Artificial Neural Networks

Based on “Machine Learning”, T. Mitchell, McGRAW Hill, 1997, ch. 4

Acknowledgement:

The present slides are an adaptation of slides drawn by T. Mitchell

PLAN

1. Introduction
   Connectionist models
   2 NN examples: ALVINN driving system, face recognition
2. The perceptron; the linear unit; the sigmoid unit
   The gradient descent learning rule
3. Multilayer networks of sigmoid units
   The Backpropagation algorithm
   Hidden layer representations
   Overfitting in NNs
4. Advanced topics
   Alternative error functions
   Predicting a probability function
   [Recurrent networks]
   [Dynamically modifying the network structure]
   [Alternative error minimization procedures]
5. Expressive capabilities of NNs

Connectionist Models

Consider humans:

- Neuron switching time: .001 sec.
- Number of neurons: 10^{10}
- Connections per neuron: 10^{4-5}
- Scene recognition time: 0.1 sec.
- 100 inference steps doesn't seem like enough

→ much parallel computation

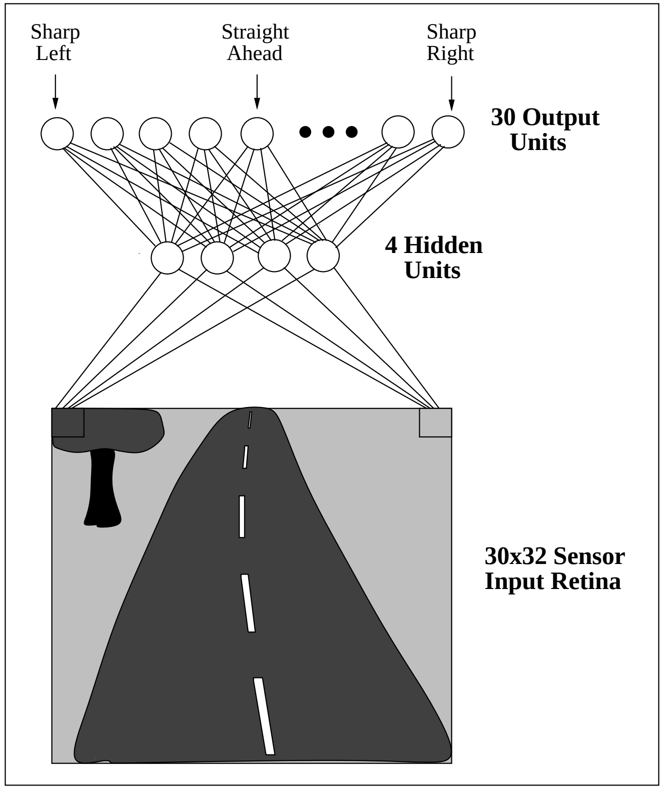
Properties of artificial neural nets

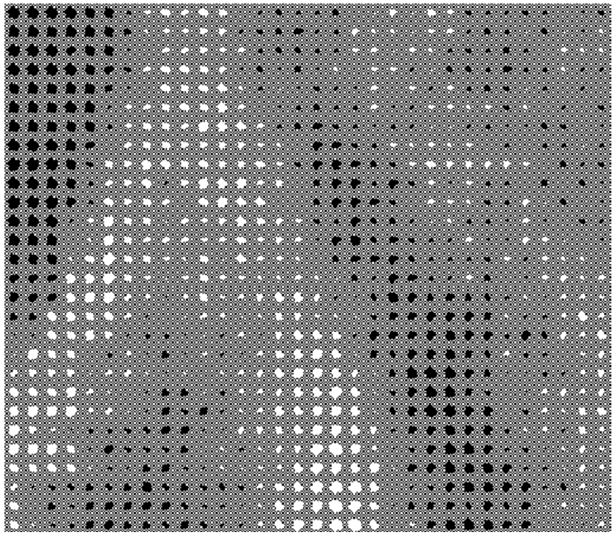
- Many neuron-like threshold switching units
- Many weighted interconnections among units
- Highly parallel, distributed process
- Emphasis on tuning weights automatically

A First NN Example:

ALVINN drives at 70 mph on highways

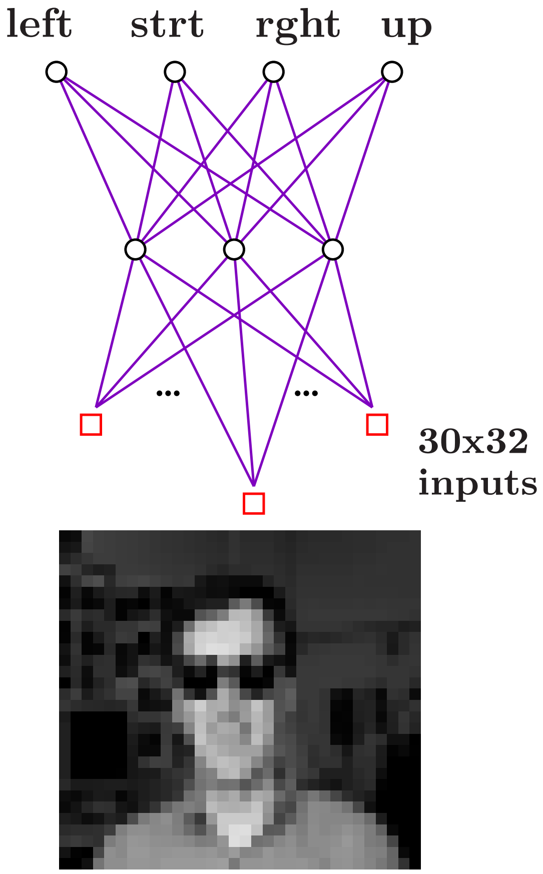
[Pomerleau, 1993]





A Second NN Example

Neural Nets for Face Recognition

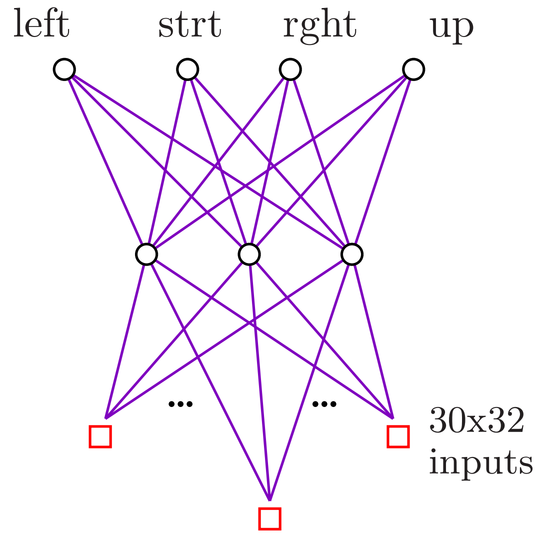


Typical input images:

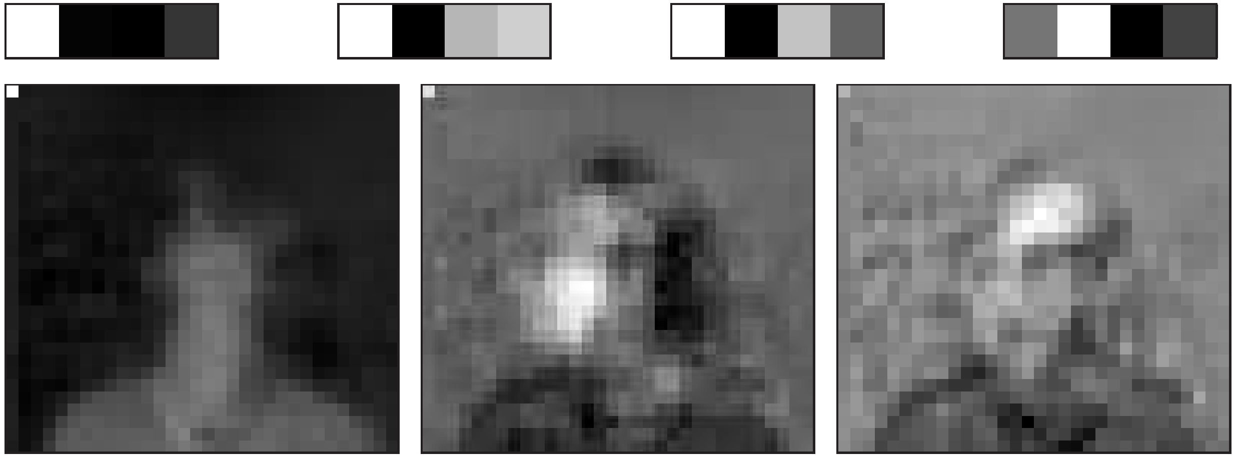
http://www.cs.cmu.edu/~tom/faces.html

Results:
90% accurate learning head pose,
and recognizing 1-of-20 faces

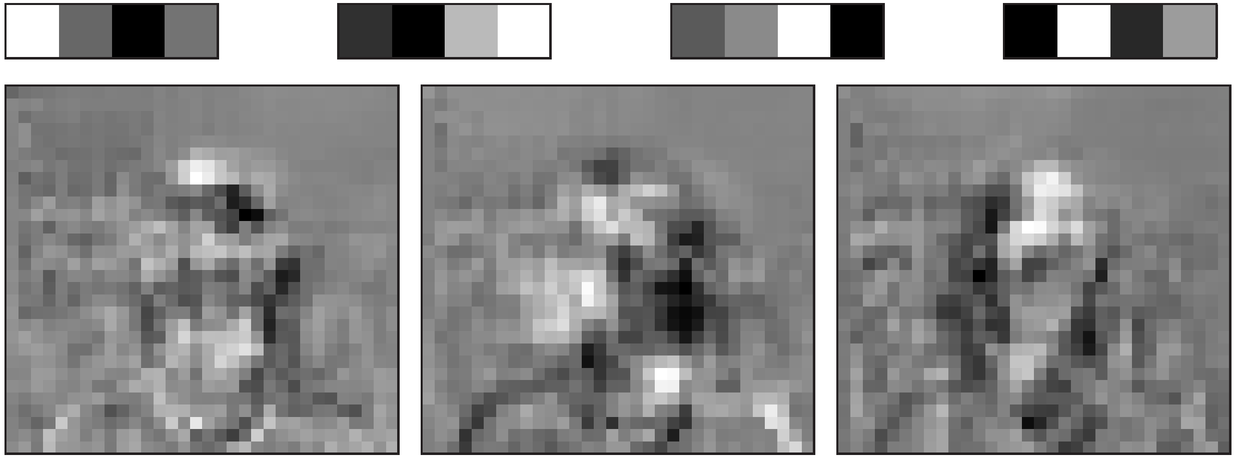
Learned Weights



after 1 epoch:



after 100 epochs:



Design Issues for these two NN Examples

See Tom Mitchell's "Machine Learning" book,
pag. 82-83, and 114 for ALVINN, and
pag. 112-177 for face recognition:

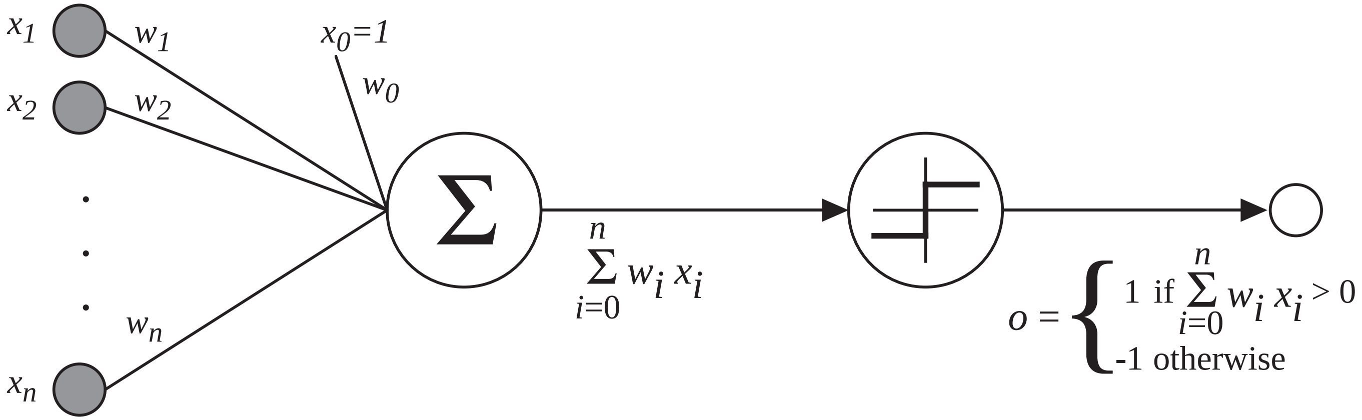
- input encoding
- output encoding
- network graph structure
- learning parameters:
  η (learning rate), α (momentum), number of iterations

When to Consider Neural Networks

- 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

2. The Perceptron

[Rosenblat, 1962]

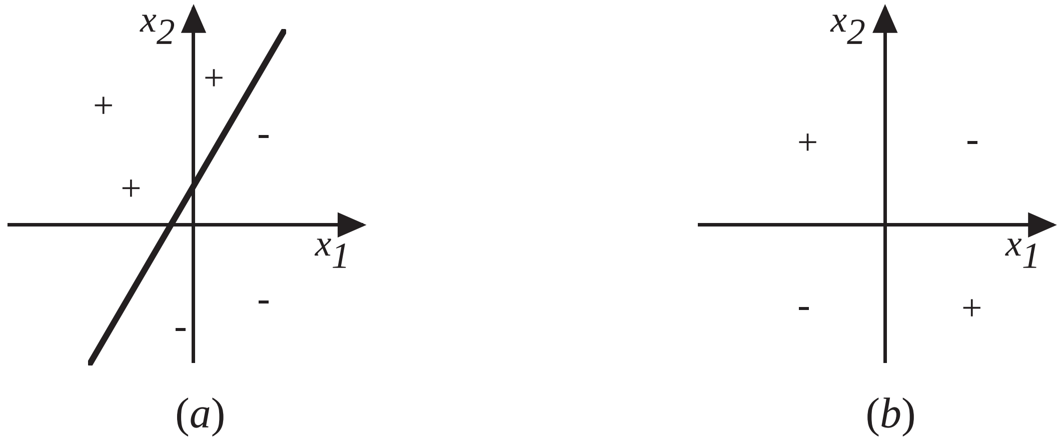

x_1
w_1
x_2
w_2
$x_0=1$
w_0
w_n
x_n
Σ
$\sum_{i=0}^{n} w_i x_i$
$o = \begin{cases} 1 & \text{if } \sum_{i=0}^{n} w_i x_i > 0 \\ -1 & \text{otherwise} \end{cases}$

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

Sometimes we'll use simpler vector notation:

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

Decision Surface of a Perceptron



Represents some useful functions

- What weights represent $g(x_1, x_2) = AND(x_1, x_2)$?

But certain examples may not be linearly separable

- Therefore, we'll want networks of these...

The Perceptron Training Rule

$$w_i \leftarrow w_i + \Delta w_i \textbf{ with } \Delta w_i = \eta(t - o)x_i$$

or, in vectorial notation:

$$\vec{w} \leftarrow \vec{w} + \Delta\vec{w} \textbf{ with } \Delta\vec{w} = \eta(t - o)\vec{x}$$

where:

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

It will converge (proven by [Minsky & Papert, 1969])

- **if the training data is linearly separable**
- **and** η **is sufficiently small.**

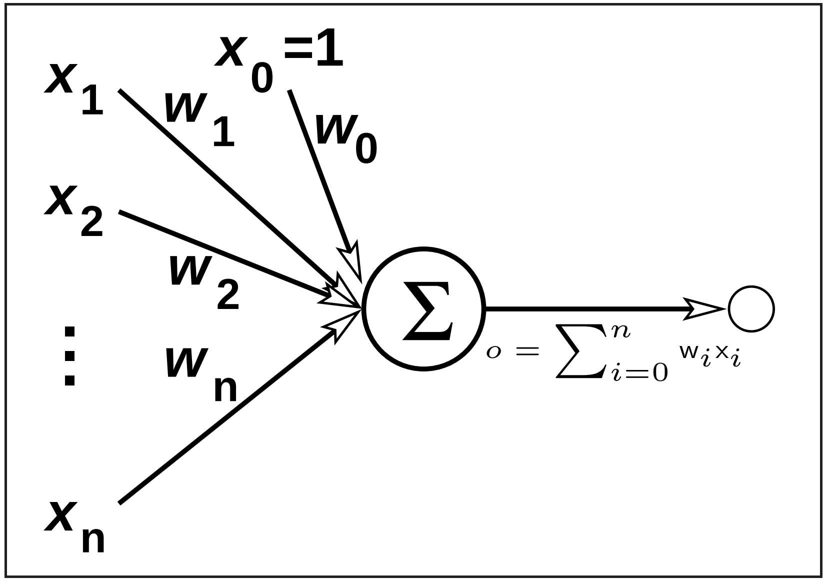
$2'$. The Linear Unit

To understand the perceptron's training rule, consider a (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***

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



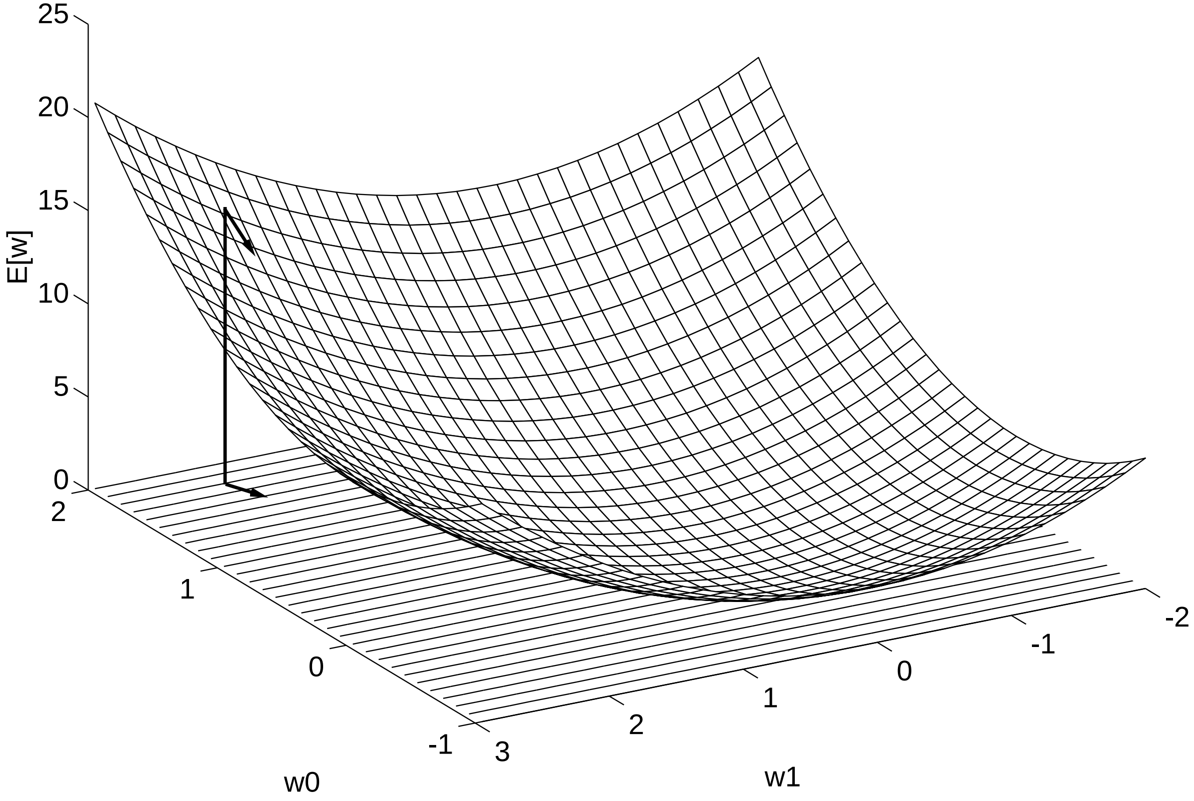
where D is set of training examples.

The linear unit uses the descent gradient training rule, presented on the next slides.

Remark:

Ch. 6 (Bayesian Learning) shows that the hypothesis $h = (w_0, w_1, \ldots, w_n)$ that minimises E is the most probable hypothesis given the training data.

The Gradient Descent Rule



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:

$$\vec{w} = \vec{w} + \Delta \vec{w},$$

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

Therefore,

$$w_i = w_i + \Delta w_i,$$

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

The Gradient Descent Rule for the Linear Unit

Computation

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

Therefore

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

The Gradient Descent Algorithm for the Linear Unit

```
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
            $t$ is the target output value.
        $\eta$ is the learning rate (e.g., .05).
    • Initialize each $w_i$ to some small random value
    • Until the termination condition is met
        – Initialize each $\Delta w_i$ to zero.
        – For each $\langle \vec{x}, t \rangle$ in $training\_examples$
            ∗ Input the instance $\vec{x}$ to the unit and compute the output $o$
            ∗ For each linear unit weight $w_i$
                  $\Delta w_i \leftarrow \Delta w_i + \eta(t - o)x_i$
        – For each linear unit weight $w_i$
                  $w_i \leftarrow w_i + \Delta w_i$
```

Convergence

[Hertz et al., 1991]

The gradient descent training rule used by the linear unit is guaranteed to converge to a hypothesis with minimum squared error

- given a sufficiently small learning rate η
- even when the training data contains noise
- even when the training data is not separable by H

Note: If η is too large, the gradient descent search runs the risk of overstepping the minimum in the error surface rather than settling into it. For this reason, one common modification of the algorithm is to gradually reduce the value of η as the number of gradient descent steps grows.

Remark

Gradient descent (and similary, gradient ascent: $\vec{w} \leftarrow \vec{w} + \eta \nabla E$) is an important general paradigm for learning. It is a strategy for searching through a large or infinite hypothesis space that can be applied whenever

- the hypothesis space contains continuously parametrized hypotheses
- the error can be differentiated w.r.t. these hypothesis parameters.

Practical difficulties in applying the gradient method:

- if there are multiple local optima in the error surface, then there is no guarantee that the procedure will find the global optimum.
- converging to a local optimum can sometimes be quite slow.

To alleviate these difficulties, a variation called incremental (or: stochastic) gradient method was proposed.

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

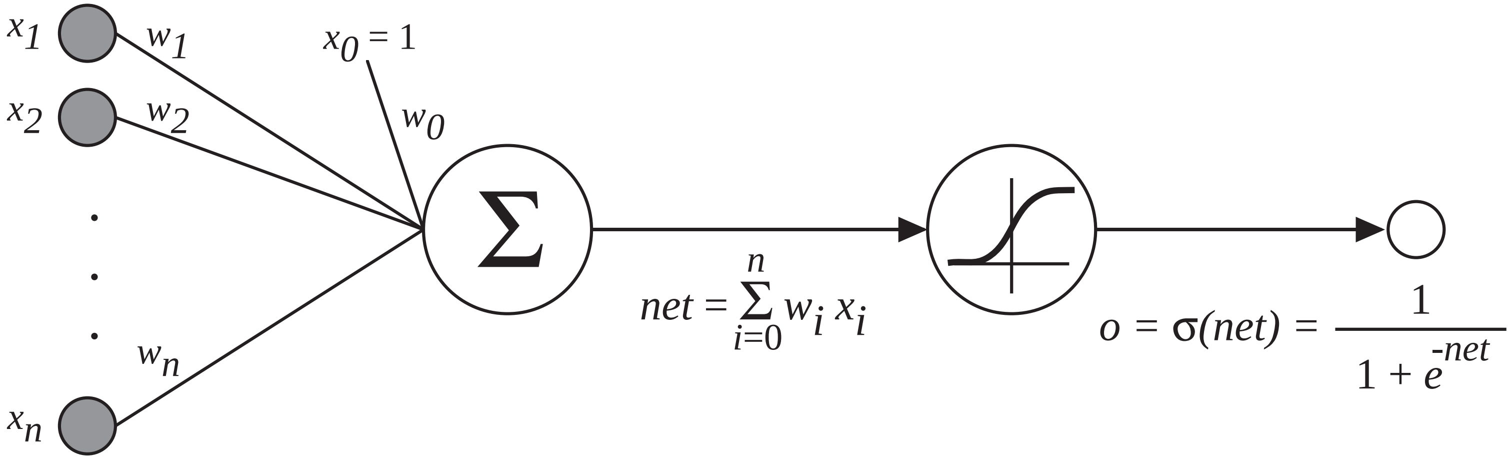
$$E_D[\vec{w}] \equiv \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2$$

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

Covergence:

The *Incremental Gradient Descent* can approximate the *Batch Gradient Descent* arbitrarily closely if η is made small enough.

2″. The Sigmoid Unit


x_1
x_2
x_n
w_1
w_2
w_n
$x_0 = 1$
w_0
Σ
$net = \sum_{i=0}^{n} w_i x_i$
$o = \sigma(net) = \frac{1}{1 + e^{-net}}$

$\sigma(x)$ **is the sigmoid function** $\frac{1}{1+e^{-x}}$

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

We can derive gradient descent rules to train

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

Error Gradient for the 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 $net_d = \sum_{i=0}^{n} w_i x_{i,d}$

But

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

So:

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

Remark

Instead of gradient descent, one could use linear programming to find hypothesis consistent with separable data.

[Duda & Hart, 1973] have shown that linear programming can be extended to the non-linear separable case.

However, linear programming does not scale to multilayer networks, as gradient descent does (see next section).

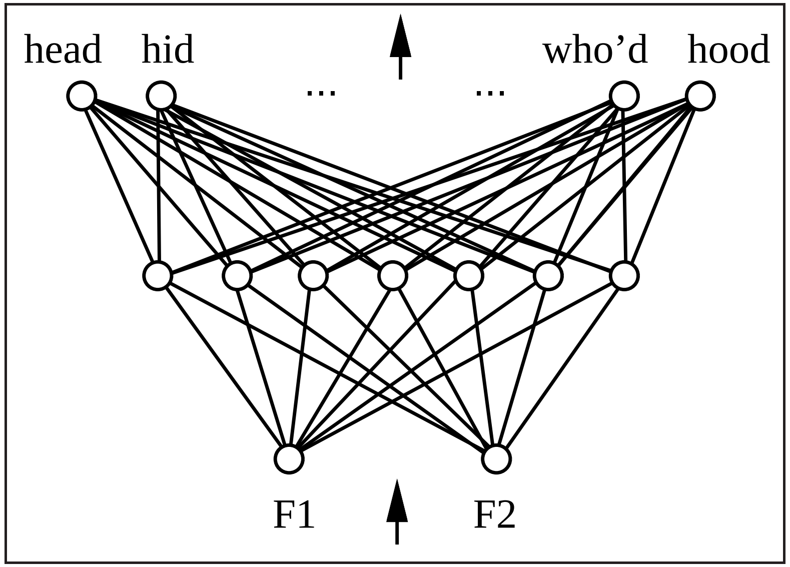
3. Multilayer Networks of Sigmoid Units

An example

This network was trained to recognize 1 of 10 vowel sounds occurring in the context "h_d" (e.g. "head", "hid").

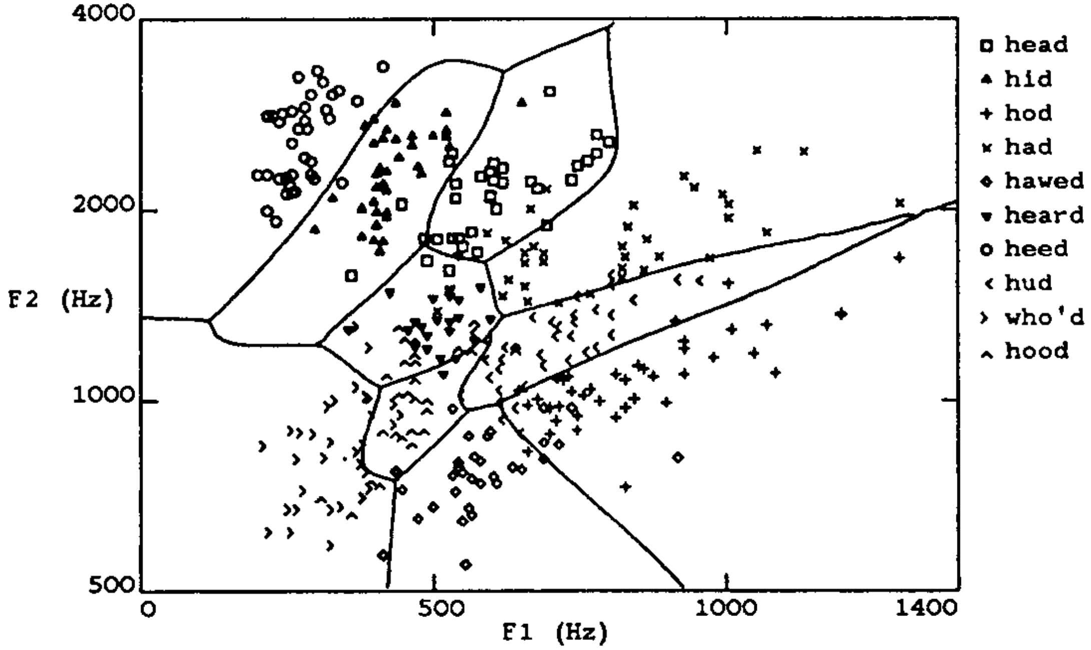
The *inputs* have been obtained from a spectral analysis of sound.

The 10 network *outputs* correspond to the 10 possible vowel sounds. The network prediction is the output whose value is the highest.



This plot illustrates the highly non-linear decision surface represented by the learned network.

Points shown on the plot are test examples distinct from the examples used to train the network.



from [Haug & Lippmann, 1988]

3.1 The Backpropagation Algorithm (Rumelhart et al., 1986)

Formulation for a feed-forward 2-layer network of sigmoid units, the stochastic version

Idea: Gradient descent over the entire vector of network weights.

Initialize all weights to small random numbers.

Until satisfied, // *stopping criterion* to be (later) defined

for each training example,

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:**
   $\delta_h \leftarrow o_h(1 - o_h) \sum_{k \in outputs} w_{kh}\delta_k$
4. **update each network weight w_{ji}:**
   $w_{ji} \leftarrow w_{ji} + \Delta w_{ji}$ **where** $\Delta w_{ji} = \eta \delta_j x_{ji}$,
   and x_{ji} is the ith input to unit j.

Derivation of the Backpropagation rule,

(following [Tom Mitchell, 1997], pag. 101–103)

Notations:

x_{ji}: the ith input to unit j;
(${j}$ could be either hidden or output unit)

w_{ji}: the weight associated with the ith input to unit j

$\boldsymbol{net}_j = \sum_i w_{ji} x_{ji}$

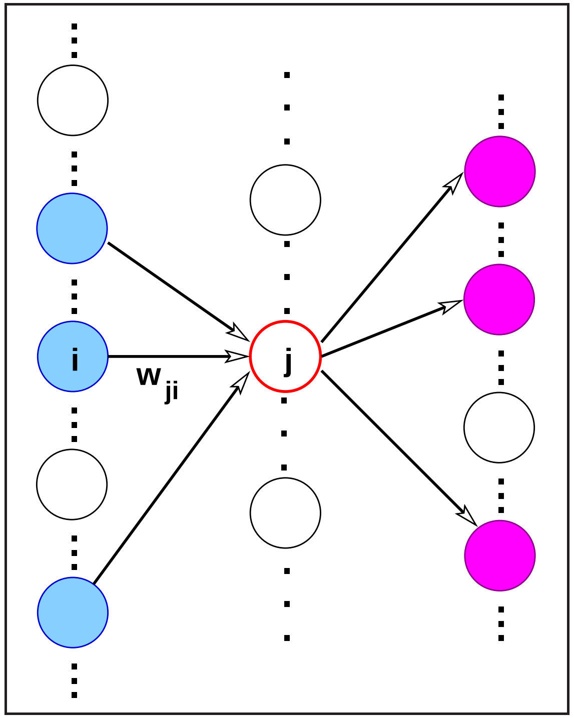
σ: the sigmoid function

o_j: the output computed by unit j; ($o_j = \sigma(\boldsymbol{net}_j)$)

outputs: the set of units in the final layer of the network

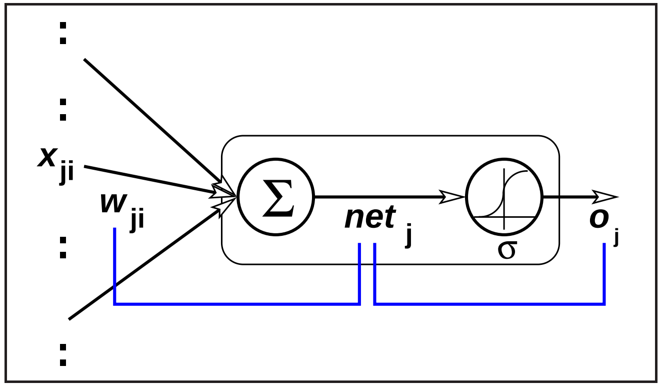
Downstream(j): the set of units whose immediate inputs include the output of unit j

E_d: the training error on the example d (summing over all of the network output units)



Legend: in magenta color, units belonging to *Downstream(j)*

Preliminaries



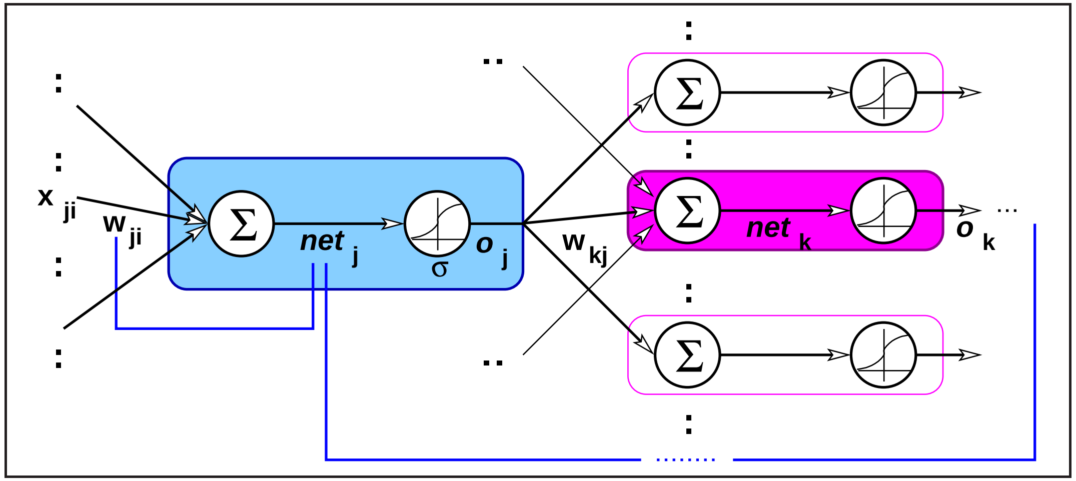
$$E_d(\vec{w}) = \frac{1}{2}\sum_{k\in outputs}(t_k - o_k)^2 = \frac{1}{2}\sum_{k\in outputs}(t_k - \sigma(\boldsymbol{net}_k))^2$$

Common staff for both hidden and output units:

$$\boldsymbol{net}_j = \sum_i w_{ji}x_{ji}$$

$$\Rightarrow \frac{\partial E_d}{\partial w_{ji}} = \frac{\partial E_d}{\partial \boldsymbol{net}_j}\frac{\partial \boldsymbol{net}_j}{\partial w_{ji}} = \frac{\partial E_d}{\partial \boldsymbol{net}_j}x_{ji}$$

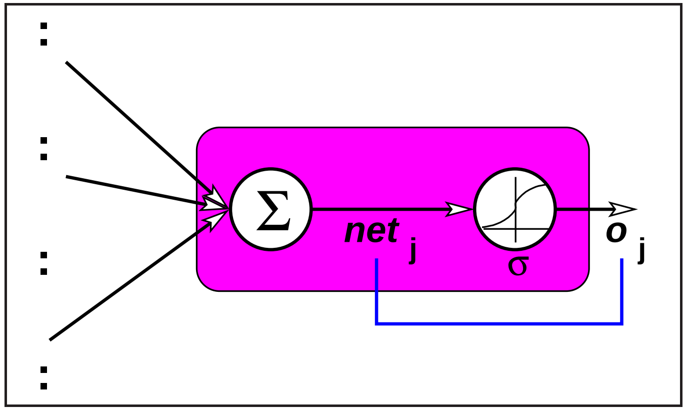
$$\Rightarrow \Delta w_{ji} \overset{def}{=} -\eta\frac{\partial E_d}{\partial w_{ji}} = -\eta\frac{\partial E_d}{\partial \boldsymbol{net}_j}x_{ji}$$



Note: **In the sequel we will use the notation:** $\delta_j = -\frac{\partial E_d}{\partial \boldsymbol{net}_j} \Rightarrow \Delta w_{ji} = \eta\delta_j x_{ji}$

Stage/Case 1: Computing the increments (Δ) for output unit weights

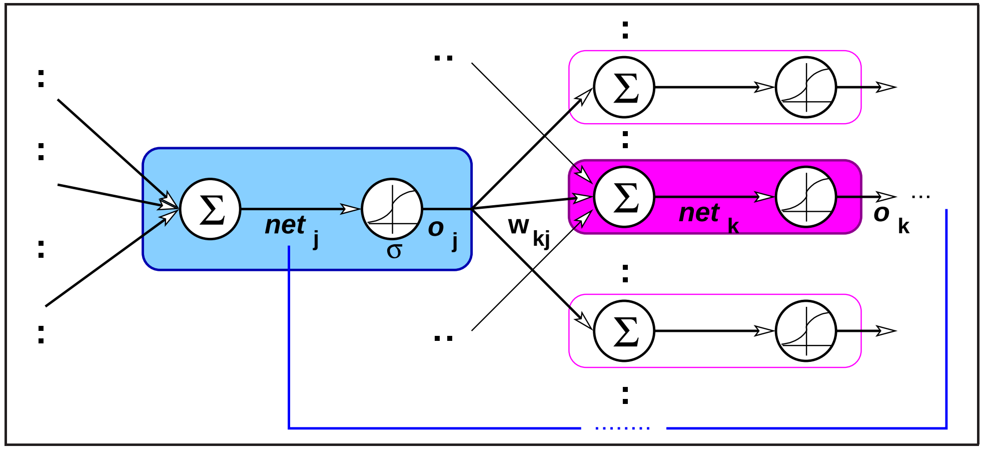
$$
\begin{aligned}
\frac{\partial E_d}{\partial net_j} &= \frac{\partial E_d}{\partial o_j}\frac{\partial o_j}{\partial net_j} \\
\frac{\partial o_j}{\partial net_j} &= \frac{\partial \sigma(net_j)}{\partial net_j} = o_j(1-o_j) \\
\frac{\partial E_d}{\partial o_j} &= \frac{\partial}{\partial o_j}\frac{1}{2}\sum_{k\in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



$$
\begin{aligned}
\Rightarrow \frac{\partial E_d}{\partial net_j} &= -(t_j-o_j)o_j(1-o_j) = -o_j(1-o_j)(t_j-o_j) \\
\Rightarrow \delta_j &\overset{not.}{=} -\frac{\partial E_d}{\partial net_j} = o_j(1-o_j)(t_j-o_j) \\
\Rightarrow \Delta w_{ji} &= \eta\delta_j x_{ji} = \eta o_j(1-o_j)(t_j-o_j)x_{ji}
\end{aligned}
$$

Stage/Case 2: Computing the increments (Δ) for hidden unit weights

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



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

Therefore:

$$\delta_j \overset{not}{=} -\frac{\partial E_d}{\partial net_j} = o_j (1 - o_j) \sum_{k \in Downstream(j)} \delta_k w_{kj}$$

$$\Delta w_{ji} \overset{def}{=} -\eta \frac{\partial E_d}{\partial w_{ji}} = -\eta \frac{\partial E_d}{\partial net_j} \frac{\partial net_j}{\partial w_{ji}} = -\eta \frac{\partial E_d}{\partial net_j} x_{ji} = \eta \delta_j x_{ji} = \eta \; [\; o_j (1 - o_j) \sum_{k \in Downstream(j)} \delta_k w_{kj} \;] \; x_{ji}$$

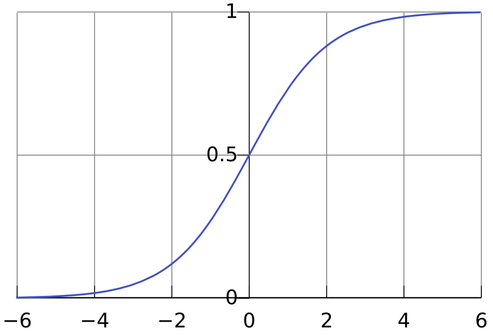
Convergence of Backpropagation
for NNs of Sigmoid units

Nature of convergence

- The weights are initialized near zero; therefore, initial decision surfaces are near-linear.

  Explanation: o_j **is of the form** $\sigma(\vec{w} \cdot \vec{x})$**, therefore** $w_{ji} \approx 0$ **for all** i, j**;**

  note that the graph of σ **is approximately liniar in the vecinity of** 0**.**



- Increasingly non-linear functions are possible as training progresses
- Will find a local, not necessarily global error minimum. In practice, often works well (can run multiple times).

More on Backpropagation

- Easily generalized to arbitrary directed graphs
- Training can take thousands of iterations → slow!
- Often include weight momentum α

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

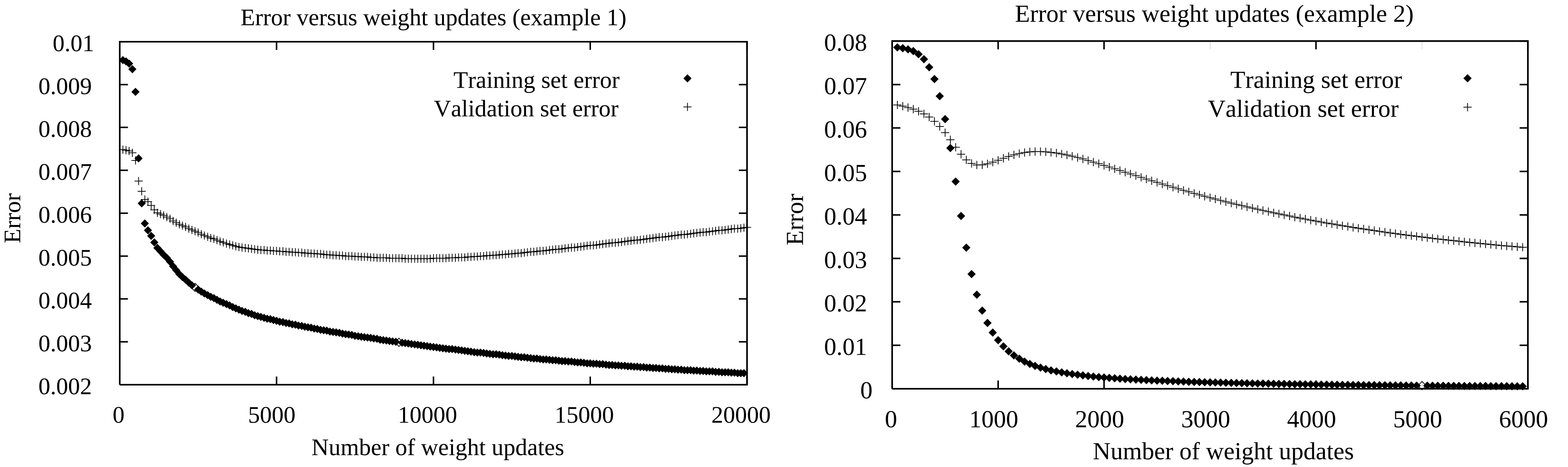
  Effect:

  - speed up convergence (increase the step size in regions where the gradient is unchanging);
  - “keep the ball rolling” through local minima (or along flat regions) in the error surface

- Using network after training is very fast
- Minimizes error over *training* examples;
  Will it generalize well to subsequent examples?

3.2 Stopping Criteria when Training ANNs and Overfitting

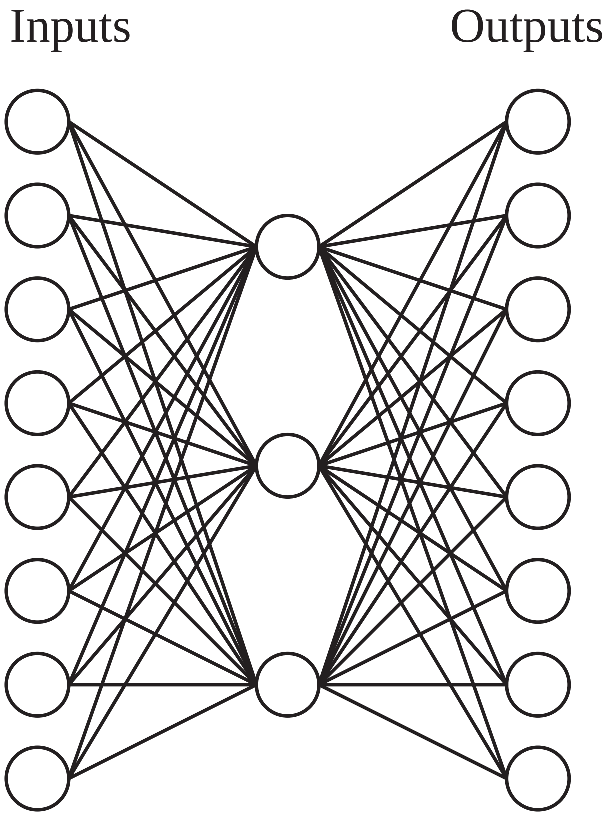
(see Tom Mitchell's "Machine Learning" book, pag. 108-112)



Plots of the error E, as a function of the number of weights updates, for two different robot perception tasks.

3.3 Learning Hidden Layer Representations

An Example: Learning the identity function $f(\vec{x}) = \vec{x}$



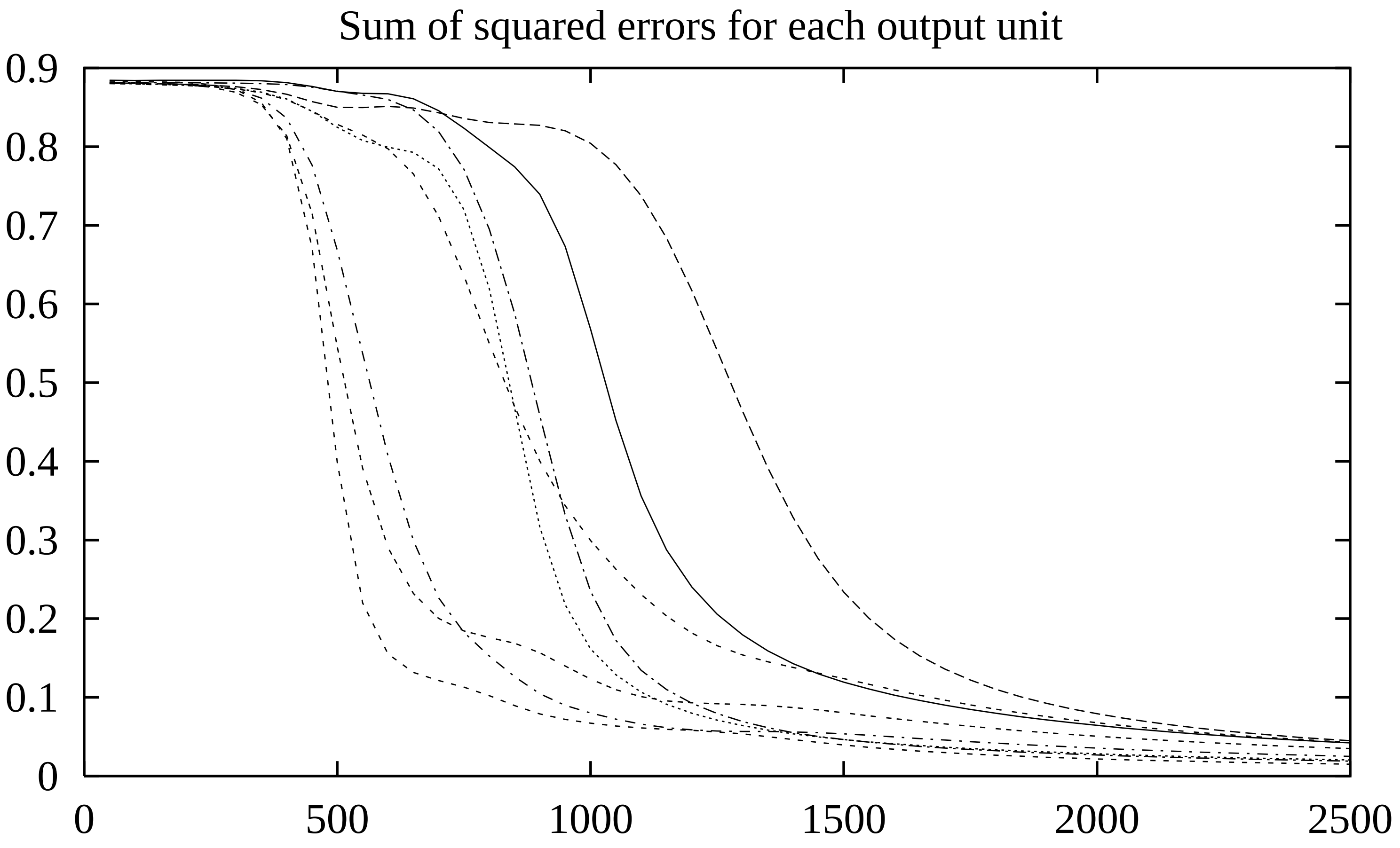
Input		Output
10000000	→	10000000
01000000	→	01000000
00100000	→	00100000
00010000	→	00010000
00001000	→	00001000
00000100	→	00000100
00000010	→	00000010
00000001	→	00000001

Learned hidden layer representation:

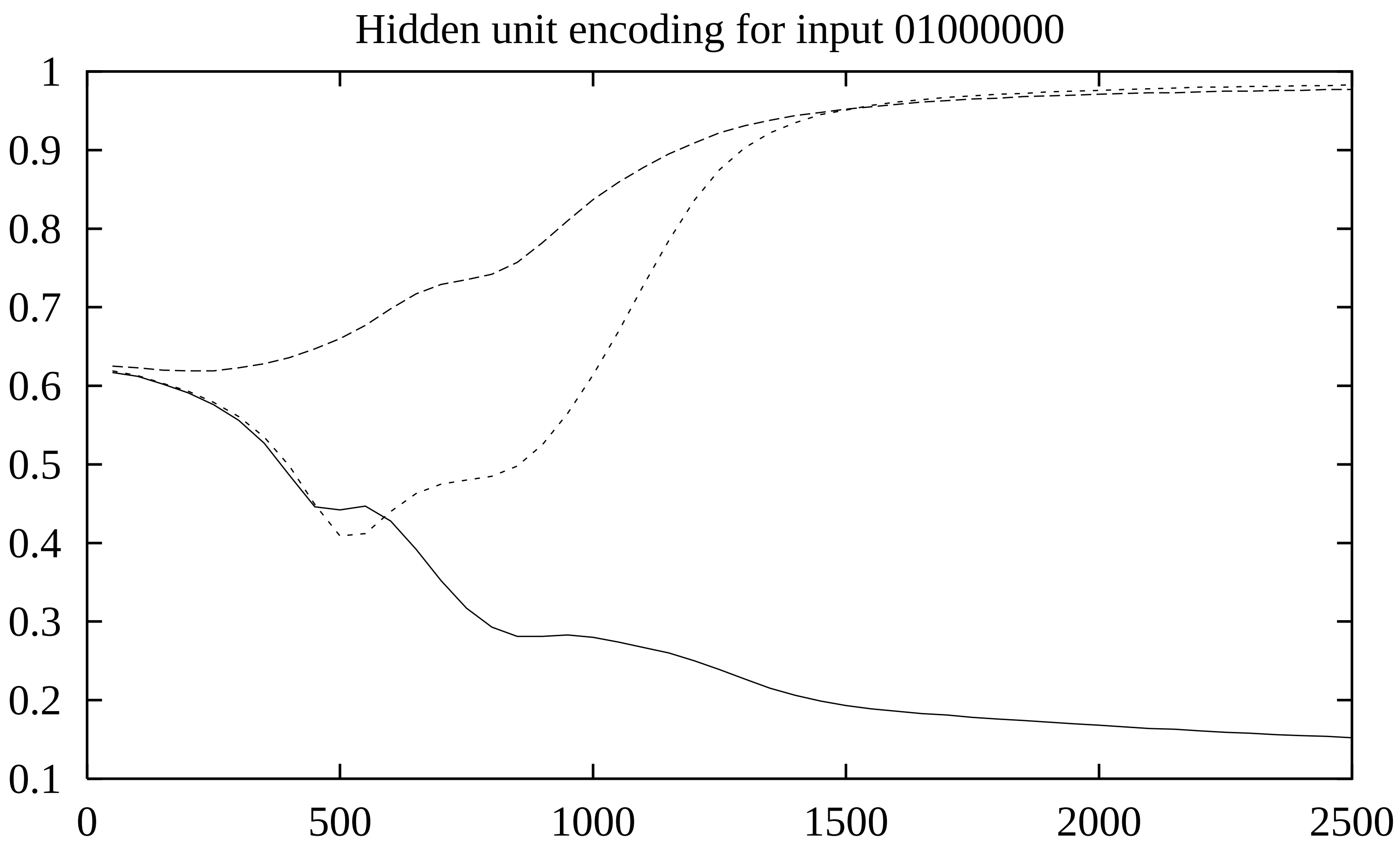
Input		Hidden Values				Output
10000000	→	.89	.04	.08	→	10000000
01000000	→	.15	.99	.99	→	01000000
00100000	→	.01	.97	.27	→	00100000
00010000	→	.99	.97	.71	→	00010000
00001000	→	.03	.05	.02	→	00001000
00000100	→	.01	.11	.88	→	00000100
00000010	→	.80	.01	.98	→	00000010
00000001	→	.60	.94	.01	→	00000001

After 8000 training epochs, the 3 hidden unit values encode the 8 distinct inputs. Note that if the encoded values are rounded to 0 or 1, the result is the standard binary encoding for 8 distinct values (however not the usual one, i.e. $1 \rightarrow 001,\ 2 \rightarrow 010$**, etc).**

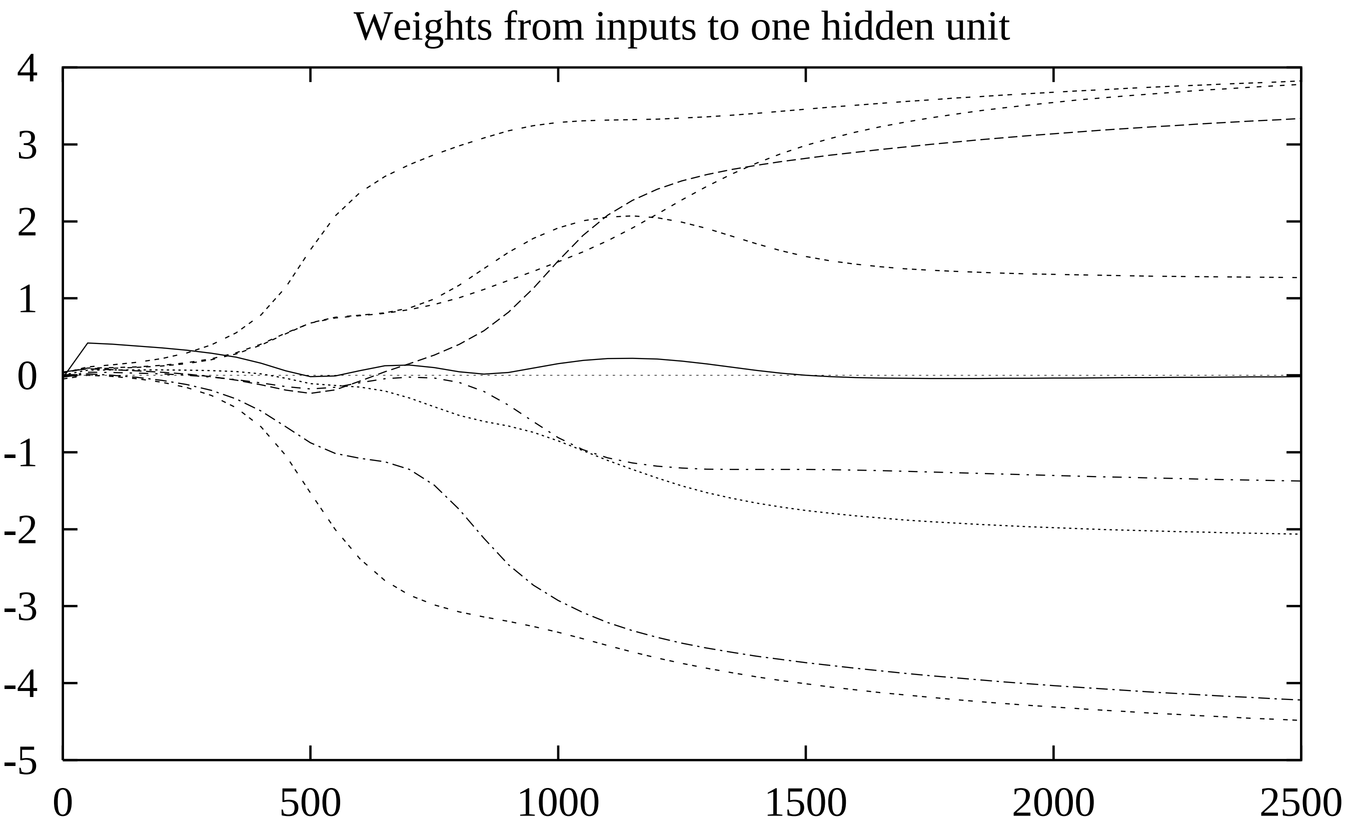
Training (I)


Sum of squared errors for each output unit
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0
500
1000
1500
2000
2500

Training (II)


Hidden unit encoding for input 01000000
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
500
1000
1500
2000
2500

Training (III)



4. Advanced Topics

4.1 Alternative Error Functions (see *ML* book, pag. 117-118):

- 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

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

- Tie together weights: e.g., in phoneme recognition network;
- Minimizing the cross entropy (see next 3 slides):

$$-\sum_{d \in D} t_d \log o_d + (1 - t_d) \log(1 - o_d)$$

where o_d, the output of the network, represents the estimated probability that the training instance x_d is associated the label (target value) 1.

4.2 Predicting a probability function:

Learning the ML hypothesis using a NN

(see Tom Mitchell's "Machine Learning" book, pag. 118, 167-171)

Let us consider a non-deterministic function (LC: one-to-many relation) $f : X \to \{0, 1\}$**.**

Given a set of independently drawn examples
$D = \{< x_1, d_1 >, \ldots, < x_m, d_m >\}$ **where** $d_i = f(x_i) \in \{0, 1\}$**,**
we would like to learn the probability function $g(x) \overset{def.}{=} P(f(x) = 1)$**.**
The ML hypotesis $h_{ML} = \boldsymbol{argmax}_{h \in H}\, P(D \mid h)$ **in such a setting is**

$$h_{ML} = \boldsymbol{argmax}_{h \in H}\, G(h, D)$$

where $G(h, D) = \sum_{i=1}^{m} [d_i\, \boldsymbol{ln}\, h(x_i) + (1 - d_i)\, \boldsymbol{ln}\, (1 - h(x_i))]$

We will use a NN for this task.
For simplicity, a single layer with sigmoidal units is considered.
The training will be done by gradient ascent:

$$\vec{w} \leftarrow \vec{w} + \eta \nabla G(D, h)$$

The partial derivative of $G(D,h)$ with respect to w_{jk}, which is the weight for the kth input to unit j, is:

$$
\begin{aligned}
\frac{\partial G(D,h)}{\partial w_{jk}} &= \sum_{i=1}^{m} \frac{\partial G(D,h)}{\partial h(x_i)} \cdot \frac{\partial h(x_i)}{\partial w_{jk}} \\
&= \sum_{i=1}^{m} \frac{\partial (d_i \,\boldsymbol{ln}\, h(x_i) + (1-d_i)\,\boldsymbol{ln}\,(1-h(x_i)))}{\partial h(x_i)} \cdot \frac{\partial h(x_i)}{\partial w_{jk}} \\
&= \ldots \\
&= \sum_{i=1}^{m} \frac{d_i - h(x_i)}{h(x_i)(1-h(x_i))} \cdot \frac{\partial h(x_i)}{\partial w_{jk}} \qquad \textbf{and because} \\
\frac{\partial h(x_i)}{\partial w_{jk}} &= \sigma'(x_i)\, x_{i,\,jk} = h(x_i)(1-h(x_i))\, x_{i,\,jk} \qquad \textbf{it follows that} \\
\frac{\partial G(D,h)}{\partial w_{jk}} &= \sum_{i=1}^{m} (d_i - h(x_i))\, x_{i,\,jk}
\end{aligned}
$$

Note: Here above we denoted $x_{i,\,jk}$ the kth input to unit j for the ith training example, and σ' the derivative of the sigmoid function.

Therefore

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

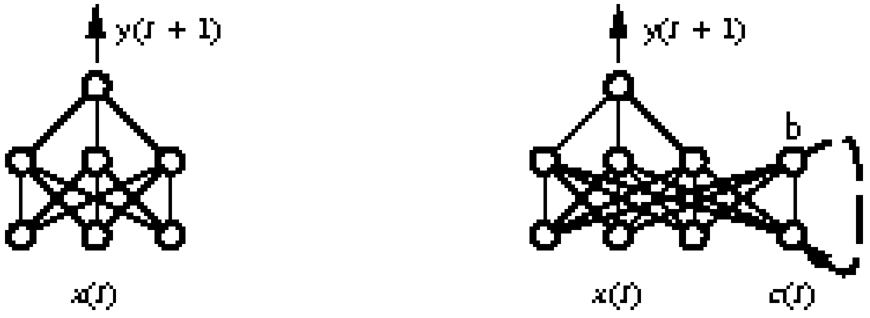
with

$$\Delta w_{jk} = \eta \frac{\partial G(D,h)}{\partial w_{jk}} = \eta \sum_{i=1}^{m} (d_i - h(x_i)) x_{i,jk}$$

4.3 Recurrent Networks

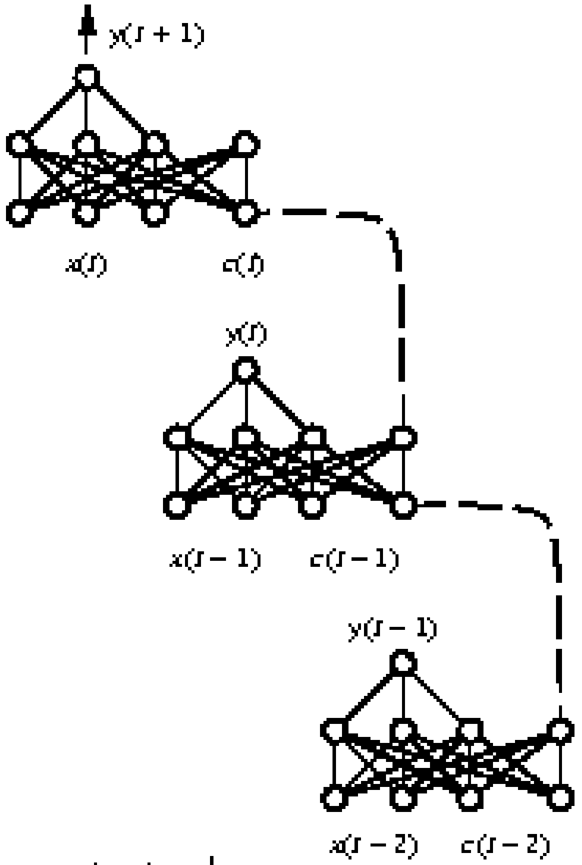
- applied to time series data
- can be trained using a version of Backpropagation algorithm
- see [Mozer, 1995]

An example:



(a) Feedforward network

(b) Recurrent network



(c) Recurrent network unfolded in time

4.4 Dynamically Modifying the Network Structure

Two ideas:

- **Begin with a network with no hidden unit, then grow the network until the training error is reduced to some acceptable level.**

  Example: Cascade-Correlation algorithm, [Fahlman & Lebiere, 1990]

- **Begin with a complex network and prune it as you find that certain connections w are inessential.**
  E.g. see whether $w \simeq 0$, or analyze $\frac{\partial E}{\partial w}$, i.e. the effect that a small variation in w has on the error E.

  Example: [LeCun et al. 1990]

4.5 Alternative Optimisation Methods for Training ANNs

See Tom Mitchell's "Machine Learning" book, pag. 119

- **linear search**
- **conjugate gradient**

4.6 Other Advanced Issues

- Ch. 6:

  A Bayesian justification for choosing to minimize the sum of square errors

- Ch. 7:

  The estimation of the number of needed training examples to reliably learn boolean functions;

  The Vapnik-Chervonenkis dimension of certain types of ANNs

- Ch. 12:

  How to use prior knowledge to improve the generalization acuracy of ANNs

5. Expressive Capabilities of [Feed-forward] ANNs

Boolean functions:

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

Continuous functions:

- Every bounded continuous function can be approximated with arbitrarily small error, by a 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].

Summary / What you should know

○ The gradient descent optimisation method

● The thresholded perceptron;
the training rule, the test rule;
convergence result

The linear unit and the sigmoid unit;
the gradient descent rule (including the proofs);
convergence result

● Multilayer networks of sigmoid units;
the Backpropagation algorithm
(including the proof for the stochastic version);
convergence result

○ Batch/online vs stochastic/incremental gradient descent
for artifical neurons and neural networks;
convergence result

○ Overfitting in neural networks; solutions