Introduction to Machine Learning: CS 436/580L

Perceptrons

Instructor: Arti Ramesh

Binghamton University

Administrivia

- HW1 grades are out
  - One week for disputes
  - All disputes go to TAs first
- HW 2 Questions
  - Filter special characters
  - Normalize words as much as possible
- Quiz on Thursday, Oct 19th
  - 10 – 20 mins
- Midterm Nov 2nd
- Big Quiz thoughts

Who needs probabilities?

- Previously: model data with distributions
- Joint: P(X,Y)
  - e.g. Naïve Bayes
- Conditional: P(Y|X)
  - e.g. Logistic Regression
- But wait, why probabilities?
- Lets try to be error-driven!

mpg	cylinders	displacemen	horsepower	weight	acceleration	modelyear	maker
good	4	97	75	2265	18.2	77	asia
bad	6	199	90	2648	15	70	america
bad	4	121	110	2600	12.8	77	europe
bad	8	350	175	4100	13	73	america
bad	6	198	95	3102	16.5	74	america
bad	4	108	94	2379	16.5	73	asia
bad	4	113	95	2228	14	71	asia
bad	8	302	139	3570	12.8	78	america
:	:	:	:	:	:	:	:
:	:	:	:	:	:	:	:
:	:	:	:	:	:	:	:
good	4	120	79	2625	18.6	82	america
bad	8	455	225	4425	10	70	america
good	4	107	86	2464	15.5	76	europe
bad	5	131	103	2830	15.9	78	europe

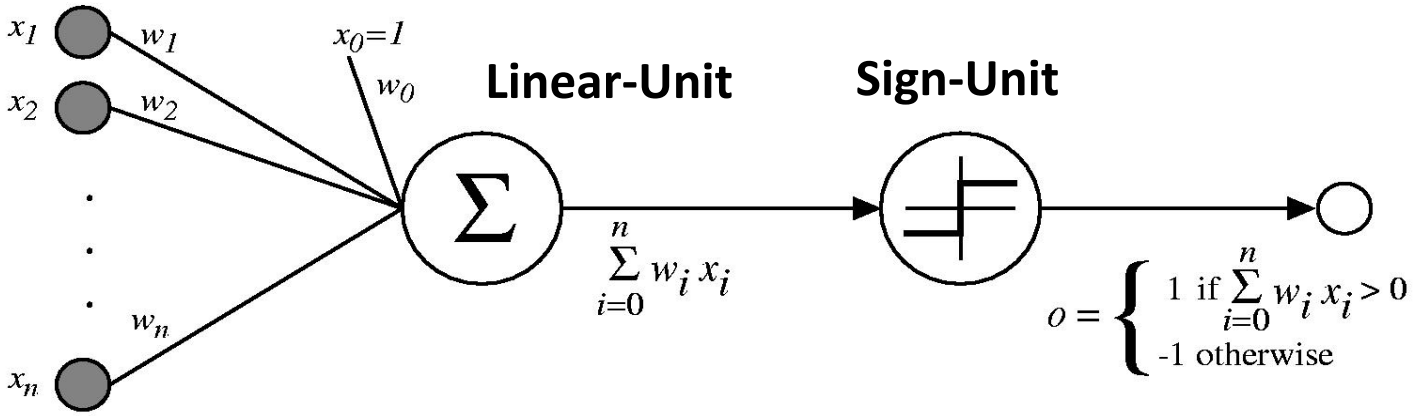
Connectionist Models

- Consider humans:
  - Neuron switching time 0.001 seconds
  - Number of neurons 10^{10}
  - Connections per neuron 10^{4-5}
  - Scene recognition time 0.1 second
  - 100 inference steps doesn't seem like enough
  - Much parallel computation needed

Properties of Neural Nets

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

Perceptron



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

Sometimes we'll use simpler vector notation:

$$o(\vec{x}) = \begin{cases} 1 & \text{if } \vec{w} \cdot \vec{x} > 0 \\ -1 & \text{otherwise.} \end{cases}$$

Linear Classifiers

- Inputs are **feature values** $(x_1, x_2, x_3, \ldots x_n)$
- Each feature has a **weight** $(w_1, w_2, w_3, \ldots w_n)$
- Sum is activation

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

  - w_0 : bias, $x_0 = 1$
- If the activation is
  - Positive, output class = +ve
  - Negative, output class = -ve

Example: Spam

- Imagine 3 features (spam is "positive" class):
  - free (number of occurrences of "free")
  - money (occurrences of "money")
  - BIAS (intercept, always has value 1)

$\vec{w} \cdot \vec{x}$

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

"free money"

x

```
BIAS  :  1
free  :  1
money :  1
...
```

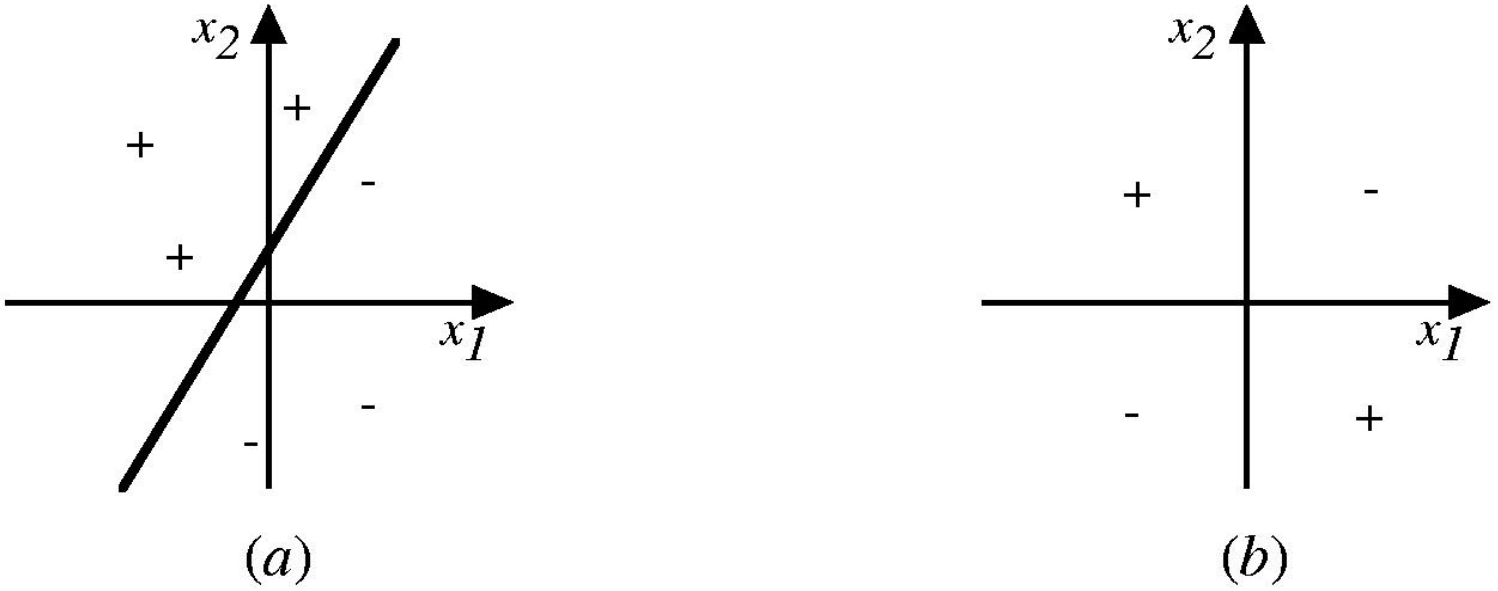
w

```
BIAS  : -3
free  :  4
money :  2
...
```

$(1)(-3) +$
$(1)(4) +$
$(1)(2) +$
$\ldots$
$= 3$

$\vec{w} \cdot \vec{x} > 0$ ➔ SPAM!!!

Decision Surface of a Perceptron



- Represents some useful functions
  - What weights represent $g(x_1, x_2) = AND(x_1, x_2)$
- But some functions are not representable
  - All not linearly separable
  - Therefore, need multi-layer network of perceptrons

Perceptron Training Rule

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

where

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

Where:

- $t = c(\vec{x})$ is target value
- o is perceptron output
- η is small constant (e.g., 0.1) called *learning rate*

Perceptron Training Rule

- Converges if the data is linearly separable
  - Provided the learning rate is sufficiently small
  - Proof on myCourses
- Convergence is not assured if data is not linearly separable
  - In fact in many cases, it will not converge
- Can we use some other algorithm to guarantee convergence?
  - YES!! – Gradient Descent
  - Gradient Descent yields a new rule for learning called **the Delta rule**

Delta Rule

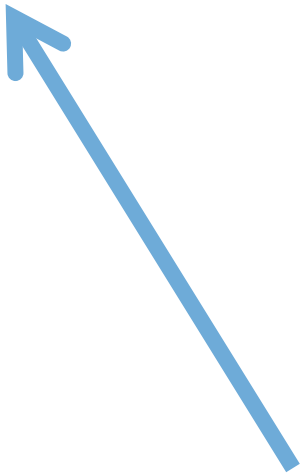
To understand, consider simpler *linear unit*, where

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

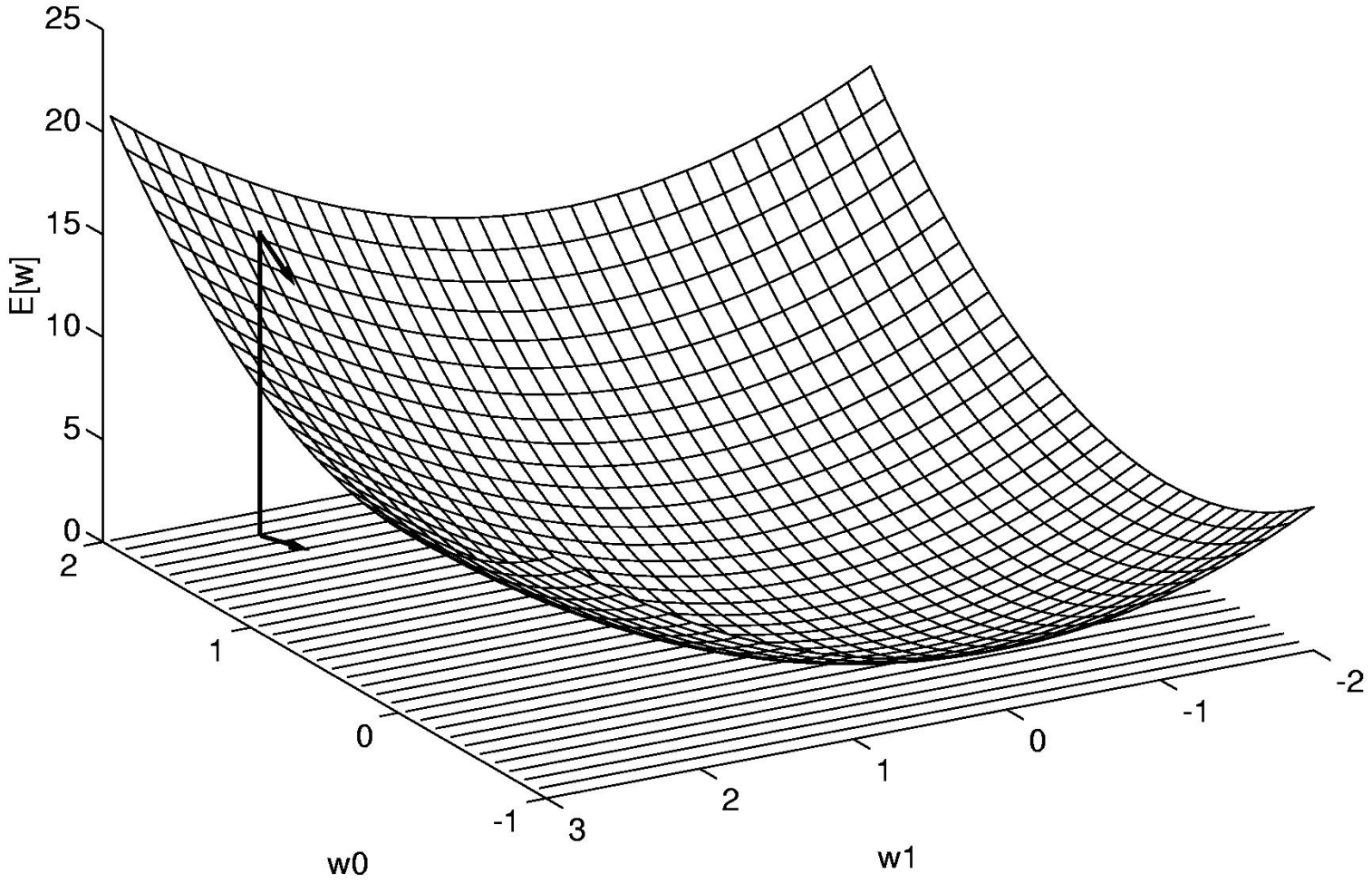
Let's learn w_i's that minimize the squared error

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

Where D is set of training examples



Gradient Descent



Gradient Descent Update

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

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

Gradient Descent Update

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.:

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

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

Gradient Descent

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

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 instance $\vec{x}$ to unit and compute 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

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

Batch Gradient Descent

Batch vs. Incremental Gradient Descent

Batch Mode Gradient Descent:

Do until convergence

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

Incremental Mode Gradient Descent:

Do until convergence

For each training example d in D

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

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

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

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

Stochastic Gradient Descent

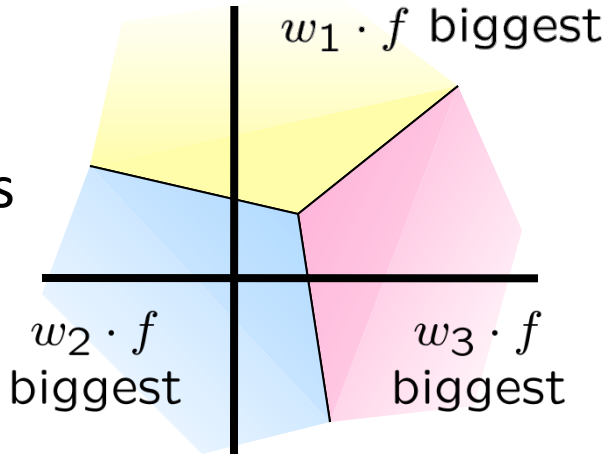
- Standard Gradient Descent
  - Error summed across all training examples
  - Computationally expensive than the stochastic version
- Stochastic Gradient Descent
  - Weights updated upon examining each training example
  - Avoid falling to local minima as it uses the error per training example

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

Multiclass Decision Rule

- If we have more than two classes:
  - Have a weight vector for each class:
  - Calculate an activation for each class



  - Highest activation wins

$$\text{activation}_w(x, y) = w_y \cdot f(x)$$

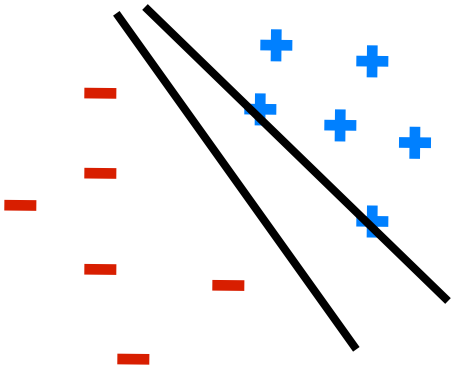
$$y = \arg\max_y \ (\text{activation}_w(x, y))$$

Properties of Perceptrons

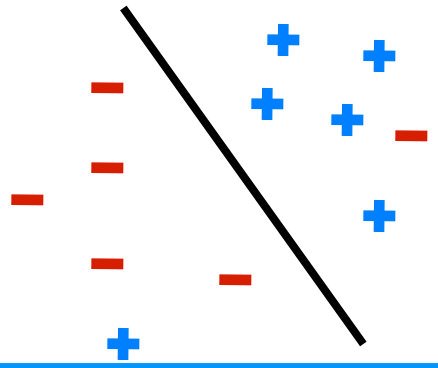
- Separability: some parameters get the training set perfectly correct

- Convergence: if the training is separable, perceptron will eventually converge (binary case)

Separable

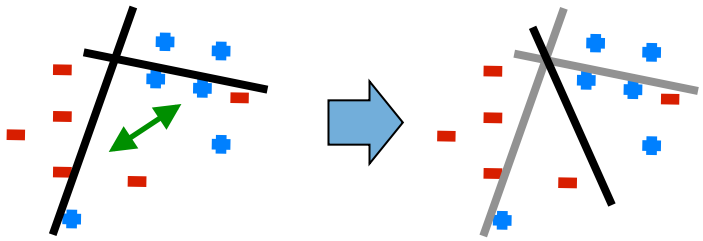


Non-Separable

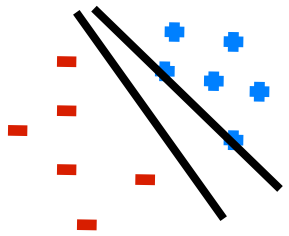


Problems with the Perceptron

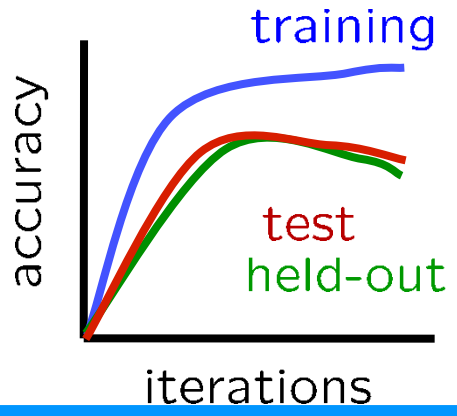
- Noise: if the data isn't separable, weights might thrash
  - Averaging weight vectors over time can help (averaged perceptron)



- Mediocre generalization: finds a "barely" separating solution

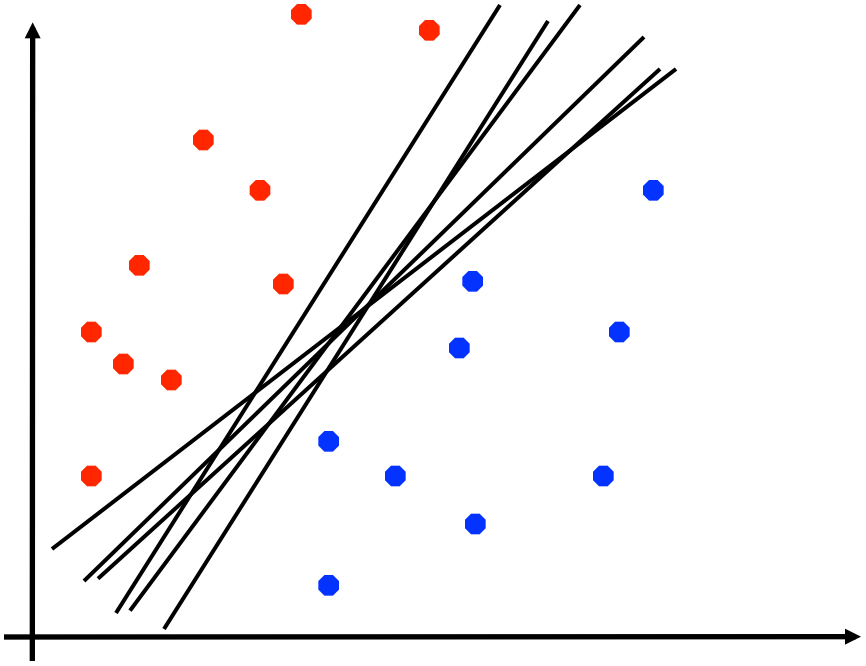


- Overtraining: test / validation accuracy usually rises, then falls
  - Overtraining is a kind of overfitting



Linear Separators

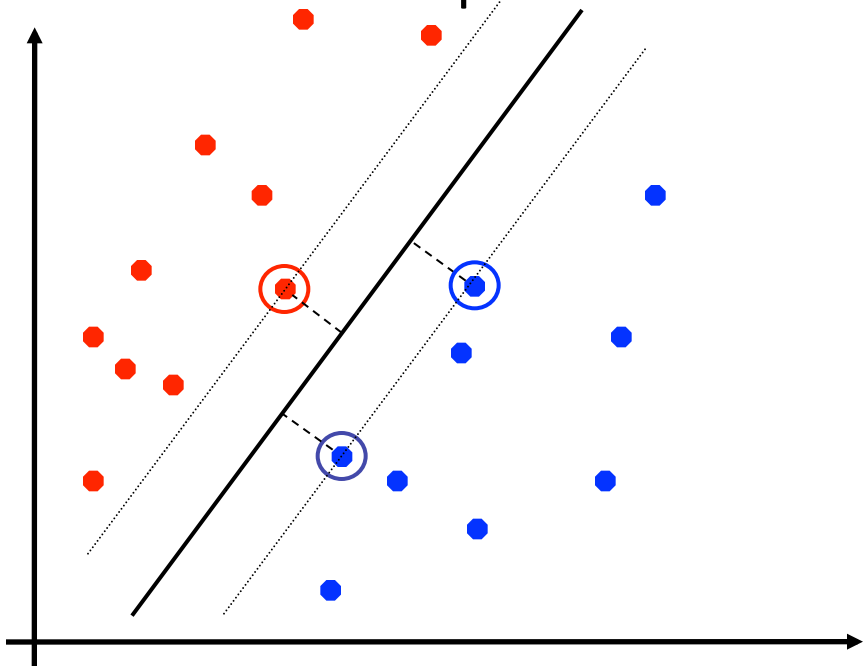
- Which of these linear separators is optimal?



Support Vector Machines

- Maximizing the margin: good according to intuition, theory, practice

- SVMs find the separator with max margin

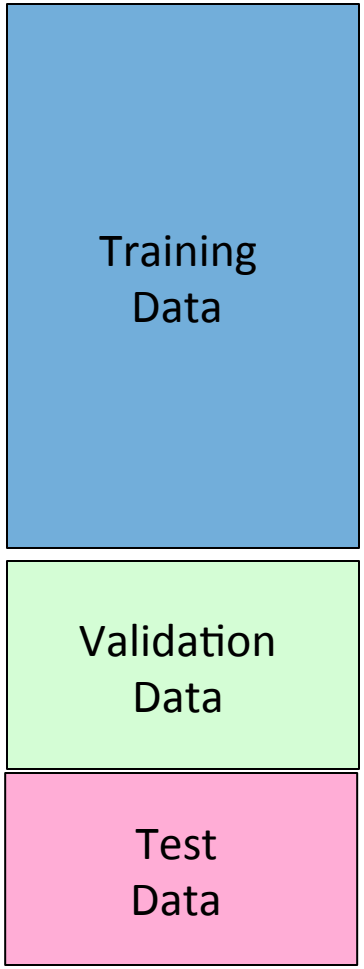


SVM

$$\min_w \frac{1}{2}||w||^2$$

$$\forall i, y \; w_{y^*} \cdot f(x_i) \geq w_y \cdot f(x_i) + 1$$

Three Views of Classification



- Naïve Bayes:
  - Parameters from data statistics
  - Parameters: probabilistic interpretation
  - Training: one pass through the data
- Logistic Regression:
  - Parameters from gradient ascent
  - Parameters: linear, probabilistic model, and discriminative
  - Training: one pass through the data per gradient step; regularization essential
- The Perceptron:
  - Parameters from reactions to mistakes
  - Parameters: discriminative interpretation
  - Training: go through the data until accuracy on validation set maxes out

Compare Algorithms So Far . . .

- Assumptions
- Kind of data
- Can it handle missing data?
- Can you generate data?
- What is training?
- Can you do multi-class?
- Disadvantages/ Advantages
- Noise
- Overfitting

The Multi-class Perceptron Alg.

- Start with zero weights
- Iterate training examples
  - Classify with current weights

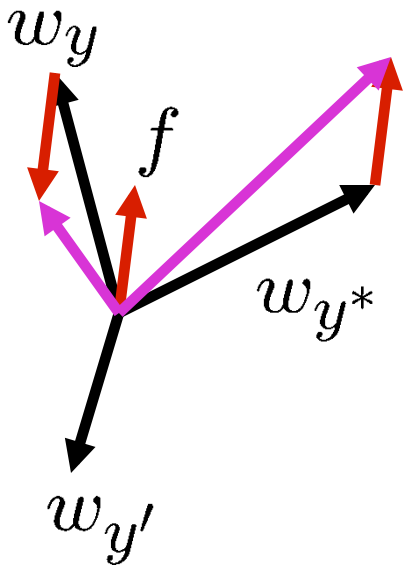
$$y = \arg\max_y \; w_y \cdot f(x)$$

$$= \arg\max_y \; \sum_i w_{y,i} \cdot f_i(x)$$

  - If correct, no change!
  - If wrong: lower score of wrong answer, raise score of right answer

$$w_y = w_y - f(x)$$

$$w_{y^*} = w_{y^*} + f(x)$$



From Logistic Regression to the Perceptron: 2 easy steps!

- Logistic Regression: (in vector notation): y is {0,1}

$$w = w + \eta \sum_j [y_j^* - p(y_j^*|x_j, w)] f(x_j)$$

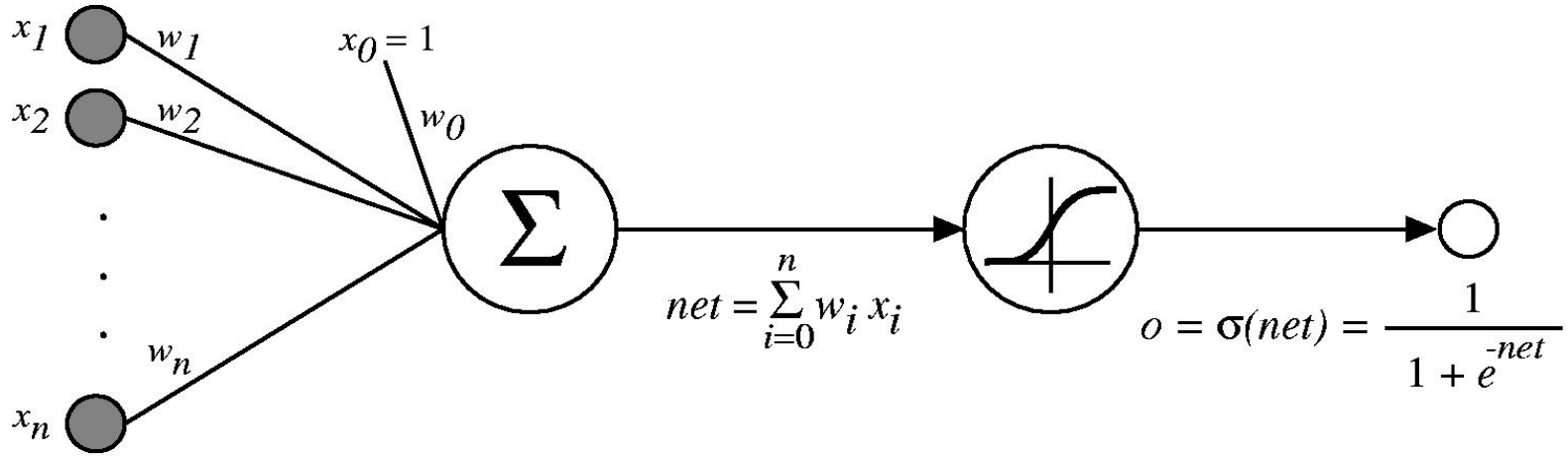
- Perceptron: y is {0,1}, y(x;w) is prediction given w

$$w = w + [y^* - y(x;w)] f(x)$$

Differences?

- Drop the Σ_j over training examples: online vs. batch learning
- Drop the dist'n: probabilistic vs. error driven learning

Sigmoid Unit



$\sigma(x)$ is the sigmoid function

$$\frac{1}{1+e^{-x}}$$

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

Error Gradient for a Sigmoid Unit

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

Error Gradient for a Sigmoid Unit

But we know:

$$\frac{\partial o_d}{\partial net_d} = \frac{\partial \sigma(net_d)}{\partial net_d} = o_d(1 - o_d)$$

$$\frac{\partial net_d}{\partial w_i} = \frac{\partial(\vec{w} \cdot \vec{x}_d)}{\partial w_i} = x_{i,d}$$

So:

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

Let: $\delta_k = -\frac{\partial E}{\partial net_k}$

$$
\begin{aligned}
\frac{\partial E}{\partial net_j} &= \sum_{k \in Outs(j)} \frac{\partial E}{\partial net_k} \frac{\partial net_k}{\partial net_j} \\
&= \sum_{k \in Outs(j)} -\delta_k \frac{\partial net_k}{\partial net_j} \\
&= \sum_{k \in Outs(j)} -\delta_k \frac{\partial net_k}{\partial o_j} \frac{\partial o_j}{\partial net_j} \\
&= \sum_{k \in Outs(j)} -\delta_k \, w_{kj} \frac{\partial o_k}{\partial net_j} \\
&= \sum_{k \in Outs(j)} -\delta_k \, w_{kj} \, o_j(1 - o_j)
\end{aligned}
$$

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

Comparison: Linear vs Sigmoid Unit

- Training rule for linear unit: $o = \sum_{i=0}^{n} w_i x_i$

$$w_i = w_i + \Delta w_i$$

where

$$\Delta w_i = -\eta \frac{\partial E}{\partial w_i} = \eta (t - o) x_i$$

- Training rule for Sigmoid unit $o = Sig(\sum_{i=0}^{n} w_i x_i)$

$$\Delta w_i = -\eta \frac{\partial E}{\partial w_i} = \eta (t - o) o (1 - o) x_i$$

- Training rule for unit of your choice!!! (e.g., *tanh*)
  - Same idea (set up the error function)
  - Use Gradient Descent (compute derivatives)

Example

"win the vote"

"win the election"

"win the game"

w_{SPORTS}

```
BIAS   :
win    :
game   :
vote   :
the    :
...
```

$w_{POLITICS}$

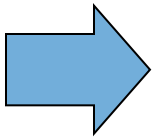
```
BIAS   :
win    :
game   :
vote   :
the    :
...
```

w_{TECH}

```
BIAS   :
win    :
game   :
vote   :
the    :
...
```

Example

"win the vote"



```
BIAS   :  1
win    :  1
game   :  0
vote   :  1
the    :  1
...
```

w_{SPORTS}

```
BIAS   : -2
win    :  4
game   :  4
vote   :  0
the    :  0
...
```

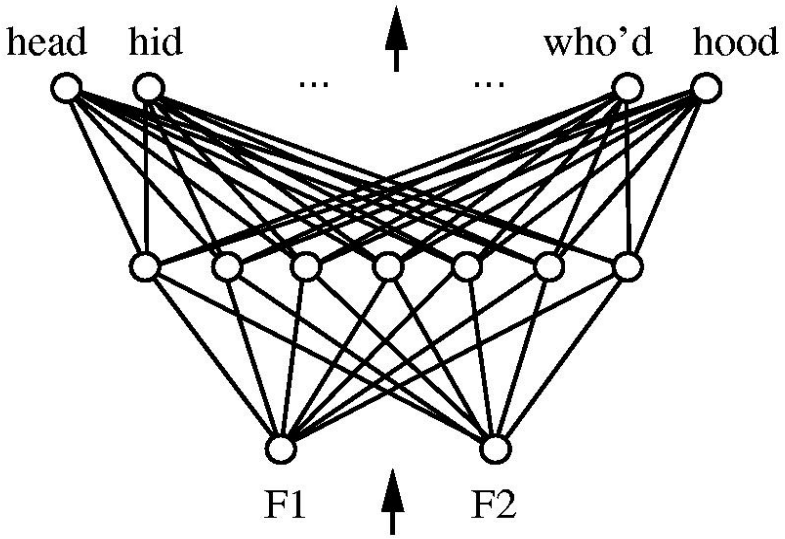
$w_{POLITICS}$

```
BIAS   :  1
win    :  2
game   :  0
vote   :  4
the    :  0
...
```

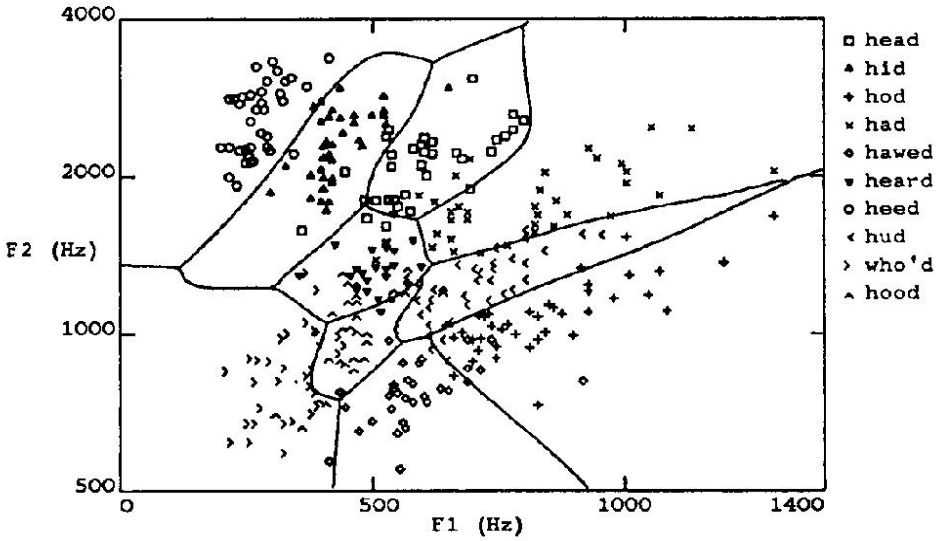
w_{TECH}

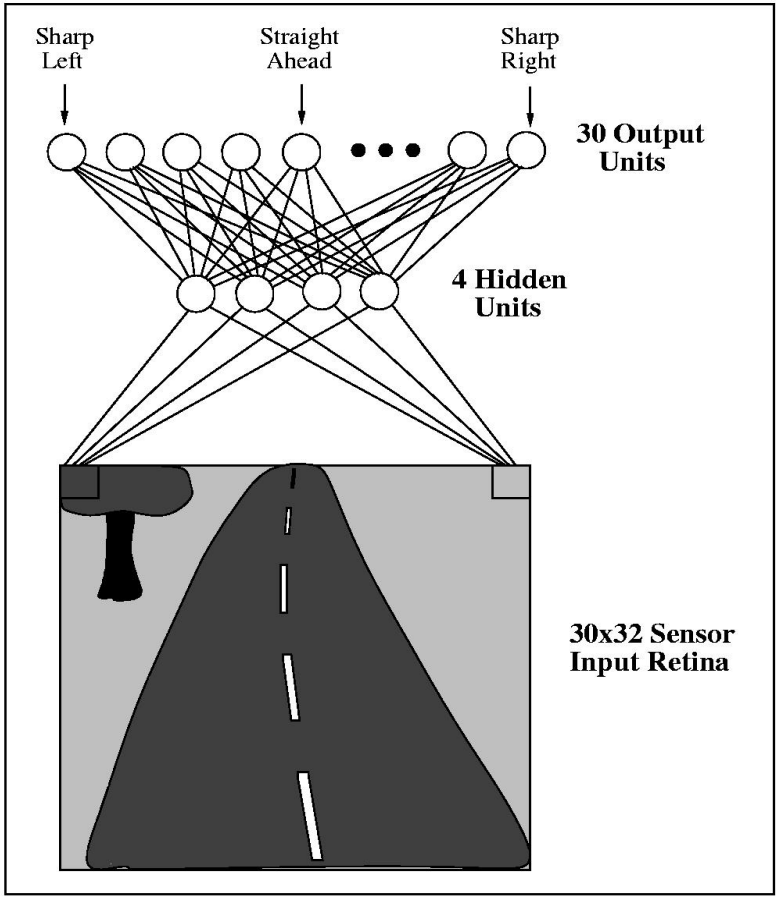
```
BIAS   :  2
win    :  0
game   :  2
vote   :  0
the    :  0
...
```

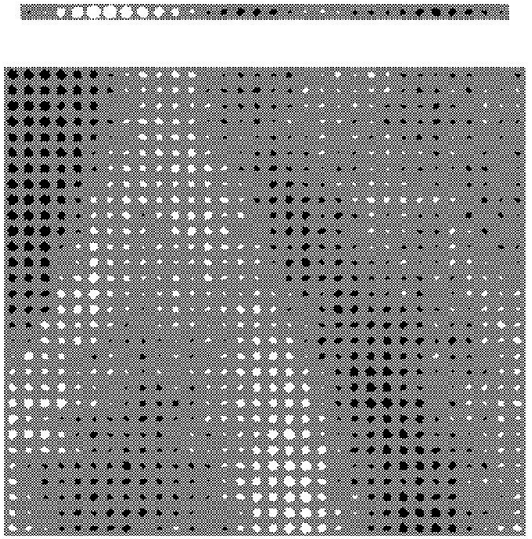
Multilayer Networks of Sigmoid Units



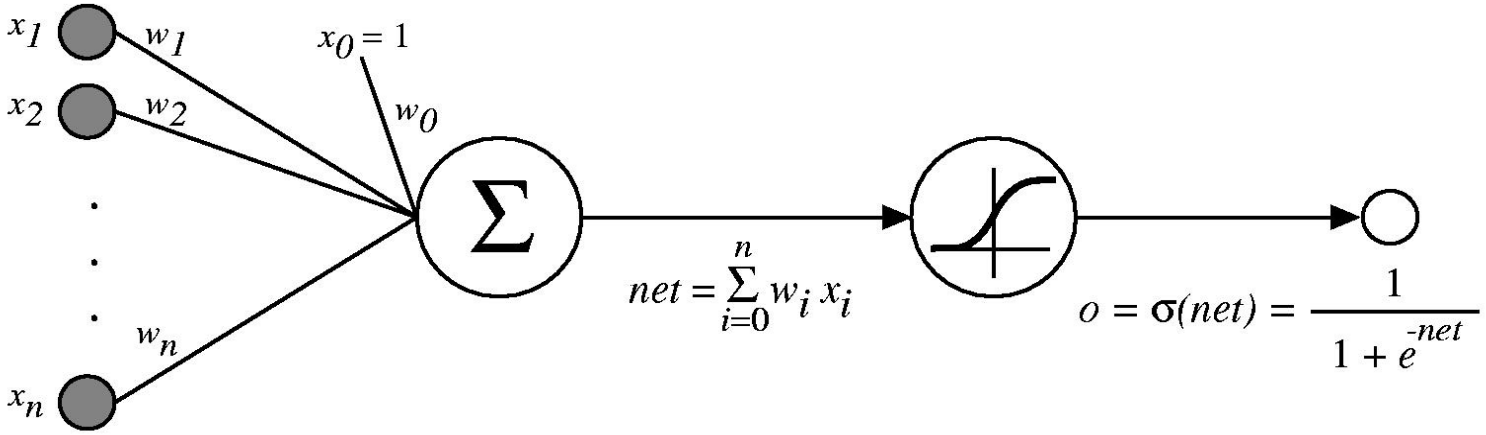
$$out(\mathbf{x}) = g\left(w_0 + \sum_k w_k g(w_0^k + \sum_i w_i^k x_i)\right)$$




Sharp
Left
Straight
Ahead
Sharp
Right
30 Output
Units
4 Hidden
Units
30x32 Sensor
Input Retina



Sigmoid Unit



$\sigma(x)$ is the sigmoid function

$$\frac{1}{1+e^{-x}}$$

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

Sigmoid Unit

We can derive gradient descent rules to train

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

Error Gradient for a Sigmoid Unit

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

Error Gradient for a Sigmoid Unit

But we know:

$$\frac{\partial o_d}{\partial net_d} = \frac{\partial \sigma(net_d)}{\partial net_d} = o_d(1 - o_d)$$

$$\frac{\partial net_d}{\partial w_i} = \frac{\partial(\vec{w} \cdot \vec{x}_d)}{\partial w_i} = x_{i,d}$$

So:

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

Let: $\delta_k = -\frac{\partial E}{\partial net_k}$

Error Gradient for a Sigmoid Unit

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

But we know:

$$\frac{\partial o_d}{\partial net_d} = \frac{\partial \sigma(net_d)}{\partial net_d} = o_d(1 - o_d)$$

$$\frac{\partial net_d}{\partial w_i} = \frac{\partial(\vec{w} \cdot \vec{x}_d)}{\partial w_i} = x_{i,d}$$

So:

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

Let: $\delta_k = -\frac{\partial E}{\partial net_k}$

$$
\begin{aligned}
\frac{\partial E}{\partial net_j} &= \sum_{k \in Outs(j)} \frac{\partial E}{\partial net_k} \frac{\partial net_k}{\partial net_j} \\
&= \sum_{k \in Outs(j)} -\delta_k \frac{\partial net_k}{\partial net_j} \\
&= \sum_{k \in Outs(j)} -\delta_k \frac{\partial net_k}{\partial o_j} \frac{\partial o_j}{\partial net_j} \\
&= \sum_{k \in Outs(j)} -\delta_k \, w_{kj} \frac{\partial o_k}{\partial net_j} \\
&= \sum_{k \in Outs(j)} -\delta_k \, w_{kj} \, o_j(1 - o_j)
\end{aligned}
$$

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

Backpropagation Algorithm

Initialize all weights to small random numbers

Until convergence, Do

For each training example, Do

1. Input it to network and compute 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

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

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

$$w_{i,j} \leftarrow w_{i,j} + \Delta w_{i,j}$$

where $\Delta w_{i,j} = \eta \delta_j x_{i,j}$

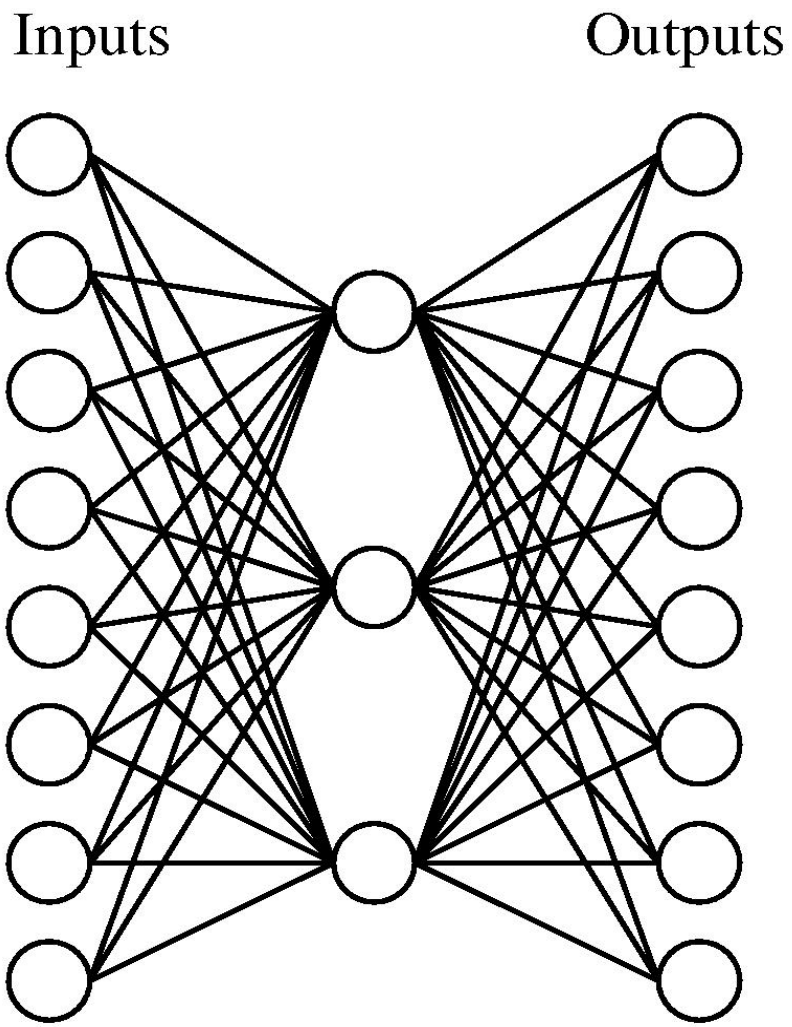
More on Backpropagation

- 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)
- Often include weight *momentum* α

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

- 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



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

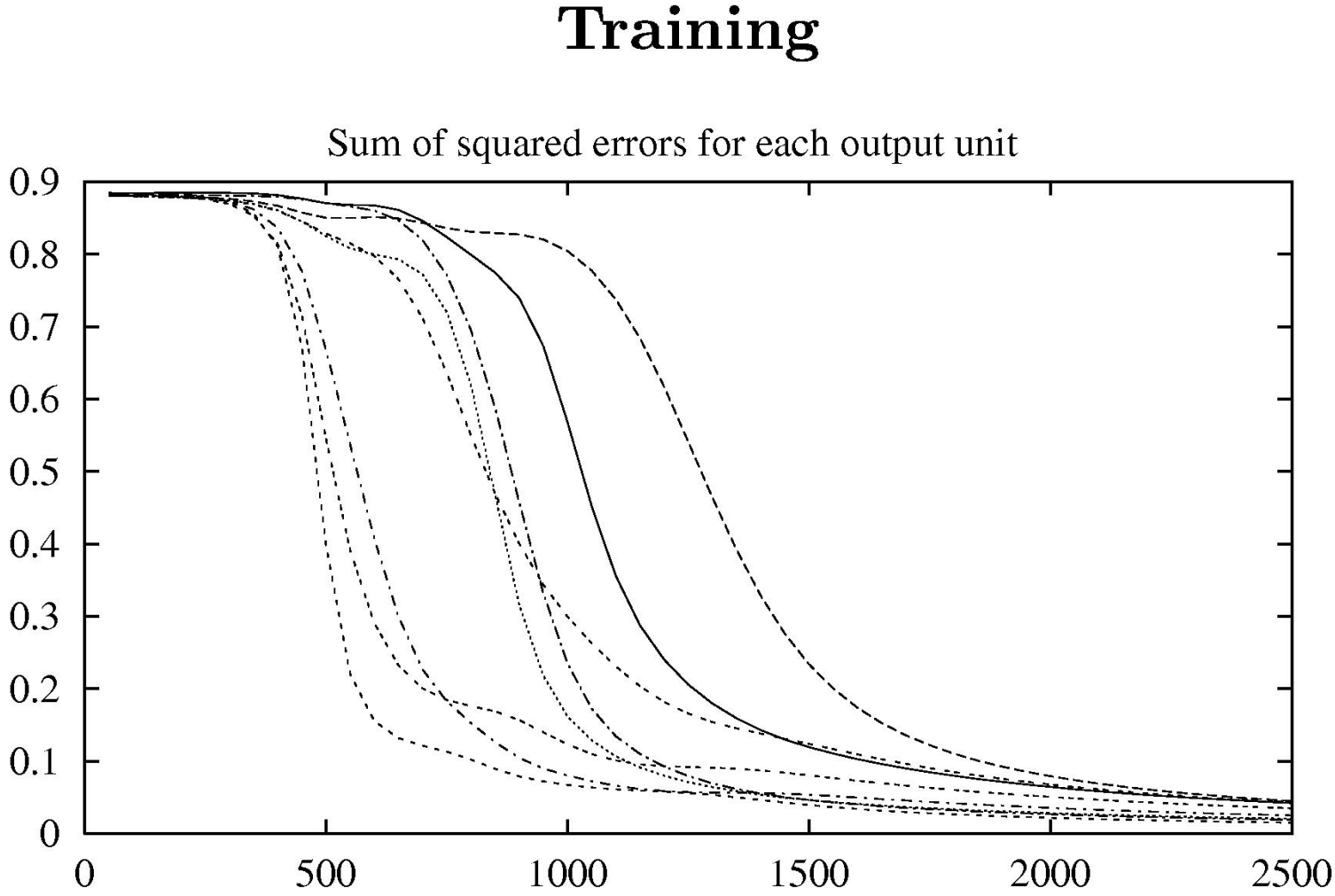
Can this be learned?

Learning Hidden Layer Representations

Learned hidden layer representation:

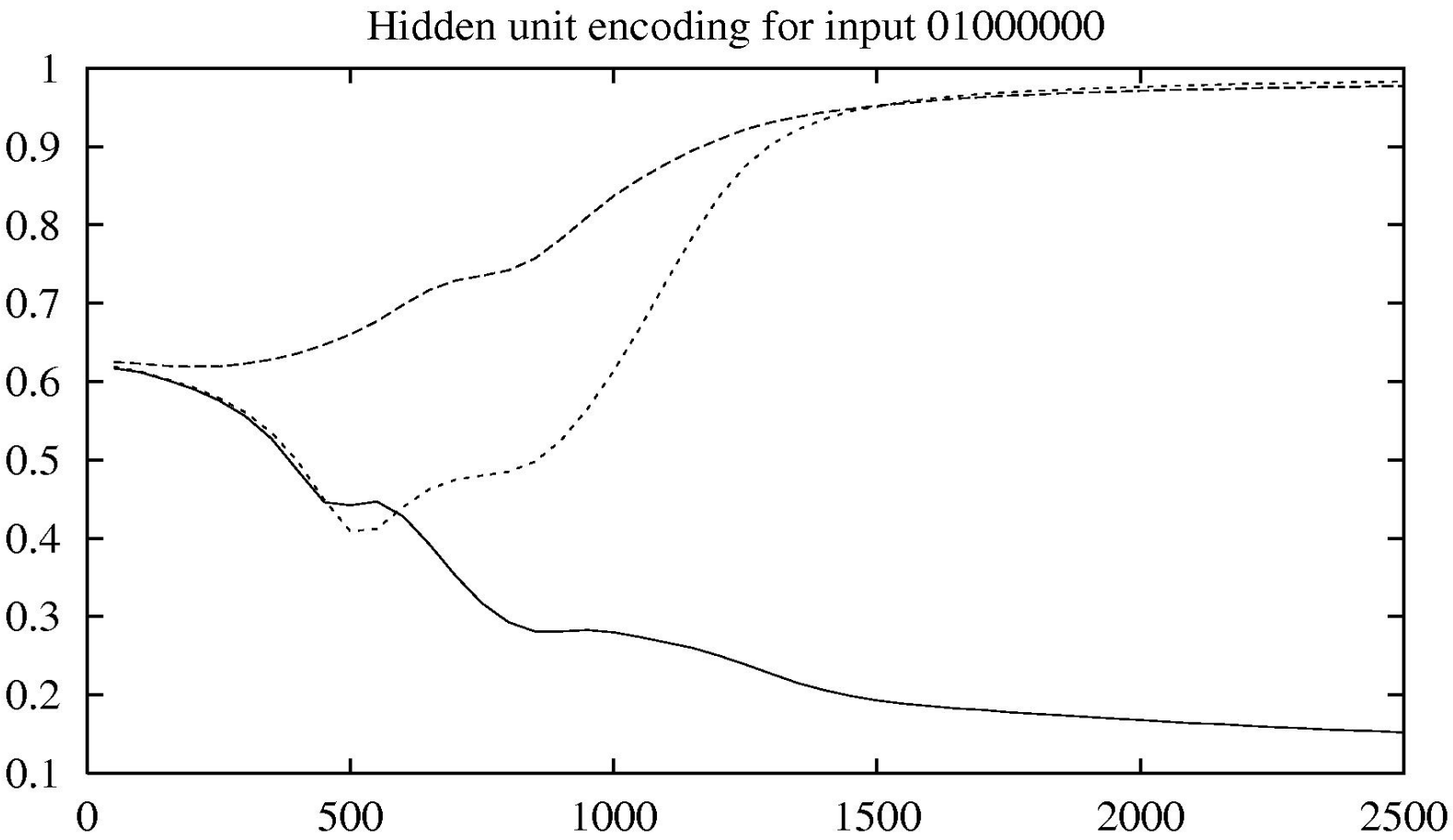
Input		Hidden Values				Output
10000000	→	.89	.04	.08	→	10000000
01000000	→	.01	.11	.88	→	01000000
00100000	→	.01	.97	.27	→	00100000
00010000	→	.99	.97	.71	→	00010000
00001000	→	.03	.05	.02	→	00001000
00000100	→	.22	.99	.99	→	00000100
00000010	→	.80	.01	.98	→	00000010
00000001	→	.60	.94	.01	→	00000001

Learning Hidden Layer Representations



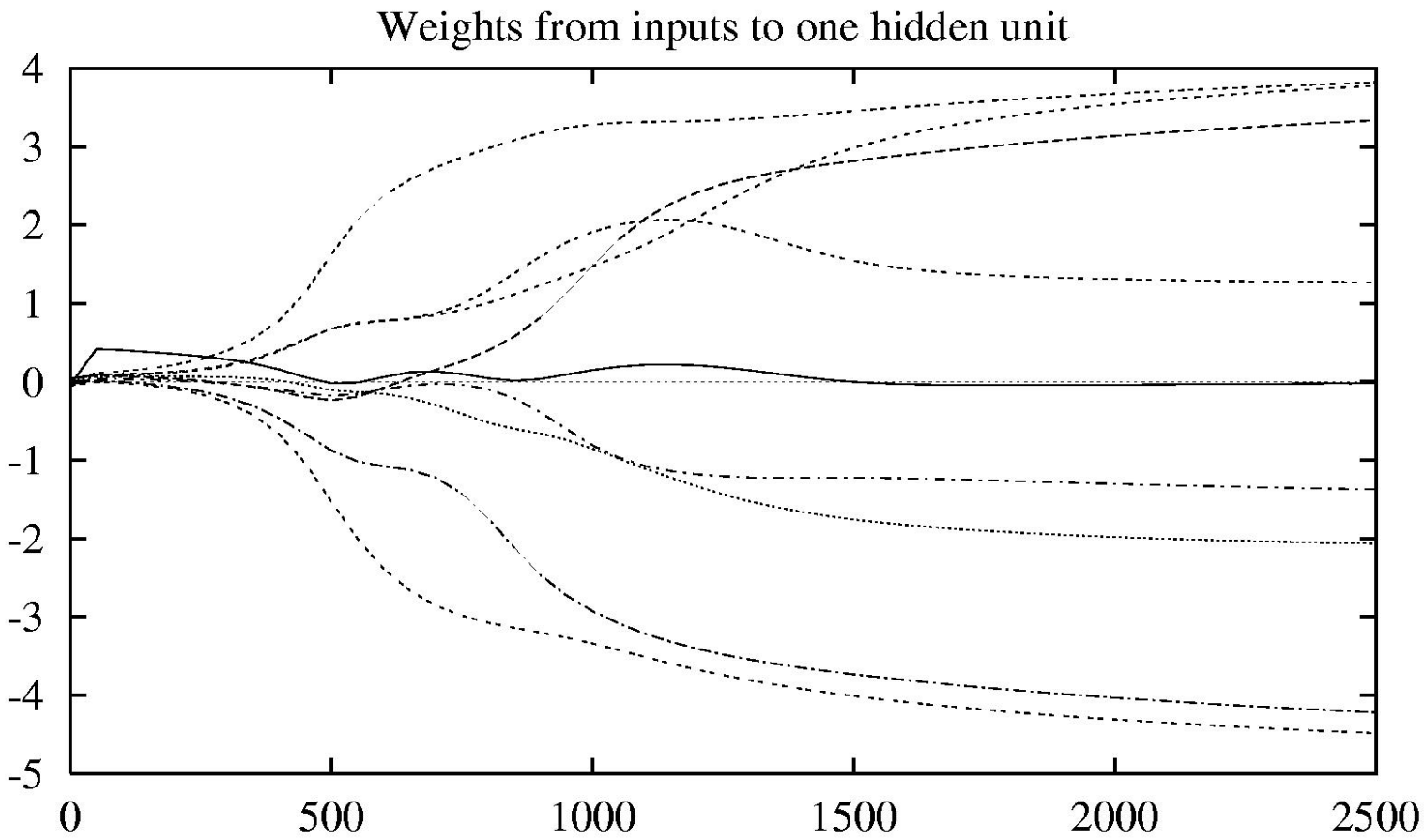
Learning Hidden Layer Representations

Training



Learning Hidden Layer Representations

Training



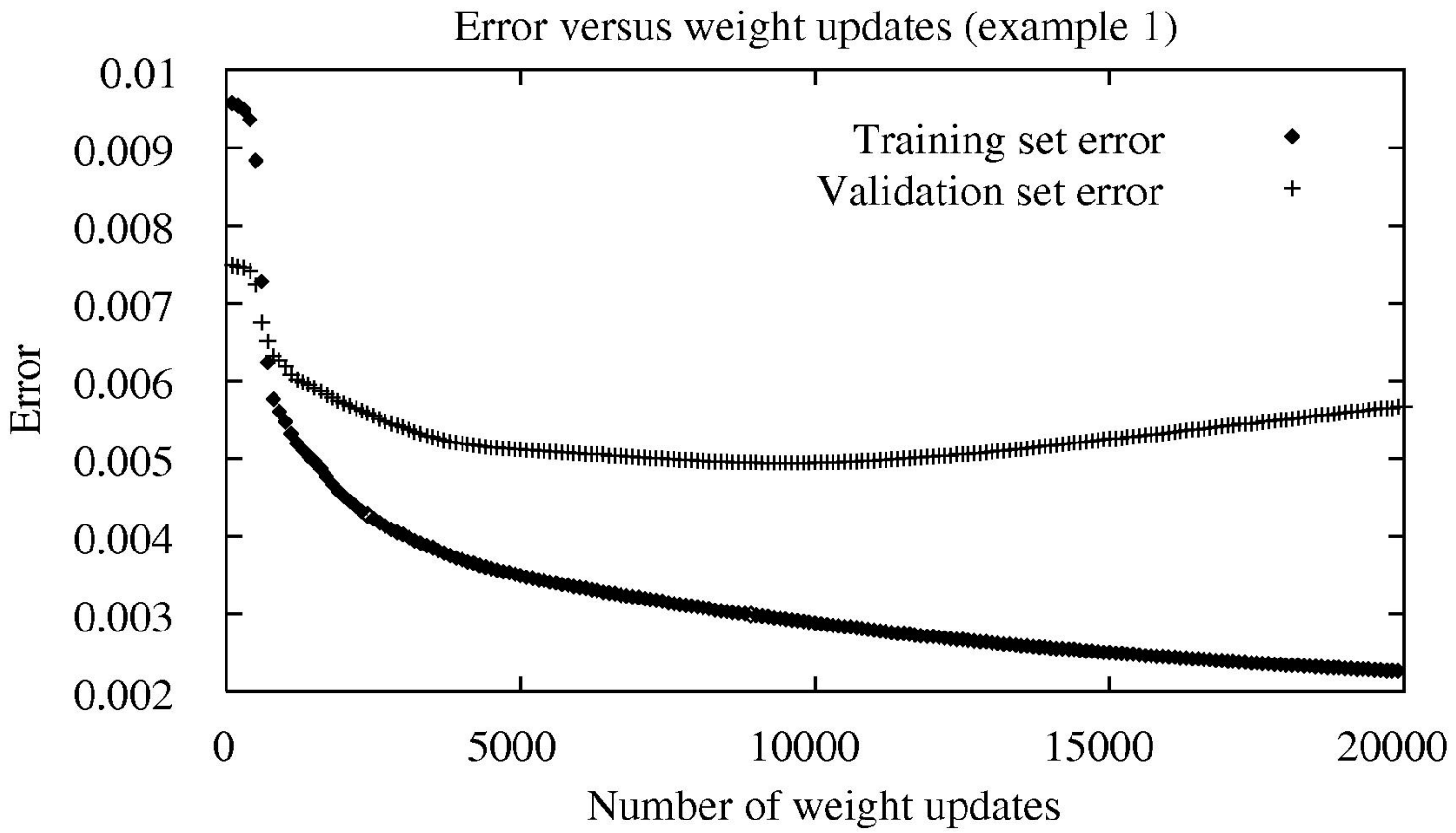
Convergence of Backpropagation

- Gradient descent to some local minimum
  - Perhaps not global minimum
  - Add momentum
  - Stochastic gradient descent
  - Train multiple nets when different initial weights
- Nature of convergence
  - Initialize weights near zero
  - Therefore, initial networks near-linear
  - Increasingly non-linear functions possible as training progresses

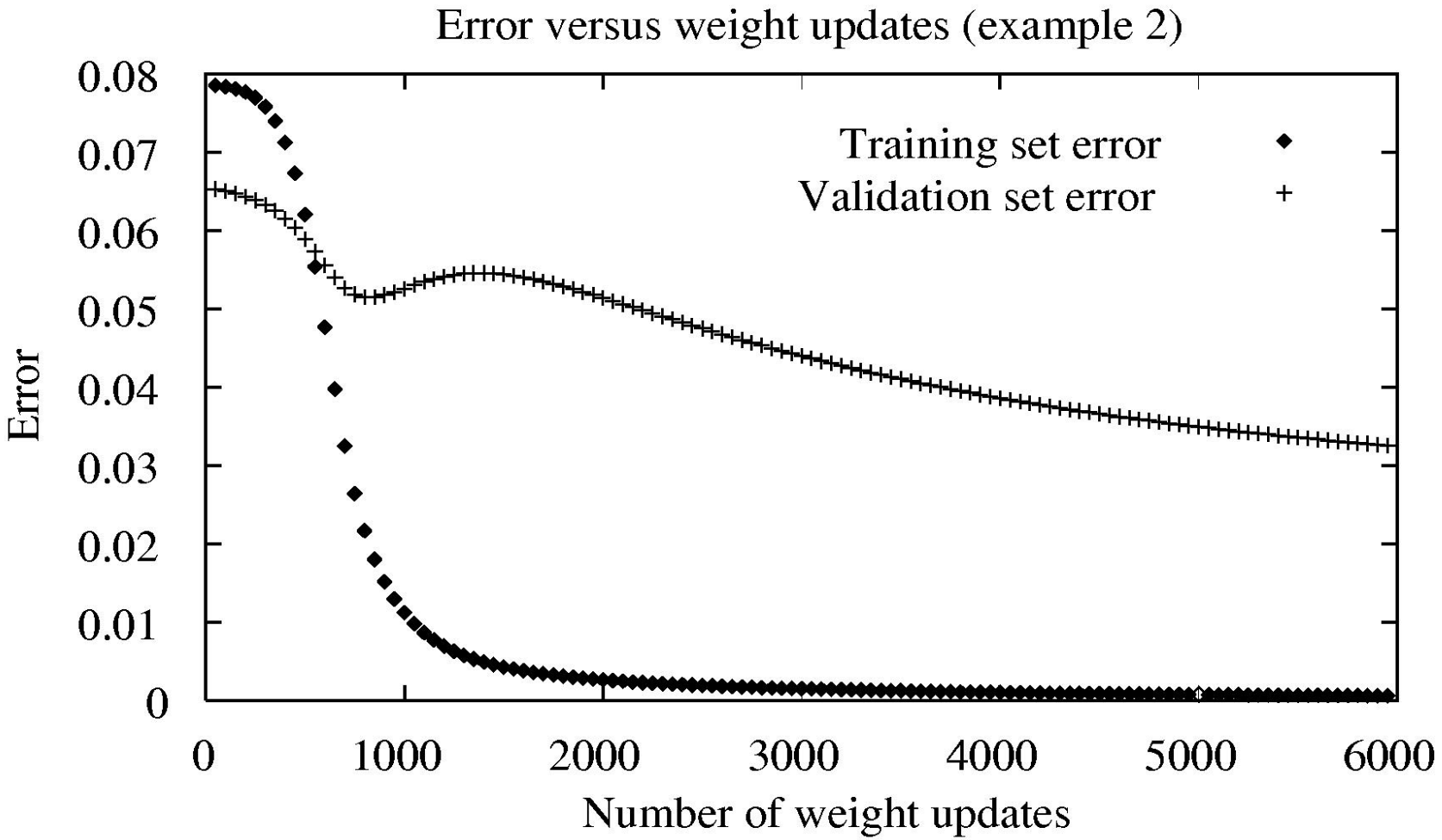
Expressiveness of Neural Nets

- 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 accurac by a network by a network with two hidden layers

Overfitting in Neural Networks



Overfitting in Neural Networks



Overfitting Avoidance

Penalize large weights:

$$E(\vec{w}) \equiv \frac{1}{2} \sum_{d \in D} \sum_{k \in outputs} (t_{kd} - o_{kd})^2 + \gamma \sum_{i,j} w_{ji}^2$$

Train on target slopes as well as values:

$$E(\vec{w}) \equiv \frac{1}{2} \sum_{d \in D} \sum_{k \in outputs} \left[(t_{kd} - o_{kd})^2 + \mu \sum_{j \in inputs} \left(\frac{\partial t_{kd}}{\partial x_d^j} - \frac{\partial o_{kd}}{\partial x_d^j} \right)^2 \right]$$

Weight sharing

Early stopping

Generalization and Overfitting

- Stopping conditions
  - Continue till error E reaches a threshold
    - Can lead to overfitting and poor generalization on test set
- Overfitting
  - Weight decay: decrease weight by a small factor each iteration
  - Use Validation Set to measure generalization accuracy
    - Generalization accuracy first decreases, then increases
  - For small datasets, k-fold cross-validation.
    - Repeat training and validation k times and average the results

Neural Nets Application Scenarios

- Many attribute-value pairs
  - Attributes may be either correlated or independent
- Target function is discrete/real-valued
- Training examples may contain errors
- Long training times are acceptable
- Fast evaluation of learned target function
- Interpretability is not a concern