

Intro. to machine learning (CSI 5325)

Lecture 8: neural networks

Greg Hamerly

Spring 2008

Some content from Tom Mitchell.

- 1 Gradient descent
- 2 Multilayer networks
- 3 Backpropagation
- 4 Hidden layer representations

Gradient Descent

Gradient-Descent(*training_examples*, η)

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

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

Summary

Perceptron training rule guaranteed to succeed if

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

Linear unit training rule 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

Incremental (Stochastic) Gradient Descent

Batch mode Gradient Descent:

Do until satisfied

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

Incremental mode Gradient Descent:

Do until satisfied

- For each training example \vec{x} in D
 - 1 Compute the gradient $\nabla E_{\vec{x}}[\vec{w}]$
 - 2 $\vec{w} \leftarrow \vec{w} - \eta \nabla E_{\vec{x}}[\vec{w}]$

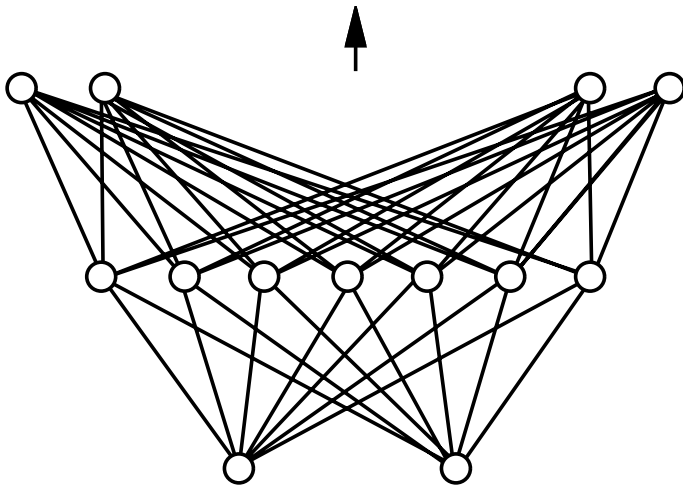
Incremental (Stochastic) Gradient Descent

$$E_D[\vec{w}] \equiv \frac{1}{2} \sum_{\vec{x} \in D} (t(\vec{x}) - o(\vec{x}))^2$$

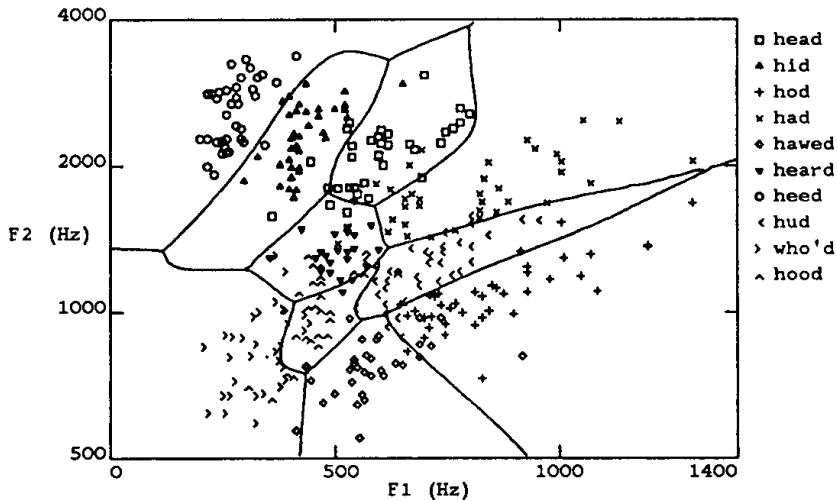
$$E_{\vec{x}}[\vec{w}] \equiv \frac{1}{2} (t(\vec{x}) - o(\vec{x}))^2$$

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

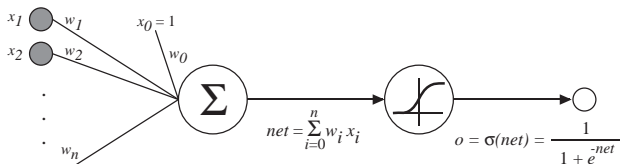
Multilayer network of sigmoid units (word identification)



Word identification – decision boundaries



Sigmoid unit



The sigmoid function is: $\sigma(x) = \frac{1}{1 + e^{-x}}$

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

We can derive gradient decent rules to train

- One sigmoid unit
- *Multilayer networks* of sigmoid units → Backpropagation

Error gradient for a single sigmoid unit

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

Where $net_d = \vec{w} \cdot \vec{x}_d$.

Error gradient for a single sigmoid unit

$$\frac{\partial E}{\partial w_i} = - \sum_d (t_d - o_d) \frac{\partial o_d}{\partial net_d} \frac{\partial net_d}{\partial w_i}$$

But we know:

$$\begin{aligned} \frac{\partial o_d}{\partial net_d} &= \frac{\partial \sigma(net_d)}{\partial net_d} = o_d(1 - o_d) \\ \frac{\partial net_d}{\partial w_i} &= \frac{\partial (\vec{w} \cdot \vec{x}_d)}{\partial w_i} = x_{id} \end{aligned}$$

So:

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

Backpropagation (multilayer network, stochastic version)

Initialize all weights to small random numbers.

Until satisfied, do

- For each training example, do
 - 1 Input the training example to the network and compute the network outputs
 - 2 For each output unit k ,

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

- 3 For each hidden unit h

$$\delta_h \leftarrow o_h(1 - o_h) \sum_{k \in \text{outputs}} w_{hk} \delta_k$$

- 4 Update each network weight w_{ij}

$$w_{ij} \leftarrow w_{ij} + \Delta w_{ij} \quad \text{where} \quad \Delta w_{ij} = \eta \delta_j x_{ij}$$

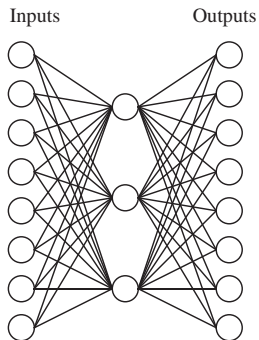
More on Backpropagation

- 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 run multiple times)
- Often include weight *momentum* α

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

- Minimizes error over *training* examples
 - Will it generalize well to subsequent examples?
- Training can take thousands of iterations \rightarrow slow!
- Using network after training is very fast

Learning Hidden Layer Representations



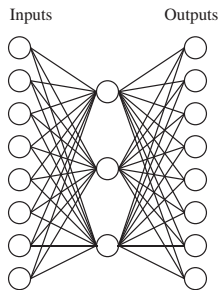
A target function:

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

Can this be learned??

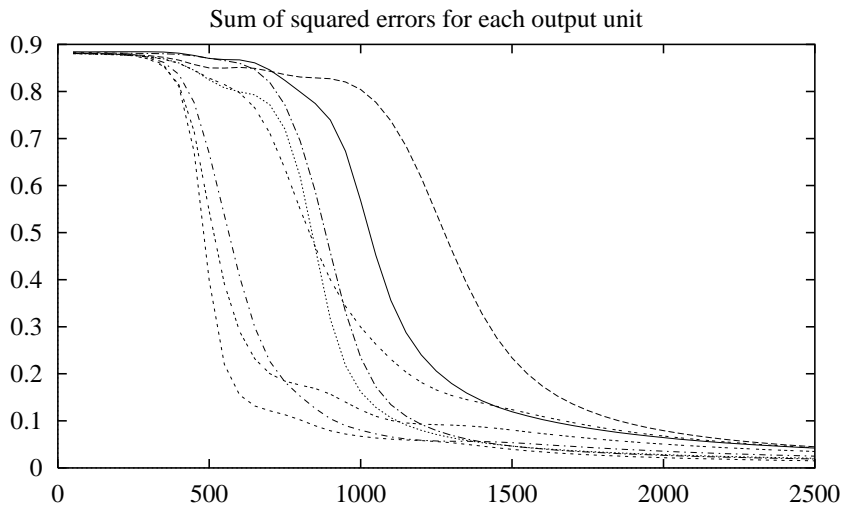
Learning Hidden Layer Representations

Learned hidden layer representation:

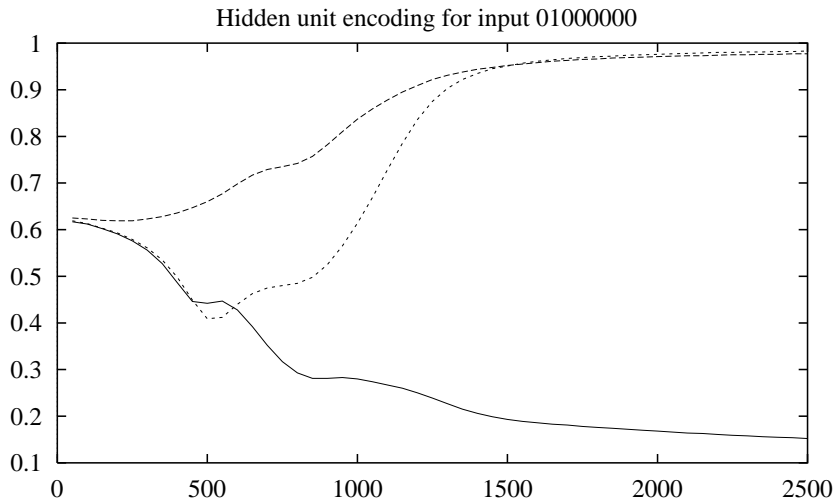


Input		Hidden Values				Output
10000000	→	.89	.04	.08	→	10000000
01000000	→	.01	.11	.88	→	01000000
00100000	→	.01	.97	.27	→	00100000
00010000	→	.99	.97	.71	→	00010000
00001000	→	.03	.05	.02	→	00001000
00000100	→	.22	.99	.99	→	00000100
00000010	→	.80	.01	.98	→	00000010
00000001	→	.60	.94	.01	→	00000001

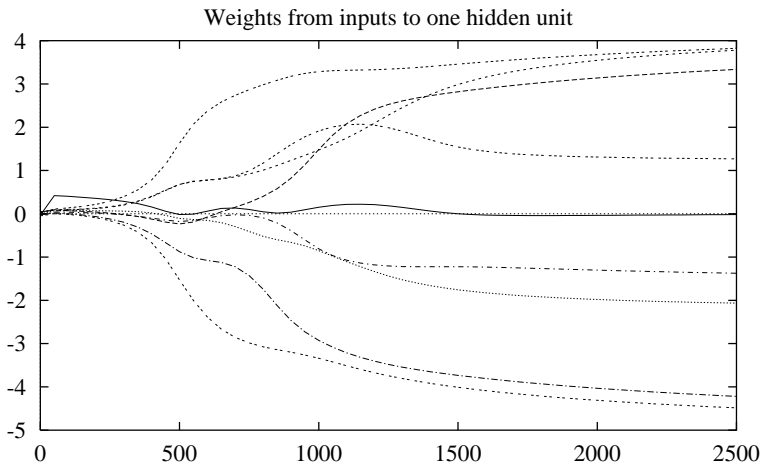
Training → reduction in output error



Training → specialization of hidden units



Training → learning of weights



Convergence of Backpropagation

Gradient descent to some local minimum

- Perhaps not global minimum...
- Add momentum
- Stochastic gradient descent
- Train multiple nets with different initial weights

Nature of convergence

- Initialize weights near zero
- Therefore, initial networks near-linear
- Increasingly non-linear functions possible as training progresses

Stochastic versus batch backpropagation

The algorithm given is stochastic backprop.

There are some potential problems with stochastic updates:

- cycling behavior
- order of examples will affect the training (can be mitigated with randomization)

Backpropagation in batch solves these problems:

- cycle over all inputs first to sum up all weight updates
- apply weight updates

...but it has its own problems:

- may take longer to learn
- may be more likely to fall into local minima

Expressive Capabilities of ANNs

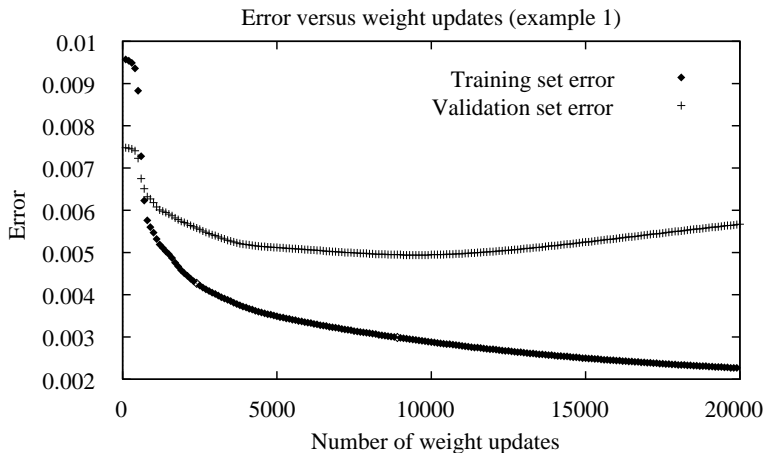
Boolean functions:

- Every boolean function can be represented by network with single hidden layer
- but might require exponential (in number of inputs) hidden units

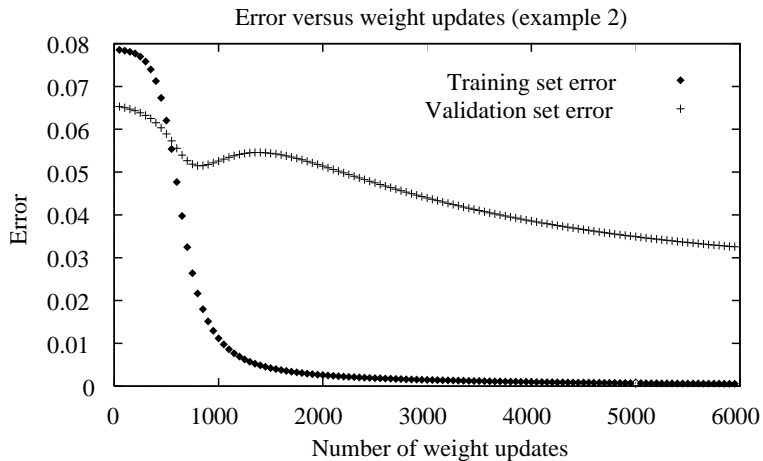
Continuous functions:

- Every bounded continuous function can be approximated with arbitrarily small error, by network with one hidden layer [Cybenko 1989; Hornik et al. 1989]
- Any function can be approximated to arbitrary accuracy by a network with two hidden layers [Cybenko 1988].

Overfitting in ANNs



Overfitting in ANNs



Ways of dealing with overfitting

- validation set
- stochastic versus batch updates
- momentum
- early stopping
- random restarts
 - choose best network
 - or keep all networks and use average or majority vote
- ...or some combination of these approaches