Artificial Neural Network (ANN)

Vani Rajasekar/CSE

Introduction

- Neural network learning methods provide a robust approach to approximating real-valued, discrete-valued, and vector-valued target functions.
- ANN learning is robust to errors in training data
- Applied to problems like interpreting visual scenes, speech recognition and robot control strategies
- In rough analogy, artificial neural networks are built out of a densely interconnected set of simple units, where each unit takes a number of real-valued inputs (possibly the outputs of other units) and produces a single real-valued output (which may become the input to many other units).

Introduction

The human brain, for example, contains

- densely interconnected network of approximately 10^{11} neurons
- each connected, on average, to 10^{4} others
- fastest neuron switching times 10^{-3} seconds quite slow compared to computer switching speeds of 10^{-10} seconds
- requires approximately 10^{-1} seconds to visually recognize

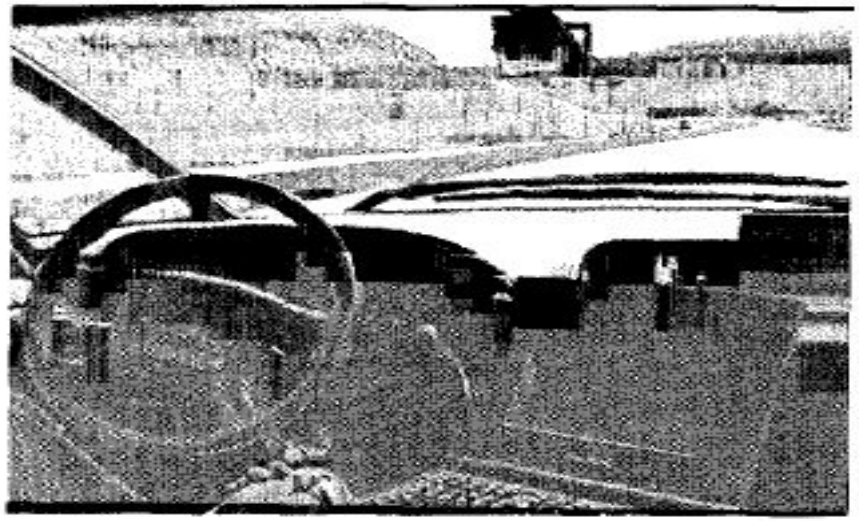
Properties of ANN

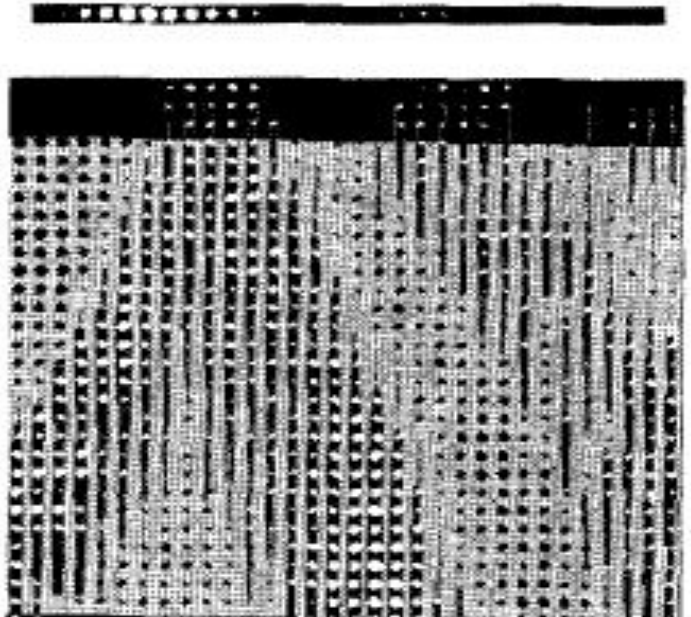
- Many neuron-like threshold switching units
- Many weighted interconnections among units
- Highly parallel computation based on distributed representation.

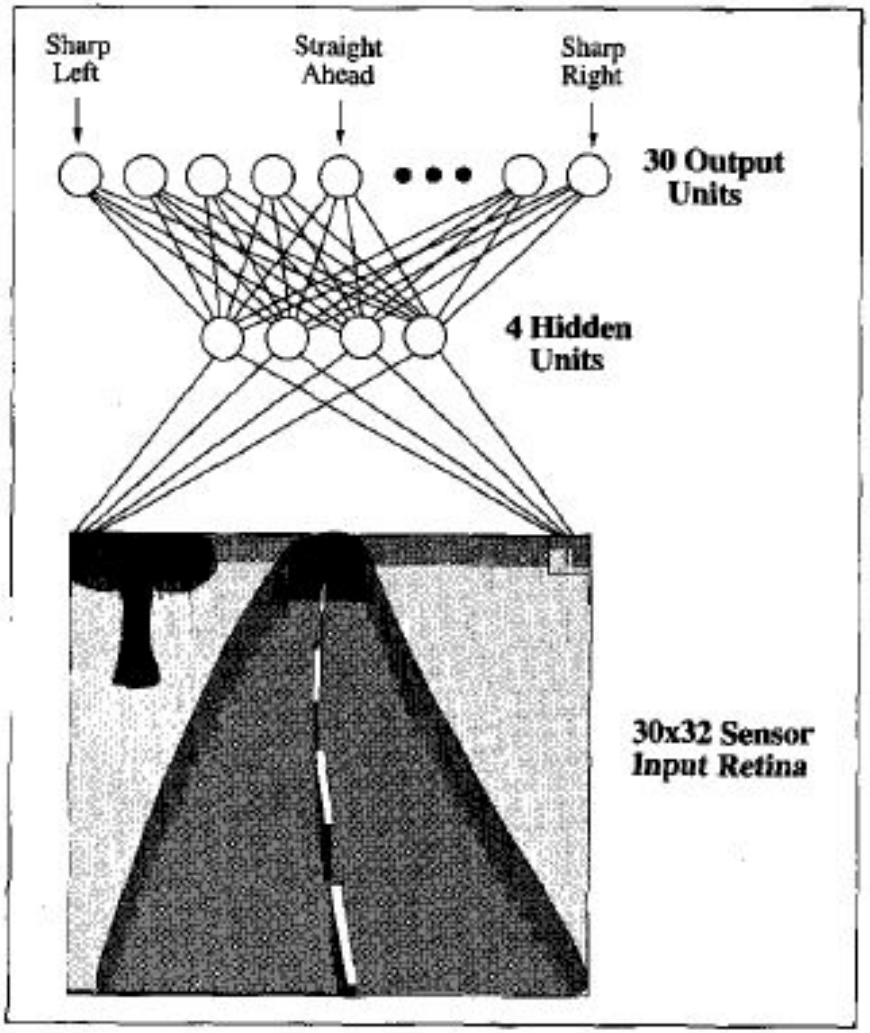
Representation of Neural Network

- A prototypical example of ANN learning is provided in 1993 called ALVINN
- It uses a learned ANN to steer an autonomous vehicle driving at normal speeds on public highways
- The input to the neural network is a 30 x 32 grid of pixel intensities obtained from a forward-pointed camera mounted on the vehicle.
- The network output is the direction in which the vehicle is steered
- The ANN is trained to mimic the observed steering commands of a human driving the vehicle for approximately 5 minutes.
- ALVINN successfully drive at speeds up to 70 miles per hour and for distances of 90 miles on public highways

ALVINN autonomous driving







ALVINN autonomous driving

- Neural network learning to steer an autonomous vehicle. The ALVINN system uses **BACKPROPAGATION** to learn to steer an autonomous vehicle driving at speeds up to **70 miles per hour.**
- The diagram on the right 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 left bottom shows weight values for one of the hidden units in this network.
- The **30 x 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.

Back Propagation

The BACKPROPAGATION algorithm is commonly used ANN learning technique. It is appropriate for problems with following characteristics

- ***Instances are represented by many attribute-value pairs.***

  Such as the pixel values in the ALVINN example

- ***The target function output may be discrete-valued, real-valued, or a vector of several real- or discrete-valued attributes***

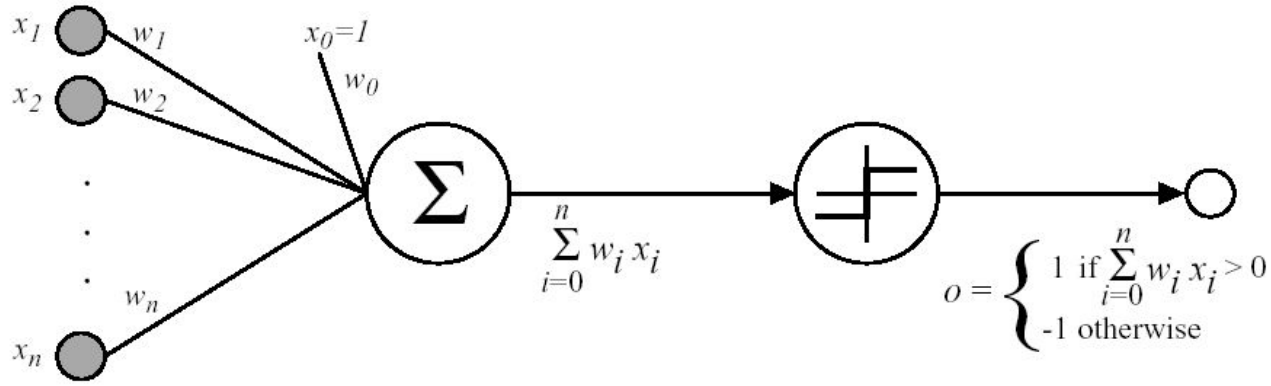
  -In ALVINN system the output is a vector of 30 attributes, each corresponding to a recommendation regarding the steering direction.

- ***The training examples may contain errors.***
- ***Long training times are acceptable***
- ***Fast evaluation of the learned target function may be required***

  ALVINN applies its neural network several times per second to continually update its steering command as the vehicle drives

Perceptrons

- One type of ANN system is based on a unit called a perceptron
- It takes a vector of real-valued inputs, calculates a linear combination of these inputs, then outputs a 1 if the result is greater than some threshold and -1 otherwise.



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

- where each wi is a real-valued constant or weight
- sometimes write the perceptron function $o(\vec{x}) = sgn(\vec{w} \cdot \vec{x})$

Perceptrons

where

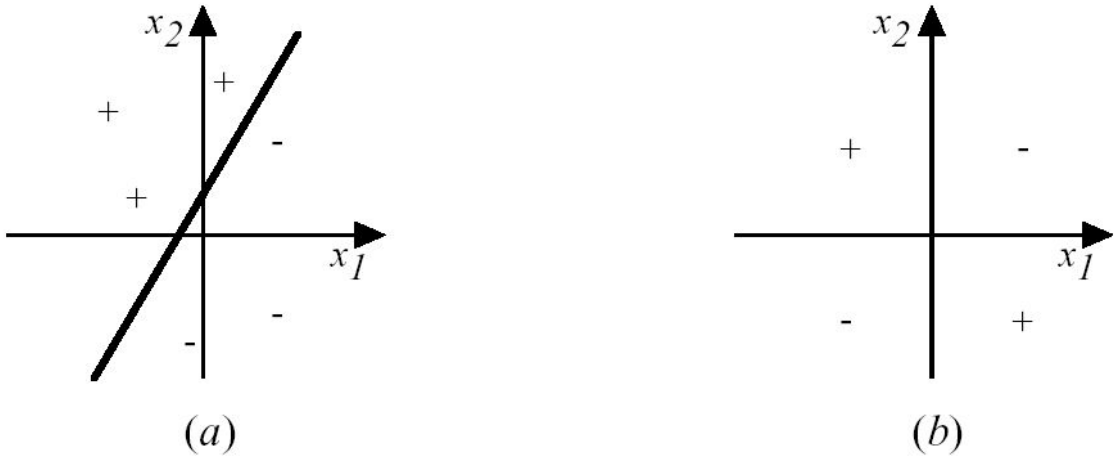
$$sgn(y) = \begin{cases} 1 \text{ if } y > 0 \\ -1 \text{ otherwise} \end{cases}$$

Representation of perceptrons:

- Perceptron can be viewed as representing a hyperplane decision surface in the n-dimensional space of instances
- The perceptron outputs a 1 for instances lying on one side of the hyperplane and outputs a -1 for instances lying on the other side
- Some sets of positive and negative examples cannot be separated by any hyperplane, they are called non linearly separable sets of examples.
- Boolean functions are represented by perceptrons include AND, OR, NAND and NOR
- some boolean functions cannot be represented by a single perceptron, such as the XOR

Perceptrons

- The decision surface represented by a two-input perceptron



- A) A set of training examples and the decision surface of a perceptron that classifies them correctly
- B) A set of training examples that is not linearly separable (i.e., that cannot be correctly classified by any straight line).

- A single perceptron can be used to represent many boolean functions.
- Eg: 1(true), -1(false)

Represents some useful functions

- Two input AND gate if w_0= -0.8, w_1= +0.5, w_2=+0.5

x1	x2	Output
-1	-1	-1
-1	+1	-1
+1	-1	-1
+1	+1	+1

- For (-1,-1) with x_0=1
  $=w_0x_0+w_1x_1+w_2x_2$
  $= -0.8-0.5-0.5= -0.18 < 0 \rightarrow -1$
- Similarly calculate for (-1,+1), (+1,-1), (+1,+1)
- OR gate w_0=0.1, w_1=0.1,w_2=0.1
- NOT gate w_0=0.5, w_1=-1

Perceptron Training Rule

- A precise learning problem is needed to determine a weight vector that causes the perceptron to produce the correct output +1 or -1
- Algorithm to solve the above are perceptron rule and delta rule

First Approach (Perceptron rule)

- Begin with random weights, then iteratively apply the perceptron to each training example
- Modifying the perceptron weights whenever it misclassifies an example.
- Repeat the above steps until the perceptron classifies all the training examples correctly

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

$$\text{where } \Delta w_i = \eta\,(t - o)\,x_i$$

Where:

- $t = c(x)$ is target value and o is perceptron output

Perceptron Training Rule

Disadvantage

- The above training examples are proven to converge separately when all the training examples are linearly separable and provided small η is used
- If the data are not linearly separable, convergence is not assured.

Gradient Descent and the Delta Rule

- To overcome the disadvantage of perceptron rule, delta rule is designed
- If the training examples are not linearly separable, the delta rule converges toward a best-fit approximation to the target concept.
- Delta rule is to use ***gradient descent to search the hypothesis*** space of possible weight vectors to find the weights that best fit the training examples
- This rule is important because gradient descent provides the basis for the BACKPROPAGATION algorithm.
- In delta rule, the task of training is considered as an ***un thresholded perceptron; that is, a linear unit for which the output o is given by***

$$o(\vec{x}) = sgn(\vec{w} \cdot \vec{x})$$

Gradient Descent and the Delta Rule

- The training error of a hypothesis relative to the training examples can be measured as

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

Where

-D is the set of training examples

-t_d is the target output for training example d

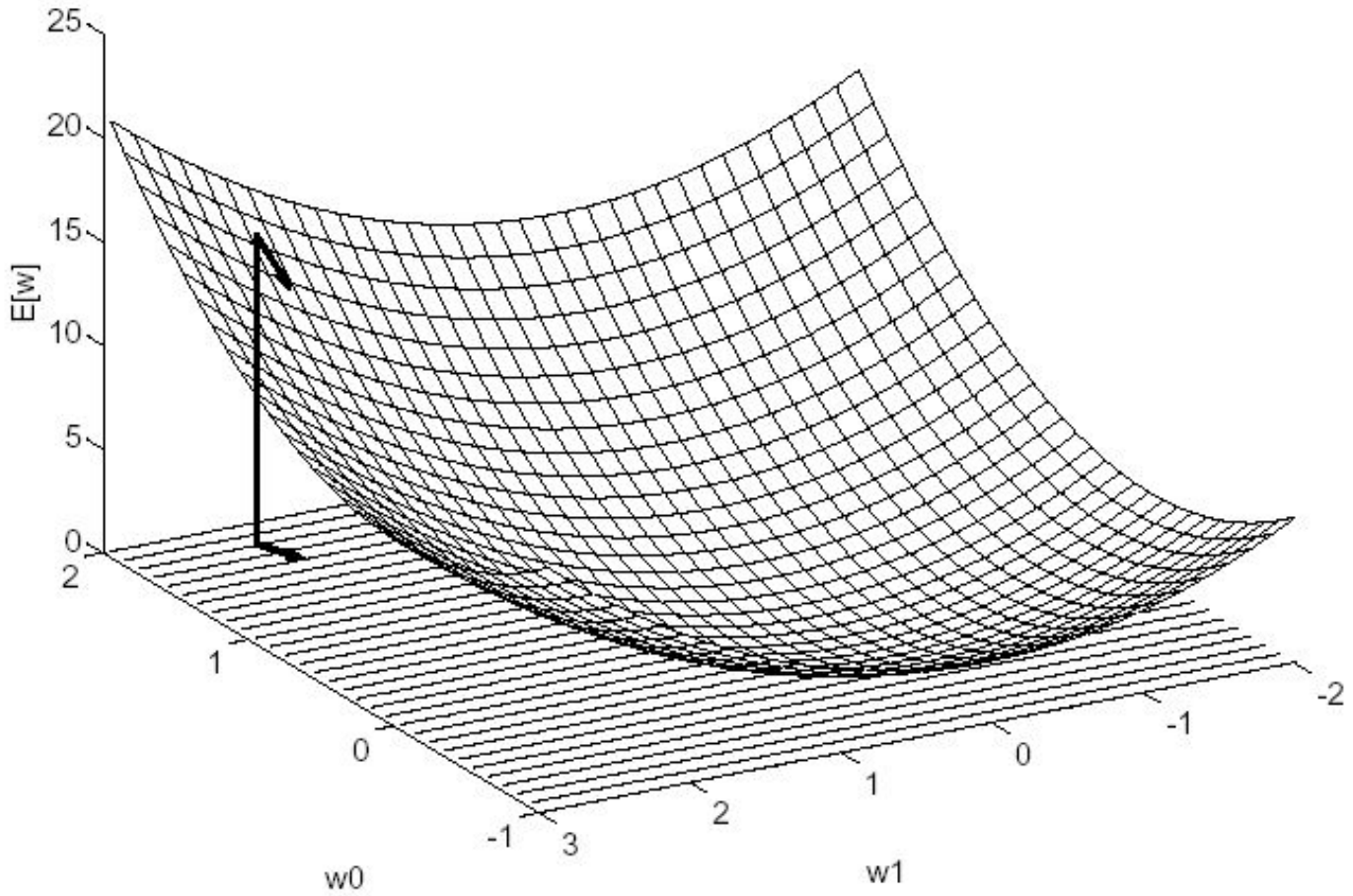
-o_d is the output of the linear unit for training example d

- This error is the half the squared difference between the target output t_d and the linear unit output o_d summed over all training examples.

Visualizing the Hypothesis Space

- To understand the gradient descent algorithm, it is helpful to visualize the entire hypothesis space of possible weight vectors and their associated E values.
- Here the axes w_0 and w_1 represent possible values for the two weights of a simple linear unit.
- The w_0, w_1 plane represents the entire hypothesis space.
- The vertical axis indicates the error E relative to some fixed set of training examples.

Visualizing the Hypothesis Space



Gradient Descent

How can we calculate the direction of steepest descent along the error surface?

- Gradient descent search determines a weight vector that minimizes E by starting with an arbitrary initial weight vector, then repeatedly modifying it in small steps.
- At each step, the weight vector is altered in the direction that produces the steepest descent along the error surface.
- This direction can be found by computing the derivative of ***E with respect to each*** component of the vector .
- This process continues until the global minimum error is reached.

Derivative of Gradient Descent

- This vector derivative is called the ***gradient of E*** *denoted as*

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

- When interpreted as a vector in weight space, the gradient specifies the direction that produces the steepest increase in E
- The negative of th $\vec{w} \leftarrow \vec{w} + \Delta\vec{w}$ ɔre gives the direction of steepest decrease

where

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

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

Derivative of Gradient Descent

- To find $\frac{\partial E}{\partial w_i}$

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

- Substitute above equation Δw_i

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

Gradient Descent Algorithm for training a linear unit

Gradient-Descent (training examples, η)

Each training example is a pair of the form $\langle x, t \rangle$ where x is the vector of input values, and t is the target output value. η is the learning rate.

- Initialize each w_i to some small random value
- Until the termination condition is met, Do
  - Initialize each Δw_i to zero.
  - For each $\langle x, t \rangle$ in *training_examples*, Do
    - Input the instance x to the unit and compute the output o
    - For each linear unit weight w_i, Do

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

  - For each linear unit weight w_i , Do

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

Stochastic Approximation to Gradient Descent

The key practical difficulties in applying gradient descent are

(1) converging to a local minimum can sometimes be quite slow (it can require many thousands of gradient descent steps)

(2) if there are multiple local minima in the error surface, then there is no guarantee that the procedure will find the global minimum.

- One variation on gradient descent is incremental gradient descent or stochastic gradient descent
- It updates the weights incrementally following the calculation of the error for each individual example
- The modified training rule is given by

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

Stochastic Approximation to Gradient Descent

- The distinct error function in stochastic approximation is given by

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

- The sequence of these weight updates, when iterated over all training examples, provides a reasonable approximation
- By making the value of learning rate sufficiently small, stochastic gradient descent can be made to approximate true gradient descent arbitrarily closely.

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. $\vec{w} \leftarrow \vec{w} - \eta \nabla E_d[\vec{w}]$

Differences between standard gradient descent and stochastic gradient descent

Standard Gradient Descent	Stochastic gradient descent
The error is summed over all examples before updating weights	Weights are updated upon examining each training example
Summing over multiple examples requires more computation	Less computation
Falls into local minima	Sometimes avoid falling into local minima

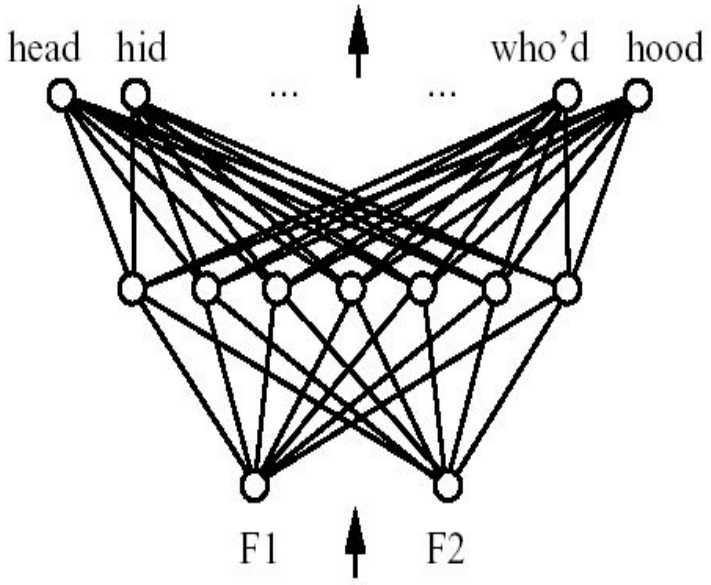
Multilayer Networks

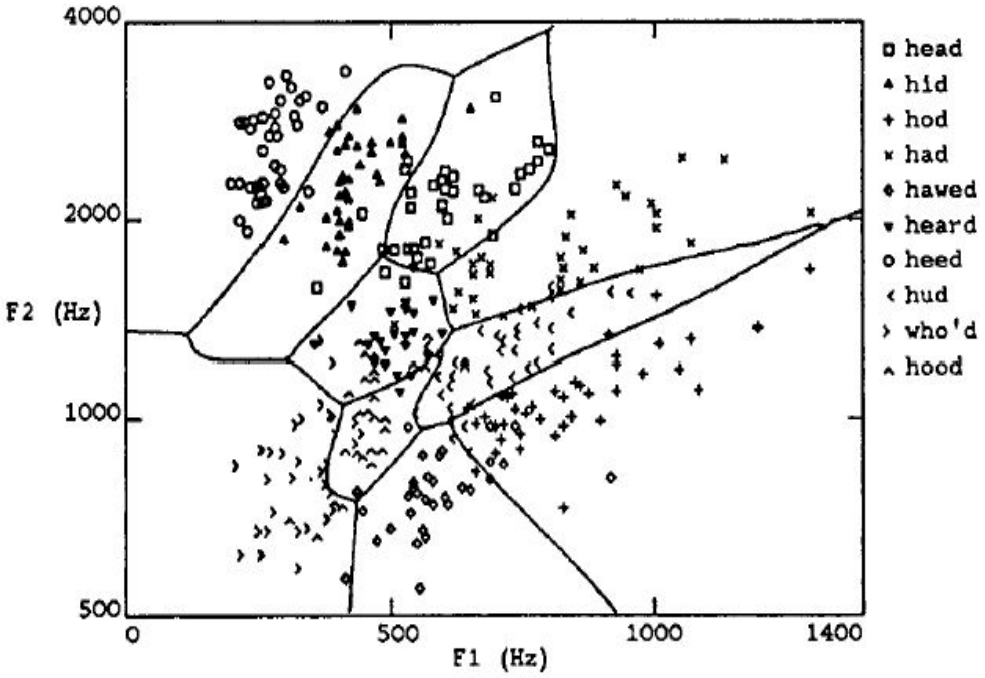
- Single perceptrons can only express linear decision surfaces
- Multilayer networks learned by the Backpropagation algorithm are capable of expressing non linear decision surfaces

Speech Recognition Task

- It involves distinguishing among 10 possible vowels, all spoken in the context of h-d (hid, had, head, hood etc)
- The input speech signal is represented by two numerical parameters (F1, F2)obtained from a spectral analysis of the sound.
- The 10 network outputs correspond to 10 possible vowel sounds
- The network prediction is the output whose value is

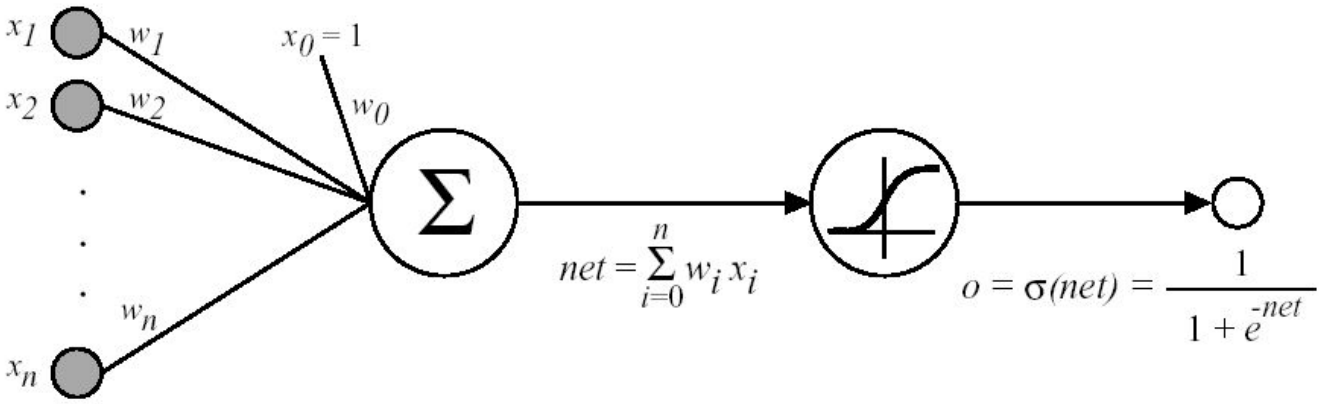
Decision regions of a multilayer feedforward network





Differentiable Threshold Unit

- Unit used as the basis for constructing multilayer networks – sigmoid unit which is very similar to perceptron but based on smoothed, differentiable threshold function.
- Like perceptron, the sigmoid unit first computes a linear combination of its inputs, then applies a threshold to the result



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

- $\sigma(x)$ is the sigmoid function

Sigmoid Unit

- The sigmoid function has the Nice property that its derivative is easily expressed in terms of its output.

We can derive gradient decent rules to train

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

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

Back propagation Algorithm

- The back propagation algorithm learns the weights for a multilayer network, given a network with a fixed set of units and interconnections.
- It employs gradient descent to attempt to minimize the squared error between the network output values and target values for these outputs.
- We are considering networks with multiple output units rather than single unit, we begin by redefining E to sum the errors over all of the network output units

$$E(\vec{w}) \equiv \frac{1}{2} \sum_{d \in D} \sum_{k \in outputs} (t_{kd} - o_{kd})^2$$

Back propagation Algorithm for feed forward networks containing two layers of sigmoid units

Backpropagation(training examples, η, n_{in},n_{out},n_{hidden})

- Each training example is a pair of the form <x, t> where

  x is the vector of input values, and t is vector of target output values
- η is the learning rate(Ex: .05) and n_{in} is the number of network inputs, n_{hidden} *is* the number of units in the hidden layer, n_{out} *is* the number of output units.
- The input from unit i to unit j is denoted by x_{ij}, and the weight from unit i to unit j is denoted by w_{ij}
- Create feed-forward network with n_{in} inputs, n_{hidden} hidden units and n_{out} output units.
- Initialize all weights to small random numbers. (between -0.05 and 0.05)

- Until the termination condition is met, Do
  - For each training example, Do
    - Propagate the input forward through the network

1. Input the training example to the network and compute the output o_u of every unit u in the network.
    - Propagate the errors backward through the network
2. For each output unit k calculate its error δ_k

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

3. For each hidden unit h, *calculate its error term* δ_h

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

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

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

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

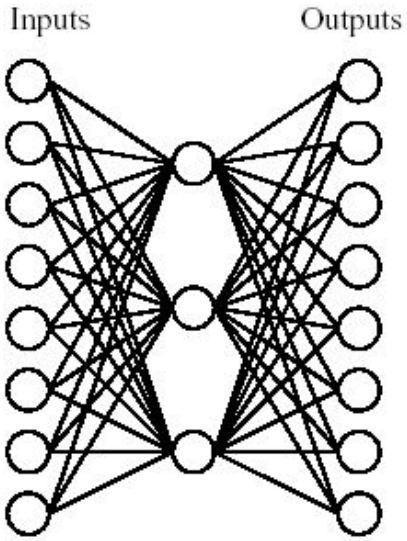
More on Backpropagation

- 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 to speedup convergence

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

- Minimizes error over *training* examples
  - Will it generalize well to subsequent examples?
- Training can take thousands of iterations → slow!
- Using network after training is very fast

Learning Hidden Layer Representations (1/2)



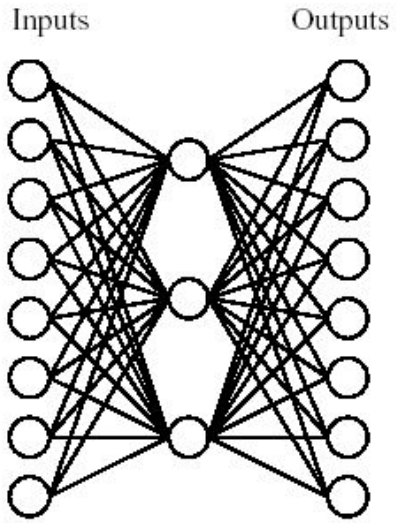
A target function:

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 (2/2)

A network:



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

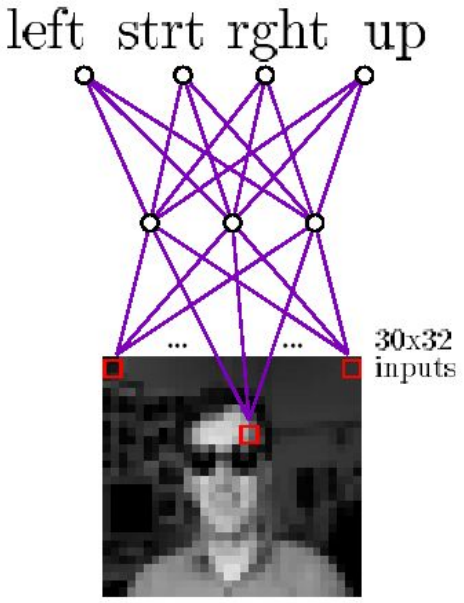
An illustrative Example: Face recognition

- To illustrate some of the practical design choices involved in applying backpropagation – face recognition task
- Learning task
  - Classifying camera images of faces of various people in various poses.
  - Images of 20 different people were collected, including approximately 32 images per person, varying the person's expression (happy, sad, angry, neutral), the direction in which they were looking.(left, right, straight ahead, up) and whether or not they were wearing sunglasses
  - Other variations
    - Background behind the person
    - The clothing worn by the person
    - Position of the person's face within the image

Target Functions

- A variety of target functions can be learned from this image data.
- Given an image as input, we could train an ANN to output the identity of the person, the direction in which the person is facing, the gender of the person, wearing sun glass or not etc.
- Consider the learning task as
  - Learning the direction in which the person is facing (to their left, right, straight, upward)

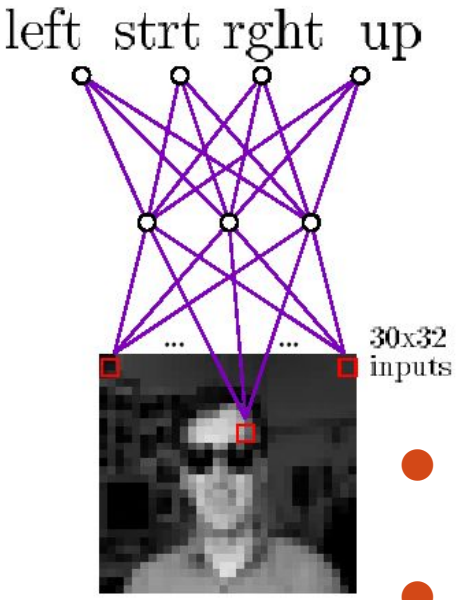
Neural Nets for Face Recognition

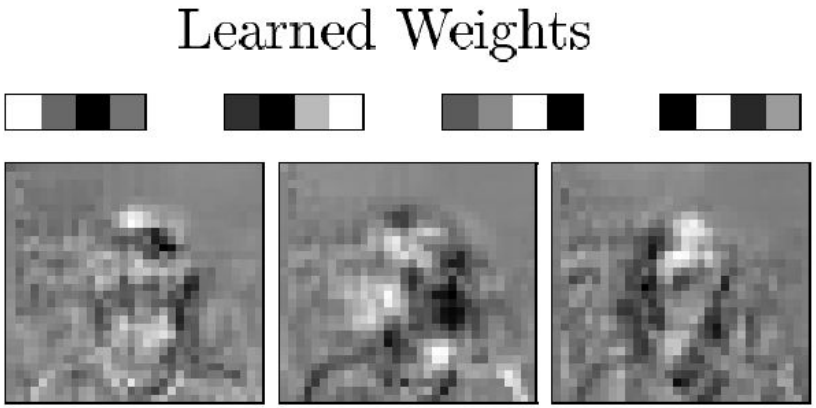


Typical input images

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

Learned Hidden Unit Weights





- Each output unit has four weights – dark(-ve), light (+ve) Blocks
- Leftmost block – weight w_0 which determines unit threshold
- Right 3 blocks – weights on inputs from three hidden units

Typical input images

Design Choices

- Input Encoding
- Output Encoding
- Network graph structure
- Other learning algorithm parameters

Input Encoding

- Preprocess the image to extract edges, regions of uniform intensity or other local image features, then input these features to the network.
- This leads to variable number of features (edges) per image, whereas the ANN has a fixed number of input units.
- The pixel intensity values ranging from 0 to 255 are linearly scaled to 0 to 1 – coarse resolution summary

Output Encoding

- ANN must output one of four values indicating the direction in which the person is looking
- We could encode this four-way classification using single output unit, assigning outputs of 0.2, 0.4, 0.6 and 0.8 to encode these four possible values.
- Instead use four distinct output units, each representing one of four possible face directions, with the highest valued output taken as the network prediction.
- This is called 1-of-n output encoding

Network Graph Structure

- Backpropagation can be applied to any acyclic directed graph of sigmoid units.
- Design choice here is, how many units to include in the network and how to interconnect them.
- Standard structure is two layers of sigmoid units (one hidden layer and one output layer)

Thank You