Project Study Neural Networks

On the dynamic behaviour of back propagation networks

October 28, 1994

By Alex Wulms

St. Nr. 8949670

1. Introduction

As neural networks are often used to solve problems which are not completely understood or which are hard to solve with the more traditional AI techniques, it is important to know how well a neural network can learn to solve such a problem.

One category of problems that can be solved with a neural network, is the category of association problems; each input of the problem space has to be associated with a correct output. These association problems are often solved with so-called back propagation (BP) networks. A BP network will be trained with a set of inputs of the problem space and the correct outputs that are corresponding to these inputs. During the training session, the weights of the network will converge to a point in the network's weight space, in which the problem examples will be known to the network; when the network has reached this point in its state space, it can give the correct output for each example input. However, due to the nature of the network, it can not always learn the problem exactly; the output produced by the network when presented with an input problem, will sometimes only be an approximation of the exact output. Therefore it is usually hard to tell when the network has learned the problem.

To understand more of the problem of determining when the BP network has learned the association problem, the dynamic behaviour of the neural network during its training should be studied. To investigate this dynamic behaviour of a BP network, a tool has been developed under OSF/Motif that can trace the internal state of a BP network during a training session. This report gives the results of the investigations done with this tool.

This report has the following structure. The following section gives a description of BP networks and which variant has been used in the tool. Section 3 will describe the properties of BP networks and the problems which can arise with them. In section 4 will be explained which parameters can be set to investigate the network and in section 5 the various ways to train the network with the tool will be given. Section 6 will describe which properties of the network can be viewed with the tool and some traces made with the tool will be analysed. In section 7 some conclusions will be given and some topics for further research can be found in section 8. The report will end with an appendix showing the exact solution for the ID1 function learned by a one layer network.

2. BP networks

BP networks are in fact so-called feed forward networks with a special learning algorithm: the error BP algorithm.

In general, a neural network consists of some units with (directed) connections between them. It is possible to associate a weight with each of these connections. For a BP network, these weights will be updated during the training session with the aid of the learning rule.

In a feed forward network, the units are placed in a number of layers with only directed connections from the units in a layer i to the units in the layer

i+1. A feed forward network consists of minimal two layers: an **input layer** and an **output layer**. When the network consists of more layers, all the layers between the input layer and the output layer, will be called **hidden layers** with hidden units. There exist two different conventions to count the number of layers in a feed forward network: one convention is to count the input layer, the hidden layers and the output layer while in the other convention only the hidden layers and the output layer will be counted. The trend is moving towards counting only the hidden layers and the output layer, thus only counting the layers that perform computations. Therefore, this convention will also be used in this report. So the most simple feed forward network is in fact a one layer network.

In Figure 1 the naming convention used in the formulas in this report and in the source of the tool, will be shown.

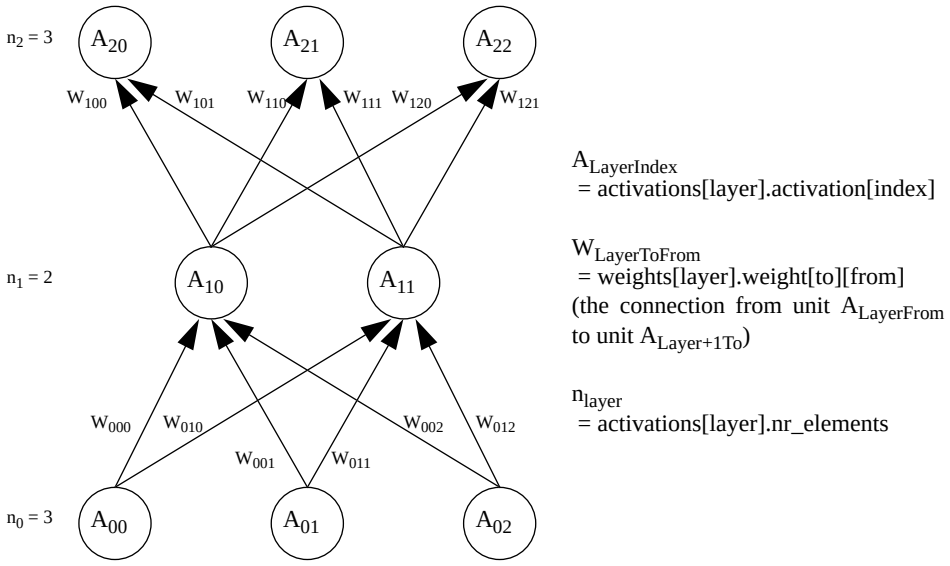


Figure 1. A two layer feed forward network, showing the naming convention used in this report and in the tool

The problem description will be inputted to the network by binding the right activation values to the input units of the network; it will be bounded to the units $A_{01}..A_{0n}$. The input unit A_{00} has a special function which will be discussed later. In the following steps, the feed forward network will feed these activation values forward to the following layers as follows:

Let A_{ij} denote the activation of unit j in layer i, then the following formula can be used to calculate $A_{i+1,k}$:

$$A_{i+1,k} = g(h_{i+1,k})$$

with $h_{i+1,k}$ the **netinput** for unit $A_{i+1,k}$:

$$h_{i+1,k} = \sum_{j=0}^{n_i} W_{ikj} A_{ij}$$

and with g some (non-linear) **activation function**. It is possible to choose any continuous differentiable function for g. A very convenient choice is to use a sigmoid function. This has also be done in the tool: the tool uses the following

sigmoid function to calculate the activations:

$$g(h) = \frac{1}{1 + e^{-h}}$$

This activation function restricts the activation values to a range of (0...1).

In fact, when calculating the netinput h_{ij} of a unit A_{ij}, one also needs to subtract a **threshold value** from the weighted sum over the activations; this threshold value sets a minimum value for the weighted sum over the activations to make the netinput pass the zero boundary. When using for example a binary threshold function for the activation function g (g(h) = 1 for h > 0, 0 otherwise), the threshold value determines when the unit becomes active. On the other hand, when using a non-linear function, the threshold value is merely a **bias value,** shifting the input to the activation function up or down. A very convenient way to implement a bias, is to clamp the activation of the units A_{i0} to a fixed value of 1: in this manner, the value $A_{i0}*W_{ik0}$ forms a bias value for unit $A_{i+1,k}$. The advantage of this method is that the bias values can also be trained, as the bias value is determined by the weight W_{ik0}, which will be updated by the BP learning mechanism.

A BP network can be trained in the following manner:
- first calculate the current output for an example input by feeding the input forwards through the network.
- calculate for each output unit, the difference between the desired output value and the current output value, which will be called the error for that unit.
- propagate the error values back to the underlaying layers.
- update the weights W_{ikj} according to these error values with the so-called gradient descent rule; the total error of the network can be expressed as an **error measure** or **cost function** E[**w**], with **w** the weight vector, of which the minimum has to be found during the training session. This minimum can be approached by adapting the weights proportional to the gradient of this cost function.

One very convenient choice for the cost functions is the following one:

$$E[\boldsymbol{w}] = \frac{1}{2}\sum_{\mu i}\left[\zeta_i^{\mu} - A_i^{\mu}\right]^2$$

with μ the patterns, ζ_i^{μ} the desired output for pattern μ and with A_i^{μ} the current output.

The general form of the gradient descent rule is as follows:

$$\Delta W_{ikj} = -\alpha\frac{\partial E}{\partial W_{ikj}}$$

with ΔW_{ikj} the change of the weight W_{ikj}, α the **learning rate** and E the cost function.

When using the above mentioned cost function, this will lead to the following formula for updating the weights with the gradient descent rule (see [1]):

$$\Delta W_{ikj} = \alpha \sum_{patterns} \delta_{i+1,k} A_{ij}$$

with A_{ij} the activation at the input end of the weight W_{ikj} and with $\delta_{i+1,k}$ a value derived from the error in the activation at the output end of the weight.

For an output unit A_{ij}, δ_{ij} will be calculated from the error in the output activation:

$$\delta_{ij} = g'(h_{ij}) [\zeta_j - A_{ij}]$$

with g' the derivative of the activation function g and with ζ_j the desired output value on unit A_{ij}.

For a hidden unit A_{ij}, δ_{ij} will be calculated from the $\delta_{i+1,k}$ of the units in the next layer:

$$\delta_{ij} = g'(h_{ij}) \sum_k W_{ikj} \delta_{i+1,k}$$

As the errors are propagated back through the network in this manner, this way of updating the weights, is called error back propagation.

For calculating ΔW_{ikj} one needs to take the sum over the errors of all the patterns that are used to train the network. However, in stead of presenting all the patterns and calculating the ΔW_{ikj} afterwards (the batch method), it is also possible to train the network in an incremental manner: present one pattern to the network and update the weights immediately, then show the next pattern and update the weights again, and so on. This incremental method turns out to have several advantages over the batch method: the network converges faster in most cases, it will not get stuck as easily in local minima and one does not need the additional storage which is necessary in the batch mode for storing the errors. For these reasons, the incremental method has been used in the tool.

3. Properties and problems of BP

When a BP network is trained, the information which the network has learned is stored in the weights between the units. During the training session, the weights will converge to such a point in the weight space, that the cost function will be in a local minimum. Thus, training the network in the above mentioned manner will locally minimize the cost function in an incremental manner.

However, there are some problems with BP:

1) The cost function is no strict rising function, it can have many local minima and maxima. When the network is somewhere near a local minimum, it can get stuck in such a local minimum and never reach the global minimum.
2) The surface of the cost function can be very flat in some areas, which leads to a very slow convergence in such an area.
3) As updating the weights is done incrementally, the network can oscillate round a certain point in the weight space: the effect of updating the weights for one pattern can have the opposite effect of updating the weights for another pattern. When this happens, the network will oscillate between these points in the weight space without converging much towards a minimum.

One way to solve these problems, is to add a **momentum parameter** β to the movement through the weight space. Informally spoken, one can compare

the movement through the weight space with the movement of a ball rolling over the cost surface with a certain momentum (see [2]). When β is set to zero, this ball will stop rolling after each update, it loses all of its speed. However, when β is set to one, the ball will keep all of its speed after one update, so it is rolling over the cost surface without experiencing any friction of the cost surface; it is experiencing a friction free motion through the weight space.

The global effect of this momentum is that the network changes in the direction of the average movement it would make through the weight space. To implement the momentum, one has to give a contribution from the previous weight update to each new weight update. This leads to the following formula for updating the weights:

$$\Delta W_{ikj}(t+1) = \alpha \sum_{patterns} \delta_{i+1,k} A_{ij} + \beta \Delta W_{ikj}(t)$$

The BP network in the tool makes use of this weight update rule with a momentum parameter. The purpose of the tool is to trace the movement of the network through the weight space during training to gain more insight in the effects of using the above mentioned update rule with an incremental update algorithm.

In this study we want to investigate the behaviour of a neural network during the training sessions near a local or a global minimum. This behaviour may depend on both the learning rate α and the momentum parameter β. Also the training order can influence the behaviour. First we will analyse the behaviour of a one layer network to see if there is any relationship between the learning parameters and the path the network walks through the weight space during the training session and to analyse the behaviour of the network when the global minimum has been reached. After this, we will also make a start with the analysis of a two layer network learning the XOR function.

Furthermore, we want to find a way to determine from the type of (local) minimum whether the network has reached a good or a bad approximation of the function to be trained when such a (local) minimum has been reached.

In this study we will only take a look at the influence of the learning rate α and the momentum parameter β.

4. The edit options built into the tool

It is possible to edit some parameters in the tool to set the network and update rule characteristics. The following options are built into the edit menu:

- Set learn parameters: when selecting this option, a popup menu will appear in which one can set the learning rate α and the momentum parameter β. The learning rate can be varied between 0 and 10 and the momentum parameter can be varied between 0 and 1.
- Choose network: with this option it is possible to choose the network that has to be trained: a one layer network or a two layer network. The networks have two input units and one output unit. Moreover, the two layer network has two hidden units in its hidden layer. Furthermore, the networks have one bias unit in each layer to achieve the threshold effect
- Choose function: this option can be used to choose the function the net-

work has to learn: the logical AND function, the logical OR function, the logical XOR function, the ID1 function (output is input1) and the ID2 function (output is input2). As the values 0 and 1 are limit cases for the sigmoid function which calculates the output activation, these values will never be reached when training the network on them. For this reason, the logical 0 is presented with a value of 0.1 and the logical 1 is presented with the value of 0.9.

- Choose order: as updating the weights is done incrementally, the order in which the training set is presented to the network, will determine the path through the weight space. To analyse the effect of the training order on the path through the weight space, it is possible to choose with this option in which order the different inputs are presented to the network. It is only possible to set fixed training orders in the current version of the tool. One topic for further research is to analyse the effect of using a random order in stead of a fixed one.

There are also two other options which can be set in the edit menu. These options both influence the way in which the diverse graphs are presented on the screen:

- Set colors: with this option one can choose the colors used in the graphs.
- Set 3D view parameters: the motion through the weight space can be visualized in a three dimensional graph, with each axis of the graph representing a weights combination. It is possible to rotate the graph round these axis, move the view position along the axis and translate the graph in the widget.

5. The train options built into the tool

It is possible to set various train options to influence the way in which the network is trained. The following options can be specified in the tool:

- Reset network: when the tool is started for the first time, the network will be initialized with random weights. Every time one chooses this option, the network will be re-initialized with these same random weights. These random weights are uniformly distributed over the interval [-1.0, 1.0). Note that it is possible to set new start weights with the *New random weights* option.
- Single step: when choosing this option, the network will be trained with a training session consisting of one step: all 4 input-output pairs will be presented to the network in the order as specified with the *Choose order* option in the edit menu.
- Train many steps: the tool will make a popup window in which the number of steps that should be used in one training session, can be set. When clicking the *Train* button in the popup, the network will be trained with as many steps as specified with the drag bar in the popup.
- New random weights: as mentioned above, is this option intended to change the random weights which are used to initialise the network. It will calculate some new random weights and the network will be initialized with these random weights. From this moment on, these new random weights will also be used when the network is reset with the *Reset network*

function.

- Set weights to zero: this last option sets all weights in the network to zero to examine if it is possible to start in the origin of the weight space. This turned out to be possible which can be explained by the fact that the gradient of the error function in this peculiar position is not for each pattern, seperately, zero. It is still an open question whether it is also possible to start in the origin of the weight space when a batch learning algorithm is used in stead of an incremental one.

6. The view options built into the tool

This tool presents various views on the behaviour of a BP network. It is possible to view the total weight space or to zoom in on a part of the weight space (with the *zoomin weight space* option). In these two views one can see how the network walks through the weight space during a training session. Furthermore, it is possible to view the converged weights against alpha; in this view, the tool will analyse for each alpha between 0 and 1.5 to which point in the weight space the network will evolve and draw these points in a graph. The last graphical view the tool offers is a so-called convolution view. This view is very similar to the previous mentioned view; it shows how the network walks through its weight space for all alpha between 0 and 1.5.

As the total weight space can be more than 3-dimensional, it is possible to choose which part of the weight space has to be viewed with the tool. One can enter an expression combining the weights which have to be combined onto one axis. Indexing the weights in these expressions has been done somehow different from the indexing method used in the previous sections; in the previous section a weight had a *layer* index, a *to* index and a *from* index. The *layer* index and the *to* index will now be combined into one index, the so-called *unit number* index. To determine the unit number of a unit, one can count the units layer by layer from left to right, skipping the bias units, starting at the first hidden layer (if it exists) and counting up to the output layer. This numbering method is shown in Figure 2 for the two networks which are built into the tool.

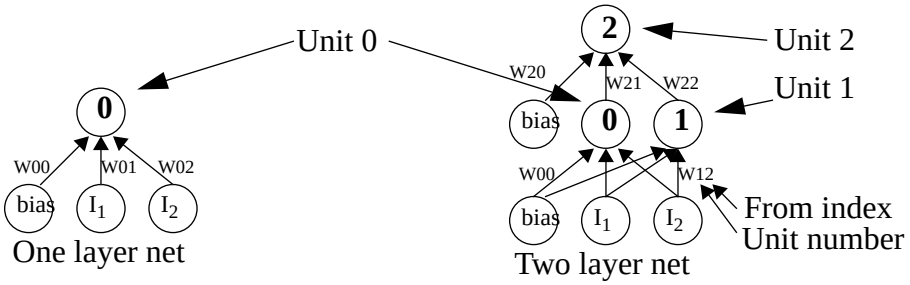


Figure 2. The unit and weight numbering used in the graphs

One can use weights, constants, the plus (+), minus (-) and multiply (*) operators and parentheses in an expression. It is also possible to use white spaces (space and tab) between the various tokens. For example, an expression combining weight W00 and W01 on one axis might look as follows:
(W00 + 0.2 * W01) / 2. It is also possible to use the implicit multiplication as used in mathematics by placing nothing at all (or white spaces) between two operands. Thus, the example expression might also be: (W00 + 0.2 W01) / 2.

6. The view options built into the tool

6.1 View total weight space

This option gives a three dimensional view on the movement through the weight space. When training the network, each training step consists of four substeps; one substep for each input-output combination. The tool traces the weights after each complete training step for this view and draws the result of this trace in the graph. The origin of the axis will always be the point (0,0,0). It is possible to set the range along the axis with the *Zoom* dragbar. When pressing the *Reset trace* button, the complete trace will be reset; it will be forgotten and the tool will start tracing from the current point in the weight space. One trace consists of maximal 10000 steps. The tool also shows the calculated values of the last traced step in a separate window. The button *Zoom in* is a short-cut to popup the *Zoomin weight space* window, which can also be selected from the view menu and the *Close* button closes the complete window. The tool not only gives a 3-dimensional view on the trace, but it also shows the 2-dimensional projections on the coordinate planes.

Figure 3 shows a trace of a one-layer networking learning the AND func-

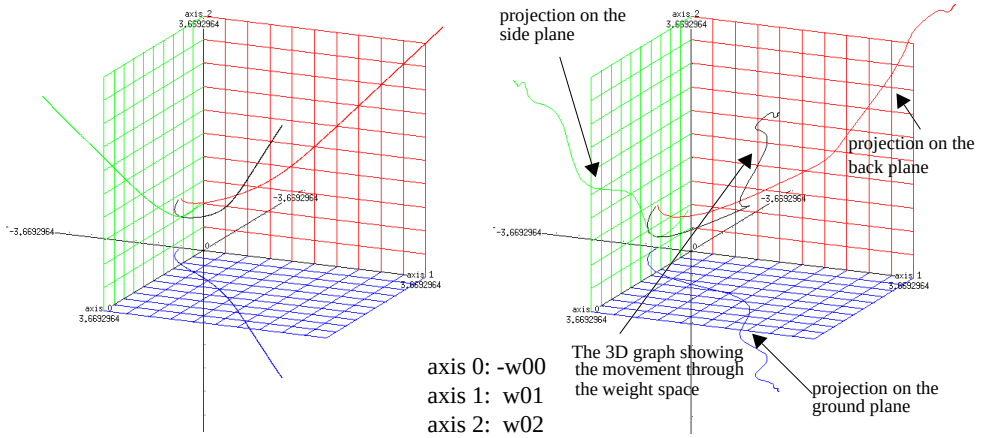


Figure 3. AND function, 3003 steps trained for two values of β

tion. The network has first been trained with the momentum parameter set to zero (β = 0) and then with the momentum parameter set to 0.97 (β = 0.97). Axis 0 is the -W00 axis (the bias weight), axis 1 is the W01 axis (weight coming from input 1) and axis 2 is the W02 axis (weight coming from input 2).

In the picture of β=0, one can see that the weights accurately follow the gradient of the error surface; the graph walks smoothly to the optimum solution (the minimum of the error measure). However, the network has not reached the optimum yet after 3003 training steps. The network will reach its optimum only after approximately 77000 training steps! The error measure at that moment is equal to 0.009720909070726696.

In the picture of β=0.97 one can see that the movement through the weight space continually overshoots the shortest route following the gradient; every time the route has to bend away into a new direction, the movement will keep something of its speed in the old direction and gradually bend towards the new direction as forced by the gradient of the error measure. However, due to the fact that the speed of the movement is much higher, the network will reach

its optimum a lot sooner; it only needs approximately 2000 steps. Now, the error measure equals to 0.009720909010578031 in the optimum point.

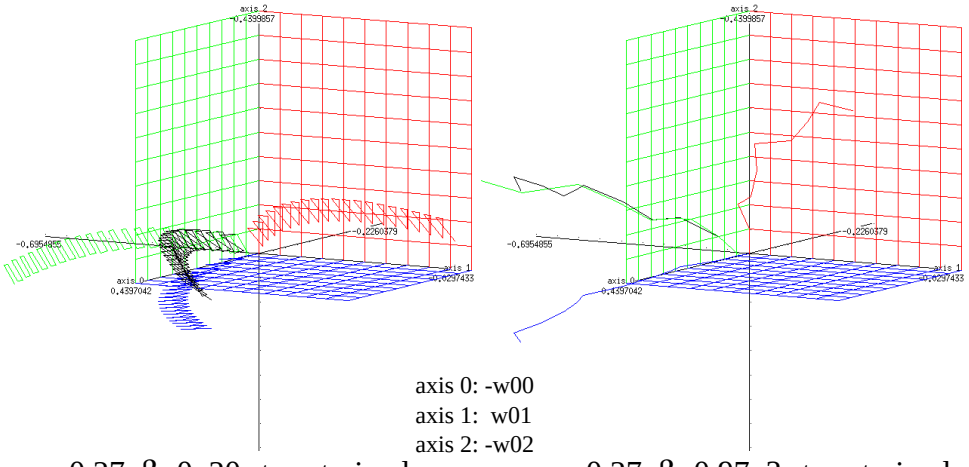
Note that the error measure in both cases is slightly different. This phenomenon will be explained in the following section.

The main advantage of this view is that it visualizes the global movement of the network through the weight space. However, it also has a disadvantage; as the weights only change very slightly between two successive steps, it is not possible to visualize what happens between these two steps. To make this visible, one has to zoom in on the graph. However, as the tool keeps the origin of the graphs at (0,0,0) and the weights move away from the origin, the point of interest will disappear from the window when using the zoom drag bar. To solve this problem, a second view has been implemented in the tool, the so-called *Zoomin weight space* view, which will be discussed in the following section.

6.2 View zoomin weight space

As mentioned in the previous section, this view has been implemented to trace the behaviour of the network between successive steps. To achieve this, the tool now traces the weights after each training substep in stead of after each complete training step. The origin of the graph will be set equal to the first weight vector in the trace, so the point of interest will not disappear from the screen when zooming in on the graph.

Each input/output combination has an own error measure with an own error surface. The total error measure is the sum of these various error measures. A consequence of this fact, combined with the incremental update algorithm, is that the weights will be adapted proportional to the gradient of the error surface introduced by the last used input/output combination. As the tool presents four different input/output combinations per training step, it can be expected that the weights will move into four different directions per training step; one direction per input/output combination. The movement of the weights through the weight space after one complete training step will be the sum of these four different movements. This effect has been shown in Figure 4, which shows the



α=0.27, β=0, 20 steps trained α=0.27, β=0.97, 3 steps trained

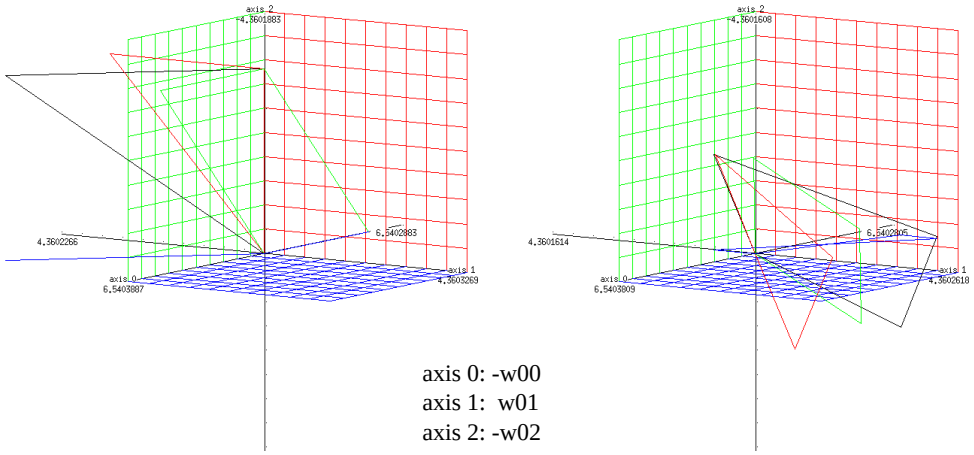
Figure 4. Zoomin trace on the AND function

first steps of the one layer network, learning the AND function with the same parameters as in the previous section.

The left picture shows a trace of the training session with the momentum set to zero. In this case is the effect of moving in another direction after each input/output combination very strong. The right picture however, shows a trace of a training session with the momentum set to 0.97. As can be seen from this picture, the large momentum makes the weights move much more in the average direction as forced by the total error measure. The individual substeps only have a slight effect on the motion through the weight space.

Note also the effect on the convergence speed: the trace in the left picture consists of twenty complete training steps, while the trace in the right picture only consists of three complete training steps. Despite the lesser trained steps, the weights have moved a longer distance through the weight space.

It is also interesting to look at the movement through the wait space when the network has converged to the optimal solution. In first instance one might expect that the network reaches one optimum point in the weight space. However, this is not the case; the network will still adapt its weights for each individual training substep. Only after a complete training step, the network will arrive in the same point again. This has been shown in Figure 5 for the one layer network, which has learned the AND function with α=0.27 and β=0 and β=0.97 respectively.



α=0.27, β=0, completely converged

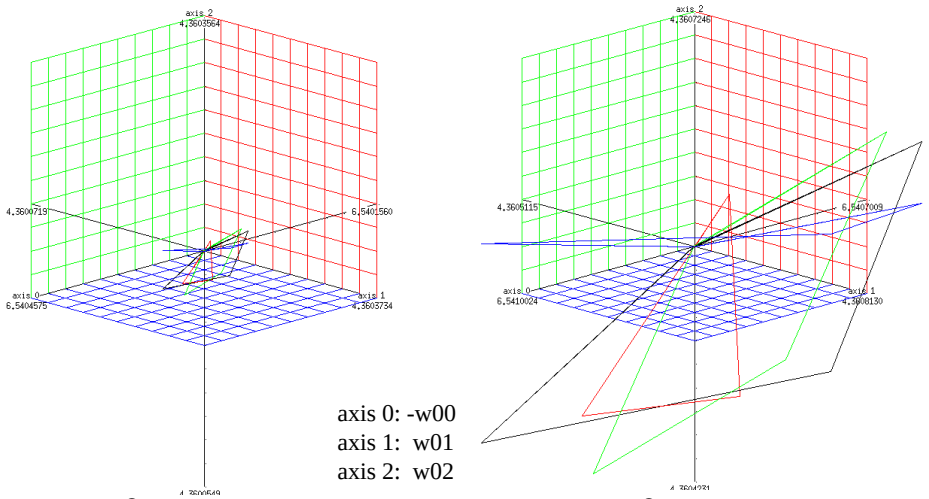
α=0.27, β=0.97, completely converged

Figure 5. Zoomin trace on the AND function

As one can see, the weights follow a different route in the β=0 and in the β=0.97 case. This is quite logical, as in the β=0.97 case, the weight change between two substeps not only depends on the surface of the error measure but also on the previous weight change. As the route through the weight space is different in the two cases, the four points that form the possible optima, are also different and as the error measure is a function of the weight vector, the error measure will also be slightly different in the two cases. This last fact has already been shown in the previous section where the error measure for the optimum solution has been given.

Note that the origin of a graph in Figure 5 consists of the weightvector after the last training step completed before the trace was started. This is one of the four points reached when the network is completely converged.

In the previous traces, the influence of the momentum parameter β has been analysed. However, it is also interesting to analyse the effect of the learning rate α. Therefore, Figure 6 shows two traces of the last steps of a one layer



α=0.19, β=0.31, last steps α=0.99, β=0.31, last steps

Figure 6. Zoomin trace on the AND function

network, learning the AND function, for two different values of α. The parameter β has been set to 0.31 in both figures.

In the left figure, α has been set to 0.19. It shows the movement through the weight space from step 71962 to step 71984. The network has reached its optimum after 71984 steps.

In the right figure, α has been set to 0.99. This figure shows the movement through the weight space from step 14670 to step 14682. After this last step, the network has reached its optimum.

Note that the network converges only very slowly to its optimum in these last steps. The convergence is that slow that it looks like the network is going back towards the same point after each complete training step. That this is not really the case, can be seen when zooming further in.

Note also that the shape of the round, which is made through the weight space in the optimum case, is the same for α=0.19 and α=0.99. The only difference is that for α=0.19, the round is smaller then for α=0.99. This in contradiction to the case where β was varied. In that case, both the shape and the size of the round changed

6.3 View converged weight against alpha

The previous view has shown that the points to which the network converges, depend on the learning rate α and the momentum parameter β. To analyse these dependencies in more extend, a third view has been implemented. In this third view, the network is trained until it has reached its optimum state, and the four points that are reached for the four input/output combinations, are plotted in a graph. This is done for 300 values of α, varying from 1.5 to 0.005.

The popup window that appears after selecting this view, shows 3 graphs at the same time. It is possible to set the weight combination that has to be

viewed for each of the graphs.

When the network has reached its optimum state, the value of α will be decreased with 0.005 and a new training session will start to find the new optimum value. When the tool starts this new training session, it can continue with the actual weights combination or it can reset the weights to the initial values. This can be controlled with the *Auto reset* button; when *Auto reset* is on, the network will be reset each time. Otherwise, it will continue with the last weights combination.

Sometimes, the convergence of the network will go wrong; the network can reach a local minimum or it can arrive on a very flat plane of the cost surface. In the first case, the network will not converge any further but the error will still be to big for some input/output combinations. In the second case, the network will only converge very slowly. It can take a few million steps to leave such a plane. The tool can detect both cases. When such a case arises, the tool can possibly give a warning and ask what to do next: continue with the current value of α or abort the training session and start a new training session with the next value of α. Whether the tool gives a warning or not, can be controlled with the *Warnings* button; when *Warnings* is on, the tool will give the warning and ask the user what to do. Otherwise, it will abort the training session and continue with the next value of α when one of the problems arises.

The last option which can be set is the *Log(delta weight tolerance)* dragbar. Each time when the network has been trained one more step, the tool has to determine whether the network has reached its optimum or not. To determine this, the network will calculate how much the weights have been changed in the last training step. When the weight change is less then the *delta weight tolerance*, the tool assumes that the network has reached its optimum. With the dragbar, one sets the ^{10}log of this *delta weight tolerance* value.

Figure 7 shows three traces of a one layer network, training the AND function, for three different values of β, with the *auto reset* switch set on. In these

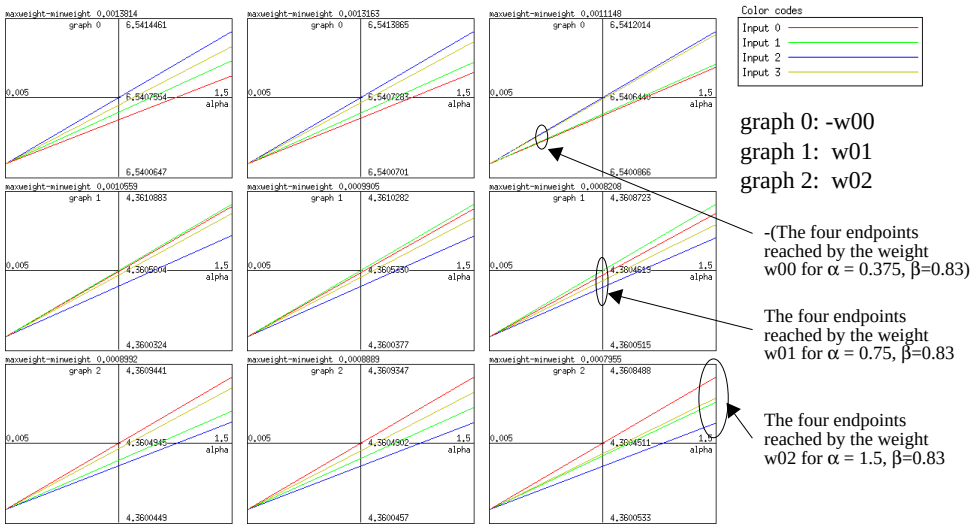


Figure 7. Converged weights, AND function, training order: 1234

graphs, the same two notions can be found as in Figure 5 and in Figure 6:

1)when β varies, the shape of the round through the weight space will vary; the four lines showing the four points reached after the substeps, walk different in the three traces.
2)when α varies, the shape of the round remains the same, only the size varies.

This picture also makes clear that for a smaller α, the size of four endpoints which can be reached will be closer to each other and that a linear relationship exists between the size of the round and the learning rate. However, at the moment, these last two conclusions can only be applied to a one layer network learning the AND function, as the path through the weight space, and therefore the endpoint which will be reached, is a function of the error measure, which depends on the network (through the weightvector) and the function to be trained.

To examine if these conclusions possibly can be generalized, the one layer network has also been trained with some other functions: the OR function, the ID1 function and the ID2 function. The traces of these training sessions can be found in Figure 8.

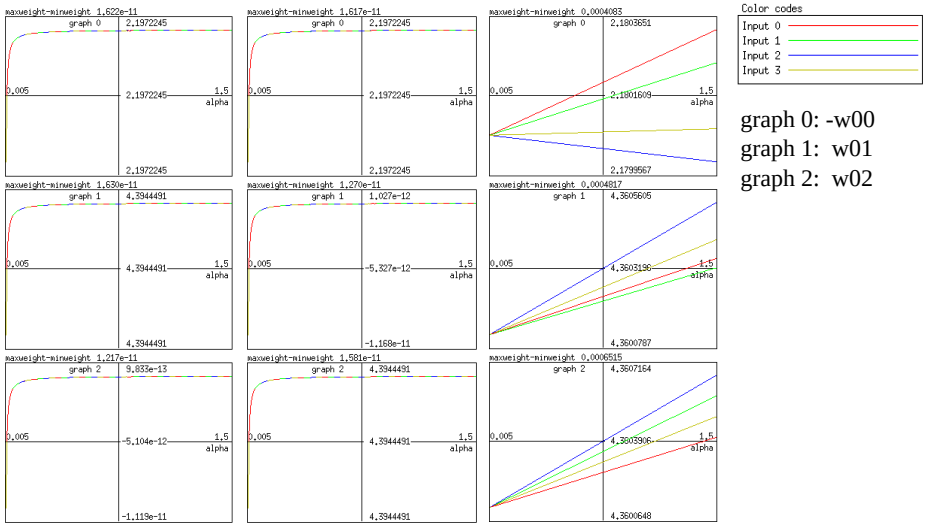


Figure 8. Converged weights, 3 different functions, training order: 1234

As can be seen in this figure, the above mentioned conclusions can clearly be applied to the graph of the OR function; in this graph, the four lines representing the weights combination after the substeps, linearly converge to one point for α=0.005. However, it is less clear for the ID1 and ID2 functions. In these graphs, the four lines all map on to each other, so it is not possible to see whether they come closer to each other for a smaller value of α. The graph only shows that the endpoints reached depend on the learning rate.

To examine why the linear relationship clearly exists for the AND and the OR function, while it apparently does not exist for the ID1 and the ID2 function, a mathematical analysis is necessary. For this analysis we will look at the behaviour of the weight update function for a one layer network which has reached its optimal state. This network is shown in Figure 9. We will analyse the weight update function, as the size of the round only depends on the weight change after each substep.

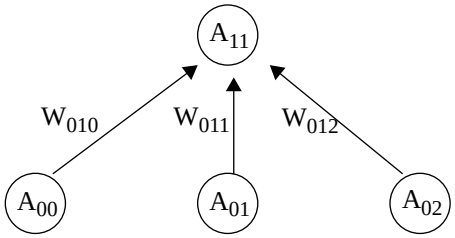


Figure 9. The one layer network considered in this section

The weight update rule[1] for one of the three weights in this network is:

$$\Delta W_{01j} = \alpha \sum_{patterns} \delta_{11} A_{0j}$$

with δ_{11} as follows:

$$\delta_{11} = g'(h_{11})\,[\zeta_1 - g(h_{11})]$$

As, for the sigmoid function, the derivative g' is equal to g(1-g), this can be rewritten as (read g as $g(h_{11})$ and ζ as ζ_1):

$$\delta_{11} = g(1-g)\,(\zeta - g)$$

Let the error in the calculated output be denoted with e, then e is defined as follows:

$$e = \zeta - g$$

Using this error e makes it possible to rewrite the function to calculate δ as follows:

$$\delta_{11} = (\zeta - e)\,(1 - \zeta + e)\,e$$

When the network has reached it optimum state, the value of e will be very small compared to the desired output ζ (ζ is 0.1 or 0.9, while e turns out to be less then 0.01). This makes it possible to approximate the formula for δ as follows:

$$\delta_{11} \approx \zeta(1-\zeta)\,e$$

This makes the weight update function in or near the optimum case as follows:

$$\Delta W_{01j} \approx \alpha \sum_{patterns} \zeta(1-\zeta)\,e A_{0j}$$

As updating the weights is done in an incremental manner, that is after presenting each individual input/output pattern p, we will get the following function for each weight update step:

$$\Delta W^p_{0ij} \approx \alpha \zeta^p \left(1 - \zeta^p\right) e^p A^p_{0j}$$

Since the desired activation ζ^p and the input activation A^p_{0j} are fixed for one input/output pattern p, this can be rewritten as:

$$\Delta W^p_{01j} \approx \alpha c^p e^p$$

1. The momentum parameter β is ignored in this analysis

This equation shows that for one and the same function, the size of the round only depends on the value of α and on the error in the calculated outputs. So, when the value of e (which is ζ-g), does not depend on α, there will be a linear relationship between α and the size of the round in the optimal case.

As such a linear relationship exists for both the AND and the OR function, we can come to the conclusion that the error in the optimum case does not depend on the value of α for these two functions. This conclusion is supported by the following data:

error \ alpha	1.03	0.76	0.50	0.27
error 1	9.71e-03	9.71e-03	9.71e-03	9.71e-03
error 2	2.35e-06	2.37e-06	2.39e-06	2.40e-06
error 3	2.37e-06	2.38e-06	2.39e-06	2.41e-06
error 4	2.35e-06	2.36e-06	2.38e-06	2.40e-06
total error	9.72e-03	9.72e-03	9.72e-03	9.72e-03

Table 1: Error measure of a one layer network that has learned the AND function with momentum parameter β = 0.50, for various values of the learning rate α.

Now we must still explain the graph of the ID1 and the ID2 function. As these two functions are almost the same (they can be transformed into each other by swapping input 1 and input 2 with each other), we will only look at the ID1 function. This function can be learned exactly by a one layer network which uses the sigmoid function. See Appendix A for the exact solution of the problem. When the network is being trained, the weights will reach a certain point in the weight space, were the function will be almost exactly known. When such a point has been reached, the error, and therefore the weight change, will be very small. As we are using computers which can only calculate with a certain accuracy, it is possible that the necessary weight change in one (sub)step will only influence the last digits of the weight. From that moment on, the network will converge very slow or it will not converge at all. Due to the shape of the sigmoid function, it is possible that the weightvector is still not at the exact solution at the moment that the convergence stops.

As the tool looks at the converge speed to determine whether the network has reached its optimal state, the tool can abort the calculations too early in such a case. This is exactly what happens with the ID1 function. When the network output is very close to the desired output, the weight change will be very small and therefore, the tool will decide that the network has reached the optimum state and abort the calculations for that value of α.

Due to the fact that the weight change is bigger for a bigger value of α, the weights will come closer to the optimal weight vector when α is bigger and remain further away from the optimal weight vector when α is smaller. So in this case, the error e (ζ-g) will depend on the value of α in the near optimal case. Thus, the graphs will not be linear for ID1 and ID2.

This theory is supported by the fact that the error measure of the ID1 function is approximately 2.54e-29 at the moment that weight w00, w01 and w02 are equal to -2.1972, 4.3944 and 1.4530e-14 respectively (for α=0.8 and β=0.50). At that moment, ΔW is too small to change both W00 and W01.

Only W02 is still influenced.

Note that in the optimal solution, the weights W00, W01 and W02 equal to approximately -2.1972, 4.3944 and 0 respectively. Thus, all three weights are close to the optimal point. However, they have not reached it yet.

6.4 View convolution

This fourth view shows the convolution of the training session; the tool trains the network for 300 different values of α (reaching from 0.05 to 1.5) and traces the path through the weight space for each separate value. These traces are shown in three graphs at a time. It is possible to set the weights combination which has to be shown in each graph.

As it is not always necessary to view the complete trace, one might for example only be interested in the last few steps of a training session, it is possible to set some trace parameters.

The first parameter is the *Trace distance*; this is the number of steps which have to be trained between to successive trace steps. Note that one trace step consists of tracing the weights combination after each of the four substeps forming one complete training step.

The second parameter is the *#steps before trace*; the network will be trained that many steps before the tool starts tracing the path through the weight space.

The last parameter is the *Max #steps to trace*; this is the maximum number of training steps which have to be made after the trace has been started.

The tool will check after every training step whether the network has reached the optimum point. It does this by checking if the weights have been changed less then set with the *Log(delta weight tolerance)* drag bar. If so, the training session will be aborted and the tool will start with the next value of α. Otherwise, the tool will check whether the network has already been trained the maximum number of steps as set in the last trace parameter. If so, the tool will again start with the next value of α and otherwise it will continue with the current value of α.

However, one restriction exists for the trace parameters; as it requires far to much memory to store the complete traces for all 300 values of α, they have to be set before the training sessions are started. So it is not possible to change for example the trace distance during the training session or change the view on the trace after the training session has been completed.

The convolution window also has an *Auto reset* button, just as the converged weights window. When the *Auto reset* is set on, the weights will be reset before tracing the network for a new value of α. Otherwise, the tool will continue with the last weights combination when switching to the next value of α.

Figure 10 shows the convolution for three different functions on a one layer network; the ID1 function, the AND function and the OR function.

The ID1 function has first been traced with β=0.10 and afterwards with β=0.50. As one can see, there is not much difference between these two traces. In both traces, and also in the traces of the AND and the OR function, the net-

work first takes large steps through the weight space, and later on, the steps

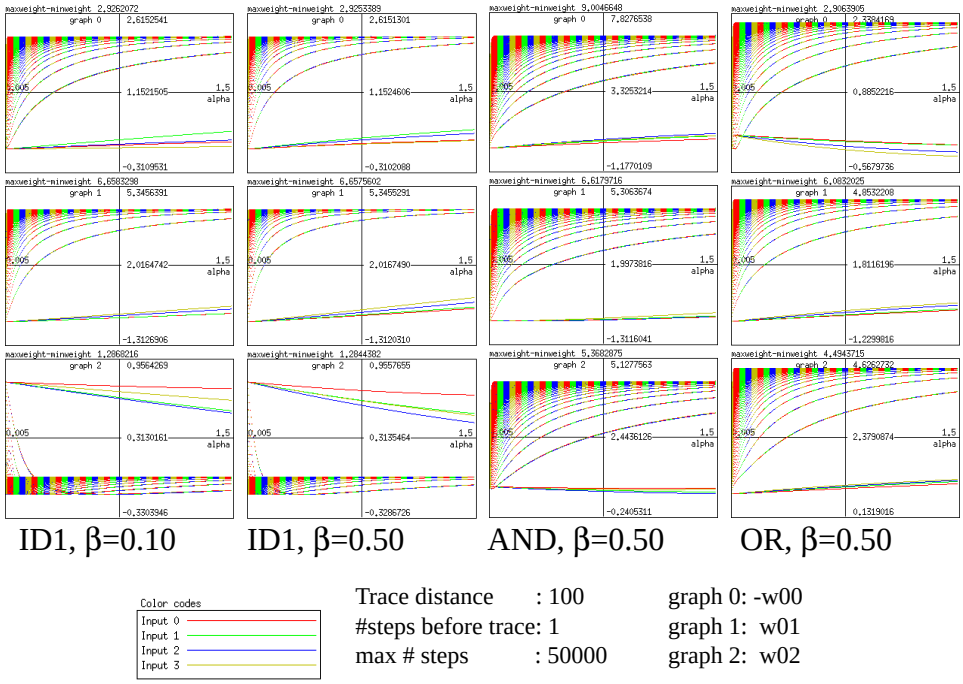


Figure 10. Convolution, various functions, training order: 1234

will be smaller; the dots showing the weight combination for one value of α, are on one end of the graph far apart from each other and on the other end, close to each other. Also in all cases, the network takes larger steps through the weight space for a larger value of α. Note that in the graphs of the ID1 function, showing w02 (graph 2), the weights first decrease to -0.13 and then increase to 0.016.

In the other two graphs (showing the trace of the AND and the OR function), the weights increase right from the beginning of the training session.

One big disadvantage of these graphs is, that it is not possible to see in what order the weight trace for one specific value of α has been plotted. So one has to use one of the other view options to see whether the weights for example have increased from the bottom of the graph to the top of the graph or that they have decreased instead

The main advantage of these graphs is, that it is possible to see the weight traces for different values of α together, so it is easier to get insight in for example the relationship between the learning rate and the convergence speed.

6.5 Test network

This last view shows some numerical information about the network; it shows the current output, the desired output and the error measure for the function to be trained. Furthermore, it shows information about the distance which the weights have walked through the weight space during one training step. The tool shows to different distances:

1) The round distance: when the network has reached its endpoint, it will walk a round through the weight space during one training step. The size

of this round is called the round distance. In fact, is this the sum over the distances walked in the four substeps, so even when the network is still converging, the round distance is defined.

2) The trend distance: before a training step, the network will be in a certain point in the weight space. After the training step, the network will be in another point in the weight space. The distance between these two points is called the trend distance.

Note that the tool only considers a three dimensional subspace of the total weight space; it only looks at the three weighted connections to the output unit to calculate the round and the trend distance.

Furthermore, the popup window shows the ratio between the round distance and the trend distance, the change of this ratio since the previous call to the test network function and the change of this change of the ratio.

These ratios have been built in the tool because the *view total weightspace* option showed that sometimes the round distance of the weights to the output unit, was very large in relation to the trend distance of these same weights. To examine this in further extent, it was necessary to built this last view in the tool to show the round distance/trend distance ratio (RTDR). The following table shows some of these rations for a two layer network training the XOR function for different values of α and β.

Note that the NTDR is only defined as long as the trend distance is not equal to zero. The network has stopped converging as soon as the trend distance is zero.

output 1 (0.1)	output 2 (0.9)	output 3 (0.9)	output 4 (0.1)	RTDR	ΔRTDR	ΔΔRTDR
$\alpha = 0.27$, $\beta = 0.31$, network passes local minimum						
0.103	0.495	0.891	0.497	1464	not def.	not def.
0.101	0.496	0.895	0.496	3316	1852	not def.
0.101	0.496	0.896	0.496	4730	1413	-439
0.101	0.496	0.896	0.496	5420	690	-723
0.101	0.496	0.896	0.496	4960	-460	-1150
0.102	0.496	0.897	0.495	2345	-2615	-2155
0.100	0.900	0.900	0.100	3	-2342	272
0.100	0.900	0.900	0.100	3	0	2342
0.100	0.900	0.900	0.100	not def.	not def.	not def.
$\alpha = 0.23$, $\beta = 0.31$, network gets stuck in local minimum						
0.103	0.496	0.892	0.498	1459	not def.	not def.
0.101	0.496	0.895	0.497	3895	2436	not def.
0.101	0.496	0.896	0.497	6554	2659	223
0.101	0.496	0.897	0.497	9335	2782	123
0.100	0.496	0.897	0.497	12202	2866	85
0.100	0.496	0.897	0.496	15133	2931	65
0.100	0.496	0.897	0.496	18117	2984	52
0.100	0.496	0.898	0.496	21145	3028	44
0.100	0.496	0.898	0.496	24210	3066	38

Table 2: Output produced by a two layer network training the XOR function. The output is taken every 10000 training steps. Also the RTDR is calculated

output 1 (0.1)	output 2 (0.9)	output 3 (0.9)	output 4 (0.1)	RTDR	ΔRTDR	ΔΔRTDR
α = 0.27, β = 0.30, network passes local minimum						
0.103	0.495	0.891	0.497	1441	not def.	not def.
0.101	0.496	0.895	0.496	3236	1795	not def.
0.101	0.496	0.896	0.496	4526	1290	-505
0.101	0.496	0.896	0.496	4959	433	-857
0.101	0.496	0.896	0.496	3921	-1039	-1472
0.101	0.895	0.895	0.108	3	-3918	-2897
0.100	0.900	0.900	0.100	3	0	3918
0.100	0.900	0.900	0.100	3	0	0
0.100	0.900	0.900	0.100	not def.	not def.	not def.
α = 0.23, β = 0.30, network gets stuck in local minimum						
0.103	0.496	0.891	0.498	1444	not def.	not def.
0.101	0.496	0.895	0.497	3866	2422	not def.
0.101	0.496	0.896	0.497	6511	2646	223
0.101	0.496	0.897	0.497	9280	2769	123
0.100	0.496	0.897	0.496	12133	2853	85
0.100	0.496	0.897	0.496	15052	2918	65
0.100	0.496	0.897	0.496	18022	2970	52
0.100	0.496	0.897	0.496	21036	3014	44
0.100	0.496	0.898	0.496	24089	3052	38

Table 2: Output produced by a two layer network training the XOR function. The output is taken every 10000 training steps. Also the RTDR is calculated

In both tables with α=0.23, the network seems to get stuck in a local minimum. As the network behaviour is the same in both tables, we will only regard the case in which β=0.30. This table only shows the results after the first 90000 training steps. However, the network has been trained up to one million training steps and it was still in the local minimum. In this table, one can see that the RTDR increases all the time. And not only the RTDR increases but also the ΔRTDR, which means that the RTDR increases at least with a quadratic speed. The ΔRTDR increases with a decreasing speed. This can be seen in the ΔΔRTDR column. However, when the network was trained many times it turned out that the ΔΔRTDR decreased slower and slower. A ΔΔRTDR of zero seems to be the limit case. This means that the ΔRTDR will become constant in the limit case and the RTDR will grow with a constant speed in the limit case. As it seems that the network is still in (or near) a local minimum as long as the RTDR is large, this means that the network will never leave this local minimum for a=0.23.

In both tables with α=0.27, the network learns the XOR function. It approaches and passes the same local minimum, as in which the network gets stuck for α=0.23. Also in this case, the RTDR first increases. But as can be seen in the ΔΔRTDR column, the ΔRTDR is decreasing until it becomes negative. As soon as this happens, the RTDR starts decreasing and in the first columns one can see that the network has left the local minimum at the time of the following measurement. Thus, the RTDR is large as long as the network is approaching or near the local minimum and it will become smaller as soon as the network is leaving the local minimum.

A possible explanation for this relation between the RTDR and the distance to a local minimum is the following one:

Near a local minimum, the network will jump forward and back between two different points in the weight space that are relative far away from each other, during one single training step. After one substep the network will jump to one of these two points and after the substep which follows the nearly opposite gradient, the network will jump almost back to the point were it came from. Therefore, the round distance will be very big. However, as the network jumps almost back to the point where it was before, the trend distance will be only very small; the network converges only very slow near a local minimum.

Thus, these two aspects combined together (a large round distance and a small trend distance) can explain why the RTDR is that large sometimes. It may be possible to make use of this for speeding up the convergence when the network is near a local minimum; when the RTDR is big, one could adapt the weights with for example 1000 times the trend distance. At this way, the network will jump a relative large distance in the direction of the trend distance, possibly jumping over the local minimum. After one such weight update, the normal training can be used again.

Note that the relationship between the RTDR and the distance to a local minimum may not exist at all in the general case. At the moment, the existence of this relationship has only be shown for a two layer network learning the XOR function.

7. Conclusions

In this report has been shown that the error measure, for a function which can not be exactly learned, in the global minimum does not have to depend strongly on the learning parameters. The learning parameters merely influence the size and the shape of the end round which the network will still make when it has reached its global minimum. The learning rate influences the size of the round and the momentum parameters influences the shape of the round.

Furthermore, it is shown that a function which can be solved exactly, does not necessarily have to be learned exactly by a neural network on a discrete computer. This can be a consequence of the restricted calculation accuracy of the computer used.

As a last point, this investigation has shown the existence of a possible relationship between the RTDR and the distance to a (local) minimum. It may be possible to make use of this relationship to detect local minima and to avoid them whenever possible.

8. Topics for future research

In this report, only very few aspects of the dynamic behaviour of BP networks have been analysed. The exact relationship between the RTDR and the shape of the weight space can be further analysed. Furthermore, it is possible to look at the influence of the training order on the behaviour, especially the influence of a random training order should be further analysed.

It is also possible to extend the tool with other activation functions and with more general forms of back propagation networks. For example, with networks which can have connections over larger distances than only one layer, like the most simple network that can learn the XOR function (see for example [3]).

9. References

1. J. Hertz, A. Krogh and R.G. Palmer, *Introduction to the theory of neural computation*, Addison Wesley, 1991
2 E.J.W. Boers, H. Kuiper, *Biological metaphors and the design of modular artificial neural networks*, Department of Computer Science and Experimental and Theoretical Psychology, Leiden University, 1992
3 I.G. Sprinkhuizen-Kuyper, E.J.W Boers, *The Error Surface of the simplest XOR Network has no local Minima*, Technical Report, Department of Computer Science, Leiden University, 1994

Appendix A. The exact solution of the ID1 function

When the network has learned a function exactly, the error measure will be zero for all patterns, thus:

$$E[\boldsymbol{w}] = \frac{1}{2}\sum_{\mu i}\left[\zeta_i^\mu - A_i^\mu\right]^2 = 0$$

This will be solved when

$$\zeta_i^\mu - A_i^\mu = 0$$

for all patterns μ and output units i.

As we have only one output unit, we get

$$\zeta^\mu - \frac{1}{1+e^{-h}} = 0$$

with h the netinput of the output unit.

This leads to

$$h = \ln\left(\frac{\zeta^\mu}{1-\zeta^\mu}\right)$$

for all patterns μ.

Thus, as we are dealing with the ID1 function, we have to solve the following linear system:

$$W_{00} + 0.1W_{01} + 0.1W_{02} = \ln\frac{1}{9}$$

$$W_{00} + 0.1W_{01} + 0.9W_{02} = \ln\frac{1}{9}$$

9. References

$$W_{00} + 0.9W_{01} + 0.1W_{02} = \ln 9$$

$$W_{00} + 0.9W_{01} + 0.9W_{02} = \ln 9$$

The solution of this system is:

$$W_{00} = \ln\left(\frac{1}{9}\right) \approx -2.197224577$$

$$W_{01} = \ln(81) \approx 4.394449155$$

$$W_{02} = 0$$

which is the exact solution of the ID1 function learned by a one layer network.