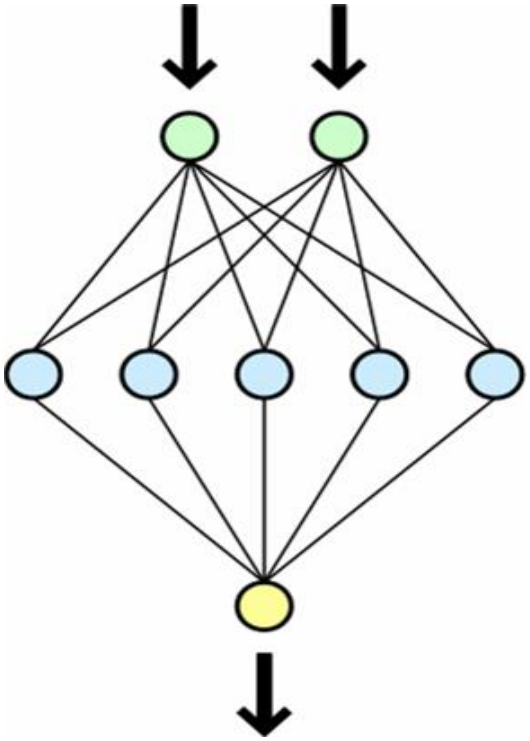
Computational learning and discovery

CSI 873 / MATH 689

Instructor: I. Griva

Wednesday 7:20 – 10:00 pm

Artificial Neural Networks (ANN) -



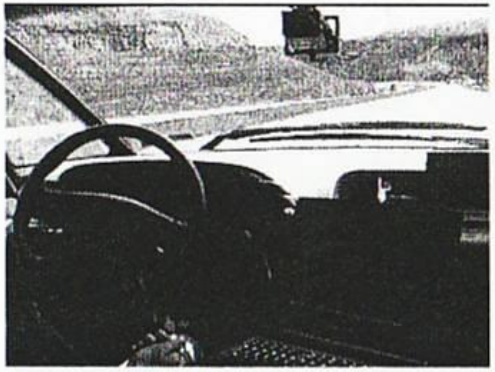
method for learning discrete-valued, real-vaued and vector-valued functions.

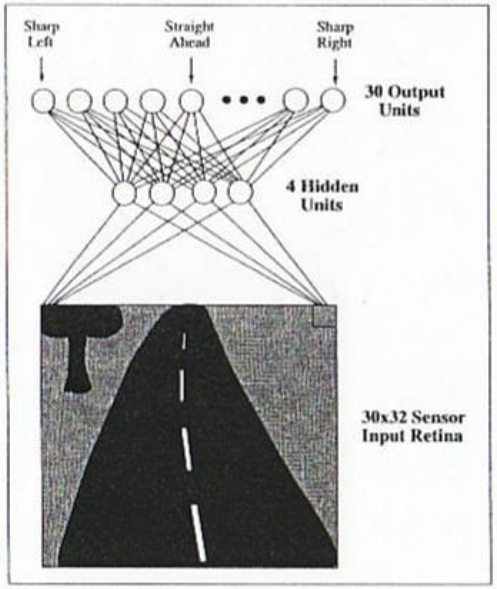
ANN is robust to errors in the training data and has been successfully applied to problems of interpreting sensor data and visual scenes, speech recognition, face recognition, learning robot control strategies.

Study of ANN has been inspired in part by the observation that biological learning systems are built out of complex web of interconnected neurons.

Example:

ALVINN – Autonomous Land Vehicle In a Neural Network





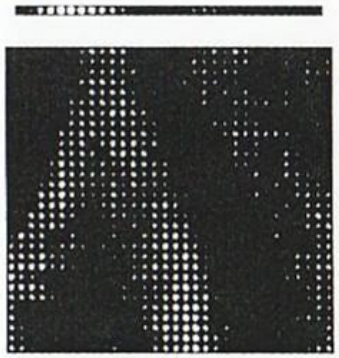


FIGURE 4.1
Neural network learning to steer an autonomous vehicle. The ALVINN system uses BACKPROPAGATION to learn to steer an autonomous vehicle (photo at top) driving at speeds up to 70 miles per hour. The diagram on the left shows how the image of a forward-mounted camera is mapped to 960 neural network inputs, which are fed forward to 4 hidden units, connected to 30 output units. Network outputs encode the commanded steering direction. The figure on the right shows weight values for one of the hidden units in this network. The 30 × 32 weights into the hidden unit are displayed in the large matrix, with white blocks indicating positive and black indicating negative weights. The weights from this hidden unit to the 30 output units are depicted by the smaller rectangular block directly above the large block. As can be seen from these output weights, activation of this particular hidden unit encourages a turn toward the left.

Appropriate problems for ANN learning

ANN learning is well-suited to problems in which the training data corresponds to noisy, complex sensor data such as inputs from cameras and microphones.

ANN is also good for the problems suitable for decision tree learning, where symbolic representations are used

In more detail...

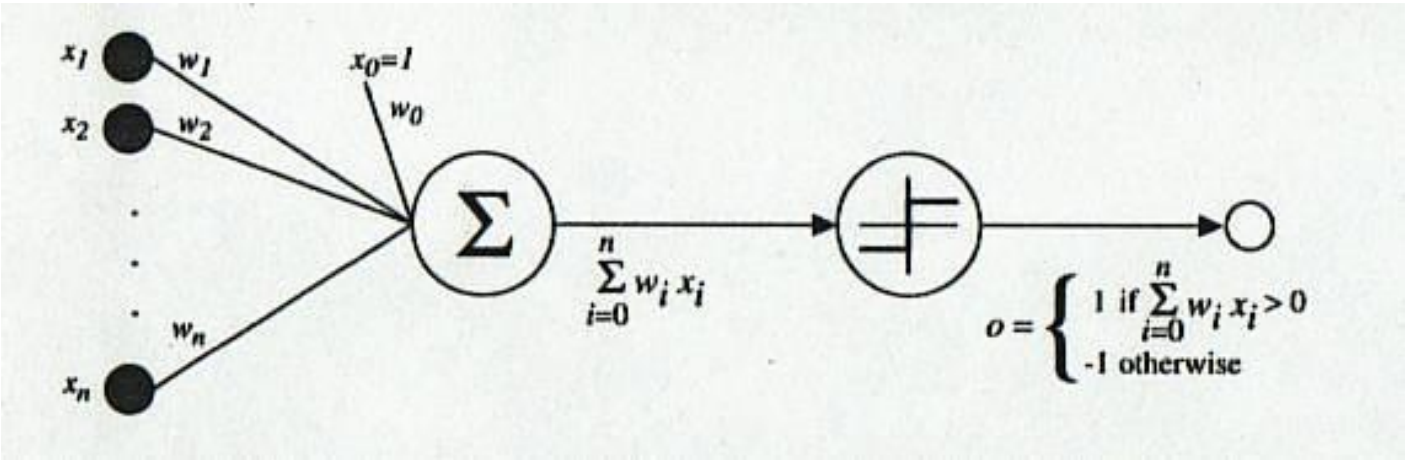
- **Instances represented by many attribute-value pairs**
- **The target function may be discrete-valued, real valued, or vector of several real- or discrete valued attributes.**
- **The training examples may contain errors**
- **Long training times are acceptable**
- **Fast evaluation of the learned target function may be required**
- **The ability of humans to understand the learned target functions is not important**

Perceptron – the basic ANN unit and the simplest ANN

$$o(x_1,\ldots,x_n)=\begin{cases}1 & \text{if } w_0+w_1x_1+w_2x_2+\cdots w_nx_n>0\\ -1 & \text{otherwise}\end{cases}$$

$$o(x_1,\ldots,x_n)=\operatorname{sgn}(\vec{w}\cdot\vec{x}),$$

$$\text{where } \vec{w}=(w_0,w_1,\ldots,w_n),\ \vec{x}=(x_0,x_1,\ldots,x_n),\ x_0=1$$



Learning perceptron means finding

$$\vec{w}=(w_0,w_1,\ldots,w_n)\in\Re^{n+1}$$ ← **Hypothesis space**

Perceptron training rule

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

where

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

t - target output

o - the output generated by the perceptron

$\eta > 0$ - a constant called the *learning rate*

Perceptron training rule converges for linearly separable training examples

$$o(x_1, \ldots, x_n) = \begin{cases} 1 & \text{if } w_0 + w_1 x_1 + w_2 x_2 + \cdots w_n x_n > 0 \\ -1 & \text{otherwise} \end{cases}$$

Delta rule – for linearly not separable training examples

The key idea is to use the gradient descent method.

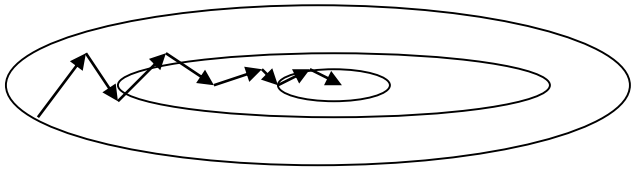
Unconstrained optimization problem

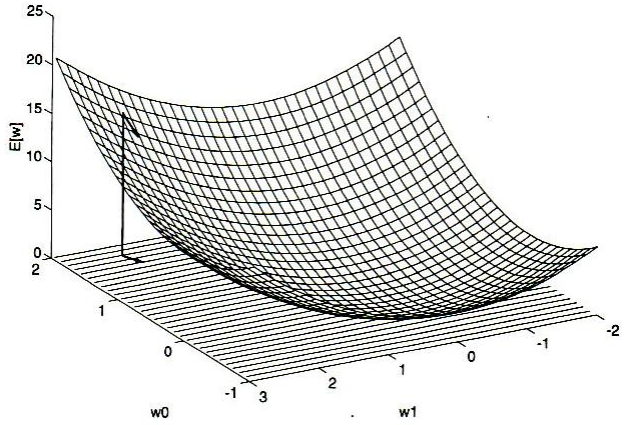
$$f : \Re^n \to \Re^1$$

$$\min f(x), \quad x \in \Re^n$$

Gradient descent method

$$x^{s+1} = x^s - \alpha_s \nabla f(x^s), \quad s = 1,2,....$$





Incremental gradient descent, or stochastic gradient descent method uses perceptron training rule

Error is defined only for current training example

$$E(\vec{w}) = 0.5(t_d - o_d)^2$$

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

where

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

t - target output

o - the output generated by the perceptron

$\eta > 0$ - a constant called the *learning rate*

Unthresholded perceptron

$$o(\vec{x}) = w_0 + w_1 x_1 + w_2 x_2 + \cdots w_n x_n = (\vec{w} \cdot \vec{x})$$

Training error

$$E(\vec{w}) = 0.5 \sum_{d \in D} (t_d - o_d)^2$$

Gradient of the training error

$$\nabla E(\vec{w}) = \left[\frac{\partial E}{w_0}, \frac{\partial E}{w_1}, \cdots, \frac{\partial E}{w_n} \right]$$

Delta update rule

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

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

or

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

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

Delta update rule

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

$$= \frac{1}{2}\sum_{d\in D}\frac{\partial}{\partial w_i}(t_d - o_d)^2$$

$$= \frac{1}{2}\sum_{d\in D}2(t_d - o_d)\frac{\partial}{\partial w_i}(t_d - o_d)$$

$$= \sum_{d\in D}(t_d - o_d)\frac{\partial}{\partial w_i}(t_d - \vec{w}\cdot\vec{x_d})$$

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

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

The key differences between standard gradient descent a stochastic gradient descent methods:

standard gradient descent	stochastic gradient descent
error is summed over all examples before updating weights	weights are updated upon examining each training example
more computational for per weight update step, but the larger step	less computational for per weight update step, but the smaller step
greater chance to stuck in a local minimum	smaller chance to stuck in a local minimum
converges for linearly non-separable case	does not converge for linearly non-separable case

Liner Programming also can be used to find the weights

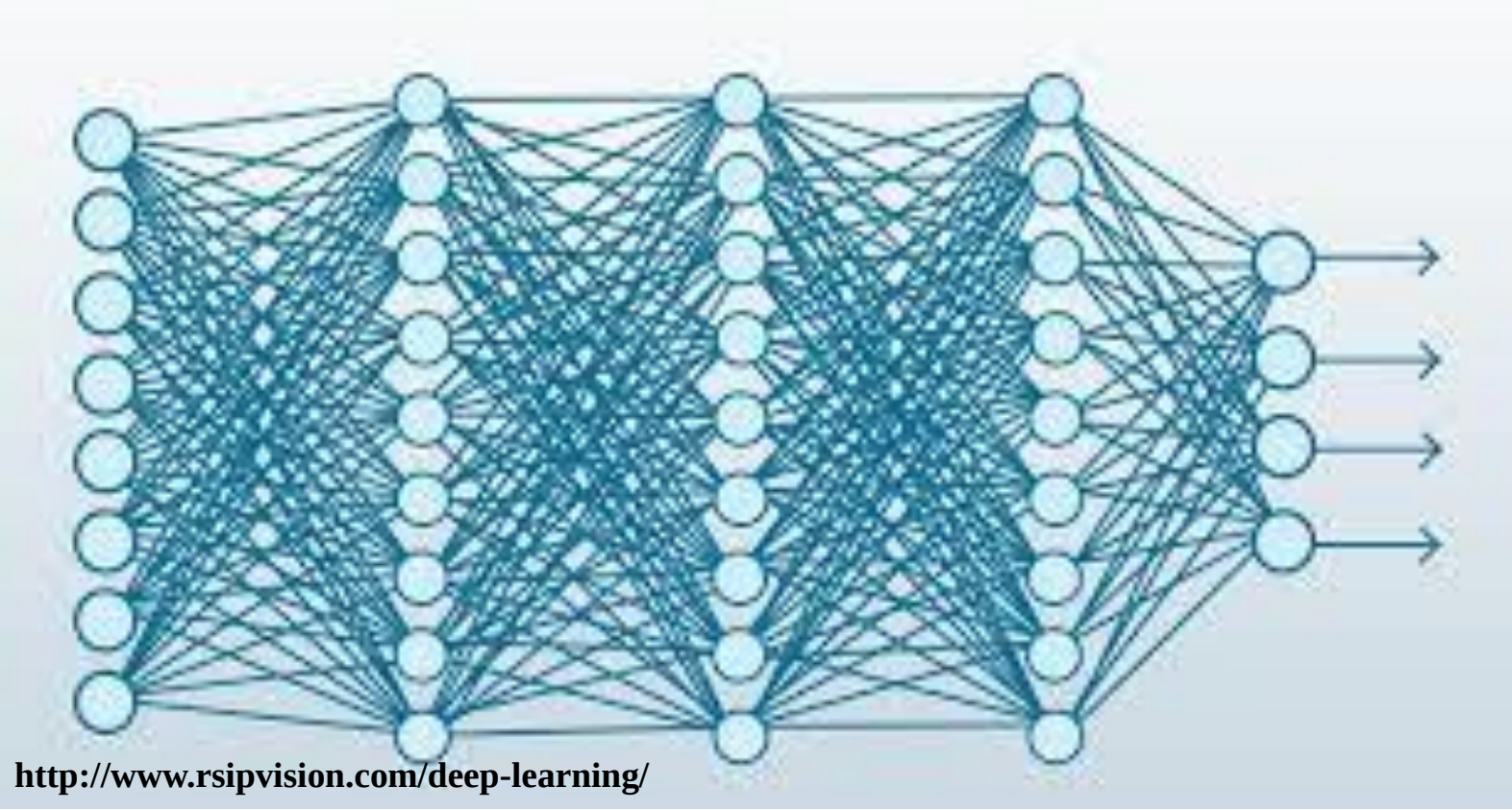
Representational power of Feedforward Networks

- **Boolean functions can be represented by a network with two layers of units, but may require a large number of units**

- **Continuous functions can be approximated with arbitrary small error by a network with two layers of units (Cybenko, 1989, Hornik et al. 1989)**

- **Any function bounded function with a finite number of discontinuities can be approximated with arbitrary accuracy by a network with three layers of units (Cybenko, 1988)**

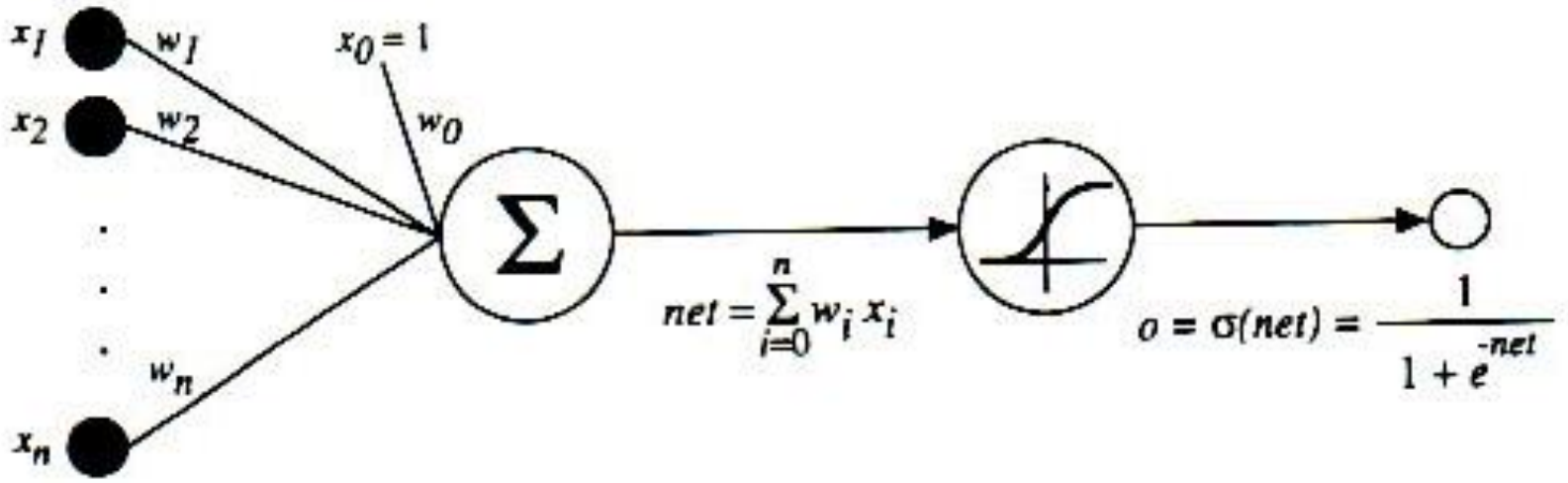
Deep Learning



http://www.rsipvision.com/deep-learning/

Yoshua Bengio and Yann LeCun, *Scaling Learning Algorithms towards AI*, Large-Scale Kernel Machine, 2007.

Sigmoid unit



$$o(x_1, \ldots, x_n) = \sigma(\vec{w} \cdot \vec{x}),$$

where

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

$$\frac{d\sigma(y)}{dy} = \sigma(y)(1-\sigma(y))$$

Backpropagation algorithm-

application of the gradient descent method to the error over all output units

$$E(\vec{w}) = 0.5 \sum_{d \in D} \sum_{k \in Outputs} (t_{kd} - o_{kd})^2$$

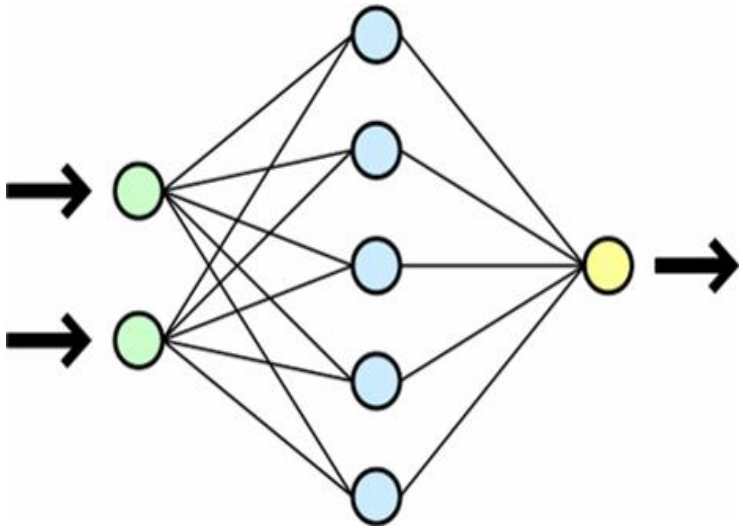
The gradient descent can be standard or stochastic (incremental)

$$E_d(\vec{w}) = 0.5 \sum_{k \in Outputs} (t_{kd} - o_{kd})^2$$

Notations

- x_{ji} = the ith input to unit j
- w_{ji} = the weight associated with the ith input to unit j
- $net_j = \sum_i w_{ji} x_{ji}$ (the weighted sum of inputs for unit j)
- o_j = the output computed by unit j
- t_j = the target output for unit j
- σ = the sigmoid function
- $outputs$ = 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

$$E_d(\vec{w}) = 0.5 \sum_{k \in Outputs} (t_k - o_k)^2$$



Derivations for output unit weights

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

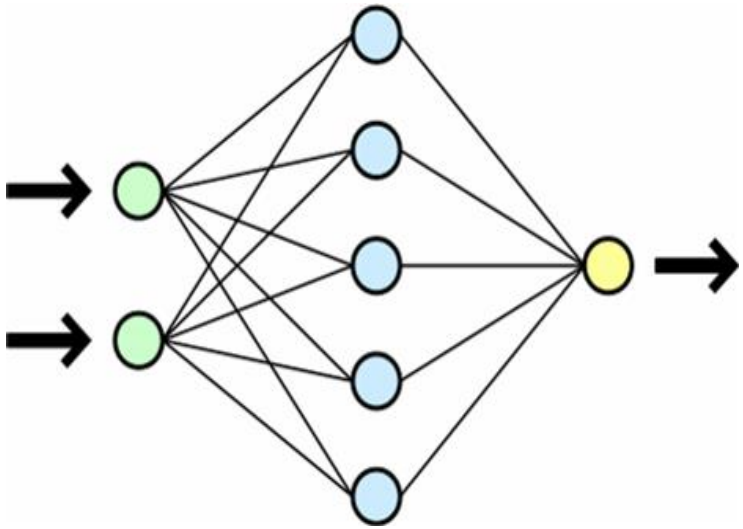
$$\frac{\partial E_d}{\partial net_j} = \frac{\partial E_d}{\partial o_j}\frac{\partial o_j}{\partial net_j}$$

$$\frac{\partial E_d}{\partial o_j} = \frac{\partial}{\partial o_j}\frac{1}{2}(t_j - o_j)^2 = \frac{1}{2}2(t_j - o_j)\frac{\partial (t_j - o_j)}{\partial o_j} = -(t_j - o_j)$$

$$\frac{\partial o_j}{\partial net_j} = \frac{\partial \sigma(net_j)}{\partial net_j} \doteq o_j(1 - o_j)$$

$$\frac{\partial E_d}{\partial net_j} = -(t_j - o_j)\, o_j(1 - o_j) = -\delta_j$$

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



Derivations for hidden unit weights

$$\frac{\partial E_d}{\partial net_j} = \sum_{k \in Downstream(j)} \frac{\partial E_d}{\partial net_k} \frac{\partial net_k}{\partial net_j}$$

$$= \sum_{k \in Downstream(j)} -\delta_k \frac{\partial net_k}{\partial net_j}$$

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

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

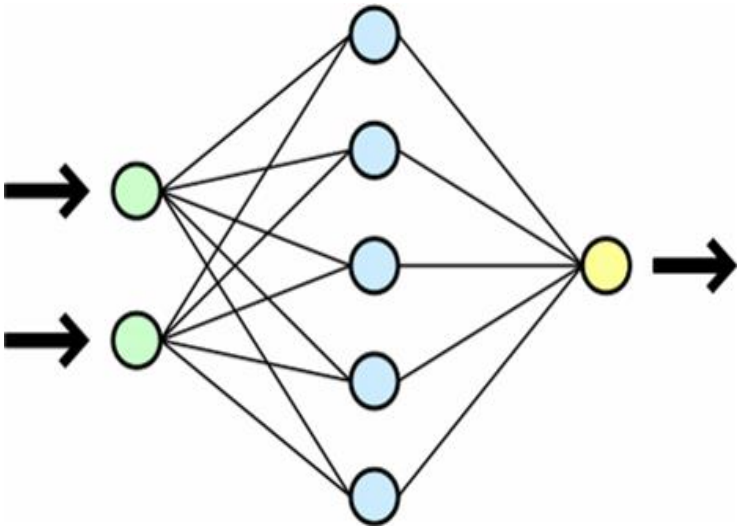
$$= \sum_{k \in Downstream(j)} -\delta_k \; w_{kj} \; o_j(1 - o_j)$$

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

$$\Delta w_{ji} = \eta \; \delta_j \; x_{ji}$$

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

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



Back propagation 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

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

  3. For each hidden unit h

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

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

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

where

$$\Delta w_{ji} = \eta \delta_j x_{ji}$$

Convergence

BP algorithm is guaranteed to find a local, not a global error minimum

Adding momentum may be useful

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

Keeping weights small may be useful

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

Convergence, practical advices

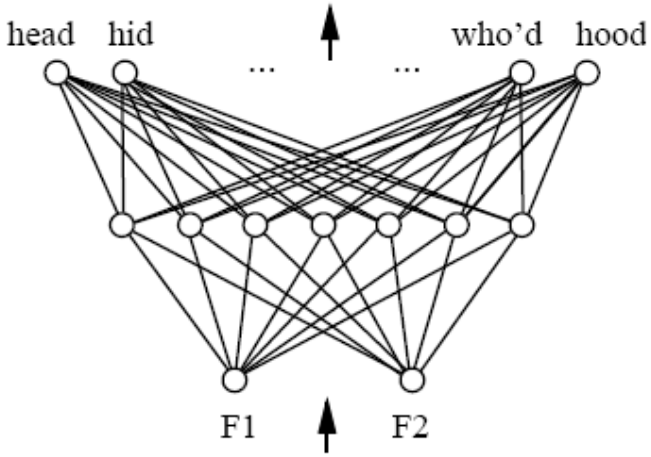
Gradient descent to some local minimum

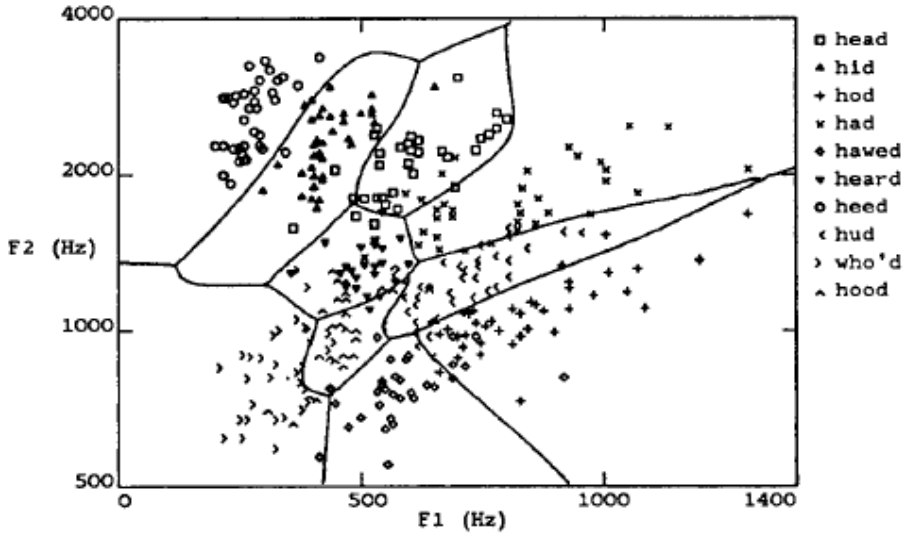
- Perhaps not global minimum...
- Add momentum
- Stochastic gradient descent
- Train multiple nets with different inital weights

Nature of convergence

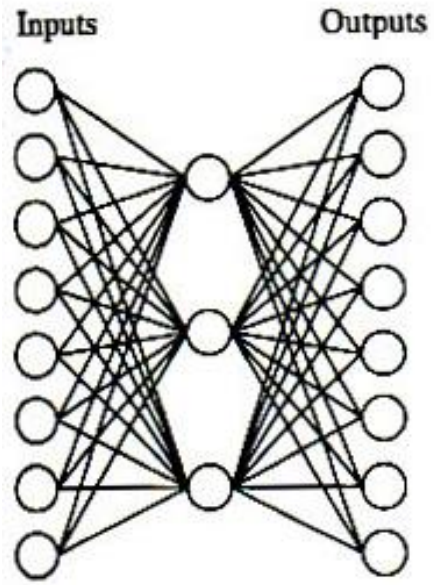
- Initialize weights near zero
- Therefore, initial networks near-linear
- Increasingly non-linear functions possible as training progresses

Hypothesis space and inductive bias



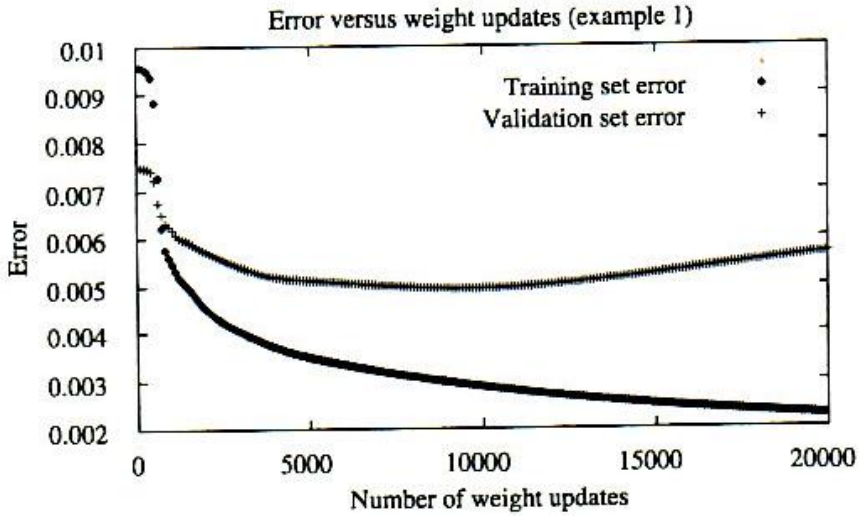


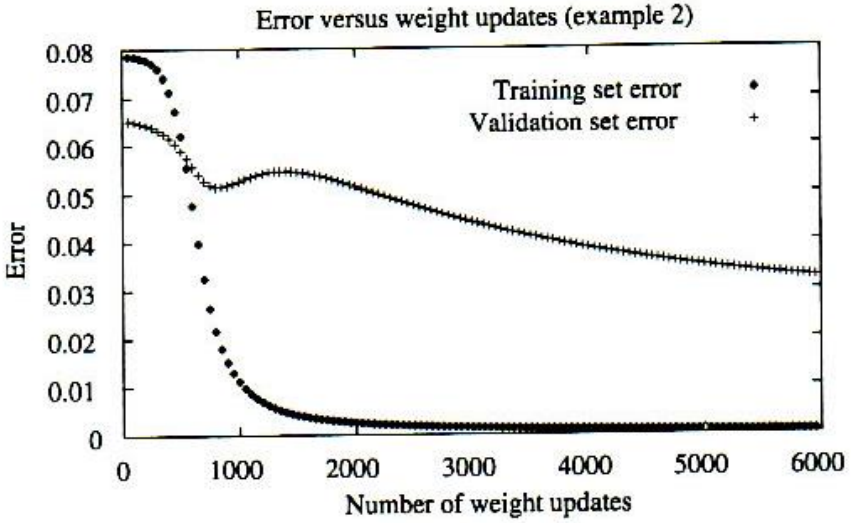
Hidden Layers Representation



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

Ovefitting, stopping criteria





Keep small weights!

Example: face recognition

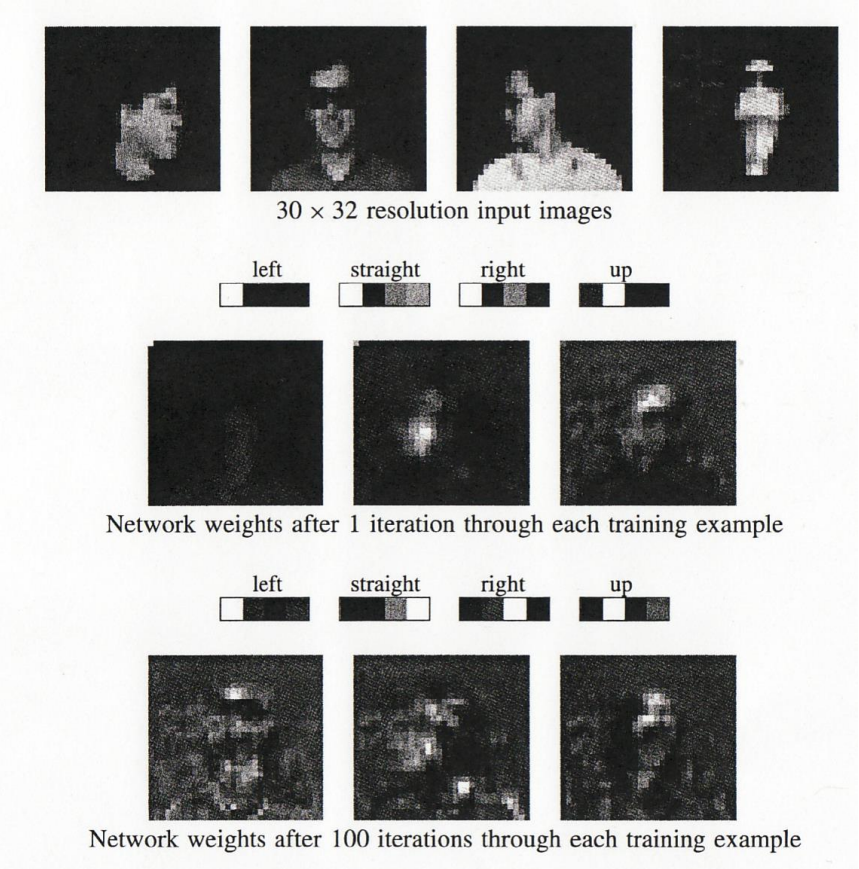


FIGURE 4.10
Learning an artificial neural network to recognize face pose. Here a $960 \times 3 \times 4$ network is trained on grey-level images of faces (see top), to predict whether a person is looking to their left, right, ahead, or up. After training on 260 such images, the network achieves an accuracy of 90% over a separate test set. The learned network weights are shown after one weight-tuning iteration through the training examples and after 100 iterations. Each output unit (left, straight, right, up) has four weights, shown by dark (negative) and light (positive) blocks. The leftmost block corresponds to the weight w_0, which determines the unit threshold, and the three blocks to the right correspond to weights on inputs from the three hidden units. The weights from the image pixels into each hidden unit are also shown, with each weight plotted in the position of the corresponding image pixel.

Alternative error functions

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

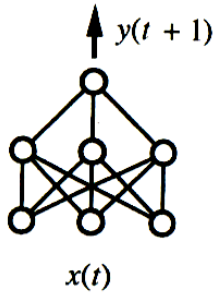
$$E(\vec{w}) \equiv \frac{1}{2} \sum_{d \in D} \sum_{k \in outputs} \left[(t_{kd} - o_{kd})^2 + \mu \sum_{j \in inputs} \left(\frac{\partial t_{kd}}{\partial x_d^j} - \frac{\partial o_{kd}}{\partial x_d^j} \right)^2 \right]$$

$$E(\vec{w}) \equiv - \sum_{d \in D} \left[t_d \log o_d + (1 - t_d) \log(1 - o_d) \right]$$

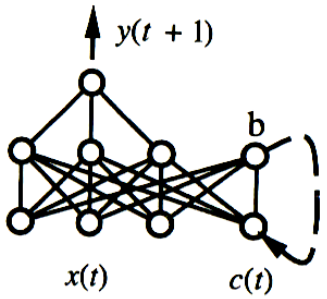
Weight sharing

Alternative minimization procedures

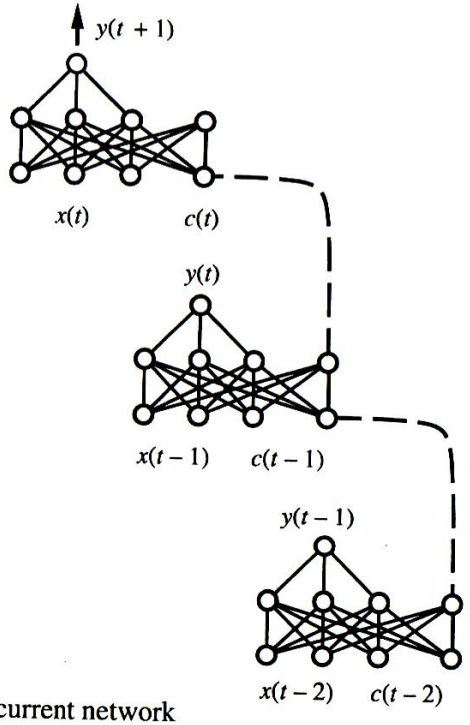
Recurrent networks



(*a*) Feedforward network



(*b*) Recurrent network



(*c*) Recurrent network unfolded in time

Dynamically modifying ANN

- **Adding complexity**
- **Pruning**

if w_{ji} **is small, then eliminate the link**

if $\frac{\partial E}{w_{ji}}$ **is small, then eliminate the link**

SUMMARY

- ANNs provide error robust methods for learning real-valued, discrete-valued, vector-valued functions

- ANNs are able to approximate any function to arbitrary accuracy, given a sufficient number of units in each layer

- Back propagation algorithm is popular method for learning network parameters

- Overfitting of training data is important issue for ANNs. Cross-validation methods are used to estimate an appropriate stopping point