ARTIFICIAL NEURAL NETWORKS

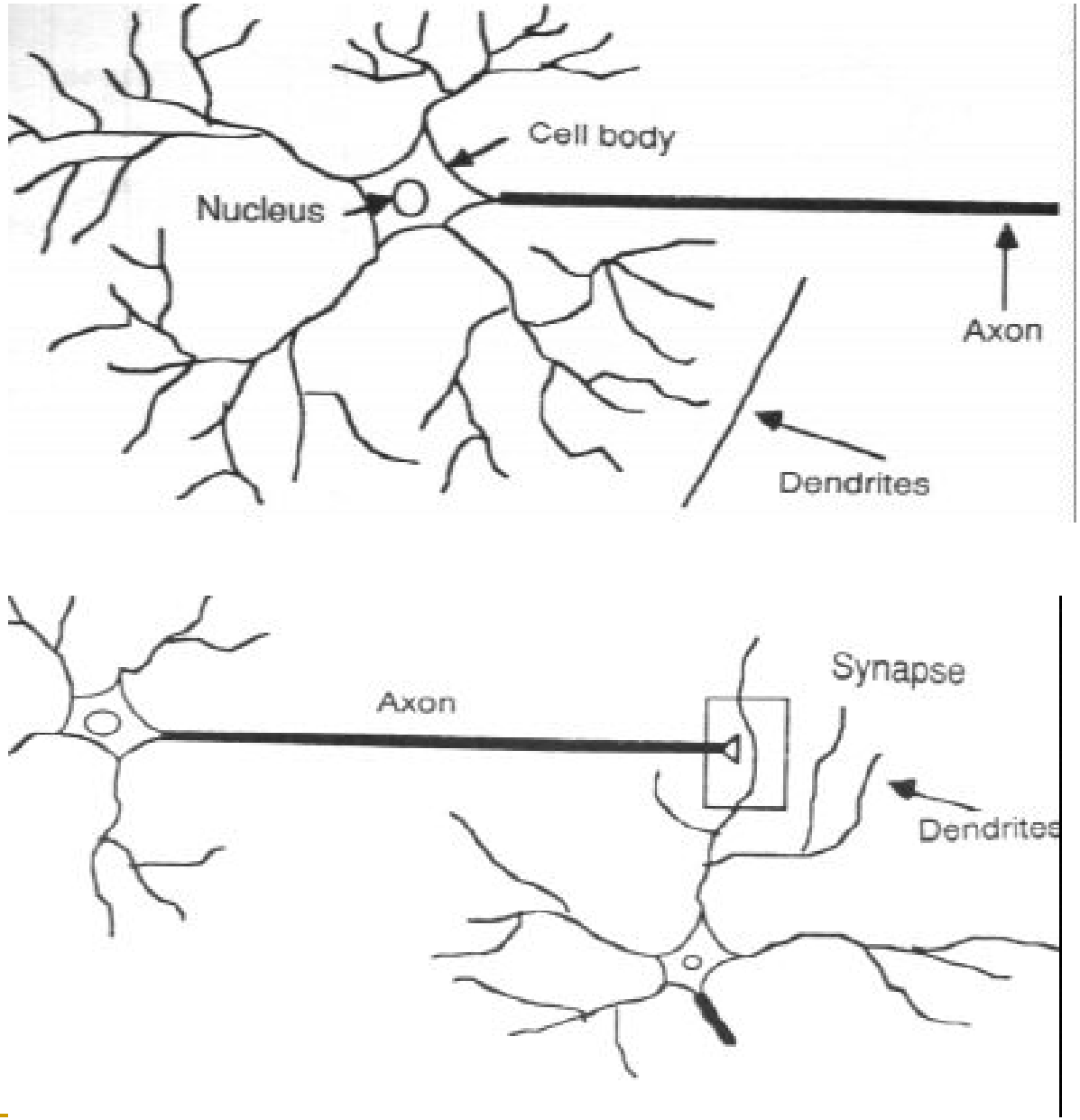
Outline

- 1. Introduction
- 2. ANN representations
- 3. Perceptron Training
- 4. Multilayer networks and Backpropagation algorithm
- 5. Remarks on the Backpropagation algorithm
- 6. Neural network application development
- 7. Benefits and limitations of ANN
- 8. ANN Applications

INTRODUCTION
Biological Motivation

- Human brain is a densely interconnected network of approximately 10^{11} neurons, each connected to, on average, 10^{4} others.
- Neuron activity is *excited* or *inhibited* through connections to other neurons.
- The fastest neuron switching times are known to be on the order of 10^{-3} sec.

- The cell itself includes a *nucleus* (at the center).
- To the right of cell 2, the *dendrites* provide input signals to the cell.
- To the right of cell 1, the *axon* sends output signals to cell 2 via the axon terminals. These axon terminals merge with the dendrites of cell 2.



Portion of a network: two interconnected cells.

- Signals can be transmitted unchanged or they can be altered by *synapses*. A synapse is able to **increase or decrease the strength** of the connection from the neuron to neuron and cause **excitation** or **inhibition** of a subsequence neuron. This is where information is stored.
- The information processing abilities of biological neural systems must follow from highly **parallel processes** operating on representations that are **distributed** over many neurons. One motivation for ANN is to capture this kind of highly **parallel computation** based on distributed representations.

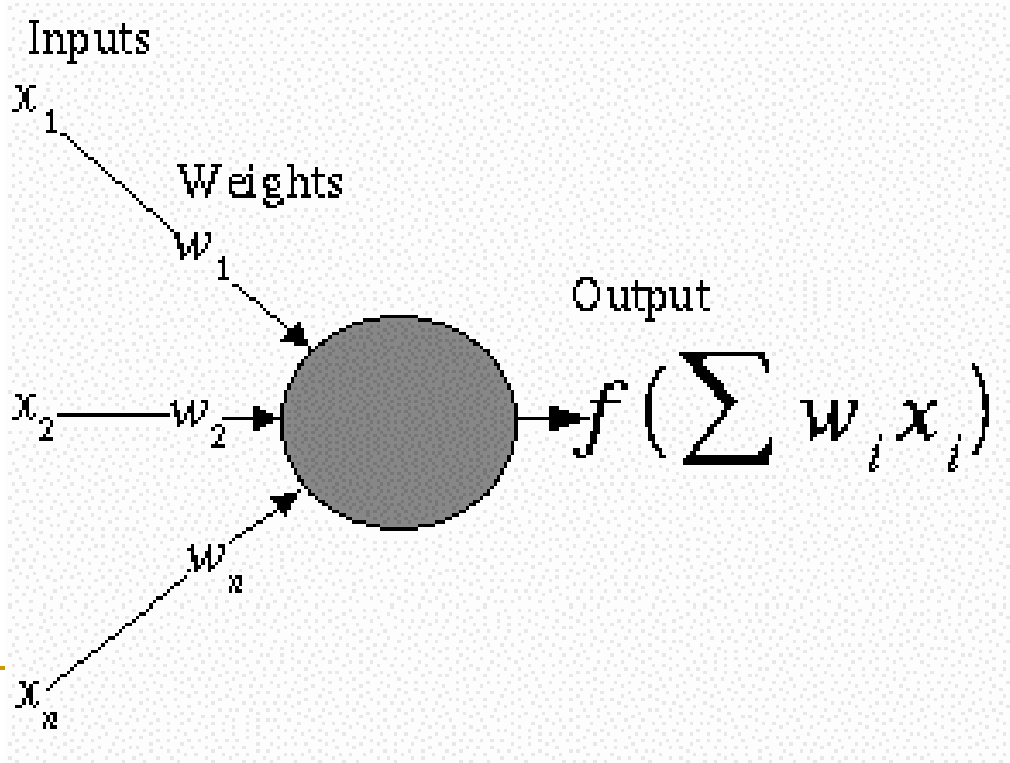
2. NEURAL NETWORK REPRESENTATION

- An ANN is composed of processing elements called or ***perceptrons***, organized in different ways to form the network's structure.

Processing Elements

- An ANN consists of perceptrons. Each of the perceptrons receives inputs, processes inputs and delivers a single output.

The input can be raw input data or the output of other perceptrons. The output can be the final result (e.g. 1 means yes, 0 means no) or it can be inputs to other perceptrons.

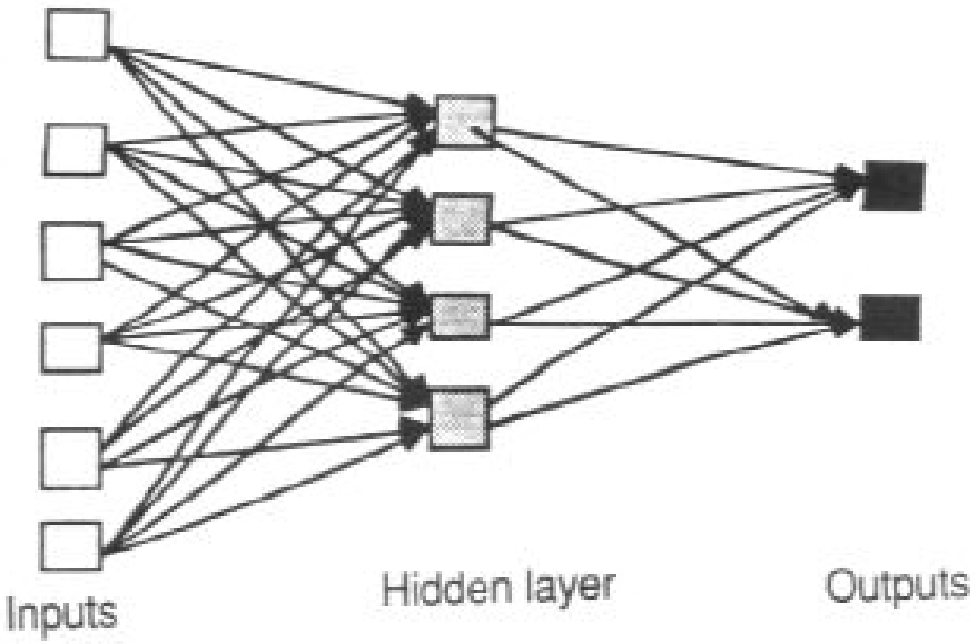


The network

- Each ANN is composed of a collection of perceptrons grouped in layers. A typical structure is shown in Fig.2.

Note the three layers: input, intermediate (called the ***hidden layer***) and output.

Several hidden layers can be placed between the input and output layers.



Appropriate Problems for Neural Network

- ANN learning is well-suited to problems in which the training data corresponds to noisy, complex sensor data. It is also applicable to problems for which more symbolic representations are used.
- The backpropagation (BP) algorithm is the most commonly used ANN learning technique. It is appropriate for problems with the characteristics:
  - Input is high-dimensional discrete or real-valued (e.g. raw sensor input)
  - Output is discrete or real valued
  - Output is a vector of values
  - Possibly noisy data
  - Long training times accepted
  - Fast evaluation of the learned function required.
  - Not important for humans to understand the weights
- Examples:
  - Speech phoneme recognition
  - Image classification
  - Financial prediction

3. PERCEPTRONS

- A perceptron 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
  - –1 otherwise.
- Given real-valued inputs x_1 through x_n, the output $o(x_1, \ldots, x_n)$ computed by the perceptron is

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

  where w_i is a real-valued constant, or *weight*.
- Notice the quantify $(-w_0)$ is a **threshold** that the weighted combination of inputs $w_1 x_1 + \ldots + w_n x_n$ must surpass in order for perceptron to output a 1.

- To simplify notation, we imagine an additional constant input $x_0 = 1$, allowing us to write the above inequality as

$$\sum_{i=0}^{n} w_i x_i > 0$$

- Learning a perceptron involves choosing values for the weights $w_0, w_1, \ldots, w_n$.

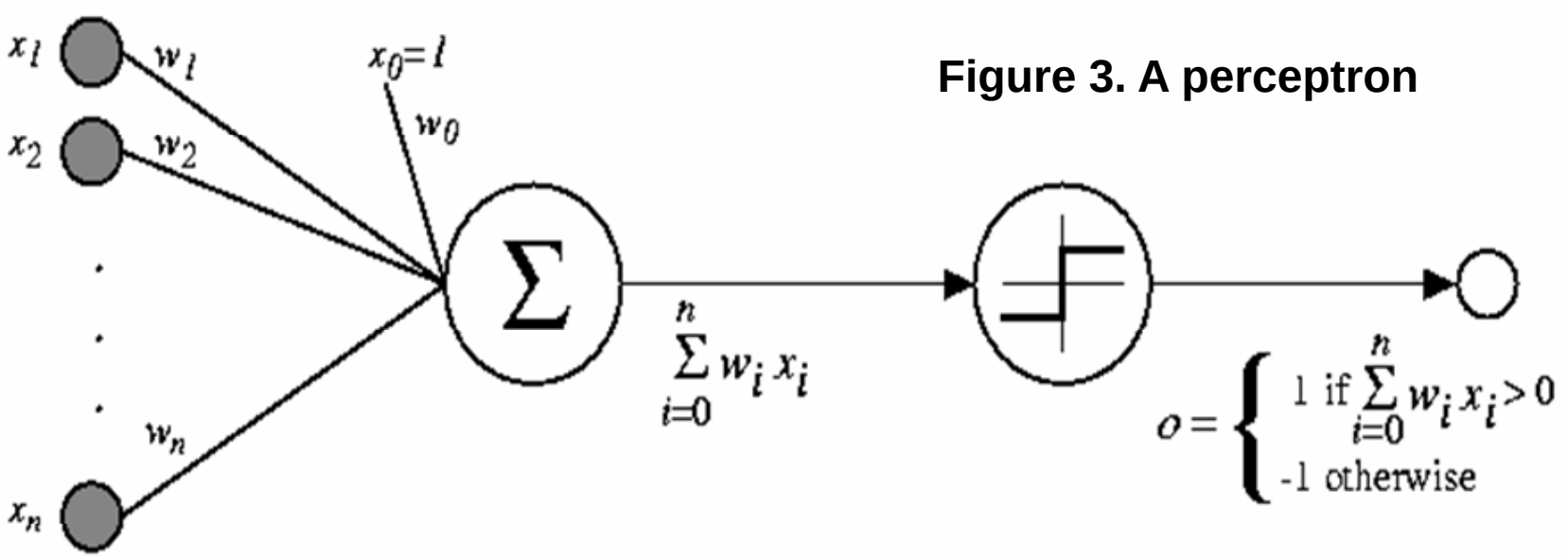


Figure 3. A perceptron

Representation Power of Perceptrons

- We can view the perceptron as representing a hyperplane **decision surface** in the *n*-dimensional space of instances (i.e. points). 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, as in Figure 4. The equation for this decision hyperplane is

$$\vec{w} \cdot \vec{x} = 0$$

Some sets of positive and negative examples cannot be separated by any hyperplane. Those that can be separated are called *linearly separated set of examples.*

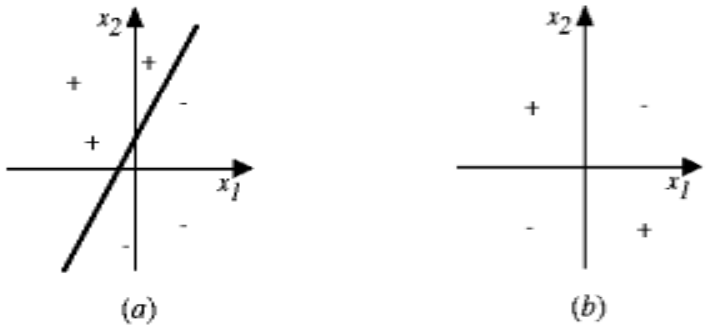


Figure 4. Decision surface

A single perceptron can be used to represent many boolean functions.

- **AND function :**

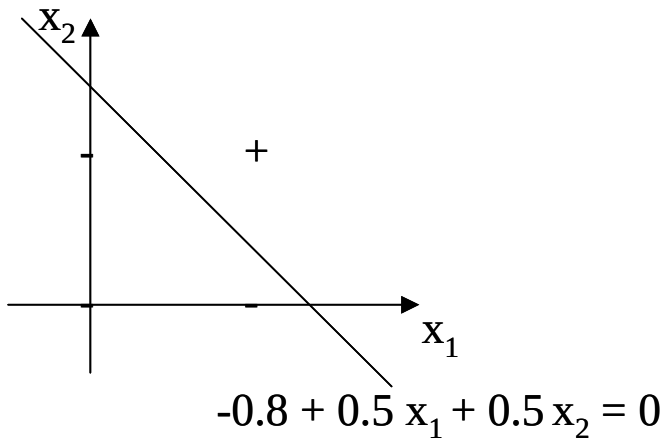
<Training examples>		
x_1	x_2	output
0	0	-1
0	1	-1
1	0	-1
1	1	1

<Test Results>			
x_1	x_2	$\sum w_i x_i$	output
0	0	-0.8	-1
0	1	-0.3	-1
1	0	-0.3	-1
1	1	0.2	1

Decision hyperplane :

$$w_0 + w_1 x_1 + w_2 x_2 = 0$$

$$-0.8 + 0.5 x_1 + 0.5 x_2 = 0$$



OR function

- The two-input perceptron can implement the OR function when we set the weights: $w_0 = -0.3$, $w_1 = w_2 = 0.5$

<Training examples>

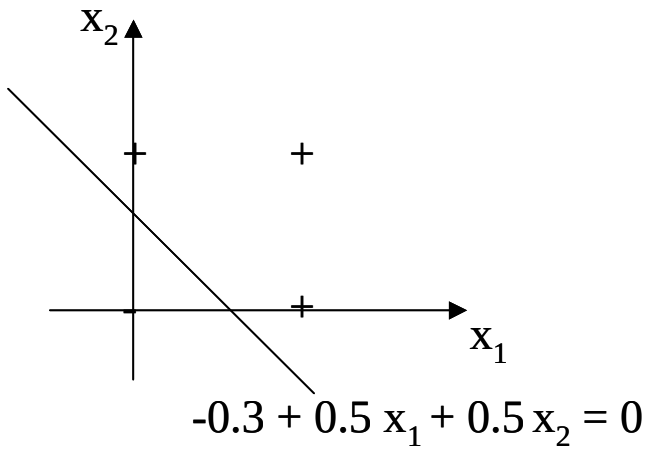
x_1	x_2	output
0	0	–1
0	1	1
1	0	1
1	1	1

Decision hyperplane :

$$w_0 + w_1 x_1 + w_2 x_2 = 0$$
$$-0.3 + 0.5 x_1 + 0.5 x_2 = 0$$

<Test Results>

x_1	x_2	$\Sigma w_i x_i$	output
0	0	–0.3	–1
0	1	0.2	–1
1	0	0.2	–1
1	1	0.7	1

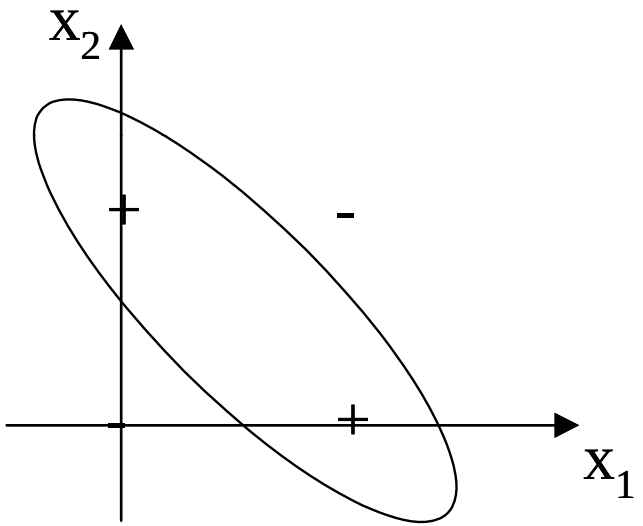


XOR function :

It's impossible to implement the XOR function by a single perception.

<Training examples>

x^1	x^2	out put
0	0	- 1
0	1	1
1	0	1
1	1	- 1



A two-layer network of perceptrons can represent XOR function.

Refer to this equation,

$$x_1 \oplus x_2 = x_1\overline{x}_2 + \overline{x}_1 x_2$$

Perceptron training rule

- Although we are interested in learning networks of many interconnected units, let us begin by understanding how to learn the weights for a single perceptron.
- Here learning is to determine a weight vector that causes the perceptron to produce the correct **+1** or **–1** for each of the given training examples.
- Several algorithms are known to solve this learning problem. Here we consider two: the **perceptron rule** and the **delta rule**.

- **One way to learn an acceptable weight vector is to begin with random weights, then iteratively apply the perceptron to each training example, modifying the perceptron weights whenever it misclassifies an example.** This process is repeated, iterating through the training examples as many as times needed until the perceptron *classifies* all training examples correctly.

- Weights are modified at each step according to the perceptron training rule, which revises the weight w_i associated with input x_i according to the rule.

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

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

- Here:

t is target output value for the current training example

o is perceptron output

η is small constant (e.g., 0.1) called *learning rate*

Perceptron training rule (cont.)

- The role of the **learning rate** is to moderate the degree to which weights are changed at each step. It is usually set to some small value (e.g. 0.1) and is sometimes made to decrease as the number of weight-tuning iterations increases.
- We can prove that the algorithm will converge
  - If training data is linearly separable
  - and η sufficiently small.
- If the data is not linearly separable, convergence is not assured.

Gradient Descent and the Delta Rule

- Although the perceptron rule finds a successful weight vector when the training examples are linearly separable, it can fail to converge if the examples are *not linearly separatable*. A second training rule, called the **delta rule**, is designed to overcome this difficulty.
- The key idea of delta rule: to use *gradient descent* to search the space of possible weight vector to find the weights that best fit the training examples. This rule is important because it provides the *basis* for the backpropagration algorithm, which can learn networks with many interconnected units.
- The delta training rule: considering the task of training an unthresholded perceptron, that is a *linear unit*, for which the output o is given by:

$$o = w_0 + w_1 x_1 + \cdots + w_n x_n \qquad (1)$$

- Thus, a linear unit corresponds to the first stage of a perceptron, without the threhold.

- In order to derive a weight learning rule for linear units, let specify a measure for the ***training error*** of a weight vector, relative to the training examples. The Training Error can be computed as the following squared error

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

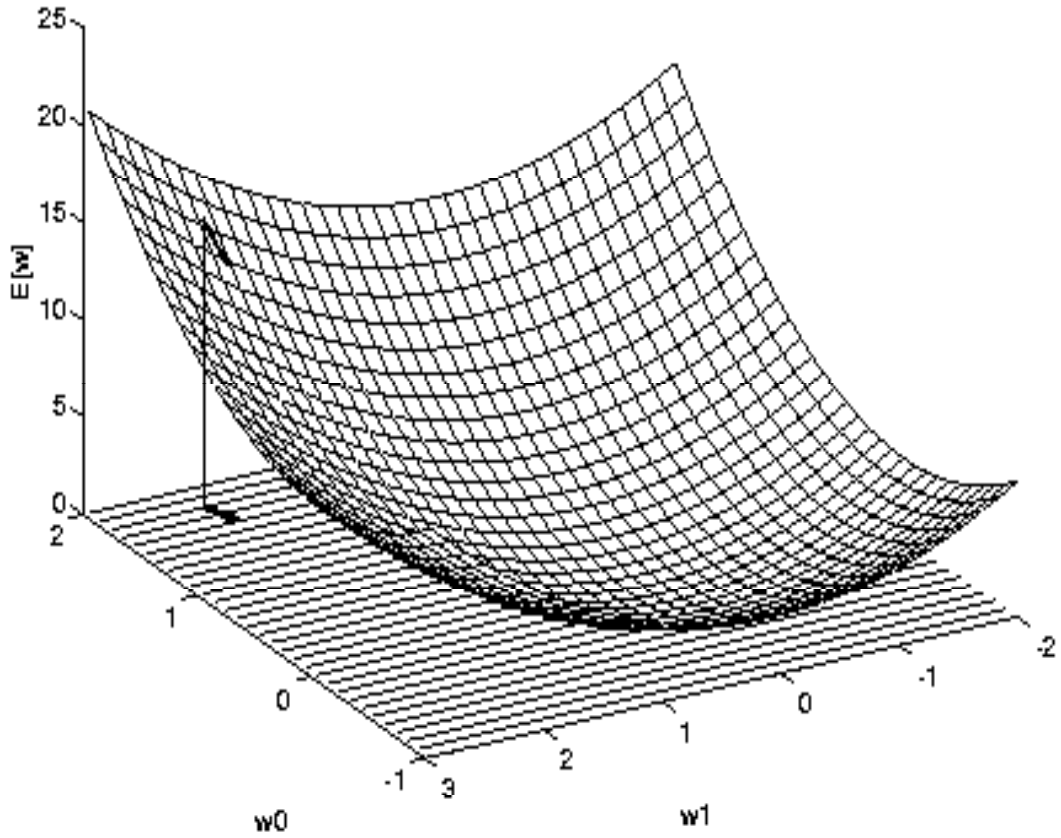
where *D* is set of training examples, t_d is the target output for the training example *d* and o_d is the output of the linear unit for the training example *d*.

Here we characterize *E* as ***a function of weight vector*** because the linear unit output *O* depends on this weight vector.

Hypothesis Space

- To understand the gradient descent algorithm, it is helpful to visualize the entire space of possible weight vectors and their associated *E* values, as illustrated in Figure 5.
  - Here the axes w_o,w_1 represents possible values for the two weights of a simple linear unit. The w_o,w_1 plane represents the entire hypothesis space.
  - The vertical axis indicates the error *E* relative to some fixed set of training examples. The error surface shown in the figure summarizes the desirability of every weight vector in the hypothesis space.
- For linear units, this **error surface** must be **parabolic** with a single global minimum. And we desire a weight vector with this minimum.

Figure 5. The error surface



How can we calculate the direction of steepest descent along the error surface? This direction can be found by computing the derivative of E w.r.t. each component of the vector w.

Derivation of the Gradient Descent Rule

- This vector derivative is called the *gradient* of E with respect to the vector $\langle w_0,\ldots,w_n \rangle$, written ∇E .

$$\nabla E[\vec{w}] \equiv \left[\frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, \cdots \frac{\partial E}{\partial w_n}\right] \quad (3)$$

Notice ∇E is itself a vector, whose components are the partial derivatives of E with respect to each of the w_i. When interpreted as a vector in weight space, the gradient specifies the **direction** that produces the steepest increase in E. The negative of this vector therefore gives the direction of steepest decrease.

Since the gradient specifies the direction of steepest increase of E, the training rule for gradient descent is

$$\mathbf{w} \leftarrow \mathbf{w} + \Delta\mathbf{w}$$

where

$$\Delta\vec{w} = -\eta \nabla E[\vec{w}] \quad (4)$$

- Here η is a positive constant called the ***learning rate***, which determines the step size in the gradient descent search. The negative sign is present because we want to move the weight vector in the direction that decreases E. This training rule can also be written in its component form

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

where

$$\Delta w_i = -\eta \frac{\partial E}{\partial w_i} \qquad (5)$$

which makes it clear that steepest descent is achieved by altering each component w_i of weight vector in proportion to $\partial E/\partial w_i$.

The vector of $\partial E/\partial w_i$ derivatives that form the gradient can be obtained by differentiating *E* from Equation (2), as

$$\frac{\partial E}{\partial w_i} = \frac{\partial}{\partial w_i}\frac{1}{2}\sum_d (t_d - o_d)^2$$
$$= \frac{1}{2}\sum_d \frac{\partial}{\partial w_i}(t_d - o_d)^2$$
$$= \frac{1}{2}\sum_d 2(t_d - o_d)\frac{\partial}{\partial w_i}(t_d - o_d)$$
$$= \sum_d (t_d - o_d)\frac{\partial}{\partial w_i}(t_d - \vec{w}\cdot\vec{x_d})$$
$$\frac{\partial E}{\partial w_i} = \sum_d (t_d - o_d)(-x_{i,d}) \qquad (6)$$

where x_{id} denotes the single input component x_i for the training example *d*. We now have an equation that gives $\partial E/\partial w_i$ in terms of the linear unit inputs x_{id}, output o_d and the target value t_d associated with the training example. Substituting Equation (6) into Equation (5) yields the **weight update rule** for gradient descent.

$$\Delta w_i = \eta \sum_{d \in D} (t_d - o_d) x_{id} \qquad (7)$$

- The **gradient descent** algorithm for training linear units is as follows: Pick an initial random weight vector. Apply the linear unit to all training examples, them compute Δw_i for each weight according to Equation (7). Update each weight w_i by adding Δw_i , them repeat the process. The algorithm is given in Figure 6.
- Because the **error surface** contains only a single global minimum, this algorithm will converge to a weight vector with minimum error, regardless of whether the training examples are linearly separable, given a sufficiently small η is used.
- If η is too large, the **gradient descent** search runs the risk of overstepping the minimum in the error surface rather than settling into it. For this reason, one common modification to the algorithm is to **gradually reduce** the value of η as the number of gradient descent steps grows.

Figure 6. Gradient Descent algorithm for training a linear unit.

GRADIENT-DESCENT($training\_examples, \eta$)

Each training example is a pair of the form $\langle \vec{x}, t \rangle$, *where* $\vec{x}$ *is the vector of input values, and* t *is the target output value.* η *is the learning rate (e.g., .05).*

- Initialize each w_i to some small random value
- Until the termination condition is met, Do
  - Initialize each Δw_i to zero.
  - For each $\langle \vec{x}, t \rangle$ in *training_examples*, Do
    - Input the instance $\vec{x}$ 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:
  - Converging to a local minimum can sometimes be quite slow (i.e., it can require many thousands of steps).
  - If there are multiple local minima in the error surface, then there is no guarantee that the procedure will find the global minimum.
- One common variation on gradient descent intended to alleviate these difficulties is called *incremental gradient descent* (or *stochastic gradient descent*). The key differences between standard gradient descent and stochastic gradient descent are:
  - In standard gradient descent, the error is summed over **all examples** before upgrading weights, whereas in stochastic gradient descent weights are updated upon examining **each training example**.
  - The modified training rule is like the training example we update the weight according to

$$\Delta w_i = \eta(t - o)\, x_i \qquad (10)$$

- Summing over multiple examples in standard gradient descent requires more computation per weight update step. On the other hand, because it uses the true gradient, standard gradient descent is often used with a larger step size per weight update than stochastic gradient descent.

Batch mode Gradient Descent:
Do until satisfied

1. Compute the gradient $\nabla E_D[\vec{w}]$
2. $\vec{w} \leftarrow \vec{w} - \eta \nabla E_D[\vec{w}]$

Incremental mode Gradient Descent:
Do until satisfied

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

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

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

- *Stochastic gradient descent* (i.e. incremental mode) can sometimes avoid falling into local minima because it uses the various gradient of *E* rather than overall gradient of *E* to guide its search.
- Both stochastic and standard gradient descent methods are commonly used in practice.

Summary

- *Perceptron training rule*
  - Perfectly classifies training data
  - Converge, provided the training examples are linearly separable

- *Delta Rule using gradient descent*
  - Converge asymptotically to minimum error hypothesis
  - Converge regardless of whether training data are linearly separable

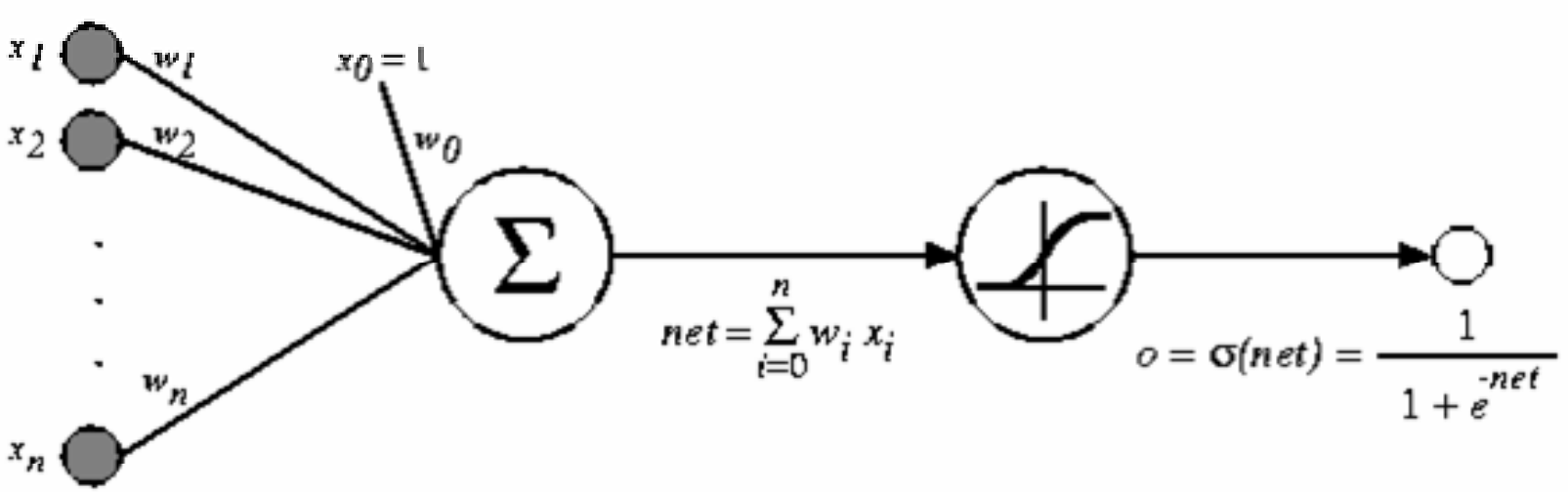
3. MULTILAYER NETWORKS AND THE BACKPROPOGATION ALGORITHM

- Single perceptrons can only express linear decision surfaces. In contrast, the kind of multilayer networks learned by the backpropagation algorithm are capaple of expressing a rich variety of **nonlinear** decision surfaces.
- This section discusses how to learn such multilayer networks using a gradient descent algorithm similar to that discussed in the previous section.

A Differentiable Threshold Unit

- What type of unit as the basis for multilayer networks ?
  - Perceptron : not differentiable -> can't use gradient descent
  - Linear Unit : multi-layers of linear units -> still produce only linear function
  - Sigmoid Unit : *smoothed*, *differentiable threshold function*

Figure 7. The sigmoid threshold unit.



$\sigma(x)$ is the sigmoid function

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

- Like the perceptron, the sigmoid unit first computes a linear combination of its inputs, then applies a threshold to the result. In the case of sigmoid unit, however, the threshold output is a continuous function of its input.
- The sigmoid function $\sigma(x)$ is also called the **logistic function**.
- Interesting property:

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

• Output ranges between 0 and 1, increasing monotonically with its input.

We can derive gradient decent rules to train

• One sigmoid unit

• *Multilayer networks* of sigmoid units $\Rightarrow$ Backpropagation

The Backpropagation (BP)Algorithm

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

- $$E(\mathbf{w}) = \tfrac{1}{2} \sum_{d \in D} \sum_{k \in outputs} (t_{kd} - o_{kd})^2 \qquad (13)$$

where *outputs* is the set of output units in the network, and t_{kd} and o_{kd} are the target and output values associated with the *k*th output unit and training example *d*.

The Backpropagation Algorithm (cont.)

- The BP algorithm is presented in Figure 8. The algorithm applies to layered feedforward networks containing 2 layers of sigmoid units, with units at each layer connected to all units from the preceding layer.
- This is an *incremental gradient descent version* of Backpropagation.
- The notation is as follows:
  - x_{ij} denotes the input from node *i* to unit *j*, and w_{ij} denotes the corresponding weight.
  - δ_n denotes the error term associated with unit *n*. It plays a role analogous to the quantity (*t* – *o*) in our earlier discussion of the delta training rule.

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

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

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

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

where

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

. The Backpropagation algorithm

- In the BP algorithm, step1 propagates the input forward through the network. And the steps 2, 3 and 4 propagates the errors backward through the network.
- The main loop of BP repeatedly iterates over the training examples. For each training example, it applies the ANN to the example, calculates the error of the network output for this example, computes the gradient with respect to the error on the example, then updates all weights in the network. This gradient descent step is iterated until ANN performs acceptably well.
- A variety of termination conditions can be used to halt the procedure.
  - One may choose to halt after a fixed number of iterations through the loop, or
  - once the error on the training examples falls below some threshold, or
  - once the error on a separate validation set of examples meets some criteria.

Adding Momentum

- Because BP is a widely used algorithm, many variations have been developed. The most common is to alter the weight-update rule in Step 4 in the algorithm by making the weight update on the *n*th iteration *depend partially on* the update that occurred during the (*n* - 1)th iteration, as follows:

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

Here $\Delta w_{i,j}(n)$ is the weight update performed during the *n*-th iteration through the main loop of the algorithm.

- *n*-th iteration update depend on (*n-1*)th iteration
- α: constant between 0 and 1 is called the *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.

REMARKS ON THE BACKPROPAGATION ALGORITHM

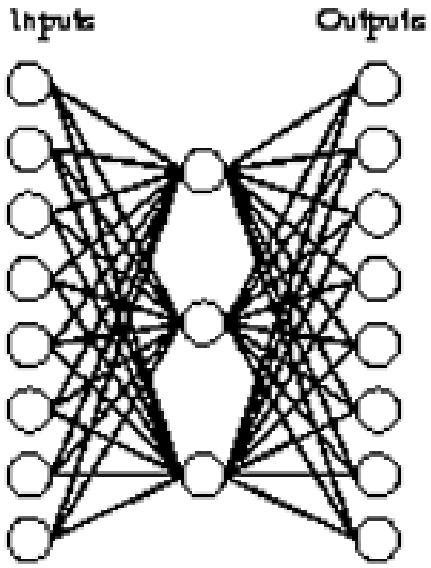
- **Convergence and Local Minima**
  - Gradient descent to some local minimum
    - Perhaps not global minimum...
  - Heuristics to alleviate the problem of local minima
    - Add momentum
    - Use stochastic gradient descent rather than true gradient descent.
    - Train multiple nets with different initial weights using the same data.

Expressive Capabilities of ANNs

- **Boolean functions:**
  - Every boolean function can be represented by network with two layers of units where the number of hidden units required grows exponentially.
- **Continuous functions:**
  - Every bounded continuous function can be approximated with arbitrarily small error, by network with two layers of units [Cybenko 1989; Hornik et al. 1989]
- **Arbitrary functions:**
  - Any function can be approximated to arbitrary accuracy by a network with three layers of units [Cybenko 1988].

Hidden layer representations

- Hidden layer representations
  - This 8x3x8 network was trained to learn the identity function.
  - 8 training examples are used.
  - After 5000 training iterations, the three hidden unit values encode the eight distinct inputs using the encoding shown on the right.



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

Learning the 8x3x8 network

Most of the interesting weight changes occurred during the first 2500 iterations.

Figure 10.a The plot shows the sum of squared errors for each of the eight output units as the number of iterations increases. The sum of square errors for each output **decreases** as the procedure proceeds, more quickly for some output units and less quickly for others.

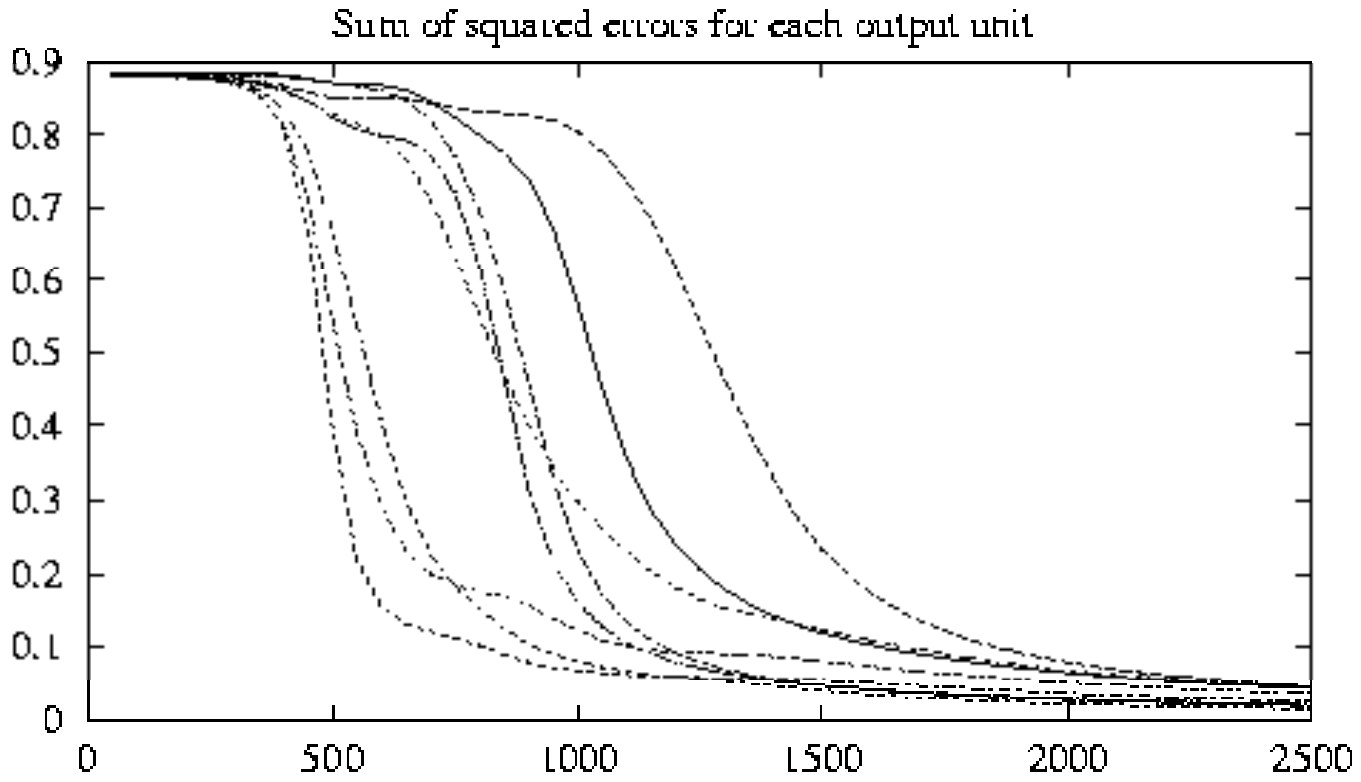
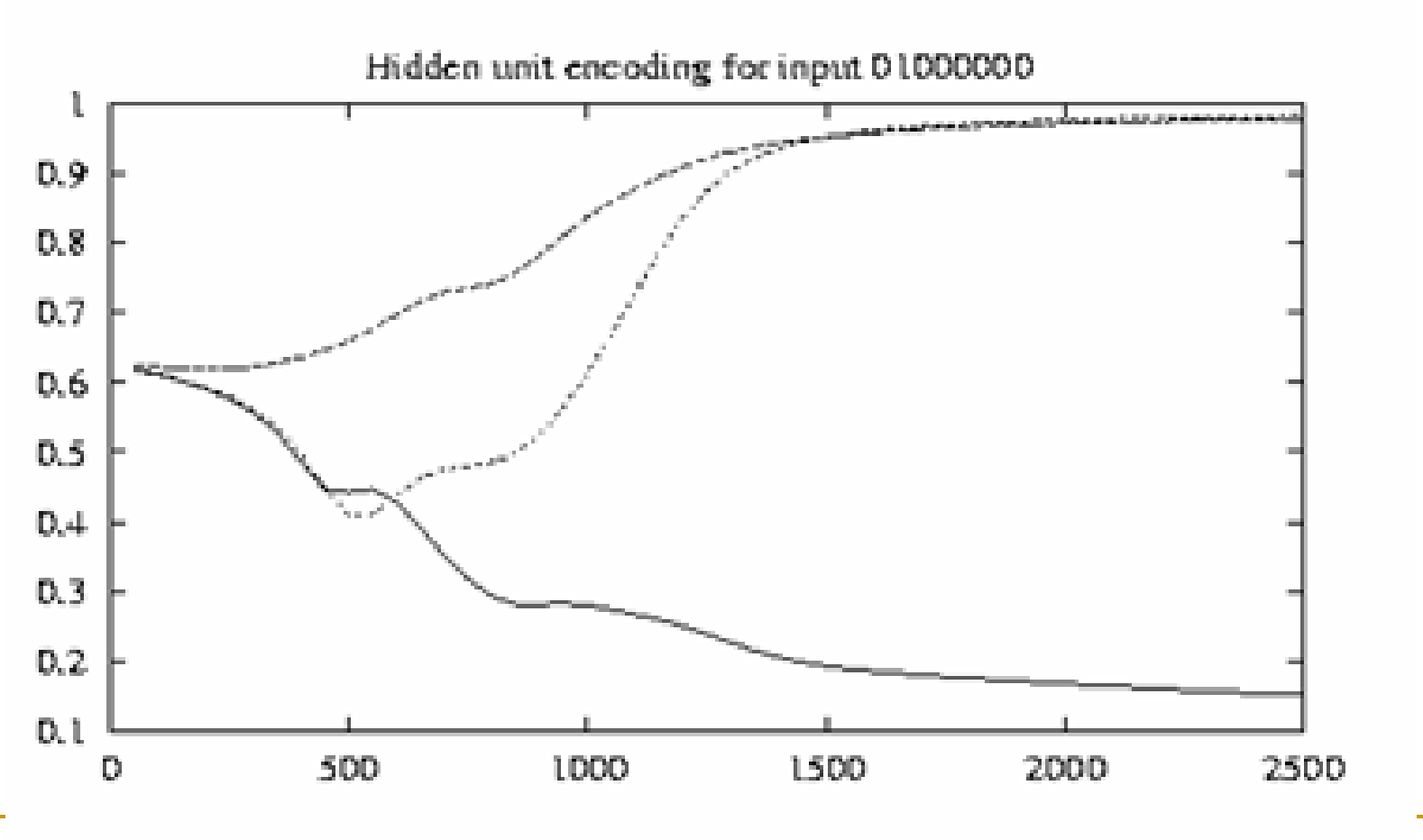
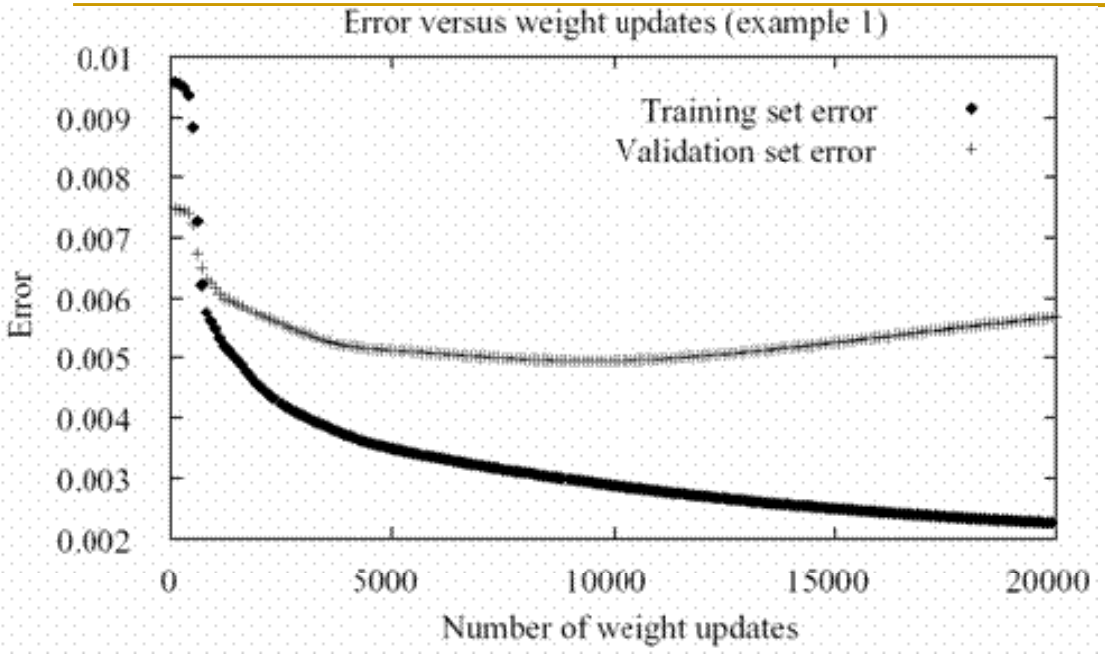


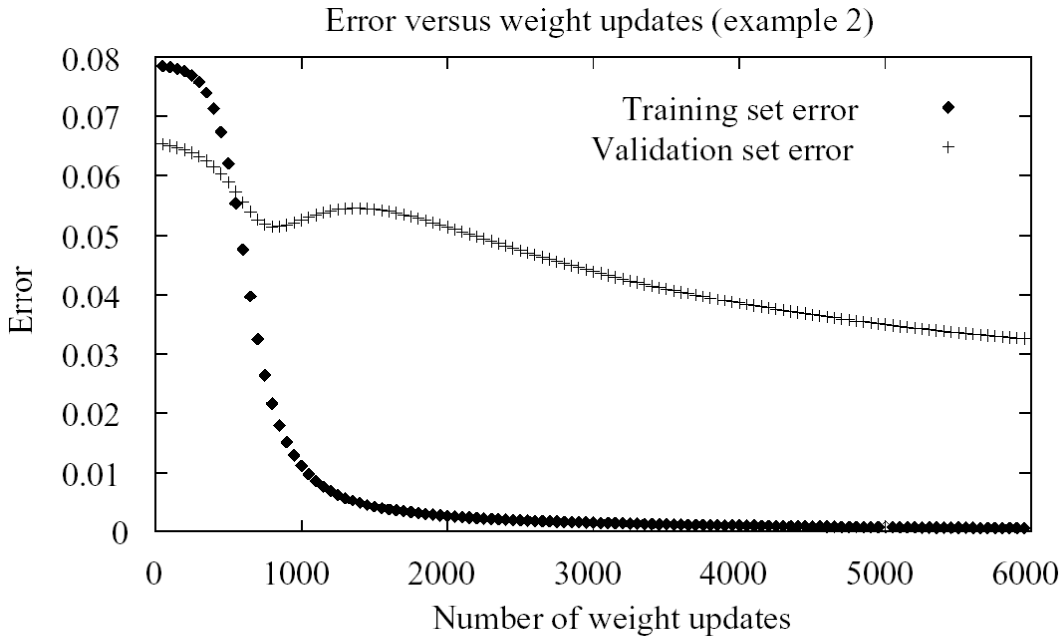
Figure 10.b Learning the $8 \times 3 \times 8$ network. The plot shows the evolving hidden layer representation for the input string “010000000”. The network passes through a number of different encodings before **converging** to the final encoding.



Generalization, Overfitting and Stopping Criterion

- Termination condition
  - Until the error E falls below some predetermined threshold
  - This is a poor strategy
- Overfitting problem
  - Backpropagation is susceptible to overfitting the training examples at the cost of decreasing **generalization accuracy** over other unseen examples.
  - To see the danger of minimizing the error over the training data, consider how the error *E* varies with the number of weight iteration.





The ***generalization accuracy*** measured over the training examples first decreases, then increases, even as the error over training examples continues to decrease.

This occurs because the weights are being tuned to fit idiosyncrasies of the training examples that are not representative of the general distribution of examples.

Techniques to overcome overfitting problem

- *Weight decay* : **Decrease** each weight by some small factor during each iteration. The motivation for this approach is to keep weight values small.
- *Cross-validation*: a set of **validation data** in addition to the training data. The algorithm monitors the error w.r.t. this validation data while using the training set to drive the gradient descent search.
  - How many weight-tuning iterations should the algorithm perform? It should use the number of iterations that produces the lowest error over the validation set.
  - Two copies of the weights are kept: one copy for training and a separate copy of the best weights thus far, measured by their error over the validation set.
  - Once the trained weights reach a higher error over the validation set than the stored weights, training is terminated and the stored weights are returned.

NEURAL NETWORK APPLICATION DEVELOPMENT

The development process for an ANN application has eight steps.

- *Step 1: (Data collection)* The data to be used for the training and testing of the network are collected. Important considerations are that the particular problem is amenable to neural network solution and that adequate data exist and can be obtained.
- *Step 2: (Training and testing data separation)* Trainning data must be identified, and a plan must be made for testing the performance of the network. The available data are divided into **training** and **testing** data sets. For a moderately sized data set, 80% of the data are randomly selected for training, 10% for testing, and 10% secondary testing.
- *Step 3: (Network architecture)* A network architecture and a learning method are selected. Important considerations are the exact number of perceptrons and the number of layers.

- *Step 4: (Parameter tuning and weight initialization)* There are parameters for tuning the network to the desired learning performance level. Part of this step is initialization of the network weights and parameters, followed by modification of the parameters as training performance feedback is received.
  - Often, the initial values are important in determining the effectiveness and length of training.
- *Step 5: (Data transformation)* Transforms the application data into the type and format required by the ANN.
- *Step 6: (Training)* Training is conducted iteratively by presenting input and desired or known output data to the ANN. The ANN computes the outputs and adjusts the weights until the computed outputs are within an acceptable tolerance of the known outputs for the input cases.

- *Step 7: (Testing)* Once the training has been completed, it is necessary to test the network.
  - The **testing** examines the performance of the network using the derived weights by measuring the ability of the network to classify the testing data correctly.
  - **Black-box testing** (comparing test results to historical results) is the primary approach for verifying that inputs produce the appropriate outputs.
- *Step 8: (Implementation)* Now a stable set of weights are obtained.
  - Now the network can reproduce the desired output given inputs like those in the training set.
  - The network is ready to use as a stand-alone system or as part of another software system where new input data will be presented to it and its output will be a recommended decision.

BENEFITS AND LIMITATIONS OF NEURAL NETWORKS

6.1 Benefits of ANNs

- *Usefulness for pattern recognition, classification, generalization, abstraction and interpretation of imcomplete and noisy inputs.* (e.g. handwriting recognition, image recognition, voice and speech recognition, weather forecasing).
- *Providing some human characteristics to problem solving* that are difficult to simulate using the logical, analytical techniques of expert systems and standard software technologies. (e.g. financial applications).
- *Ability to solve new kinds of problems*. ANNs are particularly effective at solving problems whose solutions are difficult, if not impossible, to define. This opened up a new range of decision support applications formerly either difficult or impossible to computerize.

- *Robustness*. ANNs tend to be more robust than their conventional counterparts. They have the ability to cope with imcomplete or fuzzy data. ANNs can be very tolerant of faults if properly implemented.
- *Fast processing speed*. Because they consist of a large number of massively interconnected processing units, all operating in parallel on the same problem, ANNs can potentially operate at considerable speed (when implemented on parallel processors).
- *Flexibility and ease of maintenaince*. ANNs are very flexible in adapting their behavior to new and changing environments. They are also easier to maintain, with some having the ability to learn from experience to improve their own performance.

6.2 Limitations of ANNs

- ANNs do not produce an *explicit model* even though new cases can be fed into it and new results obtained.
- *ANNs lack explanation capabilities*. Justifications for results is difficults to obtain because the connection weights usually do not have obvious interpretaions.

7. SOME ANN APPLICATIONS

ANN application areas:

- Tax form processing to identify tax fraud
- Enhancing auditing by finding irregularites
- Bankruptcy prediction
- Customer credit scoring
- Loan approvals
- Credit card approval and fraud detection
- Financial prediction
- Energy forecasting
- Computer access security (intrusion detection and classification of attacks)
- Fraud detection in mobile telecommunication networks

Customer Loan Approval with Neural Networks - Problem Statement

- Many stores are now offering their customers the possibility of applying for a loan directly at the store, so that they can proceed with the purchase of relatively expensive items without having to put up the entire capital all at once.
- Initially this practice of offering consumer loans was found only in connection with expensive purchases, such as cars, but it is now commonly offered at major department stores for purchases of washing machines, televisions, and other consumer goods.
- The loan applications are filled out at the store and the consumer deals only with the store clerks for the entire process. The store, however, relies on a financial company (often a bank) that handles such loans, evaluates the applications, provides the funds, and handles the credit recovery process when a client defaults on the repayment schedule.

- For this study, there were 1000 records of consumer loan applications that were granted by a bank, together with the indication whether each loan had been always paid on schedule or there had been any problem.

- The provided data did not make a more detailed distinction about the kind of problem encountered by those “bad” loans, which could range from a single payment that arrived late to a complete defaulting on the loan.

ANN Application to Loan Approval

- Each application had ***15 variables*** that included the number of members of the household with an income, the amount of the loan requested, whether or not the applicant had a phone in his/her house, etc.

Table 1: Input and output variables

	Input variables	Variable values
1	N° of relatives	from 1 to total components
2	N° of relatives with job	from 0 to total components
3	Telephone number	0,1
4	Real estate	0,1
5	Residence seniority	from 0 to date of loan request
6	Other loans	0, 1, 2
7	Payment method	0,1
8	Job type	0,1,2,3
9	Job seniority	from 0 to date of loan request
10	Net monthly earnings	integer
11	Collateral	0,1,2
12	Loan type	0,1,2,3
13	Amount of loan	integer value
14	Amount of installment	integer value
15	Duration of loan	integer value

Computed output variable

1 Repayment probability from 0 to 100

Desired output variable

1	Real result of grant loan	0	if paymnent irregular or null
		100	if payment on schedule

- Some of these variables were numerical (*e.g.* the number of relatives, while other used a digit as a label to indicate a specific class (*e.g.* the values 0,1,2,3 of variable 8 referred to four different classes of employment).
- For each record a single variable indicated whether the loan reached was extinguished without any problem (Z=100) or with some problem (Z=0).
- In its *a-posteriori* analysis, the bank classified loans with Z=0 as “bad loans”. In the provided data, only about 6% of the loans were classified as “bad”. Thus, any ANN that classifies loans from a similar population ought to make errors in a percentage that is substantially lower than 6% to be of any use (otherwise, it could have simply classified all loans as good, resulting in an error on 6% of the cases).

- Out of 1000 available records, 400 were randomly selected as a training set for the configuration of the ANN, while the remaining 600 cases were then supplied to the configured ANN so that its computed output could be compared with the real value of variable Z.
- Beside the network topology, there are many parameters that must be set. One of the most critical parameters is the number of neurons constituting the hidden layer, as too few neurons can hold up the convergence of the training process, while too many neurons may result in a network that can "learn" very accurately (**straight memorization**) those cases that are in the training set, but is unable to generalize what has learned to the new cases in the testing set.
- The research team selected a network with ***10 hidden nodes*** as the one that provided the most promising performance; the number of iterations was set to ***20,000*** to allow a sufficient degree of learning, without loss of performance in generalization capability.

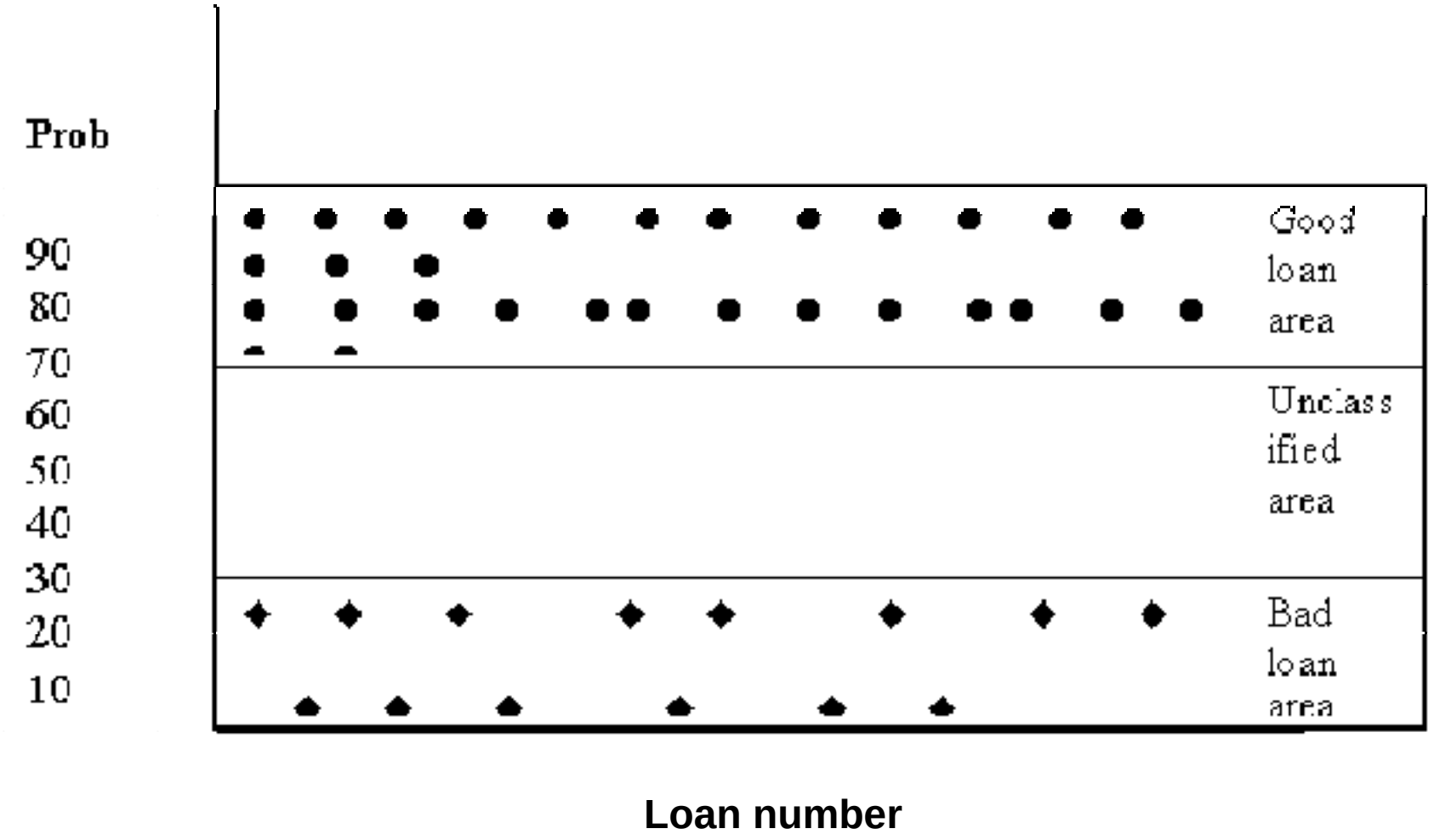
- The single output of our network turned out to be in the range from -30 to +130, whereas the corresponding “real” output was limited to the values Z=0 or Z=100. A negative value of the output would indicate a very bad loan and thus negative values were clamped to zero; similarly, output values above 100 were assigned the value of 100.
- A 30% tolerance was used on the outputs so that loans would be classified as “good” if the ANN computed a value above 70, and “bad” is their output was less then 30. Loans that fell in the intermediate band [30, 70] were left as “unclassified”. The width of this band is probably overly conservative and a smaller one would have sufficed, at the price of possibly granting marginal loans, or refusing loans that could have turned out to be good at the end. The rationale for the existence of the “unclassified” band is to provide an alarm requesting a more detailed examination unforeseen and unpredictable circumstance.

- This specific ANN was then supplied with the remaining ***600*** cases of the testing set.
- This set contained ***38*** cases that had been classified as bad (Z=0), while the remaining ***562*** cases had been repaid on schedule.
- Clearly the ANN separates the given cases into two non-overlapping bands: the good ones near the top and the bad ones near the bottom. No loan was left unclassified, so in this case there would have been no cases requiring additional (human) intervention.
- The ANN made exactly three mistakes in the classification of the test cases: those were 3 cases that the ANN classified as "good" loans, whereas in reality they turned out to be "bad". Manual, *a-posteriori* inspection of the values of their input variables did not reveal any obvious symptoms that they were "problem cases".
- What could have likely happened is that the applicant did not repay the loan as schedule due to some completely unforeseen and unpredictable circumstance. This is also supported by the fact that the bank officers themselves approved those three loans, thus one must presume that they did not look too risky at application time.

- The ANN, however, was more discriminating than the bank officers since the ANN would have denied 35 loan applications that scored less than 30.
- As is turns out, all those 35 loans had problems with their repayments and thus the bank would have been well advised to heed the network's classification and to deny those 35 applications. Had the bank followed that advise, 268 million liras would have not been put in jeopardy by the bank (out of a total of more than 3 billion liras of granted loans that were successfully repaid.)

F. D. Nittis, G. Tecchiolli & A. Zorat, Consumer Loan Classification Using Artificial Neural Networks, ICSC EIS'98 Conference, Spain Feb.,1998

Loan classification by ANN



Bankruptcy Prediction with Neural Networks

- There have been a lot of work on developing neural networks to predict bankruptcy using financial ratios and discriminant analysis. The ANN paradigm selected in the design phase for this problem was a three-layer feedforward ANN using backpropagation.
- The data for training the network consisted of a small set of numbers for well-known *financial ratios*, and data were available on the bankruptcy outcomes corresponding to known data sets. Thus, a supervised network was appropriate, and training time was not a problem.

Application Design

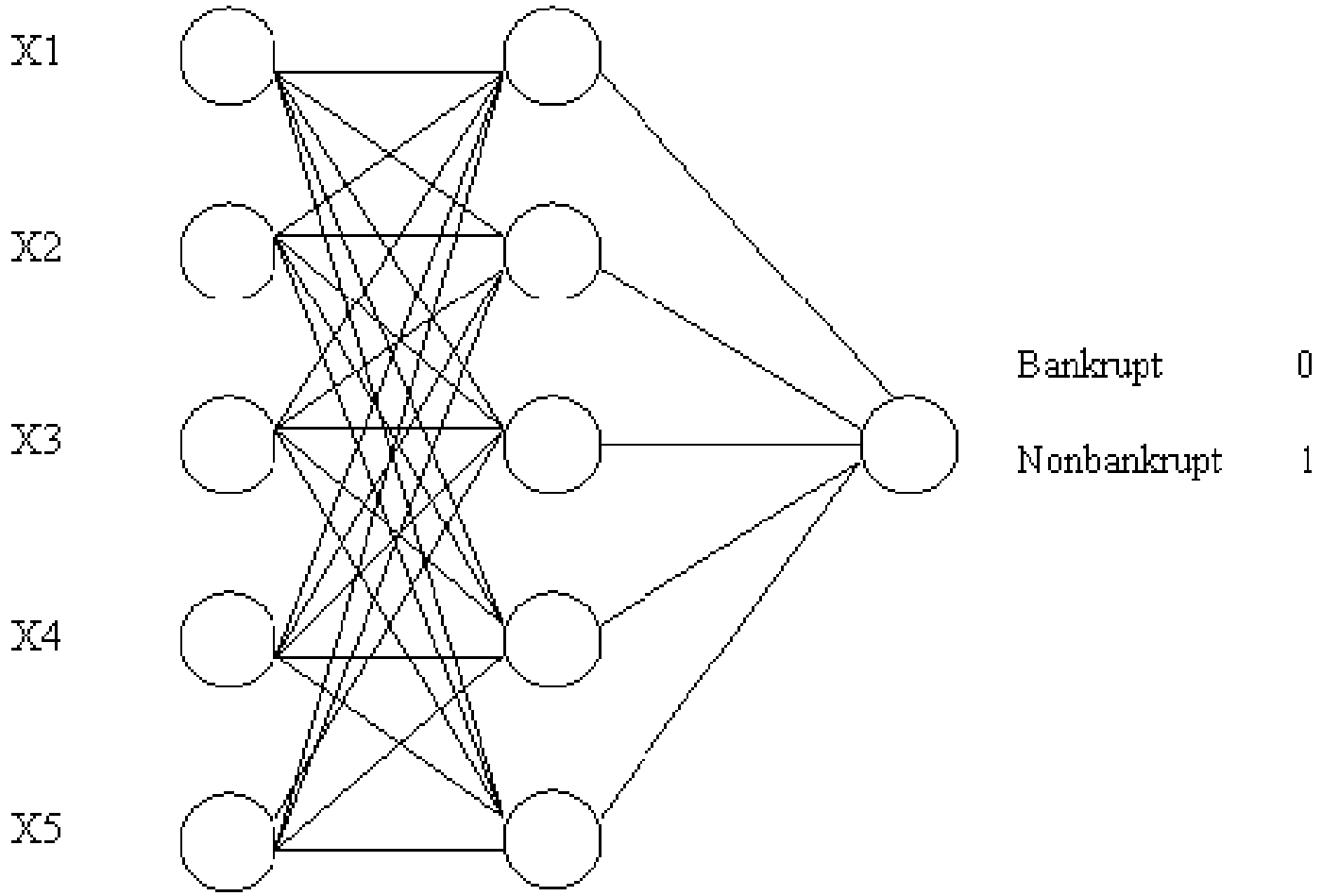
There are five input nodes, corresponding to five financial ratios:

- X1: Working capital/total assets
- X2: Retained earnings/total assets
- X3: Earnings before interest and taxes/total assets
- X4: Market value of equity/total debt
- X5: Sales/total assets

- A single output node gives the final classification showing whether the input data for a given firm indicated a potential bankruptcy (0) or nonbankruptcy (1).
- The data source consists of financial ratios for firms that did or did not go bankrupt between 1975 and 1982.
- Financial ratios were calculated for each of the five aspects shown above, each of which became the input for one of the five input nodes.
- For each set of data, the actual result, whether or not bankruptcy occurred, can be compared to the neural network's output to measure the performance of the network and monitor the training.

ANN Architecture

- The architecture of the ANN is shown in the following figure


X1
X2
X3
X4
X5
Bankrupt 0
Nonbankrupt 1

Training

- The data set, consisting of 129 firms, was partitioned into a training set and a test set. The training set of 74 firms consisted of 38 that went bankrupt and 36 that did not. The needed ratios were computed and stored in the input file to the neural network and in a file for a conventional *discriminant analysis* program for comparison of the two techniques.
- The neural network has three important parameters to be set: learning threshold, learning rate, and momentum.
  - The learning threshold allows the developer to vary the acceptable overall error for the training case.
  - The learning rate and momentum allow the developer to control the step sizes the network uses to adjust the weights as the errors between computed and actual outputs are fed back.

Testing

- The neural network was tested in two ways: by using the test data set and by comparison with discriminant analysis. The test set consisted of 27 bankrupt and 28 non-bankrupt firms. The neural network was able to correctly predict 81.5% of the bankrupt cases and 82.1% of the nonbankrupt cases.
  - Overrall, the ANN did much better predicting 22 out of the 27 actual cases (the discriminant analysis predicted only 16 cases correctly).
  - An analysis of the errors showed that 5 of the bankrupt firms classified as nonbankrupt were also misclassified by the discriminant analysis method. A similar situation occurred for the nonbankrupt cases.
- The result of the testing showed that neural network implementation is at least as good as the conventional approach. An accuracy of about 80% is usually acceptable for ANN applications. At this level, a system is useful because it automatically identifies problem situations for further analysis by a human expert.

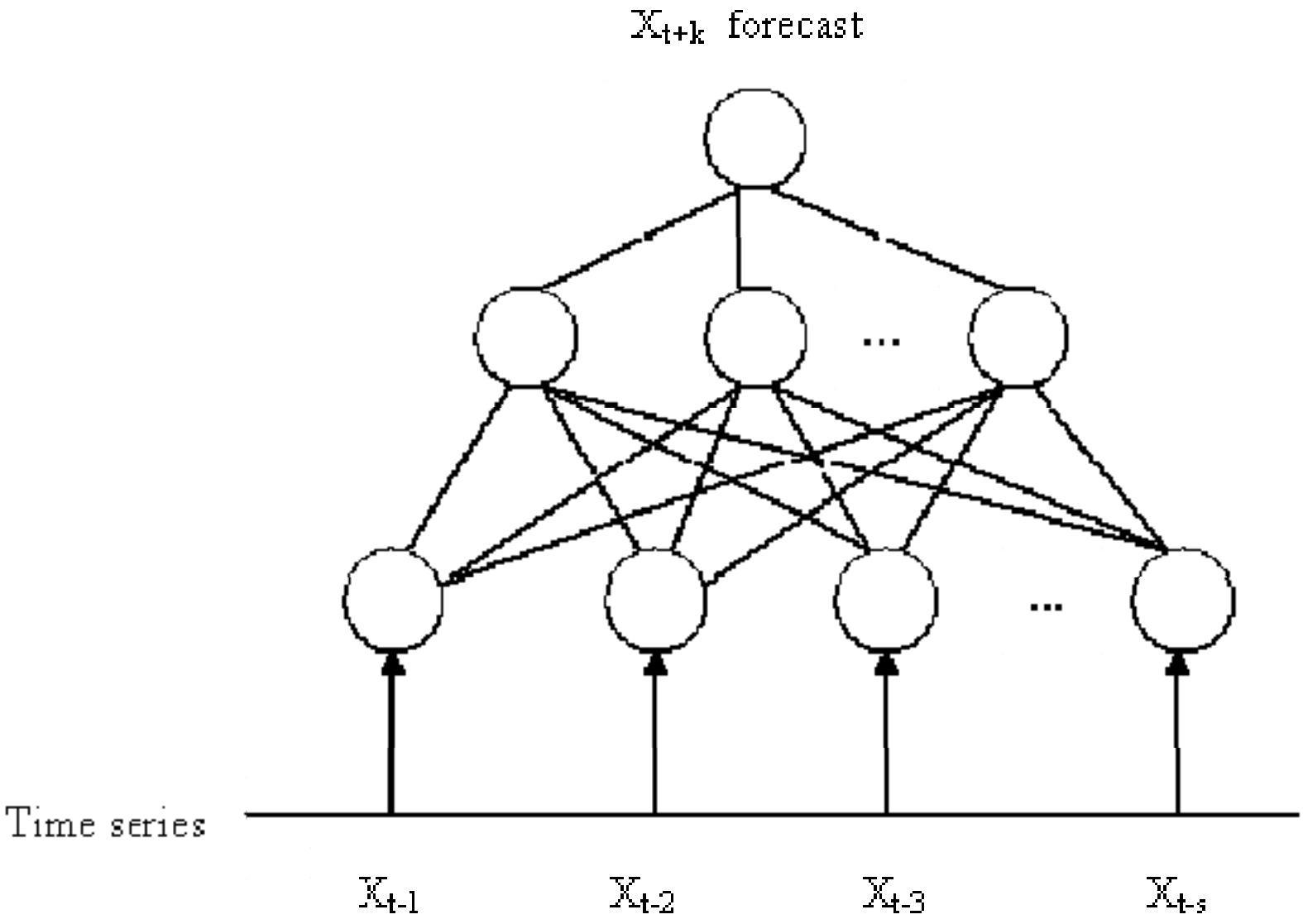
Time Series Prediction

- Time series prediction: given an existing data series, we observe or model the data series to make accurate forecasts

- Example time series
  - Financial (e.g., stocks, exchange rates)
  - Physically observed (e.g., weather, sunspots, river flow)
- Why is it important?
  - Preventing undesirable events by forecasting the event, identifying the circumstances preceding the event, and taking corrective action so the event can be avoided (e.g., inflationary economic period)
  - Forecasting undesirable, yet unavoidable, events to preemptively lessen their impact (e.g., solar maximum w/ sunspots)
  - Profiting from forecasting (e.g., financial markets)

- Why is it difficult?
  - Limited quantity of data (Observed data series sometimes too short to partition)
  - Noise (Erroneous data points, obscuring component)
  - Moving Average
  - Nonstationarity (Fundamentals change over time, nonstationary)
  - Forecasting method selection (Statistics, Artificial intelligence)
- Neural networks have been widely used as time series forecasters: most often these are feed-forward networks which employ a *sliding window* over the input sequence.
- The neural network sees the time series $X_1,\ldots,X_n$ in the form of many mappings of an input vector to an output value.

- A number of adjoining data points of the time series (the input window X_{t-s}, X_{t-s-1},…, X_t) are mapped to the interval [0,1] and used as activation levels for the input of the input layer.
- The size *s* of the input window correspondends to the number of input units of the neural network.
- In the forward path, these activation levels are propagated over one hidden layer to one output unit. The error used for the backpropagation learning algorithm is now computed by comparing the value of the output unit with the transformed value of the time series at time *t+1*. This error is propagated back to the connections between output and hidden layer and to those between hidden and output layer. After all weights have been updated accordingly, one presentation has been completed.
- Training a neural network with backpropagation learning algorithm usually requires that all representations of the input set (called one *epoch*) are presented many times. For examples, the ANN may use 60 to 138 epoches.


X_{t+k} forecast
Time series
X_{t-1}
X_{t-2}
X_{t-3}
X_{t-s}

The following parameters of the ANN are chosen for a closer inspection:

- *The number of input units*: The number of input units determines the number of periods the ANN “looks into the past” when predicting the future. The number of input units is equivalent to the size of the input window.
- *The number of hidden units*: Whereas it has been shown that one hidden layer is sufficient to approximate continuous function, the number of hidden units necessary is “not known in general”. Some examples of ANN architectures that have been used for time series prediction can be 8-8-1, 6-6-1, and 5-5-1.

- *The learning rate*: η ($0<\eta< 1$) is a scaling factor that tells the learning algorithm how strong the weights of the connections should be adjusted for a given error. A higher η can be used to speed up the learning process, but if η is too high, the algorithm will skip the optimum weights. The learning rate $\eta$is constant across presentations.
- *The momentum parameter* α ($0 < \alpha < 1$) is another number that affects the gradient descent of the weights: to prevent each connection from following every little change in the solution space immediately, the momentum term is added that keeps the direction of the previous step thus avoiding the descent into local minima. The momentum term is constant across presentations.