UTSA
The University of Texas at San Antonio™

IS 6733: Deep Learning on Cloud Platforms

01 Artificial Neural Networks

Copy Right Notice

- **Most slides in this presentation are adopted from slides of text book and various sources. The Copyright belong to the original authors. Thanks!**

Why Neural Network

- **Some tasks can be done easily by humans but are hard by conventional paradigms on Von Neumann machine with algorithmic approach**
    - Pattern recognition (old friends, hand-written characters)
    - Content addressable recall
    - Approximate, common sense reasoning (driving, playing piano, baseball player)
- **These tasks are often experience based, hard to apply logic.**

Biological Motivation

- **Humans:**
  - Neuron switching time ~0.001 second
  - Number of neurons ~10^{10}
  - Connections per neuron ~ 10^{4-5}
  - Scene recognition time ~0.1 second
  - Highly parallel computation process.
- **Biological Learning Systems are built of very complex webs of interconnected neurons.**
- **Information-Processing abilities of biological neural systems must follow from highly parallel processes operating on representations that are distributed over many neurons**

What is an neural network

- **A set of nodes (units, neurons, processing elements)**
  - Each node has input and output
  - Each node performs a simple computation by its **node function**
- **Weighted connections between nodes**
  - Connectivity gives the structure/architecture of the net
  - What can be computed by a NN is primarily determined by the connections and their weights
- **A very much simplified version of networks of neurons in animal nerve systems**

ANN vs. Bio NN

ANN

- **Nodes**
  - input
  - output
  - node function
- **Connections**
  - connection strength

Bio NN

- **Cell body**
  - signal from other neurons
  - firing frequency
  - firing mechanism
- **Synapses**
  - synaptic strength

Properties of artificial neural nets

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

When to Consider Neural Networks

- **Input is high-dimensional discrete or real-valued**
- **Output is discrete or real valued**
- **Output is a vector of values**
- **Possibly noisy data**
- **Form of target function is unknown**
- **Human readability of result is unimportant**
- **Examples:**
  - Speech phoneme recognition
  - Image classification
  - Financial prediction

History of Neural Networks

- 1943: McCulloch and Pitts proposed a model of a neuron --> Perceptron
- 1960s: Widrow and Hoff explored Perceptron networks (which they called “Adelines”) and the delta rule.
- 1962: Rosenblatt proved the convergence of the perceptron training rule.
- 1969: Minsky and Papert showed that the Perceptron cannot deal with nonlinearly-separable data sets---even those that represent simple function such as X-OR.
- 1970-1985: Very little research on Neural Nets
- 1986: Invention of Backpropagation [Rumelhart and McClelland, but also Parker and earlier on: Werbos] which can learn from nonlinearly-separable data sets.
- Since 1985: A lot of research in Neural Nets!

A Perceptron (a neuron)

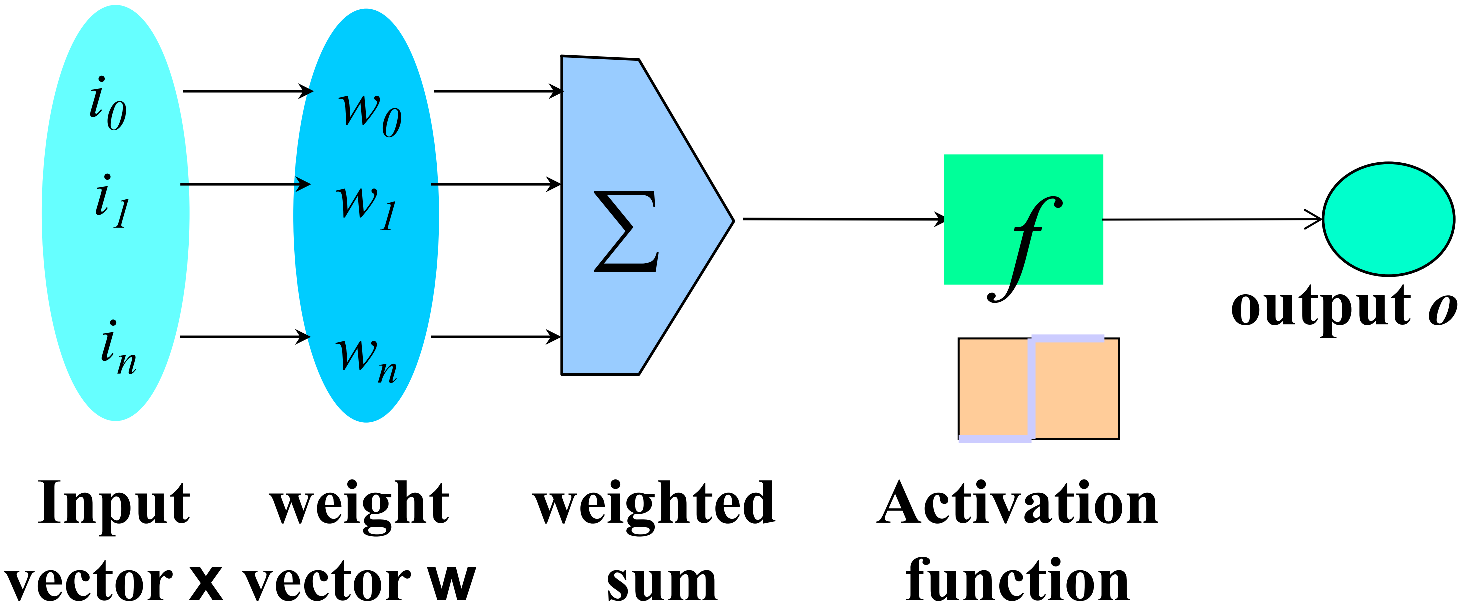
- **The network**
  - Input vector $\boldsymbol{i_j}$ (including threshold input = 1)
  - Weight vector $\boldsymbol{w} = (w_0, w_1, \ldots, w_n)$

$$net = w \cdot i_j = \sum_{k=0}^{n} w_k i_{k,j}$$

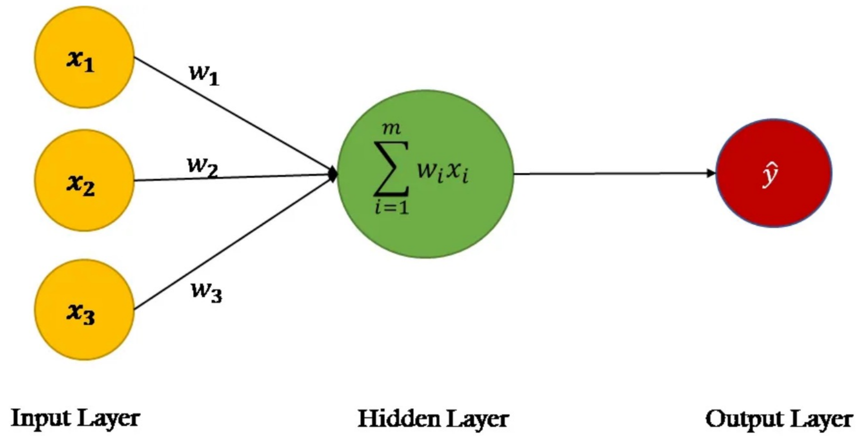
  - Output: bipolar (-1, 1) using the sign node function

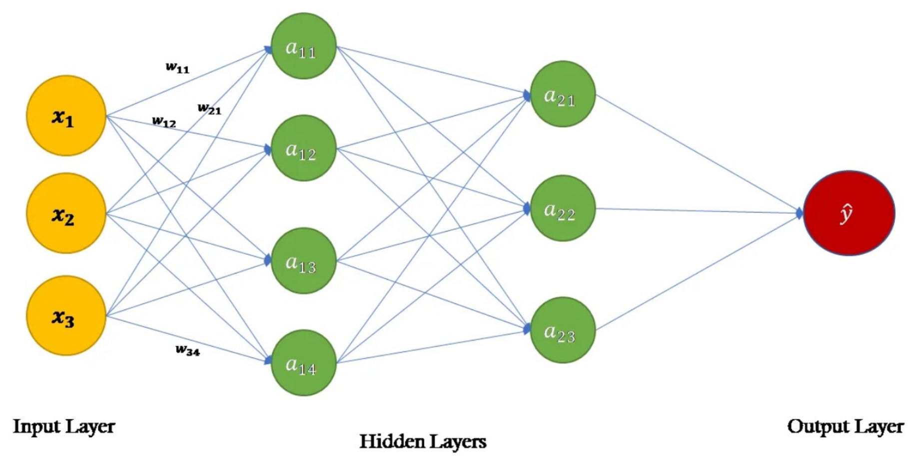
$$output = \begin{cases} 1 & \text{if } w \cdot i_j > 0 \\ -1 & \text{otherwise} \end{cases}$$

- **Training samples**
  - Pairs ($\boldsymbol{i_j}$, *class*($\boldsymbol{i_j}$)) where *class*($\boldsymbol{i_j}$) is the correct classification of $\boldsymbol{i_j}$



Aside: Multilayer Perceptron

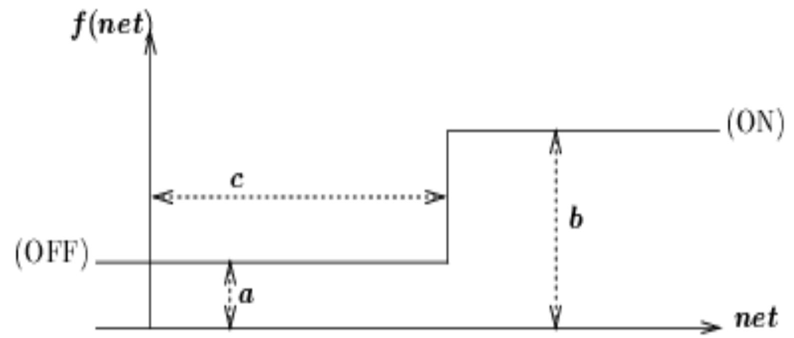




Activation functions

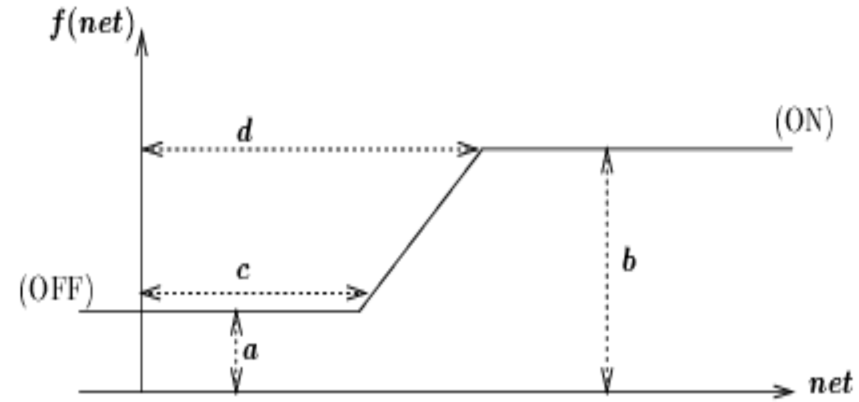
- **Step (threshold) function**

$$f(\text{net}) = \begin{cases} a & \text{if net} < c \\ b & \text{if net} > c \end{cases}$$



- **Ramp function**

$$f(\text{net}) = \begin{cases} a & \text{if net} \leq c \\ b & \text{if net} \geq d \\ a + \frac{(\text{net}-c)(b-a)}{(d-c)} & \text{otherwise} \end{cases}$$



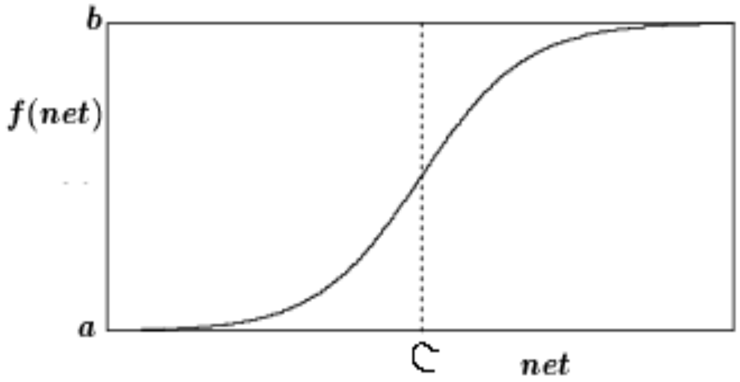
More: https://360digitmg.com/blog/activation-functions-neural-networks

Activation functions

- **Sigmoid function**
  - S-shaped
  - Continuous and everywhere differentiable
  - Rotationally symmetric about some point (*net* = *c*)
  - Asymptotically approaches saturation points

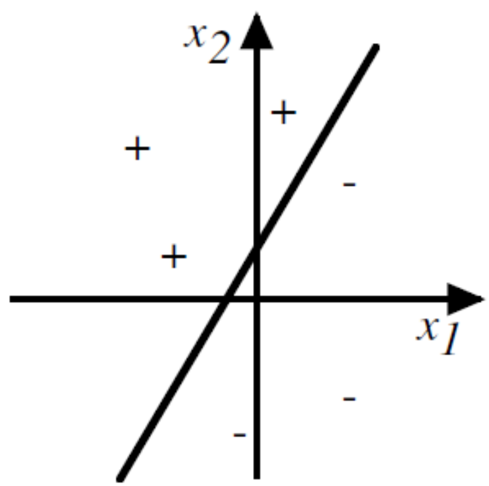
$$f(\text{net}) = z + \frac{1}{1 + \exp(-x \cdot \text{net} + y)}$$

$$f(\text{net}) = \tanh(x \cdot \text{net} - y) + z,$$



Decision Surface of a Perceptron: Linear separability

- ***n* dimensional patterns ($x_1, \ldots, x_n$)**
  - Hyperplane $w_0 + w_1 x_1 + w_2 x_2 + \ldots + w_n x_n = 0$ dividing the space into two regions
- **Can we get the weights from a set of sample patterns?**
  - If the problem is linearly separable, then YES (by perceptron learning)



Examples of linearly separable classes

- **Logical AND function**

  patterns (bipolar) decision boundary

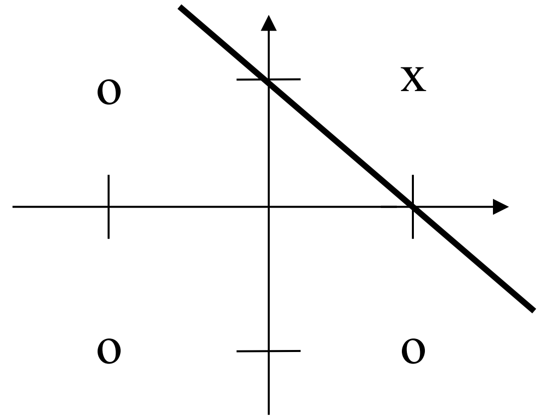
x1	x2	output
-1	-1	-1
-1	1	-1
1	-1	-1
1	1	1

$w1 = 1$

$w2 = 1$

$w0 = -1$

$-1 + x1 + x2 = 0$



x: class I (output = 1)
o: class II (output = -1)

- **Logical OR function**

  patterns (bipolar) decision boundary

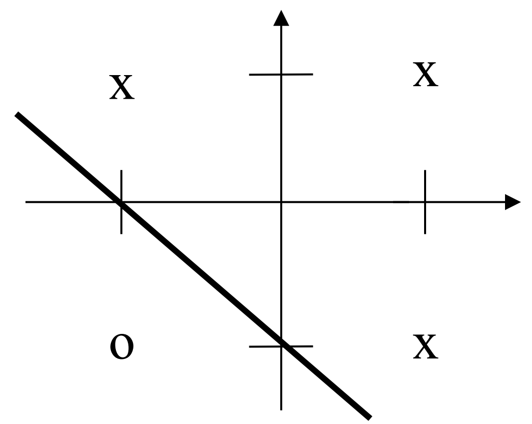
x1	x2	output
-1	-1	-1
-1	1	1
1	-1	1
1	1	1

$w1 = 1$

$w2 = 1$

$w0 = 1$

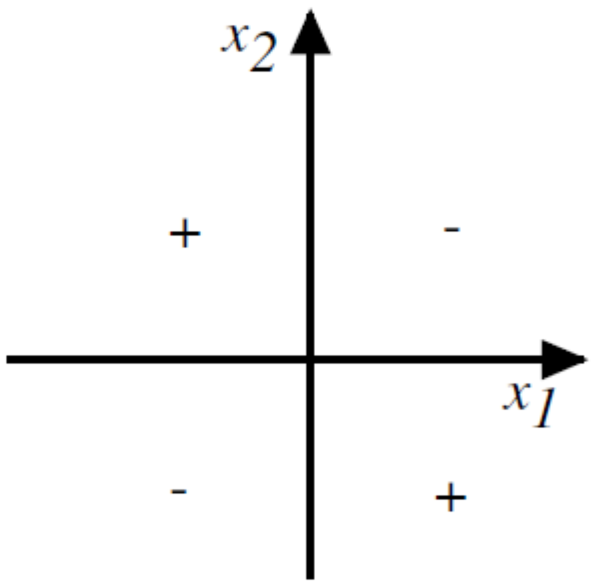
$1 + x1 + x2 = 0$



x: class I (output = 1)
o: class II (output = -1)

Functions not representable

- **Some functions are not representable by perceptron**
  - Not linearly separable



Perceptron Training Rule

- **Training:**
  - Update ***w*** so that all sample inputs are correctly classified (if possible)
  - If an input $\boldsymbol{i_j}$ is misclassified by the current ***w***
    - $class(\boldsymbol{i_j}) \cdot \boldsymbol{w} \cdot \boldsymbol{i_j} < 0$
    - change ***w*** to $\boldsymbol{w} + \Delta\boldsymbol{w}$ so that $(\boldsymbol{w} + \Delta\boldsymbol{w}) \cdot \boldsymbol{i_j}$ is closer to $class(\boldsymbol{i_j})$
- **Perceptron Training Rule**

$$w_i = w_i + \Delta w_i$$

  - Where

$$\Delta w_i = \eta(t - o)x_i$$

  - Where
    - $t = c(\vec{x})$ is the target value
    - o is perceptron output
    - η is a small positive constant, called learning rate

Perceptron Training Algorithm

- **Start with a randomly chosen weight vector w_0**
- **Let k=1;**
- **While some input vectors remain misclassified , do**
  - Let x_j be a misclassified input vector
  - Update the weight vector to $w_k = w_{k-1} + \eta(t-o)x_k$
  - Increment k;
- **End while**

Perceptron Training Rule

- **It will converge if**
  - Training data is linearly separable
  - η is a sufficiently small
- ***Theorem*: If there is a w^* such that $f(i_p \cdot w^*) = class(i_p)$ for all *P* training sample patterns $\{i_p, class(i_p)\}$, then for any start weight vector w^0, the perceptron learning rule will converge to a weight vector w^+ such that for all *p***

$$f(i_p \cdot w^+) = class(i_p)$$

(w^* and w^+ may not be the same.)

Perceptron Training Rule

- **Justification**

$$(w+\eta\cdot(t-o)\cdot x_k)\cdot x_k = w\cdot x_k + \eta\cdot(t-o)\cdot x_k\cdot x_k$$

$$\textit{then}$$

$$(w+\eta\cdot(t-o)\cdot x_k)\cdot x_k - w\cdot x_k = \eta\cdot(t-o)\cdot x_k\cdot x_k$$

$$\text{since } x_k\cdot x_k > 0$$

$$\begin{cases} > 0 & \text{if } class(i_j) = 1 \\ < 0 & \text{if } class(i_j) = -1 \end{cases}$$

$$\Rightarrow \text{new } net \text{ moves toward } class(i_j)$$

Perceptron Training Rule

- **Termination criteria: learning stops when all samples are correctly classified**
  - Assuming the problem is linearly separable
  - Assuming the learning rate (η) is sufficiently small
- **Choice of learning rate:**
  - If η is too large: existing weights are overtaken by Δw
  - If η is too small (≈ 0): very slow to converge
  - Common choice: $0.1 < \eta < 1$.

Example, perceptron learning function AND

- **Training samples**

	in_0	in_1	in_2	d
p0	1	-1	-1	-1
p1	1	-1	1	-1
p2	1	1	-1	-1
p3	1	1	1	1

- **Initial weights W(0)**

w0	w1	w2
1	1	-1

- **Learning rate = 1**

- **Present p0**
  - net = W(0)p0 = (1, 1, -1)(1, -1, -1) =1
  - p0 misclassified, learning occurs
  - W(1) = W(0) + (t-o)*p0 = (-1, 3, 1)
  - New net = W(1)p0 = -5 is closer to target (t = -1)
- **Present p1**
  - net = (-1, 3, 1)(1, -1, 1) = -3
  - no learning occurs
- **Present p2**
  - net = (-1, 3, 1)(1, 1, -1) = 1
  - W(2) = (-1, 3, 1) + (-2)(1, 1, -1) = (-3, 1, 3)
  - New net = W(2)p2= -5
- **Present p3**
  - net = (-3, 1, 3)(1, 1, 1) = 1
  - no learning occurs
- **Present p0, p1, p2, p3**
  - All correctly classified with W(2)
  - Learning stops with W(2)

Delta Rule

- **The preceptron rule fail to converge if the examples are not linearly separable.**
- **Delta rule will converge toward a best-fit approximation to the target concept if the training example are not linearly separable.**
  - The delta rule is to use gradient descent to search the hypothesis space.

Gradient Descent

- **Consider simpler linear unit, where**

$$o(x) = \vec{w} \cdot \vec{x} = w_0 + w_1 x_1 + w_2 x_2 + \ldots + w_n x_n$$

- **Let's learn w_i's that minimize the squared error**

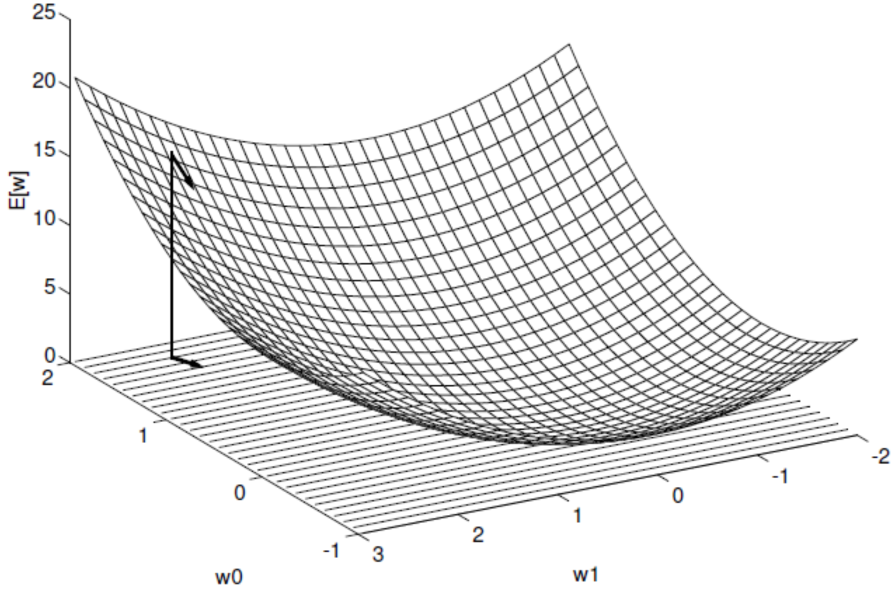
$$E(\vec{w}) = \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2$$

- **Where D is the set of training examples.**

Gradient Descent

- **Gradient**

$$\nabla E[\vec{w}] \equiv \left[\frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, \cdots \frac{\partial E}{\partial w_n}\right]$$



- **Training rule:**

$$\Delta \vec{w} = -\eta \nabla E[\vec{w}]$$

  - i.e.,

$$\Delta w_i = -\eta \frac{\partial E}{\partial w_i}$$

Gradient Descent

$$\begin{aligned}
\frac{\partial E}{\partial w_i} &= \frac{\partial}{\partial w_i}\frac{1}{2}\sum_d (t_d - o_d)^2 \\
&= \frac{1}{2}\sum_d \frac{\partial}{\partial w_i}(t_d - o_d)^2 \\
&= \frac{1}{2}\sum_d 2(t_d - o_d)\frac{\partial}{\partial w_i}(t_d - o_d) \\
&= \sum_d (t_d - o_d)\frac{\partial}{\partial w_i}(t_d - \vec{w}\cdot\vec{x_d}) \\
\frac{\partial E}{\partial w_i} &= \sum_d (t_d - o_d)(-x_{i,d})
\end{aligned}$$

$$\Delta w_i = \eta \sum_{d \in D} (t_d - o_d)\, x_{id}$$

Gradient Descent

GRADIENT-DESCENT($training\_examples, \eta$)

Each training example is a pair of the form $\langle \vec{x}, t \rangle$, where $\vec{x}$ is the vector of input values, and t is the target output value. η is the learning rate (e.g., .05).

- Initialize each w_i to some small random value
- Until the termination condition is met, Do
  - Initialize each Δw_i to zero.
  - For each $\langle \vec{x}, t \rangle$ in $training\_examples$, Do
    - Input the instance $\vec{x}$ to the unit and compute the output o
    - For each linear unit weight w_i, Do

$$\Delta w_i \leftarrow \Delta w_i + \eta (t - o) x_i$$

  - For each linear unit weight w_i, Do

$$w_i \leftarrow w_i + \Delta w_i$$

Stochastic gradient descent

- **Practical difficulties of gradient descent**
  - Converge to local minimum can sometimes be quite slow
  - If there are multiple local minima in the error surface, there is no guarantee that the procedure will find the global minimum.
- **Stochastic gradient descent: update weights incrementally**
  - Do until satisfied
    - For each training example d in D
      - Compute the gradient $\nabla E_d[\vec{x}]$
      - Then, $\vec{w} = \vec{w} - \eta \nabla E_d[\vec{w}]$
  - Stochastic (incremental) gradient descent can approximate standard gradient descent arbitrarily closely if learning rate made small enough.

Stochastic gradient descent

- **Key differences:**
  - In standard gradient descent, the error is summed over all examples before updating weights, where in stochastic gradient weights are updated upon examining each training example
  - Summing over multiple examples in standard gradient descent requires more computation per weight update step
    - Use larger step size per weight in standard gradient descent
  - In cases where there are multiple local minima with respect to E(w), stochastic gradient descent can sometimes avoid falling into these local minima.

Summary

- **Perceptron training rule updates weights on the error in the thresholded perceptron output**

$$o(\vec{x}) = \mathrm{sgn}(\vec{w} \cdot \vec{x})$$

- **Delta training rule updates weights on the error in the unthresholed linear combination of inputs**

$$o(\vec{x}) = \vec{w} \cdot \vec{x}$$

Summary

- **Perceptron training rule guaranteed to succeed if**
  - Training examples are linearly separable
  - Sufficiently small learning rate
- **Delta training rule uses gradient descent**
  - Guaranteed to converge to hypothesis with minimum squared error
  - Given sufficiently small learning rate
  - Even when training data contains noise
  - Even when training data not separable by H.

A Multilayer Neural Network

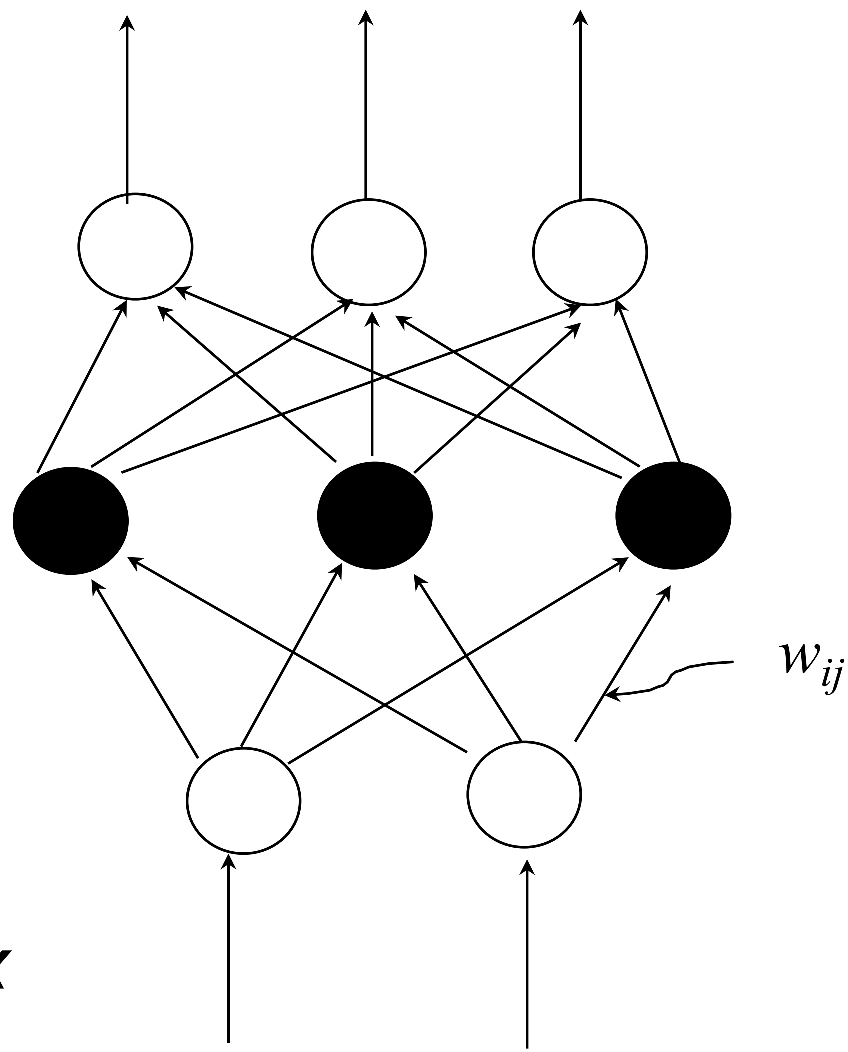
Output vector

Output layer

Hidden layer

Input layer

Input vector: *X*



How A Multilayer Neural Network Works?

- The inputs to the network correspond to the attributes measured for each training example
- Inputs are fed simultaneously into the units making up the input layer
- They are then weighted and fed simultaneously to a hidden layer
- The number of hidden layers is arbitrary, although usually only one
- The weighted outputs of the last hidden layer are input to units making up the output layer, which emits the network's prediction
- The network is feed-forward in that none of the weights cycles back to an input unit or to an output unit of a previous layer
- From a statistical point of view, networks perform nonlinear regression: Given enough hidden units and enough training samples, they can closely approximate any function

Multilayer Networks of Sigmoid Units

- **Architecture:**
  - **Feedforward** network of at least one layer of **non-linear** hidden nodes, e.g., # of layers ***L ≥ 2*** (not counting the input layer)
  - Node function is differentiable
    - most common: **sigmoid function**

$$S(net) = \frac{1}{1 + e^{(-net)}}$$

    - **Nice property:**

$$\frac{dS(x)}{dx} = S(x)(1 - S(x))$$

    - **We can derive gradient descent rules to train**
      - **One sigmoid unit**
      - **Multilayer networks of sigmoid units**

Backpropagation Learning

- **Notation:**
  - x_{ji}: the ith input to unit j
  - w_{ji}: the weight associated with ith input to unit j
  - $net_j = \sum_i w_{ji} x_{ji}$ (the weighted sum of inputs for unit j)
  - o_j: the output computed by unit j
  - t_j: the target output for unit j
  - σ: the sigmoid function
  - outputs: the set of units in the final layer of the network
  - Downstream(j): the set of units whose immediate inputs include the output of unit j.

Backpropagation Learning

- **Idea of BP learning:**
  - Update of weights in w_{21} (from hidden layer to output layer): delta rule as in a single layer net using sum square error
  - Delta rule is not applicable to updating weights in w_{10} (from input and hidden layer) because we don’t know the desired values for hidden nodes
  - **Solution**: Propagating errors at output nodes down to hidden nodes, these computed errors on hidden nodes drives the update of weights in w_{10} (again by delta rule), thus called error **BACKPROPAGATION (BP)** learning
  - How to compute errors on hidden nodes is the key
  - Error backpropagation can be continued downward if the net has more than one hidden layer
  - Proposed first by Werbos (1974), current formulation by Rumelhart, Hinton, and Williams (1986)

Backpropagation Learning

- **For each training example d every weight *w_{ji}* is updated by adding to it Δw_{ji}**

$$\Delta w_{ji} = -\eta \frac{\partial E_d}{\partial w_{ji}}$$

- **Where Ed is the error on training example d, summed over all output units in the network**

$$E(\vec{w}) = \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2$$

Backpropagation Learning

- **Noted that weight w_{ji} can influence the rest of the network only through net_j. Therefore, we can use the chain rule to write**

$$\frac{\partial E_d}{\partial w_{ji}} = \frac{\partial E_d}{\partial net_j}\frac{\partial net_j}{\partial w_{ji}} = \frac{\partial E_d}{\partial net_j} x_{ji}$$

- **Our remaining task is to derive a convenient expression of $\frac{\partial E_d}{\partial net_j}$. Two cases are considered:**
  - Unit j is an output unit for the network
  - Unit j is an internal unit.

Backpropagation Learning

- **Training rule for output unit weights**
  - net_j can influence the rest of the network only through o_j, Then

$$\frac{\partial E_d}{\partial net_j} = \frac{\partial E_d}{\partial o_j}\frac{\partial o_j}{\partial net_j}$$

  - First term:

Derivatives will be zero for all output units except j

$$\frac{\partial E_d}{\partial o_j} = \frac{\partial}{\partial o_j}\frac{1}{2}\sum_{k\in outputs}(t_k - o_k)^2$$

$$= \frac{\partial}{\partial o_j}\frac{1}{2}(t_j - o_j)^2 = \frac{1}{2}\times 2\times(t_j - o_j)\frac{\partial(t_j - o_j)}{\partial o_j}$$

$$= -(t_j - o_j)$$

Backpropagation Learning

- Second term:

$$o_j = \sigma(net_j)$$

$$\frac{\partial o_j}{\partial net_j} = \frac{\partial \sigma(net_j)}{\partial net_j} = o_j(1-o_j)$$

- Put it together:

$$\frac{\partial E_d}{\partial net_j} = -(t_j - o_j)o_j(1-o_j)$$

- Then, we have the stochastic gradient descent rule for output units

$$\Delta w_{ji} = -\eta \frac{\partial E_d}{\partial w_{ji}} = \eta(t_j - o_j)o_j(1-o_j)x_{ji}$$

Backpropagation Learning

- **Training rule for hidden unit weights**
  - net_j (j is the internal node) can influence the rest of the network through Downstream(j), Then

$$\frac{\partial E_d}{\partial net_j} = \sum_{k \in Downstream(j)} \frac{\partial E_d}{\partial net_k} \frac{\partial net_k}{\partial net_j}$$

$$= \sum_{k \in Downstream(j)} -\delta_k \frac{\partial net_k}{\partial net_j}$$

$$= \sum_{k \in Downstream(j)} -\delta_k \frac{\partial net_k}{\partial o_j} \frac{\partial o_j}{\partial net_j}$$

$$= \sum_{k \in Downstream(j)} -\delta_k w_{kj} \frac{\partial o_j}{\partial net_j}$$

$$= \sum_{k \in Downstream(j)} -\delta_k w_{kj} o_j (1 - o_j)$$

Backpropagation Learning

- **We set**

$$\delta_j = -\frac{\partial E_d}{\partial net_j} = o_j(1-o_j)\sum_{k\in Downstream(j)} \delta_k w_{kj}$$

- **Then, we have the stochastic gradient descent rule for hidden units**

$$\Delta w_{ji} = \eta \delta_j x_{ji}$$

Backpropagation Learning

BACKPROPAGATION($training\_examples, \eta, n_{in}, n_{out}, n_{hidden}$)

Each training example is a pair of the form $\langle \vec{x}, \vec{t} \rangle$, where $\vec{x}$ is the vector of network input values, and $\vec{t}$ is the vector of target network output values.

η is the learning rate (e.g., .05). n_{in} is the number of network inputs, n_{hidden} the number of units in the hidden layer, and n_{out} the number of output units.

The input from unit i into unit j is denoted x_{ji}, and the weight from unit i to unit j is denoted w_{ji}.

- Create a feed-forward network with n_{in} inputs, n_{hidden} hidden units, and n_{out} output units.
- Initialize all network weights to small random numbers (e.g., between $-.05$ and $.05$).
- Until the termination condition is met, Do
  - For each $\langle \vec{x}, \vec{t} \rangle$ in $training\_examples$, Do

    Propagate the input forward through the network:

    1. Input the instance $\vec{x}$ to the network and compute the output o_u of every unit u in the network.

    Propagate the errors backward through the network:

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

    $$\delta_k \leftarrow o_k(1 - o_k)(t_k - o_k) \tag{T4.3}$$

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

    $$\delta_h \leftarrow o_h(1 - o_h) \sum_{k \in outputs} w_{kh} \delta_k \tag{T4.4}$$

    4. Update each network weight w_{ji}

    $$w_{ji} \leftarrow w_{ji} + \Delta w_{ji}$$

    where

    $$\Delta w_{ji} = \eta \, \delta_j \, x_{ji} \tag{T4.5}$$

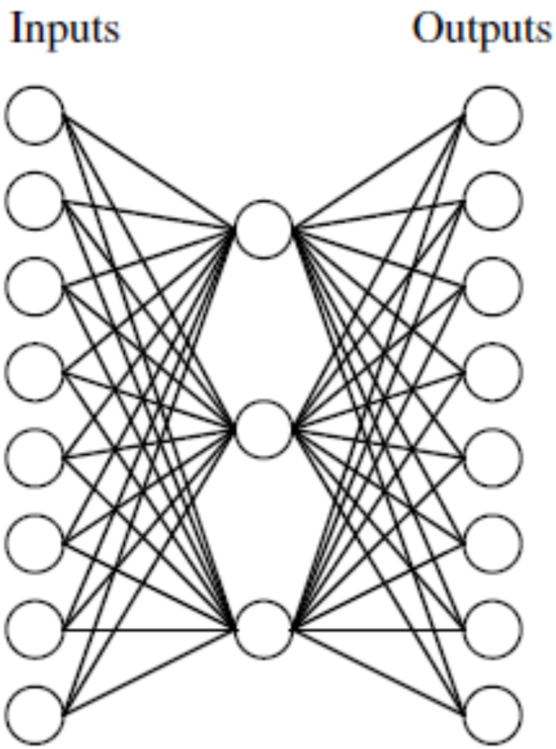
Learning Hidden Layer Representations

- **A target function**

Input		Output
10000000	$\rightarrow$	10000000
01000000	$\rightarrow$	01000000
00100000	$\rightarrow$	00100000
00010000	$\rightarrow$	00010000
00001000	$\rightarrow$	00001000
00000100	$\rightarrow$	00000100
00000010	$\rightarrow$	00000010
00000001	$\rightarrow$	00000001

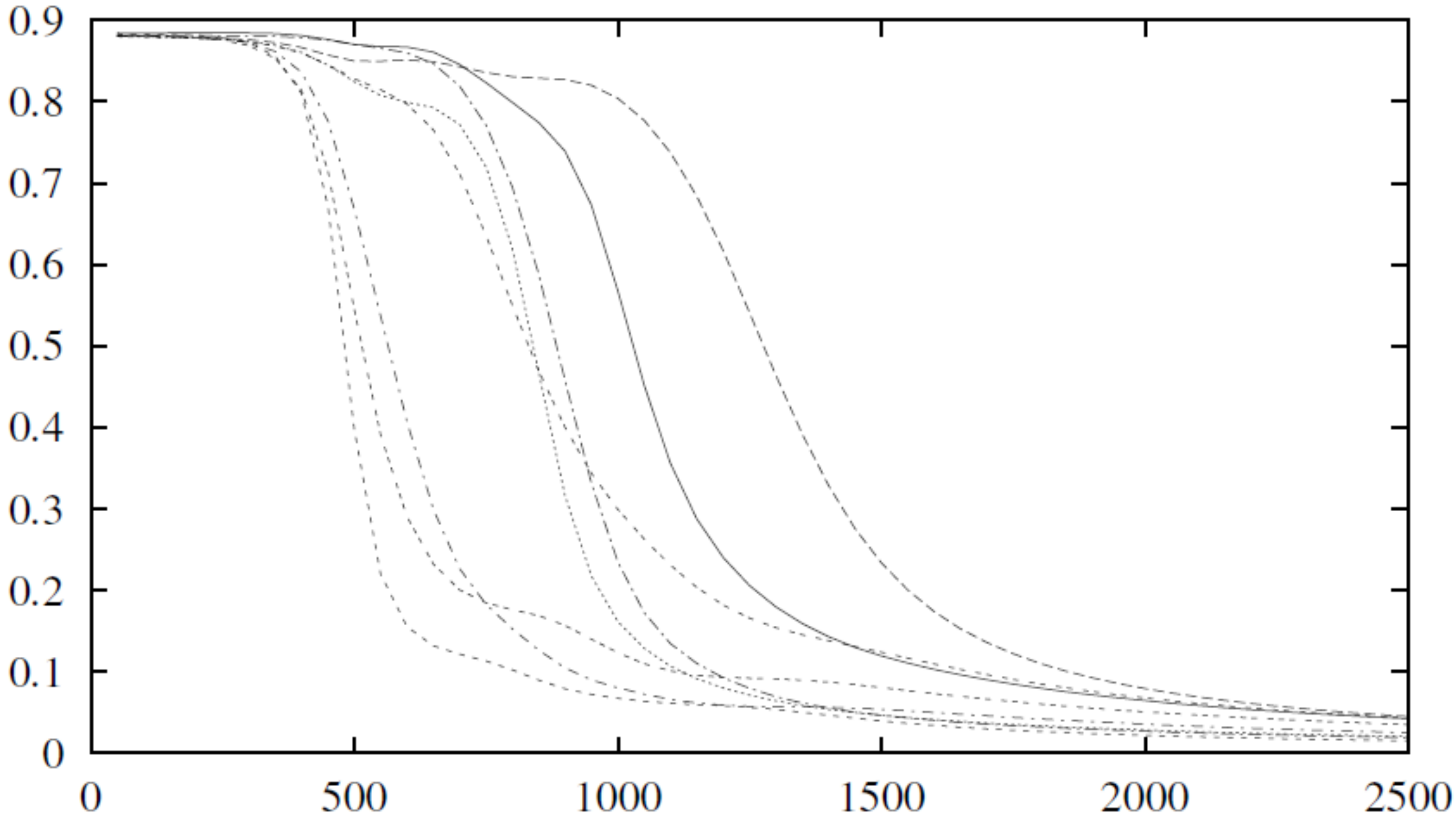
Learning Hidden Layer Representations

- **A network:**



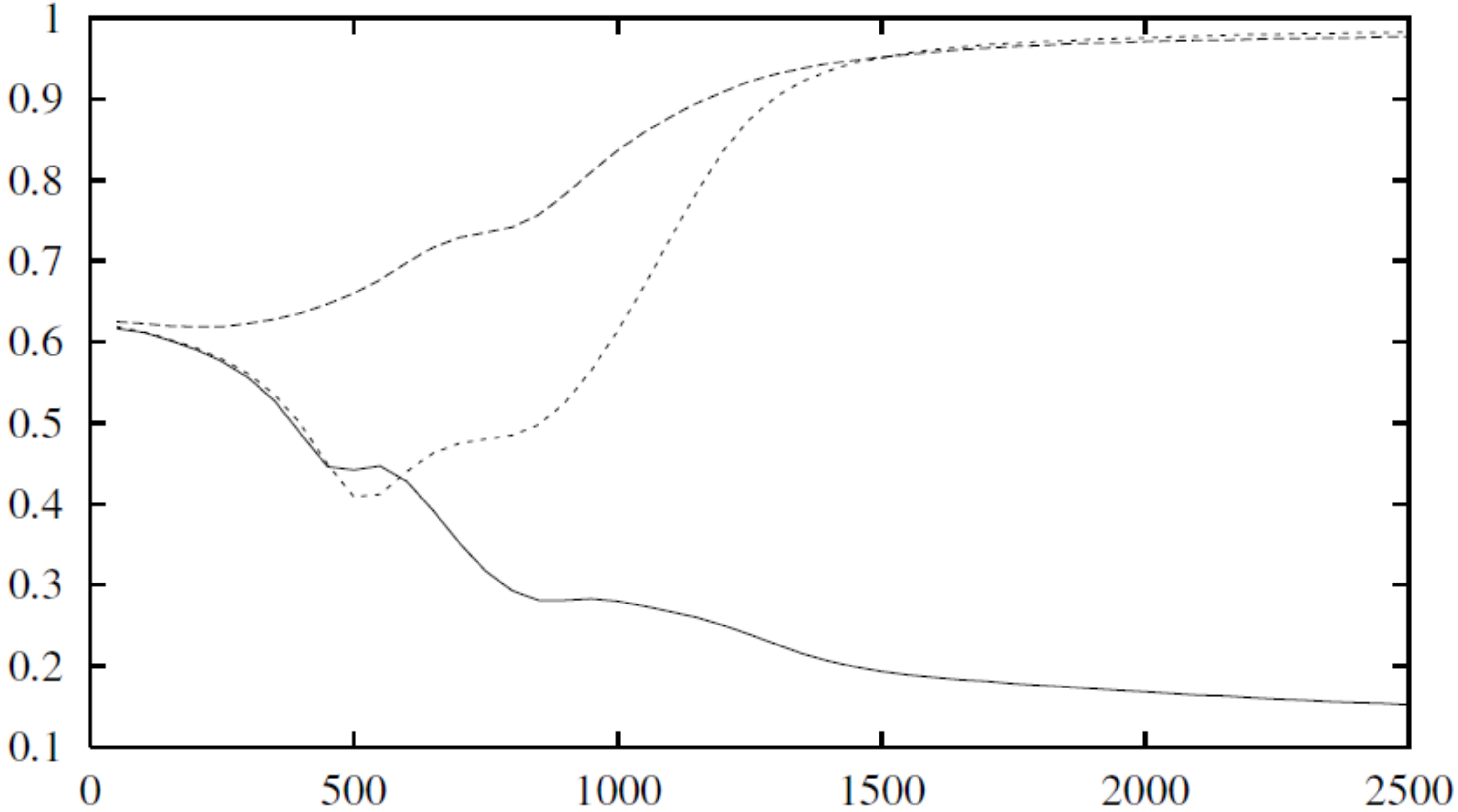
Learning Hidden Layer Representations

- **Sum of squared errors for each output unit**



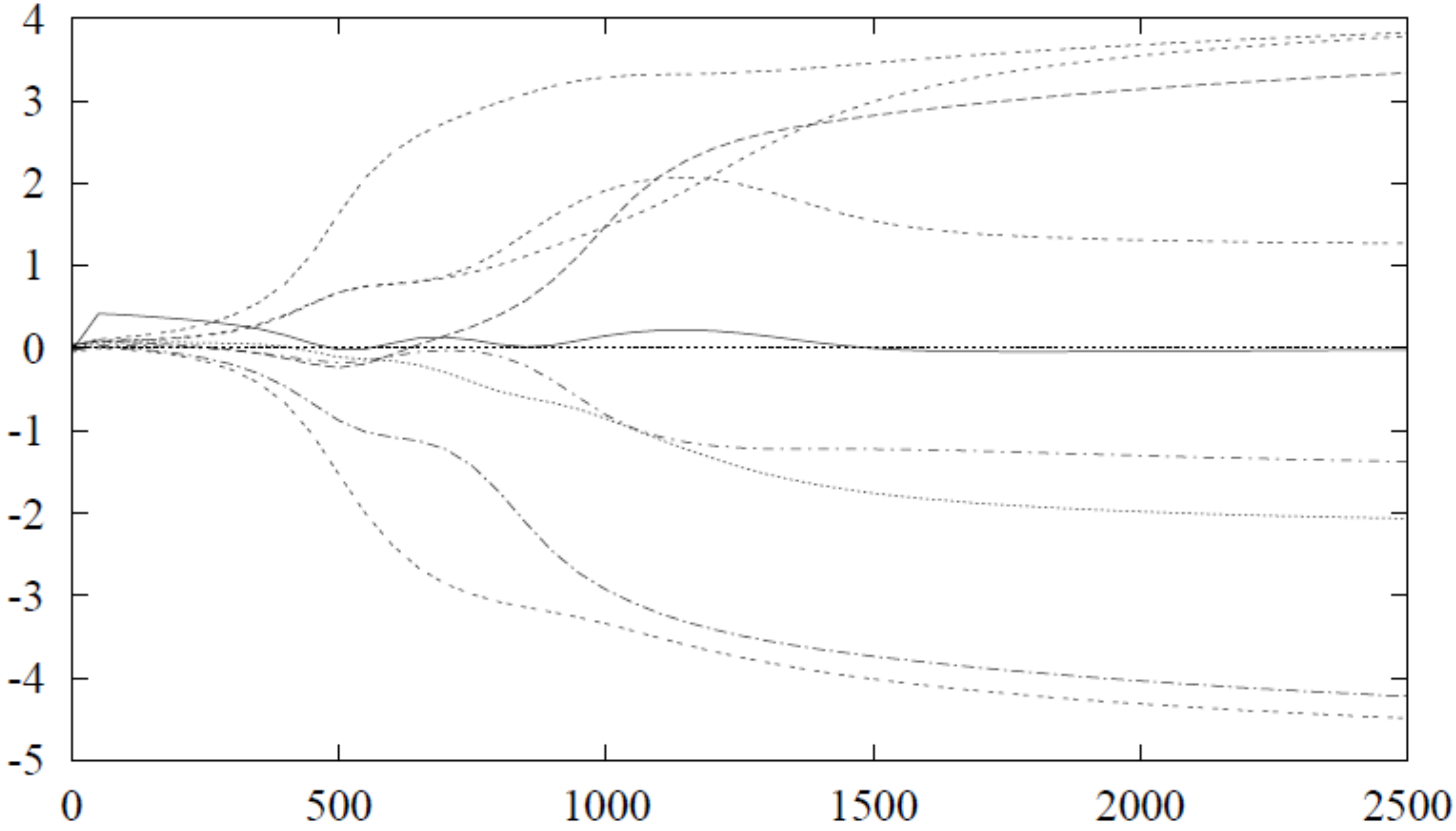
Learning Hidden Layer Representations

- **Hidden unit encoding for input 01000000**



Learning Hidden Layer Representations

- **Weights from inputs to on hidden unit**



Learning Hidden Layer Representations

- **Learned hidden layer representation after 5000 training epochs**

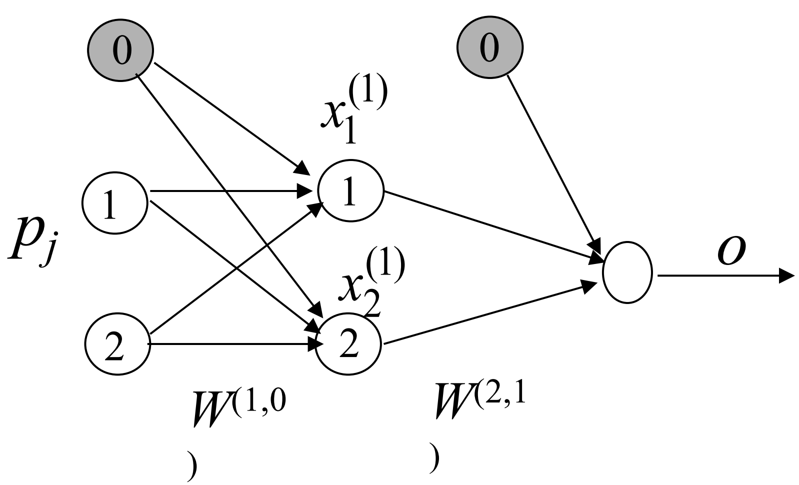
Input		Hidden Values				Output
10000000	$\rightarrow$	.89	.04	.08	$\rightarrow$	10000000
01000000	$\rightarrow$	.01	.11	.88	$\rightarrow$	01000000
00100000	$\rightarrow$	.01	.97	.27	$\rightarrow$	00100000
00010000	$\rightarrow$	.99	.97	.71	$\rightarrow$	00010000
00001000	$\rightarrow$	.03	.05	.02	$\rightarrow$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

Example, BP learning function XOR

- **Training samples (bipolar)**

	in_1	in_2	d
P0	-1	-1	-1
P1	-1	1	1
P2	1	-1	1
P3	1	1	-1

- **Network: 2-2-1 with thresholds (fixed output 1)**



- Initial weights W(0)

$$w_1^{(1,0)} : (-0.5,\ 0.5, -0.5)$$
$$w_2^{(1,0)} : (-0.5,\ -0.5,\ 0.5)$$
$$w^{(2,1)} : (-1,\ 1,\ 1)$$

- Learning rate = 0.2
- Node function: hyperbolic tangent

$$g(x) = \tanh(x) = \frac{1 - e^{-x}}{1 + e^{-x}};$$
$$\lim_{x \to \pm\infty} g(x) = \pm 1$$
$$s(x) = \frac{1}{1 + e^{-x}};$$
$$g(x) = 2s(x) - 1$$
$$s'(x) = s(x)(1 - s(x))$$
$$g'(x) = 0.5(1 + g(x))(1 - g(x))$$

Present $P_0 = (1, -1, -1)$: $d_0 = -1$

Forward computing

$$net_1 = w_1^{(1,0)} p_0 = (-0.5,\ 0.5, -0.5)\ (1, -1, -1) = -0.5$$

$$net_2 = w_2^{(1,0)} p_0 = (-0.5, -0.5, 0.5)\ (1, -1, -1) = -0.5$$

$$x_1^{(1)} = g(net_1) = 2/(1+e^{0.5}) - 1 = \text{-}0.24492$$

$$x_1^{(1)} = g(net_2) = 2/(1+e^{0.5}) - 1 = \text{-}0.24492$$

$$net_o = w^{(2,1)} x^{(1)} = (-1, 1, 1)(1, \text{-}\ 0.24492, \text{-}\ 0.24492) = \text{-}1.48984$$

$$o = g(net_o) = \text{-}0.63211$$

Error back propogating

$$l = d - o = -1 - (\text{-}0.63211) = \text{-}0.36789$$

$$\delta = l \cdot g'(net_o) = l \cdot (1 + g(net_o))(1 - g(net_o))$$
$$= \text{-}0.3679 \cdot (1 \text{ - } 0.6321)(1 + 0.6321) = -0.2209$$

$$\mu_1 = \delta \cdot w_1^{(2,1)} \cdot g'(net_1)$$
$$= \text{-}0.2209 \cdot 1 \cdot (1 \text{ - } 0.24492) \cdot (1 + 0.24492) = \text{-}0.20765$$

$$\mu_2 = \delta \cdot w_2^{(2,1)} \cdot g'(net_2)$$
$$= \text{-}0.2209 \cdot 1 \cdot (1 \text{ - } 0.24492) \cdot (1 + 0.24492) = \text{-}0.20765$$

Weight update

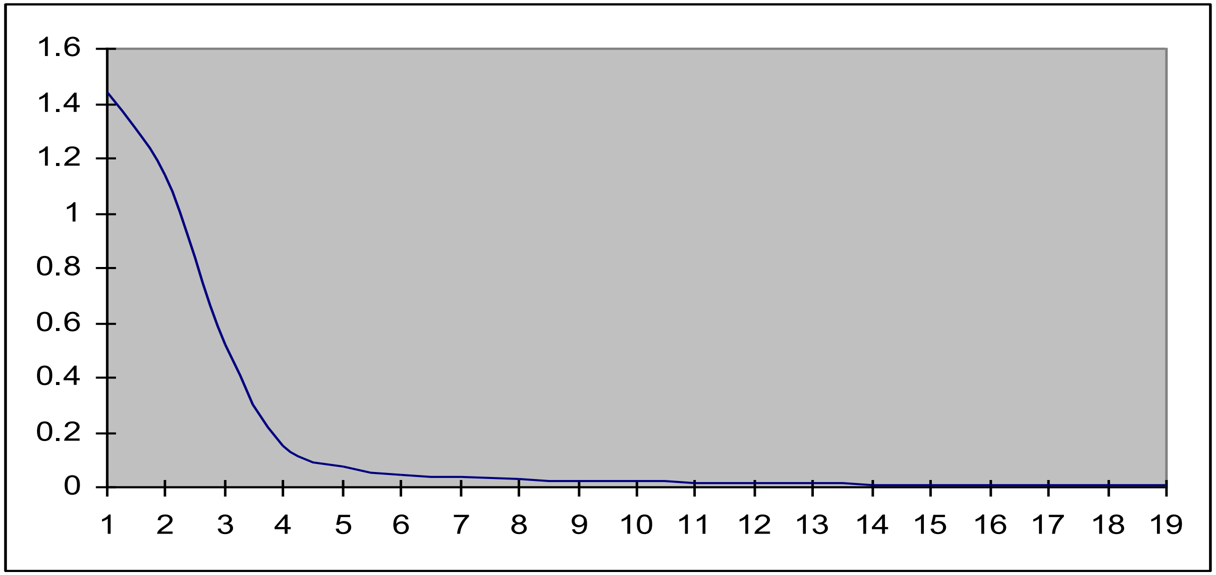
$$\Delta w^{(2,1)} = \eta \cdot \delta \cdot x^{(1)}$$
$$= 0.2 \cdot (-0.2209) \cdot (1, -0.2449, -0.2449) = (-0.0442,\ 0.0108, 0.0108)$$

$$w^{(2,1)} = w^{(2,1)} + \Delta w^{(2,1)} = (-1, 1, 1) + (-0.0442,\ 0.0108, 0.0108)$$
$$= (-0.5415,\ 1.0108, 1.0108)$$

$$\Delta w_1^{(1,0)} = \eta \cdot \mu_1 \cdot p_0 = 0.2 \cdot (-0.2077) \cdot (1, -1, -1) = (-0.0415, 0.0415, 0.0415)$$
$$\Delta w_2^{(1,0)} = \eta \cdot \mu_2 \cdot p_0 = 0.2 \cdot (-0.2077) \cdot (1, -1, -1) = (-0.0415, 0.0415, 0.0415)$$
$$w_1^{(1,0)} = w_1^{(1,0)} + \Delta w_1^{(1,0)} = (-0.5, 0.5, -0.5) + (-0.0415, 0.0415, 0.0415)$$
$$= (-0.5415, 0.5415, -0.4585)$$
$$w_2^{(1,0)} = w_2^{(1,0)} + \Delta w_2^{(1,0)} = (-0.5, -0.5, 0.5) + (-0.0415, 0.0415, 0.0415)$$
$$= (-0.5415,\ -0.4585, 0.5415)$$

Error for $P_0 = l^2$ reduced from 0.135345 to 0.102823

MSE reduction:
every 10 epochs



Output: every 10 epochs

epoch	1	10	20	40	90	140	190	d
P0	-0.63	-0.05	-0.38	-0.77	-0.89	-0.92	-0.93	**-1**
P1	-0.63	-0.08	0.23	0.68	0.85	0.89	0.90	**1**
P2	-0.62	-0.16	0.15	0.68	0.85	0.89	0.90	**1**
p3	-0.38	0.03	-0.37	-0.77	-0.89	-0.92	-0.93	**-1**
MSE	**1.44**	**1.12**	**0.52**	**0.074**	**0.019**	**0.010**	**0.007**	

After epoch 1

	$w_1^{(1,0)}$	$w_2^{(1,0)}$	$w^{(2,1)}$
init	(-0.5, 0.5, -0.5)	(-0.5, -0.5, 0.5)	(-1, 1, 1)
p0	-0.5415, 0.5415, -0.4585	-0.5415, -0.45845, 0.5415	-1.0442, 1.0108, 1.0108
p1	-0.5732, 0.5732, -0.4266	-0.5732, -0.4268, 0.5732	-1.0787, 1.0213, 1.0213
p2	-0.3858, 0.7607, -0.6142	-0.4617, -0.3152, 0.4617	-0.8867, 1.0616, 0.8952
p3	-0.4591, 0.6874, -0.6875	-0.5228, -0.3763, 0.4005	-0.9567, 1.0699, 0.9061

epoch

13	-1.4018, 1.4177, -1.6290	-1.5219, -1.8368, 1.6367	0.6917, 1.1440, 1.1693
40	-2.2827, 2.5563, -2.5987	-2.3627, -2.6817, 2.6417	1.9870, 2.4841, 2.4580
90	-2.6416, 2.9562, -2.9679	-2.7002, -3.0275, 3.0159	2.7061, 3.1776, 3.1667
190	-2.8594, 3.18739, -3.1921	-2.9080, -3.2403, 3.2356	3.1995, 3.6531, 3.6468

Strength of BP

- **Great representation power**
  - Boolean functions
    - Every Boolean function can be represented by network with single hidden layer
    - But might require exponential 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
- **Wide applicability of BP learning**
  - Only requires that a good set of training samples is available
  - Does not require substantial prior knowledge or deep understanding of the domain itself (ill structured problems)
  - Tolerates noise and missing data in training samples (graceful degrading)
- **Easy to implement the core of the learning algorithm**
- **Good generalization power**
  - Often produce accurate results for inputs outside the training set

Deficiencies of BP

- **Learning often takes a long time to converge**
  - Complex functions often need hundreds or thousands of epochs
- **The net is essentially a black box**
  - It may provide a desired mapping between input and output vectors (***x, o***) but does not have the information of why a particular ***x*** is mapped to a particular ***o.***
  - It thus cannot provide an intuitive (e.g., causal) explanation for the computed result.
  - This is because the hidden nodes and the learned weights do not have clear semantics.
    - What can be learned are operational parameters, not general, abstract knowledge of a domain
  - Unlike many statistical methods, there is no theoretically well-founded way to **assess the quality** of BP learning
    - What is the confidence level one can have for a trained BP net, with the final E (which may or may not be close to zero)?
    - What is the confidence level of o computed from input x using such net?

Deficiencies of BP

- **Problem with gradient descent approach**
  - only guarantees to reduce the total error to a **local minimum**. (***E*** may not be reduced to zero)
  - Cannot escape from the local minimum error state
  - **Not every function that is representable can be learned**
  - How bad: depends on the shape of the error surface. Too many valleys/wells will make it easy to be trapped in local minima
  - Possible remedies:
    - Try nets with different # of hidden layers and hidden nodes (they may lead to different error surfaces, some might be better than others)
    - Try different initial weights (different starting points on the surface)
    - Forced escape from local minima by random perturbation (e.g., simulated annealing)

Variations of BP nets

- **Adding momentum term (to speedup learning)**
  - Weights update at time n contains the momentum of the previous updates, e.g.,

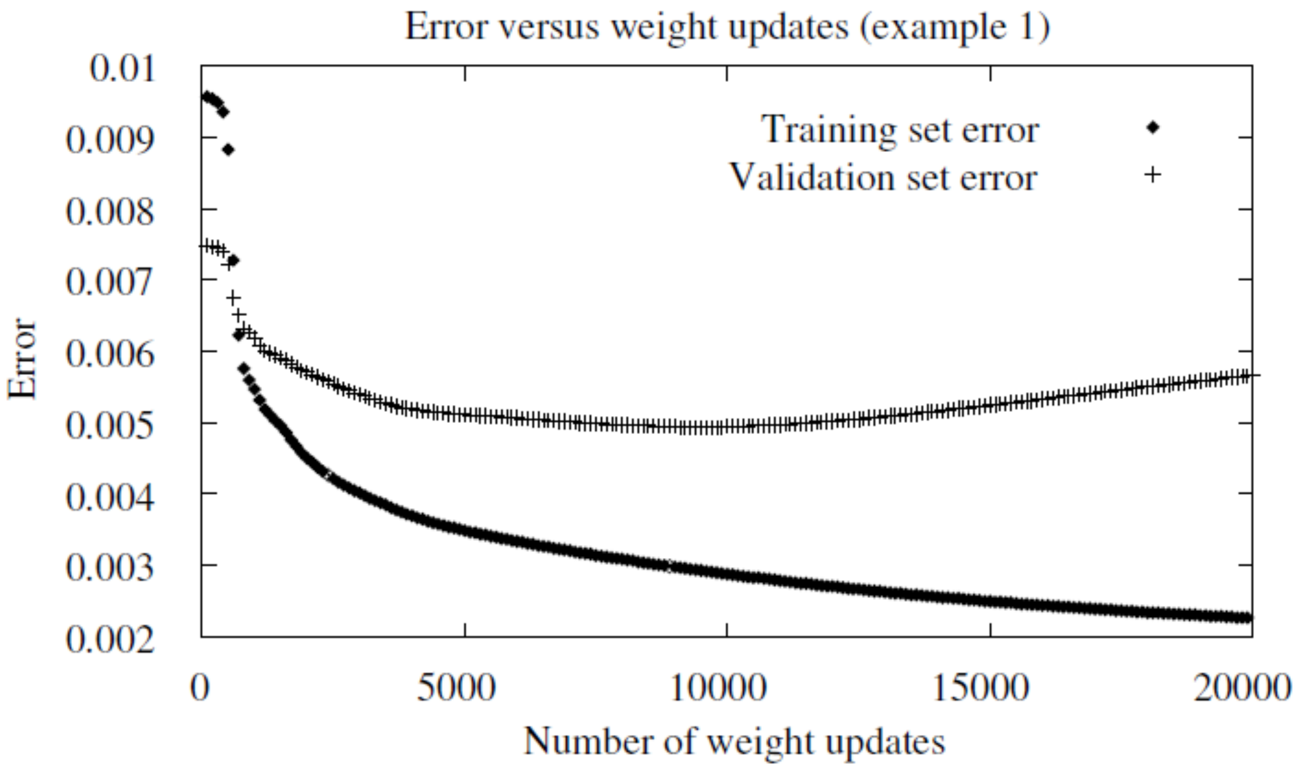
$$\Delta w_{ji}(n) = \eta \delta_j x_{ji} + \alpha \Delta w_{ji}(n-1)$$

  - Avoid sudden change of directions of weight update (smoothing the learning process)
  - Error is no longer monotonically decreasing
- **Batch mode of weight update**
  - Weight update once per each epoch (cumulated over all P samples)
  - Smoothing the training sample outliers
  - Learning independent of the order of sample

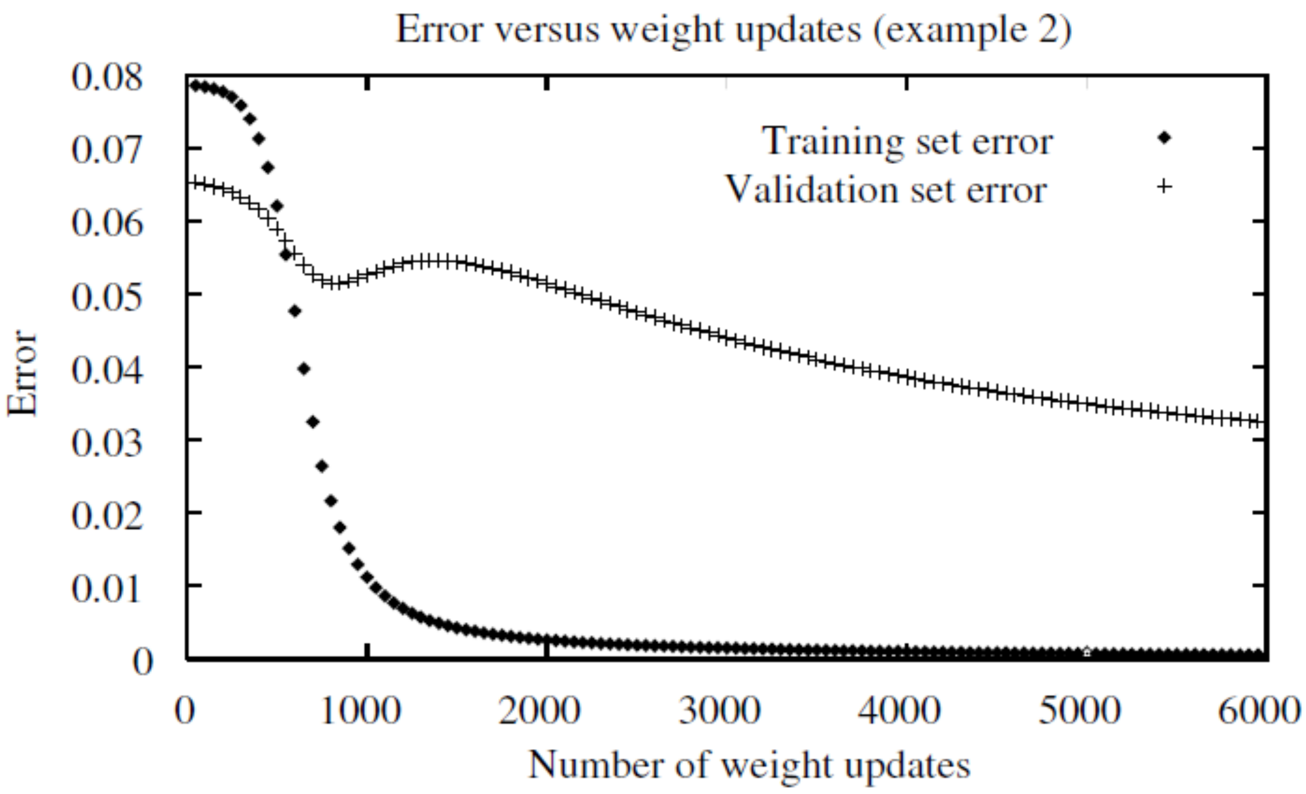
Variations of BP nets

- **Variations on learning rate η**
  - Fixed rate much smaller than 1
  - Start with large η, gradually decrease its value
  - Start with a small η, steadily double it until MSE start to increase
  - Give known underrepresented samples higher rates
  - Find the maximum safe step size at each stage of learning (to avoid overshoot the minimum E when increasing η**)**
  - **Adaptive learning rate** (delta-bar-delta method)
    - Each weight $w_{k,j}$ has its own rate $\eta_{k,j}$
    - If $\Delta w_{k,j}$ remains in the same direction, increase $\eta_{k,j}$ (E has a smooth curve in the vicinity of current w)
    - If $\Delta w_{k,j}$ changes the direction, decrease $\eta_{k,j}$ (E has a rough curve in the vicinity of current w)

Overfitting in Neural Networks



Overfitting in Neural Networks



Overfitting in Neural Networks

- **How to address the overfitting problem**
  - Weight decay: decrease each weight by some small factor during each iteration
  - Use a validation set of data

Practical Considerations

- **A good BP net requires more than the core of the learning algorithms. Many parameters must be carefully selected to ensure a good performance.**
- **Although the deficiencies of BP nets cannot be completely cured, some of them can be eased by some practical means.**
- **Initial weights (and biases)**
  - Random, [-0.05, 0.05], [-0.1, 0.1], [-1, 1]
  - Normalize weights for hidden layer ($w^{(1,\,0)}$) (Nguyen-Widrow)
    - Random assign initial weights for all hidden nodes
    - For each hidden node *j*, normalize its weight by

$$w_{j,i}^{(1,0)} = \beta \cdot w_{j,i}^{(1,0)} / \left\| w_j^{(1,0)} \right\|_2 \qquad \text{where } \beta = 0.7\sqrt[n]{m}$$

$$m = \#\text{ of hiddent nodes, } n = \#\text{ of input nodes}$$

$$\left\| w_j^{(1,0)} \right\|_2 = \beta \text{ after normalization}$$

    - Avoid bias in weight initialization:

Practical Considerations

- **Training samples:**
  - Quality and quantity of training samples often determines the quality of learning results
  - Samples must collectively represent well the problem space
    - Random sampling
    - Proportional sampling (with prior knowledge of the problem space)
  - # of training patterns needed: There is no theoretically idea number.
    - Baum and Haussler (1989): P = W/e, where

      W: total # of weights to be trained (depends on net structure)

      e: acceptable classification error rate

      If the net can be trained to correctly classify (1 – e/2)P of the P training samples, then classification accuracy of this net is 1 – e for input patterns drawn from the same sample space

      Example: W = 27, e = 0.05, P = 540. If we can successfully train the network to correctly classify (1 – 0.05/2)*540 = 526 of the samples, the net will work correctly 95% of time with other input.

Practical Considerations

- **How many hidden layers and hidden nodes per layer:**
  - Theoretically, one hidden layer (possibly with many hidden nodes) is sufficient for any L2 functions
  - There is no theoretical results on minimum necessary # of hidden nodes
  - Practical rule of thumb:
    - n = # of input nodes; m = # of hidden nodes
    - For binary/bipolar data: m = 2n
    - For real data: m >> 2n
  - Multiple hidden layers with fewer nodes may be trained faster for similar quality in some applications

Practical Considerations

- **Data representation:**
  - Binary vs. bipolar
    - Bipolar representation uses training samples more efficiently

      $$\Delta w_{j,i}^{(1,0)} = \eta \cdot \mu_j \cdot x_i \qquad \Delta w_{k,j}^{(2,1)} = \eta \cdot \delta_k \cdot x_j^{(1)}$$

      no learning will occur when $x_i = 0$ or $x_j^{(1)} = 0$ with binary rep.
    - # of patterns can be represented with n input nodes:

      binary: 2^n

      bipolar: 2^(n-1) if no biases used, this is due to (anti) symmetry

      (if output for input *x* is *o*, output for input –*x* will be –*o*)
  - Real value data
    - Input nodes: real value nodes (may subject to normalization)
    - Hidden nodes with sigmoid or other non-linear function
    - Node function for output nodes: often linear (even identity)

      e.g., $$o_k = \sum w_{k,j}^{(2,1)} x_j^{(1)}$$
    - Training may be much slower than with binary/bipolar data (some use binary encoding of real values)

Neural Network as a Classifier

- **Weakness**
  - Long training time
  - Require a number of parameters typically best determined empirically, e.g., the network topology or “structure."
  - Poor interpretability: Difficult to interpret the symbolic meaning behind the learned weights and of “hidden units" in the network
- **Strength**
  - High tolerance to noisy data
  - Ability to classify untrained patterns
  - Well-suited for continuous-valued inputs and outputs
  - Successful on a wide array of real-world data
  - Algorithms are inherently parallel
  - Techniques have recently been developed for the extraction of rules from trained neural networks