Global Academy of Technology, Bengaluru-98

Department: CSE

Academic year 2021-22

Machine Learning

Module-3

Artificial neural networks

Define Artificial neural networks.

A neural network is a massively parallel distributed processor made up of simple processing units that has a natural propensity for storing experiential knowledge and making it available for use. It resembles the brain in two respects:

1. Knowledge is acquired by the network from its environment through a learning process.

2. Interneuron connection strengths, known as synaptic weights, are used to store the acquired knowledge. [SH]

Artificial neural networks (ANNs) provide a general, practical method for learning real-valued, discrete-valued, and vector-valued functions from examples.[TM]

The study of artificial neural networks (ANNs) has been inspired in part by the observation that biological learning systems. 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).

The human brain, is estimated to contain a densely interconnected network of approximately 10^{11} neurons, each connected, on average, to $l0^4$ others. Neuron activity is typically excited or inhibited through connections to other neurons. The fastest neuron switching times are known to be on the order of $l0^{-3}$ seconds--quite slow compared to computer switching speeds of 10^{-10} seconds.

Yet humans are able to make surprisingly complex decisions, surprisingly quickly. For example, it requires approximately 10^{-1} seconds to visually recognize your mother.

Applications of Artificial Neural Network (Back propagation)

BACKPROPAGATION network use gradient descent to tune network parameters to best fit a training set of input-output pairs and is applied to problems such as

1. Interpreting visual scenes
2. Speech recognition
3. Learning robot control strategies.

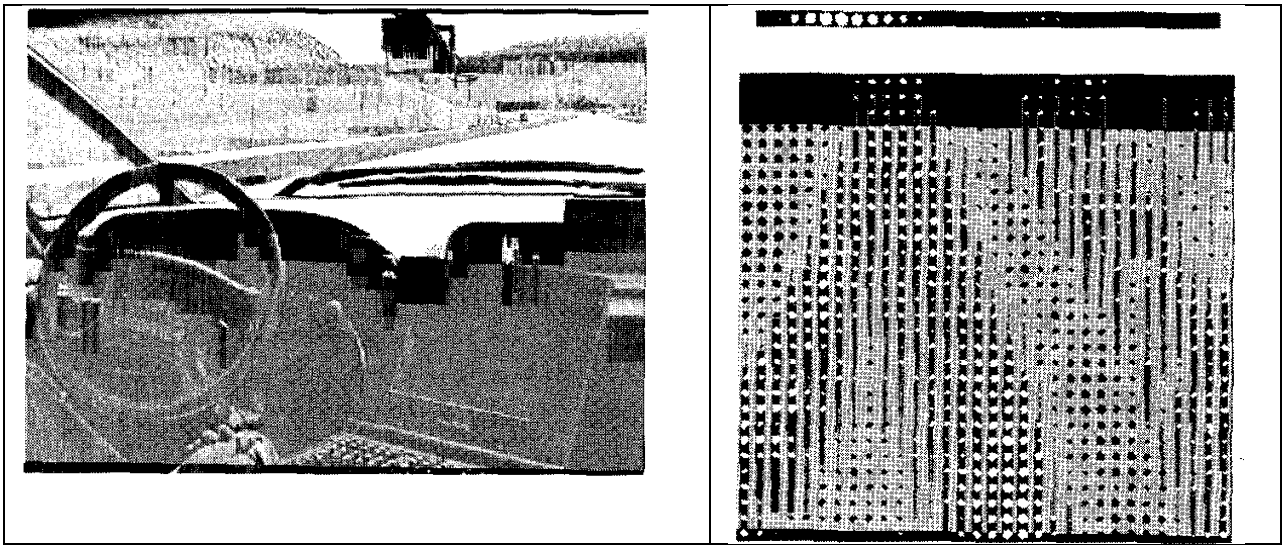
BACKPROPAGATION algorithm is surprisingly successful in many practical problems such as

1. Learning to recognize handwritten characters
2. Learning to recognize spoken words
3. Learning to recognize faces.

Some problems for neural network learning

ANN learning is well-suited to problems in which the training data corresponds to noisy, complex sensor data, such as inputs from cameras and microphones. Neural nets can be used to answer the following:

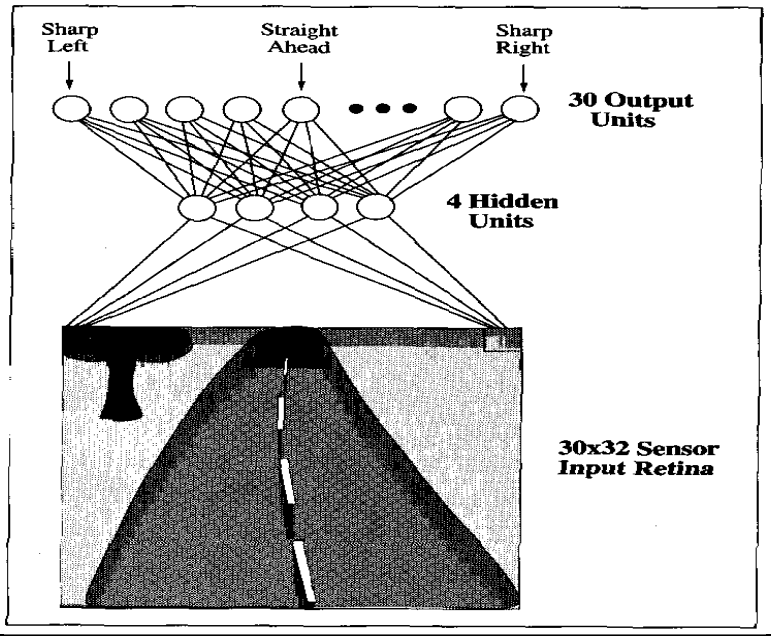
1. Pattern recognition: Does that image contain a face?
2. Classification problems: Is this cell defective?
3. Prediction: Given these symptoms, the patient has disease X
4. Forecasting: predicting behavior of stock market
5. Handwriting: is character recognized?
6. Optimization: Find the shortest path for the TSP.



Roots of work on NNs are in:

1. **Neurobiological studies:**
    a. How do nerves behave when stimulated by different magnitudes of electric current?
    b. Is there a minimal threshold needed for nerves to be activated?
    c. How do different nerve cells communicate among each other?
    d. **Psychological studies**:
    e. How do animals learn, forget, recognize and perform various types of tasks?

2. **Psycho-physical:** experiments help to understand how individual neurons and groups of neurons work.
3. **McCulloch and Pitts:** introduced the first mathematical model of single neuron, widely applied in subsequent work.

The figure on the shows Neural network learning to steer an autonomous vehicle. The ALVINN system uses **BACKPROPAGATION** to learn to steer an autonomous vehicle (photo at top) driving at speeds up to **70** miles per hour.	The figure on the shows weight values for one of the hidden units in this network. The **30** x 32 weights into the hidden unit are displayed in the large matrix, with white blocks indicating positive and black indicating negative weights.
	The diagram on the left shows how the image of a forward-mounted camera is mapped to **960** neural network inputs, which are fed forward to 4 hidden units, connected to **30** output units. Network outputs encode the commanded steering direction.

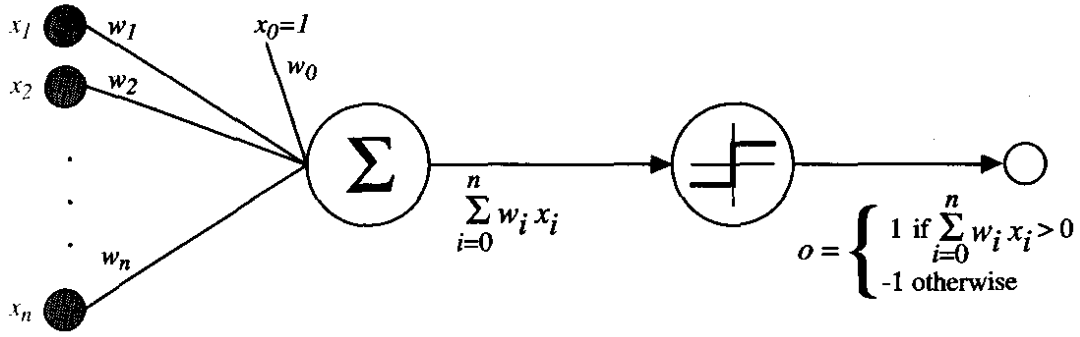
Explain the characteristics for problems that are commonly used by BACKPROPAGATION algorithm.

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

1. Instances are represented by many attribute-value pairs.
2. The target function output may be discrete-valued, real-valued, or a vector of several real- or discrete-valued attributes.
3. The training examples may contain errors.
4. Long training times are acceptable.
5. Fast evaluation of the learned target function may be required.
6. The ability of humans to understand the learned target function is not important.

With neat diagram explain the working of Perceptron ANN system.

One type of ANN system is based on a unit called a perceptron, illustrated 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 ***xl*** through ***x,,*** the output ***o(x1, . . .*** , ***x,)*** computed by the perceptron is

$$o(x_1, \ldots, x_n) = \begin{cases} 1 \text{ if } w_0 + w_1 x_1 + w_2 x_2 + \cdots + w_n x_n > 0 \\ -1 \text{ otherwise} \end{cases}$$

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.

Representational 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 illustrated in Figure 4.3.

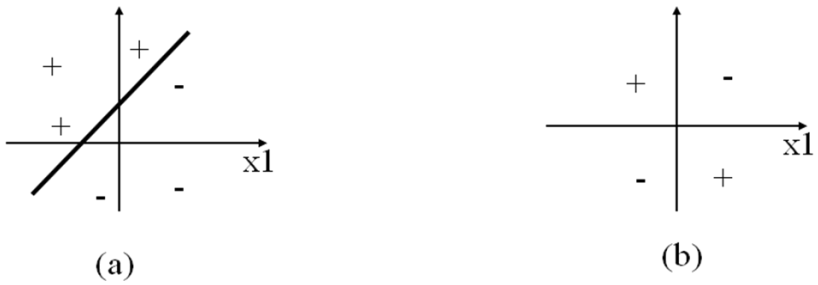


Fig: The decision surface represented by a two-input perceptron. *(a)* A set of training examples and the decision surface of a perceptron that classifies them correctly. *(b)* A set of training examples that is not linearly separable (i.e., that cannot be correctly classified by any straight line). x_1 and x_2 are the Perceptron inputs. Positive examples are indicated by "+", negative by "-".

The equation for this decision hyperplane is ***vector(W.X)***= 0. Of course, some sets of positive and negative examples cannot be separated by any hyperplane. Those that can be separated are

called linearly separable sets of examples. A single perceptron can be used to represent many boolean functions. For example, if we assume boolean values of 1 (true) and -1 (false), then one way to use a two-input perceptron to implement the AND function is to set the weights w_0 = ***-3,*** and $w_1 = w_2 = .5$.

Write about the Perceptron Training Rule and explain gradient descent.

The Perceptron Training Rule

Several algorithms are known to solve this learning problem. Here we consider two: the perceptron rule and the delta. 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

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

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.

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 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 BPN algorithm, which can learn networks with many interconnected units. It is also important because gradient descent can serve as the basis for learning algorithms that must search through hypothesis spaces containing many different types of continuously parameterized hypotheses.

$$o(\vec{x}) = \vec{w} \cdot \vec{x} \quad \text{---(1)}$$

Thus, a linear unit corresponds to the first stage of a perceptron, without the threshold. 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. Although there are many ways to define this error, one common measure that will turn out to be especially convenient is

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

where D is the set of training examples, t_d is the target output for training example *d*, and o_d is the output of the linear unit for training example d. By this definition, $E(\vec{w})$ is simply half the squared difference between the target output t_d and the hear unit output o_d, summed over all training examples. Here we characterize ***E*** as a function of $\vec{w}$ **,** because the linear unit output ***o*** depends on this weight vector. Of course ***E*** also depends on the particular **set** of training examples, but we assume these are fixed during training, so we do not bother to write E as an explicit function of these.

Show that the DERIVATION OF THE GRADIENT DESCENT RULE is

The visualization of the entire hypothesis space can be done with possible weight vectors and their associated E values, as illustrated in Figure below. Here the axes w_0 *and* w_1 represent possible values for the two weights of a simple linear unit. The w_0, w_1 plane therefore 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 thus summarizes the desirability of every weight vector in 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 depicted in Figure**.** This process continues until the global minimum error is reached.

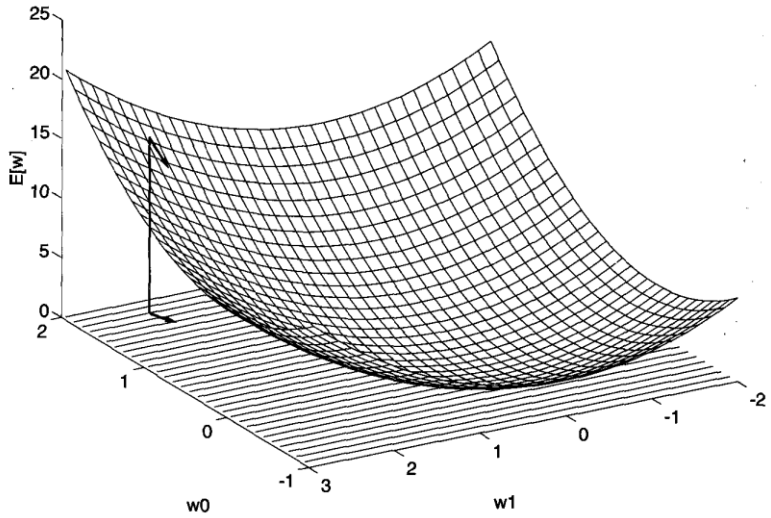


Fig: Error of different hypotheses. For a linear unit with two weights, the hypothesis space H is the w_0, w_1 plane. The vertical axis indicates the error of the corresponding weight vector

hypothesis, relative to a fixed set of training examples. The arrow shows the negated gradient at one particular point, indicating the direction in the w_0, w_1 plane producing steepest descent along the error surface.

(To calculate the direction of steepest descent along the error surface)

Gradient

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

Training rule:

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

i.e.,

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

$$\begin{aligned}
\frac{\partial E}{\partial w_i} &= \frac{\partial}{\partial w_i} \frac{1}{2} \sum_d (t_d - o_d)^2 \\
&= \frac{1}{2} \sum_d \frac{\partial}{\partial w_i} (t_d - o_d)^2 \\
&= \frac{1}{2} \sum_d 2(t_d - o_d) \frac{\partial}{\partial w_i} (t_d - o_d) \\
&= \sum_d (t_d - o_d) \frac{\partial}{\partial w_i} (t_d - \vec{w} \cdot \vec{x_d}) \\
\frac{\partial E}{\partial w_i} &= \sum_d (t_d - o_d)(-x_{i,d}) \quad \text{--(6)}
\end{aligned}$$

Substituting Equation (2) into Equation (1) yields the weight update rule for gradient descent

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

Gradient descent 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)

(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)

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

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

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

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.
- In cases where there are multiple local minima with respect to $E(\vec{w})$, stochastic gradient descent can sometimes avoid falling into these local minima because it uses the various $\nabla E_d(\vec{w})$ rather than $\nabla E(\vec{w})$ to guide its search.

Algorithm for Incremental (Stochastic) Gradient Descent

Batch mode Gradient Descent:
Do until satisfied

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

Incremental mode Gradient Descent:
Do until satisfied

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

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

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

Incremental Gradient Descent can approximate *Batch Gradient Descent* arbitrarily closely if η made small enough

MULTILAYER NETWORKS AND THE BACKPROPAGATION ALGORITHM

The kind of multilayer networks learned by the **BACKPROPACATION** algorithm are capable of expressing a rich variety of nonlinear decision surfaces. For example, a typical multilayer network and decision surface is depicted in Figure below**.** Here the speech recognition task involves distinguishing among 10 possible vowels, all spoken in the context of "h-d" (i.e., "hid," "had," "head," "hood," etc.). The input speech signal is represented by two numerical parameters obtained from a spectral analysis of the sound, allowing us to easily visualize the decision surface over the two-dimensional instance space. As shown in the figure, it is possible for the multilayer network to represent highly nonlinear decision surfaces that are much more expressive than the linear decision surfaces of single units shown earlier in Figure**.**

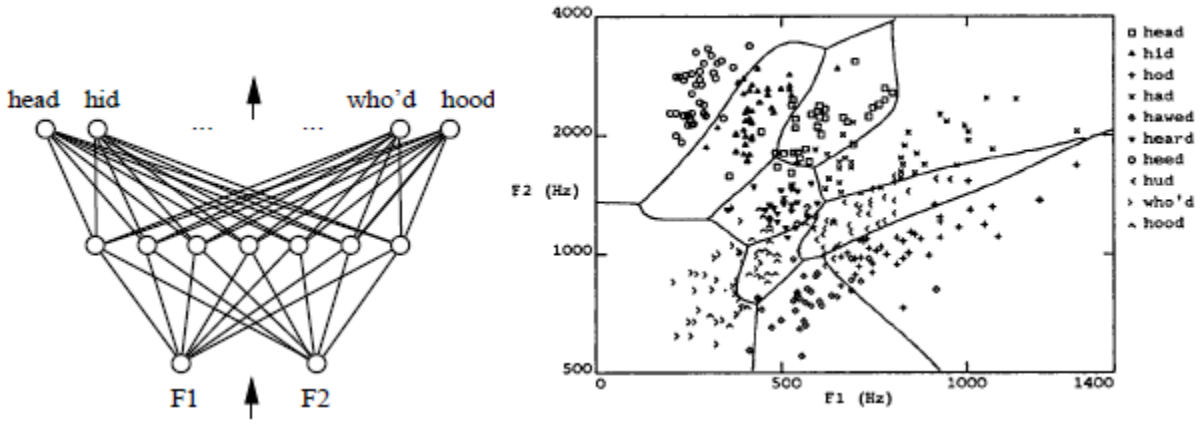
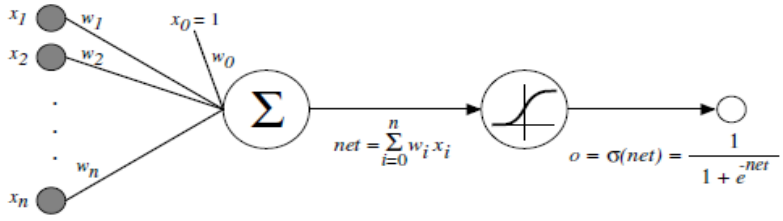


FIGURE 4.5
Decision regions of a multilayer feedforward network. The network shown here was trained to recognize 1 of ***10*** vowel sounds occurring in the context "hd" (e.g., "had," "hid"). The network input consists of two parameters, F1 and F2, obtained from a spectral analysis of the sound. The 10 network outputs correspond to the 10 possible vowel sounds. The network prediction is the output whose value is highest. The plot on the right illustrates the highly nonlinear decision surface represented by the learned network. Points shown on the plot are test examples distinct from the examples used to train the network. (Reprinted by permission from Haung and Lippmann **(1988).)**

Explain the unit that is the basis constructing multilayer networks? (sigmoid unit)
Multiple layers of cascaded linear units produce only linear functions, and we prefer networks capable of representing highly nonlinear functions. The perceptron unit is another possible choice, but its discontinuous threshold makes it undifferentiable and hence unsuitable for gradient descent. What we need is a unit whose output is a nonlinear function of its inputs, but whose output is also a differentiable function of its inputs. One solution is the sigmoid unit-a unit very much like a perceptron, but based on a smoothed, differentiable threshold function.The sigmoid unit is illustrated in Figure below. 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.



$\sigma(x)$ is the sigmoid function

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

Nice property: $\frac{d\sigma(x)}{dx} = \sigma(x)(1 - \sigma(x))$ --(12)

We can derive gradient decent rules to train

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

Derive an Error Gradient equation for a Sigmoid Unit

$$\begin{aligned}\frac{\partial E}{\partial w_i} &= \frac{\partial}{\partial w_i}\frac{1}{2}\sum_{d\in D}(t_d - o_d)^2 \\ &= \frac{1}{2}\sum_d \frac{\partial}{\partial w_i}(t_d - o_d)^2 \\ &= \frac{1}{2}\sum_d 2(t_d - o_d)\frac{\partial}{\partial w_i}(t_d - o_d) \\ &= \sum_d (t_d - o_d)\left(-\frac{\partial o_d}{\partial w_i}\right) \\ &= -\sum_d (t_d - o_d)\frac{\partial o_d}{\partial net_d}\frac{\partial net_d}{\partial w_i}\end{aligned}$$

But we know:

$$\frac{\partial o_d}{\partial net_d} = \frac{\partial \sigma(net_d)}{\partial net_d} = o_d(1 - o_d)$$

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

So:

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

The BACKPROPAGATION Algorithm (BPA)

The BPA 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 begin by redefining E to sum the errors over all of the network output units

$$E(\vec{w}) \equiv \frac{1}{2}\sum_{d\in D}\sum_{k\in outputs}(t_{kd} - o_{kd})^2 \quad \text{--(13)}$$

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 learning problem faced by BACKPROPAGATION search a large hypothesis space defined by all possible weight values for all the units in the network. The error in that diagram is replaced by our new definition of E, and the other dimensions of the space correspond now to all of the weights associated with all of the units in the network. As in the case of training a single unit, gradient descent can be used to attempt to find a hypothesis to minimize E.

One major difference in the case of multilayer networks is that the error surface can have multiple local minima, in contrast to the single-minimum parabolic error surface. Unfortunately, this means that gradient descent is guaranteed only to converge toward some local minimum, and not necessarily the global minimum error. Despite this obstacle, in practice BACKPROPAGATION been found to produce excellent results in many real-world applications.

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

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

- Gradient descent over entire *network* weight vector
- Easily generalized to arbitrary directed graphs
- Will find a local, not necessarily global error minimum
  - In practice, often works well (can run multiple times)
- Often include weight *momentum* α

$$\Delta w_{i,j}(n) = \eta \delta_j x_{i,j} + \alpha \Delta w_{i,j}(n-1) \quad \text{--(18)}$$

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

Derive BACKPROPAGATION weight-tuning Rule

The specific problem we address here is deriving the stochastic gradient descent rule implemented by the previous algorithm. The stochastic gradient descent involves iterating through the training examples one at a time, for each training example d descending the gradient of the error E_d with respect to this single example. In other words, for each training example d every weight w_{ji} is updated by adding to it Δw_{ji}

$$\Delta w_{ji} = -\eta \frac{\partial E_d}{\partial w_{ji}} \quad \text{--- (21)}$$

where E_d is the error on training example d, summed over all output units in the network

$$E_d(\vec{w}) \equiv \frac{1}{2} \sum_{k \in outputs} (t_k - o_k)^2$$

Here ***outputs*** is the set of output units in the network, t_k is the target value of unit k for training example d, and o_k is the output of unit k given training example d. The derivation of the stochastic gradient descent rule is conceptually straightforward, but requires keeping track of a number of subscripts and variables, adding a subscript ***j*** to denote to the jth unit of the network as follows:

- x_{ji} = the ith input to unit j
- w_{ji} = the weight associated with the ith input to unit j
- $net_j = \sum_i w_{ji} x_{ji}$ (the weighted sum of inputs for unit j)
- o_j = the output computed by unit j
- t_j = the target output for unit j
- σ = the sigmoid function
- $outputs$ = the set of units in the final layer of the network
- $Downstream(j)$ = the set of units whose immediate inputs include the output of unit j

We now derive an expression for $\frac{\partial E_d}{\partial w_{ji}}$ in order to implement the stochastic gradient descent rule seen in the previous Equation**.** To begin, notice that weight w_{ji} can influence the rest of the network only through net_j**.** Therefore, we can use the chain rule to write

$$\frac{\partial E_d}{\partial w_{ji}} = \frac{\partial E_d}{\partial net_j} \frac{\partial net_j}{\partial w_{ji}}$$

$$= \frac{\partial E_d}{\partial net_j} x_{ji} \quad \text{--- (22)}$$

Given previous Equation**,** our remaining task is to derive a convenient expression for $\frac{\partial E_d}{\partial net_j}$**.** We consider two cases in turn: the case where unit j is an output unit for the network, and the case where j is an internal unit.

Case 1: Training rule for Output Unit Weights. Just **as** w_{ji} can influence the rest of the network only through net_j**,** net_j**,** can influence the network only through o_j**.** Therefore, we can invoke the chain rule again to write

$$\frac{\partial E_d}{\partial net_j} = \frac{\partial E_d}{\partial o_j}\frac{\partial o_j}{\partial net_j} \quad \textbf{--- (23)}$$

To begin, consider just the first term in Equation ***(4.23)***

$$\frac{\partial E_d}{\partial o_j} = \frac{\partial}{\partial o_j}\frac{1}{2}\sum_{k \in outputs}(t_k - o_k)^2$$

The derivatives $\frac{\partial}{\partial o_j}(t_k - o_k)^2$ will be zero for all output units k except when k = j. We therefore drop the summation over output units and simply set k = j.

Next consider the second term in Equation **(4.23).** Since $o_j = \sigma(net_j)$, the

$$\begin{aligned}\frac{\partial E_d}{\partial o_j} &= \frac{\partial}{\partial o_j}\frac{1}{2}(t_j - o_j)^2 \\ &= \frac{1}{2}2(t_j - o_j)\frac{\partial(t_j - o_j)}{\partial o_j} \\ &= -(t_j - o_j)\end{aligned} \quad \textbf{-- (24)}$$

derivative $\frac{\partial o_j}{\partial net_j}$ is just the derivative of the sigmoid function, which we have already noted is equal to $\sigma(net_j)(1 - \sigma(net_j))$. Therefore,

$$\begin{aligned}\frac{\partial o_j}{\partial net_j} &= \frac{\partial \sigma(net_j)}{\partial net_j} \\ &= o_j(1 - o_j)\end{aligned} \quad \textbf{- (25)}$$

Substituting expressions ***(4.24)*** and ***(4.25)*** into ***(4.23),*** we obtain

$$\frac{\partial E_d}{\partial net_j} = -(t_j - o_j)\ o_j(1 - o_j) \quad \textbf{- (26)}$$

and combining this with Equations (4.21) and (4.22), we have the stochastic gradient descent rule for output units

$$\Delta w_{ji} = -\eta\frac{\partial E_d}{\partial w_{ji}} = \eta\ (t_j - o_j)\ o_j(1 - o_j)x_{ji} \quad \textbf{--(27)}$$

Note this training rule is exactly the weight update rule implemented by Equations (T4.3) and (T4.5) in the algorithm of Table 4.2. Furthermore, we can see now that ***Sk*** in Equation (T4.3) is equal to the quantity $-\frac{\partial E_d}{\partial net_k}$. In the remainder of this section we will use ***Si*** to denote the quantity $-\frac{\partial E_d}{\partial net_i}$ for an arbitrary unit ***i.***

Case 2: Training Rule for Hidden Unit Weights. In the case where ***j*** is aninternal, or hidden unit in the network, the derivation of the training rule for w_{ji} must take into account the indirect ways in which w_{ji} can influence the network outputs and hence E_d. For this reason, we will find it useful to refer to the set of all units immediately downstream of unit j in the network (i.e., all units whose direct inputs include the output of unit j). We denote this set of units by ***Downstream(*** j). Notice that net_j can influence the network outputs (and therefore *Ed)* only through the units in ***Downstream(j).*** Therefore, we can write

$$\begin{aligned}\frac{\partial E_d}{\partial net_j} &= \sum_{k \in Downstream(j)} \frac{\partial E_d}{\partial net_k} \frac{\partial net_k}{\partial net_j} \\ &= \sum_{k \in Downstream(j)} -\delta_k \frac{\partial net_k}{\partial net_j} \\ &= \sum_{k \in Downstream(j)} -\delta_k \frac{\partial net_k}{\partial o_j} \frac{\partial o_j}{\partial net_j} \\ &= \sum_{k \in Downstream(j)} -\delta_k \, w_{kj} \frac{\partial o_j}{\partial net_j} \\ &= \sum_{k \in Downstream(j)} -\delta_k \, w_{kj} \, o_j(1 - o_j)\end{aligned} \quad \text{--- (28)}$$

Rearranging terms and using δ_j to denote $-\frac{\partial E_d}{\partial net_j}$, we have

$$\delta_j = o_j(1 - o_j) \sum_{k \in Downstream(j)} \delta_k \, w_{kj}$$

and

$$\Delta w_{ji} = \eta \, \delta_j \, x_{ji}$$

which is precisely the general rule from Equation (20) for updating internal unit weights in arbitrary acyclic directed graphs.

REMARKS ON THE BACKPROPAGATION ALGORITHM

Convergence and Local Minima

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 (18). Momentum can sometimes carry the gradient descent procedure through narrow local

minima (though in principle it can also carry it through narrow global minima into other local minima!).

2. Use stochastic gradient descent rather than true gradient descent. The stochastic approximation to gradient descent effectively descends a different error surface for each training example, relying on the average of these to approximate the gradient with respect to the full training set. These different error surfaces typically will have different local minima, making it less likely that the process will get stuck in any one of them.
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. Alternatively, all networks can be retained and treated as a "committee" of networks whose output is the (possibly weighted) average of the individual network outputs.

What set of functions can be represented by feedforward networks?

Of course the answer depends on the width and depth of the networks. Although much is still unknown about which function classes can be described by which types of networks, three quite general results are known:

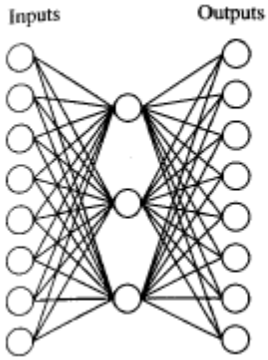
Boolean functions. Every boolean function can be represented exactly by some network with two layers of units, although the number of hidden units required grows exponentially in the worst case with the number of network inputs.

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.

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. The proof of this involves showing that any function can be approximated by a linear combination of many localized functions that have value 0 everywhere except for some small region, and then showing that two layers of sigmoid units are sufficient to produce good local approximations.

Hidden Layer Representations

One intriguing property of BACKPROPAGATION ability to discover useful intermediate representations at the hidden unit layers inside the network. Because training examples constrain only the network inputs and outputs, the weight-tuning procedure is free to set weights that define whatever hidden unit representation is most effective at minimizing the squared error E. This can lead BACKPROPAGATION to define new hidden layer features that are not explicit in the input representation, but which capture properties of the input instances that are most relevant to learning the target function.



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

Fig: Learned Hidden Layer Representation. This 8 x 3 x 8 network was trained to learn the identity function, using the eight training examples shown. After 5000 training epochs, the three hidden unit values encode the eight distinct inputs using the encoding shown on the right. Notice if the encoded values are rounded to zero or one, the result is the standard binary encoding for eight distinct values.

Consider, for example, the network shown in Figure above. 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.