POLYTECHNIC UNIVERSITY
Department of Computer and Information Science

Backpropagation in Multilayer Perceptrons

K. Ming Leung

Abstract: A training algorithm for multilayer perceptrons known as backpropagation is discussed.

Directory

Last Revision Date: April 17, 2007

Table of Contents

1. Introduction

Single-layer networks are capable of solving only linearly separable classification problems. Researches were aware of this limitation and have proposed multilayer networks to overcome this. However they were not able to generalize their training algorithms to these multilayer networks until the thesis work of Werbos in 1974. Unfortunately this work was not known to the neural network community until after it was rediscovered independently by a number of people in the middle 1980s. The training algorithm, now known as backpropagation (BP), is a generalization of the Delta (or LMS) rule for single layer perceptron to include differentiable transfer function in multilayer networks. BP is currently the most widely used NN.

2. Multilayer Perceptron

We want to consider a rather general NN consisting of L layers (of course not counting the input layer). Let us consider an arbitrary layer, say ℓ, which has N_ℓ neurons $X_1^{(\ell)}, X_2^{(\ell)}, \ldots, X_{N_\ell}^{(\ell)}$, each with a transfer function $f^{(\ell)}$. Notice that the transfer function may be dif-

ferent from layer to layer. As in the extended Delta rule, the transfer function may be given by any differentiable function, but does not need to be linear. These neurons receive signals from the neurons in the preceding layer, $\ell - 1$. For example, neuron $X_j^{(\ell)}$ receives a signal from $X_i^{(\ell-1)}$ with a weight factor $w_{ij}^{(\ell)}$. Therefore we have an $N_{\ell-1}$ by N_ℓ weight matrix, $\mathbf{W}^{(\ell)}$, whose elements are given by $w_{ij}^{(\ell)}$, for $i = 1, 2, \ldots, N_{\ell-1}$ and $j = 1, 2, \ldots, N_\ell$. Neuron $X_j^{(\ell)}$ also has a bias given by $b_j^{(\ell)}$, and its activation is $a_j^{(\ell)}$. To simplify the notation, we will use $n_j^{(\ell)} (= y_{\text{in},j})$ to denote the net input into neuron $X_j^{(\ell)}$. It is given by

$$n_j^{(\ell)} = \sum_{i=1}^{N_{\ell-1}} a_i^{(\ell-1)} w_{ij}^{(\ell)} + b_j^{(\ell)}, \qquad j = 1, 2, \ldots, N_\ell.$$

Thus the activation of neuron $X_j^{(\ell)}$ is

$$a_j^{(\ell)} = f^{(\ell)}(n_j^{(\ell)}) = f^{(\ell)}(\sum_{i=1}^{N_{\ell-1}} a_i^{(\ell-1)} w_{ij}^{(\ell)} + b_j^{(\ell)}).$$

We can consider the zeroth layer as the input layer. If an input vector $\mathbf{x}$ has N components, then $N_0 = N$ and neurons in the input layer have activations $a_i^{(0)} = x_i, i = 1, 2, \ldots, N_0$.

Layer L of the network is the output layer. Assuming that the output vector $\mathbf{y}$ has M components, then we must have $N_L = M$. These components are given by $y_j = a_j^{(L)}, j = 1, 2, \ldots, M$.

For any given input vector, the above equations can be used to find the activation for each neuron for any given set of weights and biases. In particular the network output vector $\mathbf{y}$ can be found. The remaining question is how to train the network to find a set of weights and biases in order for it to perform a certain task.

3. Backpropagation Algorithm

We will now consider training a rather general multilayer perceptron for pattern association using the BP algorithm. Training is carried out supervised and so we assume that a set of pattern pairs (or associations): $\mathbf{s}^{(q)} : \mathbf{t}^{(q)}, q = 1, 2, \ldots, Q$ is given. The training vectors $\mathbf{s}^{(q)}$ have N components,

$$\mathbf{s}^{(q)} = \begin{bmatrix} s_1^{(q)} & s_2^{(q)} & \ldots & s_N^{(q)} \end{bmatrix},$$

and their targets $\mathbf{t}^{(q)}$ have M components,

$$\mathbf{t}^{(q)} = \begin{bmatrix} t_1^{(q)} & t_2^{(q)} & \ldots & t_M^{(q)} \end{bmatrix}.$$

Just like in the Delta rule, the training vectors are presented one at a time to the network during training. Suppose in time step t of the training process, a training vector $\mathbf{s}^{(q)}$ for a particular q is presented as input, $\mathbf{x}(t)$, to the network. The input signal can be propagated forward through the network using the equations in the last section and the current set of weights and biases to obtain the corresponding network output, $\mathbf{y}(t)$. The weights and biases are then adjusted using

the steepest descent algorithm to minimize the square of the error for this training vector:

$$E = \|\mathbf{y}(t) - \mathbf{t}(t)\|^2,$$

where $\mathbf{t}(t) = \mathbf{t}^{(q)}$ is the corresponding target vector for the chosen training vector $\mathbf{s}^{(q)}$.

This square error E is a function of all the weights and biases of the entire network since $\mathbf{y}(t)$ depends on them. We need to find the set of updating rule for them based on the steepest descent algorithm:

$$w_{ij}^{(\ell)}(t+1) = w_{ij}^{(\ell)}(t) - \alpha \frac{\partial E}{\partial w_{ij}^{(\ell)}(t)}$$

$$b_{j}^{(\ell)}(t+1) = b_{j}^{(\ell)}(t) - \alpha \frac{\partial E}{\partial b_{j}^{(\ell)}(t)},$$

where $\alpha(> 0)$ is the learning rate.

To compute these partial derivatives, we need to understand how E depends on the weights and biases. First E depends explicitly on the network output $\mathbf{y}(t)$ (the activations of the last layer, $\mathbf{a}^{(L)}$), which

then depends on the net input into the $L-$th layer, $\mathbf{n}^{(L)}$. In turn $\mathbf{n}^{(L)}$ is given by the activations of the preceding layer and the weights and biases of layer L. The explicit relation is: for brevity, the dependence on step t is omitted

$$\begin{aligned} E &= \|\mathbf{y} - \mathbf{t}(t)\|^2 = \|\mathbf{a}^{(L)} - \mathbf{t}(t)\|^2 = \|f^{(L)}(\mathbf{n}^{(L)}) - \mathbf{t}(t)\|^2 \\ &= \left\| f^{(L)} \left(\sum_{i=1}^{N_{L-1}} a_i^{(L-1)} w_{ij}^{(L)} + b_j^{(L)} \right) - \mathbf{t}(t) \right\|^2 . \end{aligned}$$

It is then easy to compute the partial derivatives of E with respect to the elements of $\mathbf{W}^{(L)}$ and $\mathbf{b}^{(L)}$ using the chain rule for differentiation. We have

$$\frac{\partial E}{\partial w_{ij}^{(L)}} = \sum_{n=1}^{N_L} \frac{\partial E}{\partial n_n^{(L)}} \frac{\partial n_n^{(L)}}{\partial w_{ij}^{(L)}}.$$

Notice the sum is needed in the above equation for correct application of the chain rule. We now define the sensitivity vector for a general

layer ℓ to have components

$$s_n^{(\ell)} = \frac{\partial E}{\partial n_n^{(\ell)}} \qquad n = 1, 2, \ldots, N_\ell.$$

This is called the sensitivity of neuron $X_n^{(\ell)}$ because it gives the change in the output error, E, per unit change in the net input it receives.

For layer L, it is easy to compute the sensitivity vector directly using the chain rule to obtain

$$s_n^{(L)} = 2\left(a_n^{(L)} - t_n(t)\right)\dot{f}^{(L)}(n_n^{(L)}), \qquad n = 1, 2, \ldots, N_L.$$

where $\dot{f}$ denotes the derivative of the transfer function f. We also know that

$$\frac{\partial n_n^{(L)}}{\partial w_{ij}^{(L)}} = \frac{\partial}{\partial w_{ij}^{(L)}}\left(\sum_{m=1}^{N_{L-1}} a_m^{(L-1)} w_{mn}^{(L)} + b_n^{(L)}\right) = \delta_{nj} a_i^{(L-1)}.$$

Therefore we have

$$\frac{\partial E}{\partial w_{ij}^{(L)}} = a_i^{(L-1)}\ s_j^{(L)}.$$

Similarly,

$$\frac{\partial E}{\partial b_j^{(L)}} = \sum_{n=1}^{N_L} \frac{\partial E}{\partial n_n^{(L)}} \frac{\partial n_n^{(L)}}{\partial b_j^{(L)}},$$

and since

$$\frac{\partial n_n^{(L)}}{\partial b_j^{(L)}} = \delta_{nj},$$

we have

$$\frac{\partial E}{\partial b_j^{(L)}} = s_j^{(L)}.$$

For a general layer, ℓ, we can write

$$\frac{\partial E}{\partial w_{ij}^{(\ell)}} = \sum_{n=1}^{N_\ell} \frac{\partial E}{\partial n_n^{(\ell)}} \frac{\partial n_n^{(\ell)}}{\partial w_{ij}^{(\ell)}} = \sum_{n=1}^{N_\ell} s_n^{(\ell)} \frac{\partial n_n^{(\ell)}}{\partial w_{ij}^{(\ell)}}.$$

$$\frac{\partial E}{\partial b_j^{(\ell)}} = \sum_{n=1}^{N_\ell} \frac{\partial E}{\partial n_n^{(\ell)}} \frac{\partial n_n^{(\ell)}}{\partial b_j^{(\ell)}} = \sum_{n=1}^{N_\ell} s_n^{(\ell)} \frac{\partial n_n^{(\ell)}}{\partial b_j^{(\ell)}}.$$

Since

$$n_n^{(\ell)} = \sum_{m=1}^{N_{\ell-1}} a_m^{(\ell-1)} w_{mn}^{(\ell)} + b_n^{(\ell)}, \qquad j = 1, 2, \ldots, N_\ell,$$

we have

$$\frac{\partial n_n^{(\ell)}}{\partial w_{ij}^{(\ell)}} = \delta_{nj} a_i^{(\ell-1)}$$

$$\frac{\partial n_n^{(\ell)}}{\partial b_j^{(\ell)}} = \delta_{nj},$$

and so

$$\frac{\partial E}{\partial w_{ij}^{(\ell)}} = a_i^{(\ell-1)} \; s_j^{(\ell)}, \qquad \frac{\partial E}{\partial b_j^{(\ell)}} = s_j^{(\ell)}.$$

Therefore the updating rules for the weights and biases are (now we put back the dependency on the step index t)

$$w_{ij}^{(\ell)}(t+1) = w_{ij}^{(\ell)}(t) - \alpha \; a_i^{(\ell-1)}(t) \; s_j^{(\ell)}(t)$$

$$b_j^{(\ell)}(t+1) = b_j^{(\ell)}(t) - \alpha\, s_j^{(\ell)}(t),$$

In order to use these updating rules, we need to be able to compute the sensitivity vectors $\mathbf{s}^{(\ell)}$ for $\ell = 1, 2, \ldots, L-1$. From their definition

$$s_j^{(\ell)} = \frac{\partial E}{\partial n_j^{(\ell)}} \qquad j = 1, 2, \ldots, N_\ell,$$

we need to know how E depends on $n_j^{(\ell)}$. The key to computing these partial derivatives is to note that $n_j^{(\ell)}$ in turn depends on $n_i^{(\ell-1)}$ for $i = 1, 2, \ldots, N_{\ell-1}$, because the net input for layer ℓ depends on the activation of the previous layer, $\ell - 1$, which in turn depends on the net input for layer $\ell - 1$. Specifically

$$n_j^{(\ell)} = \sum_{i=1}^{N_{\ell-1}} a_i^{(\ell-1)} w_{ij}^{(\ell)} + b_j^{(\ell)} = \sum_{i=1}^{N_{\ell-1}} f^{(\ell-1)}(n_i^{(\ell-1)}) w_{ij}^{(\ell)} + b_j^{(\ell)}$$

for $j = 1, 2, \ldots, N_\ell$. Therefore we have for the sensitivity of layer $\ell - 1$

$$\begin{aligned} s_j^{(\ell-1)} &= \frac{\partial E}{\partial n_j^{(\ell-1)}} = \sum_{i=1}^{N_\ell} \frac{\partial E}{\partial n_i^{(\ell)}} \frac{\partial n_i^{(\ell)}}{\partial n_j^{(\ell-1)}} \\ &= \sum_{i=1}^{N_\ell} s_i^{(\ell)} \frac{\partial}{\partial n_j^{(\ell-1)}} \left(\sum_{m=1}^{N_{\ell-1}} f^{(\ell-1)}(n_m^{(\ell-1)}) w_{mi}^{(\ell)} + b_i^{(\ell)} \right) \\ &= \sum_{i=1}^{N_\ell} s_i^{(\ell)} \dot{f}^{(\ell-1)}(n_j^{(\ell-1)}) w_{ji}^{(\ell)} = \dot{f}^{(\ell-1)}(n_j^{(\ell-1)}) \sum_{i=1}^{N_\ell} w_{ji}^{(\ell)} s_i^{(\ell)}. \end{aligned}$$

Thus the sensitivity of a neuron in layer $\ell - 1$ depends on the sensitivities of all the neurons in layer ℓ. This is a recursion relation for the sensitivities of the network since the sensitivities of the last layer L is known. To find the activations or the net inputs for any given layer, we need to feed the input from the left of the network and proceed forward to the layer in question. However to find the sensitivities for any given layer, we need to start from the last layer and use the recursion relation going backward to the given layer. This is why the training algorithm is called backpropagation.

In summary, the backpropagation algorithm for training a multi-layer perceptron is

1. Set α. Initialize weights and biases.
2. For step $t = 1, 2, \ldots,$ repeat steps a-e until convergence.
   a Set $\mathbf{a}^{(0)} = \mathbf{x}(t)$ randomly picked from training set.
   b For $\ell = 1, 2, \ldots, L$, compute

$$\mathbf{n}^{(\ell)} = \mathbf{a}^{(\ell-1)}\mathbf{W}^{(\ell)} + \mathbf{b}^{(\ell)} \qquad \mathbf{a}^{(\ell)} = f^{(\ell)}(\mathbf{n}^{(\ell)}).$$

   c Compute for $n = 1, 2, \ldots, N_L$

$$s_n^{(L)} = 2\left(\mathbf{a}_n^{(L)} - \mathbf{t}_n(t)\right) \dot{f}^{(L)}(\mathbf{n}_n^{(L)}).$$

   d For $\ell = L-1, \ldots, 2, 1$ and $j = 1, 2, \ldots, N_\ell$, compute

$$s_j^{(\ell)} = \dot{f}^{(\ell)}(n_j^{(\ell)}) \sum_{i=1}^{N_{\ell+1}} w_{ji}^{(\ell+1)} s_i^{(\ell+1)}.$$

   e For $\ell = 1, 2, \ldots, L$, update

$$w_{ij}^{(\ell)}(t+1) = w_{ij}^{(\ell)}(t) - \alpha\, a_i^{(\ell-1)}(t)\, s_j^{(\ell)}(t),$$

$$b_j^{(\ell)}(t+1) = b_j^{(\ell)}(t) - \alpha\, s_j^{(\ell)}(t).$$

To compute the updates for the weights and biases, we need to find the activations and sensitivities for all the layers. To obtain the sensitivities, we also need $\dot{f}^{(\ell)}(n_j^{(\ell)})$. That means that in general we need to keep track of all the $n_j^{(\ell)}$ as well.

In NNs trained using the backpropagation algorithm, there are two functions often used as the transfer functions. One is the Log-Sigmoid function

$$f_{\text{logsig}}(x) = \frac{1}{1+e^{-x}}$$

which is differentiable and its value goes smoothly and monotonically between 0 and 1 for x around 0. The other is the hyperbolic tangent Sigmoid function

$$f_{\text{tansig}}(x) = \frac{1-e^{-x}}{1+e^{-x}} = \tanh(x/2)$$

which is also differentiable, but its value goes smoothly between -1 and 1 for x around 0.[2] It is easy to see that the first derivatives of

these functions are given in terms of the same functions alone:

$$\dot{f}_{\text{logsig}}(x) = f_{\text{logsig}}(x)\,[1 - f_{\text{logsig}}(x)]$$

$$\dot{f}_{\text{tansig}}(x) = \frac{1}{2}\,[1 + f_{\text{tansig}}(x)]\,[1 - f_{\text{tansig}}(x)]$$

Since $f^{(\ell)}(n_j^{(\ell)}) = a_j^{(\ell)}$, in implementing the NN on a computer, there is actually no need to keep track of $n_j^{(\ell)}$ at all (and thus saving memory).

4. Variations of the Basic Backpropagation Algorithm

Because the training process of a multilayer NN using the basic BP can be rather time consuming (days or even weeks of training time for many practical problems), a number of variations of the basic BP algorithm are available to accelerate the convergence.

4.1. Modified Target Values

When the output target vectors are of bipolar form and the transfer functions for the neurons in the output layer are hyperbolic tangent

Sigmoid functions, since the required output values of ± 1 occur at the asymptotes of the transfer functions, those values can never be reached. The inputs into these neurons need to have extremely large magnitudes. However this happens only when the weights eventually take on extremely large magnitudes also. Thus the convergence are typically very slow. The same is true if binary vectors are used together with the binary Sigmoid function.

One easy way out is to consider the net to have learned a particular training vector if the computed output values are within a specified tolerance of some modified desired values. Good modified values for the hyperbolic tangent Sigmoid function is ± 0.8. Use of modified values too close to 0 may actually prolong the convergence.

4.2. Other Transfer Functions

The arctangent function is sometimes used as transfer function in BP. It approaches its asymptotic values (saturates) more slowly than the hyperbolic tangent Sigmoid function. Scaled so that the function

value varies range between -1 and 1, the function is

$$f_{\arctan}(x) = \frac{2}{\pi} arctan(x),$$

with derivative

$$\dot{f}_{\arctan}(x) = \frac{2}{\pi} \frac{1}{1+x^2}.$$

For some applications, where saturation is not especially beneficial, a non-saturating transfer function may be used. One suitable example is

$$f_{\log}(x) = \begin{cases} \log(1+x) & \text{for } x \geq 0 \\ -\log(1+x) & \text{for } x < 0. \end{cases}$$

Note that it is continuous everywhere, and its derivative is given by

$$\dot{f}_{\log}(x) = \begin{cases} \frac{1}{1+x} & \text{for } x > 0 \\ \frac{1}{1-x} & \text{for } x < 0 \end{cases}$$

and is also continuous everywhere, including $x = 0$.

Radial basis functions are also used as transfer functions in BP. These functions have non-negative responses that are localized around particular input values. A common example is the Gaussian function,

$$f_{\text{gaussian}}(x) = \exp(-x^2)$$

which is localized around $x = 0$. Its derivative is

$$\dot{f}_{\text{gaussian}}(x) = -2x \exp(-x^2) = -2x f_{\text{gaussian}}(x).$$

4.3. Momentum

Convergence is sometimes faster if a momentum term is added to the weight update formulas. In the simplest form of BP with momentum, the momentum term is proportional to the change in the weights in the previous step:

$$\Delta w_{ij}^{(\ell)}(t+1) = \alpha \; a_i^{(\ell-1)}(t) \; s_j^{(\ell)}(t) + \mu \Delta w_{ij}^{(\ell)}(t),$$

where the momentum parameter μ lies in the range (0 1). Therefore updates of the weights proceed in a combination of the current gradient direction and that of the previous gradient. Momentum allows

the net to make reasonably large weight adjustments as long as the corrections are in the same general direction for several successive input patterns. It also prevents a large response to the error from any one single training pattern.

4.4. Batch Updating

In some cases it is advantageous to accumulate the weight correction terms for several patterns (or even an entire epoch if there are not too many patterns) and make a single weight adjustment for each weight rather than updating the weights after each pattern is presented. This procedure has a smoothing effect on the correction terms somewhat similar to the use of momentum.

4.5. Variable Learning Rates

Just like the original Delta rule for a single layer NN, the use of variable learning rates may accelerate the convergence. Each weight may even have its own learning rate. The learning rates may vary adaptively with time as training progress.

4.6. Adaptive Slope

An adjustable parameter can be introduced into the transfer function to control the slope of the function. For example instead of the hyperbolic tangent Sigmoid function

$$f_{\text{tansig}}(x) = \frac{1 - e^{-x}}{1 + e^{-x}} = \tanh(x/2)$$

we can use the hyperbolic tangent Sigmoid function

$$f_{\text{tansig}}(x) = \frac{1 - e^{-\sigma x}}{1 + e^{-\sigma x}} = \tanh(\sigma x/2),$$

where $\sigma(> 0)$ controls the magnitude of the slope. The larger σ is, the larger is the derivative of the transfer function. Each neuron can have its own transfer function with a different slope parameter.

Updating formulas for the weights, biases, and slope parameters can be derived. Although the net input into neuron $X_j^{(\ell)}$ is still given

by

$$n_j^{(\ell)} = \sum_{i=1}^{N_{\ell-1}} a_i^{(\ell-1)} w_{ij}^{(\ell)} + b_j^{(\ell)}, \qquad j = 1, 2, \ldots, N_\ell,$$

its activation is now given by

$$a_j^{(\ell)} = f^{(\ell)}(\sigma_j^{(\ell)} n_j^{(\ell)}) = f^{(\ell)} \left(\sigma_j^{(\ell)} \left[\sum_{i=1}^{N_{\ell-1}} a_i^{(\ell-1)} w_{ij}^{(\ell)} + b_j^{(\ell)} \right] \right),$$

where $\sigma_j^{(\ell)}$ is the slope parameter for neuron $X_j^{(\ell)}$.

The sensitivities for neurons in the output layer are given by

$$s_n^{(L)} = 2 \left(a_n^{(L)} - t_n(t) \right) \dot{f}^{(L)}(\sigma_n^{(L)} n_n^{(L)}), \qquad n = 1, 2, \ldots, N_L.$$

The sensitivities of all the other layers can then be obtained from the following recursion relation

$$s_j^{(\ell-1)} = \dot{f}^{(\ell-1)}(n_j^{(\ell-1)}) \sum_{i=1}^{N_\ell} w_{ji}^{(\ell)} s_i^{(\ell)} \sigma_i^{(\ell)}.$$

The updating rules for the weights, biases, and slope parameters are given by

$$\Delta w_{ij}^{(\ell)}(t) = -\alpha\, a_i^{(\ell-1)}(t)\, s_j^{(\ell)}(t)\, \sigma_j^{(\ell)}(t)$$

$$\Delta b_j^{(\ell)}(t) = -\alpha\, s_j^{(\ell)}(t)\, \sigma_j^{(\ell)}(t),$$

$$\Delta \sigma_j^{(\ell)}(t) = -\alpha \frac{\partial E}{\partial \sigma_j^{(\ell)}(t)} = \alpha\, s_j^{(\ell)}(t)\ \ n_j^{(\ell)}(t).$$

The slope parameters are typically initialized to 1 unless one has some ideas of what their proper values are supposed to be. The use of adjustable slopes usually improves the convergence, however at the expense of slightly more complicated computation per step.

5. Multilayer NN as Universal Approximations

One use of a NN is to approximate a continuous mapping. It can be shown that a feedforward NN with an input layer, a hidden layer, and an output layer can represent any continuous function exactly. This is known as the Kolmogorov mapping NN existence theorem.

With the use of appropriate transfer functions and a sufficiently large number of hidden layers, a NN can approximate both a function and its derivative. This is useful for applications such as a robot learning smooth movements.

References

[1] See Chapter 6 in Laurene Fausett, "Fundamentals of Neural Networks - Architectures, Algorithms, and Applications", Prentice Hall, 1994.

[2] The hyperbolic tangent Sigmoid function may be good for input vectors that have continuous real-values, and the target vectors have components in the range $[-1\ 1]$. We can easily change this range to any other finite interval, such as $[a\ b]$, by defining a new transfer function

$$g(x) = \gamma f_{\text{logsig}}(x) + \eta$$

which is related to f_{logsig} via a linear transformation. Since we want $g = a$ when $f_{\text{logsig}} = -1$, therefore we have $-\gamma + \eta = a$. Also we want $g = b$ when $f_{\text{logsig}} = 1$, therefore we have $\gamma + \eta = b$. Solving for a and b gives

$$\eta = \frac{a+b}{2}, \qquad \gamma = \frac{b-a}{2}.$$

Therefore

$$g(x) = \frac{1}{2}\left((b-a) f_{\text{logsig}}(x) + a + b\right).$$

16