
Chapter 4: Artificial Neural Networks

CS 536: Machine Learning
Littman (Wu, TA)

Administration

iCML-03: instructional Conference on
Machine Learning

<http://www.cs.rutgers.edu/~mlittman/courses/ml03/iCML03/>

Weka assignment

<http://www.cs.rutgers.edu/~mlittman/courses/ml03/hw1.pdf>

Artificial Neural Networks

[Read Ch. 4]

[Review exercises 4.1, 4.2, 4.5, 4.9, 4.11]

- Threshold units
- Gradient descent
- Multilayer networks
- Backpropagation
- Hidden layer representations
- Example: Face Recognition
- Advanced topics

Connectionist Models

Consider humans:

- Neuron switching time $\sim .001$ second
 - Number of neurons $\sim 10^{10}$
 - Connections per neuron $\sim 10^{4-5}$
 - Scene recognition time $\sim .1$ second
 - 100 inference steps doesn't seem like enough
- much parallel computation

Artificial Networks

Properties of artificial neural nets (ANNs):

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

When to Consider ANNs

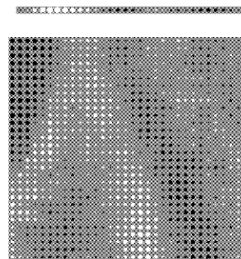
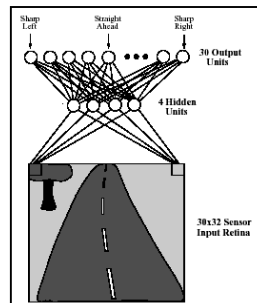
- Input is high-dimensional discrete or real-valued (e.g. raw sensor input)
- Output is discrete or real valued
- Output is a vector of values
- Possibly noisy data
- Form of target function is unknown
- Human readability of result is unimportant

ANNs: Example Uses

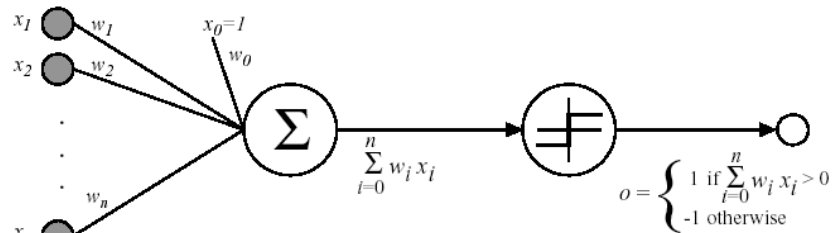
Examples:

- Speech phoneme recognition [Waibel]
- Image classification [Kanade, Baluja, Rowley]
- Financial prediction
- Backgammon [Tesauro]

ALVINN drives on highways



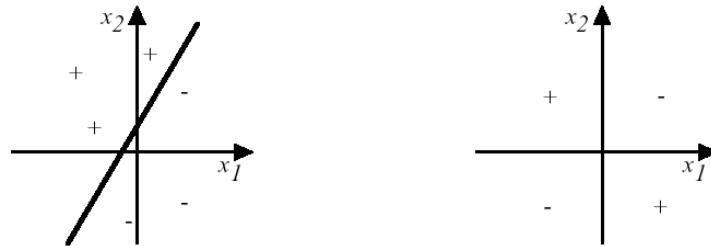
Perceptron



$$o(x_1, \dots, x_n) = \begin{cases} 1 & \text{if } w_0 + w_1 x_1 + \dots + w_n x_n > 0 \\ -1 & \text{otherwise.} \end{cases}$$

Or, more succinctly: $o(\mathbf{x}) = \text{sgn}(\mathbf{w} \cdot \mathbf{x})$

Perceptron Decision Surface



A single unit can represent some useful functions

- What weights represent $g(x_1, x_2) = \text{AND}(x_1, x_2)$?

But some functions not representable

- e.g., not linearly separable
- Therefore, we'll want networks of these...

Perceptron training rule

$$w_i \leftarrow w_i + \eta w_i$$

where

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

Where:

- $t = c(\mathbf{x})$ is target value
- o is perceptron output
- η is small constant (e.g., .1) called the *learning rate* (or *step size*)

Perceptron training rule

Can prove it will converge

- If training data is linearly separable
- and η sufficiently small

Gradient Descent

To understand, consider simpler *linear unit*, where

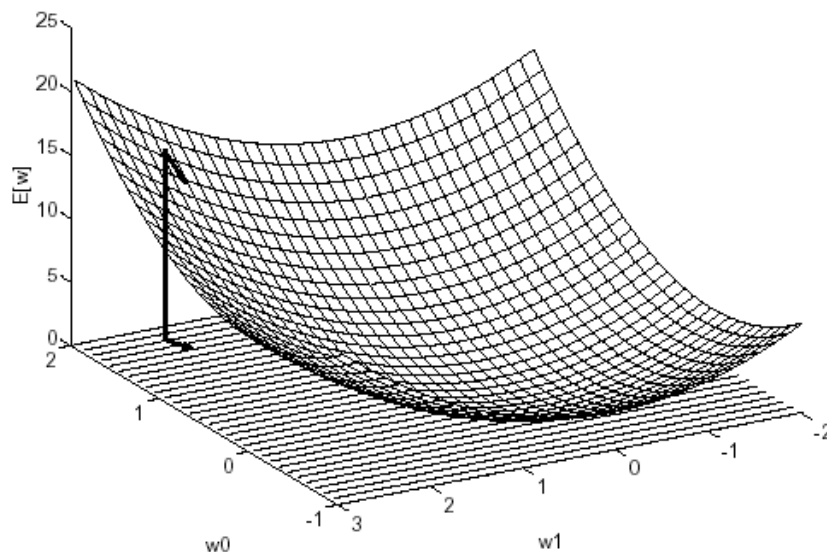
$$o = w_0 + w_1x_1 + \dots + w_nx_n$$

Let's learn w_i 's to minimize squared error

$$E[\mathbf{w}] \equiv 1/2 \sum_{d \in D} (t_d - o_d)^2$$

Where D is set of training examples

Error Surface



Gradient Descent

Gradient

$$\nabla E[\mathbf{w}] = [\partial E / \partial w_0, \partial E / \partial w_1, \dots, \partial E / \partial w_n]$$

Training rule:

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

in other words:

$$\Delta w_i = -\eta \partial E / \partial w_i$$

Gradient of Error

$$\partial E / \partial w_i$$

$$= \partial / \partial w_i \frac{1}{2} \sum_d (t_d - o_d)^2$$

$$= \frac{1}{2} \sum_d \partial / \partial w_i (t_d - o_d)^2$$

$$= \frac{1}{2} \sum_d 2 (t_d - o_d) \partial / \partial w_i (t_d - o_d)$$

$$= \sum_d (t_d - o_d) \partial / \partial w_i (t_d - \mathbf{w} \cdot \mathbf{x}_d)$$

$$= \sum_d (t_d - o_d) (-x_{i,d})$$

Gradient Descent Code

GRADIENT-DESCENT(training examples, η)

Each training example is a pair of the form $\langle \mathbf{x}, t \rangle$, where \mathbf{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 $\langle \mathbf{x}, t \rangle$ in training examples, Do
 - Input the instance \mathbf{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 will succeed if

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

Linear unit training 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 H separable

Stochastic Gradient Descent

Batch mode Gradient Descent:

Do until satisfied

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

Incremental mode Gradient Descent:

Do until satisfied

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

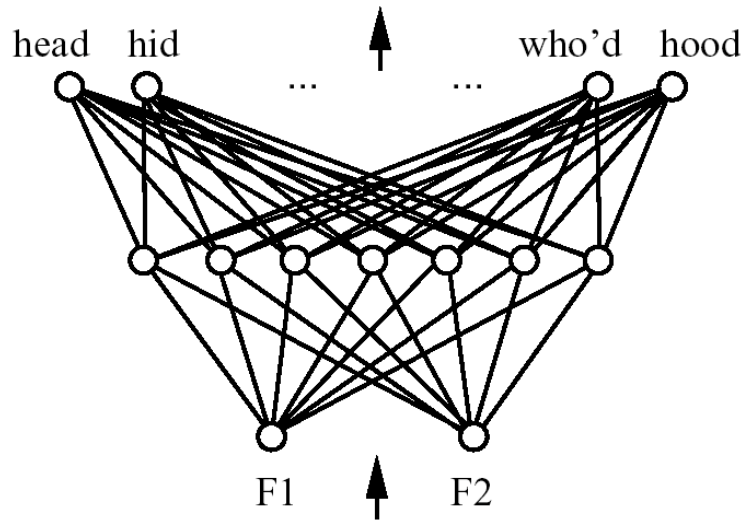
More Stochastic Grad. Desc.

$$E_D[\mathbf{w}] \equiv 1/2 \sum_{d \in D} (t_d - o_d)^2$$

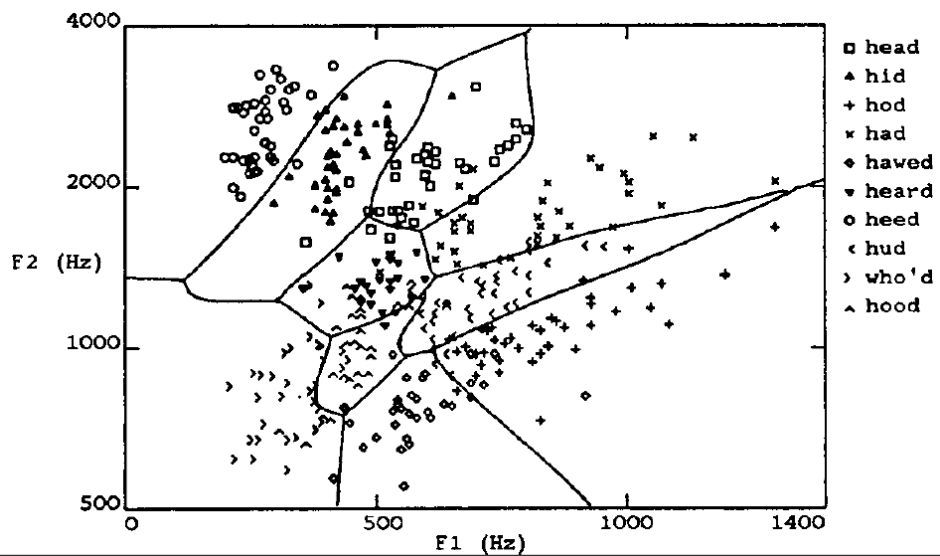
$$E_d[\mathbf{w}] \equiv 1/2 (t_d - o_d)^2$$

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

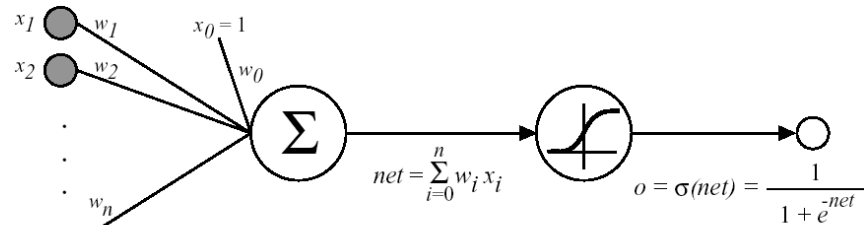
Multilayer Networks



Decision Boundaries



Sigmoid Unit



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

Derivates of Sigmoids

Nice property:

$$d \sigma(x) / dx = \sigma(x) (1 - \sigma(x))$$

We can derive gradient decent rules to train

- One sigmoid unit
- *Multilayer networks* of sigmoid units \square *Backpropagation*

Error Gradient for Sigmoid

$$\begin{aligned}\partial E / \partial w_i &= \partial / \partial w_i \frac{1}{2} \sum_d (t_d - o_d)^2 \\ &= \frac{1}{2} \sum_d \partial / \partial w_i (t_d - o_d)^2 \\ &= \frac{1}{2} \sum_d 2 (t_d - o_d) \partial / \partial w_i (t_d - o_d) \\ &= \sum_d (t_d - o_d) (-\partial o_d / \partial w_i) \\ &= - \sum_d (t_d - o_d) (\partial o_d / \partial net_d \partial net_d / \partial w_i)\end{aligned}$$

Even more...

But we know:

$$\begin{aligned}\partial o_d / \partial net_d &= \partial \sigma(net_d) / \partial net_d = o_d (1 - o_d) \\ \partial net_d / \partial w_i &= \partial (\mathbf{w} \cdot \mathbf{x}_d) / \partial w_i = x_{i,d}\end{aligned}$$

So:

$$\partial E / \partial w_i = - \sum_d (t_d - o_d) o_d (1 - o_d) x_{i,d}$$

Backpropagation Algorithm

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
$$\delta_k = o_k(1 - o_k)(t_k - o_k)$$
 3. For each hidden unit h
$$\delta_h = o_h(1 - o_h) \sum_{k \text{ in outputs}} w_{h,k} \delta_k$$
 4. Update each network weight $w_{i,j}$
$$w_{i,j} \leftarrow w_{i,j} + \delta w_{i,j} \text{ where } \delta w_{i,j} = \delta_j x_{i,j}$$

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)

More more

- Often include weight *momentum* α
$$\alpha w_{i,j}(n) = \alpha \alpha_j x_{i,j} + \alpha \alpha w_{i,j}(n-1)$$
- Minimizes error over training examples
 - Will it generalize well to subsequent examples?
- Training can take thousands of iterations
 - α slow!
- Using network after training is very fast