The Brain: A Paradox

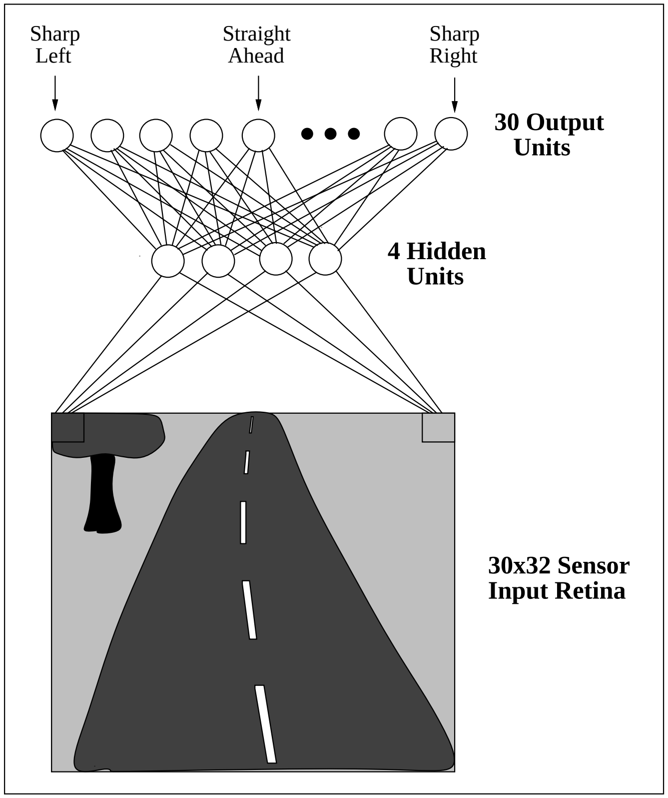
- The brain contains 10^{11} "neurons", each of which may have upto 10^4 i/o connections.
- Each neuron is "slow", with a switching time of 1 msec.
- Yet the brain is astonishingly fast (and reliable) at computationally intensive tasks like vision, speech recognition, and retrieving stored knowledge.
- Neural nets or "connectionism" is a field based on the assumption that a computational architecture similar to the brain will duplicate (at least some of) its wonderful abilities.

A Brief History of Neural Networks (Pomerleau)

- **1955-65:** Rosenblatt's Perceptron.
- **Late 60's:** Minksy and Papert publish definite analysis of perceptrons
- **1975:** Werbos' Ph.d. thesis at Harvard (Beyond regression) defines backpropagation.
- **1985:** PDP book published that ushers in modern era of neural networks.
- **1990's:** Neural networks enter mainstream applications.

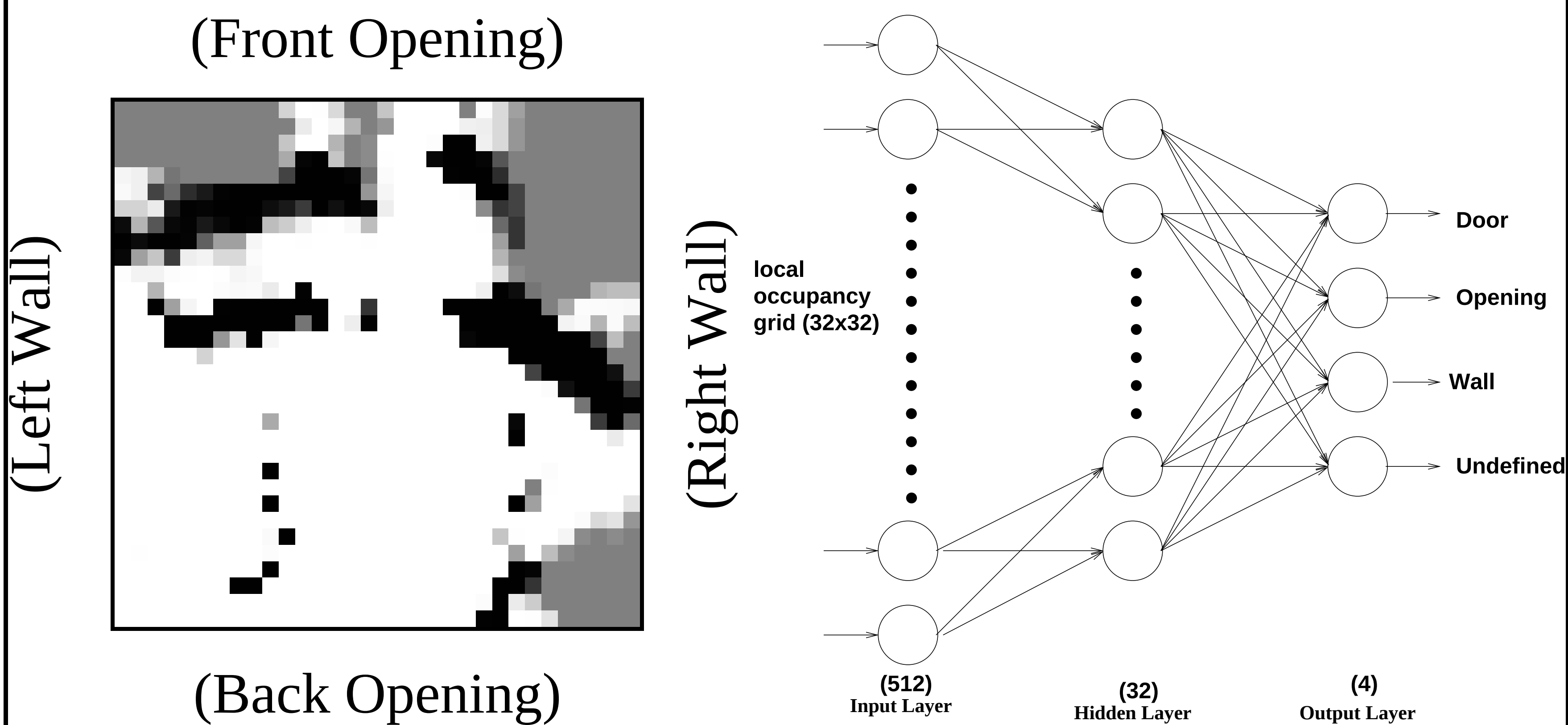
ALVINN: A Neural Network-based Autonomous Vehicle

(Pomerleau)



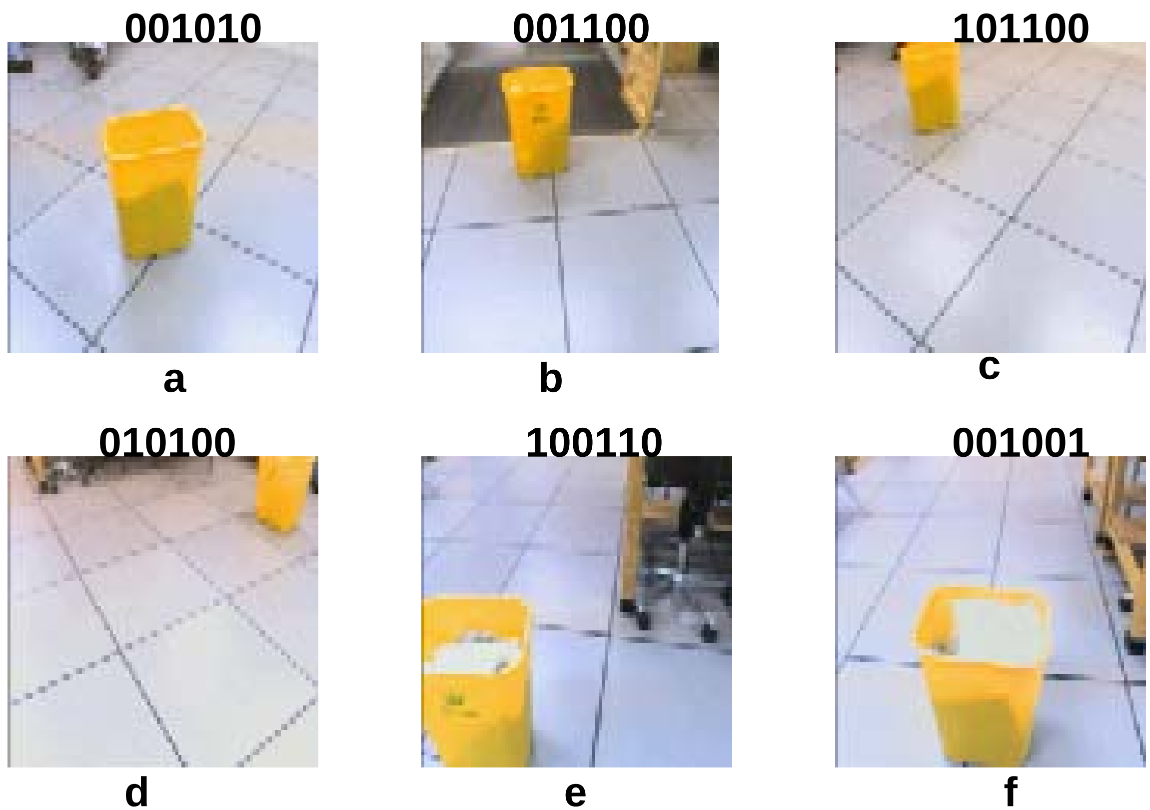
PAVLOV: A Neural-Net based Navigation Architecture

(Khaleeli)



PAVLOV: Learning to Find Trashcans

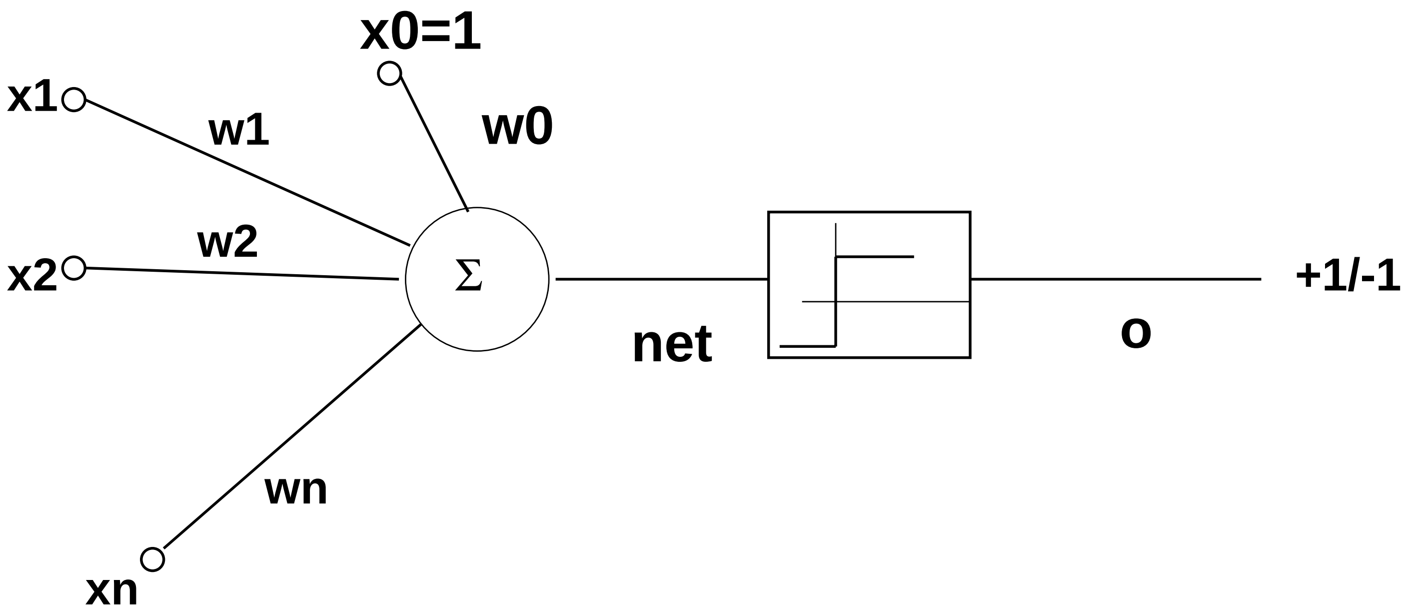
(Theocharous)



Problems Suited to Neural Networks

- Input space is high-dimensional and continuous
- Output space is multi-dimensional and discrete/continuous
- Training examples are noisy
- Long training times are feasible
- Explanation of learned structure is not necessary
- Fast computing of output given input

Single Layer Perceptrons



$$net = \sum_{i=0}^{n} w_i x_i$$

$o = +1$ if $net > 0$ else $o = -1$.

Single Layer Perceptrons

Simplest net over real-valued input.

$$o(x_1, \ldots, x_n) = 1 \quad \text{if} \sum_{i=0}^{N} w_i x_i > 0, \quad -1 \quad \text{otherwise}$$

$$o = +1 \Rightarrow ClassA$$

$$o = -1 \Rightarrow ClassB$$

Example: Let $N = 2$. Then

$$w_0 + w_1 x_1 + w_2 x_2 = 0$$

$$x_2 = -\frac{w_1}{w_2} x_1 - \frac{w_0}{w_2}$$

The Perceptron Learning Algorithm

1. **Initialize weights and threshold:** Set weights w_i to small random values.
2. **Present Input and Desired Output:** Set the inputs to the example values x_i and let the desired output be t.
3. **Calculate Actual Output:**

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

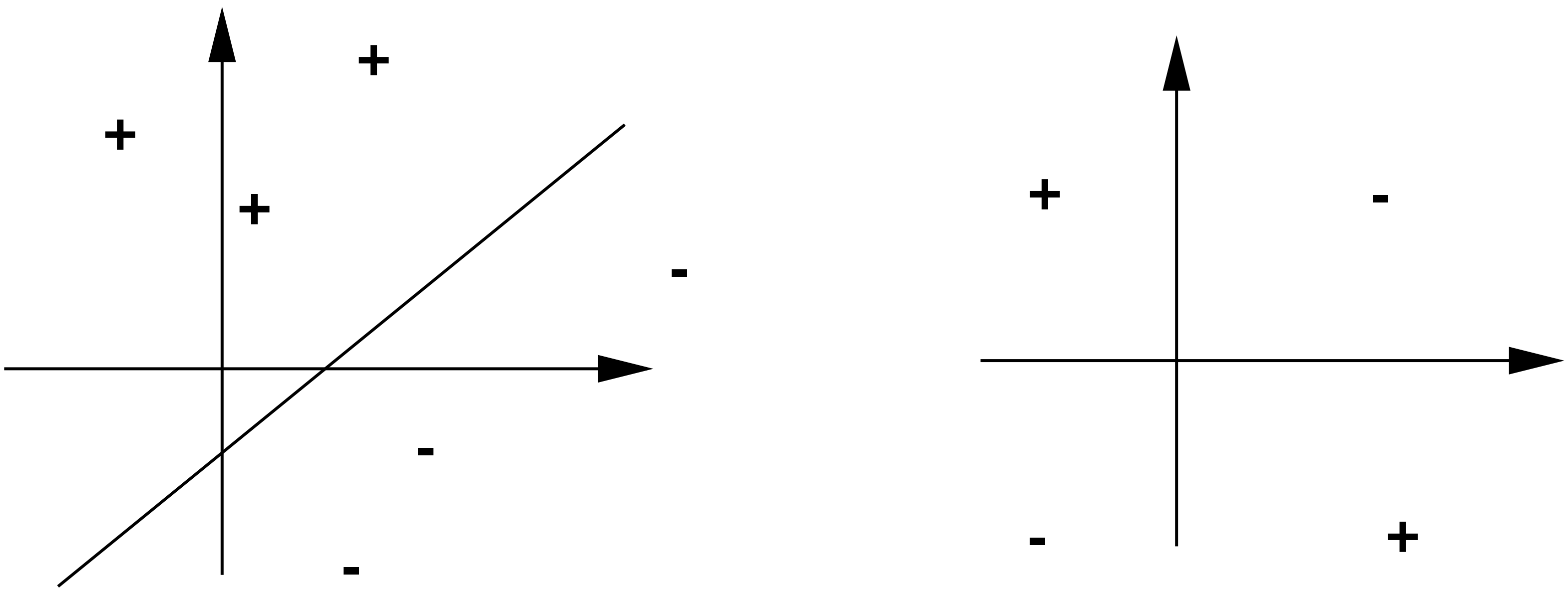
4. **Adapt Weights:** If actual output is different from desired output, then

$$w_i \Leftarrow w_i + \alpha(t - o)x_i$$

   where $0 < \alpha < 1$ is the learning rate.

5. Repeat from Step 2 until done.

Decision Surface of a Perceptron



Some linearly separable functions: AND,...

Not all functions are linearly separable (e.g. XOR).

Gradient Descent in Error Space

Gradient Descent in Error Space

- Given a set of weights w_i, the mean-squared error over the set of training instances is

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

- The function $E(\vec{w})$ defines an error surface in weight space.
- To find the weight vector that yields the lowest error, we can do gradient descent along the error surface.
- The direction of steepest descent is given by the *gradient* function

$$\nabla E(\vec{w}) = \left[\frac{\partial E}{\partial w_1}, \ldots, \frac{\partial E}{\partial w_n} \right]$$

Learning by Gradient Descent

- The training rule for gradient descent is

$$\Delta w = -\eta \nabla E(\vec{w})$$

- The weight w_i is changed by the amount

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

- For a linear unit (unthresholded perceptron), the weight update is

$$\Delta w_i = \eta \sum_{d \in D} (t_d - o_d) x_i^d$$

Incremental (Stochastic) Gradient Descent

Batch Mode: Do until error < minimum

1. Compute the gradient $\nabla E_D[\vec{w}]$
2. $\vec{w} \leftarrow \vec{w} - \eta \nabla E_D[\vec{w}]$

Incremental Mode: Do until error < minimum

1. For each training example $d \in D$
    - Compute the gradient $\nabla E_d[\vec{w}]$
    - $\vec{w} \leftarrow \vec{w} - \eta \nabla E_d[\vec{w}]$

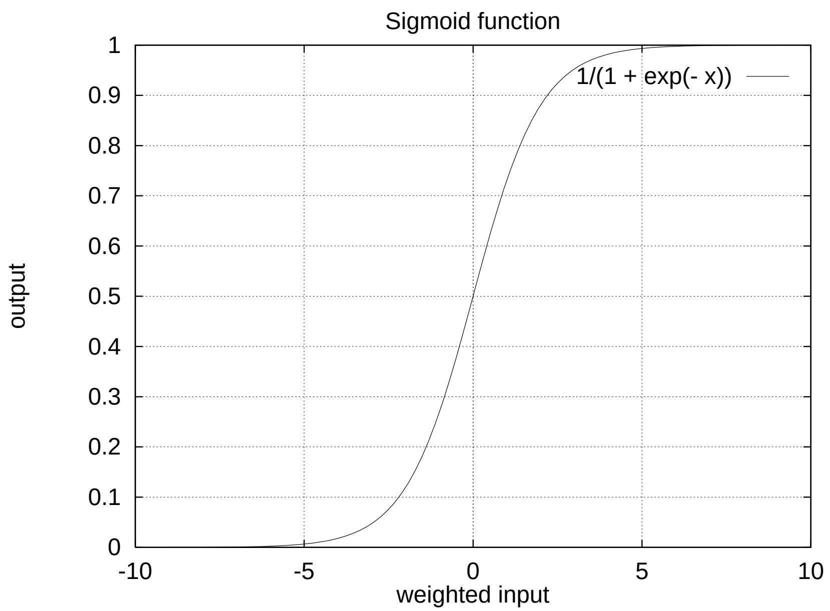
$$E_D[\vec{w}] = \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2$$
$$E_d[\vec{w}] = \frac{1}{2} (t_d - o_d)^2$$

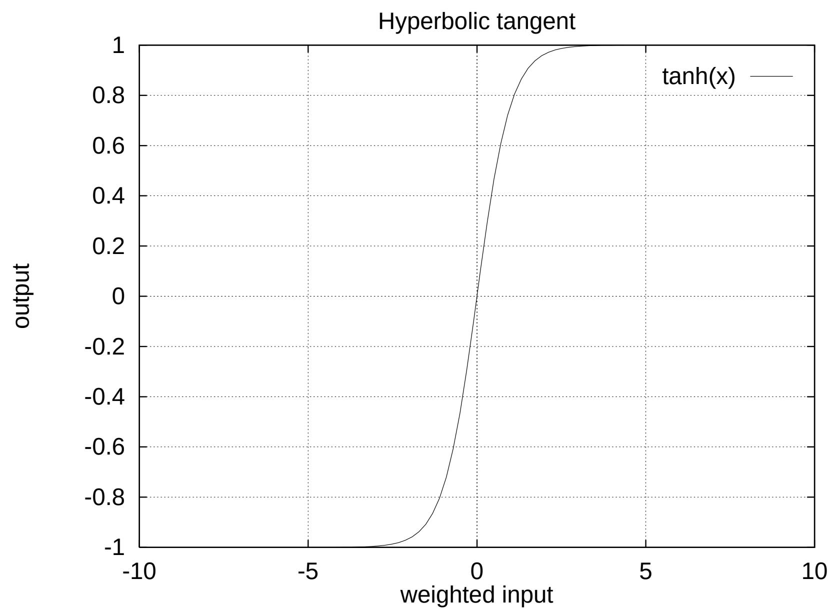
Given small enough η, incremental SG can approximate batch SG.

Summary

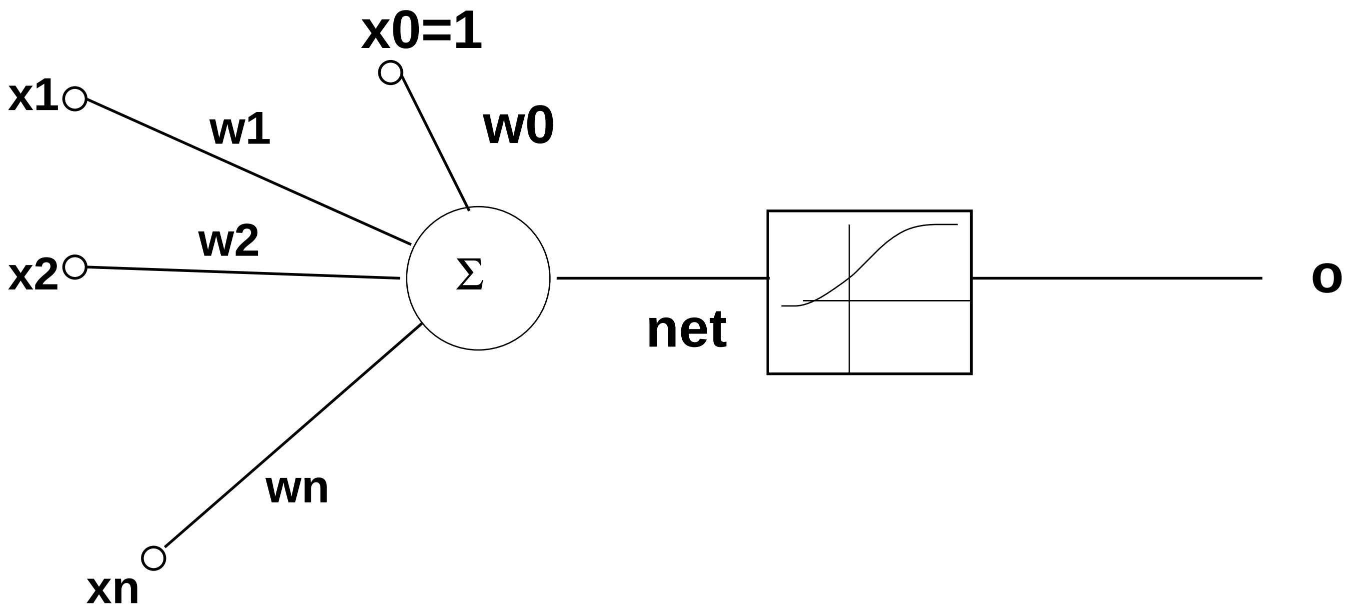
- Linear training unit uses gradient descent
- Guaranteed to converge to hypothesis with MSE
  - Provided learning rate η is sufficiently small
  - Even when training data is not describable in H
- Perceptron training rule guaranteed to succeed if
  - Training examples are linearly separable
  - Sufficiently small learning rate

Smooth Differentiable Units





Sigmoid Units



$$net = \sum_{i=0}^{n} w_i x_i$$

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

Training Sigmoid Networks

If $\sigma(x) = \frac{1}{1+e^{-x}}$

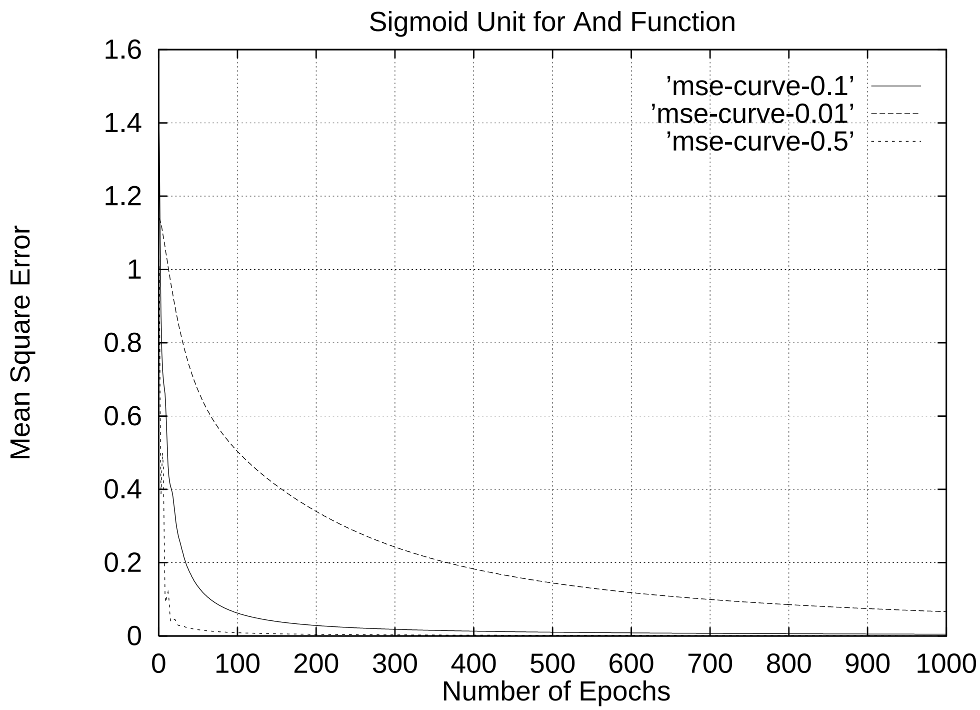
Note that

$$\frac{d\sigma(x)}{dx} = \sigma(x)(1 - \sigma(x))$$

Error gradient for sigmoid units:

$$\begin{aligned}
\frac{\partial E}{\partial w_i} &= \frac{\partial}{\partial w_i} \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2 \\
&= \frac{1}{2} \sum_d \frac{\partial}{\partial w_i} (t_d - o_d)^2 \\
&= -\sum_d (t_d - o_d) \frac{\partial o_d}{\partial net_d} \frac{\partial_d}{\partial w_i} \\
&= -\sum_d (t_d - o_d) o_d (1 - o_d) x_i^d
\end{aligned}$$

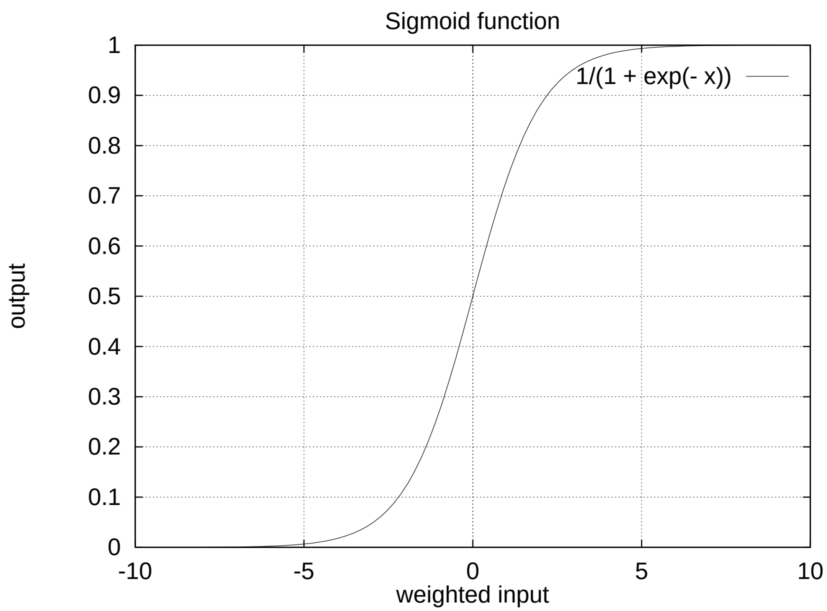
Learning the AND Function with a Sigmoid Unit

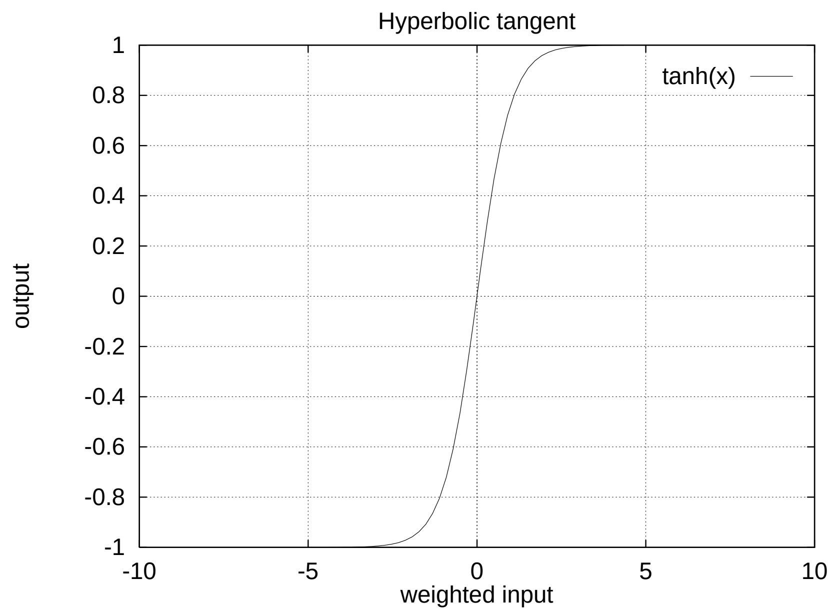


Limitations of Threshold and Perceptron Units

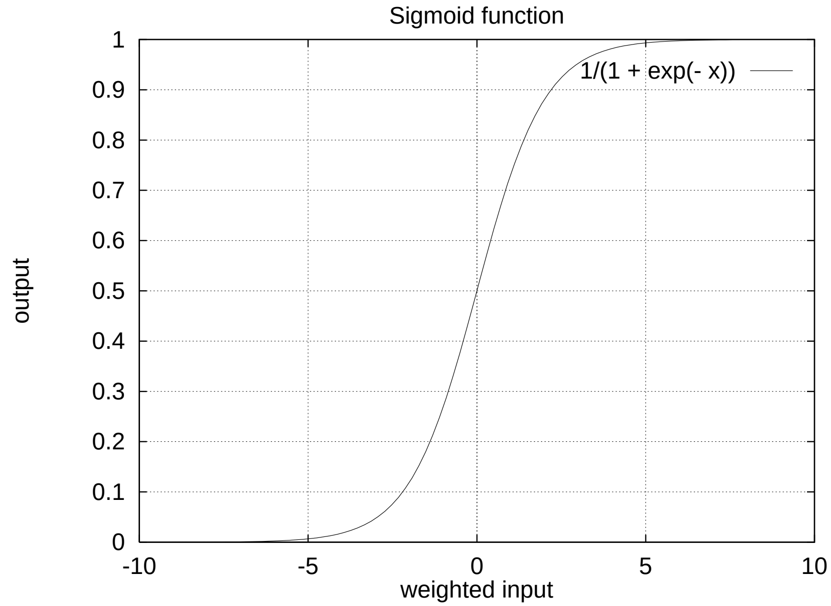
- Perceptrons can only learn linearly separable classes
- Perceptrons cycle if classes are not linearly separable
- Threshold units converge always to MSE hypothesis
- Network of perceptrons – how to train?
- Network of threshold units – not necessary! (why?)

Smooth Differentiable Units





Sigmoid Units



$$net = \sum_{i=0}^{n} w_i x_i$$

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

Training a Sigmoid Unit

If $\sigma(x) = \frac{1}{1+e^{-x}}$

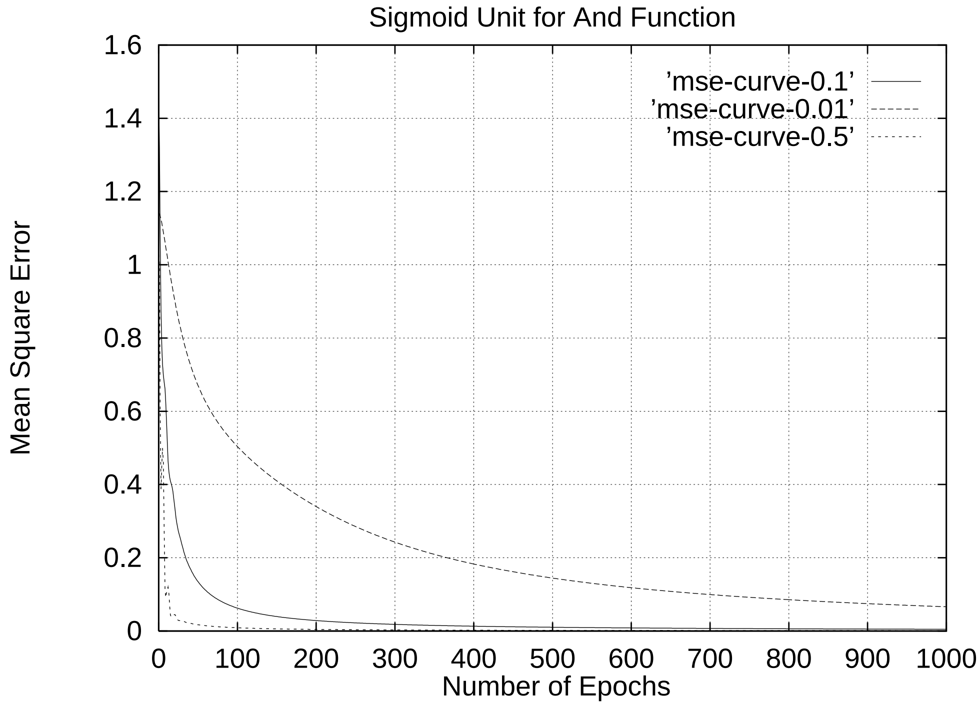
Note that

$$\frac{d\sigma(x)}{dx} = \sigma(x)(1 - \sigma(x))$$

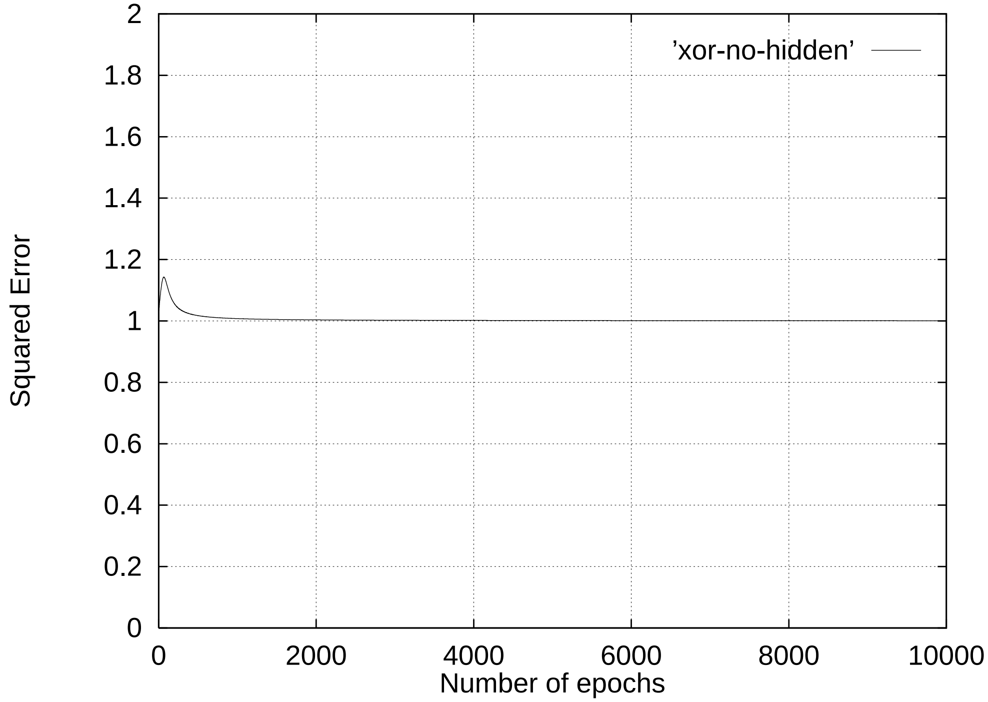
Error gradient for sigmoid unit:

$$\begin{aligned}
\frac{\partial E}{\partial w_i} &= \frac{\partial}{\partial w_i} \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2 \\
&= \frac{1}{2} \sum_d \frac{\partial}{\partial w_i} (t_d - o_d)^2 \\
&= -\sum_d (t_d - o_d) \frac{\partial o_d}{\partial net_d} \frac{\partial net_d}{\partial w_i} \\
&= -\sum_d (t_d - o_d) o_d (1 - o_d) x_i^d
\end{aligned}$$

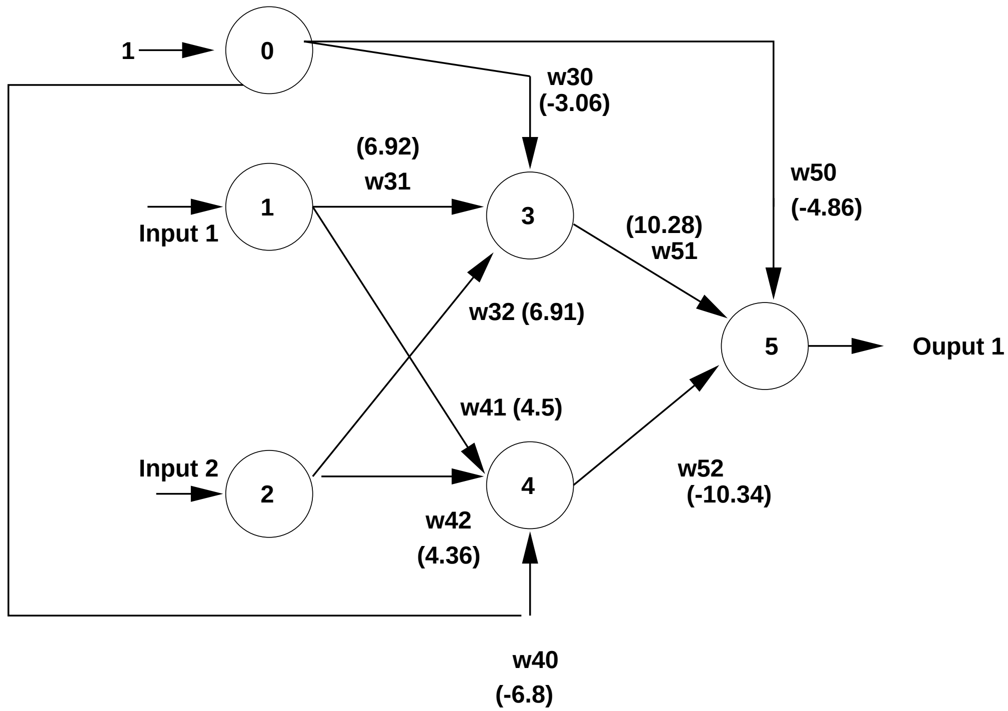
Learning the AND Function with a Sigmoid Unit



Cannot learn XOR Function with 1 sigmoid unit

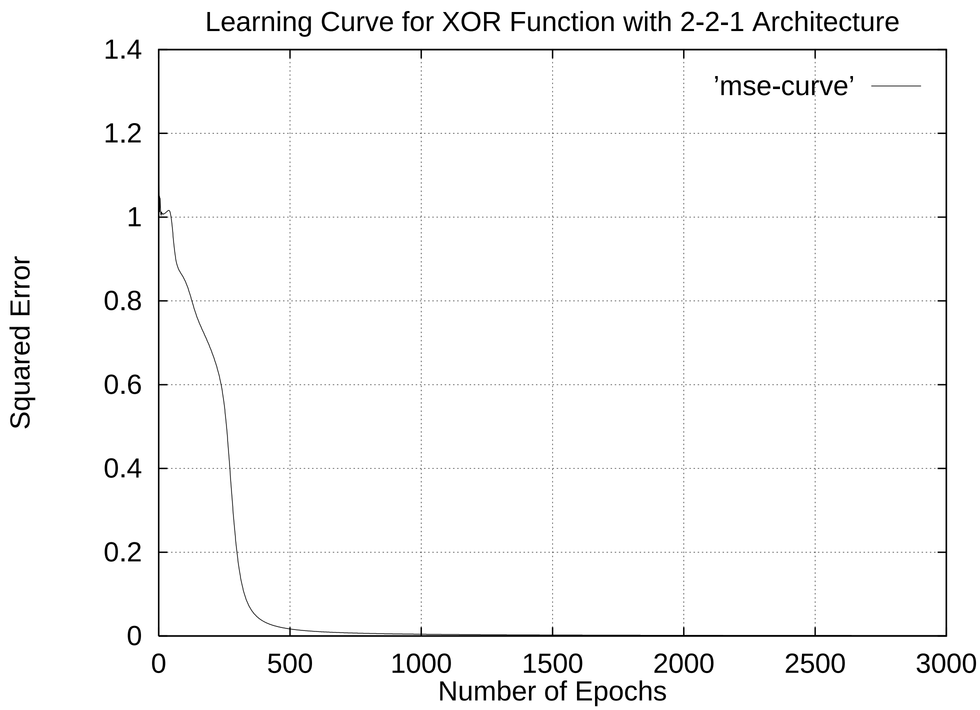


Computing XOR Function with A Feedforward Network



Input1	Input2	o3	o4	Ouput 1
0	0	0.04	0.001	0.011
0	1	0.98	0.08	0.99
1	0			
1	1			

Learning the XOR Function



The Backpropagation Algorithm: Batch Version

Initialize weights to small random values. Repeat until MSE < minimum. Repeat for every training example in data set

1. **Forward Propagation:** Input the training example to the net, and compute the network outputs.
2. **Backward Propagation:**
   - For each output unit k

$$\delta_k \leftarrow \delta_k + o_k(1 - o_k)(t_k - o_k)$$

   - For each hidden unit h

$$\delta_h \leftarrow \delta_h + o_h(1 - o_h) \sum_{k \in \text{Downstream}(h)} w_{kh}\delta_k$$

Update each network weight w_{ji} by $w_{ji} \Leftarrow w_{ji} + \eta\delta_j x_{ji}$

Stochastic Gradient Backpropagation

Initialize weights to small random values. Repeat until MSE < minimum. For each training example

1. **Forward Propagation:** Input the training example to the net, and compute the network outputs.
2. **Backward Propagation:**
   - For each output unit k

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

   - For each hidden unit h

     $$\delta_h \leftarrow o_h(1 - o_h) \sum_{k \in \text{Downstream}(h)} w_{kh}\delta_k$$

3. Update each network weight w_{ji} by

   $$w_{ji} \Leftarrow w_{ji} + \eta \delta_j x_{ji}$$

Derivation of the Backpropagation Algorithm

We need to determine how each weight w_{ji} affects the output of the network. Then, each weight is modified by

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

where E_d is the error on the training example d, summed over all outputs of the network

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

Define $net_j = \sum_i w_{ji} x_{ji}$. Note that the weight w_{ji} can only influence the network output via net_j. So, we can use the chain rule to get

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

Training Rule for Output Units

Using the chain rule again, we get $\frac{\partial E_d}{\partial net_j} = \frac{\partial E_d}{\partial o_j} \frac{\partial o_j}{\partial net_j} = -\delta_j$

For the first term:

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

For the 2nd term:

$$\begin{aligned}
\frac{\partial o_j}{\partial net_j} &= \frac{\partial}{\partial net_j} \frac{1}{1 + e^{-net_j}} \\
&= o_j(1 - o_j)
\end{aligned}$$

Combining these two, we get

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

Training Rule for Hidden Units

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

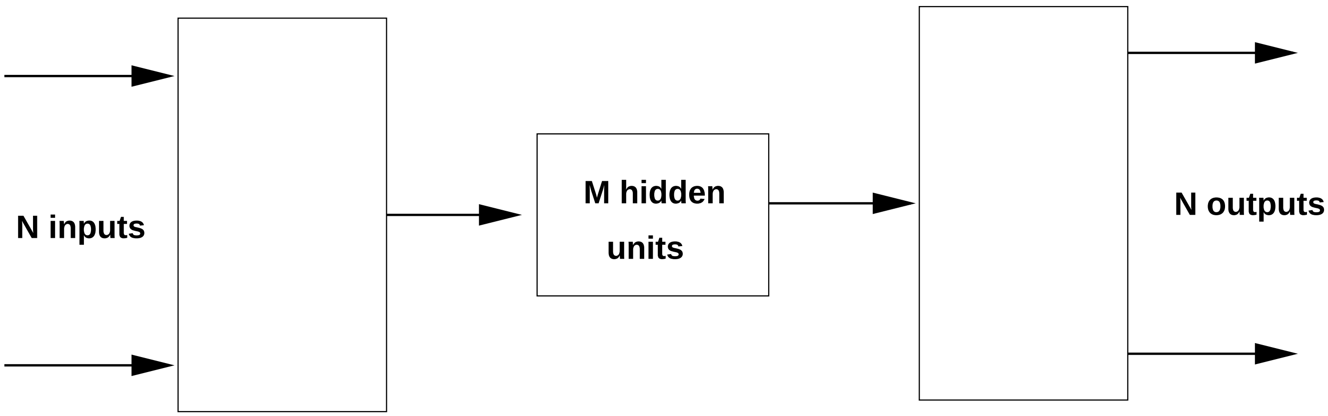
Some Practical Issues

- Convergence typically means the output of the desired unit is > 0.9 (if correct output is 1) or < 0.1 (if correct output is 0).
- Choice of initial weights impacts the convergence rate. One good heuristic when N is large is to choose weights randomly between $(-1/N, 1/N)$ where N is input size.
- Larger η can produce faster convergence, but may cause instability.
- It is usually useful to add a momentum term to the weight adjustment rule

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

- Choice of network topology (e.g. number of hidden units), input encoding, learning and momentum rates etc. are all important.

Learning the Encoder Function



Examples: N = 4, M = 2
N = 8, M = 3

Can we make M to be small enough to force the network to "discover" a clever encoding?

Hidden units "discover" binary encoding!

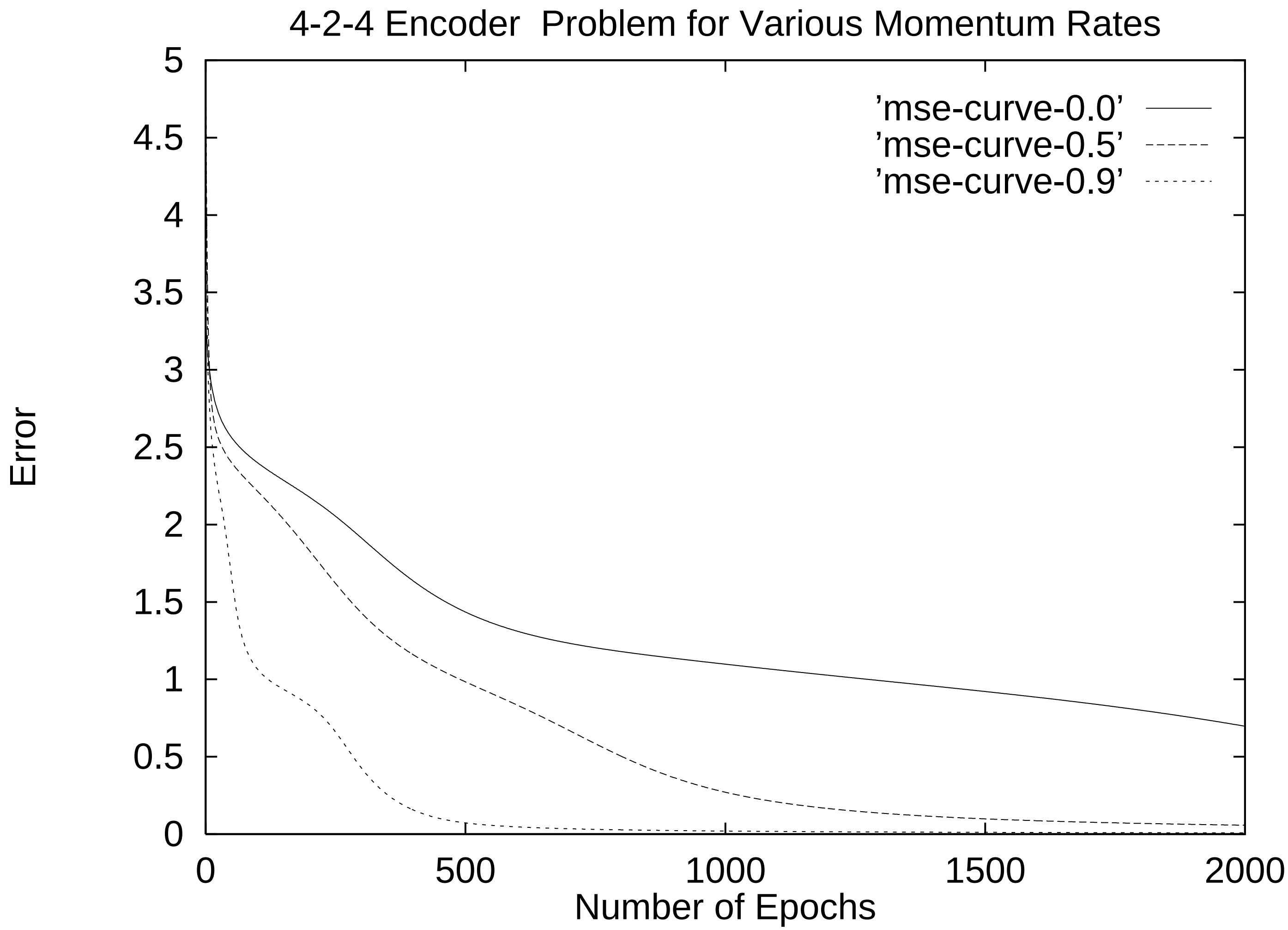
4-2-4 Network:

```
0.99  0.99   ->  0.99  0.01  0.01  0.00
0.01  0.98   ->  0.01  0.99  0.00  0.01
0.96  0.01   ->  0.02  0.00  0.99  0.01
0.01  0.01   ->  0.00  0.02  0.02  0.98
```

8-2-8 Network:

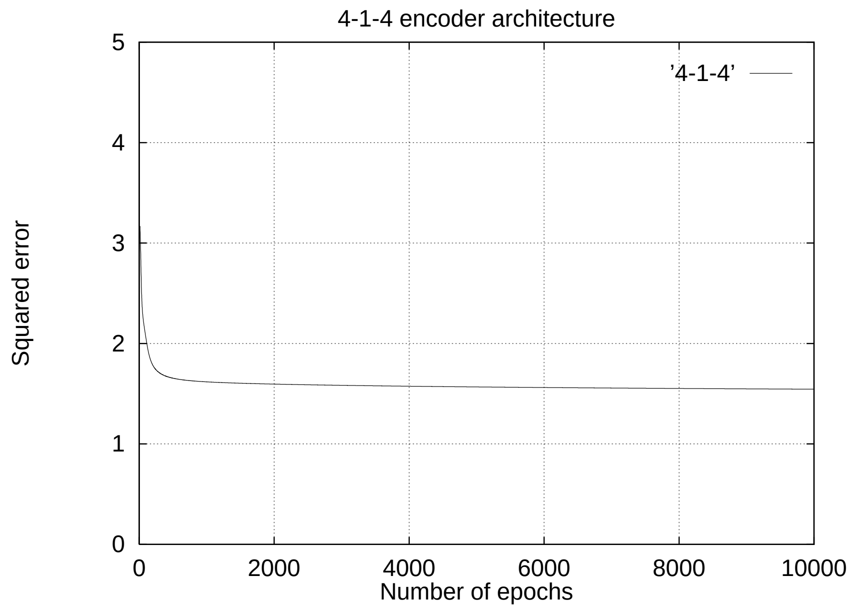
```
0.98  0.01  0.95   ->  0.97  0.00  0.00  0.03  0.02  0.00  0.00  0.01
0.02  0.00  0.18   ->  0.00  0.97  0.00  0.00  0.02  0.00  0.03  0.02
0.99  0.98  0.02   ->  0.00  0.00  0.97  0.02  0.00  0.00  0.02  0.02
0.99  0.99  0.99   ->  0.01  0.00  0.02  0.96  0.00  0.02  0.00  0.00
0.02  0.18  0.99   ->  0.02  0.02  0.00  0.00  0.97  0.02  0.00  0.00
0.01  0.99  0.76   ->  0.00  0.00  0.00  0.03  0.02  0.96  0.02  0.00
0.03  0.71  0.00   ->  0.00  0.02  0.02  0.00  0.00  0.03  0.97  0.00
0.96  0.04  0.01   ->  0.02  0.02  0.02  0.00  0.00  0.00  0.00  0.97
```

Learning the Encoder Function

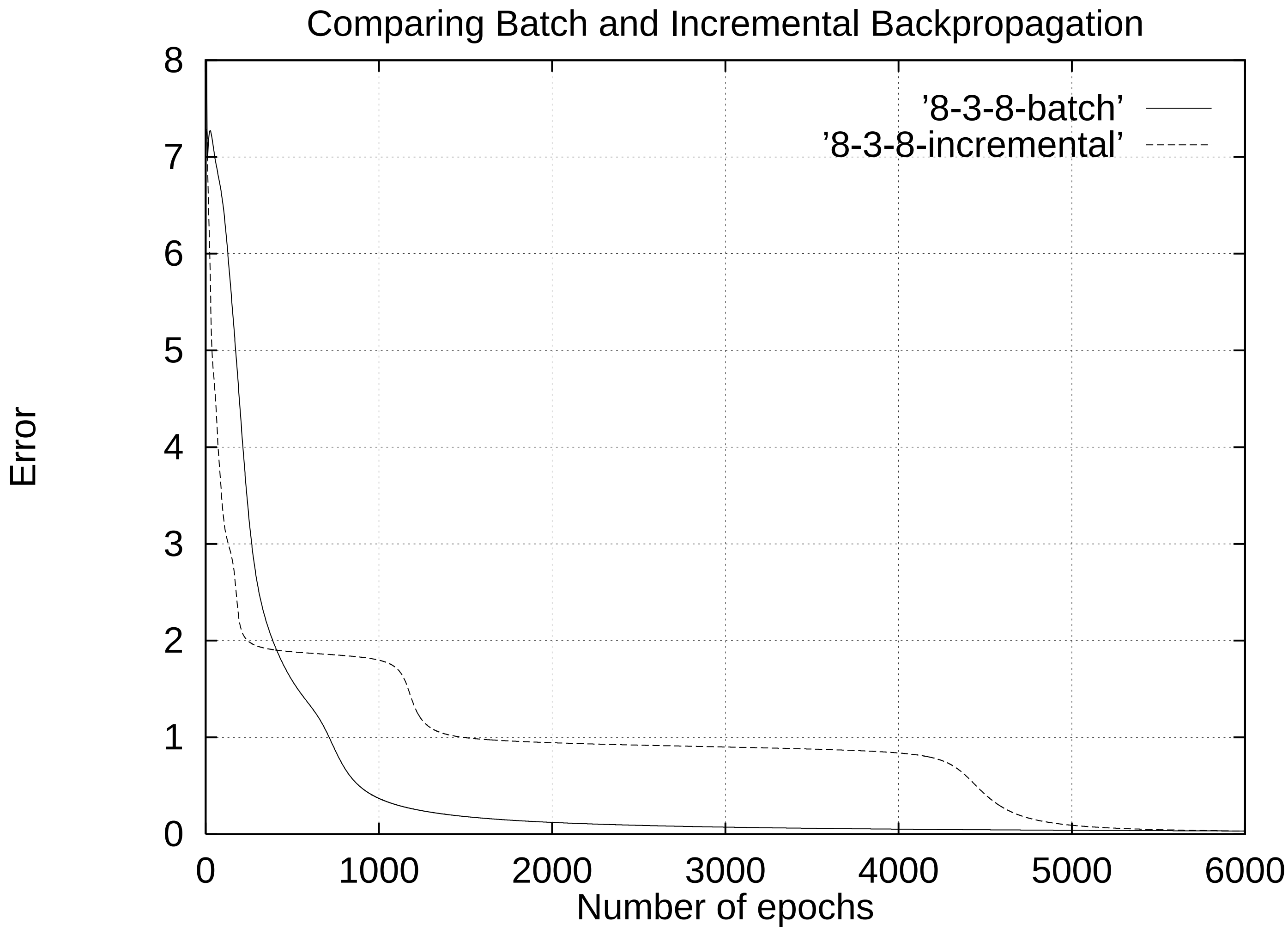


Is it possible to learn 4-1-4?

```
1.00  ->  0.99  0.00  0.11  0.07
0.00  ->  0.00  0.93  0.31  0.14
0.18  ->  0.01  0.08  0.26  0.12
0.23  ->  0.01  0.02  0.25  0.12
```



Batch vs. Incremental Backpropagation



Some Practical Successes of Backpropagation

- Learning pronounciations of English words (NETTALK).
- Handwritten character recognition of postal zip codes (AT & T).
- Driving an autonomous land vehicle (a Ford truck) at highway speeds (ALVINN).
- Recognizing spoken words (isolated speech) (Lang, Waibel, Hinton).
- Adaptive Optics (Arizona State).