
NEURAL NETWORKS

1

Neural Networks

- Neural Networks (NNs) also known as
 - Artificial Neural Networks (ANNs),
 - Connectionist Models, and
 - Parallel Distributed Processing (PDP) Models

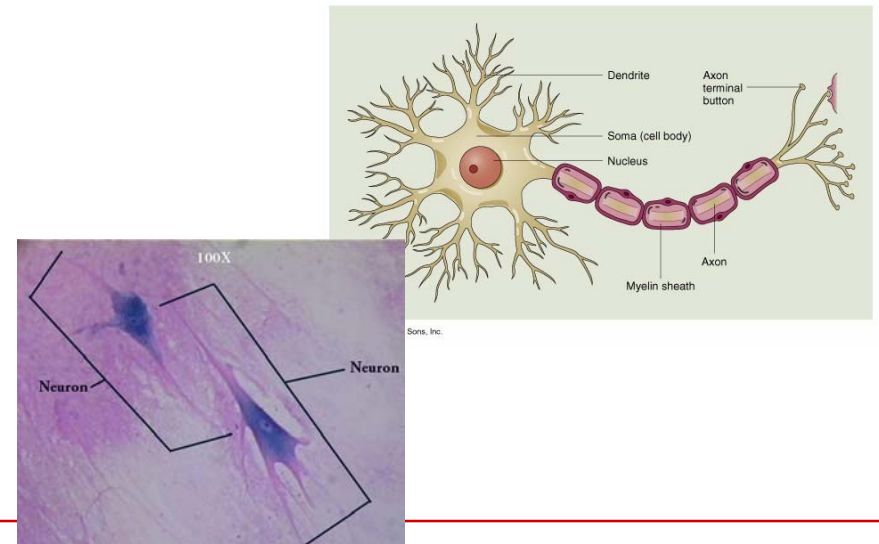
2

Biology

- The brain doesn't seem to have a CPU.
- Instead, it has many simple, parallel, asynchronous units, called *neurons*.
- Each neuron is a single cell has a number of relatively short fibers, called *dendrites*, and one long fiber, called an *axon*.
- The end of the axon branches out into more short fibers.
- Each fiber "connects" to the dendrites and cell bodies of other neurons.
- The "connection" is actually a short gap, called a *synapse*.

3

Biological Neuron



4

How Neurons Work

- The dendrites of surrounding neurons emit chemicals (neurotransmitters) that move across the synapse and change the electrical potential of the cell body.
- Sometimes the action across the synapse increases the potential, and sometimes it decreases it.
- If the potential reaches a certain threshold, an electrical pulse, or action potential, will travel down the axon, eventually reaching all the branches, causing them to release their neurotransmitters. And so on.

5

How Neurons Change

- There are changes to neurons that are presumed to reflect or enable learning:
 - The synaptic connections exhibit plasticity.
 - I.e., the degree to which a neuron will react to a stimulus across a particular synapse is subject to long-term change over time (long-term potentiation).
 - Neurons also will create new connections to other neurons.
 - Other changes in structure also seem to occur, some less well understood than others.

6

Neurobiology Constraints on Human Information Processing

- Number of neurons: 10^{12}
- Number of connections: 10^4 per neuron
- Neuron death rate: 10^5 per day
- Neuron birth rate: 0
- Connection birth rate: very slow
- Performance: about 10^2 msec, or about 100 sequential neuron firings for "many" tasks

7

How Do Neurons Do it?

- Basically, all the billions of neurons in the brain are active at once.
 - So, this is truly massive parallelism.
- But probably not parallelism of the sort we are used to in conventional Computer Science.
 - Sending messages (i.e., patterns that encode information) is probably too slow to work.
 - So information is probably encoded some other way. e.g., by the connections themselves.

8

AI/Cognitive Science Implication

- Explain cognition by richly connected networks transmitting simple signals.
- Sometimes called
 - connectionist computing (by Jerry Feldman).
 - Parallel Distributed Processing, or PDP (by Rumelhart, McClelland and Hinton).
 - neural networks (NN)
 - artificial neural networks (ANN)
 - emphasizing that the relation to biology is generally rather tenuous.

9

General Comments

- Parallelism in AI is not new.
 - spreading activation, etc.
- Neural models for AI is not new.
 - Indeed, is as old as AI, and some subdisciplines, e.g., computer vision, have continuously thought this way.
- Much neural network work makes biologically implausible assumptions about how neurons work.
 - “neurally inspired computing” rather “brain science”.
- Relation between NN and “symbolic AI”?
 - Some claim NN models don’t have symbols or representations.
 - Others think of NNs as simply being an “implementation-level” theory.
 - NNs started out as a branch of statistical pattern classification, and is headed back that way.
 - In any case, NNs gives us important insights into how to think about cognition.

10

Why Neural Networks?

- The autonomous local processing of each individual unit combines with similar simple behavior of many other units to produce “interesting,” complex, global behavior
- Intelligent behavior is an “emergent” phenomenon
- Solving problems using a processing model that is similar to the brain may lead to solutions to complex information processing problems that would be difficult to achieve using traditional symbolic approaches in AI
- Associative memory access is directly represented. Hence pattern-directed retrieval and matching operations are promoted.
- Robust computation because knowledge is distributed and continuous rather than discrete or digital. Knowledge captured in a large number of fine-grained units and can match noisy and incomplete data
- Fault tolerant architecture because computations can be organized so as not to depend on a fixed set of units and connections

11

Artificial Neural Networks

- Fine-grained, parallel, distributed computing model characterized by
 - A large number of very simple, neuron-like processing elements called units, PEs, or nodes
 - A large number of weighted, directed connections between pairs of units
 - Weights may be positive or negative real values
 - Local processing in that each unit computes a function based on the outputs of a limited number of other units in the network
 - Each unit computes a simple function of its input values, which are the weighted outputs from other units. If there are n inputs to a unit, then the unit’s output, or activation is defined by $a = g((w_1 * x_1) + (w_2 * x_2) + \dots + (w_n * x_n))$. Thus each unit computes a (simple) function g of the linear combination of its inputs.

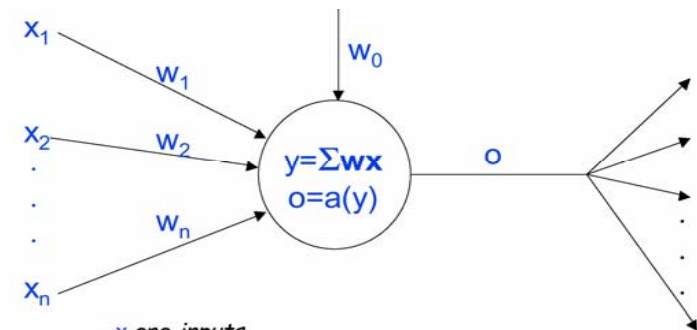
12

Models of a Neuron

- All connectionist/NN models have fairly similar models of a neuron.
- There is a collection of units (i.e., would-be neurons), each of which has
 - a number of weighted inputs from other units
 - inputs represent degree to which other unit is “firing”,
 - weights represent how much unit wants to listen to other unit.
 - a threshold that the weighted inputs are compared against
 - The threshold has to be crossed for the unit to do something (more or less).
 - a single output to another bunch of units
 - what the unit “decided” to do, given all the inputs and its threshold.

13

A Unit



x_i are inputs
 w_i are weights
 w_0 serves as the threshold, with x_0 fixed at 1.
 y is weighted sum of inputs (including threshold)
 o is output = some activation function of y

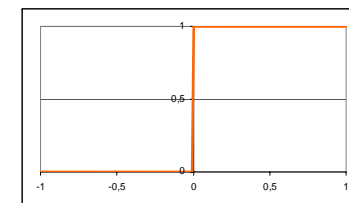
14

Properties of Units

- Inputs and outputs are numbers, sometimes constrained to be within range, e.g., $[0,1]$ or $[-1,1]$.
- The weights can be + or –, i.e., excitatory or inhibitory.
- A unit works by
 - summing together weighted inputs and threshold, i.e.,
 $y = -q + \sum x_j w_j$ (where x_j are the inputs to the unit, and q , the threshold).
 - y is input to an activation function, a ; output of activation function, o , is unit's output.
- For simplicity, instead of a threshold, we give each node an additional input, usually called x_0 , which is always set to 1.
 - Then negation of weight from unit to each node acts like threshold (or “bias”).

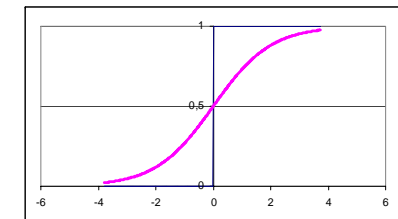
15

Some Common Definitions of What a Unit Computes



■ Hard Limiter

$$y = \begin{cases} 0 & x \leq 0 \\ 1 & x > 0 \end{cases}$$



Sigmoid

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

16

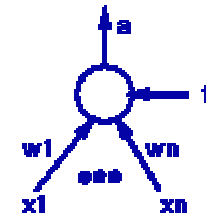
Note

- The units are continuously active.
 - (Actually, real neurons fire all the time; what changes is the rate of firing, from a few to a few hundred impulses a second.)
- The weights of the units are not fixed for all time.
 - Indeed, learning in a NN system is basically a matter of changing weights.

17

Perceptrons

- Simplest "interesting" class of neural networks
 - "1 layer" network -- i.e., one input layer and one output layer. In most basic form, output layer consists of just one unit.
 - Linear threshold unit (LTU) used at output layer node(s)



18

Thresholds

- The threshold associated with LTUs can be considered as another weight. That is, by definition of an LTU, output of a unit is defined by

$$\sum_{i=1}^n x_i w_i \geq t$$

- where t is a threshold value. But this is algebraically equivalent to

$$\sum_{i=1}^n x_i w_i + t(-1) \geq 0$$

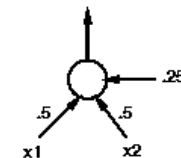
- So, in an implementation, consider each LTU as having an extra input which has a constant input value of -1 and the arc's weight is t . Now each unit has a fixed threshold value of 0, and t is an extra weight called the *bias*.
- So, from here on learning the weights in a neural network will mean learning the weights and the threshold values.

19

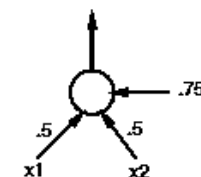
Examples

- OR function

Two inputs, both binary (0 or 1), and output binary (since output unit is an LTU):



- AND function



20

Learning in Neural Networks

- Programmer specifies numbers of units in each layer and connectivity between units, so the only unknown is the set of weights associated with the connections
- Supervised learning of weights from a set of training examples, given at I/O pairs. I.e., an example is a list of values defining the values of the input units in a given network. The output of the network is a list of values output by the output units

21

Algorithm:

- Initialize the weights in the network (usually with random values)
- repeat until stopping criterion is met
 - foreach example e in training set do
 - $O = \text{neural-net-output}(\text{network}, e)$
 - $T = \text{desired (i.e., teacher) output}$
 - $\text{update-weights}(e, O, T)$
- Note: Each pass through all of the training examples is called one **epoch**

22

Perceptron Learning Rule

- In a perceptron, we define the update-weights function in the learning algorithm above by the formula:
$$w_i = w_i + \Delta w_i$$
- where
$$\Delta w_i = \eta (T - O) x_i$$
- x_i is the input associated with the i^{th} input unit. η is a constant between 0 and 1 called the **learning rate**.

23

Notes about this update formula:

- Based on a basic idea due to Hebb that the strength of a connection between two units should be adjusted in proportion to the product of their simultaneous activations. A product is used as a means of measuring the correlation between the values output by the two units.
- Also called the Delta Rule or the Widrow-Hoff Rule
- "Local" learning rule in that only local information in the network is needed to update a weight
- Performs gradient descent in "weight space" in that if there are n weights in the network, this rule will be used to iteratively adjust all of the weights so that at each iteration (training example) the error is decreasing (more correctly, the error is monotonically non-increasing)

24

Notes about this update formula:

- Correct output ($T = O$) causes no change in a weight
- $x_i = 0$ causes no change in weight
- Does not depend on w_i
- If $T=1$ and $O=0$, then increase the weight so that hopefully next time the result will exceed the threshold at the output unit and cause the output O to be 1
- If $T=0$ and $O=1$, then decrease the weight so that hopefully next time the result will be below the threshold and cause the output to be 0.

25

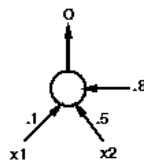
Perceptron Convergence Theorem

- If a set of examples are learnable (i.e., 100% correct classification), the Perceptron Learning Rule will find the necessary weights (Minsky and Papert, 1988)
 - In a finite number of steps
 - Independent of the initial weights
- The Perceptron Learning Rule does gradient descent search in weight space, so this theorem says that if a solution exists, gradient descent is guaranteed to find an optimal (i.e., 100% correct classification) solution for any 1-layer neural network

26

Example: Learning OR in a Perceptron

- Given initial network defined as:



- Let the learning rate parameter be $\eta = 0.2$
- Let the threshold be specified as a third weight, w_3 with constant input value $x_3 = -1$

27

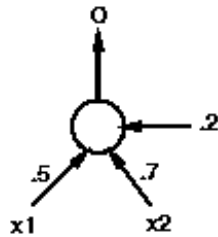
The result of executing the learning algorithm for 3 epochs

x_1	x_2	T	O	Δw_1	w_1	Δw_2	w_2	Δw_3	$w_3 (=t)$
-	-	-	-	-	.1	-	.5	-	.8
0	0	0	0	0	.1	0	.5	0	.8
0	1	1	0	0	.1	.2	.7	-.2	.6
1	0	1	0	.2	.3	0	.7	-.2	.4
1	1	1	1	0	.3	0	.7	0	.4
0	0	0	0	0	.3	0	.7	0	.4
0	1	1	1	0	.3	0	.7	0	.4
1	0	1	0	.2	.5	0	.7	-.2	.2
1	1	1	1	0	.5	0	.7	0	.2
0	0	0	0	0	.5	0	.7	0	.2
0	1	1	1	0	.5	0	.7	0	.2
1	0	1	1	0	.5	0	.7	0	.2
1	1	1	1	0	.5	0	.7	0	.2

28

Learned Network

- So, the final learned network is:



29

Linear Separability

- Perceptron output (assuming a single output unit) determined by the separating hyperplane defined by

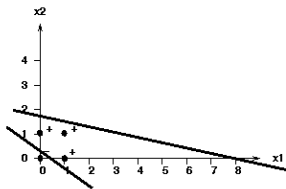
$$\sum_{i=1}^n x_i w_i = t$$

- So, Perceptrons can only learn functions that are linearly separable
- Example

- The weights in the initial network for the OR function given above defines a separating line $(0.1 * x_1) + (0.5 * x_2) - 0.8 = 0$. Rewriting, this is equivalent to the line defined by $x_2 = (-0.2 * x_1) + 1.6$, that is a line with slope -0.2 and x_2 -intercept 1.6.
- The weights in the final network for the OR function define a separating line $(0.5 * x_1) + (0.7 * x_2) - 0.2 = 0$, or $x_2 = (-0.7 * x_1) + 0.3$
- The initial and final separating lines can be shown graphically in terms of the two-dimensional space of possible inputs as follows. Notice that the initial separating line classifies all four examples as 0, but the final separating line correctly has all the examples with target value 0 on one side and all the examples with target value 1 on the other side.

30

Linear Separability

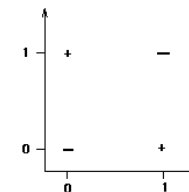


- In general, the goal of learning in a perceptron is to adjust the separating line by modifying the weights until all of the examples with target value 1 are on one side of the line, and all of the examples with target value 0 are on the other side of the separating line, where the "line" is in general a hyperplane in an n -dimensional space, and n is the number of input units.

31

XOR - A Function that Can Not be Learned by a Perceptron

- The Exclusive OR function can be shown graphically as follows, where + corresponds to an output of 1, and - corresponds to a desired output of 0.



- Clearly from the figure it is evident that there does not exist a line that can separate the two classes. Hence, XOR is not a linearly-separable function, and cannot be learned by a Perceptron.
- Other examples of functions that are not linearly-separable: parity and "determining if all of the 1s in a 2D binary array are mutually connected to one another by paths of 1s"

32

Beyond Perceptrons

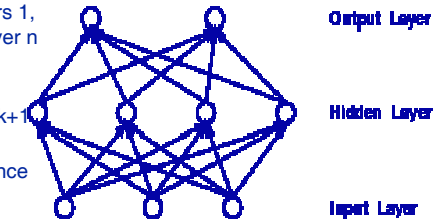
- Perceptrons are too weak a computing model because they can only learn linearly-separable functions
- We need to define more complex neural networks in order to enhance their functionality

33

Multi-layer, feedforward networks

- Multi-layer, feedforward networks generalize 1-layer networks (i.e., Perceptrons) to n -layer networks as follows:

- Partition units into $n+1$ "layers," such that layer 0 contains the input units, layers 1, ..., $n-1$ are the hidden layers, and layer n contains the output units
- Each unit in layer k , $k=0, \dots, n-1$, is connected to all of the units in layer $k+1$
- Connectivity means bottom-up connections only, with no cycles, hence the name "feedforward" nets
- Programmer defines the number of hidden layers and the number of units in each layer (input, hidden, and output)
- Example of a 2-layer feedforward network:



34

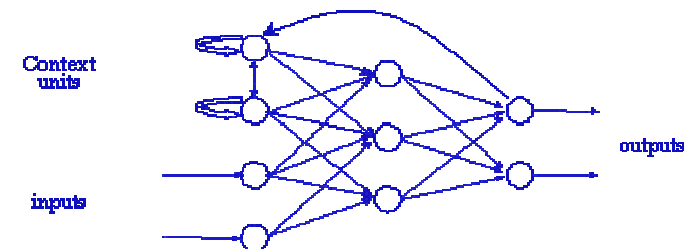
Multi-layer, feedforward networks

- 2-layer feedforward neural nets (i.e., nets with one hidden layer) with an LTU at each hidden and output layer unit, can compute functions associated with classification regions that are convex regions in the space of possible input values. That is, each unit in the hidden layer acts like a Perceptron, learning a separating line. Together, all of the hidden units' separating lines are combined by the output unit(s) to classify an example by intersecting all of the half-planes defined by the separating lines.
- 3-layer feedforward neural nets (i.e., nets with two hidden layers) with an LTU at each hidden and output layer unit, can compute arbitrary functions (hence they are universal computing devices) although the complexity of the function is limited by the number of units in the network.

35

Recurrent networks

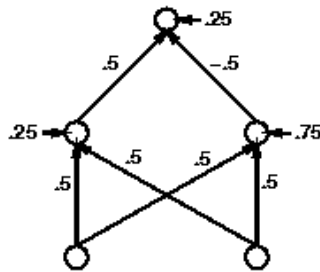
- are multi-layer networks in which cyclic (i.e., feedback) connections are allowed. In particular, if the output of a unit is connected to the other units in its own layer, then this special case of cyclic connections define networks with mutual inhibition links.



36

Computing XOR using a 2-Layer Feedforward Network

- The following network computes XOR. Notice that the left hidden unit effectively computes OR and the right hidden unit computes AND. Then the output unit outputs 1 if the OR output is 1 and the AND output is 0.



37

Backpropagation Learning in Feedforward Neural Nets

- Method for learning weights in feedforward networks due to Rumelhart, Hinton, and Williams, 1986, which generalizes the Delta Rule
- Cannot use the Perceptron Learning Rule for learning in Feedforward Nets because for hidden units we don't have teacher (i.e., desired) values
- Must solve the **Credit Assignment Problem** -- i.e., when there is an error at an output unit (i.e., a non-zero difference between the actual output of the unit and the teacher output), which weights in the network should be updated, and how to update them? In other words, how to assign credit/blame for the output error to the weights in the network?
- Backpropagation approach: gradient-descent algorithm to minimize the error on the training data by propagating errors backwards through the network starting at the output units and working backwards towards the input units

38

Backpropagation Algorithm

- Backpropagation algorithm performs gradient descent search in weight space for learning network weights.
- If there are m weights, then each configuration of weights that defines an instance of the network is a vector, W , of length m . W can be considered to be a point in an m -dimensional weight space, where each axis is associated with one of the connections in the network.
- Given a training set of n examples, each network defined by the vector W has an associated error, E , indicating the total error on all of the training data. Usually, E is measured using the mean squared error (MSE):

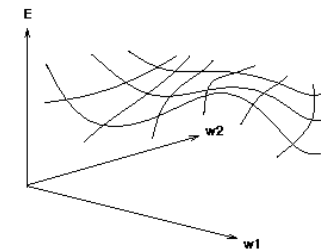
$$E = \frac{1}{n} \sum_{i=1}^n (T_i - O_i)^2$$

where T_i is the teacher value for the i th example, and O_i is the network output value for the i th example, and there are n examples in the training set.

39

Learning as search

- Consider the $m+1$ dimensional space where m dimensions are the weights, and the last dimension is the error, E . The error is non-negative and defines a surface in this weight space as shown below:



- So, the goal is to search for the point in weight space with (global) minimum mean squared error E

40

Learning as search

- To simplify the search, we'll use gradient descent to locally move in weight space, at each iteration, so as to change the weights so that the error decreases the fastest, and halt when we're at a (local) minimum in E.
- The gradient is a vector that indicates the steepest rate of increase in a function, so we'll change the weights in the opposite direction, which decreases E the fastest. That is, compute

$$\nabla E = \left(\frac{\partial E}{\partial w_1}, \frac{\partial E}{\partial w_2}, \dots, \frac{\partial E}{\partial w_m} \right)$$

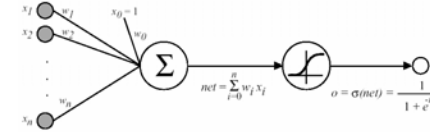
- and then change the i th weight by

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

- Computing derivatives required for calculating the gradient direction requires an activation function that is continuous, differentiable, non-decreasing and easily computed.
- Consequently, we can't use the LTU function. Instead, we'll use the sigmoid function at each unit (except the input units, of course).

41

Sigmoid Threshold Unit



- Interesting property: $\frac{d\sigma(x)}{dx} = \sigma(x)(1 - \sigma(x))$
- Output ranges between 0 and 1
- We can derive gradient descent rules to train
- One sigmoid unit
- Multilayer networks of sigmoid units → Backpropagation

42

Computing the Gradient of E

- We'll consider the problem of a 2-layer network where we must update the weights connecting nodes in the hidden layer to the output layer, and the weights connecting the nodes in the input layer to the hidden layer

43

Updating Hidden-to-Output Unit Weights

- Since we have the teacher-supplied values for the desired values output by the Output units, we can use a generalized version of the Perceptron Learning Rule to update these weights. Because a sigmoid unit is used here instead of an LTE unit, we obtain the update formula:

$$\Delta w_{ji} = \eta * a_j * (T_i - O_i) * g'(in_i) = \eta * a_j * (T_i - O_i) * O_i * (1 - O_i)$$
- where weight w_{ji} connects hidden unit j to output unit i , η is the learning rate parameter, T_i is the teacher output associated with output unit i , O_i is the actual output of output unit i , a_j is the output of hidden unit j , and g' is the derivative of the sigmoid activation function, which is known to be $g' = g(1-g)$.

44

Updating Input-to-Hidden Unit Weights

- Because we don't have teacher-supplied values for the correct outputs at the hidden units, we must infer the error at these units by "backpropagating the errors at the output units.
- Each hidden unit is connected to a given output unit, so an error at an output unit should be "distributed" to each of the hidden units; we do this in proportion to the weight of the connection from a hidden unit to the given output unit.
- In this way the total error is distributed to all of the hidden units that contributed to that error. In this way, each hidden unit will accumulate some error from each of the output units that it is connected to.
- The total error is the sum of the errors propagated back from the output units. Specifically, we get the following formula for updating the weight w_{kj} on the connection from input unit k to hidden unit j :

$$\Delta w_{kj} = \eta * I_k * a_j * (1 - a_j) * \sum [a_i(1 - a_i)(T_i - a_i)w_{ji}]$$

45

The Backpropagation 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

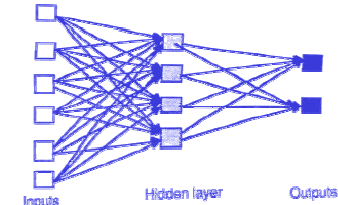
$$\delta_h \leftarrow o_h(1 - o_h) \sum_{k \in \text{outputs}} w_{hk} \delta_k$$

4. Update each network weight w_{ij}

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

where

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



46

Adding Momentum

- ◆ Another weight update is possible.

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

- n -th iteration update depend on $(n-1)$ th iteration
- α : constant between 0 and 1 -> momentum
- Role of momentum term :
 - keep the ball rolling through small local minima in the error surface.
 - Gradually increase the step size of the search in regions where the gradient is unchanging, thereby speeding convergence

47

Other Issues

- **How to Set η , the Learning Rate Parameter?**
 Use a tuning set or cross-validation to train using several candidate values for η , and then select the value that gives the lowest error
- **How to Estimate the Error?**
 - Use cross-validation (or some other evaluation method) multiple times with different random initial weights.
 - Report the average error rate.
- **How many Hidden Layers and How many Hidden Units per Layer?**
 Usually just one hidden layer is used (i.e., a 2-layer network). How many units should it contain?
 - Too few => can't learn.
 - Too many => poor generalization.
 - Determine experimentally using a tuning set or cross-validation to select number that minimizes error.
 - Genetic algorithms can be used to determine the structure.

48

■ How many examples in the Training Set?

Under what circumstances can we be assured that a net that is trained to classify $1 - \epsilon/2$ of the training set correctly, will also classify $1 - \epsilon$ of the testing set correctly?

- Clearly, the larger the training set the better the generalization, but the longer the training time required. But to obtain $1 - \epsilon$ correct classification on the testing set, training set should be of size approximately n/ϵ , where n is the number of weights in the network and ϵ is a fraction between 0 and 1.
- For example, if $\epsilon=.1$ and $n=80$, then a training set of size 800 that is trained until 95% correct classification is achieved on the training set, should produce 90% correct classification on the testing set.

■ When to Stop?

Too much training "overfits" the data, and hence the error rate will go up on the testing set. Hence it is not usually advantageous to continue training until the MSE is minimized. Instead, train the network until the error rate on a tuning set starts to increase.

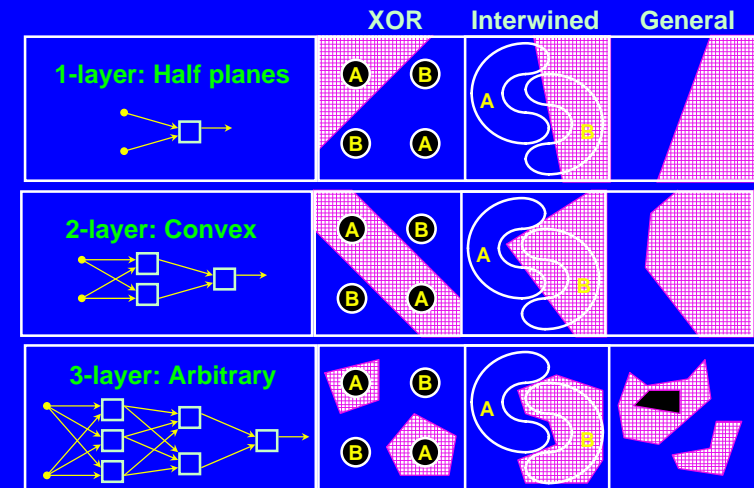
49

Advantages

- Parallel processing
- Distributed representations
- Online (i.e., incremental) algorithm
- Simple computations
- Robust with respect to noisy data
- Robust with respect to node failure
- Empirically shown to work well for many problem domains

51

MLP Decision Boundaries



50

Disadvantages

- Slow training
- Poor interpretability
- Network topology layouts ad hoc
- Hard to debug because distributed representations preclude content checking
- May converge to a local, not global, minimum of error
- Not known how to model higher-level cognitive mechanisms
- May be hard to describe a problem in terms of features with numerical values

52

Alternative Error Functions

- Penalize large weight

$$E(w) \equiv \frac{1}{2} \sum_{d \in D} \sum_i (t_i^{(d)} - o_i^{(d)})^2 + \gamma \sum_{i,j} w_{ij}^2$$

- Train on target slopes

$$E(w) \equiv \frac{1}{2} \sum_{d \in D} \sum_i \left[(t_i^{(d)} - o_i^{(d)})^2 + \mu \sum_k \left(\frac{\partial t_i^{(d)}}{\partial x_k^{(d)}} - \frac{\partial o_i^{(d)}}{\partial x_k^{(d)}} \right)^2 \right]$$

- Minimizing cross entropy

$$E(w) \equiv - \sum_{d \in D} \sum_i t_i^{(d)} \log o_i^{(d)} + (1 - t_i^{(d)}) \log(1 - o_i^{(d)})$$

53

Derivative-Based Optimization

- Based on first derivatives:

- Steepest descent (back-propagation)
- Conjugate gradient method
- And many others

- Based on second derivatives:

- Gauss-Newton method
- Levenberg-Marquardt method
- And many others

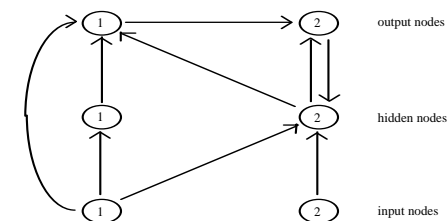
54

Derivative-Free Optimization

- Evolutionary algorithms (EAs)
- Simulated annealing (SA)
- Random search
- Downhill simplex search
- Tabu search

55

NN Topology Optimization using GA



output 1	output 2	hidden node 1	hidden node 2
101100	000110	100000	110001

Legend for the binary strings above:

- output 2
- output 1
- hidden node 2
- hidden node 1
- input 2
- input 1

56

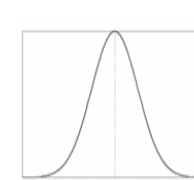
NN of Radial Basis Functions

- Motivations: better performance than sigmoid functions
 - Some classification problems
 - Function interpolation
- Definition
 - A function is radial symmetric (or is RBF) if its output depends on the distance between the input vector and a stored vector related to that function
 - Distance $u = \|\mu - i\|$ where i is the input vector, μ is the vector associated with the RBF
 - Output $\rho(u_1) \geq \rho(u_2)$ whenever $u_1 < u_2$
 - NN with RBF node function are called RBF-nets

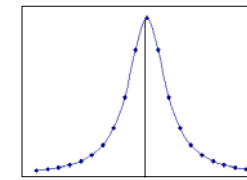
57

NN of Radial Basis Functions

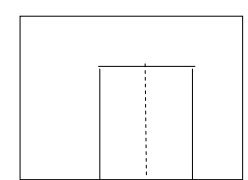
- Gaussian function is the most widely used RBF
 - $\rho_g(u) \propto e^{-(u/c)^2}$ a bell-shaped function centered at $u = 0$.
 - Continuous and differentiable
 - if $\rho_g(u) = e^{-(u/c)^2}$ then $\rho'_g(u) = e^{-(u/c)^2} (-(u/c)^2)' = -2\frac{u}{c} \rho_g(u)$
 - Other RBF
 - Inverse quadratic function, hyperspheric function, etc



μ
Gaussian function



μ
Inverse quadratic function
 $\rho_2(u) = (c^2 + u^2)^\beta$, for $\beta < 0$

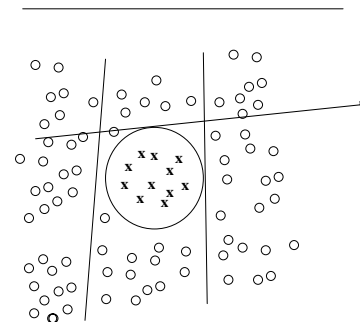


μ
hyperspheric function
 $\rho_s(u) = \begin{cases} 1 & \text{if } u \leq c \\ 0 & \text{if } u > c \end{cases}$

58

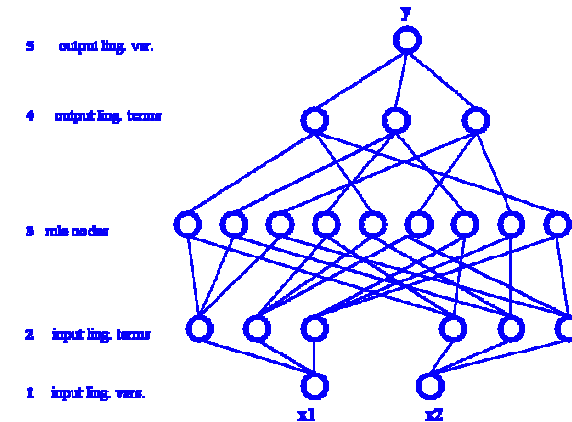
NN of Radial Basis Functions

- Pattern classification
 - 4 or 5 sigmoid hidden nodes are required for a good classification
 - Only 1 RBF node is required if the function can approximate the circle



59

Neuro Fuzzy Architecture: FDMS II



60

ART

■ The simplest ART network

- is a vector classifier
- classifies the input vector into a category depending on the stored pattern it most closely resembles
- the pattern is modified to resemble the input vector
- if no match, a new pattern is created based on input vector
- (Solution to the Stability-Plasticity dilemma)

61

ART

■ ART has both:

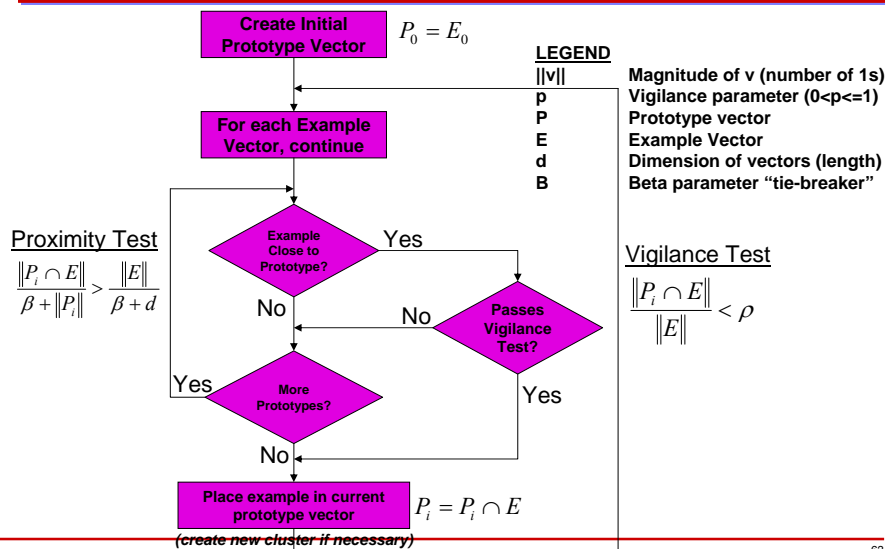
- Plasticity, new categories can be formed if there is no match on the current patterns
- Stability, environment cannot change stored patterns unless sufficiently similar

■ Variations of ART:

- ART1, unsupervised learning on binary patterns
- ART2, both analog and digital input patterns
- ART3, parallel search of distributed recognition in a multilevel network hierarchy
- ARTMAP, supervised learning
- Fuzzy ARTMAP, synthesis of neural networks, expert systems, and fuzzy logic

62

ART1



63

Self Organizing Maps

■ Based on competitive learning(Unsupervised)

- Only one output neuron activated at any one time
- Winner-takes-all neuron or winning neuron

■ In a Self-Organizing Map

- Neurons placed at the nodes of a lattice
 - one or two dimensional
- Neurons selectively tuned to input patterns
 - by a competitive learning process
- Locations of neurons so tuned to be ordered
 - formation of topographic map of input patterns
- Spatial locations of the neurons in the lattice -> intrinsic statistical features contained in the input patterns

64

Self Organizing Maps

■ Topology-preserving transformation

