The Third AFSS(1998). 113 - 117

Self-Relaxation for Multilayer Perceptron

Cheng-Yuan Liou and Hwann-Tzong Chen

Dept. of Computer Science and Information Engineering, National Taiwan University, Taipei, Taiwan, R.O.C.
Tel: 8862-23625336 ext 515, Fax: 8862-23628167, Email: cyliou@csie.ntu.edu.tw.

Abstract

We propose a way to show the inherent learning complexity for the multilayer perceptron. We display the solution space and the error surfaces on the input space of a single neuron with two inputs. The evolution of its weights will follow one of the two error surfaces. We observe that when we use the back-propagation (BP) learning algorithm [1], the wights can not jump to the lower error surface due to the implicit continuity constraint on the changes of weights. The self-relaxation approach is to explicitly find out the best combination of all neurons' two error surfaces. The time complexity of training a multilayer perceptron by self-relaxation is exponential to the number of neurons.
Keywords: Back-propagation algorithm, Multilayer perceptron, Neural network.

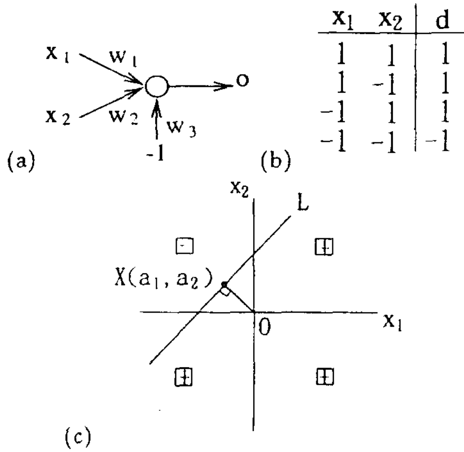
1.Introduction

It has been studied that the inherent complextiy of training a multilayer perceptron is NP-complete [2][3][4]. This suggests that any training algorithm should display such fact in one way or another. We examine the existing algorithm with a new viewpoint and develop an algorithm with such complexity explicitly.

The error surface (usually the energy surface) and the solution space for the delta learning rule were commonly displayed on weight space. Using this rule to reduce the error, the weights will be adjusted in the negative error gradient direction or the down slope direction in the weight space. The disadvantage of this kind display is that we can hardly realize (see) the movements and locations of the decision hyperplanes through observing the evolution weights. Besides, some important properties like the positive side of each decision hyperplane are also invisible in the weight space. Instead of the common display, we display the error surfaces and the solution space on input space. This display will provide the learning paths of the decision hyperplanes on the input space. Since a hyperplane has two sides, we can obtain two error surfaces according to which side of the hyperplane is positive. When we initialize the weights to form a decision hyperplane with a wrong positive side, the reduction of the error using the BP algorithm will follow the same surface which contains the initial error to keep continuity in the changes of weights. This kind of reduction will take much more evolution to detour to the solution. There is little chance to jump into the other lower error surface because the continuity requirement made on the updation of the weights. Thus, if we add the jumping capability called self-relaxation to the BP algorithm, the error can drop down immediately in the relaxation. This jumping can be achieve by reversing the signs of all weights of a neuron. We interrupt the training process by inserting this relaxation whenever the training errors show little improvement to see if there exists a better error surface to evolve. If there exists a lower error surface, we reverse the signs of a neuron's weights accordingly to switch the evolution in this surface. We illustrate the whole idea for a single neuron with two inputs in the next section and extend this idea to a multilayer perceptron in section 3.

2.Single Neuron Example

We use a single neuron example to illustrate the main idea of the relaxation. Consider a neuron with two inputs $\{x_1, x_2\}$, plus a fixed input -1 for threshold shown in Fig. 1(a). The logic we want to implement is shown in Fig. 1(b). The four input patterns are $\{x_1^{(1)}, x_2^{(1)}\} = \{1, 1\}$, $\{x_1^{(2)}, x_2^{(2)}\} = \{1, -1\}$, $\{x_1^{(3)}, x_2^{(3)}\} = \{-1, 1\}$, and $\{x_1^{(4)}, x_2^{(4)}\} = \{-1, -1\}$. The logic we want to implement is $\neg(\neg x_1 \wedge x_2)$. An arbitrary line in this figure is set as $L : -2x_1 + 2x_2 - 1 = 0$. Its corresponding two decision lines are $S^{(1)} = \{w_1 = -2, w_2 = 2, w_3 = 1\}$ or $S^{(2)} = \{w_1 = 2, w_2 = -2, w_3 = -1\}$. $S^{(2)}$ is obtained by reversing the signs of all $\{w_i\}$ in $S^{(1)}$. Both $S^{(1)}$ and $S^{(2)}$ are two possible decision lines corresponding to the line L. In this example, $S^{(1)}$ is not the proper solution because the positive side of this decision line is opposite to the desired. The better decision line of the two $\{S^{(1)}, S^{(2)}\}$ is the one gives smaller error. With the weights being initialized as random values, we can not expect that a decision line has a correct or better positive side.



x_1	x_2	d
1	1	1
1	-1	1
-1	1	1
-1	-1	-1

Figure 1:

Though the decision line can be determined by the three values, $\{w_1, w_2, w_3\}$, we try to represent a line with a point on the input space with the coordinates of x_1 and x_2. As shown in Fig. 1(c), the line L can be represented by a perpendicular point $X(a_1, a_2)$ where a_1 and a_2 are obtained by solving the following equations.

$$\frac{w_1}{w_3} = \frac{a_1}{a_1^2 + a_2^2}, \qquad \frac{w_2}{w_3} = \frac{a_2}{a_1^2 + a_2^2}.$$

The corresponding decision lines which pass (a_1, a_2) and perpendicular to $\overline{XO}$ is $C(\frac{a_1}{a_1^2+a_2^2}x_1 + \frac{a_2}{a_1^2+a_2^2}x_2 - 1) = 0$. Without loss of generality we set $C = 1$ and $C = -1$ for these two decision lines seperately. The magnitude of C will not affect the output error. This is because the hard-limited function will give $+1$ or -1 no matter how large the magnitude is. Both decision lines have the same a_1 and a_2 values. So, there are two errors $E_X^{(1)}$ and $E_X^{(2)}$ for all four input patterns on the point X and they can be calculated as

$$\begin{aligned} E_X^{(i)} &= \frac{1}{2}\sum_{p=1}^{4}(d_p - o_p^{(i)})^2 \qquad i = 1 \sim 2 \\ o_p^{(i)} &= \sigma(net_p^{(i)}) \\ &= 2(\frac{1}{1 + exp(-net_p^{(i)})} - \frac{1}{2}), \qquad p = 1 \sim 4. \end{aligned}$$

Where the variable net_p^i of the sigmoid function $\sigma(.)$ has two possible values for all patterns.

$$\begin{aligned} net_p^{(1)} &= \frac{a_1}{a_1^2 + a_2^2}x_1^{(p)} + \frac{a_2}{a_1^2 + a_2^2}x_2^{(p)} - 1 \\ & \qquad (C = 1) \\ net_p^{(2)} &= -net_p^{(1)} = -\frac{a_1}{a_1^2 + a_2^2}x_1^{(p)} - \frac{a_2}{a_1^2 + a_2^2}x_2^{(p)} + 1 \\ & \qquad (C = -1) \end{aligned}$$

We calculate $E_X^{(i)}$ at each point and plot the two error surfaces on input space. The two error surfaces on input space are shown in Fig. 2(a) and 2(b) seperately.

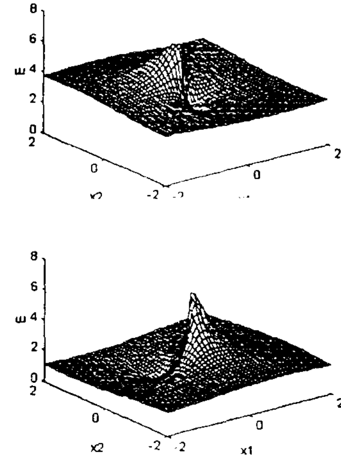


Figure 2: The two error surfaces on input space where $C = 1$ in (a) and $C = -1$ in (b).

The *actual error surface* can be obtained by replacing the sigmoid function with the hard-limited activation function

$$\sigma(net_p^{(i)}) \Longrightarrow sgn(net_p^{(i)}) = \begin{cases} +1 & , \quad net_p^{(i)} > 0 \\ -1 & , \quad net_p^{(i)} < 0 \end{cases}$$

We plot these two actual error surfaces in Fig. 3(a) and (b). The solution space where $E_X^{(2)} = 0$ on input space is shown in Fig. 4. Each point in the shaded area represents a decision line. Note that the minima of error in Fig. 2(a) and Fig. 3(a) are not zero. There exists no solution in the surface with $C = 1$. This is because the positive side of the decision line is wrong.

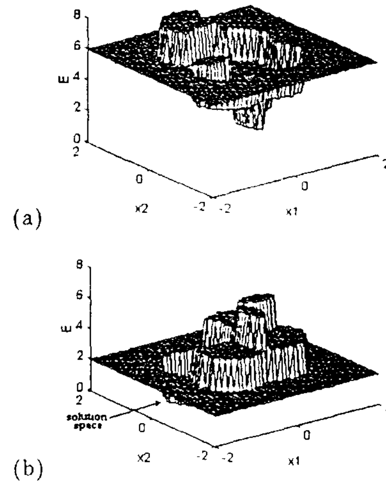


Figure 3: The two actual error surfaces on input space where $C = 1$ in (a) and $C = -1$ in (b).

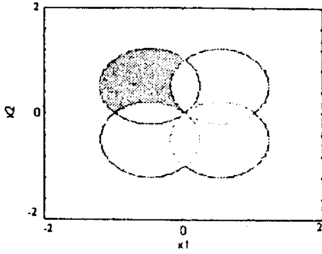


Figure 4: The shaded area is the solution space where $E_X^{(2)} = 0$ on input space.

Fig. 5(a) shows a learning path of the decision line using the BP algorithm. We see that this path will follow down and keep in one of the error surface as marked by $C = 1$ Fig. 2(a), where it starts the evolution. This path will not jump to the lower error surface unless it closes to the origin. Once a weight's sign is reversed, the path will switch to the opposite place and evolve from there. This means when d_p and o_p have different signs, o_p will pass zero ($o_p = 0$) in order to reverse its sign to match with d_p. When we reverse the weights' signs, o_p will directly reverse its signs to match the d_p and skip the detour. Normally the path will continuously follow the error surface to reach a continuous place where they can switch to the other lower error surface. This will spend much more iterations to detour to the solution space. The reason is that the movements of the hyperplanes (adjustments of weights) must satify the continuity requirement with which the BP algorithm is derived. Fig. 5(b) shows a path which starts with a good positive side.

Now we can modify the training algorithm to make it have the jumping capability. We interrupt the training algorithm whenever it shows little improvement and relax the neuron to move to a lower actual error surface. We call this step the self-relaxation step. In this step, both two actual error values corresponding to the current point (or line) are checked. We choose the decision line with smaller error value and set the positive side of the line accordingly. We will reverse the signs of the neuron's weights if the positive side of the line is altered.

The results of the training processes using regular and modified algorithm are shown in Fig. 5(a), Fig. 6 and Fig. 7(a)(b). The weights are initialized as $\{w_1 = -1.0, w_2 = 1.5, w_3 = 2.0\}$ and the learning rate η is 0.3. Note that in Fig. 7(b) the error drops down in the fifth iteration where the relaxation occurs. The performance of modified algorithm is much better than the regular one.

3.Multilayer perceptron with self-relaxation

We extend the above idea to the multilayer perceptron. An example network is shown in Fig. 8. Let N be the

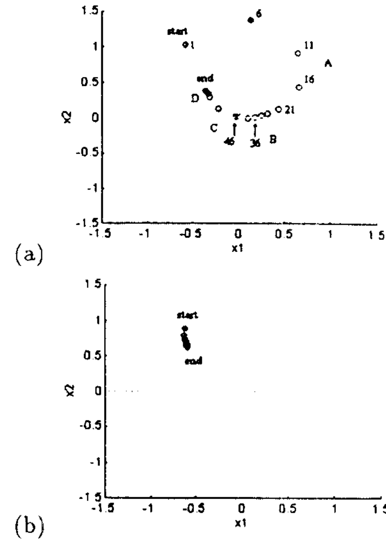


Figure 5: The evolution paths of the decision lines using BP algorithm. The weights are initialized as $\{w_1 = -1.0, w_2 = 1.5, w_3 = 2.0\}$ in (a) and $\{w_1 = 1.0, w_2 = -1.5, w_3 = -2.0\}$ in (b).

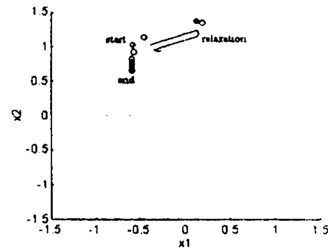


Figure 6: The evolution path of the decison line using self-relaxation. In both 5(a) and 6, the weights are initialized as $\{w_1 = -1.0, w_2 = 1.5, w_3 = 2.0\}$.

total number of neurons in the network. In this example $N = 14$.

Let $\{S_n^{(i)}, i = 1,2\}$ be the two possible decision hyperplanes for the current hyperplane of the nth neuron. One decision hyperplane can be obtained by reversing the signs of all weights of the other decision hyperplane. Let $\{S_n^{(i)}, i = 1,2\}$ be the two candidate states for the nth neuron. There are 2^N kinds of combinations of such reverse states for the network. We denote $\{\mathbf{ST}_j, j = 1 \sim 2^N\}$ to be the all combinations of these reverse states. During the relaxation we count the total number of error bits in the output layer corresponding to each $\mathbf{ST}_j$ for all inputs and select the state, $\mathbf{ST}_{min}$, with minimum number of error bits. We recommend to design special architecture to reduce the cost for large N. When there are more than one reverse states which give the same number of error bits, all of them will be evolved. We interrupt the BP training

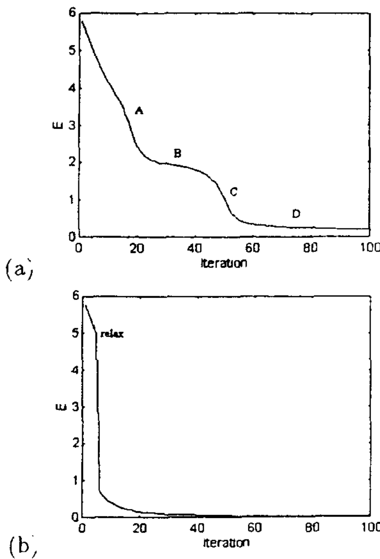


Figure 7: The learning curves. (a) Use BP algorithm for the training in Fig. 5(a). (b) Use modified algorithm (self-relaxation) for training.

process with this relaxation whenever the training shows little improvement.

We use the multilayer perceptron as shown in Fig. 8(a) to learn the patterns which are shown in Fig. 8(b). This perceptron has six neurons in the first hidden layer, four neurons in the second hidden layer and four neurons in the output layer. Each neuron is also connected to a fixed input with the value -1. The training patterns are ten capital letters $\{A \cdots J\}$ in the size of 8×12 bits. The desired outputs for these patterns are $\{0000, 0001, 0010, 0011, 0100, 0101, 0110, 0111, 1000, 1001\}$. For each iteration of the training process, we add some noises to the input patterns by changing 10 of the 96 bits of each pattern randomly. We use the BP algorithm and the modified algorithm to train the network. The weights are initialized randomly from -2 to 2 in both cases. Note the $E(\mathbf{ST}_j)$ is the total actual output error where we present all training patterns to the network. This error is obtained by reversing the neurons' weights according to $\mathbf{ST}_j$. The learning curves of both algorithms are shown in Fig. 9. By observing the performance of these two algorithms, we find that self-relaxation can help for getting lower error and achieving better solution when the training process is interrupted in suitable situations. From experience, better results can be obtained by applying the relaxation intensively at the begining stage of evolution.

When N is large, it is time-consuming to relax the network to find the reverse state $\mathbf{ST}_{min}$ which gives absolute minima among all 2^N combinations. Actually we search local minima among neighboring vertices in this N-dimensional cube in a similar way as the simplex method. We may partition the relaxation into M time intervals. For

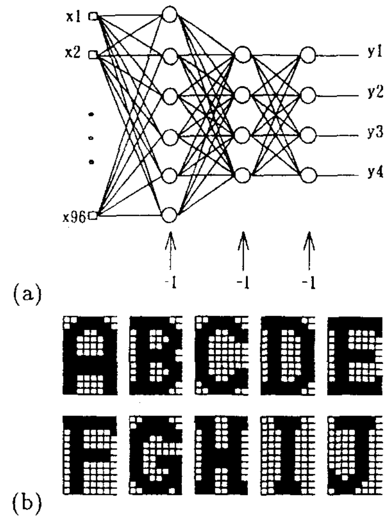


Figure 8:

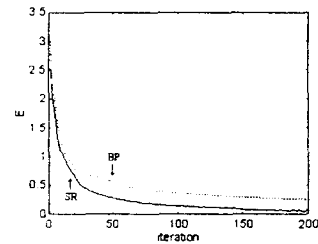


Figure 9: Typical training curves for the BP algorithm and the modified algorithm with self-relaxation.

each time interval, we relax (reverse) one neuron during this interval where reversing this neuron will give minimum error among those using all N neurons and then proceed to the next interval.

Finally, we provide two algorithms by descending the hyperplane down the error surface in input space as in Fig. 2 and 3. The error function in input space is the same as before, E_X. The training formulas for the single neuron in Fig. 1(a) in the input space are as follows.

$$\frac{\partial E_X}{\partial a_1} = \sum_{p=1}^{4}(d_p - o_p)(\frac{1}{2}(1-o_p^2))(\frac{-a_1^2+a_2^2}{(a_1^2+a_2^2)^2}x_1 + \frac{-2a_1a_2}{(a_1^2+a_2^2)^2}x_2)$$

$$\frac{\partial E_X}{\partial a_2} = \sum_{p=1}^{4}(d_p - o_p)(\frac{1}{2}(1-o_p^2))(\frac{-2a_1a_2}{(a_1^2+a_2^2)^2}x_1 + \frac{a_1^2-a_2^2}{(a_1^2+a_2^2)^2}x_2)$$

The adjustments of the point $X(a_1, a_2)$ are as follows.

$$\delta a_1 = -\eta \frac{\partial E_X}{\partial a_1}$$
$$\delta a_2 = -\eta \frac{\partial E_X}{\partial a_2}$$

As for multilayer perceptrons we provide the following learning algorithm for reference.

Given P training pairs
$$\{\mathbf{z}_1, \mathbf{d}_1, \mathbf{z}_2, \mathbf{d}_2, ..., \mathbf{z}_p, \mathbf{d}_p\},$$
where $\mathbf{z}_i$ is $((I+1)\times 1)$, $\mathbf{d}_i$ is $(K \times 1)$, and $i = 1, 2, ..., P$. Note that the $(I+1)$'th component of each $\mathbf{z}_i$ is of value -1 since input vectors have been augmented. Size J of the hidden layer having outputs $\mathbf{y}$ is selected. Note that the $(J+1)$'th component of $\mathbf{y}$ is of value -1, since hidden layer outputs have also been augmented; $\mathbf{y}$ is $(J \times 1)$ and $\mathbf{o}$ is $(K \times 1)$. The weight of every augmented component is assigned to the value 1 and is not changed during the training process.

Step 1: $\eta > 0$, E_{max} chosen.

Weights $\mathbf{W}$ and $\mathbf{V}$ are initialized at small random values; $\mathbf{W}$ is $(K \times (J+1))$, $\mathbf{V}$ is $(J \times (I+1))$. The entries of th $(J+1)$'th column of $\mathbf{W}$ and the $(I+1)$'th column of $\mathbf{V}$ are all assigned to the value 1.
$$q \leftarrow 1, p \leftarrow 1, E \leftarrow 0$$

Step 2: Training step starts here.

Input is presented and the layers' outputs computed:
$$\mathbf{z} \leftarrow \mathbf{z}_p, \mathbf{d} \leftarrow \mathbf{d}_p$$
$$y_j \leftarrow f(\mathbf{v}_j^t \mathbf{z}), \text{ for } j = 1, 2, ..., J+1$$
where $\mathbf{v}_j$, a column vector, is the j'th row of $\mathbf{V}$, and
$$o_k \leftarrow f(\mathbf{w}_k^t \mathbf{y}), \text{ for } k = 1, 2, ..., K$$
where $\mathbf{w}_k$, a column vector, is the k'th row of $\mathbf{W}$.

Step 3: Error value is computed:
$$E \leftarrow \frac{1}{2}(d_k - o_k)^2 + E, \text{ for } k = 1, 2, ..., K$$

Step 4: Error signal vectors δ_o and δ_y of both layers are computed. Vector δ_o is $(K \times 1)$, δ_y is $(J \times 1)$.

The error signal terms of the output layer in this step are
$$\delta_{ok} = \frac{1}{2}(d_k - o_k)(1 - o_k^2), \text{ for } k = 1, 2, ..., K$$

The error signal terms of the hidden layer in this step are
$$\delta_{yj} = \frac{1}{2}(1 - y_j^2) \sum_{k=1}^{K} \delta_{ok} \frac{a_{k1}}{a_{k_1}^2 + a_{k_2}^2 + \cdots + a_{k_J}^2}, \text{ for } j = 1, 2, ..., J$$

Step 5: Output layer weights are adjusted:
$$a_{kj} \leftarrow a_{kj} + \eta \delta_{ok} \mathbf{A}^t \mathbf{y}$$

$$\mathbf{A} = \begin{bmatrix} \frac{-2a_{k1}a_{kj}}{(a_{k1}^2 + a_{k2}^2 + \cdots + a_{kJ}^2)^2} \\ \frac{-2a_{k2}a_{kj}}{(a_{k1}^2 + a_{k2}^2 + \cdots + a_{kJ}^2)^2} \\ \vdots \\ \frac{a_{k1}^2 + a_{k2}^2 + \cdots - a_{kj}^2 + \cdots + a_{kJ}^2}{(a_{k1}^2 + a_{k2}^2 + \cdots + a_{kJ})^2} \\ \vdots \\ \frac{-2a_{kJ}a_{kj}}{(a_{k1}^2 + a_{k2}^2 + \cdots + a_{kJ}^2)^2} \end{bmatrix}$$

for $k = 1, 2, ..., K$ and $j = 1, 2, ..., J$

Step 6: Hidden layer weights are adjusted:
$$b_j i \leftarrow b_{kj} + \eta \delta_{yj} \mathbf{B}^t \mathbf{z}$$

$$\mathbf{B} = \begin{bmatrix} \frac{-2b_{j1}b_{ji}}{(b_{j1}^2 + b_{j2}^2 + \cdots + b_{jI}^2)^2} \\ \frac{-2b_{j2}b_{ji}}{(b_{j1}^2 + b_{j2}^2 + \cdots + b_{jI}^2)^2} \\ \vdots \\ \frac{b_{j1}^2 + b_{j2}^2 + \cdots - b_{ji}^2 + \cdots + b_{jI}^2}{(b_{j1}^2 + b_{j2}^2 + \cdots + b_{jI})^2} \\ \vdots \\ \frac{-2b_{jI}b_{ji}}{(b_{j1}^2 + b_{j2}^2 + \cdots + b_{jI}^2)^2} \end{bmatrix}$$

for $j = 1, 2, ..., J$ and $i = 1, 2, ..., I$

Step 7: If $p < P$ then $p \leftarrow p+1$, $q \leftarrow q+1$, and go to Step 2; otherwise, go to Step 8.

Step 8: The training cycle is completed.

For $E < E_{max}$ terminate the training session. Output weights $\mathbf{W}$, $\mathbf{V}$, q, and E.

If $E > E_{max}$, the $E \leftarrow 0$, $p \leftarrow 1$, and initiate the new training cycle by going to Step 2.

Reference

1. D.E. Rumelhart, G.E. Hinton, and R.J. Williams "Learning representations by back-propagating errors", Nature (London), **323**, 533-536, 1986.
2. A.L. Blum and R.L. Rivest "Training a 3-Node Neural Network is NP-Complete", Neural Networks, Vol. 5, 117-127, 1992.
3. J. Šíma "Back-propagation is not Efficient", Neural Networks, Vol. 9, 1017-1023, 1996.
4. D.H. Wolpert and W.G. Macready "No free lunch theorems for search", Tech. Rep. No. SFI-TR-95-02-010, Santa Fe Institute