Module – III
Artificial Neural Networks

- Introduction
- Neural Network representation
- Appropriate problems
- Perceptions
- Backpropagation algorithm

Introduction

Neural network learning methods provide a robust approach to approximating real-valued, discrete-valued, and vector-valued target functions. For certain types of problems, such as learning to interpret complex real-world sensor data, artificial neural networks are among the most effective learning methods currently known.

Biological Motivation

The study of artificial neural networks (ANNs) has been inspired in part by the observation that biological learning systems are built of very complex webs of interconnected neurons. 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).

While ANNs are loosely motivated by biological neural systems, there are many complexities to biological neural systems that are not modeled by ANNs, and many features of the ANNs we discuss here are known to be inconsistent with biological systems. For example, we consider here ANNs whose individual units output a single constant value, whereas biological neurons output a complex time series of spikes.

Appropriate Problems for Neural Network Learning

It is appropriate for problems with the following characteristics:

Instances are represented by many attribute-value pairs. The target function to be learned is defined over instances that can be described by a vector of predefined features. These input attributes may be highly correlated or independent of one another. Input values can be any real values.

The target function output may be discrete-valued, real-valued, or a vector of several real- or discrete-valued attributes. The value of each output is some real number between 0 and 1, which in this case corresponds to the confidence in predicting the corresponding output value.

The training examples may contain errors. ANN learning methods are quite robust to noise in the training data. Long training times are acceptable. Network training algorithms typically require longer training times than, say, decision tree learning algorithms. Training times can range from a few seconds to many hours, depending on factors such as the number of weights in the network, the number of training examples considered, and the settings of various learning algorithm parameters.

Fast evaluation of the learned target function may be required. Although ANN learning times are relatively long, evaluating the learned network, in order to apply it to a subsequent instance, is typically very fast.

The ability of humans to understand the learned target function is not important. The weights learned by neural networks are often difficult for humans to interpret. Learned neural networks are less easily communicated to humans than learned rules.

Perceptrons

One type of ANN system is based on a unit called a perceptron, illustrated in Figure below. 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 and -1 otherwise. More precisely, given inputs x1 through xn, the output o(x1, . . . , xn) computed by the perceptron is

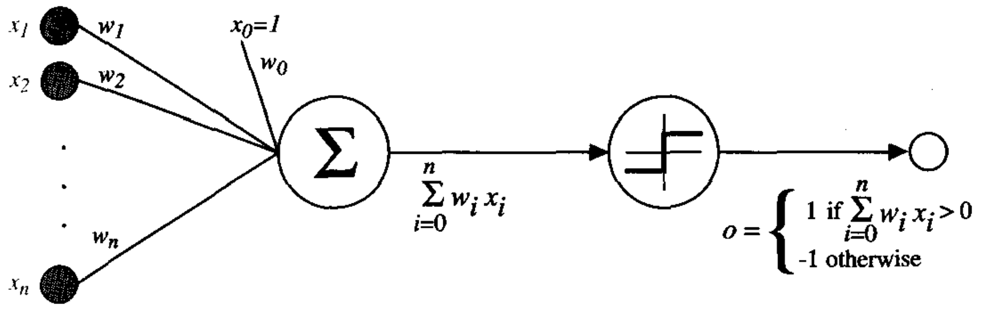

x_1
x_2
x_n
w_1
w_2
w_n
$x_0=1$
w_0
Σ
$\sum_{i=0}^{n} w_i x_i$
$o = \begin{cases} 1 \text{ if } \sum_{i=0}^{n} w_i x_i > 0 \\ -1 \text{ otherwise} \end{cases}$

FIGURE 4.2
A perceptron.

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

where each w_i is a real-valued constant, or weight, that determines the contribution of input x_i to the perceptron output. Notice the quantity ($-w_0$) is a threshold that the weighted combination of inputs $w_1x_1 + \ldots + w_nx_n$ must surpass in order for the 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 $\sum w_ix_i > 0$, or in vector form as $\vec{w} \cdot \vec{x} > 0$.
For brevity, we will sometimes write the perceptron function as

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

Where

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

The Perceptron Training Rule

Here the precise learning problem is to determine a weight vector that causes the perceptron to produce the correct +/- 1 output 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. These two algorithms are guaranteed to converge to somewhat different acceptable hypotheses, under somewhat different conditions. They are important to ANNs because they provide the basis for learning networks of many units.

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 times as 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 the target output for the current training example, o is the output generated by the perceptron, and η is a positive constant called the learning rate. 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 decay as the number of weight-tuning iterations increases.

Why should this update rule converge toward successful weight values? To get an intuitive feel, consider some specific cases. Suppose the training example is correctly classified already by the perceptron. In this case, (t - o) is zero, so that no weights are updated. Suppose the perceptron outputs a -1, when the target output is + 1. To make the perceptron output a + 1 instead of - 1 in this case, the weights must be altered to increase the value of w.x. For example, if xi > 0, then increasing wi will bring the perceptron closer to correctly classifying this example. Notice the training rule will increase w, in this case, because (t - o), η, and Xi are all positive. For example, if xi = .8, q = 0.1, t = 1, and o = - 1, then the weight update will be Awi = η(t - o)xi = O.1(1 - (-1))0.8 = 0.16. On the other hand, if t = -1 and o = 1, then weights associated with positive xi will be decreased rather than increased.

Gradient Descent and 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 separable. A second training rule, called the delta rule, is designed to overcome this difficulty. If the training examples are not linearly separable, the delta rule converges toward a best-fit approximation to the target concept.

The key idea behind the 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, which can learn networks with many interconnected units.

In order to derive a weight learning rule for linear units, let us begin by specifying a measure for the training error of a hypothesis (weight vector), relative to the training examples.

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

where D is the set of training examples, td is the target output for training example d, and od is the output of the linear unit for training example d. By this definition, E(w) is simply half the squared difference between the target output td and the hear unit output od, summed over all training examples.

Visualizing the Hypothesis Space

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 process continues until the global minimum error is reached.

Derivation of the Gradient Descent Rule

How can we calculate the direction of 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 w. This vector derivative is called the gradient of E with respect to vector w, written as $\nabla E(\vec{w})$

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

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

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

Where

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

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

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

We need an efficient way of calculating the gradient at each step. The vector of $\frac{\partial E}{\partial w_i}$ derivatives that form the gradient can be obtained by differentiating E as

$$\begin{aligned}
\frac{\partial E}{\partial w_i} &= \frac{\partial}{\partial w_i}\frac{1}{2}\sum_{d\in D}(t_d - o_d)^2 \\
&= \frac{1}{2}\sum_{d\in D}\frac{\partial}{\partial w_i}(t_d - o_d)^2 \\
&= \frac{1}{2}\sum_{d\in D}2(t_d - o_d)\frac{\partial}{\partial w_i}(t_d - o_d) \\
&= \sum_{d\in D}(t_d - o_d)\frac{\partial}{\partial w_i}(t_d - \vec{w}\cdot\vec{x_d}) \\
\frac{\partial E}{\partial w_i} &= \sum_{d\in D}(t_d - o_d)(-x_{id})
\end{aligned}$$

where xid denotes the single input component xi for training example d. We now have an equation that gives in terms of the linear unit inputs xid, outputs Od, and target values td associated with the training examples. Substituting the above equation yields the weight update rule for gradient descent

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

To summarize, 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, then compute Δw_i for each weight according to the above equation. Update each weight wi by adding , Δw_i then repeat this process. This algorithm is given below.

```
GRADIENT-DESCENT(training_examples, η)
    Each training example is a pair of the form ⟨$\vec{x}$, t⟩, 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 $\Delta w_i$ to zero.
    • For each ⟨$\vec{x}$, t⟩ 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
                $\Delta w_i \leftarrow \Delta w_i + \eta(t - o)x_i$          (T4.1)
    • For each linear unit weight $w_i$, Do
                $w_i \leftarrow w_i + \Delta w_i$          (T4.2)
```

Stochastic Approximation to Gradient Descent

Gradient descent is an important general paradigm for learning. It is a strategy for searching through a large or infinite hypothesis space that can be applied whenever (1) the hypothesis space contains continuously parameterized hypotheses (e.g., the weights in a linear unit), and (2) the error can be differentiated with respect to these hypothesis parameters. The key practical difficulties in applying gradient descent are (1) converging to a local minimum can sometimes be quite slow (i.e., it can require many thousands of gradient descent steps), and (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 common variation on gradient descent intended to alleviate these difficulties is called incremental gradient descent, or alternatively stochastic gradient descent. Whereas the gradient descent training rule presented in the above equation computes weight updates after summing over all the training examples in D, the idea behind stochastic gradient descent is to approximate this gradient descent search by updating weights incrementally, following the calculation of the error for each individual example. The modified training rule is like the training rule given by above equation except that as we iterate through each training example we update the weight according to $\Delta w_i = \eta(t - o)\ x_i$
where t, o, and xi are the target value, unit output, and ith input for the training example in question. To modify the gradient descent algorithm given above to implement this stochastic approximation, Equation (T4.2) is simply deleted and Equation (T4.1) replaced by
$w_i \leftarrow w_i + \eta(t - o)\ x_i$

The key differences between standard gradient descent and stochastic gradient descent are:

- In standard gradient descent, the error is summed over all examples before updating weights, whereas in stochastic gradient descent weights are updated upon examining each training example.
- 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.

Multilayer networks and Backpropagation Algorithm

What type of unit shall we use as the basis for constructing multilayer networks?

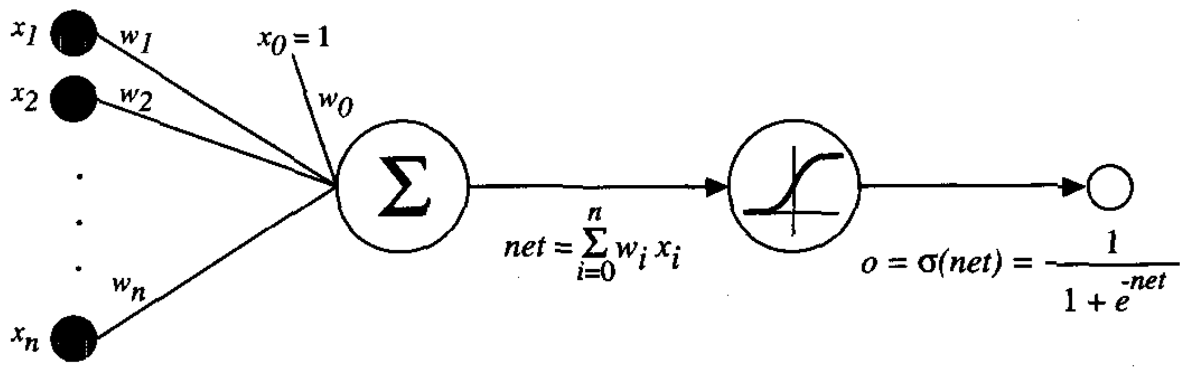

x_1
x_2
x_n
w_1
w_2
w_n
$x_0 = 1$
w_0
Σ
$net = \sum_{i=0}^{n} w_i x_i$
$o = \sigma(net) = \frac{1}{1 + e^{-net}}$

FIGURE 4.6
The sigmoid threshold unit.

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 the sigmoid unit, however, the threshold output is a continuous function of its input. More precisely, the sigmoid unit computes its output *o* as

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

Where

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

σ is often called the sigmoid function or, alternatively, the logistic function. Its output ranges between 0 and 1. Because it maps a very large input domain to a small range of outputs, it is often referred to as the *squashing function* of the unit. The sigmoid function has the useful property that its derivative is easily expressed in terms of its output

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

The Backpropagation Algorithm

The BACKPROPAGATION 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 the target values for these outputs.

Because we are considering networks with multiple output units rather than single units as before, we redefine E to sum the errors over all of the network output units

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

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 kth output unit and training example d.

```
BACKPROPAGATION($training\_examples, \eta, n_{in}, n_{out}, n_{hidden}$)
    Each training example is a pair of the form $\langle \vec{x}, \vec{t} \rangle$, where $\vec{x}$ is the vector of network input
    values, and $\vec{t}$ is the vector of target network output values.
    $\eta$ is the learning rate (e.g., .05). $n_{in}$ is the number of network inputs, $n_{hidden}$ the number of
    units in the hidden layer, and $n_{out}$ the number of output units.
    The input from unit i into unit j is denoted $x_{ji}$, and the weight from unit i to unit j is denoted
    $w_{ji}$.
• Create a feed-forward network with $n_{in}$ inputs, $n_{hidden}$ hidden units, and $n_{out}$ output units.
• Initialize all network weights to small random numbers (e.g., between $-.05$ and $.05$).
• Until the termination condition is met, Do
    • For each $\langle \vec{x}, \vec{t} \rangle$ in $training\_examples$, Do
        Propagate the input forward through the network:
        1. Input the instance $\vec{x}$ 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 network output unit $k$, calculate its error term $\delta_k$
                $\delta_k \leftarrow o_k(1 - o_k)(t_k - o_k)$                         (T4.3)
        3. For each hidden unit $h$, calculate its error term $\delta_h$
                $\delta_h \leftarrow o_h(1 - o_h) \sum_{k \in outputs} w_{kh}\delta_k$                 (T4.4)
        4. Update each network weight $w_{ji}$
                $w_{ji} \leftarrow w_{ji} + \Delta w_{ji}$
           where
                $\Delta w_{ji} = \eta\, \delta_j\, x_{ji}$                             (T4.5)
```

The BACKPROPAGATION Algorithm is presented above. The algorithm as described here applies to layered feedforward networks containing two layers of sigmoid units, with units at each layer connected to all units from the preceding layer. This is the incremental, or stochastic, gradient descent version of BACKPROPAGATION. The notation used here is the same along with the following extensions:

- An index (e.g., an integer) is assigned to each node in the network, where a "node" is either an input to the network or the output of some unit in the network.
- x_{ji} denotes the input from node i to unit j, and w_{ji} 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. As we shall see later, $\delta_n = -\frac{\partial E}{\partial net_n}$.

Notice the algorithm above begins by constructing a network with the desired number of hidden and output units and initializing all network weights to small random values. Given this fixed network structure, the main loop of the algorithm then repeatedly iterates over the training examples. For each training example, it applies the network to the example, calculates the error of the network output for this example, computes the gradient with respect to the error on this example, then updates all weights in the network. This gradient descent step is iterated (often thousands of times, using the same training examples multiple times) until the network performs acceptably well.

[Refer to Numerical problem done in class]

Derivation of Backpropagation Rule

[Very important – Refer to notes given in the class]

Remarks on the Backpropagation Algorithm

1. Convergence and Local Minima
2. Representational Power of Feedforward Networks
3. Hypothesis Space Search and Inductive Bias
4. Hidden Layer Representations
5. Generalization, Overfitting and Stopping Criterion

Convergence and Local Minima

The BACKPROPAGATION Algorithm implements a gradient descent search through the space of possible network weights, iteratively reducing the error E between the training example target values and the network outputs. Because the error surface for multilayer networks may contain many different local minima, gradient descent can become trapped in any of these. As a result, BACKPROPAGATION over multilayer networks is only guaranteed to converge toward some local minimum in E and not necessarily to the global minimum error.

Common heuristics to attempt to alleviate the problem of local minima include:

1. Add a momentum term to the weight-update rule as described in Equation below:

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

   Where Momentum can sometimes carry the gradient descent procedure through narrow local minima.
2. Use stochastic gradient descent rather than true gradient descent.
3. Train multiple networks using the same data, but initializing each network with different random weights. If the different training efforts lead to different local minima, then the network with the best performance over a separate validation data set can be selected.

Representational Power of FeedForward Networks

What set of functions can be represented by feedforward networks? The answer depends on the width and depth of the networks.

1. **Boolean functions.** Every boolean function can be represented exactly by some network with two layers of units, although the number of hidden units required grows exponentially. For each possible input vector, create a distinct hidden unit and set its

weights so that it activates if and only if this specific vector is input to the network. This produces a hidden layer that will always have exactly one unit active. Now implement the output unit as an OR gate that activates just for the desired input patterns.

2. **Continuous functions.** Every bounded continuous function can be approximated with arbitrarily small error (under a finite norm) by a network with two layers of units. The theorem in this case applies to networks that use sigmoid units at the hidden layer and (unthresholded) linear units at the output layer. The number of hidden units required depends on the function to be approximated.
3. **Arbitrary functions**. Any function can be approximated to arbitrary accuracy by a network with three layers of units. Again, the output layer uses linear units, the two hidden layers use sigmoid units, and the number of units required at each layer is not known in general.

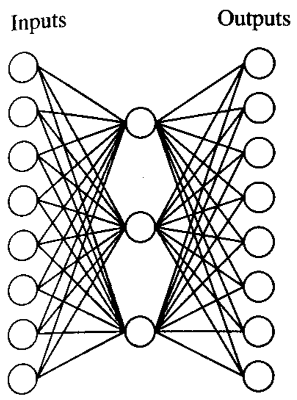
Hypotheses Space Search and Inductive Bias

It is interesting to compare the hypothesis space search of BACKPROPAGATION to the search performed by other learning algorithms. For BACKPROPAGATION, every possible assignment of network weights represents a syntactically distinct hypothesis that in principle can be considered by the learner. In other words, the hypothesis space is the n-dimensional Euclidean space of the n network weights. Notice this hypothesis space is continuous, in contrast to the hypothesis spaces of decision tree learning and other methods based on discrete representations. The fact that it is continuous, together with the fact that E is differentiable with respect to the continuous parameters of the hypothesis, results in a well-defined error gradient that provides a very useful structure for organizing the search for the best hypothesis.

What is the inductive bias by which BACKPROPAGATION generalizes beyond the observed data? Given two positive training examples with no negative examples between them, BACKPROPAGATION will tend to label points in between as positive examples as well.

Hidden Layer Representations

Consider, for example, the network shown in Figure below. Here, the eight network inputs are connected to three hidden units, which are in turn connected to the eight output units. Because of this structure, the three hidden units will be forced to re-represent the eight input values in some way that captures their relevant features, so that this hidden layer representation can be used by the output units to compute the correct target values.



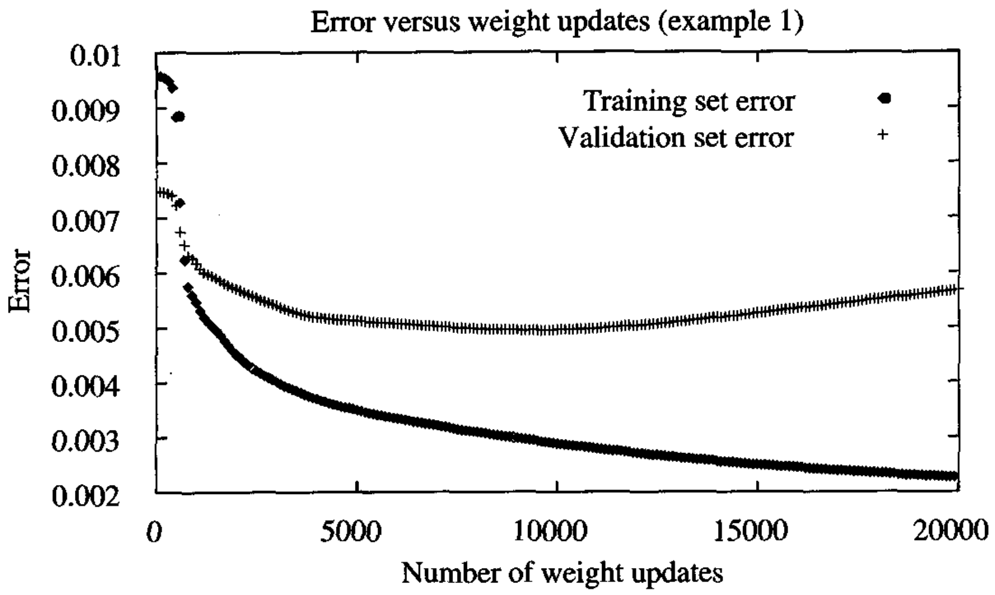
Input		Hidden Values				Output
10000000	→	.89	.04	.08	→	10000000
01000000	→	.15	.99	.99	→	01000000
00100000	→	.01	.97	.27	→	00100000
00010000	→	.99	.97	.71	→	00010000
00001000	→	.03	.05	.02	→	00001000
00000100	→	.01	.11	.88	→	00000100
00000010	→	.80	.01	.98	→	00000010
00000001	→	.60	.94	.01	→	00000001

The network must learn to reproduce the eight inputs at the corresponding eight output units. Although this is a simple function, the network in this case is constrained to use only three hidden units. Therefore, the essential information from all eight input units must be captured by the three learned hidden units.

When BACKPROPAGATION applied to this task, using each of the eight possible vectors as training examples, it successfully learns the target function. What hidden layer representation is created by the gradient descent BACKPROPAGATION algorithm? By examining the hidden unit values generated by the learned network for each of the eight possible input vectors, it is easy to see that the learned encoding is similar to the familiar standard binary encoding of eight values using three bits (e.g., 000, 001, 010, . . . , 111). The exact values of the hidden units for one typical run of BACKPROPAGATION are shown in Figure above. This ability of multilayer networks to automatically discover useful representations at the hidden layers is a key feature of ANN learning.

Generalization, Overfitting and Stopping Criterion

What is an appropriate condition for terminating the weight update loop? One obvious choice is to continue training until the error E on the training examples falls below some predetermined threshold. In fact, this is a poor strategy.



Consider the plot in the above figure. The lower of the two lines shows the monotonically decreasing error E over the training set, as the number of gradient descent iterations grows. The upper line shows the error E measured over a different validation set of examples, distinct from the training examples. This line measures the *generalization accuracy* of the network-the accuracy with which it fits examples beyond the training data.

Notice the generalization accuracy measured over the validation examples first decreases, then increases, even as the error over the training examples continues to decrease. This is because of Overfitting.

Several techniques are available to address the overfitting problem for BACKPROPAGATION learning. One approach, known as **weight decay**, is to decrease each weight by some small factor during each iteration. This is equivalent to modifying the definition of E to include a penalty term corresponding to the total magnitude of the network weights. The motivation for this approach is to keep weight values small, to bias learning against complex decision surfaces.

One of the most successful methods for overcoming the overfitting problem is to simply provide a set of validation data to the algorithm in addition to the training data. The algorithm monitors the error with respect to this validation set, while using the training set to drive the gradient descent search.

How many weight-tuning iterations should the algorithm perform? Clearly, it should use the number of iterations that produces the lowest error over the validation set, since this is the best indicator of network performance over unseen examples.