IIT KHARAGPUR

NPTEL ONLINE CERTIFICATION COURSES

Data Mining

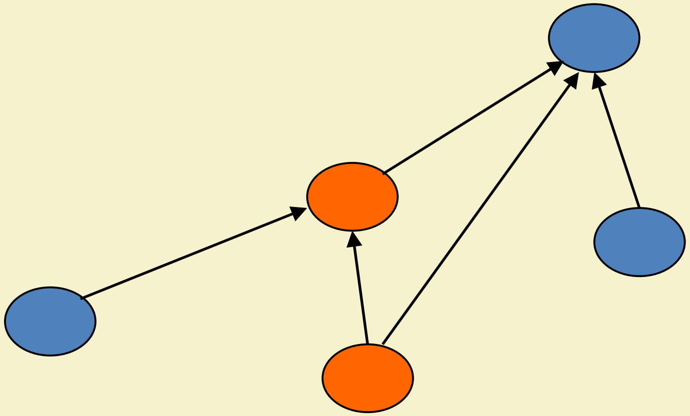
Week 6: Artificial Neural Networks

Pabitra Mitra

Computer Science and Engineering, IIT Kharagpur

Neural networks

- Networks of processing units (neurons) with connections (synapses) between them
- Large number of neurons: 10^{10}
- Large connectitivity: 10^5
- Parallel processing
- Distributed computation/memory
- Robust to noise, failures

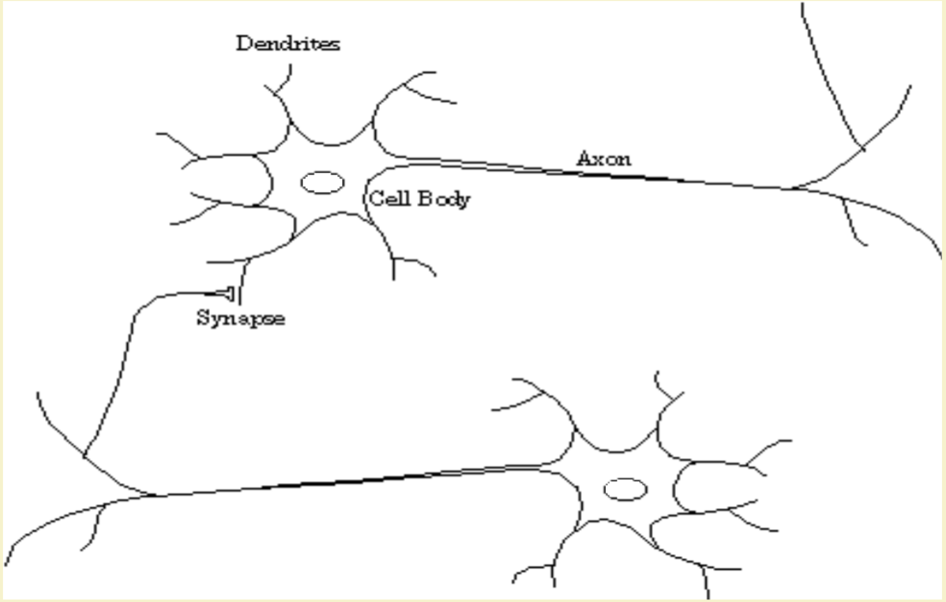


Connectionism

- Alternative to *symbolism*
- Humans and evidence of connectionism/parallelism:
  - Physical structure of brain:
    - Neuron switching time: 10^{-3} second
  - Complex, short-time computations:
    - Scene recognition time: 10^{-1} second
    - 100 inference steps doesn't seem like enough

      → much parallel computation
- Artificial Neural Networks (ANNs)
  - Many neuron-like threshold switching units
  - Many weighted interconnections among units
  - Highly parallel, distributed process
  - Emphasis on tuning weights automatically (search in weight space)

Biological neuron



Biological neuron

- dendrites: nerve fibres carrying electrical signals to the cell
- cell body: computes a non-linear function of its inputs
- axon: single long fiber that carries the electrical signal from the cell body to other neurons
- synapse: the point of contact between the axon of one cell and the dendrite of another, regulating a chemical connection whose strength affects the input to the cell.

Biological neuron

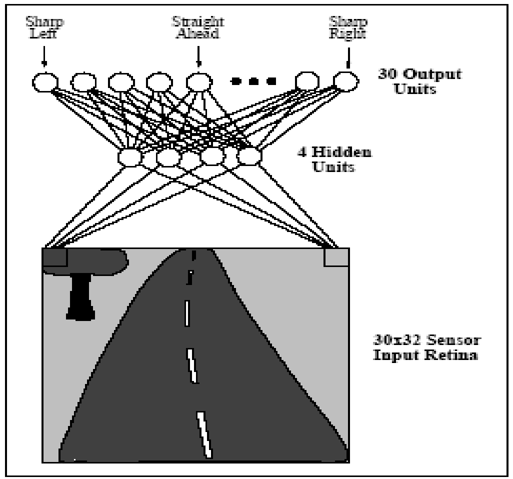
- A variety of different neurons exist (motor neuron, on-center off-surround visual cells…), with different branching structures
- The connections of the network and the strengths of the individual synapses establish the function of the network.

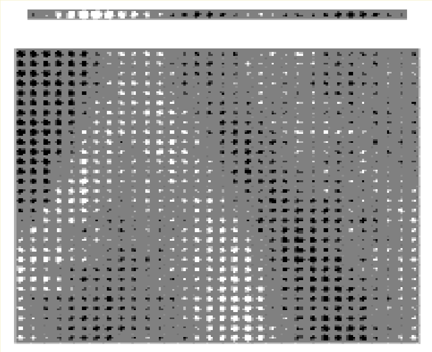
When to consider ANNs

- Input is
  - high-dimensional
  - discrete or real-valued
    - e.g., raw sensor inputs
  - noisy
- *Long training times*
- Form of target function is unknown
- *Human readability is unimportant*
- Especially good for complex recognition problems
  - Speech recognition
  - Image classification
  - Financial prediction

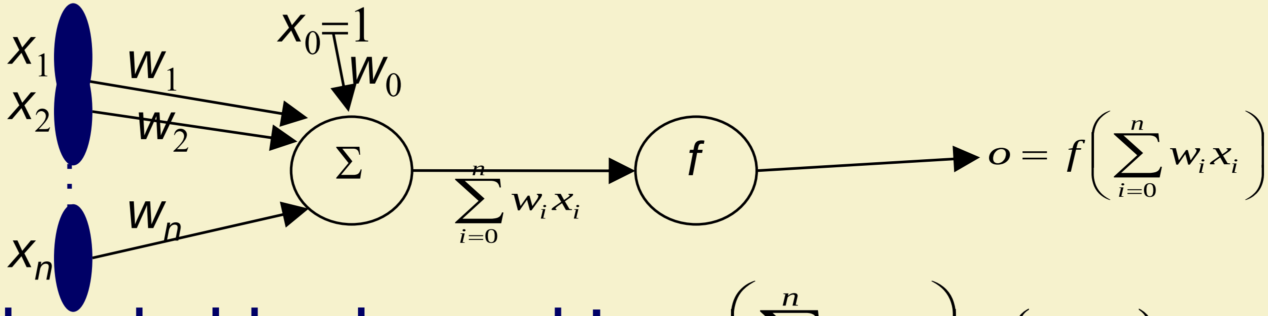
Problems too hard to program

- ALVINN: a perception system which learns to control the NAVLAB vehicles by watching a person drive





Perceptron

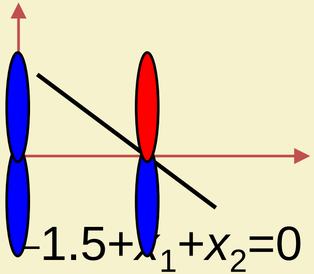


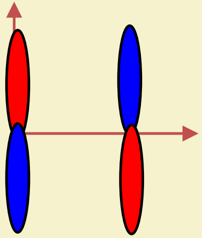
- $-w_0$: threshold value or bias $\left(\sum_{i=1}^{n} w_i x_i\right) - (-w_0)$
- *f (or o())* : activation function (thresholding unit), typically:

$$f(x) = \begin{cases} 1 & x > 0 \\ -1 & \text{otherwise} \end{cases}$$

Decision surface of a perceptron

- Decision surface is a hyperplane given by $\sum_{i=0}^{n} w_i x_i = 0$
- 2D case: the decision surface is a line
- Represents many useful functions: for example, $x_1 \wedge x_2$?
- $x_1 \vee x_2$?
- x_1 XOR x_2 ? Not linearly separable!

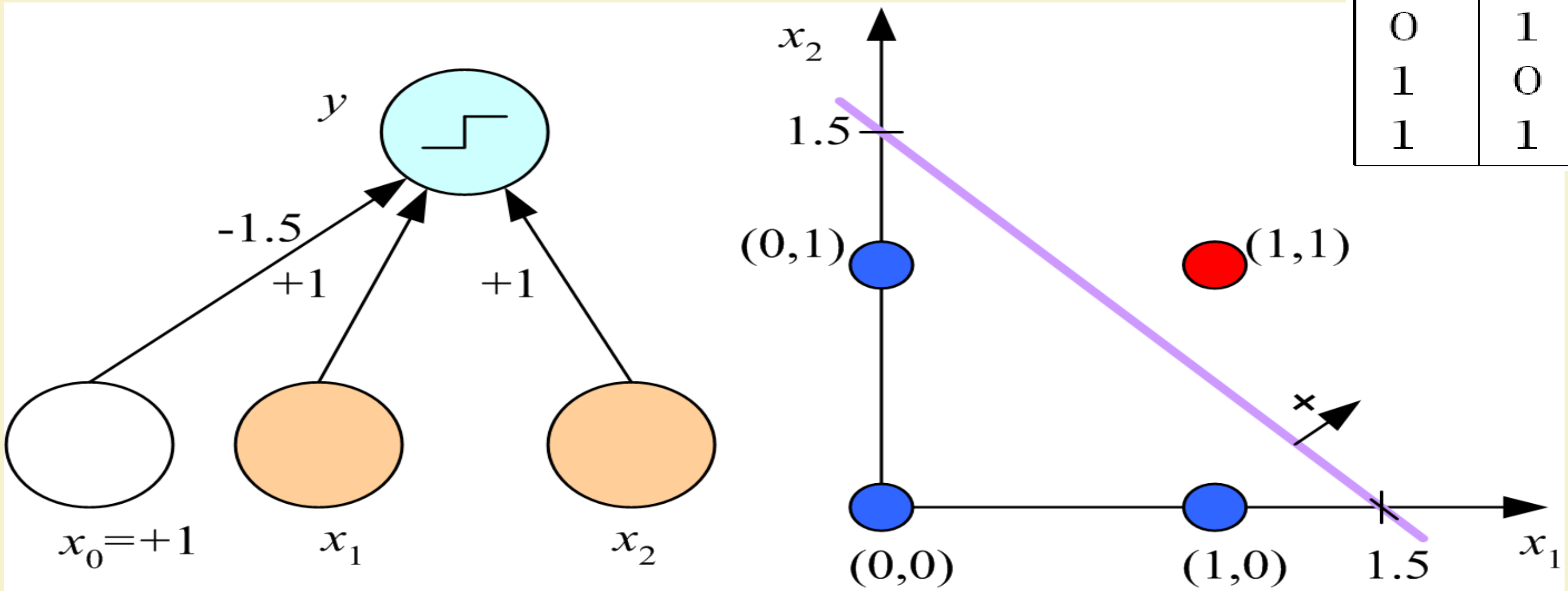




- Generalization to higher dimensions
  - Hyperplanes as decision surfaces

Learning Boolean AND

x_1	x_2	r
0	0	0
0	1	0
1	0	0
1	1	1

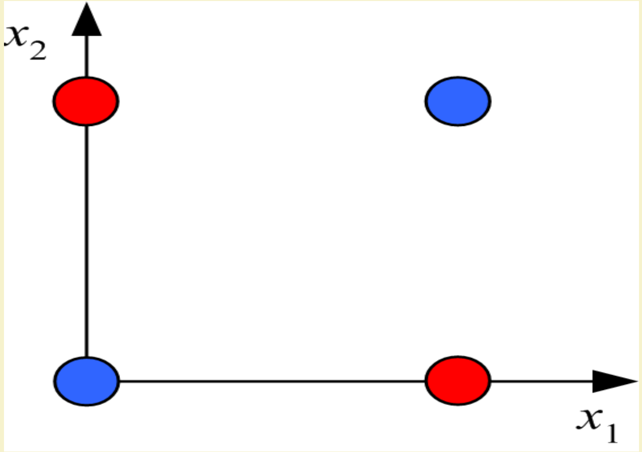


XOR

x_1	x_2	r
0	0	0
0	1	1
1	0	1
1	1	0

- No w_0, w_1, w_2 satisfy:

$$\begin{aligned} & & w_0 &\le 0 \\ & w_2 + & w_0 &> 0 \\ w_1 + & & w_0 &> 0 \\ w_1 + & w_2 + & w_0 &\le 0 \end{aligned}$$



(Minsky and Papert, 1969)

Boolean functions

- Solution:
  - network of perceptrons
  - Any boolean function representable as DNF
    - 2 layers
    - Disjunction (layer 1) of conjunctions (layer 2)
- Example of XOR
  - (X1=1 AND X2=0) OR (X1=0 AND X2=1)
- Practical problem of representing high-dimensional functions

Training rules

- Finding learning rules to build networks from TEs
- Will examine two major techniques
  - Perceptron training rule
  - Delta (gradient search) training rule (for more perceptrons as well as general ANNs)
- Both focused on learning weights
  - Hypothesis space can be viewed as set of weights

Perceptron training rule

- ITERATIVE RULE: $w_i := w_i + \Delta w_i$
  - where $\Delta w_i = \eta\,(t - o)\,x_i$
  - t is the target value
  - o is the perceptron output for x
  - η is small positive constant, called the learning rate
- Why rule works:
  - E.g., t = 1, o = -1, $x_i = 0.8$, $\eta = 0.1$
  - then $\Delta w_i = 0.16$ and $w_i x_i$ gets larger
  - o converges to t

Perceptron training rule

- The process will converge if
  - training data is linearly separable, and
  - η is sufficiently small
- But if the training data is not linearly separable, it may not converge (Minsky & Pappert)
  - Basis for Minsky/Pappert attack on NN approach
- Question: how to overcome problem:
  - different model of neuron?
  - different training rule?
  - both?

Gradient descent

- Solution: use alternate rule
  - More general
  - Basis for networks of units
  - Works in non-linearly separable cases
- Let $o(x) = w_0 + w_1 x_1 + \ldots + w_n x_n$
  - Simple example of linear unit (will generalize)
  - Omit the thresholding initially
- D is the set of training examples $\{d = \langle x, t_d \rangle\}$
- We will learn w_i's that minimize the squared error

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

Error minimization

- Look at error E as a function of weights {wi}
- Slide down gradient of E in weight space
- Reach values of {wi} that correspond to minimum error
  - Look for global minimum
- Example of 2-dimensional case:
  - $E = w1*w1 + w2*w2$
  - Minimum at $w1=w2=0$
- Look at general case of n-dimensional space of weights

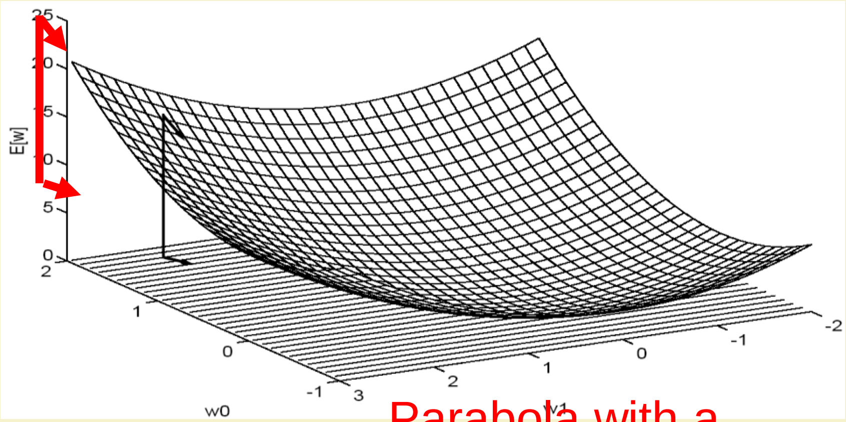
Gradient descent

- Gradient "points" to the steepest increase:

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

- Training rule: $\Delta \vec{w} = -\eta \nabla E[\vec{w}]$
  where η is a positive constant (learning rate)

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



Parabola with a single minima

How might one interpret this update rule?

Gradient descent

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

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

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

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

$$= \sum_{d \in D} (t_d - o_d)(-x_{i,d})$$

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

$$\Delta w_i = \sum_{d \in D} \left(\eta (t_d - o_d) x_{i,d} \right)$$

Gradient descent algorithm

Gradient-Descent (*training examples*, η)

Each training example is a pair $\langle x, t \rangle$: 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
- Repeat until the termination condition is met
  1. Initialize each Δw_i to zero
  2. For each training example $\langle x, t \rangle$
     - Input x to the unit and compute the output o
     - For each linear unit weight w_i

$$\Delta w_i \leftarrow \Delta w_i + \eta\,(t - o)\, x_i$$

  3. For each linear unit weight w_i

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

- At each iteration, consider reducing η

Also called
- LMS (Least Mean Square) rule
- Delta rule

Incremental (Stochastic) Gradient Descent

Batch mode Gradient Descent:

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

- Repeat
  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:

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

- Repeat
  - For each training example *d* in *D*
    1. Compute the gradient $\nabla E_d[\vec{w}]$
    2. $\vec{w} \leftarrow \vec{w} - \eta \nabla E_d[\vec{w}]$
- Incremental can approximate batch if η is small enough

Incremental Gradient Descent Algorithm

Incremental-Gradient-Descent (*training examples*, η)

Each training example is a pair $\langle x, t \rangle$*: 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
- Repeat until the termination condition is met
  1. Initialize each Δw_i to zero
  2. For each $\langle x, t \rangle$
     - Input x to the unit and compute output o
     - For each linear unit weight w_i

$$w_i \leftarrow w_i + \eta (t - o) x_i$$

Perceptron vs. Delta rule training

- Perceptron training rule guaranteed to succeed if
  - Training examples are linearly separable
  - Sufficiently small learning rate
- Delta training rule uses gradient descent
  - Guaranteed to converge to hypothesis with minimum squared error
    - Given sufficiently small learning rate
    - Even when training data contains noise
    - Even when training data not linearly separable
- Can generalize linear units to units with threshold
  - Just threshold the results

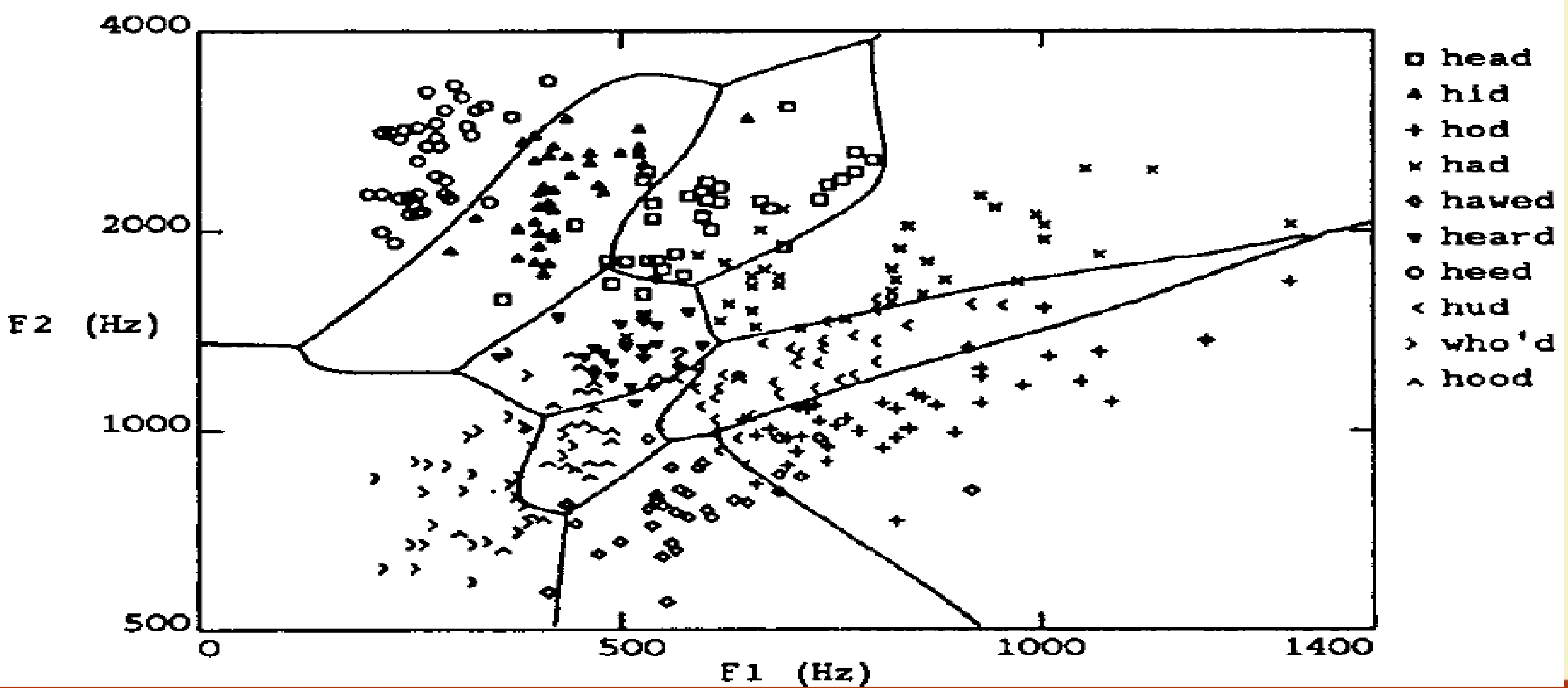
Perceptron vs. Delta rule training

- Delta/perceptron training rules appear same *but*
  - Perceptron rule trains discontinuous units
    - Guaranteed to converge under limited conditions
    - May not converge in general
  - Gradient rules trains over continuous response (unthresholded outputs)
    - Gradient rule always converges
      - Even with noisy training data
      - Even with non-separable training data
  - Gradient descent generalizes to other continuous responses
  - Can train perceptron with LMS rule
    - get prediction by thresholding outputs

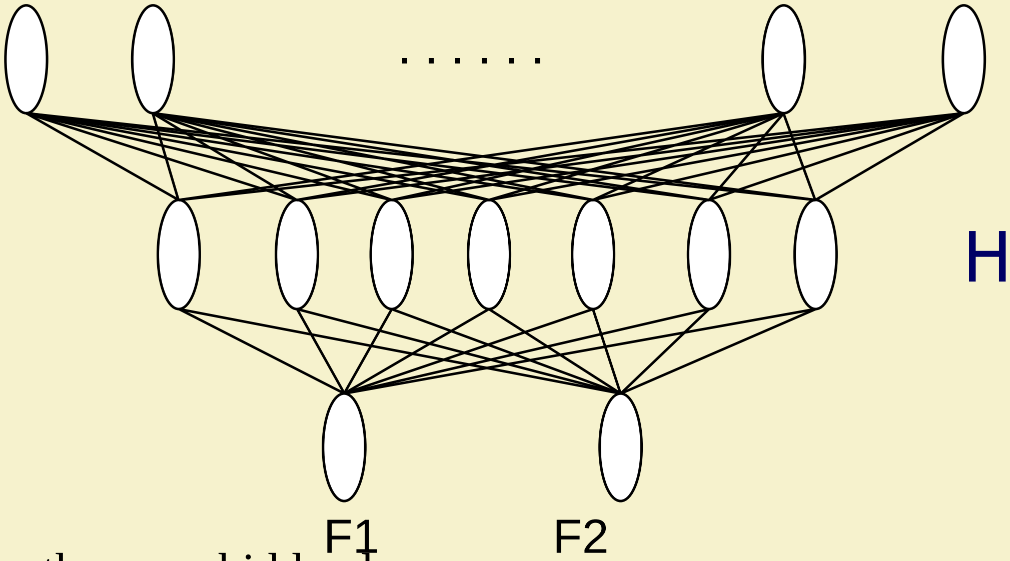
Multilayer networks of *sigmoid* units

- Needed for relatively complex (i.e., typical) functions
- Want non-linear response units in many systems
  - Example (next slide) of phoneme recognition
  - Cascaded nets of linear units only give linear response
  - Sigmoid unit as example of many possibilities
- Want differentiable functions of weights
  - So can apply gradient descent
    - Minimization of error function
  - Step function perceptrons non-differentiable

Speech recognition example



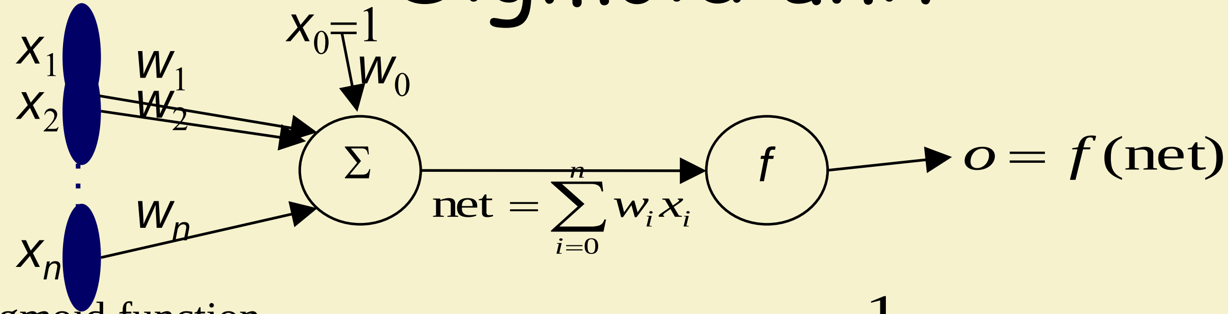
Multilayer networks



Hidden layer

- Can have more than one hidden layer

Sigmoid unit



- f is the sigmoid function

$$f(x) = \frac{1}{1+e^{-x}}$$

- Derivative can be easily computed:

$$\frac{df(x)}{dx} = f(x)\left(1 - f(x)\right)$$

- Logistic equation
  - used in many applications
  - other functions possible (tanh)
- Single unit:
  - apply gradient descent rule
- Multilayer networks: backpropagation

Error Gradient for a Sigmoid Unit

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

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

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

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

$$= -\sum_{d\in D}(t_d - o_d)\frac{\partial o_d}{\partial \mathrm{net}_d}\frac{\partial \mathrm{net}_d}{\partial w_i}$$

net: linear combination
o (output): logistic function

$$\frac{\partial o_d}{\partial \mathrm{net}_d} = \frac{\partial f(\mathrm{net}_d)}{\partial \mathrm{net}_d} = f(\mathrm{net}_d)\big(1 - f(\mathrm{net}_d)\big) = o_d(1 - o_d)$$

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

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

… Incremental Version

- Batch gradient descent for a single Sigmoid unit

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

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

- Stochastic approximation

$$E_d = \frac{1}{2}(t_d - o_d)^2$$

$$\boxed{\frac{\partial E_d}{\partial w_i} = -(t_d - o_d)o_d(1-o_d)x_{i,d}}$$

Backpropagation procedure

- Create FFnet
  - n_i inputs
  - n_o output units
    - Define error by considering *all* output units
  - n hidden units
- Train the net by propagating errors backwards from output units
  - First output units
  - Then hidden units
- Notation: x_ji is input from unit i to unit j

  w_ji is the corresponding weight
- Note: various termination conditions : Error, # iterations,…

Backpropagation (stochastic case)

- Initialize all weights to small random numbers
- Repeat

  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

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

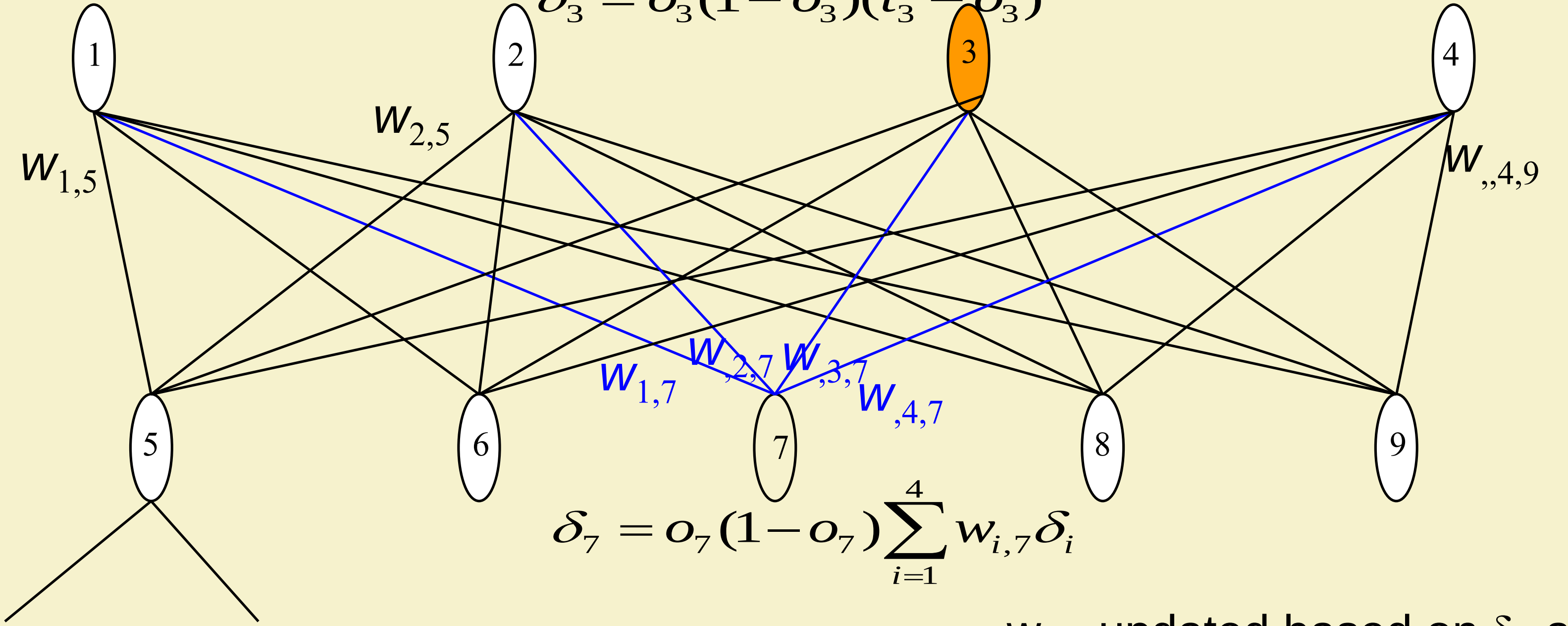
  4. Update each network weight $w_{j,i}$

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

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

Errors propagate backwards

$$\delta_3 = o_3(1 - o_3)(t_3 - o_3)$$



$$\delta_7 = o_7(1 - o_7)\sum_{i=1}^{4} w_{i,7}\delta_i$$

- Same process repeats if we have more layers

$w_{1,7}$ updated based on δ_1 and $x_{1,7}$

Properties of Backpropagation

- Easily generalized to arbitrary directed (acyclic) graphs
  - Backpropagate errors through the different layers
- Training is slow but applying network after training is fast

Convergence of Backpropagation

- Convergence
  - Training can take thousands of iterations → slow!
    - Gradient descent over entire network weight vector
    - Speed up using small initial values of weights:
      - Linear response initially
  - Generally will find local minimum
    - Typically can find good approximation to global minimum
  - Solutions to local minimum trap problem
    - Stochastic gradient descent
    - Can run multiple times
      - Over different initial weights
    - Committee of networks
    - Can modify to find better approximation to global minimum
      - include weight momentum α

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

        » Momentum avoids local max/min and plateaus

Example of face recognition

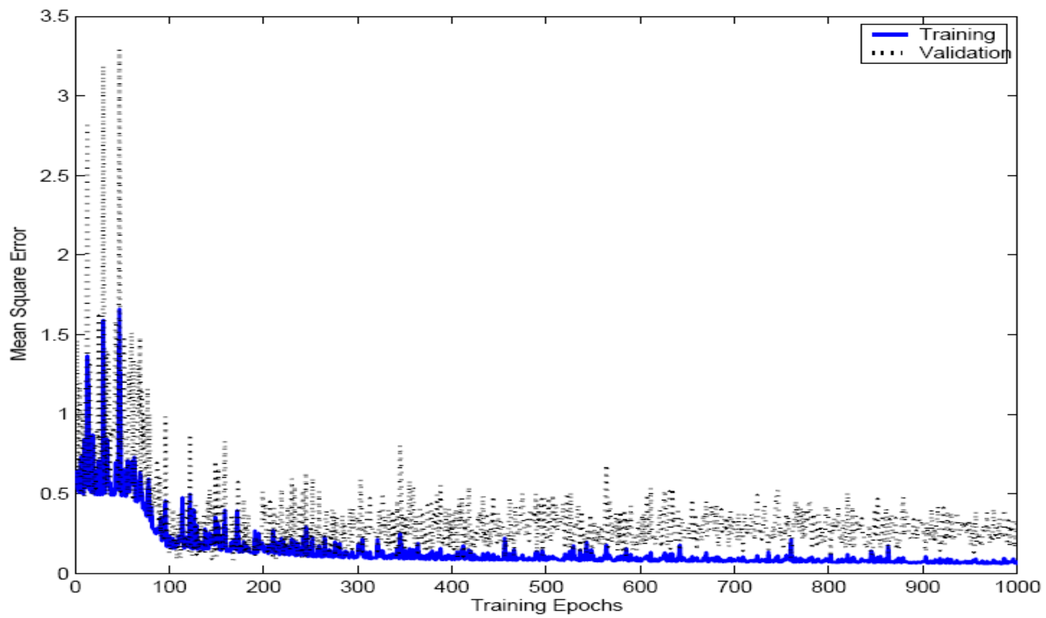
- Task: recognize faces from sample of
  - 20 people in 32 poses
  - Choose output of 4 values for direction of gaze
  - 120x128 images (256 gray levels)
- Can compute many functions
  - Identity/direction of face (used in book)/…
- Design issues
  - Input encoding (pixels/features/?)
    - Reduced image encoding (30x32)
  - Output encoding (1 or 4 values?)
    - Convergence to .1/.9 and not 0/1
  - Network structure (1 layer of 3 hidden units)
  - Algorithm parameters
    - Eta=.3; alpha=.3; stochastic descent method
- Training/validation sets
- Results: 90% accurate for head pose

Some issues with ANNs

- Interpretation of hidden units
    - Hidden units "discover" new patterns/regularities
    - Often difficult to interpret
- Overfitting
- Expressiveness
    - Generalization to different classes of functions

Dealing with overfitting

- Complex decision surface
- Divide sample into
  - Training set
  - Validation set
- Solutions
  - Return to weight set occurring near minimum over validation set
  - Prevent weights from becoming too large
    - Reduce weights by (small) proportionate amount at each iteration



Training
Validation
Mean Square Error
Training Epochs
0
0.5
1
1.5
2
2.5
3
3.5
0
100
200
300
400
500
600
700
800
900
1000

Expressiveness

- Every Boolean function can be represented by network with a single hidden layer
  - Create 1 hidden unit for each possible input
  - Create OR-gate at output unit
  - *but* might require exponential (in number of inputs) hidden units

Expressiveness

- Every bounded continuous function can be approximated with arbitrarily small error, by network with one hidden layer (Cybenko et al ‘89)
  - Hidden layer of sigmoid functions
  - Output layer of linear functions
- Any function can be approximated to arbitrary accuracy by a network with two hidden layers (Cybenko ‘88)
  - Sigmoid units in both hidden layers
  - Output layer of linear functions

Extension of ANNs

- Many possible variations
  - Alternative error functions
    - Penalize large weights
      - » Add weighted sum of squares of weights to error term
  - Structure of network
    - Start with small network, and grow
    - Start with large network and diminish
- Use other learning algorithms to learn weights

Extensions of ANNs

- Recurrent networks
  - Example of time series
    - Would like to have representation of behavior at t+1 from arbitrary past intervals (no set number)
    - Idea of simple recurrent network
      - hidden units that have feedback to inputs
- Dynamically growing and shrinking networks

Summary

- Practical method for learning continuous functions over continuous and discrete attributes
- Robust to noise
- Slow to train but fast afterwards
- Gradient descent search over space of weights
- Overfitting can be a problem
- Hidden layers can invent new features

End of Artificial Neural Networks