Chapter 5 - Fault Tolerant Artificial Neural Networks

One very interesting property of biological neural networks of the more developed animals is their tolerance against damage to individual neurons. It is believed this is a consequence of the distributed representation used by the nervous systems of such animals. The large number of neurons and interconnections available makes it easier to obtain such a distributed representation by exploiting redundancy, as was earlier realized by von Neumann [Neu56]. However, the presence of a large number of neurons and interconnections is not enough to guarantee such fault tolerance. The learning mechanism has to be able to exploit the capability of the available hardware according to its priorities.

In the case of biological neural networks a solution tolerant to loss of neurons has a high priority since a graceful degradation of performance is very important to the survival of the organism. It is still unclear how living organisms achieve such fault tolerance since not enough is known about their learning mechanisms. For artificially created systems, based on artificial neural networks or not, such a graceful degradation in relation to damage of its internal components is highly desirable since it directly reduces maintenance costs and increases overall productivity.

In this chapter we propose a simple modification of the training procedure commonly used with the Back-Propagation algorithm in order to increase the tolerance of the feedforward multi-layered ANN to internal hardware failures such as the loss of hidden units. The proposed method switches, during training, between the different possible fault configurations, i.e. all possible configurations are also trained and forced to share the same set of network weights. The conventional Back-Propagation algorithm can be considered as a special case of the proposed method, where the set of possible configurations contains only the no-fault configuration. The benefits of the proposed method are demonstrated in this chapter for a bit-mapped image recognition problem and in chapter 7 for a nonlinear control application (control of an inverted pendulum).
5.1 - Approaches to Fault Tolerance

The classical approach to achieve fault tolerance has been to duplicate several times the system performing an important task. If it is easy to detect when the system is faulty within the required time period, one could simply have just one of the duplicates functioning until it becomes faulty. Then it is simply replaced by one of its non-faulty copies. In this scheme only one of the copies is functioning at a given time. This is known as *standby redundancy* [GrBo72]. In other cases it may be difficult or cumbersome to decide if the system is faulty within the required time period. One alternative in such cases is to have all copies functioning at the same time receiving the same input signals. Each copy generates its own output independently of the other copies. The overall system output can then be produced by using a voting procedure, such as a majority decision. This type of redundancy is known as *majority voting redundancy* [DhSi81] and was proposed by von Neuman ([Neu56], [Sho68]). The simplest majority-voting system contains 3 elements where at least any 2 elements and the "voter" are required to be working successfully. This is known as a *triple-modular redundant (TMR) system* ([DhSi81], [GrBo72]).

The problem with the classical approach is that it can be costly to produce the duplicates, i.e. the trade off between cost and redundancy. Although it seems always necessary to have some redundancy in order to have some degree of fault tolerance, it may be possible to exploit the redundancy in more effective ways. In this chapter we aim to exploit the redundancy that exists in artificial neural networks in terms of a large number of units and weights to make them fault tolerant.

One effective way to exploit redundancy, assuming that we are dealing with complex tasks, is to divide the task into sub-tasks and to have several sub-systems. Assume that it is possible to perform such division in such a way that there is no one-to-one correspondence between the sub-tasks and sub-systems. Each sub-system contributes to a number of sub-tasks and each sub-task is performed by a number of sub-systems. In this case, if the number of sub-systems and sub-tasks is large enough and no sub-system is vital to any of the sub-tasks, the loss of a relatively small set of sub-systems randomly selected will probably affect the performance of all sub-tasks. However, the loss of performance in each sub-task will be small. Consequently the overall loss of performance will also be small and the overall system will degrade gracefully. The
workload to execute the task and the sub-tasks are distributed over the sub-systems and the power of each sub-system is also distributed over several sub-tasks. Furthermore, if the sub-systems can operate in parallel and the sub-tasks can be performed at the same time, the time necessary to execute the overall task will be very much reduced.

The major difficulty with such an approach is to devise the task decomposition into sub-tasks and the division of the power of each sub-system to each sub-task.

As an example of the above strategy imagine that the task is to paint a square board with dimensions 10m-by-10m and there are 1000 painters available. The standby redundancy approach would use only one painter at a time until he becomes "faulty" (perhaps too tired in this case). Then he is replaced by another one and so on. Alternatively we can divide the task into 100 sub-tasks where each sub-task is to paint a 1m-by-1m allocated part of the board. The 1000 painters then go around and paint just a small portion of a large number of the 1m by 1m squares. The loss of 100 randomly selected painters probably will affect all squares but only a small portion of each square will be left unpainted, that is the loss of painters is uniformly distributed over the 100 squares, the sub-tasks.

In another case if we are forced to allocate each group of 10 painters to paint only a specific area, the alternative is to increase the size of the area painted by each group. Then, assuming that painters are lost in sets of groups, if a group is lost, its assigned area will be at least partially painted by the other groups. Unlike the previous situation, in this case the power of each sub-system is dedicated (or said to be localized) to a specific sub-task. However, by increasing the scope of the sub-tasks, the fault tolerance of the system is improved.

5.2 - Fault Tolerance in Artificial Neural Networks

Since Artificial Neural Networks are composed of a large number of simple computational units operating in parallel, they have the potential to provide fault tolerance. However just the presence of a large number of units cannot guarantee that the ANN will be fault tolerant. The learning algorithm has to be able to exploit (or organize) the existing excess of units in such a way that the network is fault tolerant, if fault tolerance is one of our requirements. In other words, the learning algorithm is responsible for decomposing the task into the appropriate sub-tasks and divides the
computational power of the units among such sub-tasks accordingly.

Today the most popular algorithm used to train a feedforward multi-layered artificial neural network (FF ANN) is the Back-Propagation (BP) algorithm. The BP algorithm, by using a gradient search procedure, tries to minimize an error function defined as the squared output error averaged over the set of training data. No priority, however, is given to the network fault tolerance. For a particular network topology (number of hidden layers, number of hidden units in each layer), it is very likely that there are several solutions (sets of weights) which will have similar satisfactory squared output errors over the set of training data but different degrees of tolerance to hardware failures. The solution to which the BP algorithm will converge (assuming that it does converge) depends on factors such as the initial network weight values, size of the learning rate, and pattern order presentation.

In order to give some priority to network fault tolerance, we are then left with the choice of two main paths: 1) to modify the BP algorithm but still keeping its basic features, since, relatively, it is a simple algorithm that is not computationally demanding; or 2) use a more complex algorithm that carries a greater computational overhead but with the benefit of faster convergence. Each option will have its own strengths and weaknesses. One important point that has to be taken into consideration is the trade off between the desired fault tolerance and the network accuracy when there is no fault. Most methods tend to improve fault tolerance while decreasing network accuracy for the no-fault case.

Bugmann et al. [BSRP92] propose to apply the BP algorithm until it converges to a solution with a small output error. Then training is stopped and the network is tested for fault tolerance for loss of hidden units. The hidden unit which causes the largest increase in the error function is determined and duplicated (its incoming weights and bias are kept the same but its outgoing weights are halved). In order to keep the number of hidden units constant, they propose to remove the hidden unit which causes the smallest increase in the error function or apply some pruning technique (for instance [NoHi92a], [NoHi92b]). After such duplication and removal/pruning, network training is resumed with the BP algorithm in order to further reduce the output error.

We can envisage that a possible generalization of the duplication stage of this technique is to make $X$ copies of each hidden unit, where $X$ could be set as an integer measure of the degree of importance of the particular hidden unit. The larger the
increase in the error function caused by the loss of a hidden unit, the more important that hidden unit is. However, such modification of the original algorithm has yet to be investigated. Note that, since learning continues after the duplication stage if no measures are taken, the hidden units that were duplicated will have exactly the same weights. Probably learning will be easier if such a constraint is not imposed. One easy way to overcome such a constraint is to make imperfect copies, i.e. some noise is added to the weights of the duplicate units.

Still on the subject of duplication, Izui and Pentland [IzPe90] mathematically analyze feedforward ANN with a high degree of unit duplication using exact copies, the simplest form of redundancy. More specifically, they assume that each input and hidden unit is duplicated $M$ times. Normally it is assumed that the effect of such duplication would be only to improve the fault tolerance of the network. Surprisingly, they found out that feedforward ANN with such duplication, the simplest form of redundancy, learn considerably faster than unduplicated ones. They also showed that feedback ANN with such duplication will also converge faster to their solution. They argue that duplicated feedforward and feedback networks also require less accuracy in inter-unit communication, i.e. the weights can be specified with less accuracy.

In general the use of redundancy results in networks with a large number of units. However, as the number of units increases, the difficulty of synchronizing all units also increases. The use of asynchronous operation (as seen in biological systems) avoids such difficulty. Furthermore, Izui and Pentland [IzPe90] shows that training an asynchronous network using a gradient descent rule is equivalent to training a synchronous network using an update rule which takes into consideration the second-order derivative of the cost function, in effect combining first- and second-order derivatives. Therefore the use of asynchronous operation has also the computational benefit that it can speed up training.

Neti et al. [NSY92] pose the fault tolerance issue of feedforward neural networks as a nonlinear constrained optimization problem by defining the concept of a maximally fault tolerant neural network. The problem of training a maximally fault tolerant neural network is defined as finding a set of weights that performs the required mapping (according to a set of input-desired output training patterns) with the added constraint that when any of the hidden units is removed the mapping error (measured over the set of training patterns) should not increase by more than a user-defined parameter $\epsilon > 0$. 
In general reducing the size of \( \varepsilon \) reduces the number of possible solutions. If \( \varepsilon \) is too small there is even the possibility that no solution exists. He applies an unspecified quadratic programming algorithm to solve an example.

A problem with the above formulation arises if there is even just one hidden unit which causes a large increase in the mapping error while all the other units cause a small increase. Depending on the problem in hand, too much emphasis will be placed on just one unit, making the problem more difficult than it needs to be. Another problem is that quadratic programming algorithms require a much greater computational effort than the Back-Propagation algorithm. Moreover, Saarinen et al. [SBC91], not considering the fault tolerant issue, argue that many ANN training problems are ill-conditioned and may not be solved efficiently by second-order optimization algorithms.

### 5.3 - Back-Propagation with Increased Fault Tolerance

The Back-Propagation algorithm tries to minimize the scalar cost function \( J \) defined as the squared output error averaged over the set of training data, or:

\[
J = E \left[ (T - y)^2 \right]
\]  

where \( T \) and \( y \) denote respectively the desired (target) and actual network output column vectors. However, we propose to redefine the cost function as a weighted sum of the errors for all possible system configurations [NaZa93]:

\[
J^* = \frac{1}{NC} \sum_{i=1}^{NC} \lambda_i \ E \left[ (T - y_i)^2 \right]
\]  

where \( NC \geq 1 \) is the number of possible configurations, \( y_i \) is the network output vector for configuration \( i \) and \( \lambda_i/\ NC \) can be interpreted as the probability of configuration \( i \) occurring (\( \lambda_i \geq 0 \)). The set of possible configurations normally includes the no-fault and fault configurations. Observe that the cost function \( J \) can be seen as a particular case of \( J^* \) where just the no-fault configuration is considered. Examples of faults are: 1) the loss of a set of weights or a set of units in any of the layers (including the case where the faulty weights or units are in different layers); and 2) having the output of a unit stuck at some value. The above redefinition of the cost function has also been independently proposed by Vallet and Kerlirzin [VaKe91].

An epoch is defined as the presentation during training of all input-desired output
pairs from the training set, with each pair being presented just once. One way to implement the minimization of the cost function $J$ over the training set is to randomly choose one training pattern from the training set for each iteration of the BP algorithm and to update the network weights after each presentation. This is sometimes called *random incremental updating* as opposed to *sequential cumulative updating* [Zur92] when the patterns are presented with a fixed sequence and the weights are updated only at the end of the epoch.

In the same manner, the minimization of cost function $J^*$ can be implemented by randomly selecting for each epoch a possible configuration from the set of possible configurations and then, within such an epoch, using random incremental updating. A possible variation is to select randomly the configuration whenever a new training pattern is selected. The basic difference is the frequency used to change the network configuration being trained. We refer to such procedures as *switching training methods*. When the BP algorithm is used with such a switching training method, we refer to the entire training procedure as the *BPS algorithm*.

Consecutive weight updates use the network weights given at the end of the previous update, even if a different configuration was used there. Note that this means that the same network weights are shared by all network configurations, i.e. the weights for the fault configurations are a subset of the weights for the no-fault configuration. This is very different from using a different set of weights for each possible configuration. In the last case a fault detection algorithm and a much larger storage capacity would be needed, since we will need to decide which fault has occurred and then to load the weights for that particular fault configuration into the network. On the other hand, the training algorithm would try to optimise the network response for each configuration, without considering the other possible configurations, and consequently we could, theoretically, get a better solution for each fault configuration.

Consider the particular case where the ANN has only one layer of hidden units, no direct connections from input to output units, and the possible set of faults is defined as the loss of each one of the hidden units, with only one of them lost in each fault. The loss of hidden unit is defined here as the output of the faulty hidden unit fixed to 0 for any input and for any set of incoming weights. The cost function $J^*$ can be written as:
where $NH$ is the number of hidden units and $\lambda^{[0]}$ and $y^{[0]}$ correspond respectively to the probability and network output for the no-fault configuration. The network output $y^{[i]}$ can be calculated in vectorial notation as:

$$y^{[i]} = F\left[ W_{oh} \left( S^{[i]} \text{out}_h \right) + \text{bias}_o \right]$$

where $F(\cdot)$ is the function used in the output layer, $W_{oh}$ is the weight matrix between the hidden and the output layer, $\text{out}_h$ is the column vector with the output of the units in the hidden layer, and $\text{bias}_o$ is the bias vector for the output units.

The matrix $S^{[i]}$ is a square matrix with dimension $NH$ defined for $i = 0$ as the identity matrix, and for $1 \leq i \leq NH$ as $\text{diag}(1,\ldots,1,0,1,\ldots,1)$, i.e. the diagonal is composed of 1’s except the zero $i^{th}$ element. We can define another vector $C$ which contains the integers 0 to $NH$ with each integer repeated several times. The number of repetitions for a particular number $i$ divided by the number of elements in vector $C$ should correspond to the probability of fault for hidden unit $i$ if $1 \leq i \leq NH$, or to the probability of the no-fault configuration if $i = 0$. An element $j$ of vector $C$ is then randomly chosen with uniform distribution. This corresponds to the choice of $S^{[i]}$ where $i = C_j$, i.e. a fault at hidden unit $i$ if $1 \leq i \leq NH$, or no fault if $i = 0$. Since, in effect, the vector $C$ contains the set of possible network configurations and their relative probabilities, we refer to $C$ as the configuration vector.

Assume that: 1) the vector $\text{bias}_o$ and the matrix $W_{oh}$ have been initialized; 2) the configuration vector $C$ has been properly defined and contain $NEC$ elements, where $NEC \geq 1$; 3) we want to select a possible configuration from the vector $C$ every $SWEPO$ epochs, where $SWEPO \geq 1$. Consider that we want to train the ANN for $NEPO$ epochs, where $NEPO \geq 1$. The BPS algorithm can then be summarized as follows:

$$J' = \frac{1}{NH+1} \sum_{i=0}^{NH} \lambda^{[i]} E\left[ \left| T - y^{[i]} \right|^2 \right]$$

where $NH$ is the number of hidden units and $\lambda^{[0]}$ and $y^{[0]}$ correspond respectively to the probability and network output for the no-fault configuration. The network output $y^{[i]}$ can be calculated in vectorial notation as:
The BPS Algorithm: (% are comments):

1) \( SWE := 0 \); % Initialize this variable
2) Loop \( EPO \) from 1 to \( NEPO \); % Loop for the number of epochs
3) If \( SWE = 0 \); % Select a possible network configuration
   Using a uniform random distribution, select an integer \( j \)
in the interval \([1,NEC]\).
   \( NH := C(j) \); % get the number of the hidden unit to be killed
   If \( NH > 0 \); % kill temporarily this hidden unit
   \( WTEMP := W_{oh}(\cdot,NH) \); % save the weights from hidden unit \( NH \)
   \( W_{oh}(\cdot,NH) := 0 \); % set all the weights from hidden unit \( NH \) to 0
   EndIf \( NH \);
   EndIf \( SWE \);
4) If \( NH = 0 \); % Train the net
   % The no-fault configuration case
   Train the ANN for 1 epoch using the BP algorithm.
   otherwise;
   % The fault configuration case
   Train the ANN for 1 epoch using the BP algorithm
   keeping \( W_{oh}(\cdot,NH) := 0 \).
   EndIf \( NH \);
5) \( SWE := SWE + 1 \); % Increment this variable
6) If \( SWE = SWEPO \); % Restore the weights of the killed hidden unit
   If \( NH > 0 \);
   \( W_{oh}(\cdot,NH) := WTEMP \);
   EndIf \( NH \)
   \( SWE := 0 \); % reset this variable
   EndIf \( SWE \);
7) EndLoop \( EPO \)

It is important to note that there is no unique solution for the fault tolerance
problem as it has been proposed, i.e. considering the loss of hidden units. A simple
example is enough to explain this. Consider an ANN with one input, one output and two
hidden linear units. The cost function \( J^* \) can be written as:

\[
J^* = \lambda^{[0]} E \left\{ \left[ T - \left( W_{oh_i} W_{hi_i} + W_{oh_j} W_{hi_j} \right) x \right] ^2 \right\} +
\lambda^{[1]} E \left\{ \left[ T - \left( W_{oh_i} W_{hi_i} \right) x \right] ^2 \right\} - \lambda^{[2]} E \left\{ \left[ T - \left( W_{oh_i} W_{hi_i} \right) x \right] ^2 \right\}
\]

(5.5)

Note that the weights appear in eq. 5.5 always in the same pairs. Therefore eq. 5.5 could
be rewritten in terms of only 2 unknowns, the scalars \( W_1 \) and \( W_2 \) where:

\[
W_1 = W_{oh_i} W_{hi_i} \quad \text{and} \quad W_2 = W_{oh_j} W_{hi_j}
\]

In other words, the input weights can not be separated from the output weights. In
essence, this is a problem of parameter identifiability.
5.4 - Simulations

In this section we simulate the proposed technique of switching between the different ANN fault configurations during training in order to achieve a more fault tolerant solution while keeping a reasonable small input-output mapping error.

The particular problem considered is a bit-mapped image recognition problem. Figure 5.1 illustrates the 16 patterns used to train the network. They were taken from the 8 by 8 character bitmaps used by the IBM PC XT (0-9,A-F) and were edited to remove the redundant last row and column, since they contain only 0’s, to form the 7 by 7 bitmaps. Each pixel of the bit-mapped image used as input is associated with an input unit and, therefore, the input layer has 49 inputs. The output layer uses a 1-of-16 grandmother coding, where only one of the output units should be activated for each input pattern from the training set. There is only 1 hidden layer which can have 6, 8, 10 or 12 units and there is no direct connection from the input to the output layer.

The hidden and output units use the hyperbolic tangent ($\tanh(x)$) as the activation function. The input and desired output signals were scaled to lie between $-1$ and 1. The network weights and biases were initialized as small random values with a uniform distribution in the interval $[-1/3, 1/3]$. The learning rate and momentum rate were set

![Figure 5.1 - The image bitmaps used to train the ANN](image)

† Many thanks to Peter Green for his expertise in extracting the bitmaps from the inner depths of the IBM PC and making it available to us.
to 0.25/8 and 0.1 respectively. In order to speed up learning a value of 0.1 was added
to the derivative of the activation function as suggested by Fahlman [Fah89].

The ANN was trained using random incremental updating. The network was
trained, in all cases, using 600 epochs. Care was taken to ensure that the weight initial
values and the pattern presentation order were the same for the BP and BPS algorithms.

For the case where the BPS algorithm was used, it was assumed that all hidden
units have the same probability of failure and the no-fault configuration is as probable
as the fault configurations. The only type of fault considered is the output of only one
of the hidden units is fixed at 0. When a particular network configuration was chosen,
it was kept during training only in that epoch, i.e. SWEPO = 1.

The no-fault configuration was tested, using the same patterns shown in fig. 5.2,
at the end of every 4 epochs, making a total of 150 tests. The root-mean-square error
(RMS Error) calculated at each test is defined as:

\[
RMS Error = \frac{1}{N_{out}} \sqrt{\sum_{pat=1}^{N_{pat}} E_{pat}^2}
\]

(5.6)

where \( N_{out} \) = number of output units = 16, \( N_{pat} \) = number of patterns used during
test = 16 and

\[
E_{pat}^2 = (T_{pat} - y_{[0]})^T (T_{pat} - y_{[0]})
\]

(5.7)

Figure 5.2 shows the RMS error history for the no-fault configuration when 6,
8, 10 and 12 hidden units were used and the ANN was trained with the BP (fig. 5.2a)
or the BPS algorithm (fig. 5.2b). Table 5.1 compares the RMS error for the no-fault
configuration after training. There are no misclassifications for any of the no-fault
configurations.

After training, the RMS error for each fault configuration was also calculated
using eqs. 5.6 and 5.7, with \( y_{[0]} \) been replaced by \( y_{[i]} \). Table 5.2 shows the mean and
standard deviation for the RMS error and the number of misclassifications (minimum,
mean, maximum values) when the ANN is tested with each one of the fault
configurations.

From table 5.2 we can see that, by using the BPS algorithm, the mean RMS error
for the fault configurations was reduced to less than 50% when compared with the case
when the BP algorithm was used. Observe also that: 1) for the cases 6 and 8 hidden
units the maximum number of misclassifications was reduced from 15 and 5 (out of 16
patterns) to just 1 misclassification; and 2) the standard deviation of the RMS error was also greatly reduced indicating that the degree of importance of each hidden unit is more uniform, i.e. the computational (or representational) load is more evenly spread over the set of hidden units.

![Graphs showing RMS error history for different hidden unit configurations](image)

Figure 5.2 - RMS error history for the no-fault configuration for cases: (a) using the BP algorithm; (b) using the BPS algorithm.

<table>
<thead>
<tr>
<th>Number of Hidden Units</th>
<th>BP</th>
<th>BPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>384</td>
<td>308</td>
</tr>
<tr>
<td>8</td>
<td>248</td>
<td>217</td>
</tr>
<tr>
<td>10</td>
<td>174</td>
<td>68</td>
</tr>
<tr>
<td>12</td>
<td>142</td>
<td>62</td>
</tr>
</tbody>
</table>

Table 5.1 - RMS errors for the no-fault configuration after training

<table>
<thead>
<tr>
<th>Number of Hidden Units</th>
<th>BP: RMS Error Misclass. [min,mean,max]</th>
<th>BPS: RMS Error Misclass. [min,mean,max]</th>
<th>Ratio - RMS Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>2923 (774) 3,6,15</td>
<td>1229 (98) 0,0,167,1</td>
<td>0.420 (0.127)</td>
</tr>
<tr>
<td>8</td>
<td>1940 (270) 0.2,13,5</td>
<td>945 (185) 0.0,0,125,1</td>
<td>0.487 (0.685)</td>
</tr>
<tr>
<td>10</td>
<td>1024 (176) 0.0,0,30,1</td>
<td>441 (58) 0,0,0,0</td>
<td>0.431 (0.33)</td>
</tr>
<tr>
<td>12</td>
<td>764 (187) 0,0,0,0</td>
<td>377 (71) 0,0,0,0</td>
<td>0.494 (0.38)</td>
</tr>
</tbody>
</table>

Table 5.2 - RMS error mean (standard deviation) and number of misclassifications for the fault configurations after training.
From table 5.1 we can also observe that, surprisingly, when training was performed with the BPS algorithm the RMS error for the no-fault configuration was also reduced. So in this case it was possible to obtain a fault tolerant solution and at the same time reduce the input-output mapping error. At this point we speculate that, perhaps, by adding an extra random component to the search procedure via the switching training method, the BP algorithm settles to a better solution, in the same way that training with a random order pattern presentation tends to work better than with a sequential order presentation [Zur92] or like the simulated annealing technique [KGV83]. However, more investigation is needed to clarify such issue.

5.5 - Regularizing the Estimation Problem by Considering Fault Tolerance

In this section we present some results that, although partial, can help to understand the consequences of defining a cost function that considers the network fault tolerance. Training a feedforward ANN, as Saarinen et al. [SBC91] and other authors have pointed out, can be seen as an estimation problem. In this section we show that under certain assumptions, by giving some consideration to the network fault tolerance, the problem of minimizing the cost function $J^*$ (eq. 5.3) can be seen as a regularized version of the problem where we seek to minimize $J$ (eq. 5.1). In other words, the number of local minima may be smaller for function $J^*$ than for function $J$. In certain cases, each local minimum of $J^*$ will have close to it a local minimum of $J$. Basically, we are interested in the set of weights that result in a small value for the cost function $J$ but that are also fault tolerant solutions.

It may be possible that such understanding, when completed, will lead to a cost function that has a unique solution. In that case there will be only one local minimum point that is also the global minimum. If such solution produces a small input-output network mapping error, it may be possible then to use effectively second-order algorithms which have a much better convergence rate than first-order algorithms such as the Back-Propagation algorithm. Therefore the time necessary to train a network would be greatly reduced.
5.5.1 - Considering The Input Weights Fixed

Assuming that the output unit is linear consider that: 1) there is only one output unit; 2) the output unit is linear; and 3) the output bias is zero. There is no loss of generality since: 1) each output unit receives its own set of weights from the hidden units; 2) if the nonlinear function used in the output unit is reversible (such as the sigmoid or hyperbolic tangent functions) then we can uniquely determine the net input of the output unit given a valid value for its output; and 3) the weight vector could be extended to include the bias of the output unit.

In a FF ANN with just one hidden layer if we assume that input weights, i.e. the weights between the input layer and the hidden layer $W_{hi}$ are fixed, the original least-mean-square problem can then be posed as finding the output unit weight vector $W_{oh}$ that minimizes $J$ where:

$$J = \frac{1}{2} E \left[ \left( T - W_{oh}^T \text{out}_h \right)^2 \right]$$

(5.8)

and $W_{oh}$ and $\text{out}_h$ are column vectors with $NH$ components, where $NH$ is the number of hidden units. The number of network inputs and the type of nonlinearity used in the hidden units is not important at this point since we can always calculate the vector $\text{out}_h$.

The stationary points of $J$ are given by $\partial J/\partial W_{oh} = 0$:

$$\frac{\partial J}{\partial W_{oh}} = E \left[ \left( T - W_{oh}^T \text{out}_h \right) \left( -\text{out}_h \right) \right] = 0$$

(5.9)

Using the property that $W_{oh}^T \text{out}_h$ is a scalar, from the previous equation we can write:

$$E \left[ T \text{out}_h \right] = E \left[ \text{out}_h \text{out}_h^T \right] W_{oh}$$

(5.10)

The cost function $J$ has a unique minimum if the matrix $\partial^2 J/\partial W_{oh}^2$ is positive definite where:

$$\frac{\partial^2 J}{\partial W_{oh}^2} = E \left[ \text{out}_h \text{out}_h^T \right] \Delta R$$

(5.11)

and $R$ is the correlation matrix of the output of the hidden units (we adopt the convention that, if $Y$ and $X$ are column vectors, $\partial Y/\partial X$ results in a matrix which element at position $i,j$ is given by $\partial Y_i/\partial X_j$). By definition the matrix $R$ is positive semidefinite ($\det R \geq 0$) [Pap84]. If the matrix $R$ is singular, then there is more than one solution for the weight vector $W_{oh}$. This is the case, for instance, when two hidden units produce the
same output because they receive from the inputs units the same weights and have the same bias.

On the other hand, if we consider the network fault tolerance in relation to faults in the hidden units, we aim to minimize the cost function $J^*$:

$$J^* = \frac{1}{2} \sum_{i=0}^{NH} \lambda_i^{(i)} E \left[ \left( T - W_{oh}^T S^{(i)}_{\text{out}_h} \right)^2 \right]$$

where the factor $(NH+1)$ can be interpreted as included in $\lambda_i^{(i)}$. The stationary points of $J^*$ are given by:

$$\frac{\partial J^*}{\partial W_{oh}^2} = - \sum_{i=0}^{NH} \lambda_i^{(i)} E \left[ \left( T - W_{oh}^T S^{(i)}_{\text{out}_h} \right) \left( -S^{(i)}_{\text{out}_h} \right) \right] = 0$$

Then the cost function $J^*$ has a unique minimum if the matrix $\frac{\partial^2 J^*}{\partial W_{oh}^2}$ is positive definite where:

$$\frac{\partial^2 J^*}{\partial W_{oh}^2} = - \sum_{i=0}^{NH} \lambda_i^{(i)} E \left[ S^{(i)}_{\text{out}_h} S^{(i)}_{\text{out}_h}^T \right] = F \odot R + F_d$$

Defining $\bar{\lambda} = \lambda^{(0)} + \lambda^{(1)} + \lambda^{(2)} + \ldots + \lambda^{(NH)}$, the matrices $F$ and $F_d$ are defined as $F_d = \bar{\lambda} - \lambda^{(i)} - \lambda^{(j)}$ and $F_d = \text{diag}(\lambda^{(1)} R_{11}, \lambda^{(2)} R_{22}, \ldots, \lambda^{(NH)} R_{NH NH})$, and the symbol $\odot$ denotes element-by-element matrix multiplication.

Assume that the true system target output $T$ was generated by:

$$T = \left[ W_{oh}^0 \right]^T \text{out}_h + \varepsilon$$

where $W_{oh}^0$ is the true weight vector and $\varepsilon$ is a statistically described perturbation such as measurement noise. Then, assuming that $\varepsilon$ is uncorrelated with $\text{out}_h$:

$$E \left[ T \text{out}_h \right] = E \left[ \text{out}_h \text{out}_h^T \right] W_{oh}^0 - E \left[ \varepsilon \text{out}_h \right] = R W_{oh}^0$$

$$\sum_{i=0}^{NH} \lambda_i^{(i)} E \left[ T S^{(i)}_{\text{out}_h} \right] = L_d R W_{oh}^0$$

where $L_d$ is a $NH$-by-$NH$ diagonal matrix defined as $L_d = \text{diag}(\bar{\lambda} - \lambda^{(1)}, \bar{\lambda} - \lambda^{(2)}, \ldots, \bar{\lambda} - \lambda^{(NH)})$.

Finally, eq. 5.13 can be rewritten as:

$$L_d R W_{oh}^0 = F \odot R + F_d W_{oh}^0$$

In the special case when all the faulty configurations are equally probable, i.e. $\lambda^{(i)} = \lambda^* > 0$ for $1 \leq i \leq NH$, eq. 5.15 can be simplified and written as:
where $R_d = \text{diag}(R_{11}, R_{22}, ..., R_{NHNH})$. Note that in this special case if $R_{ii} > 0$ for all $i$ and $\lambda^* > 0$, we can guarantee that the cost function $J'$ has a unique minimum solution since the matrix $\frac{\partial^2 J'}{\partial W_{oh}^2}$ is positive definite. This happens because, even if the matrix $R$ is singular, the matrix $\frac{\partial^2 J'}{\partial W_{oh}^2}$ is defined as the summation of 2 symmetric matrices, where one of them is positive semidefinite and the other positive definite.

It is interesting to study how the relationship between the true parameter vector $W_{oh}^0$ and its estimate $W_{oh}$ varies when the coefficients $\lambda^{[0]}$ and $\lambda^*$ change. Assume that the matrix $R$ can be partitioned as:

$$R = \begin{bmatrix} R^A & 0 \\ 0 & R^B \end{bmatrix}$$

where $R^A$ and $R^B$ are respectively singular and non singular square matrices and $\dim R^A = N_1; \dim R^B = N_2$. Assume that the first $N_1$ components of the vector $\text{out}_h$ are equal to each other. Therefore $R^A_{ij} = \gamma > 0$, $R^B_{ij} = \text{diag}(R^B)$ and

$$R_d = \begin{bmatrix} \gamma I_{N_1} & 0 \\ 0 & R^B_d \end{bmatrix}$$

where the notation $I_M$ denotes the identity matrix with dimension $M$ and $A = \text{diag}(B)$ means that $A$ is a diagonal matrix and $A_{ii} = B_{ii}$. From eq. 5.19 we can write:

$$W_{oh} = \begin{bmatrix} (Q^A)^{-1} R^A & 0 \\ 0 & (Q^B)^{-1} R^B \end{bmatrix} W_{oh}^0$$

where $\dim Q^A = \dim R^A = N_1; \dim Q^A = \dim R^B = N_2$ and:

$$\alpha_1 = \frac{\lambda^{[0]} - (NH - 1) \lambda^*}{\lambda^{[0]} + (NH - 1) \lambda^*}$$

$$\alpha_2 = \frac{\lambda^*}{\lambda^{[0]} + (NH - 1) \lambda^*}$$

$$Q^A = \alpha_1 \gamma \begin{bmatrix} 1 & \ldots & 1 \end{bmatrix} + \alpha_2 \gamma I_{N_1}$$
\( Q^B = \alpha_1 R^B + \alpha_2 R^B_d \) \hspace{1cm} (5.24)

Note that if \( R^B \) is a diagonal matrix then:

\[
R^B_d = R^B \Rightarrow Q^B = R^B \Rightarrow (Q^B)^{-1} R^B = I_{N_1}
\]

Applying the Matrix Inversion Lemma [WeZa91] we can find an analytic expression for the inverse of matrix \( Q^A \) and after some algebraic manipulation we find that:

\[
(Q^A)^{-1} R^A = \beta \begin{bmatrix} 1 \\ i \\ 1 \end{bmatrix} 
\]

(5.25)

where the scalar \( \beta \) is defined as:

\[
\beta = \frac{\left( \lambda^{[0]} / \lambda^* \right) + NH - 1}{N_1 \left( \lambda^{[0]} / \lambda^* \right) + N_1 \left( NH - 2 \right) + 1} \quad (5.26)
\]

Note that \( \beta \) does not depend on \( \gamma \) or the elements of the matrix \( R^B \), as long as the diagonal of \( R^B \) contains only positive elements. The dimension of \( R^B \) affects \( \beta \) since it determines \( N_2 \) and indirectly \( NH \). Finally, from eqs. 5.20 and 5.26 we have:

\[
W_{oh_i} = \beta \sum_{j=1}^{N_1} W_{oh_j}^0 \quad 1 \leq i \leq N_1
\]

(5.27)

Figure 5.3 shows the parameter \( \beta \) as a function of the ratio \( \lambda^{[0]} / \lambda^* \) for \( NH = 5 \) and \( N_1 = 3 \). Note that as the ratio \( \lambda^{[0]} / \lambda^* \) increases a) \( \beta \) tends asymptotically to \( 1/N_1 \) (eq.
5.26); and b) $Q^b$ tends to $R^b$ (eq. 5.22). As expected in the limit such parameters provide the correct input-output behaviour since only the no-fault configuration is being considered. However, the parameters will converge to the values that also give the best fault tolerant solution.

It is important to consider the penalty paid for including the fault tolerance criterion. Such a penalty is paid through the output error, or equivalently the parameter estimation error, for the no-fault configuration. The relative parameter estimation error for the first $N_1$ parameters of the weight vector $W_{oh}$ can be defined as:

$$E_w = \frac{\beta - 1}{N_1 - 1} = \frac{1}{N_1 - 1} \left( N_1 \left( \frac{\lambda^{[0]}}{\lambda^*} \right)^{N_1} + N_1 (NH - 2) + 1 \right)$$

and $E_w$ is maximum when $\lambda^{[0]} = 0$, i.e. the probability that one and only one of the hidden units will fail (with its output fixed to zero) is 1. Note that when $N_1$ is large, $\max(E_w)$ is approximately $1/NH$, i.e. the upper bound for the parameter estimation error tends to zero as the number of parameters increase. If we consider that the matrix $R^b$ is diagonal then the estimation error exists only for the first $N_1$ parameters of the weight vector $W_{oh}$. However, the upper bound for the parameter estimation error for the last $N_2$ parameters also decreases as $N_1$ or $N_2$ increases, even if the matrix $R^b$ is non-diagonal, since $Q^b$ also tends to $R^b$ in this situation (eqs. 5.22-5.24). This shows that, even if the cost function does not contain the no-fault model, it is possible to achieve a small input-output mapping error for the no-fault model.

5.5.2 - Considering The Output Weights Fixed

In the previous section we considered the input weights, i.e. the layer of weights that connect the input layer to the hidden layer, as fixed weights and we looked for a set of output weights that minimize the cost function $J^*$. In this section we consider that the output weights, i.e. the weights that connect the hidden layer to the output layer, are fixed and that there is only one input unit. The same analysis is valid for networks with more than one input unit since for each input unit we can define an independent weight vector that links the input unit to the hidden units.

Consider for a moment that the hidden units are linear and that there is only one output unit. The hidden unit output is therefore given by:
where $W_{hi}$ is a column vector with $NH$ components. From eq. 5.12, the cost function $J^*$ is then:

$$J^* = \frac{1}{2} \sum_{i=0}^{NH} \lambda^{[i]} E \left[ \left( T - W_{oh}^T S^{[i]} W_{hi} x \right)^2 \right]$$

(5.30)

Since the matrix $S^{[i]}$ is symmetric, then:

$$W_{oh}^T S^{[i]} W_{hi} = W_{hi}^T S^{[i]} W_{oh}$$

and as in eq. 5.13 we have:

$$\frac{\partial J^*}{\partial W_{hi}} = \sum_{i=0}^{NH} \lambda^{[i]} E \left[ \left( T - W_{hi}^T S^{[i]} W_{oh} x \right) \left( -S^{[i]} W_{oh} x \right) \right] = 0$$

(5.31)

Following eq. 5.14 finally:

$$\frac{\partial^2 J^*}{\partial W_{hi}^2} = \sum_{i=0}^{NH} \lambda^{[i]} E \left[ \left( S^{[i]} W_{oh} x \right) \left( -S^{[i]} W_{oh} x \right) \right]^T - F \otimes R^* + F_d^*$$

(5.32)

where:

$$R^* = E \left[ \left( W_{oh} x \right) \left( W_{oh} x \right)^T \right] = W_{oh} W_{oh}^T E \left[ x^2 \right]$$

(5.33)

and $F_d^* = \text{diag}(\lambda^{[1]} R_{11}^*, \lambda^{[2]} R_{22}^*, ..., \lambda^{[NH]} R_{NH NH}^*)$. As in the previous case, when all the fault configurations are equally probable, and if $E [x^2] \neq 0$ and the vector $W_{oh}$ contains only non-zero elements, then the cost function has a unique solution. In other words, under the above conditions, for every given vector $W_{oh}$ there exists a unique vector $W_{hi}$ that minimizes the cost function $J^*$ since the matrix $\frac{\partial^2 J^*}{\partial W_{hi}^2}$ is positive definite.

It should be possible to extend the above result to the case where there is more than one output unit and the hidden units use a sigmoidal function.

### 5.5.3 - Considering Input and Output Weights as Variables

At the end of section 5.3 we showed that there is no unique set of weights for the fault tolerant problem when both layers of weights are variable, at least with linear hidden units. Basically, this is a parameter identifiability problem since the input weights could not be separated from the output weights.

One simple way to solve this parameter identifiability problem is to extend the switching idea. Consider the problem where we want to find the parameters $a$ and $b$ that...
minimize the following cost function:

\[
J_1 = \frac{1}{2} E \left[ (T - abx)^2 \right]
\]  

(5.34)

where \( T \) is the true system output, \( x \) is the system input and \( T = kx \). If we write the stationary conditions, we have:

\[
\begin{bmatrix}
\frac{\partial J_1}{\partial a} \\
\frac{\partial J_1}{\partial b}
\end{bmatrix} = (ab - k) \rho \begin{bmatrix}
b \\
a
\end{bmatrix} = \begin{bmatrix} 0 \\
0\end{bmatrix}
\]

(5.35)

where \( \rho = E[x^2] \). If we assume that \( \rho \neq 0 \), then the stationary conditions imply that \( k = ab \), i.e. there are an infinite number of solutions for \((a,b)\) that result in the correct input-output behaviour. On the other hand, consider now the problem of minimizing \( J_2 \) where:

\[
J_2 = \frac{\lambda^{[0]}}{2} E \left[ (T - abx)^2 \right] + \frac{\lambda^{[a]}}{2} E \left[ (T - bx)^2 \right] + \frac{\lambda^{[b]}}{2} E \left[ (T - ax)^2 \right]
\]

(5.36)

In this cost function the switching constrains each parameter in turn to one, instead of zero as before. The stationary conditions are now:

\[
\begin{bmatrix}
\frac{\partial J_2}{\partial a} \\
\frac{\partial J_2}{\partial b}
\end{bmatrix} = \rho \begin{bmatrix}
\lambda^{[0]} & \lambda^{[b]} \\
\lambda^{[a]} & \lambda^{[a]}
\end{bmatrix} \begin{bmatrix} ab - k \\
0
\end{bmatrix} = \begin{bmatrix} 0 \\
0\end{bmatrix}
\]

(5.37)

Assuming again that \( \rho \neq 0 \), from the stationary conditions we can write:

\[
k = a \left( b^2 \frac{\lambda^{[0]}}{b \lambda^{[0]} + \lambda^{[b]}} + \frac{\lambda^{[b]}}{\lambda^{[a]}} \right) = b \left( a^2 \frac{\lambda^{[0]}}{a \lambda^{[0]} + \lambda^{[a]}} + \frac{\lambda^{[a]}}{\lambda^{[a]}} \right)
\]

(5.38)

Assuming that \( \lambda^{[a]} = \lambda^{[b]} = \lambda^* \), then:

\[
\frac{\lambda^{[0]}}{\lambda^*} \left[ b^2 (a - 1) - a^2 (b - 1) \right] + a - b = 0
\]

(5.39)

The solution for eq. 5.39 is \( a = b \). Substituting this result in eq. 5.38 and assuming that \( \lambda^{[0]} \gg \lambda^* \), we have that \( ab \approx k \). It is interesting to note that we would get this same solution if the cost function considers the length of the parameter vector as in:
Such a cost function could be interpreted as selecting, from the multiple solutions that result in the correct input-output behaviour, the one that has the smallest length.

By switching off and on the outputs of the hidden units we solve the problem of redundancy across the hidden layer, what we could call horizontal redundancy. When we switch the input and output weights to one, instead of to zero, we solve the problem of redundancy through the hidden layer, i.e. the vertical redundancy. Both methods can be combined by defining for each hidden unit: 1) a probability the output of the unit will be fixed to zero (fault type 1); 2) a probability that its input weights will be fixed to one (fault type 2); 3) a probability that its output weights will be fixed to one (fault type 3); and 4) a probability that no failure will occur. One cost function that considers such possibilities is:

\[
J = \frac{\lambda^0}{2} E\left[(T - abx)^2\right] + \frac{\lambda^*}{2} \left[a^2 + b^2\right]
\]  
(5.40)

where \(y^0\) = network output for the no-fault model, \(\lambda^0\) = probability of the no-fault model, \(y^{[ij]}\) = network output when fault \(ij\) occurs, \(\lambda^{[ij]}\) = probability of type \(j\) fault occurring to hidden unit \(i\). It is implicitly assumed that no more than one hidden unit is faulty at any given time.

**5.6 - Conclusions**

In this chapter we presented the BPS algorithm. Such an algorithm can be used to obtain the weights of a feedforward ANN which performs the desired input-output mapping with a small error and is robust to loss of hidden units. In the example simulated, the BPS algorithm clearly outperformed the BP algorithm in terms of achieving the desired robustness without increasing the desired mapping error.

The mathematical foundation of the BPS algorithm was analyzed and it was shown that, in some situations, the switching regularizes the mapping (or estimation) problem. Such analysis opened the possibility of generalizing the switching method. Future investigations may be able to show that such generalized switching methods result in a cost function that has a unique solution. In such cases there would be only a local minimum that is also the global minimum and second-order minimization
algorithms could be used effectively. Consequently the training of the feedforward ANNs would be much more reliable and faster.

In the next chapter we show how non-standard ANNs can be used to solve the problem of extremum control of asymmetrical functions with finite dither.
Chapter 6 - Extremum Control Using Artificial Neural Networks

Optimization problems in engineering can often be posed as the maximization of a static and noisy performance index. The task of maximizing such a performance index is known as extremum control ([WeZa91], [WeSc90]). Basically the extremum control task is to climb a hill, given only noisy measurements of the height at different locations.

This chapter shows how artificial neural network concepts can be used to solve the extremum control problem with a static asymmetric performance index ([NZM92], [NzM93]). A novel non-standard neural network structure is proposed and used to develop a model of a static system using the available noisy measurements of the performance index. This ANN model has the nice properties that: a) the input that maximizes its output is also adapted and is readily available; and b) the model is flexible enough to accommodate non-quadratic functions. Therefore the optimum input for static systems with an asymmetric performance index can be estimated with a small error, even if the system is excited by a dither with a large amplitude. The standard Back-Propagation algorithm, with the necessary modifications, is used to adapt the ANN parameters.

One possible extension to the multi-input case is also proposed and two simulation examples are shown. The first simulation example is for the single input case while the second example considers a two-input case where the goal is to find the location of an object within a larger image.

6.1 - The Extremum Control Problem

An extremum controller is basically a self-tuning optimizer. It continuously adjusts the input of the static system under observation such that the single output of
this system is maximized. In dynamical feedback control systems the extremum controller can be used to determine the setpoint by finding the optimum operating point and to track it when it changes. The task of the feedback controller is then to keep the system output close to the setpoint. To justify the ‘static’ assumption the sampling period for the extremum controller has to be large enough such that (in relation to the extremum controller) the dynamics of the system can be ignored.

The differences between the problems covered by extremum control and classical optimization theory [GMW81] are: a) the function to be optimized by the extremum controller is not known a priori (although we need to make some general assumptions about its behaviour); and b) the measurements of the system output are noisy.

Examples of practical application of extremum controllers are: a) adaptive optimization of the spark ignition angles of an automotive engine for different conditions of load and speed ([WeSc90], [ScWe90]); b) control of the air-fuel ratio for optimal combustion in an internal combustion engine, where the optimum point depends on the temperature and fuel quality [DrLi51]; c) adjustment of the blade angle of windmills and water turbines of the Kaplan type in order to give maximum output power [AsWi89]; and d) optimization of the biomass productivity of continuous fermentors [GoYd89].

Extremum control systems can be classified into four types ([Ste80], [Bla62]):

a) **Perturbation systems**, where a small periodic test signal is added to the input signal and its effect on the output is used to derive local information about the slope of the performance index;

b) **Switching systems**, where the input changes at a constant rate, until the extremum is passed; then the direction of the input is reversed accordingly to some fixed rule;

c) **Self-driving systems**, where the output measurements are used directly to determine the input signal, for instance, when the time derivative of the output signal is used to drive the input signal via an integrator and other auxiliary circuits;

d) **Model-based systems**, where measurements of the input and output signals are used by some identification procedure to build a model of the performance index over a large region; the model is then used to determine the optimum input.

In this chapter we will deal with model-based extremum control systems. This type of
controller exploits the noise suppression properties of recursive parameter estimation algorithms by using them to update the free parameters of a chosen model. This avoids the need to use the noisy output measurements directly to estimate the local derivatives of the performance index (using, for instance, finite difference methods [GMW81]). If the chosen model matches the observed system and if the recursive parameter estimation algorithm produces sensible parameter values, then the optimum point for the model should be close to the optimum point of the observed system.

Denoting by \( x_0 \) the input value that maximizes the system output (the optimum input value) the model-based extremum control approach can be summarized in the following steps:

1) collect a set of input-output data points around the estimated \( x_{op} \),
2) using the new data, update the system model,
3) update the estimate of \( x_{op} \),
4) go back to step 1).

Step 3 can be greatly simplified by a proper selection of the model structure. This is the case if we can constrain our model structure such that, for a given set of model parameters, \( x_0 \) can be easily calculated or if \( x_0 \) is one of the model parameters. For instance, the first case happens when we assume that the uncorrupted system output \( y \) is modelled by \( y = ax^2 + bx + c \). Then, if the parameters \( a \) and \( b \) are estimated, \( x_0 \) can be calculated as \( x_0 = -b/(2a) \). The second case happens if we assume as the system model \( y = k(x-x_0)^2 + c \), so when the model is updated, the estimate of \( x_0 \) is automatically updated as well, since it is part of the model.

Another point that must be taken into consideration is that the selection of a particular model structure will affect the set of algorithms that can be used to estimate the parameters of the model. For instance, for the above example, all parameters of the first model structure are related linearly to the output, so we have the option of using the Recursive Least Squares algorithm (RLS). We do not have this option for the second model structure and, in general, more complicated estimation algorithms will have to be used. On the other hand, the second model structure avoids some problems such as a possible division by zero in the calculation of \( x_0 \). However, if we write the second model in incremental form, we can still apply the RLS algorithm [WeSc90].

Figure 6.1 illustrates the basic algorithm for the extremum controller. For simplicity, we assume that the output noise is additive and has zero mean. A dither, i.e.
a zero mean random test perturbation signal, is added to the estimated value of $x_0$ in order to force an exploration of the local shape of performance index. In some cases it is possible to show that the dither signal assures persistent excitation of the system ([WeSc90], [WeZa91], [BoZa90]).

6.2 - The Quadratic System Model

One possible approach is to try to model the true system with a static quadratic model and to use algorithms derived from linear estimation theory to estimate the parameters of such a model [WeZa91]. One argument in favour of the quadratic model is that around any point on a smooth function we can always fit a quadratic model, assuming that the region of interest is small enough. In terms of an expansion in Taylor series this is equivalent to assuming that we can neglect the third and higher derivatives of the function that expresses the true input-output relationship.

As a consequence of such assumptions, one may be interested in the robustness of such algorithms, i.e. under what circumstances will our estimate of $x_0$ converge to the true $x_0$, the optimum input value?

Bozin and Zarrop [BoZa90] have proved, using the ODE method, that for a quadratic model, if the estimate of $x_0$ converges, it will converge to a value close to the optimum value providing the unmodelled nonlinearities or dynamics are small. Moreover, they prove that in the presence of even large unmodelled nonlinearities, the estimate of $x_0$ will converge to the optimum value for any static true performance index with a single extremum point if: a) the true performance index is symmetric or b) the
dither amplitude decays towards zero at a sufficiently slow rate. So, if the true performance index has a large asymmetry and the dither amplitude is kept large or decays to zero too quickly, the use of a quadratic model will inevitably result in a bias in the estimate of the optimum input. This happens simply because the quadratic model is not flexible enough to reflect asymmetry and a mismatch between the system true model and the estimated model is inevitable.

It is this last result that we improve in this chapter. The basic idea is to modify the standard feedforward neural network model such that we can estimate \( x_0 \) with small error when the true performance index is an unknown asymmetric function with a single extremum point and dither amplitude remains large. This can be useful in situations where it is undesirable to allow the dither amplitude to decay to zero, for instance, when we want to improve the tracking ability of our extremum controller or when the true performance index is "coarse-grained" as in the second simulation example in this chapter.

6.3 - Adapting the Artificial Neural Network to Extremum Control

Considering for the moment the single input extremum control problem, if we use a FF ANN in figure 6.1 with hidden units executing a quadratic function of their net input, we will have exactly the original quadratic formulation that we are trying to escape from. From the representation point of view there would be no advantage of having more than one hidden unit, because there would always be an equivalent quadratic FF ANN with one hidden unit performing the same symmetric input-output mapping.

If we use a FF ANN with sigmoidal functions in the hidden layer as the system model, we will need an optimization algorithm to find the input that maximizes the FF ANN output, since such a FF ANN does not have an obvious optimum input.

Another option would be to use a FF ANN with Radial Basis Functions, such as gaussian units, in the hidden layer and to force all hidden units to have the same center. Because the network output is a linear combination of the outputs of the hidden units, the network output would be maximum when the output of the hidden units is maximum. Since all hidden units have the same center position, their outputs would be maximum at the same input. Therefore the optimal network input corresponds to the
common center of the hidden units. By increasing the number of hidden units and
allowing them to have different widths, such a FF ANN can execute an input-output
mapping that a similar ANN with a smaller number of hidden units can not imitate.
However, since a RBF is a symmetric function, the input-output mapping of such ANNs
can only be a symmetric function.

From the above discussion we can see that we need the hidden units of the FF
ANN to execute a single input-single output function $f(x)$ with the following properties:

P.1 - $f(x) \to 0$ when $x \to \pm \infty$
P.2 - If $x$ is finite, $df(x)/dx = 0$ only when $x = 0$
P.3 - $f(x) \neq f(-x)$
P.4 - $f(x) \geq 0$

So, in the $x$ space, $f(x)$ is an asymmetric local function with only one extremum and it
has only non-negative values. An example of a function with such properties is:

$$f(x) = \frac{2 + v^2}{1 + \exp(-v^2 x) + v^2 \exp(x)}$$ (6.1)

Figure 6.2 shows how the asymmetry of such a function changes for three values of the
$v$ parameter. For $v = \pm 1$, this function is symmetric around $x = 0$. The numerator $2 + v^2$
is just a scaling factor such that the maximum value of the function is equal to 1. Note
as well that since $v$ is squared in eq. 6.1, the value of the function is the same for $v$
and $-v$.

The proposed FF ANN executes the following equations, where $i = 1, 2, ..., NH$:
where $f_i$ denotes $f$ in eq. 6.1 with $v$ replaced by $v_i$; $NH = \text{number of hidden units}$; $w^h = \text{weights from the input to the hidden units (input weights)}$; $w^{ou} = \text{weights from the hidden units to the output (output weights)}$; $out^h = \text{output of the hidden units}$. Figure 6.3 shows a diagram of such a FF ANN.

Observe in the above equations that, in contrast to a standard FF ANN using sigmoidal (squashing) functions: a) the input $x$ and the parameter $x_0$ use input weights with the same absolute value but with opposite signs; b) there is no bias term for the hidden units, only for the output unit; and c) the output weights are squared such that the output of the hidden units are always weighted by non-negative numbers. By squaring the output weights we are including the assumption that the second derivative of true function is always non-positive (true function is a hill, not a valley), therefore we avoid the possibility that the contribution of any hidden unit would be, even temporarily before convergence is achieved, used to approximate a valley. Consequently, if there is at least one hidden unit with a non-zero input weight, a non-zero output weight and non-zero associated parameter $v$, then $x_0$ is the only value of $x$ that maximizes the network output and the network output function has only one extremum point. The formal proof of this important result is delayed until section 6.6 where it is
proved for the multi-input case (Theorem 6.1).

In relation to estimating the network parameters (training the network) the
difference is that the output value of one of the input units \((x_0)\) and the coefficients of
asymmetry of the hidden units \((v)\) are extra network parameters that need to be adjusted
as well.

Each of the ANN parameters executes a different role: the input weights and the
coefficients of asymmetry control the width and the asymmetry of the response of each
hidden unit, the output weights control the influence of each hidden unit in the ANN
output, the output bias and \(x_0\) control respectively the vertical and horizontal shift of the
ANN response.

The number of parameters that need to be adjusted by a training algorithm are:
\(NH\) input weights + \(NH\) output weights + \(NH\) \(v\)'s + 1 bias + 1 \(x_0\), making a total of
\(3NH + 2\) parameters.

6.4 - Training the Neural Network

The Back-Propagation algorithm (chapter 2) can be directly applied to the
proposed FF ANN, with some extra equations added to take into consideration the non-
standard ANN structure. The basic idea is, at each time step \(k\), to change the ANN
parameters in the direction that decreases the cost function \(E\) given by:

\[
E = \frac{1}{2} \left( y - y^{NN} \right)^2 = \frac{1}{2} e^2
\]  

(6.5)

Denoting by \(P\) the vector that contains all ANN parameters \((NH\) input weights, \(NH\)
output weights, \(NH\) \(v\)'s, bias and \(x_0\)) and by \(p\) one of its elements, then:

\[
P_{k+1} = P_k - \eta_k \frac{\partial E}{\partial P} (P = P_k)
\]  

(6.6)

where \(\eta_k\) is the learning rate for the ANN parameter \(p\) at time step \(k\) and the term \(\partial E/\partial p\)
is calculated analytically using the chain rule and evaluated using the numerical
parameter values of \(P\) at time step \(k\). For instance, the term \(\partial E/\partial x_0\) for \(x_0\) is:

\[
\frac{\partial E}{\partial x_0} = \frac{\partial E}{\partial y^{NN}} \sum_{i=1}^{NH} \frac{\partial y^{NN}}{\partial out_i^h} \frac{\partial out_i^h}{\partial net_i} \frac{\partial net_i}{\partial x_0}
\]  

(6.7)

or:
\[ \frac{\partial E}{\partial x_0} = -e \sum_{i=1}^{NH} (w_i^{ou})^2 \frac{\partial out^h_i}{\partial net_i} (-w_i^h) \]  

(6.8)

Analogously, the following equations apply for the other network parameters, where \( i = 1, 2, ..., NH \):

\[ \frac{\partial E}{\partial v_i} = -e (w_i^{ou})^2 \frac{\partial out^h_i}{\partial v_i} \]  

(6.9)

\[ \frac{\partial E}{\partial w_i^h} = -e (w_i^{ou})^2 \frac{\partial out^h_i}{\partial net_i} (x - x_0) \]  

(6.10)

\[ \frac{\partial E}{\partial w_i^{ou}} = -2 e w_i^{ou} out^h_i \]  

(6.11)

\[ \frac{\partial E}{\partial bias} = \frac{\partial E}{\partial y^{NN}} \frac{\partial y^{NN}}{\partial bias} = -e \]  

(6.12)

where:

\[ \frac{\partial out^h_i}{\partial net_i} = -f_i \left[ \frac{v_i^2}{2 + v_i^2} \right] \exp(net_i) - \exp(-v_i^2 net_i) \]  

(6.13)

\[ \frac{\partial out^h_i}{\partial v_i} = f_i \left[ \frac{2 v_i}{2 + v_i^2} \right] \left\{ 1 - f_i \left[ \exp(net_i) - net_i \exp(-v_i^2 net_i) \right] \right\} \]  

(6.14)

and \( f_i \) denotes \( f(net_i) \) which is given by eq. 6.1 with \( v \) replaced by \( v_i \).

Finally, the Neural Network Extremum Controller algorithm can be summarized in the following steps:

1) Initialize the ANN parameter vector \( P = P_0 \);
2) Set \( k = 1 \);
3) Generate \( dither_k \);
4) Set the system input to: \( x_k = x_{0_k} + dither_k \);
5) Collect the true system output \( y_k \);
6) Use eqs. 6.2-6.4 to calculate the ANN output \( y_k^{NN} \);
7) Use eqs. 6.6-6.12 to update the ANN parameter vector \( P \), i.e. update \( P_{k-1} \) to \( P_k \);  
8) Increment the time step counter: \( k = k + 1 \), and loop back to step 3.
Note that \( x_0 \) is one of the elements of the vector \( P \).

6.5 - Simulation of a Single Input Example

The following simulation of the proposed single input ANN extremum controller illustrates the above concepts.

The true system output is expressed by the following smooth asymmetric function:

\[
y = \begin{cases} 
a_1(x-x_0)^2, & \text{if } x \leq x_0 \\
-a_2(x-x_0)^2, & \text{if } x > x_0
\end{cases}
\]  

The measurements of the true system output are corrupted by zero mean gaussian noise. In this simulation we use: \( a_1 = 0.25, a_2 = 0.1, x_0 = -2 \) (the true optimum input), standard deviation of the output noise = 0.1.

To construct the ANN, 6 hidden units were used and the output weights, the input weights and the parameters \( v \) were initialized by choosing, with equal probability, 1 or −1 and then adding a small random number with zero mean and uniformly distributed on the interval \([-l l]\), where \( l = 0.4 \) for the input and output weights and \( l = 0.3 \) for the parameters \( v \). Additionally, the initial output weights were then divided by 3, in order to limit the initial maximum ANN output. The parameter bias was set to 0.4 and the initial estimate of \( x_0 \) to −4.

The simulation covered 400 time steps (iterations), and the dither was generated as a random number with uniform distribution in the interval \([-3 3]\) throughout the simulation, in other words, with no decay in the dither amplitude. A global decreasing learning rate \( \eta \) was defined for each time step \( k \) as:

- for \( 1 \leq k \leq 99 \): \( \eta_k = 1.00*0.6 \);
- for \( 100 \leq k \leq 199 \): \( \eta_k = 0.75*0.6 \);
- for \( 200 \leq k \leq 299 \): \( \eta_k = 0.50*0.6 \);
- for \( 300 \leq k \leq 400 \): \( \eta_k = 0.25*0.6 \);

The learning rates for the ANN \( x_0 \), the input weights and the parameters \( v \) were set at each time step to be equal to the global learning rate. The learning rates for the output weights and the output bias were set at each time step to be a tenth of the global learning rate. The basic idea is to have different learning rates for different types of
parameters since they perform different tasks, as was explained in section 6.3.

At each time step, the ANN parameter routine was executed 3 times using the above learning rates, i.e. the initial point in the parameter space that is used to calculate the direction of change was updated twice within each time step.

Figure 6.4 shows the true function and the ANN approximation before and after being trained. Figures 6.5 and 6.6 show respectively how the ANN estimate of $x_0$ and the coefficients of asymmetry $v$ change during the ANN training.

From figures 6.4 and 6.5 we can see that after being trained: a) the neural network estimate of the true function remains in the band $\pm 1$ measurement noise standard deviation for an interval larger than $x_0 \pm 2$, and b) despite the measurement noise and the large asymmetry, the neural network estimate of the optimum input has
only a small bias.

For comparison, we rerun the simulation program using a quadratic model, i.e. $y = -a^2(x-x_0)^2 + c$. Such a model corresponds to the FF ANN introduced before with just one hidden unit performing the function $out^h = -(net^h)^2$ and with the input weight fixed and equal to 1. The parameter $a$ corresponds therefore to the output weight. Other possibilities are to fix the output weight and to vary the input weight or to vary both weights.

The same gradient descent approach was used to update the 3 parameters of the quadratic model ($a$, $x_0$, $c$) and the same output noise and dither realizations were used. As before, a global decreasing learning rate was defined using the same intervals (1-99, 100-199, 200-299, 300-400) and the same multiplicative factors (1, 0.75, 0.5, 0.25) for a total of 400 time steps. However, the initial global learning rate was set to 0.3 instead of 0.6 and the parameter updating routine was called just once at each time step, instead of 3 times as before. The learning rate for each parameter of the model in relation to the global learning rate ($glr$) was set at each time step as: a) for parameter $a$: $glr/50$, for parameter $x_0$: $glr$, for parameter $c$: $2*glr$. The initial parameter estimates were: $a = (0.175)^{1/2}$, $x_0 = -4$, $c = 0.4$. Note that, for parameters $x_0$ and $c$, these are the same initial estimates used in the ANN model.

Figure 6.7 shows the time evolution of the estimate of $x_0$ for 3 runs, where each run used the quadratic model and the same setup but with different parameters for the true model (parameters $a_1$ and $a_2$ in eq. 6.15): 1) $(a_1,a_2)=(0.1,0.1)$, 2) $(a_1,a_2)=(0.25,0.25)$,

![Figure 6.7 - The time evolution of the estimate of the optimum input $x_0$ for the quadratic model when the true model is symmetric or asymmetric. Compare this figure with figure 6.5](image-url)
3) \((a_1, a_2) = (0.25, 0.1)\). Note that the square of the initial estimate of parameter \(a\) is the average of 0.1 and 0.25.

From fig. 6.7 we can see that, when the quadratic model is used and the true performance index is symmetric (quadratic in this case when \(a_1 = a_2\)), the estimate of the optimum input converges to the true value, otherwise there is large bias in the estimate. Figure 6.7 should be compared with fig. 6.5.

### 6.6 - An Extension to Multi-Input Extremum Control

The quadratic model can also be applied to the multi-input-single output case and some convergence results can be obtained as shown by Zarrop and Rommens [ZaRo93].

A possible and straightforward extension of the proposed artificial neural network extremum controller proposed in the previous sections to cover the multi-input-single output case is simply to modify the equation for the net input of each hidden unit. Equation 6.2 is then modified to:

\[
net_i^h = \sum_{j=1}^{NP} w_{ij}^h [x_j - x_{0j}]
\]

(6.16)

where \(NP\) = number of inputs (\(x\) and \(x_0\) become vectors with \(NP\) components) and \(w_{ij}^h\) = weight from input \([x_j - x_{0j}]\) to hidden unit \(i\). The other equations used to calculate the ANN output (eqs. 6.3 and 6.4) are kept the same.

In relation to training such an ANN, we can still apply the BP algorithm in the same way as in the single input case. Therefore, only eqs. 6.8 and 6.10 need to be modified to:

\[
\frac{\partial E}{\partial x_{0j}} = -e \sum_{i=1}^{NH} \left(w_{i}^{ou}\right)^2 \frac{\partial out_i^h}{\partial net_i} \left[-w_{ij}^h\right]
\]

(6.17)

\[
\frac{\partial E}{\partial w_{ij}^h} = -e \left(w_{ij}^{ou}\right)^2 \frac{\partial out_i^h}{\partial net_i} [x_j - x_{0j}]
\]

(6.18)

with \(i = 1, 2, ..., NH\) and \(j = 1, 2, ..., NP\) and the other equations are still applicable.

For such an ANN the number of parameters that need to be adjusted by a training algorithm are: \(NH*NP\) input weights + \(NH\) output weights + \(NH\) \(v\)'s + 1 bias + \(NP\) components of \(x_0\) making a total of \((NP+2)*NH + NP + 1\) parameters.
In a two dimensional input space the output of each hidden unit will be a ridge. Such ridges will cross each other at the point \( x_0 \) and it is necessary at least two ridges with different orientations to uniquely define the point \( x_0 \). The following theorem generalizes this statement for any number of inputs [NZM92].

**Theorem 6.1:** If \( x \) and \( x_0 \) are vectors with real finite components and if the ANN output is given by eqs. 6.16, 6.3, 6.1 and 6.4, then the ANN output has a unique maximum point and it occurs when \( x = x_0 \), given that, within the set of hidden units, we can select a subset such that:

C.1) the number of selected hidden units (\( NH \)) is greater or equal to the number of inputs (\( NP \));

C.2) the selected hidden units have non-zero output weights and non-zero coefficients of asymmetry;

C.3) the input weights of the selected hidden units form a full rank matrix, i.e a matrix with rank equal to the number of inputs.

**Proof:** We need to prove that for a finite real \( x \), \( \partial y_{NN}/\partial x = [\partial y_{NN}/\partial x_1, ..., \partial y_{NN}/\partial x_{NP}]^T = 0 \) if and only if \( x = x_0 \), where \( \text{dim} \ x = NP \) and \( 0 = [0, ..., 0]^T \). Without loss of generality we can assume that \( x_0 = 0 \), since \( x_0 \) can be seen as a translation of the origin of the coordinate system \([x_1, ..., x_{NP}]\).

**Sufficiency:** If \( x = 0 \) then \( \text{net}_i^h = 0 \), where \( i = 1, ..., NH \). From P.2 in section 6.3, we have that \( \partial f(\text{net}_i^h)/\partial \text{net}_i^h = 0 \) when \( \text{net}_i^h = 0 \). Therefore, from eq. 6.4 if \( x = 0 \) then \( \partial y_{NN}/\partial x = 0 \).

**Necessity:** If \( \partial y_{NN}/\partial x = 0 \), then:

\[
\frac{\partial y_{NN}}{\partial x_j} = \sum_{i=1}^{NH} \left( w_{ij} \right)^2 \frac{\partial f_i}{\partial \text{net}_i^h} w_{ij} = 0
\]  

(6.19)

and consequently we can write:

\[
\sum_{j=1}^{NP} x_j \frac{\partial y_{NN}}{\partial x_j} = 0
\]  

(6.20)

Now, by combining eqs. 6.19 and 6.20 we have:
Finally:

\[
\sum_{j=1}^{NP} \sum_{i=1}^{NH} x_j w_{ij}^w \frac{\partial f_i}{\partial \text{net}_h^i} w_{ij}^h = 0
\]  
(6.21)

\[
\sum_{i=1}^{NH} \left( \sum_{j=1}^{NP} x_j w_{ij}^w \text{net}_h^i \right)^2 \frac{\partial f_i}{\partial \text{net}_h^i} \left[ \sum_{j=1}^{NP} w_{ij}^h x_j \right] = 0
\]  
(6.22)

Finally:

\[
\sum_{i=1}^{NH} \left( \sum_{j=1}^{NP} x_j w_{ij}^w \text{net}_h^i \right)^2 \frac{\partial f_i}{\partial \text{net}_h^i} \text{net}_h^i = 0
\]  
(6.23)

This last equation implies that, if the output weights \(w^w\) and the \(v\)'s are different from zero, then \(\text{net}_h^i = 0\) (from fig. 6.2 and eqs. 6.1 and 6.13 we can see that: 1) if \(v_i = 0\) then \(f_i = 1\) and \(\partial f_i/\partial \text{net}_h^i = 0\); 2) if \(v_i \neq 0\) and \(\text{net}_h^i \neq 0\) then \(\partial f_i/\partial \text{net}_h^i < 0\) for finite \(\text{net}_h^i\)).

Since \(\text{net}_h^i = w^h x\), where \(w^h\) is the input weight matrix with dimension \(NH\) by \(NP\) (\(NH\) is the number of selected hidden units), then if \(NH \geq NP\) and if \(w^h\) has full rank, i.e. rank \(NP\), then \(\text{net}_h^i = 0\) implies \(x = 0\). Therefore, given the previous conditions, \(\partial y^{NN}/\partial x = 0\) implies \(x = 0\), as we want to show.

\[\square\]

6.7 - Simulation of a Two Input Example

As an example of the application of the proposed ANN-based multi-input extremum controller, we discuss in this section the practical problem of locating an object within a larger image ([NZM92],[NZM93]). The basic idea is to find the parameters of a mask with a pre-determined shape in order to maximize a function that is related to the position, size and orientation of the object image. Here we assume that such an image is available as a set of pixels. For each pixel, there is an associated pixel value (its gray level) and a pixel position (in real-world coordinates). Using a discrete mass analogy, we use the interpretation that the pixel values are like "masses" concentrated at the center of the pixel area. Therefore a function that uses the pixel positions as continuous input variables will have discontinuities.

We illustrate the application of the proposed neural network extremum controller in this area by using a simplified version of this problem. We assume that we already know that the object has a rectangular shape with a known width and orientation but unknown length and position (within certain limits). Moreover, we know that the object
lies along the horizontal axis with a known vertical position and the background of the image (all pixels that do not belong to the object) has all pixel values set to zero.

Let’s assume for this example that the object image is specified by the following vectors:

\[ X_{Obj\text{Pos}} = [10, 10.5, 11, ..., 15.5, 16], \]
\[ X_{Obj\text{Pix}} = [0.5, ..., 0.5] \]

In other words, the center of leftmost pixel of the object image is at position \( X_{Obj} = 10 \) units, the object image size is \( S_{Obj} = 6 \) units, the horizontal distance between the centers of two neighbouring pixels is 0.5 units and the object image contains 13 pixels, each one with a 0.5 value. So, the unknown true optimum input value is \( x_0 = [X_{Obj}, S_{Obj}]^T = [10, 6]^T \). Figure 6.8 illustrates the object position and size.

The algorithm starts by setting the initial values for the mask parameters, in this case, the position of the left side of the mask = \( X_{Mask} \) and the mask length = \( S_{Mask} \). Looking only inside the mask, i.e. using only local information, we add the value of all pixels that are inside the mask. This gives us a measure of the overlapping area between the object and the mask. However, we need to put some penalty on the size of the mask, because if the mask can become very large the object will be completely inside the mask for a range of mask sizes and we will not notice any change in the summation of the value of all pixels inside the mask. So, we need to define a function (our performance index) of the mask size and the overlapping area that has only one extreme point (a maximum point) and the input that produces this extremum point is the correct mask position and size. There are several ways to define such a function. One possibility for

![Figure 6.8 - The object image and the initial features of the mask](image-url)
this performance index is:

\[
y = L^* \left( \frac{L^*}{S^*} \right)^\alpha
\]

(6.24)

where \( L^* = L / L_{\text{max}} \), \( L = \) summation of the value of all pixels inside the mask and \( L_{\text{max}} = \) estimate of \( \text{max}(L) = \) estimate of the summation of all object pixels values; \( S^* = S_{\text{Mask}} / S_{\text{O max}} \), \( S_{\text{O max}} = \) estimate of an upper limit for the size of the object and \( \alpha \) is a real positive number. We use \( \alpha = 2 \) in this simulation. The parameters \( L_{\text{max}} \) and \( S_{\text{O max}} \) are kept constant throughout the simulation and they act as normalization constants setting the maximum value of the performance index. The above function is maximum when \( L \) is maximum, that is when the mask contains the whole object, and \( S_{\text{Mask}} \) is the smallest value such that \( L \) is still at its maximum. Note that for a fixed object \( L \) is a function of \( X_{\text{Mask}} \) and \( S_{\text{Mask}} \).

If we assume that we can initialize \( X_{\text{Mask}} \) and \( S_{\text{Mask}} \) such that we know that the whole object is within the mask, then we can set \( L_{\text{max}} \) to \( \text{max}(L) \) by simply adding all pixels values within the mask in the first iteration. This is the procedure adopted here and illustrated in figure 6.8. A value of 15 is used for \( S_{\text{O max}} \) and for the initial \( S_{\text{Mask}} \). The parameter \( X_{\text{Mask}} \) was initialized as 5 (see fig. 6.8). Since the maximum possible value of \( L^* \) is 1, the true performance index has an unknown maximum value of \( (SO_{\text{Obj}} / SO_{\text{O max}})^\alpha = (6/15)^2 = 6.25 \) for \( [X_{\text{Mask}}, S_{\text{Mask}}] = [10, 6] \).

The ANN input and output weights and the \( v \)’s were initialized as in the single input example. The parameter \( \text{bias} \) was initialized to zero and, as explained before, \( x_0 \) was initialized to \([5, 15]^T\).

Other parameters used in the simulation were: six hidden units, dither uniformly distributed in the range \([-3 3]\) without decay for both input variables, 400 time steps (or iterations), a constant global learning rate \( glr \) of 0.1/6, learning rate (\( lr \)) for output weights = \( glr/10 \), \( lr \) for \( v \)’s = \( glr/5 \), \( lr \) for input weights = \( glr \), \( lr \) for \( x_0 = glr*10 \), \( lr \) for \( \text{bias} = glr \), the parameter updating routine was called 3 times in each time step.

Figure 6.9 shows the time evolution of the mask parameters. Although the true performance index is not a continuous function of the mask position and size, their estimates converged to values very close to the true optimum point (10 and 6 respectively). Figure 6.10 shows the output of the ANN at each time step for the estimate of \( x_0 \).
Figure 6.9 - The mask history. The object is situated between positions 10 and 16.  

Figure 6.10 - The output of the ANN for the estimate of optimum input $x_0$.  

Figure 6.11 - The true performance index as a function of the mask position and size, its ANN approximation after being trained and the respective contour plots.
Figure 6.11 shows the true performance index as a function of the mask position and size. This same figure also shows the ANN approximation after being trained and the respective contour plots.

Figure 6.12 shows the superimposed contour plots of the true performance index and its network approximation, highlighting the region of the dither ([-3, 3] for both dimensions) around the optimum point [10, 6]. Figure 6.13 shows the same contour plots for the case where the dither was increased to [-6, 6] in both dimensions and 600 times steps were used. Again we can see that the network converges to a value very close the optimum point.

6.8 - Conclusions

This chapter showed that the extremum control of an static asymmetric performance index with a non-decaying dither can be achieved by using an ANN approach. Basically the conceptual difference and the novelty is the use of an appropriate nonlinear modelling technique and the corresponding parameter estimation algorithm.

It was shown that the proposed technique applies to the single input and multi input cases. The proposed approach was compared with the quadratic approach in the single input case and it was shown that the use of the first approach results in a much smaller error in the estimation of the optimum input. A two input example was also simulated where, despite a large asymmetry in the true performance index and a large dither, the optimum input was estimated with a small error.
In the next chapter we show how ANN can be used for the control of nonlinear dynamical systems.
Chapter 7 - Dynamical Control Using Artificial Neural Networks

The field of control theory is currently well developed in the area of analysis and design of linear time-invariant dynamical systems. However, the area of control of nonlinear dynamical systems is much less advanced and few general results are available. Therefore each system is treated on a case-by-case basis. Among other features, the ability of feedforward ANNs to approximate arbitrary nonlinear mappings has attracted the attention of several control engineers since it provides a new approach to the difficult problem of nonlinear dynamical control.

In this chapter we firstly review some approaches proposed in the literature to integrate feedforward ANNs in the general control structure. The concept of feedback-error-learning is then introduced and we propose a feedback-error-learning control structure. Such a control structure is then mathematically analysed and we show that, at least for the case of single input-single output linear dynamical systems, when certain requirements are satisfied, the inverse dynamical model can be correctly identified.

In the subsequent section the technique of using a variable feedback controller is proposed. Simulations of the control of a two-joint robot by an ANN are presented and we show that the use of a variable feedback controller improves the performance of the neural controller in relation to other trajectories that were not used during training.

Finally, the concept of fault-tolerant ANN, introduced in chapter 5, is explored in order to improve the fault tolerance of the neural controller in relation to faults in the ANN. The control of an inverted pendulum is used as the simulation example in this case.

7.1 - Artificial Neural Networks and Dynamical Control

Hunt et. al. [HSZG92] highlight the following features of ANN as important to
their application in control:

1) ANNs have the theoretical ability to approximate arbitrary nonlinear mappings. There is also the possibility that such an ANN approximation is more parsimonious, i.e. it requires less parameters, than other competitive techniques such as orthogonal polynomials, splines, or Fourier series. However, this has yet to be theoretically proven [Son93].

2) Since ANNs can have multi-inputs and multi-outputs, they can be naturally used for control of multivariable systems.

3) ANNs can be trained off-line using past data records of the system to be controlled or they can be adapted on-line in order to compensate for changes in the controlled system.

4) Since ANNs are parallel distributed processing devices, they can be implemented in parallel hardware. Therefore, as a consequence of the currently possible very fast processing capability, ANNs can be used in real-time control. Also due to their distributed organization, ANNs have the possibility of offering, when properly trained, a good level of fault tolerance against internal damage to the network itself.

5) By using ANNs it may be possible to perform efficient sensor data fusion where symbolic and numerical information received from different types of sensor can be naturally integrated. Similar to performing sensor data fusion, one could perform actuator integration, where several actuators act on different systems without a one-to-one correspondence between actuator and system (each actuator provides signals to several systems and each system receive signals from several actuators). Such features could result in neural controllers that are robust to loss of sensors and actuators.

If one could obtain all these features simultaneously, the result would be very impressive: a real-time multi-variable nonlinear adaptive fault-tolerant controller. Nature, through evolution, has managed to achieve such controllers in biological systems. However, the current artificial neural and non-neural systems are still far from achieving such a level of refinement.
7.2 - Neural Control Architectures

In the previous chapters it was explained that one of the problems of applying ANNs is to decide how to construct them, e.g. to decide the number of layers, the number of units in each layer and the activation functions for each unit. When using ANNs for control, an additional problem is to decide how to integrate the ANN into the control structure, i.e. the neural control architecture. Werbos ([Wer90], [NaMc91]) classifies the current neural control architectures into five categories: supervised control, adaptive critic, back-propagation through time, direct inverse control and neural adaptive control. In this section we explore the basic characteristics of these neural control architectures.

7.2.1 - Supervised Control

The first neural net controller was probably the one built by B. Widrow and F. W. Smith at Stanford University in 1963 ([Wid87], [WiSm63]). It was used to control a motorized cart that had an inverted pendulum on its top. The cart moved along a uni-dimensional finite and straight track. The controller was required to balance the inverted pendulum in the upright position, keeping the cart position within certain bounds. The controller was of the bang-bang type, i.e. it could only apply a force of constant magnitude in the left or right direction.

Figure 7.1 illustrates the neural control architecture proposed by Widrow and Smith [WiSm63]. The basic idea is to use an already existing controller to train the neural controller. This type of neural control architecture is known as supervised control.
In their original 60’s work the teaching controller implemented a linear switching surface, i.e. the location of the switching surface was a linear function of the state variables of the system being controlled. With the switch shown in figure 7.1 in position A, the neural controller is being trained to approximate the desired switching surface provided by the teaching controller and the teaching controller still provides the control action for the system. When the switch is in position B, training stops and the ADALINE (a TLU with bipolar output) controls the system.

The network weights were adapted by using the LMS algorithm (chapter 2) such that the network is trained to associate its binary input variables to the desired bipolar output, the "bang-bang" control action.

The input variables were firstly encoded (for instance by a grandmother or "single spot" code) such that the network could approximate switching surfaces more complicated than hyperplanes. Such coding is in effect quantizing the state space. In this way it is possible to show that the ADALINE can approximate (except for quantization effects) any switching surface that does not contain cross-product terms, i.e. terms of the form \(x_i x_j\) where \(i \neq j\) [WiSm63].

Twenty five years after developing the original cart-pole balancer, Widrow and Viral Tolat [ToWi88] showed that the network could also be trained by using as input visual images of the cart-pole system, instead of using direct measurements of the state variables. Such a modification makes possible the use of a human as the teaching controller to provide the correct control actions. According to Hecht-Nielsen [Hec90], in order to use a human as the teaching controller it is necessary to build a computer simulation of the dynamical system and camera and then to run the simulation slowed down by a factor of 10 or 100, otherwise humans are not able to balance the pole.

This last case illustrates a major application of the supervised control architecture, i.e. to train the ANN to imitate a human expert. This can be specially useful in situations where it is not viable to have humans controlling the system continuously, for instance in dangerous environments. It is also important to note that it can be very difficult to formulate explicitly the rules used by humans to control a system. This is not a problem for the ANN since, when properly trained, it learns to extract such rules from the set of examples provided and therefore it is not necessary to provide an explicit formulation of the control law. Another important point is that the ANN could extract the information from the examples whether the input sensor data was
provided as direct measurements or as crude images. In other words, the ANN is flexible in relation to the particular representation used to code the training data.

On the other hand, one should note that the solution of the cart-pole balancer problem can also be obtained using classical control theory, as Geva and Sitte [GeSi93] and Hecht-Nielsen [Hec90] pointed out. The important point is the approach used, not the particular problem solved in this case per se.

7.2.2 - Adaptive Critic and Reinforcement Learning

In order to use the supervised control architecture it is necessary to have the existence of a "teacher", for instance a human expert in the particular control task, that can provide the correct action for every state of the system. For this reason it is said that in supervised control the ANN "learns with a teacher".

In some situations, however, no such a teacher is available. Furthermore, a series of actions have to be taken before it is possible to evaluate the success of the actions. In general, the success of such actions is simply evaluated as a "success" or as a "failure", i.e. the evaluation is not quantitative but only qualitative. In this case the ANN is said to "learn with a critic". The crucial problem in this case is known as the temporal credit assignment problem [HKP91], i.e. which actions should receive the blame in case of failure or the credit in case of success. This is different from structural (or spatial) credit assignment, where the problem is to attribute the network output error to different units or weights.

The idea of the adaptive critic neural architecture is based on the principle of reinforcement learning, a term borrowed from the theories of animal learning, i.e. reward the correct actions and punish the wrong actions. In 1973 Widrow, Gupta and Maitra proposed what they called selective bootstrap adaptation [WGM73]. Once a series of actions is taken and the result is evaluated as success or failure, the basic idea of selective bootstrap adaptation is as follows:

a) if the result is evaluated as a success, the actions are rewarded by applying the LMS algorithm to each action using as desired output the output already used in that action. They called this positive bootstrap adaptation or learning by reward.

b) if the result is evaluated as failure, the actions are punished by applying the LMS algorithm to each action using as desired output the
inverse of the output used in that action. They called this negative bootstrap adaptation or learning by punishment.

Figure 7.2 illustrates the arrangement proposed by Widrow, Gupta and Maitra [WGM73] where the bootstrap control input determines if the action should be rewarded or punished by setting the position of the switch.

In order to illustrate the idea Widrow, Gupta and Maitra applied the bootstrap adaptation technique to a simulated card game of blackjack. By using a single ADALINE with the input variables properly coded (to a set of binary variables), they found out that the network could learn to play the game very well without explicitly knowing the rules or objective of the game. The only feedback provided by the environment is if the at the end of the game the network has won or lost. In each game the network has to take a series of actions which are recorded with the associated network inputs. If the network has won, the game is replayed, i.e. all input patterns are reapplied to the network input side, and the weights are adapted using positive bootstrap adaptation. Alternatively, if the network has lost, the weights are adapted using negative bootstrap adaptation. In this way, all experience acquired by playing the game and observing the final qualitative evaluation of a series of actions is stored in the network weights.

In 1983 Barto, Sutton and Anderson [BSA83] showed that the cart-pole problem could be solved by reinforcement learning by combining, as figure 7.3 illustrates, two units with a decoder that divides the state space into non-overlapping regions. The only evaluation of success or failure available is the failure signal, that is only non-zero when the pole falls more than a certain limit or the cart hits the track boundary. A new
learning trial begins after a failure signal is received. One unit is called the Associative Search Element (ASE) and it is responsible for selecting one of the possible actions (left or right) for each time step. The ASE is a stochastic unit, i.e. some source of randomness is used to force the network to explore the space of possible solutions. Before training the weights of the ASE unit are initialized such that, for any state of the car-pole system, both actions are equally probable. The other unit is called the Adaptive Critic Element (ACE) and it is responsible for generating an internal reinforcement signal.

The ASE could be trained using only the failure signal provided by the cart-pole system as the reinforcement signal. However, learning would be very slow since the reinforcement signal would be zero most of the time. The role of the ACE is exactly to learn how to generate an improved version of the external failure signal that can be used as the internal reinforcement signal. In this way the ASE can learn between the failures, not just when there is a failure. Therefore learning is considerably faster when compared with the case where only the ASE is used. However, in absolute terms learning can be very slow, which is not very surprising, given that feedback provided by the environment is delayed and qualitative.

The ACE learns its task using the method of Temporal Differences (TD) ([BSA83], [Sut88]). Basically the ACE learns to predict for each region of the system state space (indicated by the output of the decoder) the failure signal ([And89], [HKP91]), where the strength of the prediction indicates how soon the failure is expected to occur.
As the cart-pole moves from one region to the other, the internal reinforcement signal sent to the ASE is the difference between the predictions of failure for the current and previous regions. Actions that result in a increase in the prediction of failure are penalized. Actions that result in a decrease in the prediction of failure are rewarded.

Rosen, Goodwin and Vidal [RGV88] managed to speed up learning by adding two heuristic procedures: 1) the constant recurrence learning heuristic: reinforce any cycle, i.e. if a region has been visited more than once during one trial, reward it; 2) the short recurrence learning heuristic: reward a short cycle more than a longer one.

One important decision that has to taken in the design illustrated in figure 7.3 is to how to divide the system state space into non-overlapping regions. A very fine quantization allows a better approximation of complex functions but will require a longer time to train, while a coarse quantization will result in faster learning with worse approximation. Anderson ([And89], [And88]) showed that the combination of a decoder with two units (the ASE and the ACE) can be replaced by two multi-layer networks, i.e. with hidden units, that were called respectively the action and evaluation networks (see fig. 7.3). In this way the networks will develop, if learning is successful, their own adaptive representation of the system state space.

Rodrigues, Nascimento and Yoneyama [RNY91] also proposed to use a multi-layer network, which was trained using reinforcement learning, to tune the parameters of a PID controller until a pre-specified controller performance is achieved.

More recently some researchers have begun to investigate the relationships between reinforcement learning methods and dynamic programming and optimal control ([SBW92], [WaDa92]).

7.2.3 - Back-Propagation Through Time (BPTT)

In some situations the user can specify a cost function to be minimized (or maximized) that, as in the case of the adaptive critic, depends on a series of actions. However, since a cost function is specified, after the actions are executed, the user can evaluate the performance of the controller quantitatively. Furthermore, if a model of the plant (the system being controlled) can be developed, then the BPTT method is used to calculate the derivative of the cost function with respect to current actions [Wer90]. This derivative is then used to update the ANN responsible for generating the actions, i.e. the ANN controller. Figure 7.4 shows the location of the ANN controller in relation to the
The plant may be nonlinear and assumed to perform the nonlinear mapping \( x_{k+1} = A(x_k, u_k) \) where \( x \) are the plant states and \( u \) the plant inputs.

In order to illustrate the technique, Nguyen and Widrow [NgWi90b] used the BPTT method to train an ANN to back-track a truck with a trailer to a specific point of a loading dock, with the requirement that the back of the trailer is as parallel as possible to the loading dock.

The first task is to develop a model of the plant, which is basically to perform nonlinear plant identification. Nguyen and Widrow [NgWi90b] trained using the Back-Propagation algorithm a FF ANN to emulate the plant. The inputs to the ANN are the plant states \( x_k \) and plant inputs \( u_k \) and the desired ANN outputs are the plant states at the next time step \( x_{k+1} \) (see figure 7.5). It is normally assumed that the plant states are
directly observable without noise. The training of the ANN emulator consists of positioning the truck at an arbitrary position and applying a sequence of random inputs $u_k$.

After the ANN emulator is trained, the following procedure is used to train the ANN controller:

1) The ANN controller receives $x_0$ and generates $u_0$. The truck moves and generates $x_1$. Without updating the weights in both ANNs, the controller receives $x_1$ and generates $u_1$, and so on for a maximum number of steps (specified by the user) or until the truck hits the dock. This is the final position $x_{NT}$, where $NT$ is the number of time steps in this trial.

2) The state $x_{NT}$ is compared with the desired final state $x^d_{NT}$. The difference is used by the BP algorithm to update the weights of the ANN controller, while the weights of the ANN emulator are kept fixed. However, instead of being updated just once, the ANN controller is updated $NT$ times, as if a large network with $NT$ copies of the ANN emulator and ANN controller were used. Figure 7.6 illustrates this part of the procedure.

3) The truck and trailer are initialized at another position and steps 1 and 2 are repeated.

Strictly speaking all the weight changes for each of the $NT$ stages would have to be saved so that they could be added together at the end of the trial. In this way the ANN controller would only be updated at the end of each trial. In practice, however, the weight changes can be added immediately to the weights of the ANN controller as they are calculated.
Note that, when training the ANN controller, the real truck and trailer are used in the forward pass to obtain the final position of the trailer. However, the ANN emulator is needed by the BP algorithm such that the final error $e_{NT}$ (calculated at the output of the plant) can be back-propagated to change the weights of the ANN controller. Such an error cannot be back-propagated through the real plant but only through a suitable mathematical model, in this case the ANN emulator.

One important point is that Nguyen and Widrow [NgWi90b] used the interesting training strategy of dividing the training phase into several sessions, where each session was composed of several trials and the sessions had increasing levels of difficulty. It took about 20,000 backups to train the ANN controller, but afterwards the ANN could back the truck from several initial positions with which it had not been trained.

7.2.4 - Direct Inverse Control

In this neural control architecture the basic idea is to train an ANN as the inverse dynamical model of the plant. After being trained the ANN is simply used as a feedforward controller such that the composition of ANN and plant act as the identity mapping [Zur92]. The assumption in this case is that, at least in the space where the ANN is being used, the inverse dynamical model of the plant is uniquely defined and is stable.

It is possible to use a state-space formulation or an input-output formulation. In the former case the ANN inputs are the current state vector $x_k$ and the desired state at the next time step $x_{k+1}$. In the latter case the ANN inputs are: 1) the desired plant output at the next time step $y_{k+1}^d$; 2) the current and past plant outputs $y_k, y_{k-1}, ..., y_{k-Ny}$; and 3) the past plant inputs $u_{k-1}, ..., u_{k-Nu}$. In both cases the ANN output is $u_k$, the input that should be applied at the plant at time step $k$. Note that plant input and output at any particular time step can be vectors.

Psaltis, Sideris and Yamamura [PSY87] propose three neural control architectures based on the idea of learning the inverse dynamical model of the plant: Indirect Learning, Generalized Learning and Specialized Learning.

Assuming an input-output representation figure 7.7 shows the Indirect Learning architecture. Additional inputs to the ANNs, not shown in figure 7.7, are the past plant inputs and current and past plant outputs. Two ANNs are used. The ANN that receives the plant output as one of the inputs is trained to emulate the inverse dynamical model.
The other ANN is a copy of the ANN that is being trained.

One advantage of this architecture is that the ANN can be trained on-line, i.e. it learns while it is performing a useful task. Furthermore, since the ANN inputs are the desired plant outputs, the ANN can be trained on the particular region of interest of the output domain.

Sometimes, however, this architecture may not work as desired [PSY87]. The reason is that, according to simulations, the ANN converges to a set of weights for which a large number of different desired outputs $y_{k+1}^d$ are mapped to the same plant input $u_k$. The copy of the ANN then maps $y_{k+1}$, the output of the plant, to the same $u_k$. So the error used to correct the network is zero, although the total error $y_{k+1}^d - y_{k+1}$ will not be zero. One example is when the ANN response to any inputs is zero. Then its copy also has a zero output and the weights of both networks do not change. In other words, the problem is that this architecture is not goal-orientated, i.e. not orientated to decrease the plant output error.

The Generalized Learning architecture (figure 7.8) avoids this possibility by
generating directly the inputs $u_k$ to the plant and collecting the plant output $y_{k+1}$. Then plant output $y_{k+1}$ and the past plant output and past plant inputs are used as inputs to the ANN. The ANN output $v_k$ is compared with the respective plant input $u_k$. The disadvantage associated with this architecture is that it can only be used for off-line training, because it is necessary to have a training phase before the ANN can be used as a controller. During this training phase, another controller or a human expert is used to generate the inputs $u$ to the plant. Another point is that, in order to plan the training stage, it is necessary to know the input operational range of the plant. Consequently there is the risk of training the ANN on regions over which the plant will not operate during the control phase or of not training the ANN over some important regions.

The Specialized Learning architecture (figure 7.9), as in the case of Indirect Learning, uses as input to the ANN controller the desired response of the plant $y^d_{k+1}$ and applies the output of the ANN as the input to the plant $u_k$. However in the Specialized Learning architecture the plant response $y_{k+1}$ is compared with $y^d_{k+1}$ and the difference is used to train the ANN. This architecture can learn on-line, but to change the weights the error $y^d_{k+1} - y_{k+1}$ must be back-propagated through the plant, which is unknown or only approximately known. In order to do this, Psaltis et al. [PSY87] suggest interpreting the plant as an additional layer of the ANN with non-modifiable weights. They also suggest that when the plant is unknown the partial derivatives of the output of the plant in relation to its inputs (the so-called Jacobian of the plant) around some operating point can be estimated by some perturbation method such as by changing slightly each input and measuring the change at the output or by using the changes in the previous time steps.

Figure 7.9 - The Specialized Learning architecture
In order to back-propagate the plant output error through the plant Saerens and Soquet [SaSo91] proposed using the signs of the partial derivatives of the plant outputs in relation to its inputs since this qualitative knowledge is more easily available than the quantitative information. Using the state space approach to specify the ANN inputs they applied this scheme to train a two joint robot to follow a moving target in a 2D space and to solve the cart-pole problem with good results. However, they pointed out that this learning architecture can result in slow training.

Note that another possibility to back-propagate the error at the plant output in order to update the weights of the ANN controller is to train off-line an ANN to emulate the forward dynamical model of the plant as Jordan proposes [Jor89]. Then the error at the plant output is back-propagated not through the plant but through the ANN emulator.

It is possible to combine the advantages of these two learning architectures by first using the Generalized Learning architecture to train the ANN controller off-line over a large region of interest and then using the Specialized Learning to fine-tune it on-line around some specific operating points. The initial training using the Generalized Learning should speed up the convergence and make easier any necessary relearning when the plant changes or when new operating points are defined [PSY87].

7.2.5 - Neural Adaptive Control

The neural adaptive control architecture basically follows the same designs as conventional linear adaptive control, with the difference that the linear mappings used in the latter case are replaced by ANNs. Possible designs arise in Self-Tuning and Model-Reference adaptive control.

Narendra and Parthasarathy ([NaPa90], [Nar90]) provide several examples of how to perform neural adaptive control based on the model-reference approach, most of them for single-input single-output plants, but also some examples for the multi-input multi-output case.

The aim in model-reference adaptive control is to make the plant behave like a reference model, which must be supplied by the designer. First, an ANN learns to emulate the plant in the identification stage. This identification is initially carried out off-line. Later, the parameters of a second ANN, which acts as the controller, are updated using the model of the plant generated by the first ANN. The identification process continues also during the control stage, so that the model represented by the
ANN emulator is fine-tuned on-line. Figure 7.10 illustrates this two stage procedure (identification and then control), which is known as *Indirect Adaptive Control*.

One of the examples used by Narendra and Parthasarathy [NaPa90] is the control of a plant whose dynamics are unknown and described by:

\[ y_{k+1} = \frac{y_k}{1+y_k^2} + [u_k]^3 \]  \hspace{1cm} (7.1)

The identification model was chosen as two ANNs:

\[ \hat{y}_{k+1} = N_f[y_k] + N_g[u_k] \]  \hspace{1cm} (7.2)

where \( N_f[y_k] \) approximates \( f[y_k] = y_k / [1+y_k^2] \) and \( N_g[u_k] \) approximates \( g[u_k] = u_k^3 \). First a random input uniformly distributed over the interval \([-2,2]\] was used in the off-line identification stage. Hence, the \( N_g \) approximates \( g \) only over this interval. For inputs in that range, it was observed that the plant output \( y \) varied over the interval \([-10,10]\), so \( N_f \) approximates \( f \) only over the interval \(-10 \leq y \leq 10\).

If the reference model is: \( y_{m,k+1} = 0.6 y_{m,k} + r_k \), then the input to the plant could be calculated as:

\[ g[u_k] = y_{k+1} - f[y_k] - 0.6 y_k + r_k - f[y_k] \]  \hspace{1cm} (7.3)

Since the true function \( f[.\.\] is unknown, we have to use its approximation \( N_f[.\.\] . So the control input \( u_k \) can be generated using the following expression:

\[ u_k = \hat{g}_{inv}[N_f[y_k] + 0.6 y_k + r_k] \]  \hspace{1cm} (7.4)

where \( \hat{g}_{inv} \) denotes the approximation of the inverse of the function \( g[.\.\] . If another ANN \( N_c \) was adjusted so that \( N_g(N_c(r)) = r \), then this network \( N_c \) is the approximation of the
inverse of the function $g[.]$. The interval $[-4,4]$ was used to train off-line this last ANN, so that its output was within the interval used to train the network $N_c$. Finally, the control input $u_k$ is calculated by:

$$u_k = N_c \left[ -N_f[y_k] + 0.6y_k + r_k \right]$$  \hspace{1cm} (7.5)

Figure 7.11 shows the overall adaptive control system.

Paying attention to the fact that an ANN can only approximate the inverse or forward model of the plant over a specified finite region, Sanner and Slotine [SaSl92] and Tzirkel-Hancock and Fallside [TzFa92] proposed to add a sliding control term to the ANN controller. If the state of the plant is outside the region where the ANN has good approximation properties, then the ANN output and any ANN adaptation are turned off and only the sliding controller is used. Then, whenever the state of the plant returns to the region where the ANN provides a good approximation the sliding controller output is turned off, the ANN regains control of the plant and ANN training proceeds. A modulation mechanism [SaSl92] is provided to achieve a smooth transition between the ANN and sliding controllers.

Willis et al. ([WDMT91], [WMDT92]) and Saint-Donat et al. [SBM91] proposed neural predictive control methods, i.e. to use the adaptive model of the plant developed by the ANN to calculate the control action in order to minimise the sum of squares of future setpoint tracking errors. The time interval considered for such minimization is called the output prediction horizon. Another innovation proposed by Willis et al.
([WDMT91], [WMDT92]) was the use of ANNs composed of sigmoidal units where their outputs are filtered in time, for instance by a first-order low-pass filter:

\[ y'_k = \lambda y'_{k+1} + (1 - \lambda) y_k \]  \hfill (7.6)

where \( \lambda \) is the time constant and \( 0 \leq \lambda \leq 1 \). The basic idea is that by including dynamics in the network model in this way the ANN modelling capability will hopefully be expanded and therefore a smaller ANN (perhaps with a smaller number of delayed plant inputs and outputs as the ANN input) could be used. Since the filter constant \( \lambda \) is not known beforehand it may be necessary to adjust \( \lambda \) during training. In order to use the BP algorithm to adjust \( \lambda \) it is necessary to calculate the gradient of the cost (error) function in respect to \( \lambda \). A possible approach suggested by Willis et al. ([WDMT91], [WMDT92]) to determine \( \lambda \) is to use the chemotaxis algorithm, an algorithm similar to the Simulated Annealing technique. The chemotaxis algorithm assumes that the ANN parameters follow a multivariate gaussian distribution with zero mean. The ANN weights and any other possible parameters are adjusted simply by adding gaussian distributed random values to them. The new parameters are accepted if such adjustments result in a smaller prediction error.

Finally, for the benefit of completeness, we should mention that it is also possible to use unsupervised learning algorithms to train ANN to be used in control systems. For instance, Ritter, Martinetz and Schulten [RMS92] shows how the Kohonen algorithm can be applied to solve the dynamical control problem for a three joint robot arm.

7.3 - The Feedback-Error-Learning Method

In 1987 Kawato and co-workers ([KFS87], [MKSS88]) proposed the feedback-error-learning method to train an ANN to perform dynamical control of a robotic manipulator. Their main motivation was to propose a model of the computational scheme used in the central nervous system (CNS) for motor learning. Their basic idea is to combine an already available and tuned conventional feedback controller with an ANN acting as the feedforward controller. The feedback controller should at least be good enough to stabilize the plant when used alone, but it does not need to be optimally tuned. For simplicity such a feedback controller is normally a PID controller.

The aim is to adapt the ANN in order to minimize the tracking error \(|e|\) defined
as the difference between a reference signal $\text{ref}$ and the measured output $y$, normally a subset of the state vector $x$. In order to achieve this Kawato [MKSS88] proposed using the output of the feedback controller as the ANN output error and therefore called such a learning method Feedback-Error-Learning. Although apparently Kawato and his co-workers have not realized this, the feedback-error-learning rule can be interpreted as a method of minimizing the mean-squared-value of the output of the feedback controller, as we will show later.

It is important to note that by using the feedback error signal as the ANN output error the problem of back-propagating the control error through the plant (or through a model of the plant) is avoided [MKSS88]. Furthermore, the ANN can be trained on-line and the training method is goal orientated since, when the output tracking error is zero, the output of the feedback controller will also be zero (in reality if there is a integral component in the feedback controller, its output can be a non-zero constant, in which case a bias term at the linear output unit of the ANN can be used to cancel such constant output).

Before being trained, the ANN is initialized such that its output is zero for any input. Hopefully, as the ANN is being trained, it will smoothly take over control from the feedback controller and at the same time improve the overall control performance. In this way the ANN is being trained to be the inverse dynamical model of the plant.

### 7.3.1 - The Original Feedback-Error-Learning Control Structure

Since Kawato was mainly concerned with robot dynamical control he proposed composing the ANN input by using, at each time step, the desired angle joints and their respective desired velocities and desired accelerations. He also proposed that the ANN could be simply a linear combination of nonlinear functions, where these nonlinear functions could be determined by analysing the inverse dynamical robot model. Therefore, for a robot with $N$ joints, the feedforward controller would be formed by $N$ ANNs, each with $3N$ inputs and 1 linear output.

The number of nonlinear functions to be used in each ANN is a function of the particular dynamical robot model chosen. Figure 7.12 illustrates such a control structure for a two-joint robot model. Note that in each ANN there is only one layer of modifiable weights and therefore there is no need for using an algorithm such as Back-Propagation since there are no hidden weights to be adjusted. However there is also the
considerable inconvenience for the designer of specifying the nonlinear functions that will basically perform a nonlinear transformation of the desired angles, velocities and accelerations. One simple way to avoid such inconvenience is to use an ANN composed, for instance, of sigmoidal units. In [Kaw90] Kawato presents a few incomplete simulation results using this approach for a 3-joint robot arm.

In fig. 7.12 the following notation was adopted: $q^d_k, q_k =$ desired and actual joint angle at time step $k$, $T^f_k = $ feedback torque generated as the output of the feedback controller, $T^v_k = $ feedforward torque generated as the ANN output, and $T_k =$ torque applied to each robot joint. From fig. 7.12 we have: $T_k = T^f_k + T^v_k$.

Since the output of the feedback controller $T^f_k$ is used as the ANN output error, the role of the training algorithm can be interpreted as being to adapt the network weights in order to associate the vector $q^d_k$ and its derivatives with the vector $T_k$. Since in the robot control problem we can define the plant states as the joint position and joint velocities, the above is equivalent to mapping from $[(x^d_k)^T (x_{k+1}^d)^T]$ to $T_k$, i.e. the desired trajectory for each joint with the torque that should be applied to each joint.
Since in order to learn the true inverse dynamical model of the plant, it is necessary to associate the vector $[(x_k)^T (x_{k+1})^T]$ to $T_k$, the implicit assumption of the feedback-error-learning method is that the feedback controller is adequate so that, when used alone to control the plant, the plant *approximately* follows the correct trajectory, i.e. $[(x_k^d)^T (x_{k+1}^d)^T] \approx [(x_k)^T (x_{k+1})^T]$ [NaMc91]. Therefore one can state that the role of the feedback controller is to provide an approximate solution for the problem. Because of this, when used within such a control structure, the ANN tends to be trained more rapidly than in other situations.

The Feedback-Error-Learning method can be seen as a special case of the adaptive critic neural control architectures (section 7.2.2) [Wer90], where the feedback controller performs the role of the critic. In this case however, the critic is in general (but not necessarily) non-adaptive, as we show later in this chapter.

The role of the critic (the feedback controller) is not to provide the desired ANN output, but to provide a signal that can be used as an evaluation of the ANN performance. The role of the training algorithm is to adapt the ANN in order to maximize the evaluation signal provided by the critic, which means minimizing the amplitude of the feedback controller output.

Kawato’s interpretation is to see the feedback controller as an imperfect teacher and that, by being trained by this teacher, the ANN learns to surpass the teacher [KFS87]. Such an interpretation, however, is incorrect since the ANN is not trained to imitate the feedback controller but to make the output of the feedback controller (and consequently the output error) as small as possible. In other words, the feedback controller provides the ANN output error, not the ANN output.

7.3.2 - The Modified Feedback-Error-Learning Control Structure

In this section we propose some modifications to the feedback-error-learning control structure proposed by Kawato and develop a mathematical formulation for the learning rule used. Kawato’s original design (illustrated in figure 7.12) was appropriate to be used with robot manipulators. The modifications that we propose generalize the design allowing its use with a larger class of systems. Furthermore, we show, for single input-single output linear time-invariant plants, how the reference signal should be designed.

Kawato’s original design extracts the variation of the reference signal (the desired
joint angle profile) by using its higher order derivatives, normally calculated numerically using some central difference method. Therefore the first modification is to substitute these higher order derivatives by a tapped delay line [ONY91]. The ANN input becomes the current value of the reference signal and a limited number of its past samples. A limitation of such an arrangement is that the plant must be observable (i.e. from its output trajectory we can determine its state trajectory), since we are specifying the desired state trajectory of the plant by specifying only the desired trajectory of the plant output.

The second modification is to use a delayed version of the reference signal to calculate the input of the feedback controller, i.e. for single input-single output plants the output error is defined as $e_k = y_k - ref_{k-M}$, where $M$ is a non-negative integer. The use of such a delay was proposed by Widrow and Stearns [WiSt85] for a related non-neural controller architecture.

Figure 7.13 illustrates the proposed neural control structure for the feedback-error-learning method when the plant is single input-single output. The length of the tapped delay line $L$ (a non-negative integer) and the delay $M$ to be applied to the reference signal have to be specified by the designer. The same control structure can be applied in the case of MIMO (multi-input multi-output) plants simply by extending the ANN input vector to include all current values of the reference signal vector and its past
values, where the number of past samples to be collected for each component of the reference signal can be different for each component. The delay to be applied to each component of the reference signal in order to calculate the input of the feedback controller can also be different for each component. Each component of the feedback controller output is then used as the error for one of the ANN output units.

The use of the delay $M$ makes the task easier for the ANN since it then needs to implement an approximation of the delayed inverse of the plant. Without such a delay ($M = 0$), the ANN would have to implement the inverse of the plant, a more difficult task since the ANN would have to act as a predictor in order to compensate for the time that the physical plant takes to react to a new input. Since in many applications a delayed inverse is acceptable, the use of the delayed reference signal is not a major limitation.

One problem of trying to do inverse modeling is if the plant is linear and non-minimum phase. Such plants have inverse models that are unstable. However, the delayed inverse models of linear non-minimum phase plants have two-sided impulse responses, are stable and can be approximated by linear filters with a finite number of finite coefficients, the so-called FIR filters (finite impulse response filters) [WiSt85]. For this reason, when dealing with linear systems a good choice for the proposed control structure illustrated in fig. 7.13 is $L = 2M$, and an ANN without hidden units and with linear outputs, i.e. a linear ANN. In this way, the number of ANN weights are equally divided between the two sides of the impulse response. By including the delay $M$ the ANN can approximate delayed inverse models of minimum-phase and non-minimum-phase plants without the requirement of knowing a priori whether or not the plant is minimum-phase.

7.3.3 - Mathematical Analysis of the Modified Feedback-Error-Learning Control Structure

Let the plant be a single-input-single-output open-loop stable linear time-invariant dynamical system that, using a discrete-time domain notation, is described by the following equation:

$$y_k = a_1 y_{k-1} + a_2 y_{k-2} + \ldots + a_{Na} y_{k-Na} - b_0 u_k - b_1 u_{k-1} - b_2 u_{k-2} - \ldots - b_{Nb} u_{k-Nb} \quad (7.7)$$

or equivalently:
Designing for this case the ANN as a linear filter we have:

\[
G(z) = \frac{Y(z)}{U(z)} = \frac{b_0 + b_1 z^{-1} + \ldots + b_N z^{-N} \beta_0 + \beta_1 z^{-1} + \beta_2 z^{-2} + \ldots}{1 - a_1 z^{-1} - a_2 z^{-2} - \ldots - a_N z^{-N}} = \beta_0 + \beta_1 z^{-1} + \beta_2 z^{-2} + \ldots
\]  
(7.8)

From fig. 7.13 we have that:

\[
G^{\text{NN}}(z) = \frac{U^{\text{NN}}(z)}{\text{Ref}(z)} = \alpha_0 + \alpha_1 z^{-1} + \alpha_2 z^{-2} + \ldots - \alpha_L z^{-L}
\]  
(7.9)

Combining eqs. 7.8-7.12 we have:

\[
E(z) = z^{-M} \text{Ref}(z) - Y(z)
\]  
(7.10)

\[
G^{\text{FB}}(z) = \frac{U^{\text{FB}}(z)}{E(z)}
\]  
(7.11)

\[
U(z) = U^{\text{FB}}(z) + U^{\text{NN}}(z)
\]  
(7.12)

Combining eqs. 7.8-7.12 we have:

\[
Y(z) = G(z) U(z) = G(z) \left[ U^{\text{FB}}(z) + U^{\text{NN}}(z) \right]
\]  
(7.13)

\[
Y(z) = G(z) \left[ z^{-M} G^{\text{FB}}(z) + G^{\text{NN}}(z) \right] \text{Ref}(z) - G(z) G^{\text{FB}}(z) Y(z)
\]  
(7.14)

And finally:

\[
\frac{Y(z)}{\text{Ref}(z)} = G(z) \left[ z^{-M} G^{\text{FB}}(z) + G^{\text{NN}}(z) \right] \frac{1}{1 + G(z) G^{\text{FB}}(z)}
\]  
(7.15)

Following the same development, we have:

\[
U^{\text{FB}}(z) = G^{\text{FB}}(z) E(z) = G^{\text{FB}}(z) \left[ z^{-M} - \frac{Y(z)}{\text{Ref}(z)} \right] \text{Ref}(z)
\]  
(7.16)

\[
\frac{U^{\text{FB}}(z)}{\text{Ref}(z)} = G^{\text{FB}}(z) \left[ z^{-M} - G(z) G^{\text{NN}}(z) \right] \frac{1}{1 + G(z) G^{\text{FB}}(z)}
\]  
(7.17)

Therefore if \(G^{\text{NN}}(z) = z^{-M} / G(z)\), then from eqs. 7.15 and 7.17 we have:

a) \(Y(z) / \text{Ref}(z) = z^{-M}\), and

b) \(U^{\text{FB}}(z) / \text{Ref}(z) = E(z) / \text{Ref}(z) = 0\).

However, since \(G^{\text{NN}}(z)\) has a finite number of parameters (see eq. 7.9), the ANN can only be an approximation of the delayed inverse model of the plant. As the number of network parameters increases the network approximation improves and the magnitude of the output of the feedback controller decreases.

Let’s assume that the feedback loop without the ANN is stable such that the polynomials \(\gamma(z)\) and \(\phi(z)\) converge (i.e. \(\exists \ 0 < \lambda < 1, c > 0, d > 0, N > 0\) such that
|γ| < cλ and |φ| < dλ for all i ≥ N) where:

$$\gamma(z) = \frac{G^{FB}(z)}{1 + G(z) G^{FB}(z)} = \gamma_0 + \gamma_1 z^{-1} + \gamma_2 z^{-2} + \ldots \quad (7.18)$$

and

$$\phi(z) = \frac{G(z) G^{FB}(z)}{1 + G(z) G^{FB}(z)} = G(z) \gamma(z) = \phi_0 + \phi_1 z^{-1} + \phi_2 z^{-2} + \ldots \quad (7.19)$$

By substituting eqs. 7.18 and 7.19 in eq. 7.17 we can write:

$$U^{FB}(z) = \left[ z^{-M} \gamma(z) - \phi(z) G^{NN}(z) \right] Ref(z) \quad (7.20)$$

The learning problem can then be defined as finding the coefficients of the ANN, \((\alpha_0, \alpha_1, \ldots, \alpha_L)\) that minimize the square of the output of the feedback controller, i.e. we desire to minimize the following scalar cost function \(J\):

$$J = \frac{1}{2} E \left( u_{FB}^2 \right) = \frac{1}{2} E \left( \left( z^{-M} \gamma(z) Ref_k - \phi(z) \alpha^T ref^*_k \right)^2 \right) \quad (7.21)$$

where \(\alpha^* = [\alpha_0, \alpha_1, \ldots, \alpha_L]^T\) and \(ref^*_k = [ref_{k-1} ref_{k-2} \ldots ref_{k-L}]^T\) and both of these column vectors have \(L+1\) components where \(L \geq 0\).

The stationary points of \(J\) are given by \(\partial J / \partial \alpha^* = 0\) where:

$$\frac{\partial J}{\partial \alpha^*} = E \left[ u_{FB} \frac{\partial u_{FB}}{\partial \alpha^*} \right] = E \left[ u_{FB} \left( -\phi(z) ref^*_k \right) \right] = 0 \quad (7.22)$$

The cost function \(J\) has a unique minimum if the matrix \(\partial^2 J / \partial \alpha^{*2}\) is positive definite where:

$$\frac{\partial^2 J}{\partial \alpha^{*2}} = E \left[ \phi(z) ref^*_k \phi(z) ref^*_k \right] \Delta F^l \quad (7.23)$$

and \(F^l\) is the correlation matrix of the vector \(\phi(z) ref^*_k\) and it has dimensions \(L+1\) by \(L+1\). From eqs. 7.20 and 7.22 we have:

$$F^l \alpha^* = F^r \quad (7.24)$$

where:

$$F^r = E \left[ \left( z^{-M} \gamma(z) ref^*_k \right) \phi(z) ref^*_k \right] \quad (7.25)$$

and \(F^r\) is a column vector with \(L+1\) rows. Assume that the reference signal \(ref_k\) is stationary, i.e. \(E[ref_{k} ref_{k-i}] = \rho = \rho_{-i}\). Then element \((i', j')\) of \(F^l\) (denoted by \(F^l_{i', j'}\)) and element \(i'\) of \(F^r\) (denoted by \(F^r_{i'}\)) can be written as:
where $1 \leq i' \leq L+1$ and $1 \leq j' \leq L+1$. Note that: a) $F^l$ is a symmetric matrix with the Toeplitz (banded) structure common to covariances; and b) apart from having a stable closed-loop response, no other condition is imposed on the feedback controller.

Finally, if the reference signal is sufficiently exciting so that matrix $F^r$ is positive definite, then the system of linear equations given by eq. 7.24 can be solved. The calculated ANN parameters $\alpha_i$, $0 \leq i \leq L$, will be the set that minimises the mean value of the square of the feedback controller output. The particular values of $L$ and $M$ will determine how small the minimum is.

The following numerical example illustrates the use of the above equations. First let’s assume that the reference signal $ref_k$ is such that $\rho_i = \rho_{-i} = 0$ for $i \geq 2$. Then eqs. 7.26 and 7.27 can be rewritten as:

\begin{equation}
F^l' = E \left\{ \sum_{i=0}^{\infty} \sum_{j=0}^{\infty} \phi_i \rho_{i-j-(i' - j')} \sum_{i=0}^{\infty} \sum_{j=0}^{\infty} \phi_j \rho_{i-j-(j' - i')} \right\} = \sum_{i=0}^{\infty} \sum_{j=0}^{\infty} \phi_i \phi_j \rho_{i-j-(i' - j')} \tag{7.26}
\end{equation}

\begin{equation}
F^r' = E \left\{ \sum_{i=0}^{\infty} \sum_{j=0}^{\infty} \gamma_i \rho_{i-(i' + j')} \sum_{i=0}^{\infty} \sum_{j=0}^{\infty} \phi_j \rho_{i-(j' + i')} \right\} = \sum_{i=0}^{\infty} \sum_{j=0}^{\infty} \gamma_i \phi_j \rho_{i-(i' + j')} \tag{7.27}
\end{equation}

where the sequence $s_k$ is a white noise signal uniformly distributed between $-1$ and $1$.

If the scalars $S_0$ and $S_1$ are set respectively to 1 and 0.7, then by definition we have:

\begin{equation}
ref_k = S_0 s_k + S_1 s_{k-1} \tag{7.30}
\end{equation}

where the sequence $s_k$ is a white noise signal uniformly distributed between $-1$ and $1$. If the scalars $S_0$ and $S_1$ are set respectively to 1 and 0.7, then by definition we have:

\begin{equation}
E[s_k s_{k+i}] = \begin{cases} 
\rho_i = \frac{1}{3}, & \text{for } i = 0 \\
0, & \text{for } i \neq 0
\end{cases} \tag{7.31}
\end{equation}
Let $G(z) = z^{-1} + 0.5z^{-2}$, $G^{FB}(z) = k_p + k_i z^{-1} / (1 - z^{-1})$ (a PI controller) with $k_p = 0.6$, $k_i = 0.3$.

The poles of the polynomial $\phi(z)$ then have magnitudes 0.7822 and 0.6193. Truncating the infinite series in eqs. 7.28 and 7.29 after 61 terms, using $\rho_0 = 1.49 / 3$ and $\rho_1 = 0.7 / 3$, for the case $L = 3$ and $M = 1$ the numerical values for matrices $F^l$ and $F^r$ are:

$$F^r = \begin{bmatrix}
0.4087 & 0.1271 & -0.04924 & 0.07376 \\
0.1271 & 0.4087 & 0.1271 & -0.04924 \\
-0.04924 & 0.1271 & 0.4087 & 0.1271 \\
0.07376 & -0.04924 & 0.1271 & 0.4087
\end{bmatrix}$$

$$F^l = \begin{bmatrix}
0.3263 & -0.03604 & -0.03122 & 0.08937
\end{bmatrix}^T$$

and using eq. 7.24 we get:

$$\alpha^* = [0.9860, -0.4717, 0.2149, -0.08296]^T$$

Following the same procedure for $L = 7$ we get:

$$\alpha^* = [0.9965, -0.4994, 0.2491, -0.1236, 0.06061, -0.02865, 0.01279, -0.004863]^T$$

while the delayed inverse of the plant can be expressed as:

$$\frac{z^{-M}}{G(z)} = \sum_{i=0}^{\infty} \frac{1}{(-2)^i} z^{-i} = 1 - 0.5z^{-1} + 0.25z^{-2} - 0.125z^{-3} + 0.0625z^{-4} - 0.03125z^{-5} + 0.015625z^{-6} - 0.0078125z^{-7} + ...$$

The above analysis show that, under certain conditions, the cost function expressed in eq. 7.21 has a unique minimum and the ANN network parameters that minimize such a cost function can be used to form an approximation of the delayed inverse of the plant $G(z)$. At this point then we need a method that will search for this minimum point, i.e. a learning (or parameter estimation) algorithm. Denoting by $\hat{\alpha}_i(k)$ the estimated value of the ANN parameter $\alpha_i$ at time step $k$, we can simply use a gradient descent approach (as in the BP algorithm), i.e. to change the ANN parameters in the direction that decreases the cost function. Therefore we can write:
where $\eta$ is the learning rate, $i = 0, 1, \ldots, L$ and $\alpha^*(k) = [\alpha_0(k), \alpha_1(k), \ldots, \alpha_L(k)]^T$. Using eq. 7.22 we have:

$$\alpha_i(k+1) = \alpha_i(k) - \eta \left[ \frac{\partial J}{\partial \alpha_i} \right]_{\alpha^* - \alpha^*(k)}$$

(7.34)

However the polynomial $\phi(z)$, the closed-loop transfer function without the ANN, is not known a priori since it would imply knowledge of the transfer function of the plant $G(z)$. Therefore we propose to use the following learning rule:

$$\alpha_i(k+1) = \alpha_i(k) - \eta u_{k}^{FB} \sum_{j=0}^{\infty} \phi_j \, ref_{k-l-j}$$

(7.35)

This is the same rule that Kawato ([KFS87], [MKSS88]) uses in his original feedback-error-learning structure without a formal theoretical proof. Kawato argues that this learning rule was based on physiological information about the plasticity of biological neurons.

Strictly speaking the above rule would be correct only if the feedback controller is very good, in which case $\phi(z) \approx 1$, $y_k \approx ref_k - M$. Therefore training the ANN to associate its input $ref_k$ (and its past values $ref_{k-1}, \ldots, ref_{k-L}$) to the plant input $u_k$ is equivalent to training the ANN to associate $y_k$ (or more precisely $y_{k+L}, \ldots, y_{k+L-M-L}$) to $u_k$, i.e. training the ANN to learn the inverse of the dynamical model of the plant. Another equivalent interpretation is that the learning rule expressed in eq. 7.36 ignores any temporal correlation between the ANN output $u_{k}^{NN}$ and the plant input $u_k$. Since we can always write:

$$J = \frac{1}{2} E \left[ (u_{k}^{FB})^2 \right] - \frac{1}{2} E \left[ (u_k - (\alpha^*)^T ref_k^*)^2 \right]$$

(7.37)

if we could assume that $u_k$ is independent of $u_{k}^{NN}$ then eq. 7.22 could be written as:

$$\frac{\partial J}{\partial \alpha^*} = E \left[ \left( u_k - (\alpha^*)^T ref_k^* \right) \left( -ref_k^* \right) \right] = E \left[ -u_{k}^{FB} \, ref_k^* \right]$$

(7.38)

If we could assume that $\phi(z) \approx 1$, such an independence becomes evident if we use eq. 7.20 to write:
An important advantage of using the learning rule proposed in eq. 7.36 is that it can be readily generalized to the nonlinear case when the ANN can be, for instance, a multilayer perceptron and the BP algorithm is used to train the ANN, as the next sections will show.

Numerical simulations seem to support the use of the learning rule proposed in eq. 7.36. However, more theoretical work is needed to justify it formally. A possible fruitful line may be to interpret the output of the feedback controller as an approximation of the true ANN output error. This is the same as seeing the plant input \( u_k \) as an imperfect teacher [ShBr71] or, as in reinforcement learning and adaptive critic theory (see section 7.2.2), the output of the feedback controller more as a qualitative than quantitative performance index.

7.3.4 - Simulations for the Linear Case

In this section we present the results of simulations for linear plants. The example is a minimum-phase plant but the same principle could be used for a non-minimum-phase plant. In order to make comparisons between theoretical and experimental results easier the plant chosen is the same used in the numerical example shown in the previous section.

The feedback controller was implemented as a discrete-time PID (proportional-integral-derivative) controller at time step \( k \) using the following algorithm:

\[
e_k := ref_{k-M} - y_k \\
e^i := e^i + e_k \\
u_k^{FB} := k_p e_k + k_i e^i + k_d \left( e_k - e^d \right) \\
e^d := e_k
\]  

(7.40)

where \( k = 0, 1, ..., NT \), and the variables \( e^i \) and \( e^d \) were initialized as zero at the beginning of the simulation. The ANN was implemented as a FIR linear filter with an added bias, i.e. with no hidden units and a linear output such that:

\[
u_k^{NN} = bias - \sum_{i=0}^{L} \alpha_i ref_{k-i}
\]  

(7.41)

Consequently the number of parameters to be adjusted by the learning rule is \( L+2 \) and they were initialized as zero such that before being trained the ANN output is zero for
any reference signal used as input.

It is also assumed that the simulation starts at its zero state, i.e. \( y_k = 0 \) for \( k \leq 0 \) and \( u_k = 0 \) for \( k < 0 \). In relation to the reference signal it is assumed that \( ref_k = ref_0 \) for \( k < 0 \).

The ANN parameters were adjusted by the following learning rules:

\[
\alpha_i(k+1) = \alpha_i(k) + \eta u_k^{FB} \cdot ref_{k-i},
\]

\[
\text{bias}(k+1) = \text{bias}(k) + \eta u_k^{FB},
\]

where \( \eta \) = learning rate.

The plant is described by

\[
y_{k+1} = u_k + 0.5 u_{k-1}.
\]

The parameters of the feedback controller were set to \([k_p, k_i, k_d] = [0.6, 0.3, 0]\). The parameters \( L \) and \( M \) were set respectively to 3 and 1. The learning rate was set to 0.08/3. The total number of steps \( NT \) used in this simulation was set to 3600.

The reference signal \( ref_k \) was generated for \( 0 \leq k \leq 2000 \) as:

\[
ref_k = s_k + 0.7 s_{k-1}
\]

where, as explained in the previous section, the sequence \( s_k \) is generated to simulate

![Figure 7.14 - The plant output y and the error e](image-url)
white noise uniformly distributed between $-1$ and $1$.

For the period $2001 \leq k \leq 3600$ the reference period was a square wave with amplitude varying between $2$ and $-2$ and with period $400$ time steps. The ANN was trained only during the period $0 \leq k \leq 2000$.

Figure 7.14 shows the plant output $y$ and the error $e$ (defined in eq. 7.10 as the difference between the delayed reference signal $\text{ref}_{k-M}$ and the plant output $y_k$). Figure 7.15 shows how the $5$ ANN parameters vary during training. At the end of the training period (2000 time steps) their values were:

$$[\alpha_0 \quad \alpha_1 \quad \alpha_2 \quad \alpha_3 \quad \text{bias}] = [1.0077 \quad -0.5094 \quad 0.2252 \quad -0.0829 \quad 0.001774]$$

According to previous section the expected values for the ANN parameters are:

$$[\alpha_0 \quad \alpha_1 \quad \alpha_2 \quad \alpha_3 \quad \text{bias}] = [0.9860 \quad -0.4717 \quad 0.2149 \quad -0.0829 \quad 0]$$

Note in figure 7.14 that after $500$ time steps the error was very small. The plant output followed very closely the square wave reference signal, although the ANN was no longer being trained.

### 7.3.5 - Using a Variable Feedback Controller

In the next section we present the results of simulations where the plant is nonlinear (more specifically a robot with two revolute joints) and a variable feedback controller is used.

As we showed before, the aim of the control structure illustrated in fig. 7.13 is to perform closed-loop identification of the inverse dynamical model of the plant. If the
plant is assumed to be linear as in the previous section, the ANN can simply be a FIR filter with an added bias as in eq. 7.41. The addition of the delay $M$ was necessary to guarantee the existence of an approximate inverse dynamical model.

In the nonlinear case we have to assume that an inverse dynamical model also exists and to use an ANN capable of approximating nonlinearities, such as a FF ANN with nonlinear hidden units. In these simulations we will use the multi-layer perceptron (MLP) but in principle any other ANN model capable of nonlinear modelling, such as the RBF ANNs, could also be used. Again we have the problem of how to specify the ANN topology since the ANN must be such that it can approximate the inverse of the (assumed unknown) plant.

Another problem is to design the reference signal such that the ANN will converge to a close approximation of the inverse dynamical model of the plant. As in the linear case it is important that the reference signal is exciting enough so that important features of the plant appear and are detectable at its output. At the same time the reference signal must be such that the control of the plant and the dynamics of the ANN parameters are stable.

When trained with reference signals that are not very exciting (especially when the plant is nonlinear) the ANN can converge to an approximation of the inverse model that is very good but only for the training signal(s). If training stops and the ANN is used with reference signals different from the ones used during training, the quality of control can be worse than when only the feedback controller is used. In other words, the neural controller may not be able to generalize well from the training trajectory to other trajectories not seen before.

One technique that can be used to alleviate this problem is to use an extra degree of freedom available in the control structure, i.e. to vary slowly the gains of the feedback controller. Such an idea has also been used in the "classical" closed loop identification of linear systems (see [SoSt89], chapter 10) where it is possible to show that, by shifting between different feedback controllers, the accuracy of the estimates is increased. By varying slowly the gains of the feedback controller we hope to excite different plant modes and to make their effects noticeable at the output so that they can be better identified.

Assume a simple case where the training session consists of a set number of runs where in each run a fixed number of periods of the same reference signal is used. One
alternative to varying the gains of the feedback controller is to gradually reduce the
gains, for instance, by multiplying the gains of the previous run by a positive constant
($< 1$). By reducing the gains of the feedback controller we hope to decrease the quality
of the feedback controller and thereby increase the output errors. At the beginning of
training a stronger feedback signal may be more suitable since it may provide the
necessary stability, given that the ANN output is still not significant. Later on, a weaker
feedback controller may be more appropriate since it will increase the output tracking
errors and force the ANN to be trained in regions where it would not be trained
otherwise.

Another alternative, the one that was used in these simulations, is to adapt the
gains of the feedback controller according to some performance criteria for the previous
runs, for instance, to decrease the gains if the RMS (root-mean-square) error calculated
for the previous run was reduced and to increase the gains otherwise. This avoids the
situation where the quality of the feedback controller becomes too poor while the ANN
has not yet received enough training, and consequently the plant becomes unstable.

7.3.6 - Simulations for the Nonlinear Case with a Variable Feedback Controller

Assuming that the robot is moving freely in its workspace, its dynamical
equations, in general, can be written in vectorial notation as [Sch90]:

\[
\tau = D(q) \ddot{q} + H(q, \dot{q}) + G(q) + B(q) \tag{7.42}
\]

where:

- $\tau$ = vector of torques (for revolute joints) or forces (for prismatic joints) applied
  by the actuators at the joints of the robot arm;
- $q$ = generalized coordinates (angles if revolute joints or distances if prismatic
  joints);
- $D(q) \ddot{q}$ = acceleration term that represents the forces and torques generated by
  the motions of the links of the robot arm;
- $H(q, \dot{q}) = \text{Inertia Matrix; a symmetric and positive-definite matrix;}$
- $G(q) = \text{position term that represents the loading due to gravity;}$
- $B(q, \dot{q}) = \text{velocity term that represents the friction opposing the motion of the robot
  arm.}$
Moreover, this equation assumes that the robot arm is rigid and it does not include the actuator dynamics.

For a two-joint robot with revolute joints, assuming that the masses are concentrated at the joint ([LBDD88], [GuSe89]), we have (see figure 7.16):

\[
\tau_1 = D_{11} \ddot{q}_1 + D_{12} \ddot{q}_2 + H_1 (2 \dot{q}_1 \dot{q}_2 + \dot{q}_2^2) + B_1 + G_1
\]

\[
\tau_2 = D_{21} \ddot{q}_1 + D_{22} \ddot{q}_2 - H_1 \dot{q}_2^2 + B_2 + G_2
\]

where:

\[
D_{11} = (m_1 + m_2) d_1^2 + m_2 d_2^2 + 2 m_2 d_1 d_2 \cos(q_2)
\]

\[
D_{12} = D_{21} = m_2 d_2^2 + m_2 d_1 d_2 \cos(q_2)
\]

\[
D_{22} = m_2 d_2^2
\]

\[
H_1 = -m_2 d_1 d_2 \sin(q_2)
\]

\[
B_1 = b_{1v} \dot{q}_1 + b_{1c} \text{sign} (\dot{q}_1)
\]

\[
B_2 = b_{2v} \dot{q}_2 + b_{2c} \text{sign} (\dot{q}_2)
\]

\[
G_1 = (m_1 + m_2) g d_1 \sin(q_1) + m_2 g d_2 \sin(q_1 + q_2)
\]

\[
G_2 = m_2 g d_2 \sin(q_1 + q_2)
\]

and:

\[
m_1, m_2 = \text{masses of the links.}
\]

\[
d_1, d_2 = \text{lengths of the links}
\]

\[
b_{1v}, b_{2v} = \text{coefficients of viscous friction.}
\]

\[
b_{1c}, b_{2c} = \text{coefficients of coulomb friction.}
\]

\[
g = \text{gravitational constant.}
\]
The parameter $m_2$ includes the mass of any existing load.

For a state-space representation, we can define the state vector $\mathbf{x}$ as:

$$\mathbf{x} = [x_1, x_2, x_3, x_4]^T = [q_1, q_2, \dot{q}_1, \dot{q}_2]^T$$

(7.45)

Then, defining the following matrices:

$$E = \begin{bmatrix} D_{22} & -D_{12} \\ -D_{12} & D_{11} \end{bmatrix}$$

(7.46)

$$F = \begin{bmatrix} \tau_1 - H_1 \left(2x_1x_4 + x_4^2\right) - B_1 - G_1 \\ \tau_2 - H_1 x_3^2 - B_2 - G_2 \end{bmatrix}$$

(7.47)

where $\text{det}(E) = D_{11} D_{22} - (D_{12})^2$, the space-state dynamics are:

$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \\ \dot{x}_3 \\ \dot{x}_4 \end{bmatrix} = \begin{bmatrix} x_3 \\ x_4 \end{bmatrix}$$

(7.48)

$$\begin{bmatrix} \dot{x}_3 \\ \dot{x}_4 \end{bmatrix} = \begin{bmatrix} D_{11} & D_{12} \\ D_{12} & D_{22} \end{bmatrix}^{-1} \begin{bmatrix} E \\ F \end{bmatrix}$$

(7.49)

In relation to the diagram of the control structure illustrated in fig. 7.13 we have: $u =$ plant input $= [\tau_1, \tau_2]^T$, $y =$ plant output $= [q_1, q_2]^T$.

The simulations are divided into two groups. In the first group the ANN is trained using a fixed feedback controller (i.e. its gains are kept constant). In the second group a variable feedback controller is used to train the ANN. The same set of nine reference trajectories (RTs) are used in each group of simulations. The ANN is first trained using the first RT, and then tested with all 9 RTs. Even when the ANN is trained with a variable feedback, it is tested using the same original feedback gains used to perform the ANN training and testing in the other group of simulations. Finally we show that by training the ANN with a variable feedback controller the RMS tracking errors are significantly reduced, i.e. the ANN generalization to other RTs is improved.

The following parameters where used to simulate the robot arm: $m_1 = m_2 = 10$ Kg, $d_1 = d_2 = 1$ m, $b_{1v} = b_{2v} = 5$ N m s, $b_{1c} = b_{2c} = 0$ N m s, $g = 9.81$ m/s². The dynamic equations of the robot arm (eqs. 7.48 and 7.49) were simulated using the classical fourth-order Runge-Kutta algorithm [PFTV88] with an integration step size $h = \Delta T/2$, where $\Delta T =$ sampling period $= 0.01$ s (100 Hz). The delays $M$ (see fig. 7.13) for each component of the RT were set to 1.
In relation to the ANN, the parameter $L$ was set for each component of the RT to 19. Therefore the ANN had 40 input units. Two hidden layers of hyperbolic tangent ($tanh$) units were used, where the hidden layer closer to the input side had 30 units and the other hidden layer had 10 units. Since there are two joints, the ANN had 2 output linear units. The ANN was a strictly feedforward network, i.e. each layer sent connections only to the next consecutive layer. All network weights and biases, except the weights and biases of the output units, were initialized as random numbers with gaussian distribution of zero mean and 1/2 as the standard deviation. The weights and biases of the two output units were initialized as zero, such that before training the ANN output is zero for any RT. The ANN was trained using the BP algorithm.

All RTs have a duration of 4 s. The initial point of a RT is the same as its final point since the basic idea was to perform the training and testing sessions by following using periodic movements. All RTs, except RT 2, were generated as:

$$q^d(t) = \frac{\pi}{4} a + b\cos\left(\frac{2\pi t}{c}\right)$$

(7.50)

The RT 2 was generated as:

$$q^d(t) = \frac{\pi}{2} a + \frac{b}{1 + \exp[-5(t-c)]}$$

(7.51)

The parameters $a$, $b$ and $c$ and the minimum and maximum values (in degrees) for each of the RTs are respectively:

<table>
<thead>
<tr>
<th>Joint 1</th>
<th>Joint 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>RT 1: 2, −2, 4 (0°, 180°)</td>
<td>1, −1, 2 (0°, 90°)</td>
</tr>
<tr>
<td>RT 2: 1, −1, 4 (0°, 90°)</td>
<td>0, 1, 1 (0°, 90°)</td>
</tr>
<tr>
<td>0, 1, 3 (0°, 90°)</td>
<td>1, −1, 3 (0°, 90°)</td>
</tr>
<tr>
<td>RT 3: 1, −1, 4 (0°, 90°)</td>
<td>1, −1, 4 (0°, 90°)</td>
</tr>
<tr>
<td>RT 4: 2, −2, 4 (0°, 180°)</td>
<td>0, −2, 2 (−90°, 90°)</td>
</tr>
<tr>
<td>RT 5: 3, −1, 4 (90°, 180°)</td>
<td>−1, 1, 4 (−90°, 0°)</td>
</tr>
<tr>
<td>RT 6: 2, −2, 4 (0°, 180°)</td>
<td>1, −1, 4 (0°, 90°)</td>
</tr>
<tr>
<td>RT 7: 3, −1, 4 (90°, 180°)</td>
<td>1, −1, 4 (0°, 90°)</td>
</tr>
<tr>
<td>RT 8: 3, −1, 4 (90°, 180°)</td>
<td>−1, 1, 2 (−90°, 0°)</td>
</tr>
<tr>
<td>RT 9: 2, −2, 4 (0°, 180°)</td>
<td>1, −1, 4/3 (0°, 90°)</td>
</tr>
</tbody>
</table>

Note that the RTs and the plant outputs (angles of each joint) are specified in radians. Figure 7.17 shows the desired joint positions of the robot arm for RT 1 at $t = 0, 1, 2,$
3 and 4 seconds. RT 2 and RT 3 are very similar with the only difference that the extreme points in RT 2 are joined by a sigmoidal function while in RT 3 they are joined by a sinusoidal function. Note that, except for RT 2, the parameter $c$ gives the period of the RT for the joint. Therefore the period of all 9 RTs for joint 1 is 4 s. For joint 2: RTs 2, 3, 5, 6 and 7 have period 4 s; RTs 1, 4 and 8 have period 2 s; and RT 9 has period 1.33 s.

The numbering for each RT was selected considering the RMS error for joint 1 when only the feedback controller (with the original fixed gains) was used to control the robot arm, such that RT 1 and RT 9 were respectively the RTs with the smallest and largest RMS errors. Considering the whole set of 9 RTs the maximum and minimum values for each joint was calculated (minimum: 0°, −90°; maximum: 180°, 90°) and such values were used to scale the ANN inputs to be between −1 and 1.

Each group of simulations consisted first of a training session and then a recall session. The training session consisted of using RT 1 for 40 runs, i.e. the total training time was only 160 s = 2.67 min. Only at the beginning of the training session the robot arm position was reset in order to coincide with the initial desired position for RT 1. In the recall session all ANN weights and biases are fixed and each one of the 9 RTs was tested, i.e the recall session consisted of 9 runs where each run used a different RT. Whenever a new RT is being tested in the recall session the arm position is reset to coincide with the initial desired position of the new RT.

During the training session the following parameters where used in the BP algorithm: a) the momentum was set to zero; and b) the learning rate for all network weights and biases was set to 1/500 for the first 20 runs and halved for the last 20 runs.

The output of the feedback controller was calculated as in eq. 7.37 with the difference that the variables $e, ref, y, u^{FB}, e_r, e_d$ are now vectors, $k_p, k_i$ and $k_d$ are now
matrices and:

\[ u_{FB}^k = k_p e_k - k_i \Delta T e^i + \frac{k_d}{\Delta T} (e_k - e^d) \]  

(7.52)

For the case where the ANN was trained with a fixed feedback controller the gains set set to: \( k_p = \text{diag}[2000 \ 500] \), \( k_i = \text{diag}[0 \ 0] \); \( k_d = \text{diag}[200 \ 100] \);

Since the duration of each RT was 4 s and the sampling period \( \Delta T \) was 0.01 s, each run had 400 time steps. In order to measure the performance of the neural controller the RMS values of some variables were calculated for each run by:

\[ \text{RMS}[\text{var}] = \sqrt{\frac{1}{NT} \sum_{k=1}^{NT} \text{var}_k^2} \]  

(7.53)

where \( NT \) = number of time steps in each run = 400, and \( \text{var} \) is replaced by each component of the vector output error \( e \), ANN output \( u^{NN} \), output of the feedback controller \( u^{FB} \), and plant input \( u \).

In the training session with the variable feedback the gains of the feedback controller \( (k_p, k_i, k_d) \) for the first run (run 1) were the original gains as in the case of fixed feedback. The following rules were used to determine the feedback gains for the subsequent runs \( (i > 1) \):

1) IF the performance of the last run (run \( i-1 \)) improved in relation to the run before that (run \( i-2 \)); AND IF the performance of the last run was better than the performance when only the original feedback controller was used: THEN decrease the feedback gains.

2) IF the performance of the last run improved in relation to the run before that; AND IF the performance of the last run was NOT better than the performance when only the original feedback controller was used: THEN do not change the feedback gains.

3) IF the performance of the last run deteriorated in relation to the run before that; THEN increase the feedback gains.

The measure of performance used for each run is the summation of the RMS error for both joints. Run 0 is the run where only the original fixed feedback controller is used. The ANN begins to be trained only in run 1. The feedback gains for run \( i \) were calculated by multiplying or dividing the feedback gains for run \( i-1 \) by 0.92, depending on whether they should be decreased or increased. Another alternative is to use different factors for increasing and decreasing the feedback gains. Figure 7.18 shows for each run
during the training session the value of the feedback gains in relation to the original feedback gains.

Considering only the case where the variable feedback controller was used during the training session, fig. 7.19 (a) and (b) show the history of the RMS values of the ANN output, the output of the feedback controller and the control signal for each joint. Figures 7.20 and 7.21 show RT 1 and RT 3 being recalled before and after training was performed only using RT 1.

Figure 7.22 shows the RMS errors for all RTs during the recall session (after training) for the cases where the ANN was trained using a fixed or variable feedback controller. For comparison figure 7.22 also shows the RMS errors for the case where only the original feedback controller is used. We can see that when trained with a fixed feedback controller the ANN cannot generalize well and it can even be considerably worse than using only the feedback controller (RTs 4, 5 and 8). We can also see that the ANN trained with the variable feedback controller generalizes considerably better than the one trained with a fixed feedback. Considering the set of all 9 RTs the ANN trained with the variable feedback had a performance much better or comparable (i.e. never much worse) to the case where only the feedback controller was used.

Note that, as expected, as the difference between a specific RT and the RT used during training increases, the performance of the ANNs deteriorate. The worst case happens for RT 9 that has a frequency greater than RT 1.
Figure 7.19 - The RMS values for the feedback and ANN controllers during training

Figure 7.20 - Recalling RT 1 before and after training with RT 1, using a variable feedback controller

Figure 7.21 - Recalling RT 3 before and after training with RT 1, using a variable feedback controller
Figure 7.22 - The RMS errors for all RTs during the recall session for the cases when the ANN is trained with a fixed or variable feedback controller

In practice, instead of just one RT, a set of several RTs and a longer session should be used to train the ANN. However, these simulations show that the feedback controller also has an important role to play in such a control structure for nonlinear identification/adaptive control.

7.4 - Fault-Tolerant Dynamical Control

One field in control theory that has not received much attention is the design of fault-tolerant controllers (also called reliable controllers), i.e. controllers that have a graceful degradation of performance in relation to a set of faults. Some possible faults are loss of actuators, loss of sensors or internal damage to the controller. However, in certain application areas such as nuclear reactors, aircrafts, space missions and chemical plants, fault tolerance (or reliability) can be a crucial requirement.

A different problem is to have a controller that is robust in relation to faults in the plant, in which case a conventional solution is: 1) to detect that a fault has happened and to determine in which part of the plant it has happened (fault detection and fault isolation); 2) switch to a previously designed controller for the specific fault; 3) if such a controller is not available (perhaps because the possibility of the specific fault was not predicted), then run a parameter identification algorithm to determine the model of the faulty plant and design a new controller for the faulty system.

The same approach could be used for faults in the controller or loss of sensors...
and actuators. One problem is to have to account for all possible faults, a cumbersome task for complex plants that normally require complex controllers with several sensors and actuators. The second problem is the delay in detecting and isolating the fault. Such a delay may be enough to make the plant unstable and cause a catastrophic failure.

The aim of a fault-tolerant controller is to have the plant, whenever a fault occurs (in the controller or in the plant itself), if not operating optimally, at least in a safe configuration until the fault is neutralised by repairing the plant or by reconfiguring the controller. Therefore an increase in the fault tolerance of the controller will increase the overall stability of the control system.

Popular control systems design, in general, do not take into consideration the possibility of faults in the controller and therefore, not surprisingly, are not fault tolerant (or reliable), as Rosenbrock and McMorran [RoMc71] pointed out. Viswanadham, Sarma and Singh [VSS87] proposed a method of designing controllers for linear plants, using a stable factorization approach, that are tolerant to loss of sensors and actuators.

Due to their distributed organization, ANN have the potential of being fault-tolerant in relation to internal damage to itself, such as the loss of internal units or weights. However, as chapter 5 shows, the training algorithm has to be able to exploit such potential. Moreover, since ANNs are adaptable (or learning) devices, they appear to be able to find alternative solutions when they suffer partial damage, i.e. they can at least partially reconfigure themselves. In this section we show how such a potential for fault tolerance to internal damage can be used to obtain fault-tolerant neural controllers.

In this section the BPS algorithm, developed in chapter 5, is used to improve the fault tolerance of the neural controller in relation to faults in the ANN [NaZa93]. The controller structure is the same used in the previous section, i.e. the modified feedback-error-learning neural control structure (see figure 7.13). The plant to be controlled is an inverted pendulum that should follow a varying reference signal. Note that this is different from aiming to maintain the pendulum in the upright position.

The dynamical equation for the inverted pendulum (see fig. 7.23) can be simply derived using the principle that the resultant torque acting on the body is equal to the time variation of the total angular momentum of the body. Assuming that the pendulum is a thin cylinder or rod of mass $m_1$ and length $a_1$ with mass $m_2$ concentrated at the tip of the pendulum, $g$ = gravity, $b_1$ = coefficient of viscous friction, $\theta$ = angular position and $\tau$ = external applied torque, we can write [Sch90]:

$$\dot{\theta} = \frac{\tau}{J}$$
where the term \( (m_1/3 + m_L) a_1^2 \) is the moment of inertia of the pendulum. In relation to the diagram illustrated in figure 7.13, we have: \( u = \) plant input = \( \tau \), \( x = [x_1, x_2]^T = [\theta, \dot{\theta}] \), \( y = x_1 = \theta \). If it is desired to force the angular position to have a large excursion, then control of such a plant is a difficult task because it is equivalent to having a linear plant where one of the parameters (the gravitational load in this case, the second term on the right side of eq. 7.54) varies according to the position of the pendulum.

The following parameters were used in our simulations: \( M = 1 \), \( L = 29 \), \( \Delta T = \) sampling period = 0.1 s, and plant parameters:
\[
[a_1, m_1, m_L, b_1, g] = [0.5 \text{ m}, 0.25 \text{ kg}, 0.25 \text{ kg}, 0.1 \text{ N m s}, 9.8 \text{ m s}^2].
\]
The plant was again simulated using the classical fourth-order Runge-Kutta algorithm with an integration step size \( h = \Delta T/2 \). The feedback controller was also a PID controller with parameters \([k_p, k_i, k_d] = [0.5, 0.5, 0.1]\) and implemented as in the previous section (see eq. 7.52).

The variable \( ISE \) (Integral of the Squared Error) is used as a performance index for the particular setup used during the simulation. Such a variable was initialized as zero and updated at each time step as: \( ISE := ISE + e_k^2 \). The reference signal \( ref \) is described by:
where \( k = 0, 1, ..., NT \). In other words, the desired angle trajectory is a sinusoidal signal varying between 0 and \(-90\) degrees with period 2 s and sampled with period \( \Delta T = 0.1 \) s. At the beginning of each simulation it is assumed that \( x_k = [0 \ 0]^T \) and \( ref_k = 0 \) for \( k \leq 0 \).

The ANN has 30 inputs \((L = 29)\), 1 hidden layer with 10 hyperbolic tangent units, 1 linear output unit and no direct connection from the input to the output layer. Before entering the ANN, the reference signal is scaled such that its magnitude lies between \(-1\) and 1. The incoming weights and biases for the hidden units were initialized as random values using a gaussian distribution with zero mean and standard deviation 0.5. Again the incoming weights and biases for the output units were initialized as zero since, before being trained, the ANN should produce a zero output for any input.

The learning rate and momentum rate used by the BP algorithm were set at the beginning of the simulation to 1/800 and zero respectively. For the second half of the training period, the learning rate was reduced to half of its initial value.

The network was trained for 400 s using the BP and BPS algorithms. Care was taken to ensure that the network initial weights were the same for both algorithms.

It was assumed that the set of possible faults was composed of the failure of each hidden unit (output of the faulty hidden unit is clamped to zero), all hidden units have the same probability of failure and the no-fault configuration is as probable as each fault configuration. Therefore there are 11 possible configurations, each of them with the same probability. When the BPS algorithm was used, one of the possible network configurations was randomly chosen for each time step \( k \), i.e. \( SWEPO = 1 \).

Figures 7.24 (a) and (b) illustrate the actual and desired trajectories for the joint angle \( (\theta_k \text{ and } ref_{k-1}) \) during the last 10 s of the training section respectively for the cases: a) training with the BP algorithm; and b) training with the BPS algorithm. Figure 7.24 (c) shows the same variables when only the feedback controller is used to control the plant for the same 400 s. Observe that the feedback controller is far from being optimally tuned for this particular plant. Despite this, the ANN still converges to a good solution for both training methods as figures 7.24 (a) and (b) show. This illustrates that, given a long enough period to train the ANN, the employed control structure is robust in relation to the quality of the feedback controller.
Figure 7.24 - Actual and desired joint angle trajectory at the end of the training session when: (a) using the BP algorithm; (b) using the BPS algorithm; (c) using only the feedback controller. Part (d) shows the RMS values, calculated at every 2 s, for the error, the controller outputs and the control signal for the previous cases (b) and (c).

Figure 7.24 (d) shows the root-mean-square (RMS) values for the control signals during the training session when the ANN is trained with the BPS algorithm and also the RMS value for the feedback controller alone. The RMS values were calculated for every 2 s interval, i.e. one period of the reference signal, using:

\[
[RMS \ var]_i = \sqrt{\frac{1}{20i-1} \sum_{k=20(i-1)}^{20i-1} var_k^2}
\]  

(7.56)

where \(i\) = interval number, \(1 \leq i \leq 200\), and \(var\) is replaced by \(e, u^{FB}, u^{NN}\) or \(u\). The case when the ANN is trained using the BP algorithm results in a graph similar to fig. 7.24 (d) where the ANN is trained with the BPS algorithm. Considering the whole training session the values for the variable ISE are: a) 43.07, when using the BP algorithm; b) 53.10, when using the BPS algorithm; and c) 332.7, when only the
feedback controller is used to control the plant.

After the training session all 11 possible configurations for the ANN were tested using the same reference signal for the same period of time (400 s) and the ANN weights and biases were kept fixed, i.e. learning was disabled. Before each test the system state $x$ was reinitialized to zero, i.e. the pendulum was placed in its initial horizontal position with a zero angular velocity. Since the calculation of the $ISE$ starts at the beginning of the simulation for the training and the testing phases, the $ISE$ also includes a transient error which is a consequence of changing the pendulum state from a rest position to a periodic movement.

Table 7.1 shows the values for the integral of the squared error $ISE$ for the test

<table>
<thead>
<tr>
<th>Hidden unit removed</th>
<th>BP</th>
<th>BPS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$ISE$</td>
<td>Ratio</td>
</tr>
<tr>
<td>None</td>
<td>22.34</td>
<td>1.00</td>
</tr>
<tr>
<td>1</td>
<td>31.95</td>
<td>1.43</td>
</tr>
<tr>
<td>2</td>
<td>58.53</td>
<td>2.62</td>
</tr>
<tr>
<td>3</td>
<td>26.33</td>
<td>1.18</td>
</tr>
<tr>
<td>4</td>
<td>60.53</td>
<td>2.71</td>
</tr>
<tr>
<td>5</td>
<td>29.47</td>
<td>1.32</td>
</tr>
<tr>
<td>6</td>
<td>53.19</td>
<td>2.38</td>
</tr>
<tr>
<td>7</td>
<td>118.40</td>
<td>5.30</td>
</tr>
<tr>
<td>8</td>
<td>25.37</td>
<td>1.14</td>
</tr>
<tr>
<td>9</td>
<td>22.70</td>
<td>1.02</td>
</tr>
<tr>
<td>10</td>
<td>64.75</td>
<td>2.90</td>
</tr>
<tr>
<td>max</td>
<td>118.40</td>
<td>5.30</td>
</tr>
<tr>
<td>mean</td>
<td>49.12</td>
<td>2.20</td>
</tr>
<tr>
<td>min</td>
<td>22.70</td>
<td>1.02</td>
</tr>
<tr>
<td>st. dev.</td>
<td>29.32</td>
<td>1.31</td>
</tr>
</tbody>
</table>

Table 7.4 - Integral of the squared error (ISE) when the ANN is tested for fault tolerance. The mean and standard deviation values include only the fault configurations.
runs of the no-fault and the 10 fault configurations for both training methods. The mean and standard deviation values were calculated considering only the ISE for the fault configurations. When the BPS algorithm was used, the ISE for the no-fault configurations increased from 22.34 to 32.93 (a variation of 47.4%), but for the fault configurations the mean ISE reduced from 49.12 to 44.98 (8.4%) and the standard deviation reduced from 29.32 to 21.69 (26%). Also the maximum possible ISE was reduced from 118.40 to 78.10 (a variation of 34%). Observe that, when the ANN was trained with the BPS algorithm, the ISE was in fact reduced with the loss of some of the hidden units.

Figure 7.25 shows the tracking error if the ANN is tested as before but with a fault occurring at 200 s, instead of at the beginning of the simulation. The no-fault configuration is used until 200 s and afterwards the ANN loses hidden unit 7. The tracking error is shown for the cases when the ANN was trained with the BP or with the BPS algorithm. From figure 7.25 we can see that, in comparison with the BP algorithm, by training the ANN with the BPS algorithm, the magnitude of the tracking error is increased before the fault and decreased after it.

7.5 - Conclusions

In the first part of this chapter the major neural control architectures were reviewed. The feedback-error-learning architecture was then analysed in more detailed.
We proposed the modified feedback-error-learning architecture and showed that it performs closed-loop linear/nonlinear identification of the inverse dynamical model of the plant. Although the inverse dynamical model is identified, the architecture can still be applied to linear non-minimum phase plants (that have an unstable inverse) since the ANN searches for a delayed inverse of the plant (which should be stable).

We show that for a single-input single-output linear time-invariant plant the minimization of the square of the output of the feedback controller results in a good approximation of the inverse delayed dynamical model of the plant. We also proposed the technique of using a variable (or adaptable) feedback controller to improve the generalization of the ANN. The results of simulations with a two-joint robot were then shown.

Finally we show that the BPS algorithm, present in chapter 5, can also be used to improve the fault tolerance of neural controllers in relation to internal damage to the ANN.
Chapter 8 - Conclusions and Directions for Further Work

In this thesis we have developed algorithms that employ artificial neural network models to solve the problems of: a) extremum control of static systems with an asymmetric performance index; b) adaptive control of nonlinear dynamical systems under feedback.

The IAC (Interactive Activation and Competition) feedback network was presented and analysed in detail. We have proved that the IAC network can also be used to solve quadratic optimization problems and, as such, is an alternative to the Hopfield network.

We have also proposed an algorithm that can be used to speed up the training of feedforward neural networks that use sigmoidal functions in the hidden layers. The basic idea is to constrain the location of the decision surfaces, which are defined by the weights arriving at the hidden units.

We have mathematically analysed the fault tolerance of feedforward neural networks and we showed that, by incorporating fault tolerance in a novel way, the problem of training the network is regularized. We have shown that in some cases the proposed cost function will have a unique minimum point. However, we have also shown that in general there is no unique solution for the set of network weights. The BPS algorithm was proposed and we showed that its application results in fault tolerant networks.

We have developed a novel non-standard neural network model and have used it to solve the extremum control problem of static systems with an asymmetric performance index. The standard Back-Propagation algorithm was modified and used to adapt the network free parameters. We have also shown, theoretically and by using simulations, that the same network model can also be used in the multi-input case.

A modified feedback-error-learning control structure was proposed and mathematically analysed. We have shown that the aim of this structure is to perform
closed-loop identification of the inverse dynamical system. The technique of using a variable (or adaptive) feedback controller was also proposed and we showed that it improves the generalization of the neural network controller. Finally, we have applied the BPS algorithm to improve the fault tolerance of the neural network controller.

The work presented in this thesis shows that the properties of nonlinear modelling, adaptability and fault tolerance exhibited by artificial neural network models can offer effective solutions to problems that may be very difficult or intractable by other approaches. On the other hand there is still the need for much more formal mathematical analysis in several areas of artificial neural networks. For instance, one of the main outstanding problems in using artificial neural networks is to decide how large the network needs to be in terms of the number of hidden units. Dynamic allocation of hidden units during training may offer a solution for static problems.

The work presented in this thesis can be further developed in several directions. In relation to chapter 3 possible areas of research are: a) to study the storage capacity of the IAC network; and b) to modify the IAC network model in order to eliminate local minima.

The algorithm proposed in chapter 4 to speed up training has the disadvantage that a permissible region for the decision surfaces has to be defined by the designer. Such a permissible region is then treated as a "hard" constraint. A possible modification that avoids the need for specifying a permissible region would be to treat the location of the decision surfaces as a "soft" constraint, so that the penalty for violating the constraints is finite.

An interesting investigation in relation to the results obtained in chapter 5 concerning fault tolerant networks would be to test the hypothesis that the cost function specified in eq. 5.41 has a unique solution and therefore solves the problem of parameter identifiability. Again, using inspiration from biology, it would interesting to investigate if the asynchronous operation of biological neural networks (there is no central clock to synchronize the neurons) has an important role in their fault tolerance to loss of neurons.

In relation to the neural network extremum controller developed in chapter 6 a possible area for further research is an investigation of the modelling capabilities of the proposed network model in the single and multi-input cases, i.e if it is possible to prove that, given enough hidden units, the proposed network model can approximate any single and multi-input unimodal asymmetric functions with an arbitrary small error. The use
of more sophisticated training methods and network model fault tolerance should also be investigated.

In chapter 7 the algorithm used to train the network to control the plant adjusts only the network weights. The network configuration (number of hidden layers and number of hidden units) has to be decided before training begins. It would be useful to investigate if the network configuration can also be adjusted at the same time that the network is being training to control the plant, using some of the techniques mentioned in chapter 2 (section 2.5.3), such as weight decay and weight pruning. Another possible research area is to investigate under which conditions the matrix $F'$, defined in eq. 7.26, is positive definite and to find a formal proof for the validity of the learning rule proposed in eq. 7.36.
References


[Ros58] - Rosenblatt, F. (1958). The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain, Psychological Review, 65, 386-408 (also as chapter 8 in [AnRo88])


[Son93] - Sontag, E. D. (1993, 6 July). Some Topics in Neural Networks and Control, Department of Mathematics, Report Number LS93-02, Rutgers University, New Brunswick, USA


