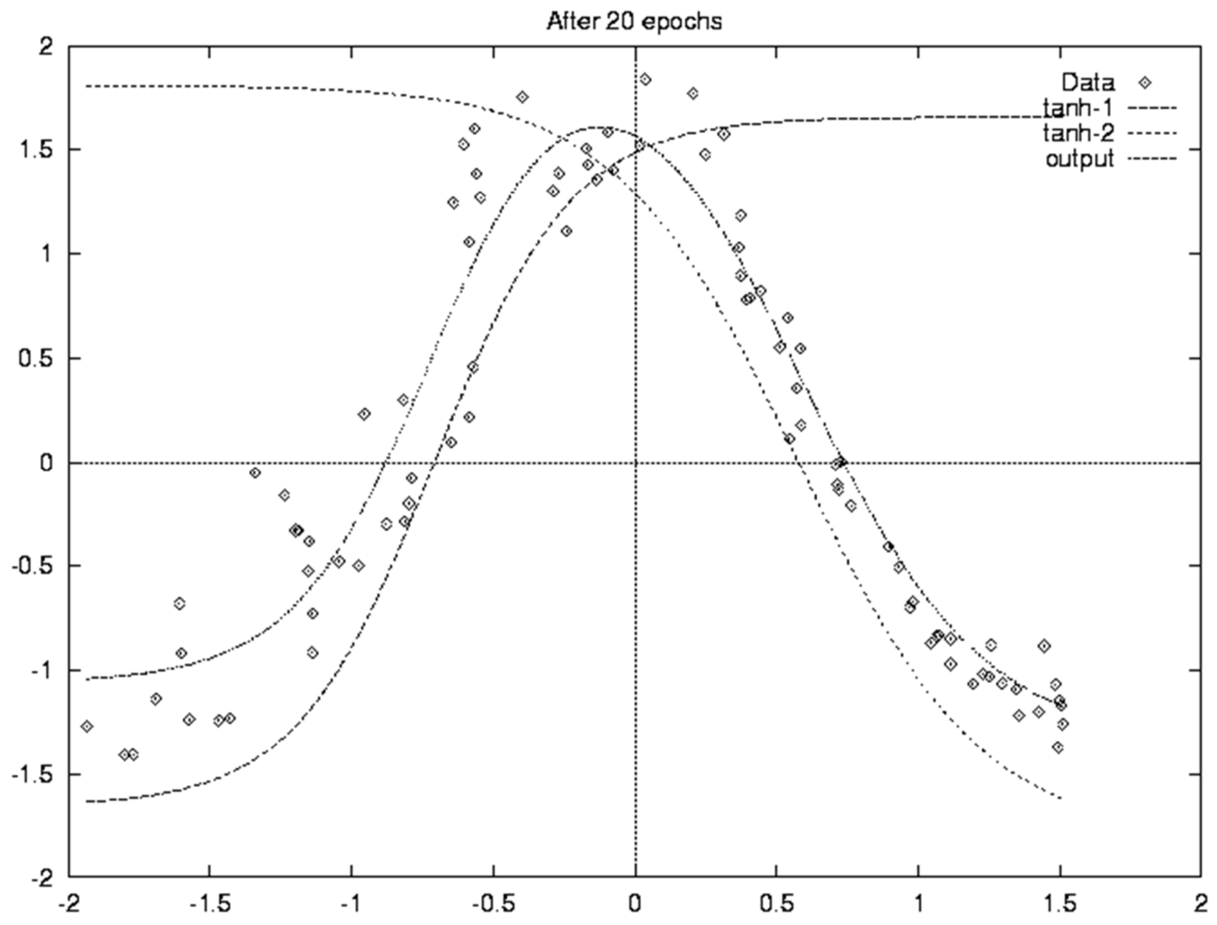
Multi-Layer Network – NonLinear Problem2

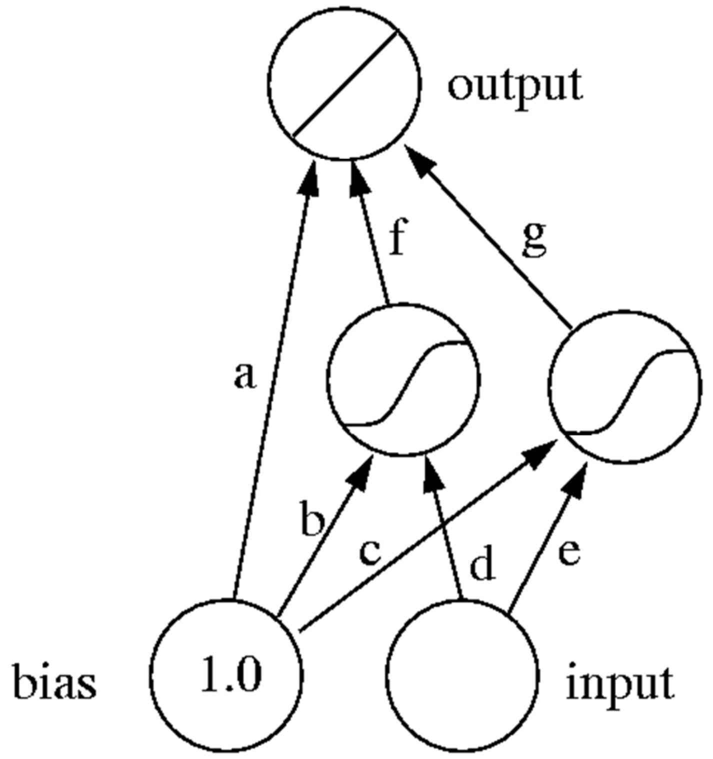




output = f tanh(x) + g tanh2(x) +a
tanh1(x) = tanh(d*x + b)
tanh2(x) = tanh(e*x + c)

Multi-Layer Network – NonLinear Problem2





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
- Any function can be approximated to arbitrary accuracy by a network with two hidden layers