UNIT-4

Multilayer Perceptron: The Perceptron, Training a Perceptron, Learning Boolean Functions, Multilayer Perceptron, MLP as a Universal Approximator, Back propagation Algorithm, Training Procedures, Dimensionality Reduction, Learning Time. [TB-1]

INTRODUCTION: -

Artificial neural network models, one of which is the perceptron, take their inspiration from the brain. There are cognitive scientists and neuroscientists whose aim is to understand the functioning of the brain and toward this aim, build models of the natural neural networks in the brain and make simulation studies.

However, in engineering, our aim is not to understand the brain perse, but to build useful machines. We are interested in artificial neural networks because we believe that they may help us build better computer systems. The brain is an information processing device that has some incredible abilities and surpasses current engineering products in many domains—for example, vision, speech recognition, and learning, to name three. These applications have evident economic utility if implemented on machines.

The human brain is quite different from a computer. Whereas a computer generally has one processor, the brain is composed of a very large (10^{11}) number of processing units, namely, neurons, operating in parallel. Though the details are not known, the processing units are believed to be much simpler and slower than a processor in a computer. What also makes the brain different, and is believed to provide its computational power, is the large connectivity. Neurons in the brain have connections, called synapses, to around 10^4 other neurons, all operating in parallel. In a computer, the processor is active and the memory is separate and passive, but it is believed that in the brain, both the processing and memory are distributed together over the network; processing is done by the neurons, and the memory is in the synapses between the neurons.

THE PERCEPTRON: -

The perceptron is the basic processing element. It has inputs that may come from the environment or may be the outputs of other perceptron. Associated with each input, $x_j \in \mathbb{R}$, $j = 1,...,d$, is a connection weight, or synaptic weight $w_j \in \mathbb{R}$, and the output, y, in the simplest case is a weighted sum of the inputs:

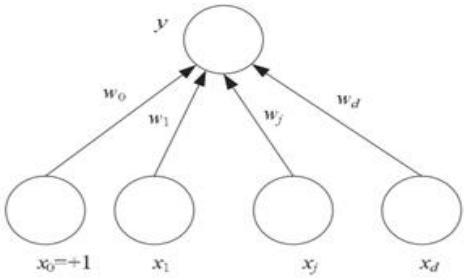


Figure 11.1 Simple perceptron. $x_j, j = 1,\ldots,d$ are the input units. x_0 is the bias unit that always has the value 1. y is the output unit. w_j is the weight of the directed connection from input x_j to the output.

$$y = \sum_{j=1}^{d} w_j x_j + w_0 \qquad (11.1)$$

w_0 is the intercept value to make the model more general; it is generally modeled as the weight coming from an extra bias unit, x_0, which is always +1. We can write the output of the perceptron as a dot product

$$y = \mathbf{w}^T \mathbf{x} \tag{11.2}$$

where $\mathbf{w} = [w_0, w_1, \ldots, w_d]^T$ and $\mathbf{x} = [1, x_1, \ldots, x_d]^T$ are *augmented* vectors to include also the bias weight and input.

During testing, with given weights, w, for input x, we compute the output y. To implement a given task, we need to learn the weights w, the parameters of the system, such that correct outputs are generated given the inputs.

When d = 1 and x is fed from the environment through an input unit, we have

$y = wx + w_0$

Which is the equation of a line with *w* as the slope and w_0 as the intercept. Thus this perceptron with one input and one output can be used to implement a linear fit. With more than one input, the line becomes a (hyper) plane, and the perceptron with more than one input can be used to implement multivariate linear fit.

The perceptron as defined in equation 11.1 defines a hyper plane and as such can be used to divide the input space into two: the half-space where it is positive and the half-space where it is negative. By using it to implement a linear discriminant function, the perceptron can separate two classes by checking the sign of the output. If we define s(·) as the threshold function

$$s(a) = \begin{cases} 1 & \text{if } a > 0 \\ 0 & \text{otherwise} \end{cases} \tag{11.3}$$

then we can

$$\text{choose} \begin{cases} C_1 & \text{if } s(\mathbf{w}^T \mathbf{x}) > 0 \\ C_2 & \text{otherwise} \end{cases}$$

Remember that using a linear discriminant assumes that classes are linearly separable. That is to say, it is assumed that a hyperplane $\mathbf{w}^T \mathbf{x} = 0$ can be found that separates $\mathbf{x}^t \in C_1$ and $\mathbf{x}^t \in C_2$. If at a later stage we need the posterior probability—for example, to calculate risk—we need to use the sigmoid function at the output as

$$o = \mathbf{w}^T \mathbf{x}$$

$$y = \text{sigmoid}(o) = \frac{1}{1 + \exp[-\mathbf{w}^T \mathbf{x}]} \tag{11.4}$$

When there are K > 2 outputs, there are K perceptron, each of which has a weight vector w_i (see figure 11.2)

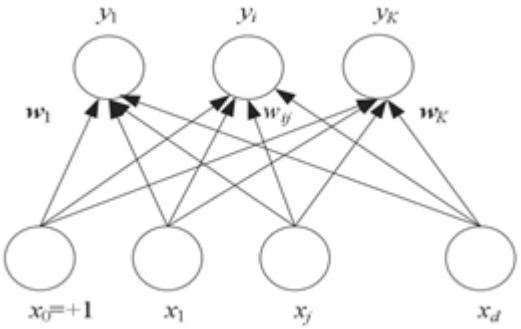


Figure 11.2 K parallel perceptrons. $x_j, j = 0, \ldots, d$ are the inputs and $y_i, i = 1, \ldots, K$ are the outputs. w_{ij} is the weight of the connection from input x_j to output y_i. Each output is a weighted sum of the inputs. When used for K-class classification problem, there is a postprocessing to choose the maximum, or softmax if we need the posterior probabilities.

$$y_i = \sum_{j=1}^{d} w_{ij} x_j + w_{i0} = \mathbf{w}_i^T \mathbf{x}$$

$$\mathbf{y} = \mathbf{W}\mathbf{x} \tag{11.5}$$

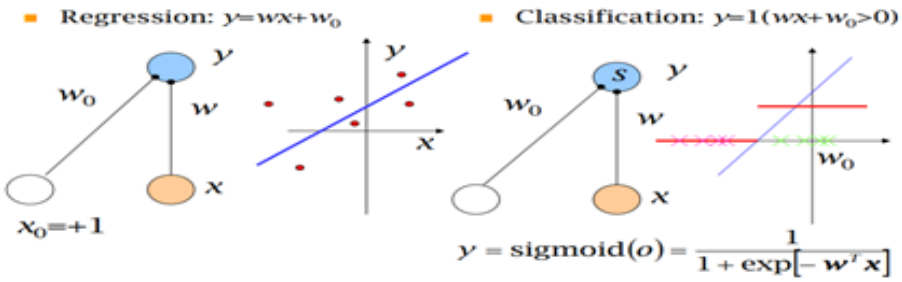
Where w_{ij} is the weight from input x_j to output y_i. W is the K × (d + 1) weight matrix of w_{ij} whose rows are the weight vectors of the K perceptron. When used for classification, during testing, we

$$\text{choose } C_i \text{ if } y_i = \max_k y_k$$

In the case of a neural network, the value of each perceptron is a local function of its inputs and its synaptic weights. However, in classification, if we need the posterior probabilities and use the softmax, we also need the values of the other outputs. So, to implement this as a neural network, we can see this as a two-stage process, where the first stage calculates the weighted sums, and the second stage calculates the softmax values; but we still denote this as a single layer of output units:

$$o_i = \mathbf{w}_i^T \mathbf{x}$$

$$y_i = \frac{\exp o_i}{\sum_k \exp o_k} \quad (11.6)$$



Perceptron model is also treated as one of the best and simplest types of Artificial Neural networks. However, it is a supervised learning algorithm of binary classifiers. Hence, we can consider it as a single-layer neural network with four main parameters, i.e., **input values, weights and Bias, net sum, and an activation function.**

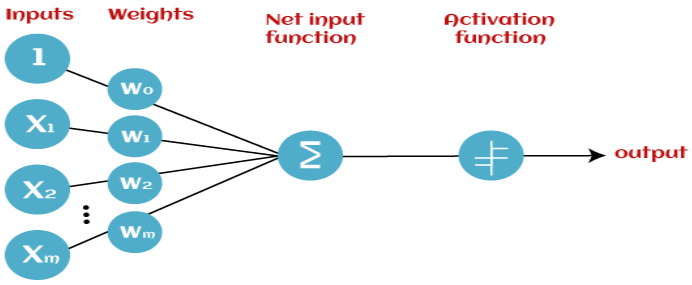
What is Binary classifier in Machine Learning?

In Machine Learning, binary classifiers are defined as the function that helps in deciding whether input data can be represented as vectors of numbers and belongs to some specific class.

Binary classifiers can be considered as linear classifiers. In simple words, we can understand it as a ***classification algorithm that can predict linear predictor function in terms of weight and feature vectors.***

Basic Components of Perceptron

Mr. Frank Rosenblatt invented the perceptron model as a binary classifier which contains three main components. These are as follows:



Input Nodes or Input Layer:

This is the primary component of Perceptron which accepts the initial data into the system for further processing. Each input node contains a real numerical value.

Weight and Bias:

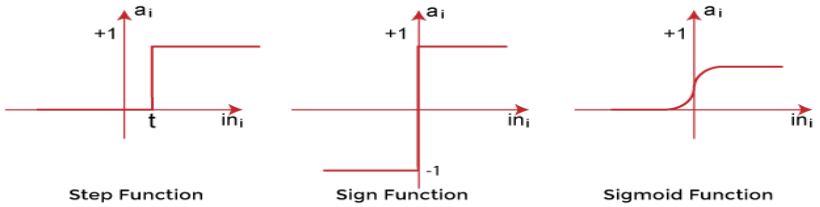
Weight parameter represents the strength of the connection between units. This is another most important parameter of Perceptron components. Weight is directly proportional to the strength of the associated input neuron in deciding the output. Further, Bias can be considered as the line of intercept in a linear equation.

Activation Function:

These are the final and important components that help to determine whether the neuron will fire or not. Activation Function can be considered primarily as a step function.

Types of Activation functions:

- Sign function
- Step function, and
- Sigmoid function



The data scientist uses the activation function to take a subjective decision based on various problem statements and forms the desired outputs. Activation function may differ (e.g., Sign, Step, and Sigmoid) in perceptron models by checking whether the learning process is slow or has vanishing or exploding gradients.

Perceptron Function: -

Perceptron function ''f(x)'' can be achieved as output by multiplying the input 'x' with the learned weight coefficient 'w'.

Mathematically, we can express it as follows:

f(x)=1; if w.x+b>0

otherwise, f(x)=0

'w' represents real-valued weights vector

'b' represents the bias

'x' represents a vector of input x values.

TRAINING A PERCEPTRON: -

The perceptron defines a hyper plane, and the neural network perceptron is just a way of implementing the hyper plane. Given a data sample, the weight values can be calculated offline and then when they are plugged in, the perceptron can be used to calculate the output values.

In training neural networks, we generally use online learning where we are not given the whole sample, but we are given instances one by one and would like the network to update its parameters after each instance, adapting itself slowly in time. Such an approach is interesting for a number of reasons.

1. It saves us the cost of storing the training sample in an external memory and storing the intermediate results during optimization. An approach like support vector machines may be quite costly with large samples, and in some applications, we may prefer a simpler approach where we do not need to store the whole sample and solve a complex optimization problem on it.
2. The problem may be changing in time, which means that the sample distribution is not fixed, and a training set cannot be chosen a priori. For example, we may be implementing a speech recognition system that adapts itself to its user.
3. There may be physical changes in the system. For example, in a robotic system, the components of the system may wear out, or sensors may degrade.

Online learning: -

In online learning, we do not write the error function over the whole sample but on individual instances. Starting from random initial weights, at each iteration we adjust the parameters a little bit to

minimize the error, without forgetting what we have previously learned. If this error function is differentiable, we can use gradient descent.
For example, in regression the error on the single instance pair with index t, (x^t , r^t), is

$$E^t(\mathbf{w}|\mathbf{x}^t, r^t) = \frac{1}{2}(r^t - y^t)^2 = \frac{1}{2}[r^t - (\mathbf{w}^T\mathbf{x}^t)]^2$$

And for j = 0,...,d, the online update is

$$\Delta w_j^t = \eta(r^t - y^t)x_j^t \quad (11.7)$$

Where η is the learning factor, which is gradually decreased in time for convergence. This is known as **stochastic gradient descent.**

Similarly, update rules can be derived for classification problems using logistic discrimination where updates are done after each pattern, instead of summing them and doing the update after a complete pass over the training set.

With two classes, for the single instance $(\mathbf{x}^t, \mathbf{r}^t)$ where $r_i^t = 1$ if $\mathbf{x}^t \in C_1$ and $r_i^t = 0$ if $\mathbf{x}^t \in C_2$, the single output is

$$y^t = \text{sigmoid}(\mathbf{w}^T\mathbf{x}^t)$$

and the cross-entropy is

$$E^t(\mathbf{w}|\mathbf{x}^t, r^t) = -r^t \log y^t - (1 - r^t)\log(1 - y^t)$$

Using gradient descent, we get the following online update rule for j = 0,...,d:

$$\Delta w_j^t = \eta(r^t - y^t)x_j^t \quad (11.8)$$

When there are $K > 2$ classes, for the single instance $(\mathbf{x}^t, \mathbf{r}^t)$ where $r_i^t = 1$ if $\mathbf{x}^t \in C_i$ and 0 otherwise, the outputs are

$$y_i^t = \frac{\exp \mathbf{w}_i^T\mathbf{x}^t}{\sum_k \exp \mathbf{w}_k^T\mathbf{x}^t}$$

and the cross-entropy is

$$E^t(\{\mathbf{w}_i\}_i|\mathbf{x}^t, \mathbf{r}^t) = -\sum_i r_i^t \log y_i^t$$

Using gradient descent, we get the following online update rule, for i = 1,...,K, j = 0,...,d:

$$\Delta w_{ij}^t = \eta(r_i^t - y_i^t)x_j^t \quad (11.9)$$

The pseudocode of the algorithm is given in figure 11.3.

```
For i = 1, ..., K
    For j = 0, ..., d
        w_ij ← rand(−0.01, 0.01)
Repeat
    For all (x^t, r^t) ∈ X in random order
        For i = 1, ..., K
            o_i ← 0
            For j = 0, ..., d
                o_i ← o_i + w_ij x_j^t
        For i = 1, ..., K
            y_i ← exp(o_i) / Σ_k exp(o_k)
        For i = 1, ..., K
            For j = 0, ..., d
                w_ij ← w_ij + η(r_i^t − y_i) x_j^t
Until convergence
```

Figure 11.3 Perceptron training algorithm implementing stochastic online gradient descent for the case with $K > 2$ classes.

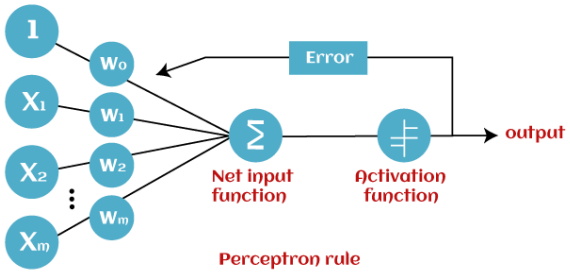
Both equations 11.7 and 11.9 have the form

$$\text{Update} = \text{LearningFactor} \cdot (\text{DesiredOutput} - \text{ActualOutput}) \cdot \text{Input} \quad (11.10)$$

Let us try to get some insight into what this does. First, if the actual output is equal to the desired output, no update is done. When it is done, the magnitude of the update increases as the difference between the desired output and the actual output increases. We also see that if the actual output is less than the desired output, update is positive if the input is positive and negative if the input is negative. This has the effect of increasing the actual output and decreasing the difference. If the actual output is greater than the desired output, update is negative if the input is positive and positive if the input is negative; this decreases the actual output and makes it closer to the desired output.

When an update is done, its magnitude depends also on the input. If the input is close to 0, its effect on the actual output is small and therefore its weight is also updated by a small amount. The greater an input, the greater the update of its weight.

Finally, the magnitude of the update depends on the learning factor, η. If it is too large, updates depend too much on recent instances; it is as if the system has a very short memory. If this factor is small, many updates may be needed for convergence.



LEARNING BOOLEAN FUNCTIONS: -

In a Boolean function, the inputs are binary and the output is 1 if the corresponding function value is true and 0 otherwise. Therefore, it can be seen as a two-class classification problem. As an example, for learning to AND two inputs, the table of inputs and required outputs is given in table 11.1.

x_1	x_2	r
0	0	0
0	1	0
1	0	0
1	1	1

Table 11.1 Input and output for the AND function.

An example of a perceptron that implements AND and its geometric interpretation in two dimensions is given in figure 11.4.

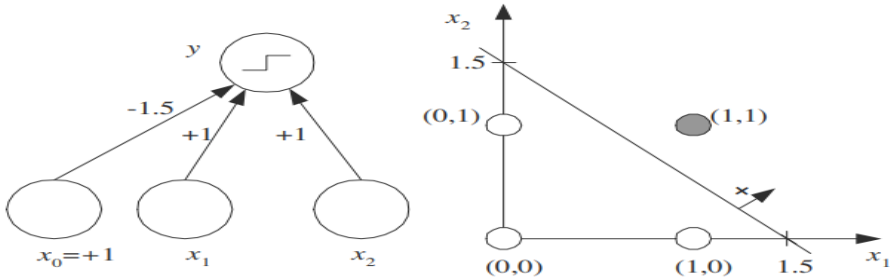


Figure 11.4 The perceptron that implements AND and its geometric interpretation.

The discriminant is

$$y = s(x_1 + x_2 - 1.5)$$

that is, $\boldsymbol{x} = [1, x_1, x_2]^T$ and $\boldsymbol{w} = [-1.5, 1, 1]^T$. Note that $y = s(x_1 + x_2 - 1.5)$ satisfies the four constraints given by the definition of AND function in table 11.1, for example, for $x_1 = 1, x_2 = 0$, $y = s(-0.5) = 0$. Similarly it can be shown that $y = s(x_1 + x_2 - 0.5)$ implements OR.

Though Boolean functions like AND and OR are linearly separable and are solvable using the perceptron, certain functions like XOR are not. The table of inputs and required outputs for XOR is given in table 11.2. As can be seen in figure 11.5, the problem is not linearly separable.

x_1	x_2	r
0	0	0
0	1	1
1	0	1
1	1	0

Table 11.2 Input and output for the XOR function.

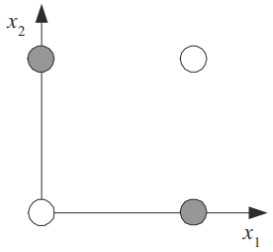


Figure 11.5 XOR problem is not linearly separable. We cannot draw a line where the empty circles are on one side and the filled circles on the other side.

Perceptron model works in two important steps as follows:

Step-1: -
In the first step first, multiply all input values with corresponding weight values and then add them to determine the weighted sum. Mathematically, we can calculate the weighted sum as follows:
∑wi*xi = x1*w1 + x2*w2 +…wn*xn
Add a special term called **bias 'b'** to this weighted sum to improve the model's performance.
∑wi*xi + b

Step-2: -
In the second step, an activation function is applied with the above-mentioned weighted sum, which gives us output either in binary form or a continuous value as follows:
Y = f(∑wi*xi + b)

MULTILAYER PERCEPTRON: -

A perceptron that has a single layer of weights can only approximate linear functions of the input and cannot solve problems like the XOR, where the discrimininant to be estimated is nonlinear. Similarly, a perceptron cannot be used for nonlinear regression. This limitation does not apply to feedforward networks with intermediate or hidden layers between the input and the output layers. If used for classification, such multilayer perceptron (MLP) can implement nonlinear discriminants and, if used for regression, can approximate nonlinear functions of the input.

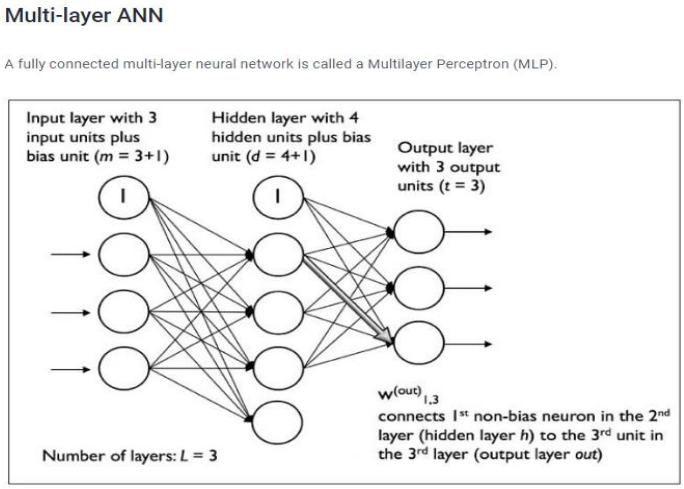
Architecture of MLP
The architecture of an MLP consists of three key components:
- The input layer,
- Hidden layer(s), and
- Output layer.

Input Layer: The input layer receives the input data and passes it to the hidden layer(s). It consists of nodes, each of which corresponds to a feature in the input data. The number of nodes in the input layer is equal to the number of features in the input data.

Hidden Layer(s): The hidden layer(s) are responsible for transforming the input data into a suitable representation for the output layer. They consist of nodes, which are connected to the input layer and

other hidden layer(s) through weights and biases. The number of hidden layer(s) and the number of nodes in each layer can be adjusted to optimize performance for a specific task. Common activation functions used in hidden layers include sigmoid, tanh, and rectified linear unit (ReLU).

Output Layer: The output layer receives the transformed representation of the input data from the hidden layer(s) and generates the final output. It consists of nodes, each of which corresponds to a class or a continuous value, depending on the task. The output layer uses a specific activation function that is appropriate for the task, such as softmax for classification and linear for regression.



Input x is fed to the input layer (including the bias), the "activation" propagates in the forward direction, and the values of the hidden units z_h are calculated (see figure 11.6). Each hidden unit is a perceptron by itself and applies the nonlinear sigmoid function to its weighted sum:

$$(11.11) \quad z_h = \text{sigmoid}(\boldsymbol{w}_h^T \boldsymbol{x}) = \frac{1}{1 + \exp\left[-\left(\sum_{j=1}^{d} w_{hj} x_j + w_{h0}\right)\right]}, \quad h = 1, \ldots, H$$

The output y_i are perceptron in the second layer taking the hidden units as their inputs

$$(11.12) \quad y_i = \boldsymbol{v}_i^T \boldsymbol{z} = \sum_{h=1}^{H} v_{ih} z_h + v_{i0}$$

where there is also a bias unit in the hidden layer, which we denote by z_0, and v_{i0} are the bias weights. The input layer of x_j is not counted since no computation is done there and when there is a hidden layer, this is a two-layer network.

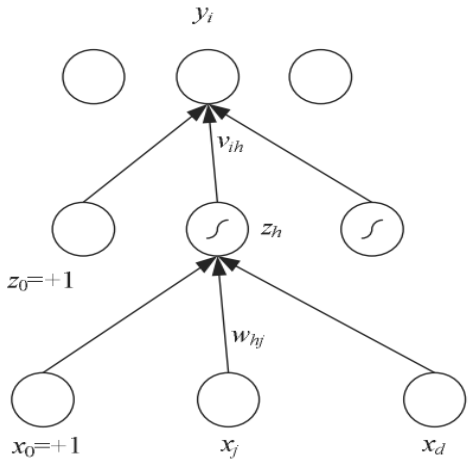


Figure 11.6 The structure of a multilayer perceptron. $x_j, j = 0, \ldots, d$ are the inputs and $z_h, h = 1, \ldots, H$ are the hidden units where H is the dimensionality of this hidden space. z_0 is the bias of the hidden layer. $y_i, i = 1, \ldots, K$ are the output units. w_{hj} are weights in the first layer, and v_{ih} are the weights in the second layer.

As usual, in a regression problem, there is no nonlinearity in the output layer in calculating y. In a two-class discrimination task, there is one sigmoid output unit and when there are K > 2 classes, there are K outputs with softmax as the output nonlinearity.

Sigmoid is the continuous, differentiable version of thresholding. We need differentiability because the learning equations we will see are gradient-based. Another sigmoid (S-shaped) nonlinear basis function that can be used is the hyperbolic tangent function, tanh, which ranges from −1 to +1, instead of 0 to +1. In practice, there is no difference between using the sigmoid and the tanh.

The output is a linear combination of the nonlinear basis function values computed by the hidden units. It can be said that the hidden units make a nonlinear transformation from the d-dimensional input space to the H-dimensional space spanned by the hidden units, and, in this space, the second output layer implements a linear function.

One is not limited to having one hidden layer, and more hidden layers with their own incoming weights can be placed after the first hidden layer with sigmoid hidden units, thus calculating nonlinear functions of the first layer of hidden units and implementing more complex functions of the inputs.

MLP AS A UNIVERSAL APPROXIMATOR: -

We can represent any Boolean function as a disjunction of conjunctions, and such a Boolean expression can be implemented by a multilayer perceptron with one hidden layer. Each conjunction is implemented by one hidden unit and the disjunction by the output unit. For example,

$$x_1 \text{ XOR } x_2 = (x_1 \text{ AND } \sim x_2) \text{ OR } (\sim x_1 \text{ AND } x_2)$$

We have seen previously how to implement AND and OR using perceptron. So two perceptron can in parallel implement the two AND, and another perceptron on top can OR them together (see figure 11.7). We see that the first layer maps inputs from the (x1, x2) to the (z1, z2) space defined by the first-layer perceptron. Note that both inputs, (0,0) and (1,1), are mapped to (0,0) in the (z1, z2) space, allowing linear separability in this second space.

Thus in the binary case, for every input combination where the output is 1, we define a hidden unit that checks for that particular conjunction of the input. The output layer then implements the disjunction.

We can extend this to the case where inputs are continuous to show that similarly, any arbitrary function with continuous input and outputs can be approximated with a multilayer perceptron. The proof of universal approximation is easy with two hidden layers.

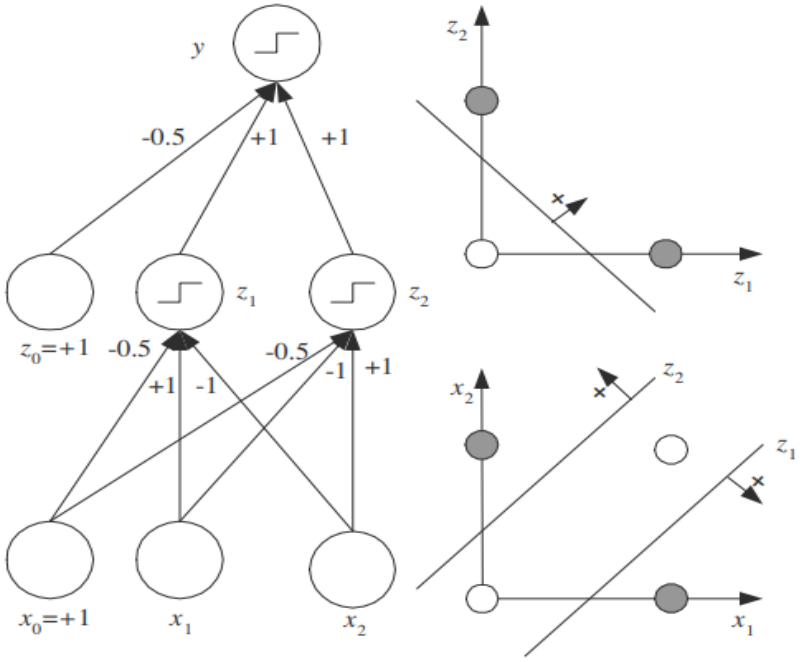


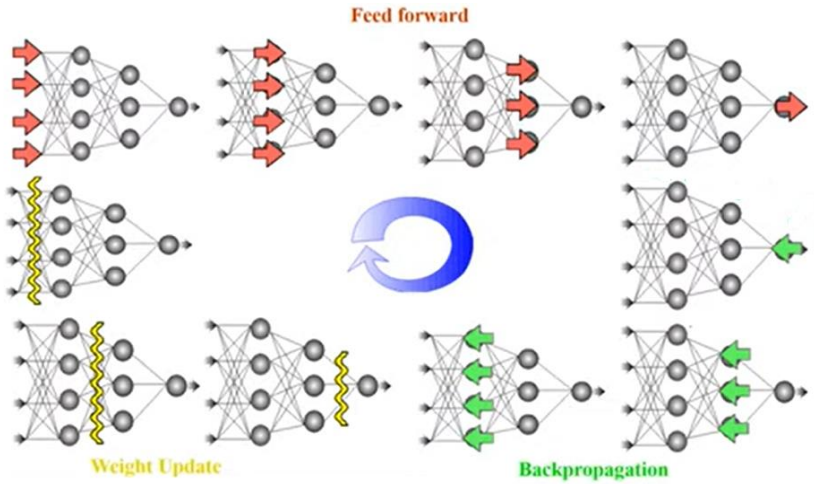
Figure 11.7 The multilayer perceptron that solves the XOR problem. The hidden units and the output have the threshold activation function with threshold at 0.

BACK PROPAGATION ALGORITHM: -

Training a multilayer perceptron is the same as training a perceptron; the only difference is that now the output is a nonlinear function of the input thanks to the nonlinear basis function in the hidden units. Considering the hidden units as inputs, the second layer is a perceptron and we already know how to update the parameters, v_{ij} , in this case, given the inputs z_h. For the first-layer weights, w_{hj}, we use the chain rule to calculate the gradient:

$$\frac{\partial E}{\partial w_{hj}} = \frac{\partial E}{\partial y_i}\frac{\partial y_i}{\partial z_h}\frac{\partial z_h}{\partial w_{hj}}$$

It is as if the error propagates from the output y back to the inputs and hence the name back propagation was coined.



The following are the concepts in back propagation Algorithm:

- Nonlinear Regression
- Two-Class Discrimination
- Multiclass Discrimination
- Multiple Hidden Layers

Nonlinear Regression: -

Let us first take the case of nonlinear regression (with a single output) calculated as

$$(11.13) \qquad y^t = \sum_{h=1}^{H} v_h z_h^t + v_0$$

with z_h computed by equation 11.11. The error function over the whole sample in regression is

$$(11.14) \qquad E(\mathbf{W}, \mathbf{v}|\mathcal{X}) = \frac{1}{2}\sum_t (r^t - y^t)^2$$

The second layer is a perceptron with hidden units as the inputs, and we use the least-squares rule to update the second-layer weights:

$$(11.15) \qquad \Delta v_h = \eta \sum_t (r^t - y^t) z_h^t$$

The first layer is also perceptron with the hidden units as the output units but in updating the first-layer weights, we cannot use the leastsquares rule directly as we do not have a desired output specified for the hidden units. This is where the chain rule comes into play. We write

$$\begin{aligned}
\Delta w_{hj} &= -\eta \frac{\partial E}{\partial w_{hj}} \\
&= -\eta \sum_t \frac{\partial E^t}{\partial y^t} \frac{\partial y^t}{\partial z_h^t} \frac{\partial z_h^t}{\partial w_{hj}} \\
&= -\eta \sum_t \underbrace{-(r^t - y^t)}_{\partial E^t/\partial y^t} \underbrace{v_h}_{\partial y^t/\partial z_h^t} \underbrace{z_h^t(1 - z_h^t)x_j^t}_{\partial z_h^t/\partial w_{hj}} \\
&= \eta \sum_t (r^t - y^t) v_h z_h^t (1 - z_h^t) x_j^t \qquad (11.16)
\end{aligned}$$

The product of the first two terms $(r^t - y^t)v_h$ acts like the error term for hidden unit h. This error is *backpropagated* from the error to the hidden unit. $(r^t - y^t)$ is the error in the output, weighted by the "responsibility" of the hidden unit as given by its weight v_h. In the third term, $z_h(1 - z_h)$ is the derivative of the sigmoid and x_j^t is the derivative of the weighted sum with respect to the weight w_{hj}. Note that the change in the first-layer weight, Δw_{hj}, makes use of the second-layer weight, v_h. Therefore, we should calculate the changes in both layers and update the first-layer weights, making use of the *old* value of the second-layer weights, then update the second-layer weights.

It is also possible to have online learning, by updating the weights after each pattern, thereby implementing stochastic gradient descent. A complete pass over all the patterns in the training set is called an epoch. We can divide the dataset of 2000 examples into batches of 500 then it will take 4 iterations to complete 1 epoch. The learning factor, η, should be chosen smaller in this case and patterns should be scanned in a random order. Online learning converges faster because there may be similar patterns in the dataset, and the stochasticity has an effect like adding noise and may help escape local minima.

It is also possible to have multiple output units, in which case a number of regression problems are learned at the same time. We have

$$y_i^t = \sum_{h=1}^{H} v_{ih} z_h^t + v_{i0} \qquad (11.17)$$

and the error is

$$E(\mathbf{W}, \mathbf{V}|\mathcal{X}) = \frac{1}{2} \sum_t \sum_i (r_i^t - y_i^t)^2 \qquad (11.18)$$

The batch update rules are then

$$\Delta v_{ih} = \eta \sum_t (r_i^t - y_i^t) z_h^t \qquad (11.19)$$

$$\Delta w_{hj} = \eta \sum_t \left[\sum_i (r_i^t - y_i^t) v_{ih} \right] z_h^t (1 - z_h^t) x_j^t \qquad (11.20)$$

$\sum_i (r_i^t - y_i^t) v_{ih}$ is the accumulated backpropagated error of hidden unit h from all output units. Pseudocode is given in figure 11.11.

Initialize all v_{ih} and w_{hj} to rand$(-0.01, 0.01)$
Repeat
  For all $(\boldsymbol{x}^t, r^t) \in \mathcal{X}$ in random order
    For $h = 1, \ldots, H$
      $z_h \leftarrow \text{sigmoid}(\boldsymbol{w}_h^T \boldsymbol{x}^t)$
    For $i = 1, \ldots, K$
      $y_i = \boldsymbol{v}_i^T \boldsymbol{z}$
    For $i = 1, \ldots, K$
      $\Delta \boldsymbol{v}_i = \eta(r_i^t - y_i^t)\boldsymbol{z}$
    For $h = 1, \ldots, H$
      $\Delta \boldsymbol{w}_h = \eta(\sum_i (r_i^t - y_i^t) v_{ih}) z_h (1 - z_h) \boldsymbol{x}^t$
    For $i = 1, \ldots, K$
      $\boldsymbol{v}_i \leftarrow \boldsymbol{v}_i + \Delta \boldsymbol{v}_i$
    For $h = 1, \ldots, H$
      $\boldsymbol{w}_h \leftarrow \boldsymbol{w}_h + \Delta \boldsymbol{w}_h$
Until convergence

Figure 11.11 Backpropagation algorithm for training a multilayer perceptron for regression with K outputs. This code can easily be adapted for two-class classification (by setting a single sigmoid output) and to $K > 2$ classification (by using softmax outputs).

Note that in this case, all output units share the same hidden units and thus use the same hidden representation, hence, we are assuming that corresponding to these different outputs, we have related prediction problems. An alternative is to train separate multilayer perceptron for the separate regression problems, each with its own separate hidden units.

Two-Class Discrimination: -
When there are two classes, one output unit suffices:

$$y^t = \text{sigmoid}\left(\sum_{h=1}^{H} v_h z_h^t + v_0\right) \quad (11.21)$$

which approximates $P(C_1|\boldsymbol{x}^t)$ and $\hat{P}(C_2|\boldsymbol{x}^t) \equiv 1 - y^t$.

$$E(\mathbf{W}, \boldsymbol{v}|\mathcal{X}) = -\sum_t r^t \log y^t + (1 - r^t) \log(1 - y^t) \quad (11.22)$$

The update equations implementing gradient descent are

$$\Delta v_h = \eta \sum_t (r^t - y^t) z_h^t \quad (11.23)$$

$$\Delta w_{hj} = \eta \sum_t (r^t - y^t) v_h z_h^t (1 - z_h^t) x_j^t \quad (11.24)$$

As in the simple perceptron, the update equations for regression and classification are identical (which does not mean that the values are).

Multiclass Discrimination: -
In a (K > 2)-class classification problem, there are K outputs

$$o_i^t = \sum_{h=1}^{H} v_{ih} z_h^t + v_{i0} \quad (11.25)$$

and we use softmax to indicate the dependency between classes; namely, they are mutually exclusive and exhaustive:

$$(11.26) \quad y_i^t = \frac{\exp o_i^t}{\sum_k \exp o_k^t}$$

where y_i approximates $P(C_i|\mathbf{x}^t)$. The error function is

$$(11.27) \quad E(\mathbf{W}, \mathbf{V}|\mathcal{X}) = -\sum_t \sum_i r_i^t \log y_i^t$$

and we get the update equations using gradient descent:

$$(11.28) \quad \Delta v_{ih} = \eta \sum_t (r_i^t - y_i^t) z_h^t$$

$$(11.29) \quad \Delta w_{hj} = \eta \sum_t \left[\sum_i (r_i^t - y_i^t) v_{ih} \right] z_h^t (1 - z_h^t) x_j^t$$

Multiple Hidden Layers: -

It is possible to have multiple hidden layers each with its own weights and applying the sigmoid function to its weighted sum. For regression, let us say, if we have a multilayer perceptron with two hidden layers, we write

$$z_{1h} = \text{sigmoid}(\mathbf{w}_{1h}^T \mathbf{x}) = \text{sigmoid}\left(\sum_{j=1}^{d} w_{1hj} x_j + w_{1h0} \right), \; h = 1, \ldots, H_1$$

$$z_{2l} = \text{sigmoid}(\mathbf{w}_{2l}^T \mathbf{z}_1) = \text{sigmoid}\left(\sum_{h=0}^{H_1} w_{2lh} z_{1h} + w_{2l0} \right), \; l = 1, \ldots, H_2$$

$$y = \mathbf{v}^T \mathbf{z}_2 = \sum_{l=1}^{H_2} v_l z_{2l} + v_0$$

where w_{1h} and w_{2l} are the first- and second-layer weights, z_{1h} and z_{2h} are the units on the first and second hidden layers, and v are the thirdlayer weights. Training such a network is similar except that to train the first-layer weights, we need to back propagate one more layer.

TRAINING PROCEDURES: -

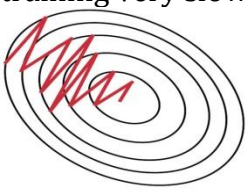
A gradient measures how much the output of a function changes if you change the inputs a little bit." A gradient simply measures the change in all weights with regard to the change in error. You can also think of a gradient as the slope of a function. The higher the gradient, the steeper the slope and the faster a model can learn. But if the slope is zero, the model stops learning.

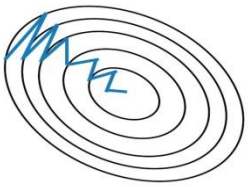
- Improving Convergence
- Overtraining
- Structuring the Network
- Hints

Improving Convergence: -

Gradient descent has various advantages. It is simple. It is local; namely, the change in a weight uses only the values of the presynaptic and postsynaptic units and the error (suitably back propagated). When online training is used, it does not need to store the training set and can adapt as the task to be learned changes. Because of these reasons, it can be (and is) implemented in hardware. But by itself, gradient descent converges slowly. When learning time is important, one can use more sophisticated optimization methods. However, there are two frequently used simple techniques that improve the performance of the gradient descent considerably, making gradient-based methods feasible in real applications.

- **Momentum: -**
  Momentum is a technique to prevent sensitive movement. When the gradient gets computed every iteration, it can have totally different direction and the steps make a zigzag path, which makes training very slow.


Stochastic Gradient Descent **without** Momentum


Stochastic Gradient Descent **with** Momentum

The idea is to take a running average by incorporating the previous update in the current change as if there is a momentum due to previous updates.

$$\Delta w_i^t = -\eta \frac{\partial E^t}{\partial w_i} + \alpha \Delta w_i^{t-1} \quad (11.30)$$

α is generally taken between 0.5 and 1.0. This approach is especially useful when online learning is used, where as a result we get an effect of averaging and smooth the trajectory during convergence. The disadvantage is that the past Δw_i^{t-1} values should be stored in extra memory.

- **Adaptive Learning Rate: -**
  In gradient descent, the learning factor η determines the magnitude of change to be made in the parameter. It is generally taken between 0.0 and 1.0, mostly less than or equal to 0.2. It can be made adaptive for faster convergence, where it is kept large when learning takes place and is decreased when learning slows down:

$$\Delta\eta = \begin{cases} +a & \text{if } E^{t+\tau} < E^t \\ -b\eta & \text{otherwise} \end{cases} \quad (11.31)$$

Thus we increase η by a constant amount if the error on the training set decreases and decrease it geometrically if it increases. Because E may oscillate from one epoch to another, it is a better idea to take the average of the past few epochs as E^t.

Overtraining: -

A multilayer perceptron with d inputs, H hidden units, and K outputs has $H(d+1)$ weights in the first layer and $K(H+1)$ weights in the second layer. Both the space and time complexity of an MLP is $\mathcal{O}(H \cdot (K+d))$. When e denotes the number of training epochs, training time complexity is $\mathcal{O}(e \cdot H \cdot (K+d))$.

In an application, d and K are predefined and H is the parameter that we play with to tune the complexity of the model. We know from previous chapters that an over complex model memorizes the noise in the training set and does not generalize to the validation set. For example, we have previously seen this phenomenon in the case of polynomial regression where we noticed that in the presence of noise or small samples, increasing the polynomial order leads to worse generalization.

Remember that initially all the weights are close to 0 and thus have little effect. As training continues, the most important weights start moving away from 0 and are utilized. But if training is continued further on to get less and less error on the training set, almost all weights are updated away from 0 and effectively become parameters. Thus as training continues, it is as if new parameters are added to the system, increasing the complexity and leading to poor generalization. Learning should be stopped early to alleviate this problem of overtraining. The optimal point to stop training, and the

optimal number of hidden units, is determined through cross-validation, which involves testing the network's performance on validation data unseen during training.

Structuring the Network: -

In some applications, we may believe that the input has a local structure. For example, in vision we know that nearby pixels are correlated and there are local features like edges and corners; any object, for example, a handwritten digit, may be defined as a combination of such primitives. Similarly, in speech, locality is in time and inputs close in time can be grouped as speech primitives. By combining these primitives, longer utterances, for example, speech phonemes may be defined. In such a case when designing the MLP, hidden units are not connected to all input units because not all inputs are correlated. Instead, we define hidden units that define a window over the input space and are connected to only a small local subset of the inputs. This decreases the number of connections and therefore the number of free parameters.

We can repeat this in successive layers where each layer is connected to a small number of local units below and checks for a more complicated feature by combining the features below in a larger part of the input space until we get to the output layer (see figure 11.14). For example, the input may be pixels. By looking at pixels, the first hidden layer units may learn to check for edges of various orientations. Then by combining edges, the second hidden layer units can learn to check for combinations of edges—for example, arcs, corners, line ends—and then combining them in upper layers, the units can look for semi-circles, rectangles, or in the case of a face recognition application, eyes, mouth, so forth. This is the example of a hierarchical cone where features get more complex, abstract, and fewer in numbers as we go up the network until we get to classes.

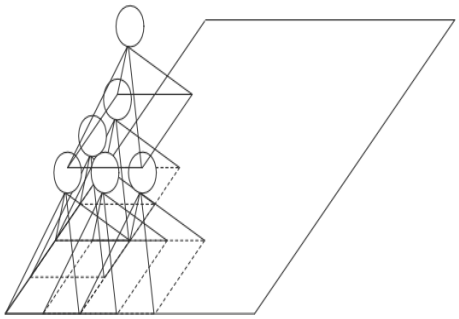


Figure 11.14 A structured MLP. Each unit is connected to a local group of units below it and checks for a particular feature—for example, edge, corner, and so forth—in vision. Only one hidden unit is shown for each region; typically there are many to check for different local features.

In such a case, we can further reduce the number of parameters by weight sharing. Taking the example of visual recognition again, we can see that when we look for features like oriented edges, they may be present in different parts of the input space. So instead of defining independent hidden units learning different features in different parts of the input space, we can have copies of the same hidden units looking at different parts of the input space (see figure 11.15). During learning, we calculate the gradients by taking different inputs, and then we average these up and make a single update. This implies a single parameter that defines the weight on multiple connections. Also, because the update on a weight is based on gradients for several inputs, it is as if the training set is effectively multiplied.

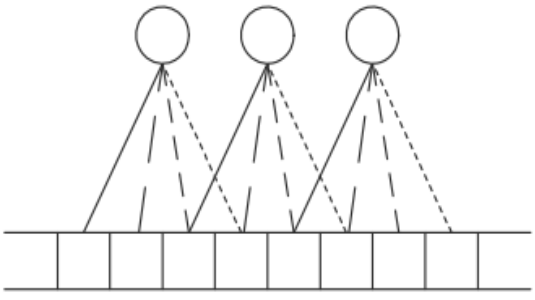


Figure 11.15 In weight sharing, different units have connections to different inputs but share the same weight value (denoted by line type). Only one set of units is shown; there should be multiple sets of units, each checking for different features.

Hints: -

The knowledge of local structure allows us to prestructure the multilayer network, and with weight sharing it has fewer parameters. The alternative of an MLP with completely connected layers has no such structure and is more difficult to train. Knowledge of any sort related to the application should be built into the network structure whenever possible. These are called hints and are the properties of the target function that are known to us independent of the training examples.

In image recognition, there are invariance hints: the identity of an object does not change when it is rotated, translated, or scaled. Hints are auxiliary information that can be used to guide the learning process and are especially useful when the training set is limited.

Figure 11.16 The identity of the object does not change when it is translated, rotated, or scaled. Note that this may not always be true, or may be true up to a point: 'b' and 'q' are rotated versions of each other. These are hints that can be incorporated into the learning process to make learning easier.

There are different ways in which hints can be used:

Virtual examples:

1. Hints can be used to create virtual examples. For example, knowing that the object is invariant to scale, from a given training example, we can generate multiple copies at different scales and add them to the training set with the same label. This has the advantage that we increase the training set and do not need to modify the learner in any way. The problem may be that too many examples may be needed for the learner to learn the invariance.
2. Hints can be used to create virtual examples. For example, knowing that the object is invariant to scale, from a given training example, we can generate multiple copies at different scales and add them to the training set with the same label. This has the advantage that we increase the training set and do not need to modify the learner in any way. The problem may be that too many examples may be needed for the learner to learn the invariance.
3. The hint may be incorporated into the network structure. Local structure and weight sharing, is one example where we get invariance to small translations and rotations.
4. The hint may also be incorporated by modifying the error function. Let us say we know that $\boldsymbol{x}$ and $\boldsymbol{x}'$ are the same from the application's point of view, where $\boldsymbol{x}'$ may be a "virtual example" of $\boldsymbol{x}$. That is, $f(\boldsymbol{x}) = f(\boldsymbol{x}')$, when $f(\boldsymbol{x})$ is the function we would like to approximate. Let us denote by $g(\boldsymbol{x}|\theta)$, our approximation function, for example, an MLP where θ are its weights. Then, for all such pairs $(\boldsymbol{x}, \boldsymbol{x}')$, we define the penalty function

$$E_h = \left[g(\boldsymbol{x}|\theta) - g(\boldsymbol{x}'|\theta)\right]^2$$

and add it as an extra term to the usual error function:

$$E' = E + \lambda_h \cdot E_h$$

DIMENSIONALITY REDUCTION: -

In a multilayer perceptron, if the number of hidden units is less than the number of inputs, the first layer performs a dimensionality reduction. The form of this reduction and the new space spanned by the hidden units depend on what the MLP is trained for. If the MLP is for classification with output units following the hidden layer, then the new space is defined and the mapping is learned to minimize classification error.

We can get an idea of what the MLP is doing by analyzing the weights. We know that the dot product is maximum when the two vectors are identical. So we can think of each hidden unit as defining

a template in its incoming weights, and by analyzing these templates, we can extract knowledge from a trained MLP. If the inputs are normalized, weights tell us of their relative importance. Such analysis is not easy but gives us some insight as to what the MLP is doing and allows us to peek into the black box.

Auto associator: -

An interesting architecture is the auto associator, which is an MLP architecture where there are as many outputs as there are inputs, and the required outputs are defined to be equal to the inputs (see figure 11.19). To be able to reproduce the inputs again at the output layer, the MLP is forced to find the best representation of the inputs in the hidden layer. When the number of hidden units is less than the number of inputs, this implies dimensionality reduction. Once the training is done, the first layer from the input to the hidden layer acts as an encoder, and the values of the hidden units make up the encoded representation. The second layer from the hidden units to the output units acts as a decoder, reconstructing the original signal from its encoded representation.

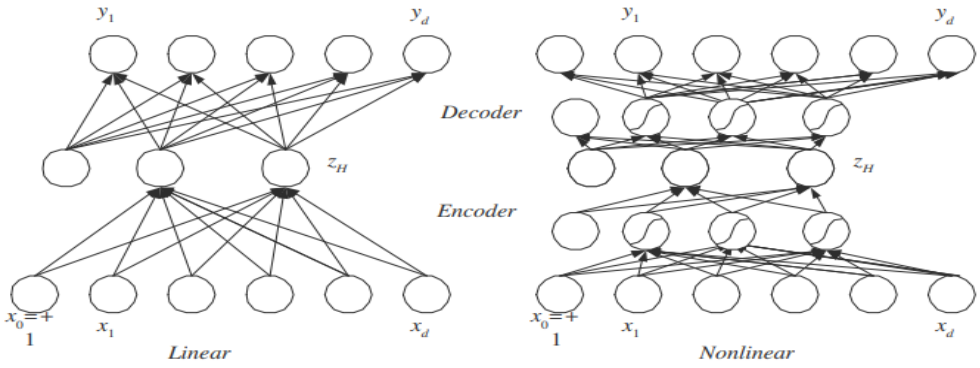


Figure 11.19 In the autoassociator, there are as many outputs as there are inputs and the desired outputs are the inputs. When the number of hidden units is less than the number of inputs, the MLP is trained to find the best coding of the inputs on the hidden units, performing dimensionality reduction. On the left, the first layer acts as an encoder and the second layer acts as the decoder. On the right, if the encoder and decoder are multilayer perceptrons with sigmoid hidden units, the network performs nonlinear dimensionality reduction.

It has been shown that an MLP with one hidden layer of units implements principal components analysis, except that the hidden unit weights are not the eigenvectors sorted in importance using the eigenvalues but span the same space as the H principal eigenvectors. If the encoder and decoder are not one layer but multilayer perceptron with sigmoid nonlinearity in the hidden units, the encoder implements nonlinear dimensionality reduction.

Sammon mapping: -

Another way to use an MLP for dimensionality reduction is through multidimensional scaling, show how an MLP can be used to learn the Sammon mapping.

Sammon stress is defined as

$$E(\theta|\mathcal{X}) = \sum_{r,s} \left[\frac{\|\boldsymbol{g}(\boldsymbol{x}^r|\theta) - \boldsymbol{g}(\boldsymbol{x}^s|\theta)\| - \|\boldsymbol{x}^r - \boldsymbol{x}^s\|}{\|\boldsymbol{x}^r - \boldsymbol{x}^s\|} \right]^2 \quad (11.40)$$

An MLP with d inputs, H hidden units, and $k < d$ output units is used to implement $g(\boldsymbol{x}|\theta)$, mapping the d-dimensional input to a k-dimensional vector, where θ corresponds to the weights of the MLP. Given a dataset of $\mathcal{X} = \{\boldsymbol{x}^t\}_t$, we can use gradient descent to minimize the Sammon stress directly to learn the MLP, namely, $g(\boldsymbol{x}|\theta)$, such that the distances between the k-dimensional representations are as close as possible to the distances in the original space.

LEARNING TIME: -

Until now, we have been concerned with cases where the input is fed once, all together. In some applications, the input is temporal where we need to learn a temporal sequence. In others, the output may also change in time. Examples are as follows:

- **Sequence recognition: -** This is the assignment of a given sequence to one of several classes. Speech recognition is one example where the input signal sequence is the spoken speech and the output is the code of the word spoken. That is, the input changes in time but the output does not.
- **Sequence reproduction: -** Here, after seeing part of a given sequence, the system should predict the rest. Time-series prediction is one example where the input is given but the output changes.
- **Temporal association: -** This is the most general case where a particular output sequence is given as output after a specific input sequence. The input and output sequences may be different. Here both the input and the output change in time.

Time Delay Neural Networks: -

The easiest way to recognize a temporal sequence is by converting it to a spatial sequence. Then any method discussed up to this point can be utilized for classification. In a time delay neural network previous inputs are delayed in time so as to synchronize with the final input, and all are fed together as input to the system.

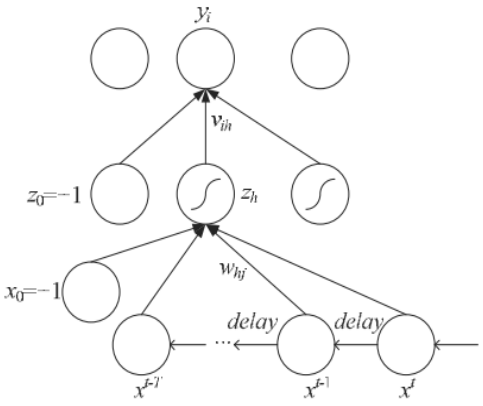


Figure 11.20 A time delay neural network. Inputs in a time window of length T are delayed in time until we can feed all T inputs as the input vector to the MLP.

Back propagation can then be used to train the weights. To extract features local in time, one can have layers of structured connections and weight sharing to get translation invariance in time. The main restriction of this architecture is that the size of the time window we slide over the sequence should be fixed a priori.

Recurrent Networks: -

In a recurrent network, additional to the feed forward connections, units have self-connections or connections to units in the previous layers. This recurrency acts as a short-term memory and lets the network remember what happened in the past.

Most frequently, one uses a partially recurrent network where a limited number of recurrent connections are added to a multilayer perceptron (see figure 11.21). This combines the advantage of the nonlinear approximation ability of a multilayer perceptron with the temporal representation ability of the recurrency, and such a network can be used to implement any of the three temporal association tasks. It is also possible to have hidden units in the recurrent backward connections, these being known as context units. No formal results are known to determine how to choose the best architecture given a particular application.

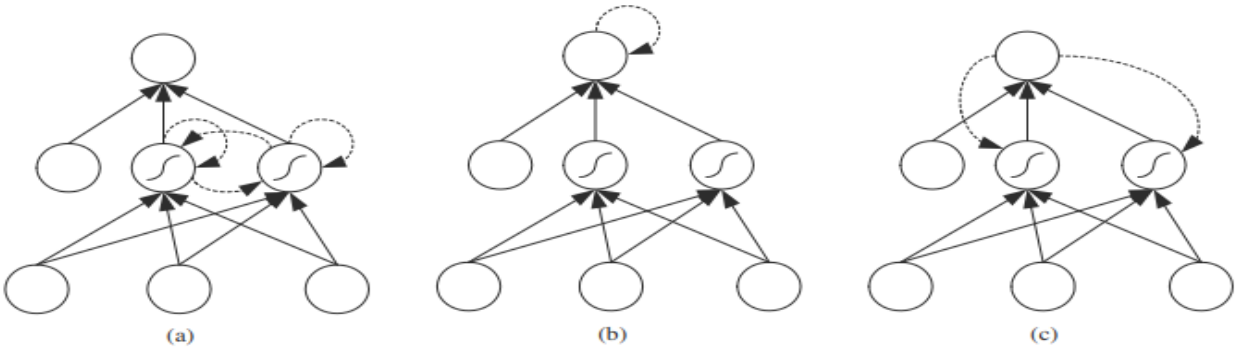


Figure 11.21 Examples of MLP with partial recurrency. Recurrent connections are shown with dashed lines: (a) self-connections in the hidden layer, (b) self-connections in the output layer, and (c) connections from the output to the hidden layer. Combinations of these are also possible.

Unfolding in time: -

If the sequences have a small maximum length, then unfolding in time can be used to convert an arbitrary recurrent network to an equivalent feed forward network (see figure 11.22). A separate unit and connection is created for copies at different times. The resulting network can be trained with back propagation with the additional requirement that all copies of each connection should remain identical. The solution, as in weight sharing, is to sum up the different weight changes in time and change the weight by the average. This is called back propagation through time. The problem with this approach is the memory requirement if the length of the sequence is large Real time recurrent learning is an algorithm for training recurrent networks without unfolding and has the advantage that it can use sequences of arbitrary length.

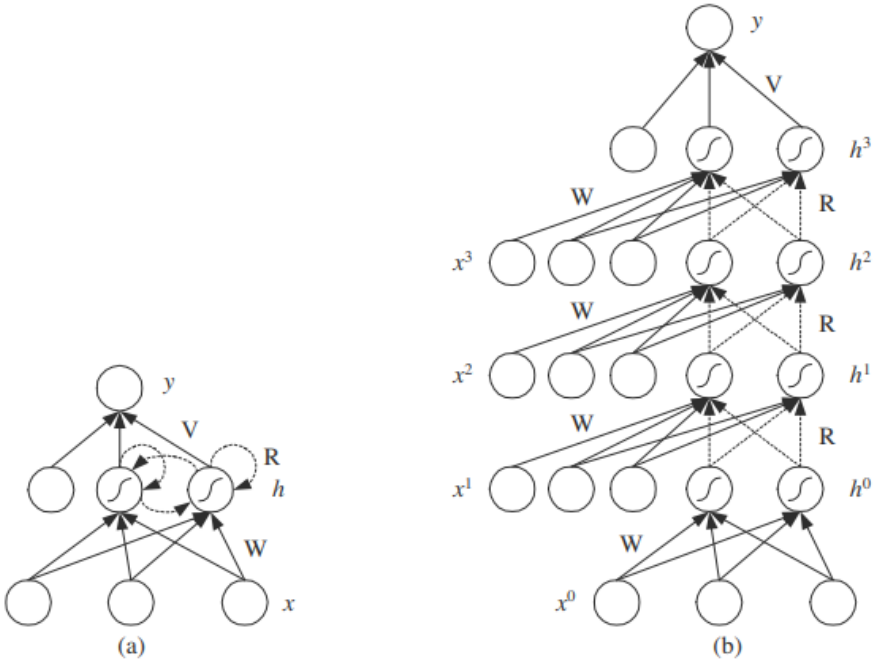


Figure 11.22 Backpropagation through time: (a) recurrent network, and (b) its equivalent unfolded network that behaves identically in four steps.

UNIT WISE IMPORTANT QUESTIONS: -

1. What is a Perceptron? Explain the working of a perceptron with a neat diagram.
2. Explain Perceptron training Algorithm

3. How Multilayer Perceptron can implement nonlinear discriminants explain.in detail.
4. Illustrate why MLP as a Universal Approximator
5. Make use of Back propagation Algorithm for training a multi- layer perceptron
6. Compare Two- class Discrimination and Multi-class Discrimination
7. Outline the concept of Training Procedures
8. List and explain the methods for Reducing Dimensionality.
9. Compare Feature Extraction and Feature Selection techniques. Explain how dimensionality can be reduced using subset selection procedure with an example.
10. What is the significance of Learning Time and Explain Recurrent Neural Networks.
11. Show the perceptron that calculates AND of its two inputs.
12. Show the perceptron that calculates OR of its two inputs.
13. Show the perceptron that calculates NOT of its input.
14. Show the perceptron that calculates NAND of its two inputs.
15. Show the perceptron that calculates NOR of its two inputs.
16. Show the perceptron that calculates XOR of its two inputs.
17. Show the perceptron that calculates the parity of its three inputs.
18. Derive the update equations when the hidden units use tanh, instead of the sigmoid. Use the fact that $\tanh' = (1 - \tanh^2)$.

SOLUTION: -

The case of nonlinear regression (with a single output) calculated as

$$y^t = \sum_{h=1}^{H} v_h z_h^t + v_0 \quad (11.13)$$

with z_h computed by equation 11.11. The error function over the whole sample in regression is

$$E(\mathbf{W}, \mathbf{v}|\mathcal{X}) = \frac{1}{2}\sum_t (r^t - y^t)^2 \quad (11.14)$$

The second layer is a perceptron with hidden units as the inputs, and we use the least-squares rule to update the second-layer weights:

$$\Delta v_h = \eta \sum_t (r^t - y^t) z_h^t \quad (11.15)$$

The first layer is also perceptron with the hidden units as the output units but in updating the first-layer weights, we cannot use the leastsquares rule directly as we do not have a desired output specified for the hidden units. This is where the chain rule comes into play. We write

$$\begin{aligned}
\Delta w_{hj} &= -\eta \frac{\partial E}{\partial w_{hj}} \\
&= -\eta \sum_t \frac{\partial E^t}{\partial y^t} \frac{\partial y^t}{\partial z_h^t} \frac{\partial z_h^t}{\partial w_{hj}} \\
&= -\eta \sum_t \underbrace{-(r^t - y^t)}_{\partial E^t/\partial y^t} \underbrace{v_h}_{\partial y^t/\partial z_h^t} \underbrace{z_h^t(1 - z_h^t)x_j^t}_{\partial z_h^t/\partial w_{hj}} \\
&= \eta \sum_t (r^t - y^t) v_h z_h^t (1 - z_h^t) x_j^t \quad (11.16)
\end{aligned}$$

The only difference from the case where hidden units use sigmoid is that the derivative of the hidden unit activations will change. In updating the first layer weights (equation 11.16), instead of $z_h(1 - z_h)$, we will have $(1 - z_h^2)$.

19. Derive the update equations for an MLP with two hidden layers.

$$z_{1h} = \text{sigmoid}(\mathbf{w}_{1h}^T \mathbf{x}) = \text{sigmoid}\left(\sum_{j=1}^{d} w_{1hj} x_j + w_{1h0}\right), \; h = 1, \ldots, H_1$$

$$z_{2l} = \text{sigmoid}(\mathbf{w}_{2l}^T \mathbf{z}_1) = \text{sigmoid}\left(\sum_{h=0}^{H_1} w_{2lh} z_{1h} + w_{2l0}\right), \; l = 1, \ldots, H_2$$

$$y_i = \mathbf{v}_i^T \mathbf{z}_2 = \sum_{l=1}^{H_2} v_{il} z_{2l} + v_0$$

Let us take the case of regression:

$$E = \frac{1}{2} \sum_t \sum_i (r_i^t - y_i^t)^2$$

We just continue back propagating, that is continue the chain rule, and we can write error on a layer as a function of the error in the layer after, carrying the supervised error in the output layer to layers before:

$$err_i \equiv r_i^t - y_i^t \text{ and } \Delta v_{il} = \eta \sum_t err_i z_{2l}$$

$$err_{2l} \equiv \left[\sum_i err_i v_i\right] z_{2l}(1 - z_{2l}) \text{ and } \Delta w_{2lh} = \eta \sum_t err_{2l} z_{1h}$$

$$err_{1h} \equiv \left[\sum_l err_{2l} w_{2lh}\right] z_{1h}(1 - z_{1h}) \text{ and } \Delta w_{hj} = \eta \sum_t err_{1h} x_j$$

20. Parity is cyclic shift invariant, for example, “0101” and “1010” have the same parity. Propose a multilayer perceptron to learn the parity function using this hint.
One can generate virtual examples by adding shifted versions of instances to the training set. Or, one can define local hidden units with weight sharing to keep track of local parity which are then combined to calculate the overall parity.