# Table of Contents

# Chapter XI- Training and Using Recurrent Networks

Version 2.0

This Chapter is Part of:

*Neural and Adaptive Systems: Fundamentals Through Simulation© by*

Jose C. Principe
Neil R. Euliano
W. Curt Lefebvre



The goal of this chapter is to introduce the following concepts:

- Why backpropagation can not train recurrent systems.
- Develop the backpropagation through time algorithm.
- Introduce and train distributed TLFNs.
- Provide the basic theory to study the gamma model
- Introduce and train fully recurrent networks.
- Explain fixed point learning.
- Explain Hopfield's computational energy.
- Present Grossberg additive neural model.
- Show applications of recurrent neural networks.
    - 1. Introduction
    - 2. Simple recurrent topologies
    - 3. Adapting the feedback parameter
    - 4. Unfolding recurrent networks in time.
    - 5. The distributed TLFN topology
    - 6. Dynamical Systems
    - 7. Recurrent systems
    - 8. Learning Rules for Recurrent Systems

- 9. Applications of recurrent networks to system identification and control
- 10. Hopfield networks
- 11. Grossberg's additive model
- 12. Beyond first order PEs: Freeman's model
- 13. Conclusions

Go to next section

# 1. Introduction

In the previous chapter, we were able to create TLFN networks that processed information over time and were easy to train. Basically, they could only implement static (but arbitrary) mappings from the present input and its memory traces to the desired response. There is often a need to extend the network capabilities to time dependent mappings. This means that short-term memory mechanisms have to be brought inside the feedforward network topologies (TLFNs), or the networks have to be made spatially recurrent , i.e. recurrent connections are created among some or all PEs. We will call these spatially recurrent networks simply recurrent networks.

The complexity of these two solutions is very different. The TLFNs have locally recurrent connections and can be made stable just by enforcing the stability of the short-term memory mechanisms, while it is much more difficult to guarantee stability of recurrent networks. Moreover, TLFNs are easier to train than recurrent systems, so they are more practical. Lastly, one can still interpret how a TLFN is processing the information by combining our knowledge of MLPs with adaptive filters, while the massive interconnectivity of the recurrent system defeats in most cases our ability to study the system.

You should have noticed how carefully we picked the focused TLFN topology to still be able to use the static backpropagation algorithm. But this is no longer possible for

distributed TLFNs nor for recurrent networks. One of the central issues that we have to address in this chapter is how to train recurrent networks. We will start by extending static backpropagation to adapt systems with delays, i.e. systems where the ordered list does not only depend upon the topology as in Chapter III, but also depends on a time order. This concept will give rise to the back propagation through time (BPTT) algorithm which trains recurrent networks with a segment of a time series. Learning a segment of a time series is called trajectory learning . This is the most general case for learning in time. BPTT will be applied to train the gamma network, distributed TLFNs, and fully recurrent networks. We will also study how recurrent systems are trained to memorize static patterns by extending static backpropagation to what has been called fixed point learning .

We will also superficially study dynamical systems in terms of main definitions and topologies. A paradigmatic example of the insight gained with dynamics is Hopfield 's interpretation of the **"**computational energy**"** of a recurrent neural system. We will cover this view and see how it can be used to interpret a recurrent system with attractors as a pattern associator. We will end the Chapter (and the book) with a description of the Freeman's model which is a new class of information processing system which is locally stable but globally chaotic.

Throughout the Chapter we will provide applications of time lagged feedforward networks ranging from nonlinear system identification, nonlinear prediction, temporal pattern recognition, sequence recognition and controls.

Go to next section

# 2. Simple recurrent topologies

All the focused TLFNs studied in Chapter X implement *static nonlinear mappings*. Although focused TLFNs have been shown universal mappers, there are cases where the desired function is beyond the power of a reasonably sized focused TLFN. The easiest case to imagine is a string that gives rise to two different outputs depending upon

the context. Either we have enough memory to span the full context, or the network will be unable to discover the mapping. Jordan and Elman proposed simple networks based on context PEs and network recurrency that are still easy to train (because the feedback parameters are fixed) and accomplish the mapping goal with small topologies (Figure 1).

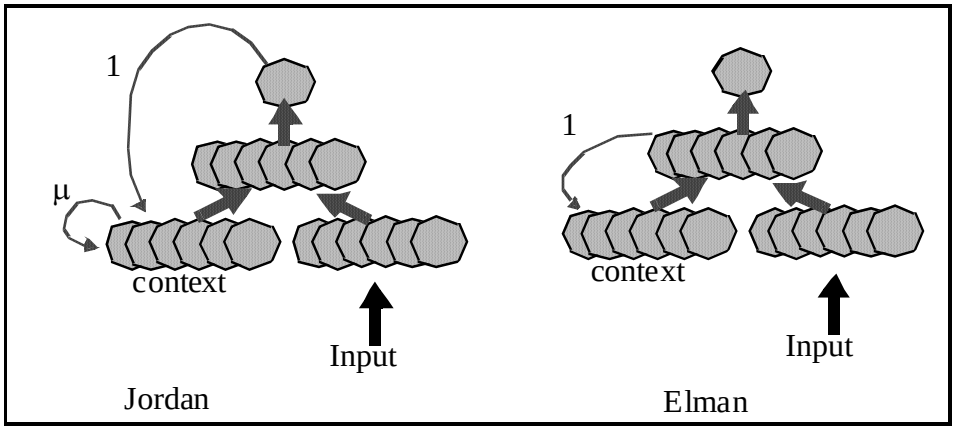



**Figure 1. Jordan (left) and Elman networks.**

Note that both the Jordan and Elman nets have *fixed feedback parameters and there is no recurrency in the input-output path*. They can be approximately trained with straight backpropagation. Elman's context layer is formed from nonlinear PEs and receives input from the hidden layer, while Jordan's context layer receives input from the output, and the context layer is built from context PEs.

These systems are in principle more efficient than the focused architectures for encoding temporal information since the "memory" is created by recurrent connections that span several layers, i.e. memory is inside the network, *so the input-output nonlinear mapping is no longer static*. However, the Jordan and Elman networks are still very special architectures that were derived with the goal of easy training. Notice that the outputs of the context layer can be thought of as external inputs (which are controlled by the network instead of by world events) such that there is no recurrency in the input-output path. Both systems have been utilized for sequence recognition and are sometimes called *sequential networks*. Jordan networks can even associate the same (fixed) input with several output sequences depending upon the context.

## NeuroSolutions 1
## 11.1 Jordan's network

**We are going to create a data set with time dependencies as first proposed by Elman. Suppose that we have 9 items coded as amplitude levels 0.1, 0.2, …. 0.9 which appear randomly in a sequence. However, when each item appears, we know that it will appear for a predetermined number of time steps. For instance, 0.1 appears for 1 time step, 0.2 for 2 time steps, and 0.9 for 9 time steps. Elman associated the random values with consonants that are followed by a pre-determined number of vowel sounds.**

**Can a network learn the time series? If you think a bit the answer must be no, because the items appear randomly. So the error must be large. However, the error should not be uniformly high over time, since there is a predictable part in the time series structure, namely, the length associated with each level. So a network that can capture the time structure should yield a low error once a given level appears at the network input.**

**Our first architecture is the Jordan network which feeds back the output of the system to a layer of context PEs. The network has a single input, and a single output. The desired response is the same as the input but advanced of one time step (i.e. the network is trained as a predictor). We will beusing backpropagation to train the system, although this is an aproximation as we will see shortly. We start with 5 hidden PEs and 5 context PEs with fixed feedback. But you can experiment with these parameters. Running the network we can observe the expected behavior in the error. The error tends to be high at the transitions between levels.**

**Since we worked in Chapter X with a time series built from two sinewaves of different frequencies (which is also a problem of time structure) let us also see if the Jordan network can learn it. The answer is negative, since the feedback is from the output, so if the output is always wrong, the feedback does not provide valuable information.**

**NeuroSolutions Example**

**NeuroSolutions 2**

**11.2 Elman's network**

**We will repeat here the previous problem but now with an architecture that feeds back the state to the context PEs, i.e. the hidden layer activations are providing the input to the context PEs.**

**The Elman network works as well as the Jordan network for the multi-level data set, and is able to solve the two sinewave problem of Chapter X. Working with the past system state seems more appealing than working with the past output. Notice also that we are using static backpropagation to train the weights of the system.**

**NeuroSolutions Example**

We saw that one of the difficult problems in the processing of time signals is to decide the length of the time window. Normally we do not know which is the length of the time neighborhood where the information relevant to process the current signal sample resides. If the window is too short then only part of the information is available, and the learning system is only working with partial information. If we increase the window too much, we may bring in information that is not relevant (i.e. the signal properties may change over time) which will negatively impact learning. We saw in Chapter X that the value of the feedback parameter controls the memory depth in the context PE, so in principle its adaptation from the data may solve our problem.

In this chapter we are going to lift the restriction of working with constant feedback coefficients and special architectures, so we have to first understand the problem created by feedback when training neural networks.

Go to next section

# 3. Adapting the feedback parameter

Let us consider the simple context PE. A very appealing idea for time processing is to *let*

*the system find the memory depth that it needs in order to represent the past of the input signal.* If we utilize the information of the output error to adapt the feedback parameter 1-$\mu$ (which we will call $\mu 1$), then the system will be working with the memory depth that provides the smallest MSE. This is in principle possible since the feedback parameter $\mu 1$ is related in a continuous way (a decaying exponential) to the PE output. For this we need to compute the sensitivity of the output to a change of the feedback parameter $\mu 1$.

Can static backpropagation be utilized to adapt $\mu 1$? *The answer is a resounding NO*. The reason can be found in the *time delay operator* ($z^{-1}$) and in the recurrent topology*. The time delay operator creates an intrinsic ordering in the computations* since the output at time n+1 becomes dependent upon the output value at time n (Figure 2). When we derived the backpropagation algorithm in Chapter III we mentioned that the algorithm would compute gradients on any ordered topology, i.e. topologies that obeyed a dependency list (see Eq. 31 of Chapter III). This dependency list was static, i.e. *only addressed the dependencies created by the network topology*. However, the *delay imposes also a time dependency on the variables*, so it will interfere with the dependency list created by the topology which does not consider time. Moreover, the recurrent connection makes a tremendous difference as we will see now. Let us compute the sensitivity of the output with respect to the weights $\mu 1$ and $\mu 2$ for the network depicted in Figure 2.

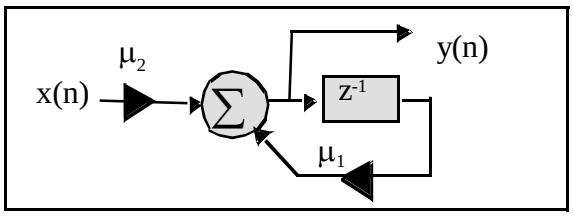



**Figure 2. A first order recurrent system**

The input-output relation is

$$y(n) = \mu_1 y(n-1) + \mu_2 x(n) \qquad \textbf{Equation 1}$$

The partial derivative of *y(n)* with respect to $\mu_2$ is simply

$$\frac{\partial}{\partial \mu_2} y(n) = x(n)$$

**Equation 2**

However, let us take the derivative of *y(n)* with respect to the feedback parameter $\mu_1$

$$\frac{\partial}{\partial \mu_1} y(n) = y(n-1)\frac{\partial}{\mu_1}\mu_1 + \mu_1 \frac{\partial}{\partial \mu_1}\big(y(n-1)\big)$$

**Equation 3**

The first term in Eq. 3 is equivalent to Eq. 2, but notice that *y(n*) also depends on *y(n-1)* because of the recursive nature of Eq. 1. This is a major difference between the static case and the recurrent system. Notice that it basically says that the effect of any change in the parameter $\mu$1 (the recurrent parameter) lasts forever!, while the effect of a change in the feedforward parameter $\mu$2 only matters in the current sample.

As you may recall the backpropagation algorithm covered in Chapter III did not include this effect since there was no feedback connections. The algorithm in Chapter III was called *static backpropagation* exactly for this reason. This also means that for recurrent topologies the equations need to be re-derived to cope with time dependencies.

## 3.1. Error criteria to train dynamic networks

The fundamental difference between the adaptation of the weights in static and recurrent networks is that in the latter, the *local gradients depend upon the time index*. Moreover, the type of optimization problems are also different because we are generally interested in quantifying the performance of adaptation within a time interval, instead of instantaneously as in the static case.

The most common error criterion for dynamic neural networks is *trajectory learning* where the cost is summed over time from an initial time *n=0* until the final time *n=T*, i.e.

$$J = \sum_{n=0}^{T} J_n = \sum_{n}\sum_{m} \varepsilon^2{}_m(n)$$

**Equation 4**

where $J_n$ is the *instantaneous error*, and *m* is the index over the output PEs (we omitted the summation on the patterns for simplicity). The time *T* is the length of the trajectory

and is related to the length of the time pattern or of the interval of interest. So, *the cost function is obtained over a time interval*, and the goal is to adapt the adaptive system weights to minimize $J_n$ over the time interval. Eq. 4 resembles the batch mode cost function if one relates the index *n* with the batch index. But here *n* is a time index, i.e. we are using different samples of the time series to compute the cost.

The static error criterion utilized for the MLPs, i.e.

$$J = \sum_m e^2{}_m$$

**Equation 5**

can still be utilized for dynamic systems and will be called fixed point learning. This cost is measuring the performance assuming that the output and desired signals do not vary over time (after the system relaxes to a steady state). Since in a dynamic network the states y(n) are normally time dependent, the sensitivity of the cost with respect to the states becomes also time dependent. So, the only way to implement fixed point learning is to let the system response stabilize (arrive at a steady state), and then apply the criterion.

Go to next section

# 4. Unfolding recurrent networks in time.

In order to apply the backpropagation procedure for recurrent networks, we need to *adapt the ordered list of dependencies for recurrent topologies*. As we saw in Eq. 1 the present value of the activation *y(n)* (also called the state) depends on the previous value *y(n-1)*. Likewise for the sensitivity of the state with respect to the feedback weight in Eq. 3 . There is a procedure called unfolding in time that produces a *time to space mapping*, i.e. it replaces a recurrent net by a much larger feedforward net with repeated coefficients. As long as the net is feedfoward we can apply the ordered list. As an example, let us take the case of the network of Figure 3, which shows a linear dynamic PE (PE#1) followed by a static nonlinear PE (PE#2). The goal is to adapt $\mu$ and $w_1$.

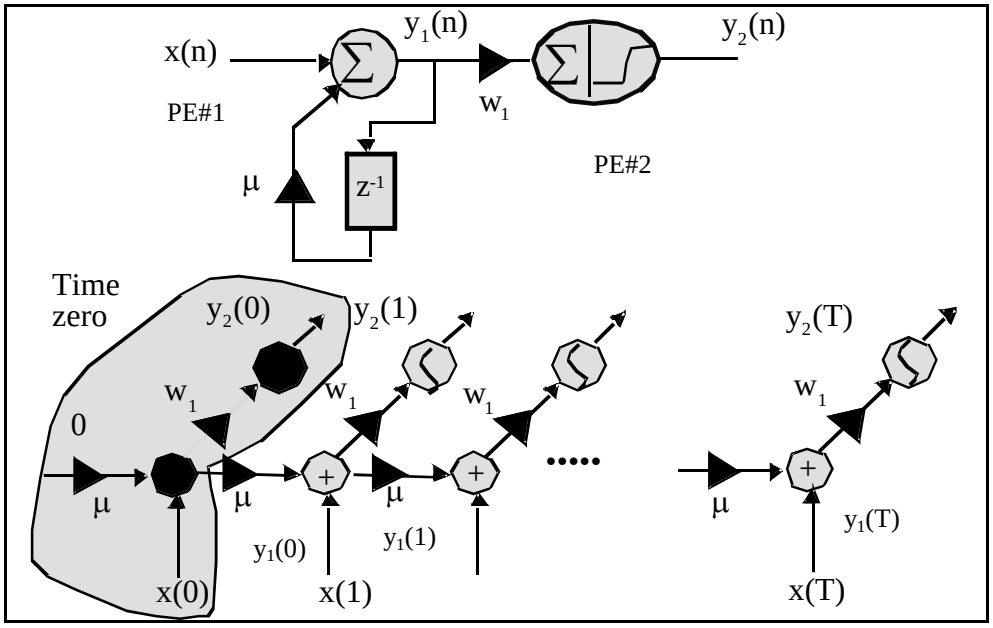



**Figure 3. Unfolding in time of a simple recurrent network**

The forward equation for this system is,

$$y_2(n) = w_1 f\left[\mu y_1(n-1) + x(n)\right] \qquad \textbf{Equation 6}$$

Each time tick *n* produces an output $y_2(n)$ that is a function of the previous value of $y_1(n-1)$ combined with the present input $x(n)$. Notice the dependence on the initial condition $y_1(n-1)$. Normally we set it to zero (y1(0)=0). Each time tick creates a stage of the unfolded network. Each additional time tick will be a similar network cascaded on the previous (Figure 3). Provided the recurrent network is in operation for a finite time, we can replace it with a static, feedforward net with T+2 inputs and T+1 outputs. *This operation is called unfolding the network in time*. From this unfolded network, we can generate *a dependency list* for the recurrent network as we did in Chapter III

$$L = \{\{\mu, w_1\}, y_0, \ldots, y_{2T+1}\} \qquad \textbf{Equation 7}$$

since the unfolded network has 2T+2 state variables. However, in this particular case we know that the state variables are only two, at different time steps. So we can rename the state variables invoking the implicit time order

$$L = \{\mu, w_1, y_1(0), y_2(0), y_1(1), y_2(1), \ldots, y_1(T), y_2(T)\}$$ **Equation 8**

Note that the weights appear first in the list since all the state variables depend upon them. Next the states are ordered by their index on the topology (as we did for the MLP), but repeating the same state variables indexed by time, from time zero to T. For the unfolded network of Figure 3 the even states correspond to the values that $y_1(n)$ takes over time.

A procedure to establish the dependency list is to first label, from left to right, all the state variables (output of adders in the topology) along with the output to create the dependency list at n = 0. Feedback loops are simply not considered. Next, we replicate the variables in the same order for the following time ticks until n=T. Therefore, in simple cases just by looking at the network structure the dependency list can be inferred quite easily, i.e. we can skip the network unfolding, once we understand the procedure. Systematic procedures by numbering the PEs exist (as done in NeuroSolutions) and will always unfold the network. Now we just need to apply the backpropagation algorithm to the dependency list of Eq. 8.

## 4.1. Extending static backpropagation to time - BPTT

We can apply the order derivative procedure explained in Chapter III to the dependency list of Eq. 7 which yields Eq. 32 . Instead we can also apply the order derivative procedure to Eq. 8 which is equivalent because we just counted the states differently. In this case we will have a double sum going over the original network state variables (in our case $y_1$ and $y_2$), and also the variables created by the time unfolding (indexed by time n=0, …,T), i.e.

$$\frac{\partial J}{\partial y_i(n)} = \frac{\partial^d J}{\partial y_i(n)} + \sum_{\tau>n}\sum_{j>i} \frac{\partial J}{\partial y_j(\tau)} \frac{\partial^d}{\partial y_i(n)} y_j(\tau)$$ **Equation 9**

Note that time $\tau > n$ to keep with the rule that we can only compute dependencies of variables that are to the right of the present variable in the list. *With this new labeling of variables time as well as topology information appears in the order derivative equation.*

We could still use the numbering of states in the unfolded network, but we would loose the power of interpreting the solution.

Likewise the sensitivity with respect to all the weights ($\mu$ and w are treated equally) can be written

$$\frac{\partial J}{\partial w_{ij}} = \sum_n \sum_k \frac{\partial J}{\partial y_k(n)} \frac{\partial^d}{\partial w_{ij}} y_k(n)$$

**Equation 10**

where we use the fact that the cost is computed over time as in Eq. 4 .

The first important observation is that the time index $\tau$ in Eq. 9 is greater than the present time *n*, so the *gradient computation in recurrent networks using the backpropagation procedure is anticipatory*. We saw that there are no physically realizable systems that are anticipatory (i.e. that respond before the input is applied). However, in digital implementations we can implement anticipatory systems during finite periods by using memory. We simply wait until the last sample of the interval of interest, and *then clock the samples backwards in time starting from the final time n*=T.

The second observation relates to the number of terms necessary to compute gradients over time. It is known that a recurrent system propagates to all future time any change of one of its state variable done at the present time (due to the recurrency). This indicates that in principle the sensitivity of one of the state variables at the present time needs also all its future time sensitivities to be computed accurately. However, notice that in Eq. 9 the summation term *blends only the explicit or direct dependence of $y_j(\tau)$ on $y_i(n)$ in time and on the topology.* So we are computing indirect time dependencies by explicit dependencies on the topology. For example, if we examine closely the feedforward equation Eq. 6 (or Figure 3) the present value of $y_1(n)$ only depends explicitly on the previous value $y_1(n-1)$. However the topology produces a link to $y_2(n)$. So in the ordered list (Eq. 8 ) *only $y_1(n+1)$ and $y_2(n)$ will depend explicitly on $y_1(n)$.* This shows the advantage of the ordered list in computing sensitivity information.

With this prologue, we are ready to apply Eq. 9 to the two state variables $y_2(n)$ and $y_1(n)$ and to the weights $\mu$ and $w_1$. For $y_2(n)$ we see that all the terms in the sums are zero since PE #2 is a non-recurrent output PE. So

$$\frac{\partial J}{\partial y_2(n)} = -\varepsilon(n)$$

**Equation 11**

where $\varepsilon(n)$ is the injected error at time $n$. For $y_1(n)$ we get

$$\frac{\partial J}{\partial y_1(n)} = 0 + \mu \frac{\partial J}{\partial y_1(n+1)} + w_1 f'\left(net_2(n)\right)\frac{\partial J}{\partial y_2(n)}$$

**Equation 12**

Let us analyze Eq. 12. We have no injected error since the PE #1 is internal. The double sum of Eq. 9 has two terms different from zero, since $y_1(n)$ has two direct dependencies, one from the time variable $y_1(n+1)$ and the other from the topology (PE #2), i.e. $y_2(n)$. The direct dependence over time gives the first nonzero term, while the dependence over the topology gives the second term.

Now for the weight gradients we get immediately,

$$\frac{\partial J}{\partial w_1} = \sum_n \frac{\partial J}{\partial y_2(n)} f'\left(net_2(n)\right) y_1(n) = \sum_n \delta_2(n) y_1(n)$$

**Equation 13**

and for the feedback parameter

$$\frac{\partial J}{\partial \mu} = \sum_n \frac{\partial J}{\partial y_1(n)} y_1(n-1) = \sum_n \delta_1(n) y_1(n-1)$$

**Equation 14**

There are many implications of this derivation. Probably the most important is that by unfolding the network in time we are still able to apply the backpropagation procedure to train the recurrent network. *This new form of backpropagation is called backpropagation through time (BPTT*) simply because now the ordered list is not only reversed in the topology (from the output to the input), but also in time (from the terminal time n=T to n=0).

The second important observation is that *BPTT is NOT local in time*. This may come as a surprise, but the beautiful locality of the backpropagation algorithm on the topology does not extend to time. We can see this easily by noting that both the state gradient (Eq. 12 )

and the weight gradient (Eq.14 ) are a function of more than one time index.

The third implication is the interpretation of the sensitivity equations as activities flowing in the dual network. Let us see what is the form of the dual PE for the linear context PE of Eq. 1. We re-write Eq.12 with our previous convention of error signals, *e(n)* and $\delta(n)$ (since the PE is an internal PE in the topology, the error that reaches it is internal),

$$e(n) = \mu_1 e(n+1) + \delta(n) \qquad \textbf{Equation 15}$$

If we interpret the error that is backpropagated from the output as the input to the dual network, the *dual of the linear recurrent PE* is as Figure 4.

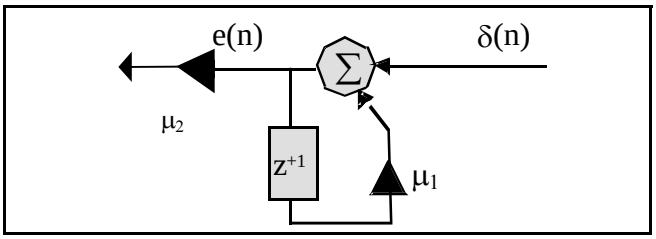



**Figure 4. Dual of the linear recurrent PE**

Notice that the dual network is constructed from the original network with the same rules presented in Chapter III, with the addition that the *delay has been substituted by an advance of one sample, i.e. the dual of the delay is the advance in time*. If we start by running the dual system from *n=T* this causes no problem. In fact running the dual system backwards reverses again the sign of the delay operator, so *the transfer function of the original and dual systems become the same*.

In Chapter III we showed how to implement static backpropagation as a data flow machine, i.e. without deriving explicitly the equations for adaptation but simply constructing the dual network and specifying the forward and the backward maps of each component. *Now we can extend the data flow procedure to dynamic neural networks since we found out the dual of the delay operator, the only operator we have not encountered before*. For instance, the equations to include the linear context PE in our data flow machine are therefore

$$\begin{aligned} forward \quad map \qquad & y(n) = \mu y(n-1) + x(n) \\ backward \quad map \qquad & e(n) = \mu e(n+1) + \varepsilon \ (n) \end{aligned}$$

**Equation 16**

and the weight update is given by Eq. 14 .

The data flow algorithm of static backpropagation must be adapted to cope with the time varying gradients. But the modifications are minor.

- First, the data for the full trajectory (T samples) is sent through the network (one sample at a time) and the activations stored locally at each PE.
- Then the set of outputs are compared to the desired trajectory using our criterion of choice to built the error sequence.
- This error sequence is reversed in time according to the BPTT procedure.
- The re-ordered error sequence is sent through the dual network and the local errors stored locally. NeuroSolutions implements this procedure using the dynamic controller.
- Once the sequence of local errors and local activations exist at each PE the weight update can take place using any of the first order search methods described in Chapter IV.

  Due to the fact that BPTT is not local in time, *we have to compute the gradients of all the weights and states in a time interval, even if only part of the system* is recurrent as in the TLFN topologies found in Chapter X. We have to pre-specify what is the initial time and the final time. This impacts the storage and computational requirements of the algorithm.

  Even for the case of the focused TLFN topology that has a single recurrent connection at the input, if BPTT is selected to adapt $\mu$, ALL gradients need to be computed over time.

  backpropagation versus BPTT

**NeuroSolutions 3**

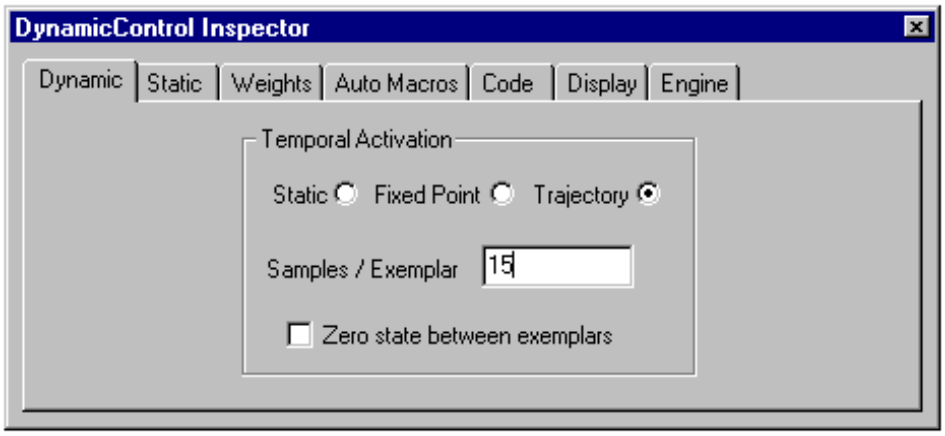
**11.3 Train the feedback parameter of Elman's network**

**This is going to be our first example using BPTT. The first thing is to notice that the controller is different, now it has 3 dials.**

**Its functionality is also different since now it has to orchestrate the firing of the data through a trajectory and command the PEs (and the duals) to store their activities (and errors). The errors have also to be reversed in time (starting from**

**the error at the last time step towards the beginning).**

**The user has one extra level of variables for BPTT which is related to the size of the trajectory and the initial conditions. Open the dynamic controller inspector and observe the dynamic level of the inspector. Notice that you have 3 choices: static (the same as the static controller), fixed point and trajectory. We will be interested in trajectory for this example. The size of the trajectory is called the number of exemplars. So when we set the number of samples per exemplars we are specifying the length of the trajectory. In the same panel we can further specify the initial conditions. If we click the box "zero states between exemplars", this means that for each new trajectory the state of the network (i.e. the values of activations and errors) will be reset . Normally we would NOT like to do that since it creates discontinuities between adjacent trajectories. Once we specify the variables in this panel the simulations work in exactly the same way as before.**




**We are ready now to use BPTT to train the feedback parameters of the Elman network for the example 2. We choose the size of trajectory as 15 samples. This means the controller will fire 15 samples at a time from the input file, each PE will store 15 activations, the network output will be compared to 15 samples of the desired response, and a 15 long error vector will be created. This error vector will be reversed in time, and fired through the dual network, one at a time. At the end of**

**the trajectory each PE and its dual will have a 15 sample long activation and error vector, and momentum learning can then be used to update the weights. Notice that the weights are updated with the cumulative error along the trajectory. The process is then repeated for the next 15 samples of the input file, etc.**

**Running the network we see that the it learns very fast. The initial values of the feedback parameters are set to 1 (no memory), and they slowly change to the appropriate value which minimizes the output MSE. This is the beauty of the context PE. It configures its memory automatically to minimize the output MSE. Notice that the time constants spread out when learning progresses to cover well all the time scales. The error is normally smaller than before.**

**NeuroSolutions Example**

## 4.2. Computing directly the gradient

As we saw in Chapter III for the MLP the gradients can be alternatively computed by applying the chain rule directly to the cost (Eq. 4 ) to obtain

$$\frac{\partial J}{\partial w_{ij}} = -\sum_n \sum_m \varepsilon_m(n) \frac{\partial}{\partial w_{ij}} y_m(n) \qquad i, j = 1, \ldots, N$$

**Equation 17**

We denote the gradient variables (sensitivity of the states) with respect to the weights as

$$\alpha^m{}_{ij}(n) = \frac{\partial}{\partial w_{ij}} y_m(n)$$

**Equation 18**

In order to compute the state sensitivities we use directly the forward equation for the neural network to yield

$$\alpha^m{}_{ij}(n) = f'\left(net_m(n)\right)\left[\delta_{im} y_j(n) + \sum_l w_{ml} \alpha^i{}_{ij}(n)\right]$$

**Equation 19**

where ***f'*** is the derivative of the nonlinearity, and $\delta_{i,m}$ is the Kronecker delta function that is 1 only when m=i. It is important to compare this equation with the one given in Chapter III

(Eq. 35 ) to see the differences brought in by the recurrency (the sum in the parenthesis).

Now if the *weights change slowly* compared with the forward dynamics we can compute these equations for every time sample instead of waiting for the end of the trajectory, i.e.

$$\frac{\partial J}{\partial w_{ij}} = -\sum_m \varepsilon_m(n) \frac{\partial}{\partial w_{ij}} y_m(n)$$

**Equation 20**

This is the reason why this procedure is called real time recurrent learning (RTRL).

It is interesting that the *RTRL algorithm is local in time*, *but not local in space so in some sense it is the dual of BPTT*. In fact, Eq. 17 shows that we are computing all the gradients with respect to every weight $w_{ij}$, so the computation is not local on the topology (in space), but it also clearly shows that the method is local in time. The time locality makes it appealing for on-line applications and VLSI implementations. The direct method was computationally unattractive for static networks when compared with backpropagation. It is even more so for dynamic networks and small trajectories as we will discuss later.

Let us apply RTRL to the topology of Figure 3. Since there are only two weights, it is easy to see that the procedure gives directly

$$\frac{\partial J}{\partial \mu} = \frac{\partial J}{\partial y_2(n)} f'\left(net_2(n)\right) w_1 \left[ y_1(n-1) + \mu \frac{\partial}{\partial \mu} y_1(n-1) \right]$$

**Equation 21**

and

$$\frac{\partial J}{\partial w_1} = \frac{\partial J}{\partial y_2(n)} f'\left(net_2(n)\right) y_1(n)$$

**Equation 22**

Notice that the formulas to compute $\mu$ and w with RTRL are different from the ones obtained for BPTT ( Eq. 13 and Eq. 14 ). However, one can show that for the same initial conditions at the end of the interval the two sets of equations provide the same updates, so they are equivalent. For small networks RTRL provides very compact equations, which may be beneficial to understand learning.

For instance, it is clear from Equations 21 and 22 that the gradients are a function of f'(.)

so in order to ensure stability the largest slope of the recurrent PE nonlinearity should be less than one, otherwise we may be amplifiying the errors. So if we use an $\alpha$ of 1 in the nonlinearity as is normally the case, we are in the marginally stable case. This is often forgotten, and will produce instability in training over time. Training the gamma filter

**NeuroSolutions 4**
**11.4 The effect of slope of nonlinearity in recurrent networks**

**Let us go back to the Elman network and modify the slope of the hidden PEs to a large value. The training will become very difficult producing most of the time unusable results. Try a large value and decrease the stepsize to see if you can control the learning. You will see it is very difficult.**

**Another important thing is to notice that the system output seems quite normal (just flat), but the internal PEs are all pegged, i.e. they are simply unusable for processing information. You may think that the system is not learning fast enough and try to bump up the learning rates which is the wrong move. Here the system is unstable, but it behaves differently than an unstable linear system (output will go to infinity) due to the nonlinearity. Recurrent systems behave in a very different way than static systems, so you have to be much more careful than before in the training.**

**We recommend that the slope of the nonlinearity be set at .8. We also recommend that you place a scope over the hidden PEs to see if they are pegged all the time. If they do then the PE is simply not responding and effectively the system has fewer degrees of freedom.**

## NeuroSolutions Example

# 4.3. Comparing RTRL and BPTT

Computational Complexity of RTRL:

RTRL computes the sensitivity of every system state with respect to all the weights. However, for trajectory learning it has the important characteristic of being local in time.

When the weights change slowly with respect to the forward dynamics, the quantities in Eq. 17 can be computed on-line, i.e. for every new time sample. This means that the state gradients $\frac{\partial}{\partial w_{ij}} y_m(n)$ are computed forward in time with every sample. If the desired signal is available for every sample of the trajectory, the errors $\varepsilon_m(n)$ are obtained and the weight gradients $\frac{\partial}{\partial w_{ij}}$ can also be computed for every new sample. Otherwise, the state gradients have to be stored for the length of the trajectory, and then at final time the weight gradients computed.

With this explanation we can compute the number of required multiplications (which take longer to execute in the computer processing unit) and storage requirements of the algorithm. For a N PE system there are $N^2$ weights, $N^3$ gradients and O( $N^4$ ) multiplications required for the gradient computation per sample. If the length of the trajectory is T this means O( $N^4 T$ ) overall computations. However, the number of items that need to be stored (assuming that the desired signal is available) is independent of time and is O( $N^3$ ), as many as the instantaneous state gradients.

The disproportionately large number of computations means that the method is realistic only for small networks. An interesting aspect is that RTRL can be applied in conjunction with static backpropagation, in networks where the feedforward path is static, but the input is recurrent (as in the focused architectures). The only component that requires redesign is the back-component of the recursive memory. When implemented, this combination is the most efficient way to train focused TLFNs since all the computations are local in time. training focused TLFNs

NeuroSolutions does not implement RTRL so, in order to train fully recurrent systems BPTT must be utilized.

## Computational Complexity of BPTT.

The basic observation to apply the backpropagation formalism in the training of recurrent networks is to modify the activations of a network evolving through time into activations of an equivalent static network where each time step corresponds to a new layer (unfolding the network in time). Static backpropagation can then be applied to this unfolded network as we have seen (Eq. 9 ). These equations can be put in a one to one correspondence with the data flow algorithm. As in the static case, there is also an intrinsic order in the computations:

1- An input is presented, and an output computed. The process is repeated for the trajectory length T, and the local activations stored locally.

2- Once the network output is known for the full trajectory, a sample by sample difference with the desired signal is computed for the full trajectory such that the error signal $\varepsilon(n)$ is generated. This error must be backpropagated from the end of the trajectory (n=N) to its beginning (n=0) through the dual topology. Local errors are also stored.

3- Once the local errors and local activations for each time sample are available, the weight update for the trajectory can be computed using the specified search methodology.

4- The process is repeated for the next trajectory. Normally trajectories are stacked to better teach the time relationships.

As we can see the computational complexity of BPTT (number of operations and storage) is much larger than static backpropagation. The number of operations to compute one time step of BPTT in a N PE fully connected network is O( $N^2$ ), so for the trajectory of size T it becomes O( $N^2T$ ). *This is much better than for RTRL*. The problem is the storage requirements. As we can see the activations need to be saved forward which gives O(NT). This means that for long trajectories compared with the size of the net, the BPTT algorithm uses more storage than RTRL. Table 1 shows a comparison of the two algorithms

| | RTRL | BPTT |
|---|---|---|
| space complexity | O( $N^3$ ) | O(NT) |
| time complexity | O( $N^4T$ ) | O( $N^2T$ ) |
| space locality | no | yes |
| time locality | yes | no |

It is interesting that BPTT is local in space but not over time, while RTRL is local in time

but not local across the net. NeuroSolutions implements the BPTT to train recurrent neural networks for trajectory learning. The interesting aspect is that the data flow algorithm explained in Chapter III for static backpropagation can also implement BPTT and fixed point learning with appropriate control of the data flow. The local maps for the neural components are associated with the PE and are independent of the type of training strategy. This attests the importance of the data-flow implementation of backpropagation as mentioned as far back as Chapter III.

Go to next section

# 5. The distributed TLFN topology

The learning machines created as a feedforward connection of memory PEs and nonlinear sigmoidal PEs were called time lagged feedforward networks (TLFN). Figure 5 shows the general topology of TLFNs.

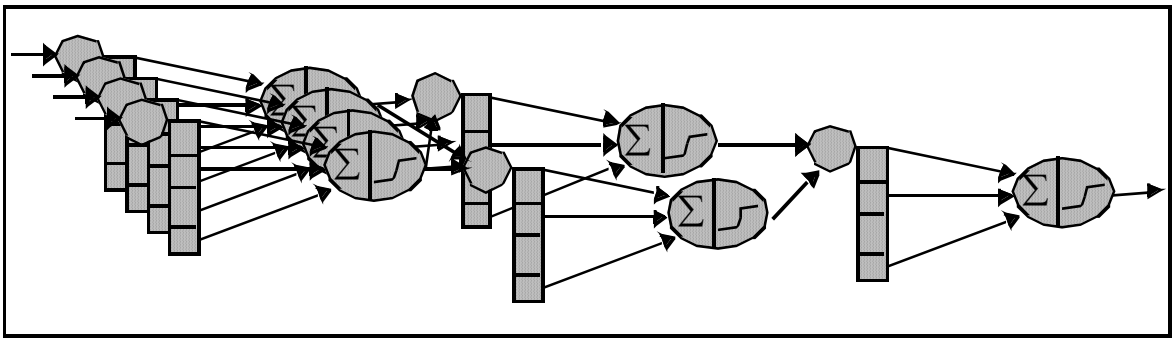


**Figure 5. A time lagged feedforward network**

The focused architectures can only produce a static nonlinear mapping of the projected representation. But there is no need to restrict ourselves to this simple case. As shown in Figure 5 one can also populate the hidden layers with memory structures and then produce time varying nonlinear maps from the projected space to the output. *Each memory PE in the TLFN is effectively processing information over time, by working with the projections of the PE activations of the previous layer on its local linear memory space*. The size of each memory space (i.e. the number of bases) is determined by the

number of memory taps.

TLFNs utilize *memory PEs that are ideal delays or locally recurrent*, such as the tap delay line PE, the gamma memory PE or the Laguerre memory PE. When the bases are created by locally IIR filters *the span of the memory space* (i.e. the amount of past information that is retained in the memory space bases) is not uniquely determined by K, but becomes also a function of the feedback parameters that the learning system can control through adaptation.

When the memory filters have feedback, TLFNs are recurrent networks. However, the global signal flow is feedforward, i.e. the networks are always stable provided the locally recurrent memory PEs are stable, and train better than fully recurrent networks. The leading characteristics of TLFN topologies is that when they are recurrent, the feedback is restricted to be local. Moreover, the recursive part is linear so we can utilize the well developed linear adaptive filter theory to partially study these networks. vector space interpretation of TLFNs

## NeuroSolutions 5
## 11.5 The issue of memory depth in the bus driver problem

**This will be a benchmark problem to find the memory depth of TLFNs. The problem is very easy to enunciate and focus on the memory depth information. It is called the bus driver problem, because it can be enunciated as the decision made by a bus driver. Suppose that a bus driver is going down a route line with many bus stops. He will stop at the next bus stop if the bell rings before the stop. However, he does not have control of how far in advance the passenger rings the bell.**

**A simple flip-flop in the bell line will solve this problem using an AND function (if there is a bus stop and the bell flip-flop is high, then stop the bus). However here we would like to use a TLFN so the issue is one of memory. The network has two inputs, one for the bus stops along the way and the other for the bell. The system will be trained with a signal that contains the information where the bus should stop.**

**From the point of view of complexity this is a trivial problem as long as the TLFN has sufficient memory (the AND can solve it). But learning the relation in time is not easy since the TLFN will be bombarded with time signals and it has to find the relation between the two inputs that makes the output follow the desired response. The input file is constructed from ones and zeros (1 means a possible stop or a ring) asynchronously spread in time. The system will be trained with static backpropagation to clearly show the effect of the memory depth.**

**The purpose of the example is to show that a TLFN with memory in the hidden layer can complement the memory at the input layer, saving eventually weights (the number of weights in the first layer tends to be always the largest).**

**We will use a gamma memory in the first layer and a tap delay line in the second layer. First we will set the hidden layer memory to 1 (just the current sample, i.e. no memory), and the number of taps in the gamma at 1. The largest time distance between a ring and a stop is 10 and occurs in the 2$^{nd}$ and the 5$^{th}$ stops. Run the network and see that it does not learn these two stops.**

**Now change $\mu$=0.5. As shown in the memory depth formula, this should provide enough depth to learn all the stops. This is indeed the case.**

**Now an alternative is to increase the memory in the hidden layer. Let us divide the number of taps between the input an the hidden layer (5 at the input and 7 at the hidden) and set the $\mu$=1 for the gamma. This solves the problem. We can even put 2 taps and 10 taps and still get a good solution. So this means that putting memory in the hidden layer "adds" to the memory in the input layer ( in fact in a nonlinear way due to the nonlinearity of the PE).**

## NeuroSolutions Example

## 5.1. How to train TLFNs

The supervised training of TLFNs will follow the gradient descent procedure on a cost

function defined by the output mean square error (MSE) over time (Eq. 4 ) . Since TLFNs are *systems with memory*, and normally the task of interest is *approximating time signals*, either real time recurrent learning RTRL or backpropagation though time BPTT must be utilized.

As we discussed extensively in Chapter III, in order to train neural networks using the data flow framework, two pieces of information are required:

- the implementation of the data flow algorithm
- the local maps for the PEs.

We saw how the dataflow algorithm is extended to BPTT and how it is implemented in the dynamic controller. With respect to the local maps, they depend exclusively on the form of the PE. The local maps for the sigmoid PEs, the softmax, the linear PE and the Synapse will be exactly the same as covered in Chapter I and III. The maps for the context PE are given in Eq. 16 . So we just need to address now the maps for the new memory PEs (gamma, Laguerre, gamma II). So BPTT is a general way to train TLFNs composed of arbitrary components.

## 5.2. Training gamma TLFNs with BPTT.

The data flow implementation is intrinsic to BPTT and it is implemented by the dynamic controller, but the local maps depend upon the PEs chosen. For the gamma PE these maps are

$$\begin{aligned} &\textit{activation} && y_k(n) = \left(1-\mu\right)y_k(n-1) + \mu y_{k-1}(n-1) \\ &\textit{error} && \varepsilon_k(n) = (1-\mu)\varepsilon_k(n+1) + \mu\varepsilon_{k+1}(n+1) + \delta(n) \end{aligned}$$

**Equation 23**

The second equation in Eq. 23 is just the application of Eq. 9 to the gamma memory. In fact, the gamma memory is a cascade of first order recursive elements where the stage is indexed by k. The extra term $\delta$(n) comes from the fact that the gamma memory dual also receives inputs from the topology. The forward map is utilized by the gamma memory component, the error map is utilized by the dual gamma memory PE. Note that in Eq. 23

the activation is equivalent to Eq. 46 , but the error in Eq. 23 is different from the formula obtained in Eq. 48 since they have been developed under two different methods (RTRL for Eq. 48 versus BPTT for Eq. 23). However, as was said previously, the gradient obtained over the same interval will be identical. The issue is that when using Eq. 23 even for the gamma filter, BPTT must be utilized. In order to update the feedback parameter using straight gradient descent, the weight update is

$$weight \quad update \qquad \frac{\partial J}{\partial \mu} = \sum_{n,k} \left[ y_{k-1}(n-1) - y_k(n-1) \right] \varepsilon_k(n-1)$$

**Equation 24**

These equations is all that is necessary to update the recursive coefficient in systems using the gamma memory, immaterial of the topology. training alternate memories

We will present below examples of two different topologies that use the gamma memory. The simplest is the gamma filter that is still a linear system that extends the linear combiner. The second topology is a TLFN with gamma memory.

## NeuroSolutions 6
## 11.6 Output MSE and the gamma memory depth

**This example illustrates the importance of the memory depth in the output MSE for system identification purposes. We will use the nonlinear system first presented in Chapter X. The topology will be a gamma filter with 5 taps. There are two parts to this example. First, we would like to show that the MSE changes with the value of μ. So we sill step by hand the values of μ from 1 to 0.1 and record the final MSE. As you will see the curve is bowl shaped, i.e. there is an intermediate value of μ that produces the smallest output MSE (in fact if we had enough resolution the curve is NOT convex, i.e. it has many minima). Notice that the Wiener filter will provide the MSE of μ=1, which is not the smallest for this example (and hardly ever is….).**

**The second step is to find out if we can train the μ parameter with BPTT to reach the minimum of the performance curve. A good rule of thumb is to use a stepsize**

**at least 10 times smaller for the feedback parameter when compared to the feedforward weights. Notice that in fact the system quickly found the best value of the μ parameter in this case.**

## NeuroSolutions Example

**NeuroSolutions 7**

**11.7 Frequency doubling with a focused TLFN**

**To get a better intuition on how different a recurrent memory is from a tap delay line, we present the following problem. We wish to construct a dynamic neural network that will double the frequency of an input sinusoid. Anyone familiar with digital signal processing knows that a nonlinear system is required for this task.**

**First let us build a one hidden layer focused gamma net with tanh nonlinearity with two PEs, a linear output node, and an input layer built from a gamma memory with 5 taps. The period of the input sinewave is set at 40 samples/period and the output at 20 samples/period. Backpropagation through time over 80 samples is utilized to adapt all the weights, including the recursive parameter m of the gamma memory.**

**Notice that the μ parameter starts at 1 (the default value that corresponds to the tap delay line). The value decreases to 0.6 in 150 iterations, yielding a memory depth of 8.3 samples . This means that with 5 taps the system is actually processing information corresponding to 8 samples which is beyond the 5 tap limit. This memory depth was found through adaptation, so it should be the best value to solve this problem.**

**Now let us reduce the size of the gamma memory from 5 taps to 3 taps, keeping the MLP architecture and the task the same. This time the μ would converge to 0.3, giving an equivalent depth of 10 samples (K/μ). This makes sense since the memory depth to solve the problem is determined by the input output map (frequency doubling), and it is the same in both cases. The second system had**

**less taps, so the parameter m that controls the memory depth adapted to a lower value. So the dynamic neural net was able to compensate for the fewer number of taps by decreasing the value of the recursive parameter, and achieving the same overall memory depth. The memory resolution in the latter case is worse than in the previous case. The tap delay line with 3 taps will never solve this problem.**

**It is very interesting to place a scope at the hidden layer to visualize how the network is able to solve the frequency doubling. The solution is very natural. The PE activations are 90 degrees apart. One is saturated to the most positive value, the other to the most negative value, yielding half sinusoids. Then the top layer can add them up easily.**

**With the trained system, let us fix the weights and see how it generalizes. Modify the input frequency and see what is the range for which the output is still a reasonable sinusoid of twice the frequency of the input.**

## NeuroSolutions Example

When the data flow implementation of BPTT is utilized to train TLFNs (see Chapter III), only the topology of the net and the local maps need to be specified, i.e. there is no need to rewrite the learning equations for each new topology. For TLFNs where the topology is normally highly complex, you will appreciate the advantage of not having to write learning equations...., and the flexibility that this brings, since any minor modification to the topology would require the rewriting of such equations. This data flow approach to train dynamic networks was implemented in NeuroSolutions since its inception (in 1992) although this fact was only published by Wan in 1996.

Go to next section

# 6. Dynamical Systems

TLFNs encapsulate the recurrencies into the processing elements, i.e. in the memory PE, and all the other connections where feedforward and instantaneous. There is no reason why a PE output can not be fed to some other PE that was already connected to the original one, producing a *feedback loop across the topology*, as we did in a restricted fashion in the Elman and Jordan networks. In the spatially recurrent systems, assuming an instantaneous map at the PE, and an instantaneous connection between the PEs (a product by a weight) will lead to infinite looping which is unrealistic and can not be modeled. *So delays have to be incorporated either at the PE or at the interconnection level to create time dependencies*. One needs a principled way to treat this type of systems, and the best strategy is to use again the ideas from dynamics and mathematical system theory, where these issues have been addressed a long time ago.

Here, the signals of interest change with time. We can think that there is a deterministic system that generates the time series data we are observing. Such a system is called a dynamical system, i.e. a system where its state changes with time (Figure 6).

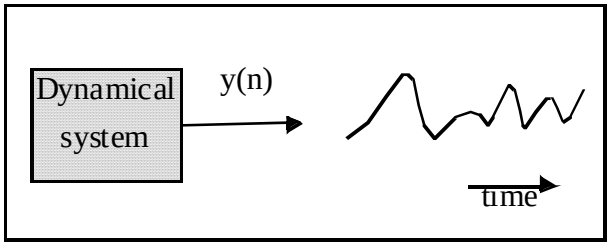



**Figure 6. A dynamical system producing a time series**

In Figure 6, *y(n)* is the system output and it is a time function. A dynamical system can be described by a relationship (a function) that links the next value *y(n+1*) to the present *y(n),* i.e.

$$y(n+1) = f\left(y(n)\right) \qquad \textbf{Equation 25}$$

This is one of the simplest dynamical systems we can think of, since it is governed by *first order dynamics*, i.e. the (next) value at time *n+1* only depends upon the (previous) value

at time *n*. As we saw in Chapter VIII, *Eq. 25 is called a first order difference equation*. If the function *f(.)* is a constant $\mu$, we obtain a first order *linear* dynamical system. We have already studied in Chapter X such a system in the form of the context PE. But *f(.)* may also be a nonlinear function, and in this case the system is a first order *nonlinear* dynamical system.

In neural networks we are very interested in first order *nonlinear* dynamical systems. In fact each PE of all the networks studied so far can be modeled in a very general way by a first order nonlinear first order difference equation, i.e.

$$y(n+1) = f\left(x(n) + wy(n)\right) \qquad \textbf{Equation 26}$$

where *x(n)* is the input to the PE, *f* is one of the nonlinearities studied, and *w* is a constant.

## 6.1. The State Space model

A productive (and principled) way to think jointly about static and dynamic PEs is to introduce the concept of *state space model*. Let us assume that there is an internal variable in the PE that describes its state at sample *n*. We will call this variable the *state variable net(n).* One can rewrite Eq.26 using the state variable as a set of equations

$$\begin{aligned} net(n+1) &= \gamma net(n) + x(n) \\ y(n+1) &= f\left(net(n+1)\right) \end{aligned} \qquad \textbf{Equation 27}$$

where $\gamma$ is a constant that may depend on the iteration. Having a set of two equations enhances our understanding. The *first equation is dynamic* and it shows the evolution of the state through time, and in neural networks it is a linear first order difference equation. *x(n)* is the combined input to the PE. The *second equation is static* (both *y(.)* and *net(.)* depend on the same instant of time), and it shows the nonlinear relation between the output of the PE and the state. A dynamic neural network will be a distributed interconnection of these PEs.

## 6.2 Static versus dynamic PEs

Now we can understand a little better the relation between the static PE covered in Chapters I to VII and the dynamic PEs we have discussed for temporal processing. Basically the static PE only represents the second equation in Eq. 27 . The function *f(.)* is the nonlinear map that we found in the McCulloch and Pitts PE or the sigmoid PEs. *For all practical purposes the static PE gets the input through the state*.

The dynamic PE is different. The next state is a function of the previous state (dynamics) and of the combined input to the PE. The state is then nonlinearly modified to provide an output. Note that the dynamic PE has dynamics of its own.

The memory PE that we studied conforms exactly with this principle, except that *f(.)* was the identity, i.e. we were using the state as the output. It is also instructive to compare Eq.27 with the equation of the nonlinear context PE, which is obtained by including the nonlinearity in Eq.3 of Chapter X, i.e.

$$y_j(n) = f\left(\sum_j w_{ij} x_i(n) + b_j + (1-\mu) y_j(n-1)\right) \qquad j \neq i$$

**Equation 28**

where *f* is one of the sigmoid nonlinearities found in Chapter III. The recurrent nonlinear PE looks like Figure 7. The quantity in parenthesis in Eq. 28 is the state equation that represents a dynamical system since it depends on two different time indices.

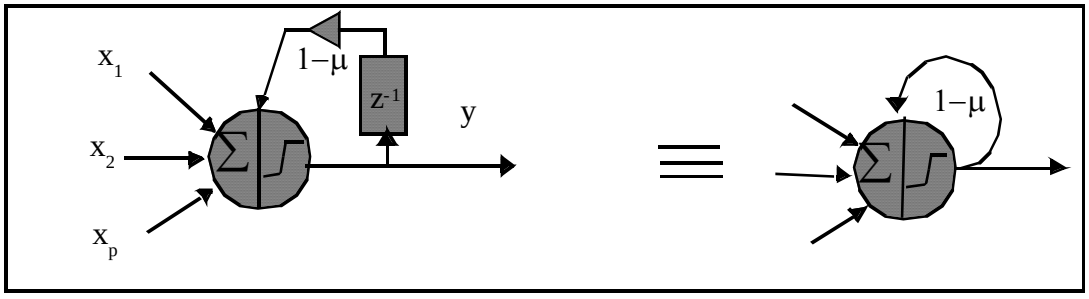



**Figure 7. Nonlinear context PE.**

Eq. 28 should be contrasted with the equation that defines the McCulloch-and-Pitts PE indexed by sample *n* that we copy below for convenience,

$$y_j(n) = f\left(\sum_i w_{ij} x_i(n) + b_j\right) \qquad j \neq i$$

**Equation 29**

Note that the M-P PE is static, i.e. the output depends only on its current input (or equivalently on the current outputs of the other PEs when connected in a network). While in Eq. 28 the output at time *n* depends on the previous output at time *n-1*. This *is the big difference between a static and a context PE*.

Go to next section

# 7. Recurrent Neural Networks

How can one construct recurrent neural networks? There are probably many different ways, but here we will describe one that allows us to construct directly recurrent neural networks from all the previous static components. Let us implement the static map of Eq.27 by the PE, and achieve total compatibility with all the PEs studied so far. Moreover, let us *assign the dynamics to the connectivity among PEs in the following way: if the connection is feedforward, the present value of the activation is used; but if the connection is a feedback connection let us include a delay of one time step in the activation.* This can be written as

$$net_i(n+1) = \sum_{j<i} w_{ij} y_j(n+1) + \sum_{j \geq i} w_{ij} y_j(n) + I_i(n+1)$$
$$y_i(n+1) = f\left(net_i(n+1)\right)$$

**Equation 30**

where I(n) is an external input eventually connected to the PE. Note that *we divided the computation of the state into two sums*: the first is the feedforward connections for which with our notation the first index of the weight is always larger than the second index, and the feedback connections where the first index is always smaller than the second. Such a system is called *the (fully) recurrent neural network* (Figure 8).

Once again we have to remember that *there are no instantaneous loops in this network*, i.e. all the feedback loops include a delay of one sample. Unfortunately this is never represented in the diagrams of recurrent networks. When a dynamical system is built, it is

a good practice to include a $z^{-1}$ symbol in the feedback connections and avoid any confusion.

The recurrent networks include the feedforward systems (MLPs and linear combiner) as well as the TLFNs as special cases. In fact, let us write the inter-connection matrix for a recurrent system with N PEs (Figure 8). In such cases the weight matrix is fully populated with non-zero values.

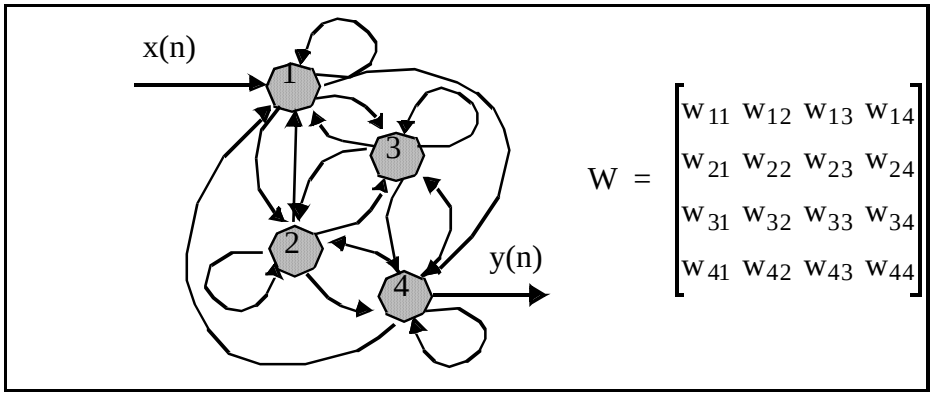



**Figure 8. Fully recurrent network**

A fully recurrent network with N PEs has $N^2$ weights. Note that the diagonal terms in **W** are the self recurrent loops to each PE. *The upper triangular weights represent feedback connections*, while the lower triangular weights represent feedforward connections.

In order for the system to become feedforward, the main diagonal weights are zero, as well as the upper triangular weights (with the numbering as in the Figure 8). The feedforward network is depicted in Figure 9. Note that the system became *static*.

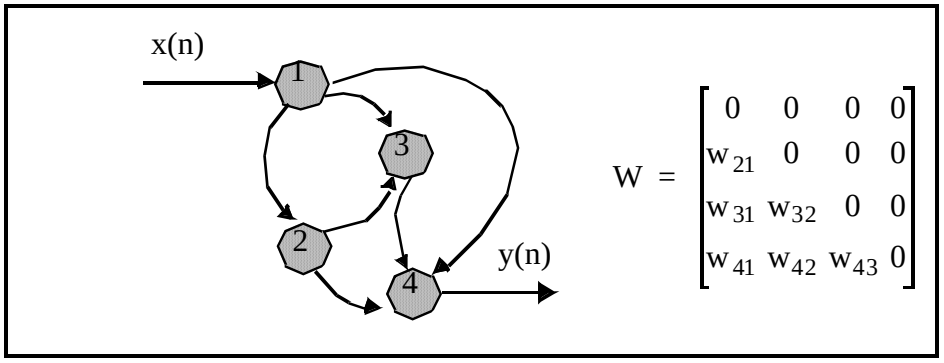



**Figure 9. The corresponding feedforward network.**

A layered system has even fewer connections different from zero (Figure 10).

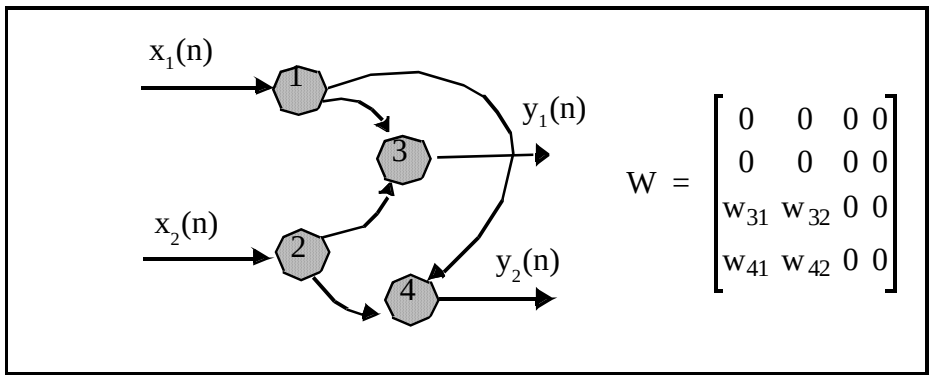



**Figure 10. The corresponding layered network (perceptron)**

In feedforward systems the first weight index is always larger than the second (with our notation). For the layered system of Figure 10 which implements an MLP we included another input and another output, otherwise there would be PEs that would not have input, or for which the output would not be connected to the external world.

The TLFNs can also be derived from the fully recurrent system when some of the PEs are made linear and their connectivity restricted to cascades as in the memory PE. This arrangement indicates a specialization (pre-wiring) for temporal processing.

Go to next section

# 8. Learning Rules for Recurrent Systems

## 8.1 Fixed Point Learning

How can we train recurrent systems to associate a static input with a static desired response? Here we will present an extension of backpropagation to train recurrent systems, which has been called *fixed point learning*. The cost function is given by Eq. 5 .

Our goal is to utilize the paradigm of static backpropagation, i.e. present a static input pattern, clamp it, compare the (steady state) response with the desired static response and backpropagate an error. With a local activation and a local error, any gradient descent update can be used to adapt the weights. Can backpropagation still be used for this task? Notice that now the system is not instantaneous as the MLP. When an input is presented to a stable recurrent system, the output **"**relaxes**"** (i.e. slowly evolves) to a final value. We say that the recurrent system has an *attractor* or a *fixed point*. After relaxing to a fixed point the system response does not change, i.e. the system becomes also static.

We are not going to provide a demonstration here (see Almeida ) , but backpropagation can be extended to train recurrent system with fixed points if the following procedure is followed:

1- Present the static input pattern for a number of samples (clamped) until the output stabilizes.

2- Compare the output with the static desired pattern, form the error and backpropagate the error through the dual system, just as was done in the static case.

3- Apply the error in the dual system (clamped) until the propagated error stabilizes.

4- Then utilize the search procedure of your choice to update the weights. Repeat the procedure for the next pattern.

There are three important remarks to be made. The input pattern and the error have to be clamped (held at a constant value) until the response stabilizes. This process converges as long as the forward system is stable, i.e. the system outputs must stabilize. If the forward system is stable the dual system is also stable, so the local errors will also stabilize. However, the relaxation time constants of the two systems can be different from each other and change from iteration to iteration. In fact experience shows that many times the relaxation time constants decrease (i.e. activations and error take longer to

stabilize) when we approach the solution.

The other aspect is that the learning rates have to be slow. Unlike the static case, here we have *two dynamical processes* at play: the dynamics of learning that change the system weights, and the dynamics of the recurrent system. The *dynamics of learning have to be much slower than the dynamics of the system* (adiabatic condition), otherwise we are not training one system but a family of systems, and no guarantee of convergence exists.

As a result of these four steps, we can see that standard backpropagation is a special case of the fixed point learning procedure (no relaxation). To implement fixed point learning one just needs to control appropriately the relaxation across iterations. So any of the recurrent topologies can be trained with fixed point learning if the goal is to create a mapping between two static patterns (the input and the desired response). An important question that does not have a clear answer is the advantage of using a recurrent topology versus a MLP to learn static mappings. Due to the fine-tuning required to successfully train a recurrent system with fixed-point learning we suggest that first the MLP be tried.

## NeuroSolutions 8
## 11.8 Learning the XOR with a recurrent neural network

**We will solve the old XOR problem but now using a recurrent topology and fixed point learning. The topology is a fully connected set of PEs that receive input from the file and produces an output through one of the PEs. We will use an ArbitrarySynapse with a weight of 1 at the output. All the other weights will be adaptive. We can solve this problem with just 2 PEs and this is what we will use.**

**We will use fixed point learning to train this system. If we go to the dynamic level of the controller inspector we can select fixed point learning. The difference between fixed point and trajectory learning is that the same sample is always sent to the network during the specified number of samples per exemplar. The goal is to let the system relax to a stable output such that we can computer a meaningful**

**error, send it to the dual network, let the dual stabilize and then compute the weight updates. The time the system takes to relax is dependent upon the current weights, so this is a difficult variable to set. But at an initial guess we can use a long relaxation time of 100 samples (enter 100 in the exemplar per sample window). We should monitor the appropriateness of the relaxation time by putting a scope on the hidden layer PEs.**

**We are ready to train the system. There is a local minima at 0.5 (input weights go to zero and the system stalls, so we may have to randomize the weights). The selection of the stepsize and momentum are very important for this problem. Notice that the system trains better for long relaxation times. We have two PEs but a lot of weights, so this topology is not as efficient as the MLP. It may however be more resistant to noise than the MLP.**

**NeuroSolutions Example**

## 8.2. Learning trajectories

One of the unique applications of recurrent networks is to learn time trajectories. A trajectory is a sequence of samples over time. Trajectory learning is required when we want to specify the system output at every time sample, such as in temporal pattern recognition, some forms of prediction (called multi-step prediction) and control applications. The cost function is given by Eq. 4 . We dare to say *that a recurrent system is naturally trained in time since it is a dynamical system*, so trajectory learning is rather important in many dynamical applications of neural networks. *Static networks can not learn time trajectories* since they do not possess dynamics of their own.

Unlike the previous case of fixed point learning, here we seek to train a network such that its output follows a pre-specified sequence of samples over time. In a sense, we do not only enforce the final position as in fixed point learning, but we are also interested in

defining the intermediate system outputs (i.e. a trajectory). Another case of interest is when the desired response is known at each time step but sometimes later in the near future. The learning system has to be run forward until the desired response is available, and then the weights updated with all the information from the time interval.

There are two basic principles to implement trajectory learning: real time recurrent learning (RTRL) and backpropagation through time (BPTT). We already covered these two training paradigms. With the same boundary conditions they compute the same gradient, hence they are equivalent. However, the computation and memory requirements of the two procedures are very different as we will see now.

## NeuroSolutions 9
## 11.9 Trajectory learning: the figure 8

**This problem will show the power that a recurrent system has to follow trajectories in time. We will start with a very simple case of learning an eight trajectory in 2 D space using an MLP. We will use prediction to train the system.**

**The more interesting thing is to actually see if the trained system can be used to generated the trajectory autonomously. Towards this goal we will feed the output back to the input and disconnect the input. Can the system still generate the figure 8? Try and verify that it is almost impossible….**

**We will then train the recurrent system with the global feedback loop, but we have to face a problem. Which will be the input to the system? The waveform generators or the feedback? Since we want the neural network to ultimately create the trajectory without an external input (just the feedback) we should disconnect the input. But notice that this became a very difficult problem to solve.**

**In fact after reset, the weights are going to be very different from the final position and so the output is going to be very different from the desired. So it will be very difficult to move the weights to the right values using the error computed over the trajectory. You may want to do this but it takes a long time to train and the**

**learning rates have to be small. Also you have to avoid local minima problem (e.g. if the trajectory length is equal to the length of the figure 8 zero is a local minimum).**

**An alternative is to use BOTH the input and the feedback, but in a scheduled way. In the beginning of training most of the information should come from the input to help put the weights approximately at the correct positions. But towards the end of training only the feedback should be active. We can either do this by segmenting the trajectory in a portion that comes from the signal generator and another portion that comes from the feedback, or simply create a weighted average between the two signals. In the beginning of training a large weight should be given to the signal generator, and towards the end of training the weight should be given just to the feedback. We will implement this latter here through a DLL with a linear scheduler.**

**Observe the system train (it takes a while…). Once it is trained you can fix the weights and you will see that it will keep producing the trajectory 8, i.e. we have created an oscillator with a very strange waveshape. This is a powerful methodology for dynamic modeling complex time series (i.e. we can replace the simple figure 8 by arbitrarily complex trajectories created by real world time series).**

## NeuroSolutions Example

A word of caution is in order here. In the literature sometimes we see training of recurrent networks with static backpropagation. This is NOT correct, since as we have seen the gradients in recurrent systems are time dependent. When static backpropagation is utilized we are *effectively truncating the gradient at the current sample*, i.e. we are basically saying that the dual system is static, which is at best an approximation and can lead to very bad results. On the other hand, this does not mean that we need the full

length of the trajectory to effectively compute the gradient over time. An interesting approach is to try to find out what is a reasonable number of samples to propagate back the gradients which leads to what has been called *truncated backpropagation (see Feldkamp )*. Truncated backpropagation is more efficient since we are no longer storing the activations and sensitivities during the full length of the trajectory.

## 8.3 Difficulties in adapting dynamic neural networks

It is fair to say that enough knowledge exists to train MLP topologies with backpropagation, which means that one can expect robust solutions. However, it is not easy to adapt dynamic neural networks, both TLFNs and fully recurrent topologies. The difficulty stems not only from the sheer computation complexity that produces slow training, but also from the type of performance surfaces, the possibility of instability (in fully recurrent), and the natural decay of gradients through the topology and through time. Training recurrent neural networks with BPTT (or with RTRL) is still today more an art than a science, so extreme care shall be exercised when training dynamic neural networks. We would like to add that TLFNs are easier to train than fully recurrent networks, and should be the starting point for any solution.

The performance surface of dynamic neural networks tend to have very narrow valleys and so the stepsize must be carefully controlled to train such systems. Adaptive stepsize algorithms are preferred here. In NeuroSolutions we can use the transmitters to schedule the stepsize by hand when automatic algorithms do not perform well.

During training the recurrent network can become unstable. The nonlinearity in the PEs will not allow the system to blow up, but the PEs will get pegged and may oscillate widely between the extreme values. Monitoring for this situation and resetting learning is normally the only way out. Effectively such PEs are not being used for processing functions, i.e. the effective number of degrees of freedom of the system is much less than the number of available PEs would suggest. There are definitions of stability more appropriate than BIBO stability for nonlinear systems but they are difficult to apply.

Another problem is the decay of the gradient information through the nonlinearities, which has been called the *long term dependency problem (see Bangio )*. A dynamic neural network in a classification task will tend to have its hidden PEs saturated to work as a finite state machine. In such cases the gradients will be attenuated greatly when the system is trained with BPTT, which makes training partial and very slow. There is no known mathematical solution to this problem, but engineering solutions have been proposed. This characteristic of dynamic networks implies that utilizing memory PEs inside the network topology as we did in the distributed TLFN network may simplify training and turn the networks more accurate. Advantage of linear memory PEs

All these aspects raise the question of the applicability of gradient descent learning to train dynamic neural networks. Recently, alternate training procedures using decoupled Kalman filter training have been proposed with very promising results. decoupled Kalman filtering

Go to next section

# 9. Applications of dynamic networks to system identification and control

The area of system identification and control is probably the leading candidate to benefit from the power of dynamic neural networks and BPTT. We already presented in Chapter X some applications of ANNs to system identification and control. The advantage of ANNs is that they are *adaptive universal mappers.* However, in Chapter X we did not know how to adapt the feedback parameters so we had to restrict ourselves to a few topologies (TDNN).

Now with the introduction of BPTT we can lift this restriction and apply ANY ANN topology (static or dynamic) to identify or control plants. There are many different ways to establish a successful strategy to identify and control plants. See Narendra for a

complete discussion. Here we will address some preliminary issues that are important for the understanding and application of the material covered in this book.

## 9.1 Adapting ANN parameters in larger dynamical systems

This issue was already covered in Chapter III where we showed how to optimally design a system built from an adaptive sub-system (a neural network) with some other sub-systems which were differentiable with fixed parameters, but static. Enhancing this view when the subsystems are dynamic is critical for system identification and control. In practical applications we may be faced with the task of including neural networks in larger systems built from unknown plants or other engineering systems (such as controllers), and of course we would still like to be able to optimally set the network parameters.

If you recall, the reason why we could easily adapt an ANN inside a larger system is that backpropagation is local in space. This means that the ANN only cares about the signals that were transmitted from the input (through an unknown subsystem) and backpropagated from the output (also through another unknwon subsystem). Since backpropagation was generalized to BPTT, we still can apply fully the same principle, i.e. we still can optimally adapt a dynamic system built from subsystems that have fixed parameters interconnected with a dynamic neural network. But since BPTT is not local in time, we have to work always with gradients from a segment of data. To preserve the ability to do on-line learning we have to advance the trajectory one sample at a time. However, for some applications the use of trajectory chunking is still practical and leads to more efficient implementations than RTRL. Probably the method more widely used in the control literature is dynamic backpropagation (see Narendra )

Dynamic backpropagation combines RTRL and BP. It adapts the static neural networks using BP, but uses RTRL to propagate forward the sensitivities. For the terminology of Figure 11, dynamic backpropagation can be written as

$$\frac{\partial e(n)}{\partial \alpha_i} = P(v,w)\frac{\partial v(n)}{\partial \alpha_i}$$

**Equation 31**

where the last term is computed with backpropagation while the sensitivity relative to the propagation of the parameter change is done with RTRL. dynamic backpropagation

Hence it can be implemented in a sample by sample basis (real time). The problem is that the forward propagation of sensitivities is very expensive computationally, and the method must be adapted to the particular system interconnectivity (block diagram). Let us study the most common examples (Figure 11). In this Figure, ANN means a neural network, while P(x,w) means a dynamic subsystem (linear or nonlinear) with constant parameters. We will assume that we only have available sequences of input-desired output pairs, and that we want to adapt the ANN parameters using this information. One can show (Narendra ) that for each case dynamic backpropagation can adapt the parameters of the ANNs

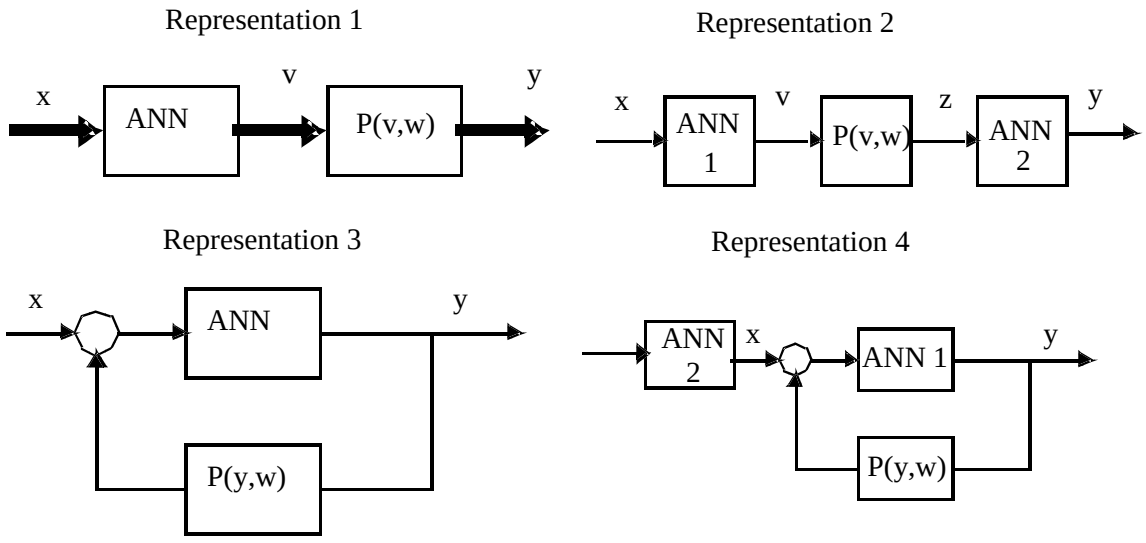



**Figure 11. The 4 most common configurations involving ANNs and fixed dynamical systems.**

The adaptation of the ANN parameters using the ideas of backpropagation through time requires simply the construction of the dual of P(v,w), and of the ANN dual according to the dataflow implementation we presented in Chapter III. This is in sharp contrast with dynamic backpropagation that requires much more computations and treats each of these cases differently.

Notice however that if the ANN is dynamic or the system P(v,w) is dynamic, BPTT must be utilized which means that a trajectory length has to be specified, and the flow of the BPTT algorithm must be preserved. *We can not do the adaptation in a sample by sample basis, only on a trajectory basis.* The length of the trajectory is important because it should be sufficiently long to capture the dynamics of the overall system, however this is not known a priori so the trajectory length is subject to experimentation. Notice also that to avoid discontinuities at the end of the trajectories the system state at the final trajectory time must be stored, and used as the initial condition for the following trajectory.

## NeuroSolutions 10
## 11.10 Training an embedded neural network in a dynamical system

**To show that we can use BPTT in a system built from ANNs and other dynamic components, let us solve the following problem. Consider a feedback system (representation 3) where the forward transfer function is** $g(v) = \frac{v}{1+4v^2}$ **and the feedback P(v,w) is linear and given by** $H(z) = \frac{(-1.12z + 0.33)}{z^2 - 0.8z + 0.15}$ **. The input is x(n) = 2sin(πn/25). The goal is to identify this feedback system assuming that we know the feedback transfer function, i.e. we want to identify the forward path only. However, we can not disconnect the two components, so training must be done in the integrated system since we just know the combined output. We will use BPTT for this task.**

**The breadboard has two basic networks. One called modeled plant and the other called ANN model. The modeled plant has an unknown part implemented by the top Axon with a DLL, and a feedback part which is known and also modeled by an Axon with a DLL. The feedback part is copied to the ANN model at the bottom of the breadboard and called known feedback model. The desired response for the ANN model comes from the modeled plant also through the DLL on the L2 component, while the input is fed both to the plant and the ANN model, so this is**

**pure system identification.**

**The ANN model is identifying only the feedforward part of the plant, which is easier than identifying the full plant with feedback. As a rule of thumb, we should include in our ANN models as much information as available from the external world. This breadboard shows how you can do it.**

**NeuroSolutions Example**

## 9.2. Indirect Adaptive Control

Probably the most general control problem is one in which we do not have an equation for the plant, but we have access to the plant input and output. How can we derive a model to control such plant? The block diagram we will be using is called indirect adaptive control and it is shown in Figure 12.

The main idea is to use two neural networks, *one to identify the plant*, and the *other to control the plant*. The identification model will then be used to pass the errors back to adapt the controller. If the plant is dynamic BPTT (or dynamic backpropagation) must be used. The Figure also shows an extra input-output loop that includes the reference model. The reference model dictates which will be the required plant behavior when the input is r(t). We can think that the one sample advance is a special case of the reference model.

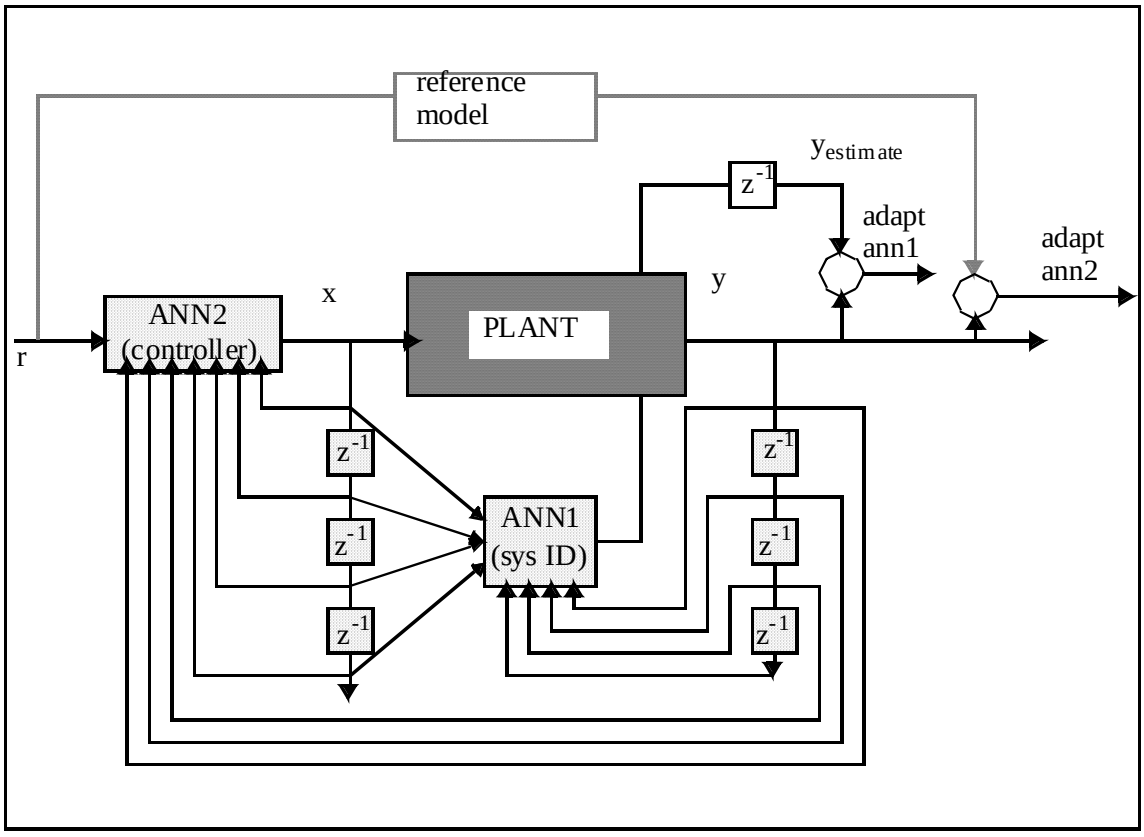



**Figure 12. Indirect adaptive contoller.**

Let us explain the Figure. Since we do not have an equation for the unknown plant it would be impossible to adapt the controller (ANN2) which requires sensitivities generated by comparing the reference input (desired behavior) to the plant output. Hence, we create a neural network identifier (ANN1) which will learn to produce an output which is similar to that of the plant. We will be using here the most general model for system ID explained in Chapter X, where the output is both a nonlinear function of the plant input (and its past values) and of the plant output (and its past values). But other choices are possible.

Once this ANN1 is adapted it behaves like the plant in terms of input-output, but most importantly, we created a known system description for which the dual can be obtained. Therefore, the controller ANN2 can be adapted by simply backpropagating the errors through the dual of ANN1.

This scheme requires that ANN1 be adapted first (eventually off-line), its parameters fixed and then the controller ANN2 adapted. When both the controller and the system identifier are trained at the same time, the stepsize for ANN 2 (the controller) has to be much smaller than that of the ANN 1 (the system identifier). Once again the rule of thumb of 10 times slower is a good starting point.

## NeuroSolutions 11
## 11.11 Training a neural controller

**We will solve the following problem with NeuroSolutions. Suppose that we want to control a plant of unknown input-output mapping using a direct controller (Fig 12). We just have the chance of exciting the plant and measuring the output. A fundamental problem is how to adapt the weights of the controller using BPTT, since we require the dual of the plant. So we have to use the black box approach of the indirect adaptive control. The reference model is just an advance of 1 sample. We will use a reference input r(n)=sin(2πn/25). The unknown plant is described by the second order differential equation**

$$y(n+1) = [f(y(n), y(n-1)] + x(n)$$

**where**

$$f[y(n), y(n-1)] = \frac{y(n)y(n-1)[y(n)+2.5]}{1+y^2(n)+y^2(n-1)}$$

**This function is implemented at the top Axon through a DLL and labeled model plant.**

**There are two part so the solution. The first part is to identify the plant, while the second is the training of the controller. In the first part we create ANN1 labeled ANN for system ID which is a one hidden layer focused TDNN receiving a common input (to the plant) which is filtered white noise. The desired response is transmitted from the plant output through the DLL placed on the L2 criterion. So this is the pure system identification configuration, and BPTT is used to adapt the plant model. In the second panel, we simply check the result of the system identification step. Notice that the backprop plane was discarded since the ANN1 (plant model) should keep the weights fixed after the identification.**

**In the third panel we implement the ANN2 (controller) of Fig 12. We use also a focused TDNN with one hidden layer to create the controller ANN2, adapted with BPTT. Notice the interconnections. ANN2 receives as input the reference input (delayed one sample), and its desired response is obtained from the difference of the plant output and the reference input. However, the error signal is propagated through the dual of the ANN1 (plant model). We have to assume that the first step is sufficient accurate since we are passing the forward signal through the unknown plant and the errors through the dual of the plant model. If the plant model and the plant are different this backpropagation strategy will have catastrophic consequences.**

**NeuroSolutions Example**

Go to next section

# 10. Hopfield networks

Hopfield networks are a special case of recurrent systems with threshold PEs, no hidden units, and where the interconnection matrix is symmetric. The original model did not have any self-recurrent connections (Figure13).

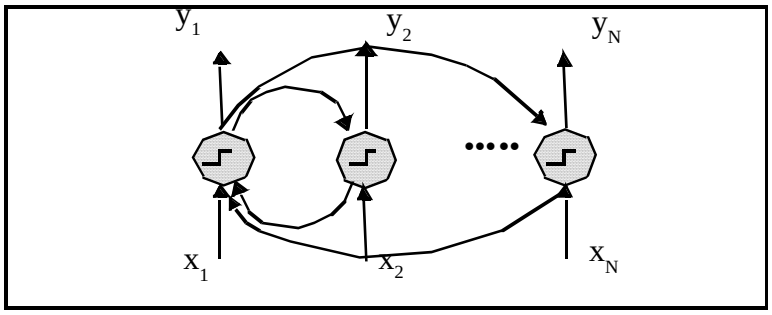



**Figure 13. The Hopfield network**

We will describe the discrete Hopfield net in discrete time as

$$y_i(n+1) = \text{sgn}\left(\sum_{j=1}^{N} w_{ij} y_j(n) - b_i\right) \qquad i = 1,\ldots N$$

**Equation 32**

where *sgn* represents the threshold nonlinearity, b is a bias. We assume that the update is done sequentially by PE number. The input binary pattern $\mathbf{x} = [x_1, x_2, \ldots x_N]^T$ works as an initial condition, i.e. it is presented to the network and then taken away to let the network relax. For simplicity the bias terms below are set to zero.

Let us see what is the function of such network. Suppose that we have an input pattern **x**, and we want this input pattern to be stable at the output, i.e. we want the system to be an autoassociator and produce an output pattern **y = x**. The condition for the fixed point implies

$$y_i = \text{sgn}\left(\sum_{j=1}^{N} w_{ij} y_j\right)$$

**Equation 33**

because the forward Eq. 32 will produce no change, i.e. the system is at the attractor. It can be shown (Hopfield ) that to meet this condition, the weight matrix becomes

$$w_{ij} \propto x_i x_j \longrightarrow w_{ij} = \frac{1}{N} x_i x_j$$

**Equation 34**

or in words, the outer product of the patterns automatically computes the weights of the Hopfield network without the need for any learning laws. Alternatively, we can utilize the Hebbian learning to create the weight matrix, just as in the static associative memory case.

The weight matrix **, W** is a symmetric matrix. But notice that this system is *dynamic*, i.e. when we present the input **x** to the network as an initial condition, the system dynamics produce an output sequence **y**(n) that takes some time to stabilize, but will approximate the input **x**.

**This first example with Hopfield networks will show the dynamics of the system. We have created a 3 input/3 output Hopfield network with two point attractors at the vertices of the unit cube (1,-1,1) and (-1,1,-1). Just examine the weight matrix to see that they comply with Eq. 34.**

**Each of these attractors creates a basin of attraction. The input will set the initial condition for the system state. Hence the system state will evolve towards the closest attractor under the "force" of the computational energy. This force is rather strong, as we can see by the "acceleration" of the system state.**

**Try inputs that will put the system state close to the boundary of the two basis (here the boundary is the plane x-y+z=0). At exactly these locations the final value is zero. However, all the other values will produce a convergence towards one of the two point attractors.**

## NeuroSolutions Example

What is interesting is that even when the input is *partially deleted* or is *corrupted by noise*, the system dynamics will still take the output sequence **y**(n) to **x**. Since the system dynamics are converging to **x**, we call this solution a *point attractor for the dynamics.*

This system can also store multiple patterns P with a weight matrix given by

$$w_{ij} = \frac{1}{N}\sum_{p=1}^{P} x^{p}{}_{i}\, x^{p}{}_{j}$$

**Equation 35**

Just like what happened in the feedforward associative memory, there is cross-talk between the memory patterns, and the recall is only possible if the number of (random) input patterns is smaller than 0.138 N, where N is the number of inputs. For the case of very large N, the probability of recall (99% of the memories recalled with an asymptotic probability of 1) of m patterns is guaranteed only if

$$\frac{m}{N} < \frac{1}{4\ln N}$$

**Equation 36**

which is rather disappointing since it approaches zero even for large N. So the intriguing property of the Hopfield network is pattern completion and noise removal, which is obtained by the convergence of the system state to the attractor.

**NeuroSolutions 13**
**11.13 Pattern completion in Hopfield networks**

**This example is to show the convergence of the Hopfield network in a more realistic situation. We have created a 64 input Hopfield to present the characters developed in Chapter VI when we also treated the case of associative memories. We would like to show that the same behavior of association is also present in the Hopfield case.**

**We created a weight matrix "by hand" using Eq. 34. Then we can present the pattern at the input (just once) and observe the output pattern appear. This will be very fast (just a few iterations, so you may want to single step through the simulation).**

**A more interesting thing is to show that even when the patterns are noise or fragmented the end result is still the full pattern. We can understand this in terms of basins of attraction. If the input puts the system state at the appropriate basin of attraction the final output will be the stored pattern which is the complete (or noise free) character.**

## NeuroSolutions Example

## 10.1 The Energy Function

We just presented the practical aspects of Hopfield networks, but we assumed that the dynamics do in fact converge to a point attractor. Under which conditions can this convergence be guaranteed? This is a nontrivial question due to the recurrent and nonlinear nature of the network.

The importance of the Hopfield network comes from a very inspiring interpretation of its function provided by Hopfield . Due to the extensive interconnectivity it may seem

hopeless to understand what the network is doing when an input is applied. We can write the dynamical equations for each PE as Eq. 32 , but due to the fact that these difference equations are highly coupled and nonlinear their solution seems to be beyond our reach. Effectively this is not so. When the weight matrix is symmetric, the PEs are threshold nonlinearities and the biases are zero, one can show that the network accepts an *energy function* H

$$H(y) = -\frac{1}{2}\sum_i \sum_j w_{ij} y_i y_j$$

**Equation 37**

derivation of energy function

The energy function is a function of the configuration of the states $\{y_i\}$ which is *non-increasing when the network responds to ANY input*. We can easily show this by taking into account that every time one PE changes state, H will decrease. When the PE does not change H remains the same. Hence, this means that the global network dynamics are pulling the system state to a minimum (along the gradient of H), *which corresponds to one of the stored patterns*. The location of the minimum in input space is specified by the weights chosen for the network. Once the system reaches the minimum (the memory) will stay there so this minimum is a fixed point or an attractor .

When the system receives an input, the system state is placed somewhere in weight space. The system dynamics will relax to the *memory that is closest to the input pattern*. So around each fixed point there is a *basin of attraction* that leads the dynamics to the minimum (effectively there are some problems, since when we load the system with patterns, spurious memories are being created which may attract the system to unknown positions). We can understand the reason why the Hopfield network is so robust to imprecisions (added noise or partial input) of the input patterns. Once the system state is in the basin of attraction for a stored pattern, the system will relax to the undistorted pattern.

The conjunction of an energy surface with minima and basin of attraction creates the

mental picture of a *computational energy* landscape, which is similar to a particle subject to a gravitational field. This metaphor is very powerful, because all of a sudden we are talking about global properties of a tremendously complex network in very simple terms. We have ways to specify each connection locally, but we also have this powerful picture of the computational energy of the system. This is pictorially depicted in Figure 14.

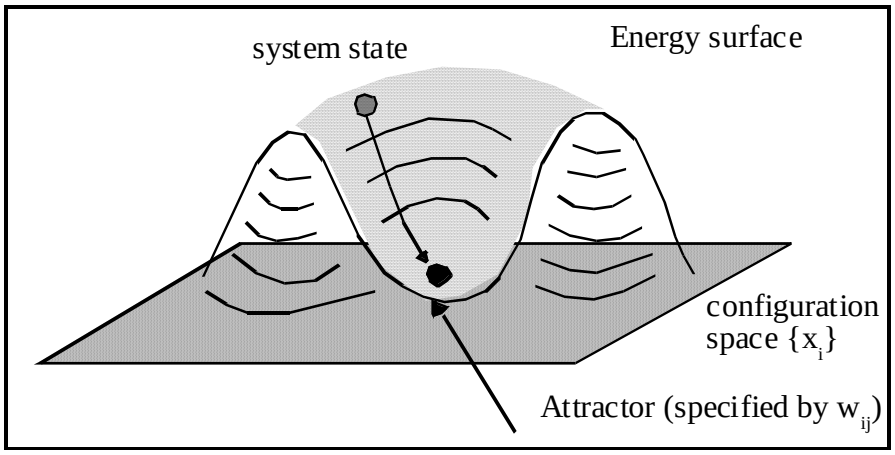



**Figure 14. Computational energy over the configuration space**

Hopfield provided until today one of the strongest links between information processing and dynamics. The existence of the computational energy function makes the convergence of the state of the Hopfield net to the stored pattern the same dynamical problem as a ball rolling down a hill in a gravitational field (which also accepts a potential function). We can then say that a *dynamical system with point attractors implements an associative memory*. No matter if the **"**hardware**"** is VLSI, a computer running these algorithms, biological neurons, or a solar system. Hopfield**'**s view was crucial for the revival of neural networks in 1987.

Note that the Hopfield net, although a dynamical system, becomes **"**static**"** after convergence. So other paradigms must be searched to understand the temporal processing that goes on permanently in our brains. Notice that dynamical systems may have singularity higher than the zero order (point attractor), such as limit cycles and chaotic attractors.

The Hopfield network normally is used with the weights pre-specified. They can be computed easily for the case of the associative memories. We can also apply the fixed point learning algorithm to train a single layer net as an associative memory. However, there is no guarantee that the net is an Hopfield net, unless the weight matrix symmetry is externally enforced.

### NeuroSolutions 14
### 11.14 Training a recurrent network as an associative memory

**Up to now we have always created the weights of the Hopfield network by hand, i.e. loading them without training. However, we can also use fixed point learning as the training rule to load the weights. Fixed point learning is selected in the dynamic controller (fix point in the dynamic level).**

**In general we can no longer say that we have an Hopfield network since the weights may not be symmetric. But this should not discourage us of using such a training procedure. In rare occasions the system may go unstable, but we have workable solutions that are obtained in a more "neural network" like procedure without actually entering by hand the weights.**

**In this example we will use the association between long distance phone calls and their price (same example of Chapter VI) to show that the "Hopfield like" network can be used as an heteroassociative memory.**

## NeuroSolutions Example

Hopfield networks have also been used in optimization, because we can link the energy surface to problem constraints. The solution is simply found by relaxing the system with the present input to find the closest solution (attractor). However, the problem of the spurious minima have limited the practical applications of such networks.

### NeuroSolutions 15
### 11.15 Hopfield networks for optimization

**The convergence of the Hopfield network to point attractors can be explored also for optimization problems. Just imagine that the solution of an optimization**

**problem is coded in the energy function of the network. Then when an input is given to the net, the system state will evolve providing the optimal answer to that particular instance, since the Hopfield net minimizes the energy. The difficulty is to encode the optimization problem in the energy function. But once this is done the simple application of an example to the net will provide an output that is the desired answer.**

**Here we will treat a simple problem: A/D conversion. In fact Hopfield and Tank showed that we can see the problem of converting an analog signal into bits as an optimization, which then can be solved by an Hopfield network.**

**We will restrict our example to two digit conversions, i.e. we will try to approximate the integer input X by**

$$X \sim x_1 + 2x_2$$

**where $x_1$ and $x_2$ will be 0 or 1. Hence the range for X is the set of integers [0, 3]. Hopfield proposed the performance index**

$$J(\mathbf{x}) = 0.5[X - \sum_{i=1}^{2} x_i 2^{i-1}]^2 - 0.5[\sum_{i=1}^{2} 2^{2i-2} x_i (x_i - 1)]$$

**where the first term computes the error and the second is a constraint that is minimized when $x_i$ are 0 or 1. This leads to a Lyapunov function of the type of Eq. 35 (but extended with the bias term) which can be implemented with**

$$W = \begin{bmatrix} 0 & -2 \\ -2 & 0 \end{bmatrix} \qquad b = \begin{bmatrix} X - 0.5 \\ 2X - 2 \end{bmatrix}$$

**So the user supplied value X is applied to the bias of the two PEs instead of the conventional input (as in the associative memory example). We have created in NeuroSolutions a very simple breadboard and two input fields. The user should enter 0,1,2,3 in the box labeled X and the double of the first digit in the box 2X. Try several inputs and observe the network provide the result.**

## NeuroSolutions Example

Although very important from the conceptual point of view, Hopfield networks have met applications with limited success, mainly due to the spurious memories that limit the capacity to store patterns.

## 10.2. Brain state in a box model

Another very interesting dynamic system that can be used as a clustering algorithm is Anderson's Brain State in a Box (BSB) model. BSB is a discrete time neural network with continuous state (as the Hopfield with sigmoid nonlinearities). However, the equation of motion is

$$y_i(n+1) = f(x_i(n) + \alpha \sum_{j=1}^{N} w_{ij} x_j(n))$$

$$f(u) = \begin{cases} 1 & if \quad u \geq 1 \\ u & if \quad -1 \leq u \leq 1 \\ -1 & if \quad u \leq -1 \end{cases}$$

**Equation 38**

The nonlinearity is threshold linear. We assume here that there is no external bias, and $\alpha$ is a parameter that controls the convergence to the fixed points. The name BSB comes from the fact that such a network has stable attractors in the vertices of the hypercube $[-1,+1]^n$, provided that W is symmetric and positive semidefinite or $\alpha \leq \frac{2}{|\lambda_{\min}|}$. We can show that BSB behaves as a gradient system that minimizes the energy of Eq. 37 . The weight matrix to create the memories in the vertices of the hypercube follows the Hebbian rule of Eq. 35 where $x_i$ are restricted to be +/-1. One can show that the asymptotic capacity of the BSB model is identical to the Hopfield network.

However, the BSB model is normally utilized as a clustering algorithm instead of an associative memory as the Hopfield network. Both systems create basin of attractions and have point attractors, but the BSB has faster convergence dynamics and the basins of attraction are more regular than in the Hopfield network. Hence it is possible to divide the input space in regions that are attracted to the corners of the hypercube, creating the

clustering function.

**NeuroSolutions 16**
**11.16 Brain State in a Box model**

**The BSB model is basically a positive feedback nonlinear system. Due to the shape of the nonlinearities the system will have to converge to one of the corners of the hypercube, i.e. the system will amplify the present input until all of the PEs are saturated. Hence the BSB is very sensitive to the initial pattern position in the input space, which has yielded its application in decision feedback. In order for the corners of the hypercube to behave as point attractors it is enough that the diagonal elements of the weight matrix be larger than the off-diagonal elements.**

**Here we will have a simple example of the BSB. We created a 2D example. Enter a value in the boxes and you will see the system relax to the closest corner. So the system is similar in function to the Hopfield network, but here the location of the attractors is predefined at the vertices of the hypercube. Hence it can be used for clustering.**

**NeuroSolutions Example**

Go to next section

# 11. Grossberg's additive model

If we abstract from the interconnectivity of the PEs, we realize that all the neural networks presented so far have a lot of things in common. Each PE has simple first order dynamics, it receives activations from other PEs multiplied by the network weights, adds them, feeds the result to the static nonlinearity, and finally weights the result by the previous value. This sequence of operations can be encapsulated in continuous time for each PE by a first order differential equation as follows,

$$\frac{d}{dt} y_i(t) = -\mu y_i(t) + f\left(\sum_{\substack{j \\ j \neq i}} w_{ij} y_j(t) + b_i\right) + I_i(t) \qquad i = 1,\ldots N$$

**Equation 39**

where I(t) represents the forcing function and $y_i(t)$ represents the state of the dynamical system, and N is the number of PEs in the system. This is called *Grossberg's additive model*, and it is one of the most widely utilized neural model*. In the additive model the weights are not a function of the state nor of time (after adaptation)*. Eq. 39 implements a computationally interesting model if the three dynamical variables, I(t), y(t), and *w* span three different time scales. The parameters *w* contain the long term system memory and *y(t)* the short term system memory. When *I(t*) is presented to the neural system, *y(t)* reflects the degree of matching with the long term memory of the system contained in the weights. Figure 15 shows the block diagram of Grossberg's model.

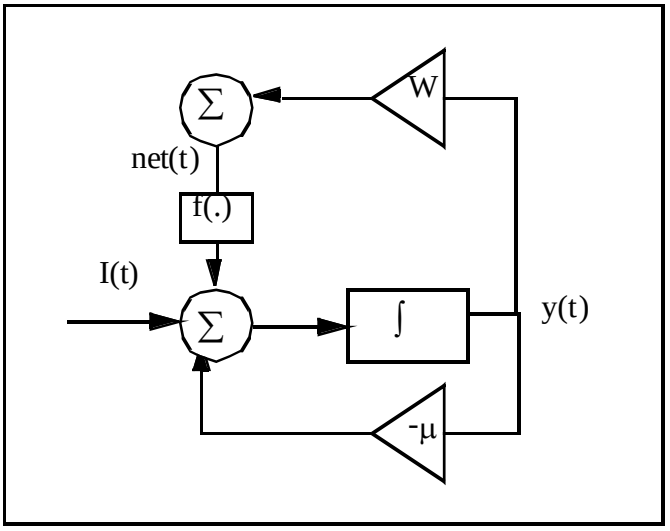



**Figure 15. Grossberg's neural model**

Notice that all the connections are vectorized (multidimensional pieces) and they represent the computation, not the actual topology. A *neural model is an abstraction of the neural network* because it characterizes the dynamics, and does not care about the interconnection matrix, i.e. the topology that we normally call the neural network. In the additive model, the interaction among the PEs is additive, and the PEs have first order dynamics. It is also interesting that the additive model without the nonlinearity f(.) defaults

to the ARMA model, the most general linear system model.

In discrete time the equation becomes a difference equation of the form

$$y_i(n+1) = (1-\mu)y_i(n) + f\left(\sum_j w_{ij}\, y_j(n) + b_i\right) + I_i(n) \qquad \begin{matrix} i = 1,\ldots N \\ j \neq i \end{matrix}$$

**Equation 40**

Grossberg's additive model gives rise to all of the neural networks studied in this book! A dynamical neural network is obtained when the *external patterns are attached to I(n), and the initial states* of the dynamical system are *held constant* (normally zero) in Eq.40. Fully recurrent neural networks are the most general implementation of dynamical neural networks. But some other special cases are also possible. For instance, when the first term of Eq. 40 is zero, it means that there are no self connections. When the first term is zero and the sum index j is kept smaller than i (i>j) the topology is feedforward but dynamic. TLFNs are also a special case of this topology, where some of the PEs are linear and pre-wired for time processing. TLFN architectures

The static mappers (MLPs) are obtained when the connections are restricted to be feedforward, *the external inputs are applied to the initial states, and the forcing functions are zero (I(n))* in Eq. 40 . When the initial states of the dynamical system are clamped by the inputs, there is no time evolution of states that characterize the relaxation, and y(n+1) can be computed in zero steps.

Recurrent neural networks are regarded as more powerful and versatile than feedforward systems. They are able to create dynamical states. They have a wealth of dynamic regimes (fixed points, limit cycles, chaotic attractors) which can be changed by controlling the system parameters. So they are very versatile and useful to model signals that are time varying, such as in time series analysis, control applications and neurobiological modeling.

Go to next section

# 12. Beyond first order dynamics: Freeman's model

Neurocomputing has been centered around Grossberg's neural model for many years, but alternate models exist. Here we briefly describe a biological realistic model of the cortex that has been proposed by Freeman following his studies of the rabbit olfactory system. The interesting thing about Freeman's model is that it is a computational model built from local coupled nonlinear oscillators, and it produces chaotic activity. Information is processed by the global chaotic dynamics, unlike any of the previous models where information processing requires stable dynamic regimes (remember the fixed points of Hopfield's associative memory).

The simplest biological system we model is the cell assembly, an aggregate of hundred of neurons. Freeman models the cell assembly (K0 model) as a second order nonlinear dynamical system which is composed of a linear part given by Eq.41

$$\frac{1}{a.b}\left[\ \frac{d^2x(t)}{dt^2}+(a+b)\frac{dx(t)}{dt}+abx(t)\ \right]=Q(x(t))$$

**Equation 41**

where *a* and *b* are constants, *x(t)* represents the system state and Q(x(t)) is a forcing function. This forcing function brings the contributions of other PEs through a nonlinearity Q(x) where

$$Q(x)=\begin{cases} Q_m[1-\exp(-\frac{e^x-1}{Q_m})] & if \quad V>-u_0 \\ -1 & if \quad V\le -u_0 \end{cases}$$

**Equation 42**

The nonlinearity belongs to the sigmoid class but it is *asymmetric*. As we can see this neuron model is divided into two parts: one is a linear time dependent operator defined as a second order differential equation followed by a non-linear static nonlinearity. This division is in tune with the models utilized in neurocomputing. The nonlinearity models the synaptic transmission, while the dynamical equation models the transmission through

axons and integration though dendrites.

In order to get a discrete network (a neural network) to approximate the continuous time behavior of Eq. 41, the impulse response of the linear sub-system will be digitized and approximated by a three tap gamma filter which we already described. This provides the following implementation for the K0 model (Figure 16)

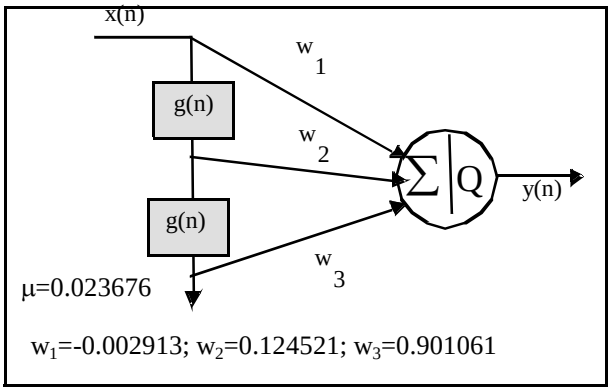



**Figure 16. Gamma implementation of the K0 model.**

The next level in the hierarchy is the modeling of interactions between cell assemblies. Freeman proposes that the individual cell assemblies modeled by the K0 are effectively connected by excitatory and inhibitory interactions with constant coupling coefficients, which represent (mitral-granule cell) interactions. He proposes the development of tetrads of K0 models, interconnected as shown in the Figure 17 ,

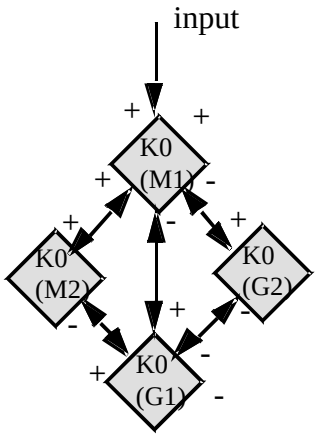



**Figure 17. KII model called here the Freeman's PE**

where the minus and the positive signs mean inhibitory and excitatory connections respectively. Each group is described by a set of 8 differential equations with fixed connections. Since all the elements so far are fixed, they represent the building block for the model of the olfactory system, and will be called the *Freeman's PE*. Freeman's PE models the processing going on in the cortical column. Each column is therefore a neural oscillator. Table I shows the coefficients of Freeman's PE which have been derived from neurophysiologic measurements. The same Table also shows the patterns stored in the model for the simulations.

**Table I - Parameters Set (e.g. WMG Means gain from element G to M. W1-4 are gains associated with f1-4(.)**

| | |
|---|---|
| **KII** | **[WMM_H; WMM_L; WGG]**= [0.3; 1.5; 0.25]<br>**[WMM; WMG; WGM; WGG;]**= [0.3; 5; 0.2, 0.25] |
| **Patterns Stored** | $\begin{bmatrix} 1&0&0&0&0&1&0&0&0&0&1&0&0&0&0&1&0&0&0&0 \\ 0&0&1&1&0&0&0&1&0&0&0&1&0&0&0&0&0&0&1&0 \\ 0&1&0&0&1&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&1&0&0&0 \\ 0&0&0&0&0&0&0&0&1&1&0&0&0&1&0&0&0&1&0&1 \end{bmatrix}$ |
| **KIII** | **[WPP; WMM_H; WMM_L; WGG]**= [0.2; 4; 0.5, 1.5]<br>**[WMP; WEM; WAM_L; WCB; WBC]**= [0.5; 1.0; 1.0; 1.5; 1.0] |
| **(Internal)** | **[WMM; WMG; WGM; WGG;]**= [0.25; 1.5; 1.5, 1.8]<br>**[WEE; WEI; WIE; WII]**= [1.5; 1.5; 1.5, 1.8]<br>**[WAA; WAB; WBA; WBB]**= [0.25; 1.4; 1.4, 1.8]<br>**[W1; W2; W3; W4]**= [1.6; 0.5; 2, 1.5]<br>**[Te1; Te2; Te3; Te4]**= [11; 15;12, 24]<br>**[Ts1; Ts2; Ts3; Ts4]**= [20; 26; 25, 39] |
| **Patterns Stored** | $\begin{bmatrix} 1&0&0&1&0&0&1&0 \\ 0&1&0&0&1&0&0&1 \end{bmatrix}$ |

The KII model is a connected arrangement of Freeman's PEs (each K0 is connected to all other K0s in the same topological position) as shown in Figure 18.

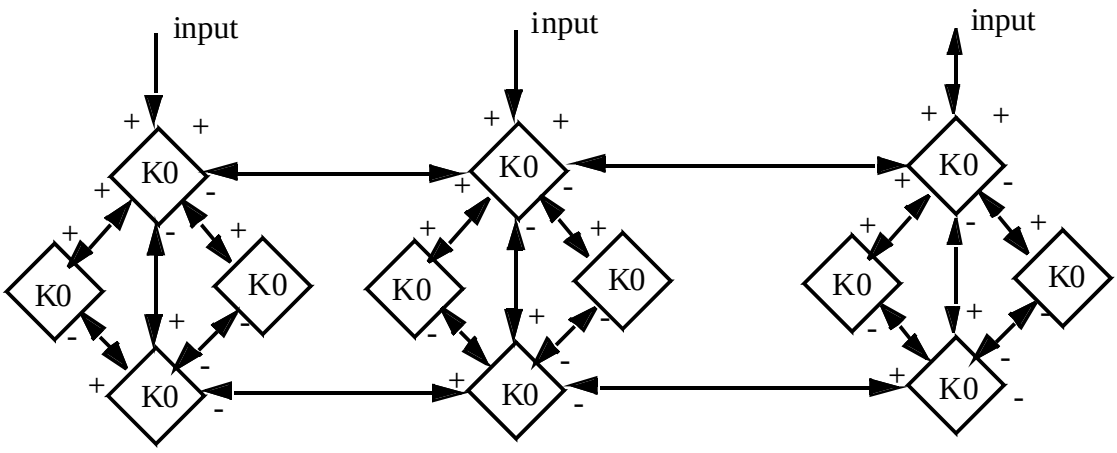



**Figure 18. The connections of several Freeman's PEs.**

The KII system models the olfactory cortex as a dynamical system with two basins of attraction. When no forcing input is applied the KII model has a fixed point at the origin, but when an input is applied the system changes state to an oscillatory regime with large amplitude. The following simulation exemplifies the dynamics.

**NeuroSolutions 17**

11.17 Simulating Freeman's KII model

**This example shows the NeuroSolutions implementation of the KII model. The KII model is an arrangement of KI sets which we called the Freeman PE. The Freeman PE is an arrangement of 4 K0 sets which are approximated by a gamma network of Fig 16. The connections to form the KI set (internal unit feedback) and KII (global feedback among channels) have been separated for readability. Notice that a DLL implements the asymmetric sigmoidal nonlinearity.**

**We would like to show the response of the KII model to a step input. The model is a 20 PE system that receives 1/0 from the input. When we plot the response of one of the PEs that receive an input, we will see that the amplitude of the PE oscillation will increase. When the input is set to zero the system state slowly decreases to zero. We will depict time plots of the input and outputs of an active and inactive PEs. We will also show the phase plot of one of the active Freeman PEs (output of**

**the excitatory input versus the inhibitory input). This example shows that the local PEs are stable oscillators with an amplitude modulated by the input. Hence the codification of information is contained as a spatio-temporal pattern of amplitudes over the model.**

## NeuroSolutions Example

According to Freeman, the central olfactory system consists of two layers of coupled oscillators, the olfactory bulb (OB) and the prepyriform cortex (PC) mediated by lumped control of the anterior nucleus (AON). The receptor input goes to periglomerular (PG) and mitral cells (M). The mitral cells transmit to granule cells (G) and to AON and PC. From PC the final output is sent to other parts of the brain by deep pyramidal cells (P) as well as back to the OB and AON.

The central olfactory system will be implemented by the KIII model, which is an hierarchical arrangement of KII models (Fig. 19). The input is a fixed weight, fully connected layer of excitatory K0 models (the periglomerular cells - PG). The second layer is the KII model described above (which models the olfactory bulb -OB). This is were learning takes place by changing the excitatory to excitatory connections using a modified Hebbian learning. The next layer is a single Freeman PE (modeling the anterior olfactory nucleous - AON), followed by another Freeman PE (modeling the prepyriform cortex - PC), which finally connects to a K0 set (modeling the pyramidal cells - P). There are extensive (but not fully) connections in the topology: the output of the OB layer is fed to the AON and the PC layers with diffuse connections (modeling the medial olfactory tract). Feedback transmission is implemented from the P and PC layers through long dispersive delays (modeling the medial olfactory tract - MOT) which are represented in Figure 19 as f(.). The parameters of the Freeman's PE in layers AON and PC are different from the OB layer (in fact creating incommensurate oscillating frequencies) - see Table.

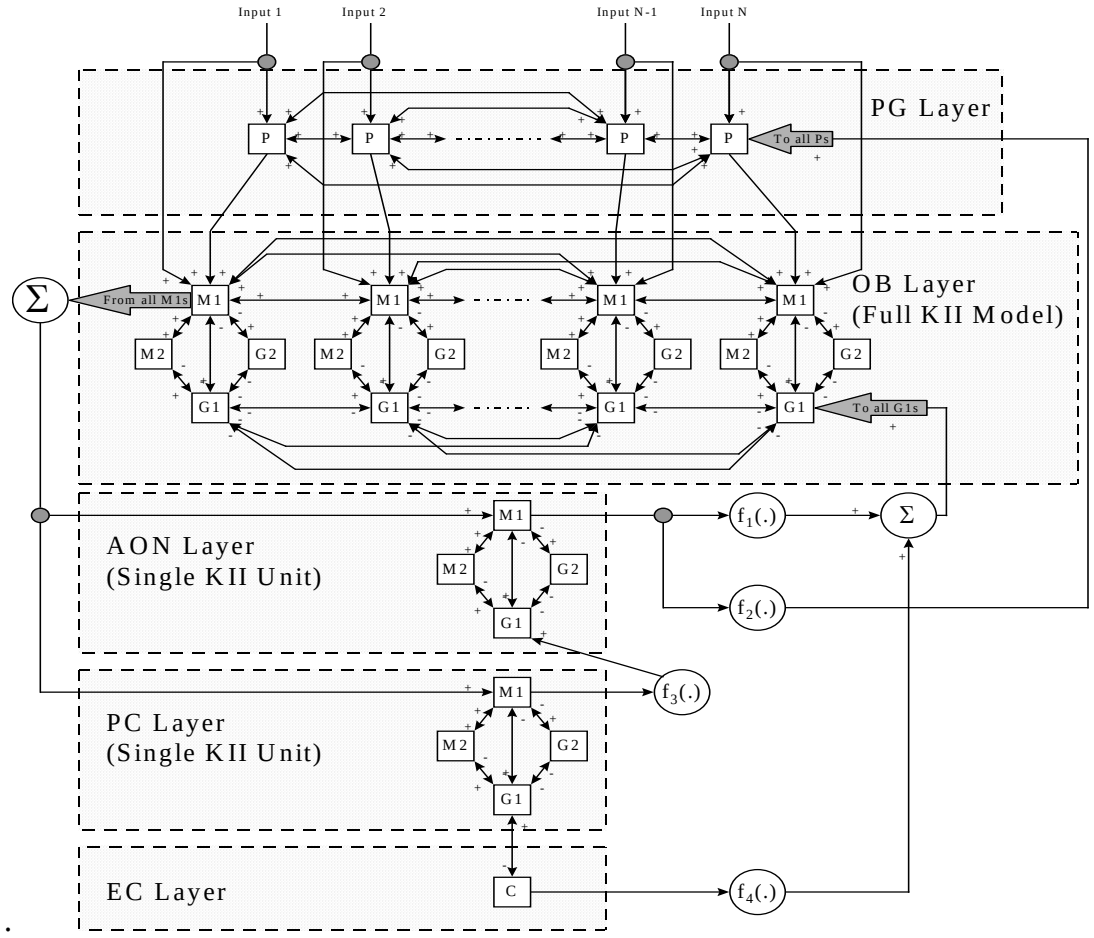



**Figure 19. KIII model**

The KIII model is a chaotic dynamical system. With no input the system at the OB layer produces chaotic time series with small amplitude and no spatial structure (the basal state). When a previously stored pattern is presented to the system, the OB layer still produces chaotic time series, but "resonates" in a stable spatial amplitude distribution. This spatial distribution is what codifies the stored pattern. Hence, the system works as an associative memory. However, the interesting thing is that the dynamics are no longer towards a point attractor as in the Hopfield network, but to a spatio-temporal chaotic attractor. Freeman states this system is able of very fast switching among patterns and is robust to noise in the patterns.

**NeuroSolutions 18**
**11.18 Simulation of Freeman's KIII model**

**This example will demonstrate the oscillation produced by a 8 PE OB layer. The breadboard is rather large. We have labeled each block according to Fig 17. Notice that the breadboard is a replication of basically two pieces: the KII model and the gamma filter implementing the dispersive delay operator. Each has different parameters according to biological measurements.**

**The input is a rounded pulse. When there is no input, the system lies in a chaotic basal state where the amplitude of each PE time series has the same basic amplitude but very complex time series as shown in the scope. When the pulse is applied the system jumps to a new state, and the phase space plot (excitatory versus inhibitory PEs) shows a phase transition to one of the wings of the chaotic attractor. When the input disappears the system goes back to its initial chaotic basal state. The system transitions are very repeatable and fast. Notice that we do not need any extra mechanism to take the system from the pattern created by the excitation (as the fixed point of the Hopfield net). The information is still coded in the amplitude of the chaotic oscillation across the PEs. All the parameters of the simulation were fixed (i.e. no learning is taking place).**

**NeuroSolutions Example**

The exciting thing about Freeman's model is that it extends the known paradigm of information processing which have low order dynamics (Hopfield nets have dynamics of order zero, i.e. fixed points) to higher order spatio-temporal dynamics. Moreover, the model was derived with neurophysiologic realism. It is too early to predict its impact in artificial neural networks research and information processing paradigms, but it is definitely an intriguing dynamical system for information processing.

Go to next section

# 13. Conclusions

This Chapter presented the development of learning rules for time processing with

distributed TLFNs and fully recurrent networks. We showed what is the difficulty of training networks with delays, and we presented an extension to static backpropagation that was able to train topologies with delays. Backpropagation through time (BPTT) shares some of the nice properties of static backpropagation (such as locality in space, efficiency since it uses the topology to compute the errors) but *it is non-local in time*. This creates difficulties that require the use of memory and backpropagating the error from the final time to the initial time (hence the name). With BPTT we harness the power to train arbitrary neural topologies to learn trajectories which is necessary to solve problems in controls, system identification, prediction and temporal pattern recognition. In a sense, this chapter closes the circle started with static nonlinear neural networks and time processing. Now we have the tools to train nonlinear dynamical systems.

In this chapter we also present a view of dynamic neural networks as nonlinear dynamical systems. Hopefully the difference between static and dynamic neural networks became more apparent. We can not discuss recurrent neural networks without presenting Hopfield networks, mainly due to the perspective of the computational energy. This analogy provides a “physical” view of computation in distributed systems, which makes us wonder about the nature of computation and the tools to quantify and design computational distributed systems.

We also alerted the reader for the idea of neural models. Neural models sit at the top of the hierarchy to systematize neural computation. Although we covered many neural networks throughout this book (linear regressors, MLPs, Hebbian, RBFs, Kohonen, linear combiners, TLFNs, recurrent networks), *they all belong to the same neural model*. This clearly leaves open the search for other neural models.

We finish the Chapter by providing a glimpse of other computational paradigms that may become very important in the future due to their biological realism and information principles based on higher order (chaotic) dynamical regimes. Freeman’s model points the ultimate paradox: the quest for systematization (understanding) and organization that

characterizes humans and their society emanates from brains that may be based on higher order nonlinear (chaotic) dynamics.

## NeuroSolutions Examples

11.1 Jordan's network

**11.2 Elman's network**

11.3 Train the feedback parameter of Elman's network

**11.4 The effect of slope of nonlinearity in recurrent networks**

11.5 The issue of memory depth in the bus driver problem

**11.6 Output MSE and the gamma memory depth**

11.7 Frequency doubling with a focused TLFN

**11.8 Learning the XOR with a recurrent neural network**

11.9 Trajectory learning: the figure 8

**11.10 Training an embedded neural network in a dynamical system**

**11.11 Training a neural controller**

**11.12 Hopfield network dynamics**

**11.13 Pattern completion in Hopfield networks**

**11.14 Training a recurrent network as an associative memory**

**11.15 Hopfield networks for optimization**

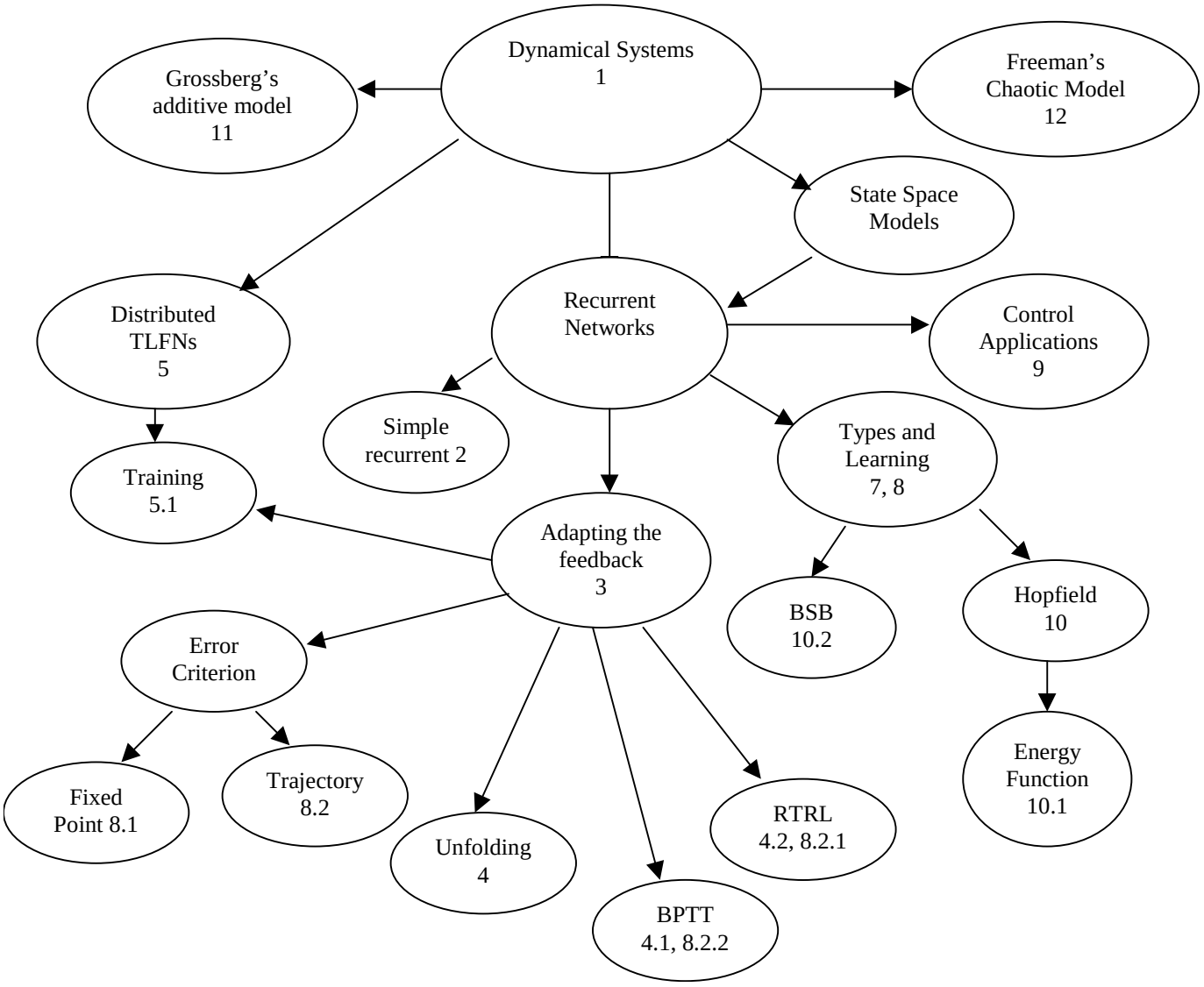
**11.16 Brain State in a Box model**

**11.17 Simulating Freeman's KII model**

**11.18 Simulation of Freeman's KIII model**

## Concept Map for Chapter XI

# Chapter XI




Go to Table of Contents

Go to Appendix

# backpropagation versus BPTT

If the network is static or dynamic but feedforward as the TDNN, and the desired signal exists for all time even for the case of trajectory learning there is no point of using BPTT. We can use straight backpropagation and add up the gradients over the specified time interval, as we did in batch learning. This can be easily proved by analyzing Eq. 4 . However, there are cases where the dynamic network is feedforward (TDNN) but the desired signal is only known at the terminal time. In this case we have to use BPTT since we do not have an explicit error at each time step.

As was specified in the text, once BPTT is applied to one portion of the network (for instance the feedback parameter in the gamma network) the easiest way is to adapt ALL the weights with BPTT , although strictly speaking, only the weights that lie in the topology to the left of the recurrent connection need to be updated with BPTT. However, it is easier to implement the learning algorithm uniformly.

Return to Text

# vector space interpretation of TLFNs

The vector space interpretation of a memory PE explains the adaptation of the TLFN weights as a local approximation of the desired response by a weighted sum of memory traces $y_k(n)$. These signals are the bases of the local projection space. Let us present the most familiar connectionist memories in this framework. We will be studying the case of a memory PE inside the neural network (Figure 20).

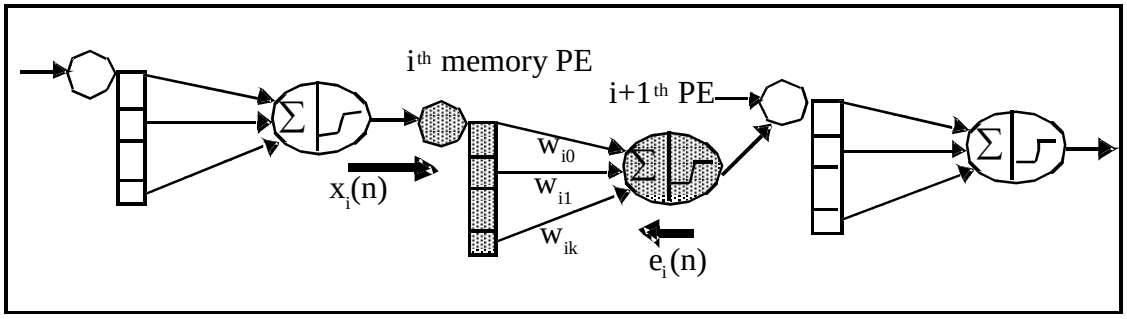



**Figure 20. Analysis of the function of the i**th memory PE

The first case discussed is when the memory PE is a context PE. In this case the projection space can not be controlled directly, as we saw in Chapter X. The system can only control the final projection through two degrees of freedom: one weight $w_{i0}$, and one feedback parameter $\mu_i$. So both the error and the input represent a point in a space of dimension given by the memory depth of the PE (which is $1/\mu$). The processing goal is to find the size and rotation for the one-dimensional projection space by changing $\mu_i$, and then finding in the line the point that is perpendicular to the error by adjusting $w_{i0}$ (i.e. the optimum projection according to Kolmogorov). As was discussed in Chapter X, this representation is appropriate when the problem requires long memory depth but does not require resolution (i.e. detail).

In the delay line PE case, $x_i(n)$ is projected in a memory space that is uniquely determined by the input signal, i.e. once the input signal x(n) is defined, the basis become x(n-k) and the only degree of freedom defining the size of the projection space is the memory order K. We can not change is the size of the projection space without a topological modification.

This memory structure has the highest resolution but lacks memory depth, since one can only improve the input signal representation by increasing K, the order of the memory. In terms of versatility, the simple context unit is better (or any memory with a recursive parameter), since the neural system can adapt the parameter $\mu$ to better project the input signal.

The gamma PE produces a basis that represents memory space of size K, but the basis are no longer a direct use of the past values of the input as in the ideal delay line. The basis are already a linear projection of the input trajectory (convolution with the gamma kernel). When the parameter $\mu$ adapts to minimize the output mean square error, the projections rotate and the span of the projection space changes as a consequence of the different time constants (the effective length of the impulse response also changes). This is the same phenomena as described by the context PE. However, the relative angle among the gamma bases vectors does not change with $\mu$. Hence, a decrease in the error must be associated with a decrease in the relative angle between the desired signal and the projection space.

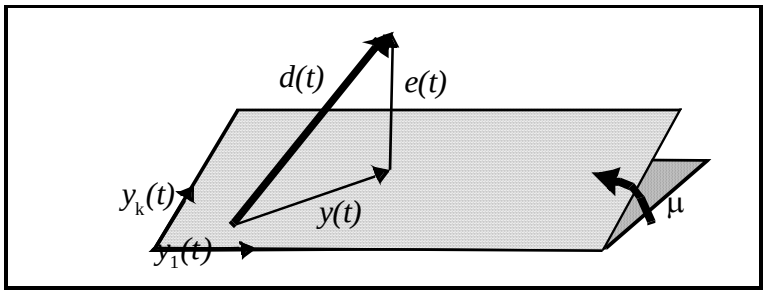



**Figure 21. Change of the relative angle between the projection space and the vector when** $\mu$ changes

So the recursive parameter in the gamma structure changes the span of the memory space with respect to the input signal (which can be visualized as a relative rotation and change in dimensionality between the input signal and the projection space). In terms of time domain analysis, the recursive parameter is finding the length of the time window (the memory depth) containing the relevant information to decrease the output mean square error. So the gamma memory modifies the position of the projection space via $\mu$ and the weights $\{w_{ik}\}$ choose in this space the point that makes the error orthogonal to the space.

The gamma PE is more powerful than the tap delay line PE, but the search for the best orientation of the projection space is not trivial. We can show that the performance surface has many local minima. Most of the times any of the local minima provide a

better solution than the linear combiner of the same order, so there is a net gain in using the gamma memory.

Return to Text

# Advantage of linear memory PEs

One of the problems of fully recurrent systems is that they tend to work as finite state machines, i.e. their PEs tend to saturate on and off during operation. If we think of Markov models, the other technology to deal with extracting information in time, this seems pretty reasonable, and in fact essential to capture the complex structure of time signals.

The problem is that recurrent systems take a long time to train, if they can be trainable at all. The difficulty has been called *the long term dependency problem*. It simply means that when we train a recurrent net with PEs that are saturated, the gradients are going to be heavily attenuated so either the relationships are poorly learned or are impossible to learn.

This is where the linear memory structures come to our rescue. The advantage of linear memories is that the gradients are not attenuated when backpropagated through them. The disadvantage is that they are unable to create nonlinear relationships (states). However, a clever intermix of memory PEs and nonlinear PEs as done in the TLFNs has been shown to provide better results than fully recurrent topologies. See Giles

Return to text

# training focused TLFNs

The goal is to be able to adapt the feedback parameter $\mu$ of a locally recurrent memory in a focused TLFN (memory at the input layer) with an algorithm that can be integrated with static backpropagation. Let us treat the case of the gamma memory. The gamma

memory forward equation for tap k is

$$y_k(n+1) = (1-\mu)y_k(n) + \mu y_{k-1}(n)$$ **Equation 43**

Let us take the derivative of y with respect to $\mu$ using the direct differentiation method

$$\frac{\partial}{\partial \mu} y_k(n+1) = (1-\mu)\frac{\partial}{\partial \mu} y_k(n) + \mu \frac{\partial}{\partial \mu} y_{k-1}(n) + y_{k-1}(n) - y_k(n)$$ **Equation 44**

So the direct differentiation method provides an on-line approximation (for small stepsizes) to adapt the $\mu$ parameter. What is necessary is to integrate this equation with the information available in a backpropagation environment.

The goal is to adapt $\mu$ using static backpropagation. Let us assume that the gamma memory PE is connected to several (p) hidden PEs of the MLP in the focused TLFN. Then we can write

$$\frac{\partial \mathrm{J}}{\partial \mu} = \sum_{\mathrm{i}} \frac{\partial \mathrm{J}}{\partial \mathrm{y_i(n)}} \frac{\partial}{\partial \mu} \mathrm{y_i(n)}$$ **Equation 45**

where $y_i(n)$ are the outputs of the gamma memory PE, the sum is extended to p and J is the output criterion. During a backpropagation sweep the errors that are propagated from the output through the dual system up to the gamma memory PE are exactly $\frac{\partial J}{\partial y_i(n)'}$. So $\mu$ can be adapted using a mix of static backpropagation and RTRL. According to Eq. 43, the dual system provides $\frac{\partial J}{\partial y_i(n)'}$ which is multiplied by Eq. 42 and then summed across the gamma PEs. Eq. 42 can be thought as a new "dual" of the gamma memory PE in the static backpropagation framework, since it computes instantaneously the sensitivities.

Notice that the implementation of Eq. 42 (the instantaneous dual gamma PE) must have

memory to store each of the previous values of the sensitivities $\frac{\partial}{\partial \mu} y_i(n)$ computed by Eq. 42. What we gain is a sample by sample update algorithm that does not require any storage since it is using static backpropagation. Since we have a single adaptive parameter, the implementation of the RLRL part (Eq. 42) is still pretty efficient. Figure 22 shows a block diagram of the steps involved in the adaptation of $\mu$. Notice that these equations are only valid when the context PE is at the input, otherwise all the gradients to the left of the gamma memory PE will become time dependent.

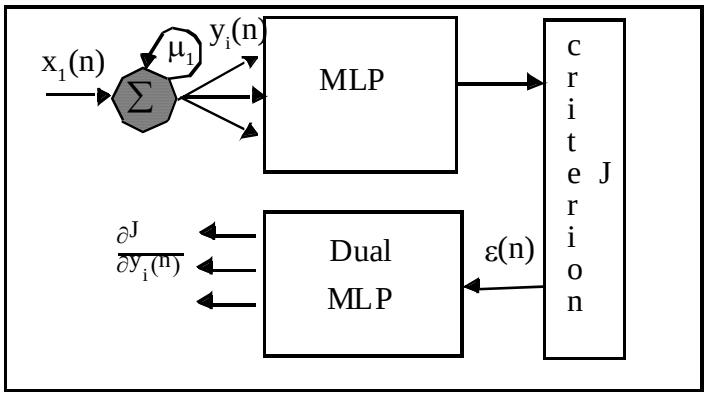



**Figure 22. How to adapt the focused architectures with a mixture backprop, RTRL.**

Return to text

# Training the gamma filter

The gamma filter was presented in Chapter X as an alternative to the Wiener filter. It is basically a gamma memory followed by an adder. Since the gamma memory is locally recurrent the gamma filter can not be trained with static backpropagation.

The gamma filter is defined as

$$y(n) = \sum_{k=0}^{K} w_k x_k(n)$$
$$x_k(n) = (1-\mu)x_k(n-1) + \mu x_{k-1}(n-1) \qquad k = 1,\ldots,K$$

**Equation 46**

where $x_0(n) = x(n)$ is the input signal and y(n) is the output. The weights and the gamma

parameter $\mu$ will be adapted using the gradients of the cost function given by Eq.4 . We can use RTRL and get immediately,

$$\Delta w_k = -\eta \frac{\partial J}{\partial w_k} = \eta \sum_{n=0}^{T} e(n) x_k(n)$$
$$\Delta \mu = -\eta \frac{\partial J}{\partial \mu} = \eta \sum_{n=0}^{T} e(n) \sum_{k=0}^{K} w_k \alpha_k(n)$$

**Equation 47**

where $\alpha_k(n) = \frac{\partial x_k(n)}{\partial \mu}$ . This gradient can be computed on-line by differentiating the gamma filter equation (Eq.46 ), yielding

$$\alpha_0(n) = 0$$
$$\alpha_k(n) = (1-\mu)\alpha_k(n-1) + \mu \alpha_{k-1}(n-1) + [x_{k-1}(n-1) - x_k(n-1)]$$

**Equation 48**

Note that using the ideas of RTRL, we can for small stepsize $\eta$ drop the summation over time in Eq. 47. Notice that these equations are local in time and the complexity to adapt a filter of order k is O(K). NeuroSolutions does not implement this procedure directly, BPTT must be used to adapt the gamma filter.

Return to text

# training alternate memories

To train TLFNs with Laguerre PEs, we need to specify two maps, the state equations, the state gradients (i.e. the backpropagated errors). Since the Laguerre PE is just a minor modification to the gamma PE, we will only present the results below.

*activation* $y_{i,k}(n) = (1-\mu_i) y_{i,k}(n-1) + y_{i,k-1}(n-1) - (1-\mu_i) y_{i,k-1}(n-1)$

*error* $\varepsilon_{i,k}(n) = (1-\mu_i) \varepsilon_{i,k}(n+1) + \varepsilon_{i,k-1}(n+1) - (1-\mu_i) \varepsilon_{i,k-1}(n+1)$

for all taps after the first (k>1). For the first tap the equations read

*activation* $y_{i,1}(n) = (1-\mu_i)y_{i,1}(n-1) + \alpha_i x_i(n-1)$

*error* $\varepsilon_{i,1}(n) = (1-\mu_i)\varepsilon_{i,1}(n+1) + \alpha_i \varepsilon_i(n+1)$

where $\alpha_i = \sqrt{1-(1-\mu_i^2)}$ and it is the implementation of the frontend lowpass filter.

The weight update using straight gradient descent is

*weight update* $\frac{\partial E}{\partial \mu_i} = \sum_k \left[ y_{i,k}(n)\varepsilon_{i,k}(n+1) - \varepsilon_{i,k}(n) y_{i,k-1}(n) \right]$

This is what is needed to train networks with a Laguerre memory PE.

To adapt the Gamma II PE in a TLFN the following two local maps are needed

$$y_{i,k}(n) = 2(1-\mu)y_{i,k}(n-1) - \left[(1-\mu)^2 + \upsilon\mu^2\right]y_{i,k}(n-2) + \mu y_{i,k-1}(n) - (1-\mu)y_{i,k-1}(n-1)$$

$$\varepsilon_{i,k}(n) = 2(1-\mu)\varepsilon_{i,k}(n+1) - \left[(1-\mu)^2 + \upsilon\mu^2\right]\varepsilon_{i,k}(n+2) + \mu\varepsilon_{i,k-1}(n) - (1-\mu)\varepsilon_{i,k-1}(n+1)$$

With these maps we can train with BPTT ANY topology that includes these memory PEs.

Return to text

# TLFN architectures

In neurocomputing it is known that a general linear delay mechanism can be represented by temporal convolutions instead of multiplicative instantaneous interactions as found in MLPs. This model has been called the *convolution model*, and in discrete time the activation of the $i^{th}$ PE reads

$$y_i(n+1) = f\left( \sum_{\substack{j=0 \\ i>j}}^{N} \sum_{k=o}^{K} w_{ijk} y_{jk}(n) \right) + I_i(n)$$

**Equation 49**

where $I_i(n)$ is a network input (if the PE is not an input $I_i(n)=0$), $y_i(n)$ is the activation at the $i^{th}$ PE at time n, N is the total number of PEs, and K is the number of states (taps) of the delay subsystem that implements the short term memory mechanism. The index k is

associated with the states of the memory filter. The activation of the $k^{th}$ tap of the $j^{th}$ PE is written $y_{jk}(n)$. The weight $w_{ijk}$ connects the $k^{th}$ tap of the $j^{th}$ PE to the $i^{th}$ PE. Since the topology is required to be feedforward, the condition i>j must be imposed.

The activation of each PE *depends explicitly* upon the past value of either the input or other states, so that the time history can be captured more easily during learning. In this sense, *the convolution model is a pre-wired additive model for the processing of time information*. One can think of the convolution model as a combination of a nonlinear zero order mapper (a nonlinear function f(.)) with a short term linear memory mechanism represented by net (.), where

$$net_{ij}(n) = \sum_{k=o}^{K} w_{ijk} x_{jk}(n) \qquad \textbf{Equation 50}$$

As written, the memory mechanism can be globally recurrent. The memories in TLFNs are a special case of Eq. 48, when $net_i(n)$ is a *locally recurrent linear filter, created by cascading delay operators*. We can recognize this arrangement as a generalized feedforward filter. The memory filter is no longer restricted to be finite impulse response (FIR) as in TDNN but it can also be an infinite impulse response (IIR) adaptive filter with local feedback. The processing function of *TLFNs can be described as a combination of nonlinear PEs and time-space mappings built from local linear projections*. The delay operators have been addressed in Chapter X.

Let us write the TLFN equations in matrix form. Define the signal vectors

$$\mathbf{x}_k = [x_{1k}, ...., x_{Nk}]^T$$
$$\mathbf{I} = [I_1, ...., I_N]^T$$

and parameter matrices $p = diag_N(p_i)$ and $W = \begin{bmatrix} w_{11k} & \dots & w_{1Nk} \\ \dots & \dots & \dots \\ w_{N1k} & \dots & w_{NNk} \end{bmatrix}$

where p is the matrix with the parameters of the generalized feedforward memory (GFM).

Here k is the index of the state variables associated with the GFM. We can eliminate k if we define the GFM state vector

$$\mathbf{X} = [\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_K]^T$$
$$\mathbf{II} = [\mathbf{I}, \mathbf{0}, \ldots., \mathbf{0}]^T$$

the nonlinearity and the matrix of decay parameters

$$F(net) = \begin{bmatrix} f(net) & 0 & 0 \\ 0 & net & 0 \\ 0 & 0 & net \end{bmatrix} \qquad P = \begin{bmatrix} p_1 & \ldots & 0 \\ .. & \ldots & \ldots \\ 0 & \ldots & p_K \end{bmatrix}$$

and the matrix of weights

$$\Omega = \begin{bmatrix} \mathbf{w}_0 & \mathbf{w} & \ldots & \mathbf{w}_k \\ p_1 & 0 & \ldots & 0 \\ 0 & \ldots & \ldots & 0 \\ 0 & \ldots & p_k & 0 \end{bmatrix}$$

Then any TLFN can be written as

$$\frac{d\mathbf{Y}}{dt} = -\mathbf{PX} + \Omega\mathbf{Y} + \mathbf{II}$$

which is a N(K+1) dimensional Grossberg model. If we analyze the structure of the matrix W we see that the matrix is not fully populated, meaning that the neural topology is prewired when compared to the most general Grossberg model.

## Gamma model Equations

Changing the memory PE will produce a different type of TLFN. The gamma model is built from combining a feedforward topology with memory PEs of the gamma type. The purpose of showing here the equations is to provide an introduction to the mathematical treatment of TLFNs from the point of view of the convolution model. At the same time, it will contrast the advantages of using the data flow implementation of backpropagation when compared to the normal equation based learning. Using the convolution model Eq.47 , we can easily obtain the dynamical equations that describe the gamma model as

$$y_i(n+1) = f\left(\sum_{\substack{j=0 \\ i>j}}^{N}\sum_{k=o}^{K} w_{ijk}\, y_{jk}(n)\right) + I_i(n)$$

$$y_{i,k}(n) = (1-\mu_i)\, y_{i,k}(n-1) + \mu_i\, y_{i,k-1}(n-1) \qquad \textbf{Equation 51}$$

The following figure represents the block diagram of the gamma model.

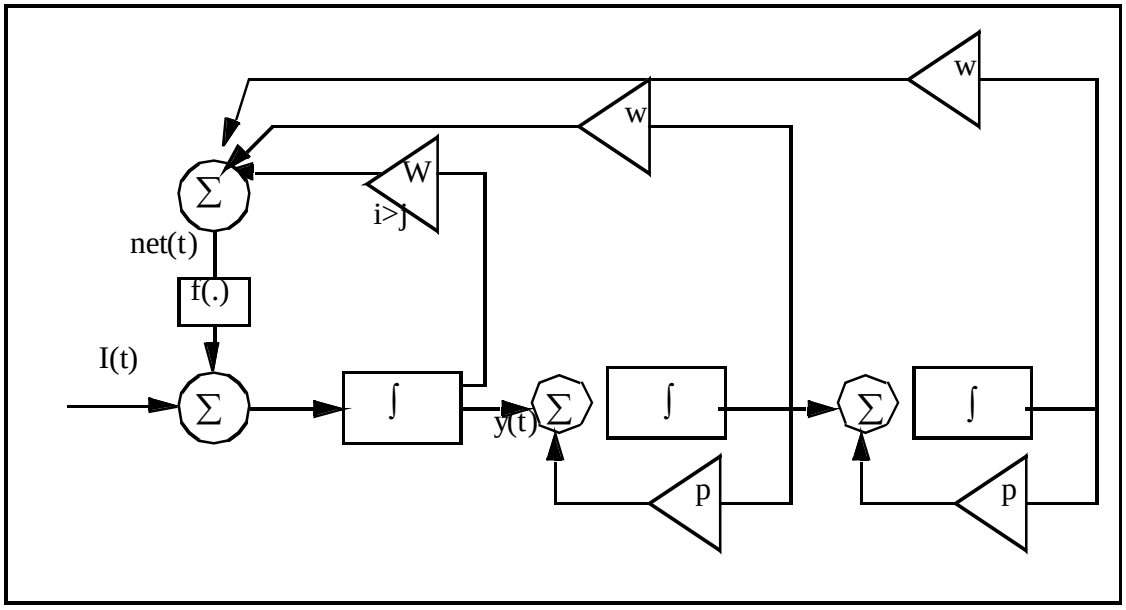



**Figure 23. The gamma neural model.**

Note that the second equation is relating the signals at the gamma PE taps, while the first equation is simply combining the temporal signals into a feedforward topology of nonlinear PEs. So the equation that is specific of the gamma memory PE is the second one.

To train the gamma model with BPTT, both the gradients with respect to the system states, and to the weights $w_{ij}$ and $\mu_i$ are necessary as given by Eq. 50. We are going to present here just the results.

$$\frac{\partial J}{\partial y_i(n)} = -e_i(n) + \mu_i \frac{\partial J}{\partial y_{i1}(n+1)} + \sum_{j>i} f_j'(net_j(n)) w_{ji} \frac{\partial J}{\partial y_j(n)}$$

$$\frac{\partial J}{\partial y_{ik}(n)} = (1-\mu_i) \frac{\partial}{\partial y_{ik}(n+1)}(J) + \mu_i \frac{\partial J}{\partial y_{i,k+1}(n+1)} + \sum_{j>i} f_j'(net_j(n)) w_{jik} \frac{\partial J}{\partial y_j(n)}$$

$$\frac{\partial J}{\partial w_{ijk}} = \sum_n f_i'(net_i(n)) x_{jk} \frac{\partial J}{\partial y_{ik}(n)}$$

$$\frac{\partial J}{\partial \mu_i} = \sum_k \sum_n \left[ x_{i,k-1}(n-1) - x_{ik}(n-1) \right] \frac{\partial J}{\partial y_{ik}(n)}$$

**Equation 52**

After analyzing these equations you can imagine the difficulty of making sure the equations are right, and then to implement training algorithms that translate without errors such procedures. But as we have been saying all along, with the data flow method we do not need these equations. We just need the dataflow through time (implemented with the dynamic controller) and local maps for the nonlinear and gamma memory PEs.

Remember that one needs to specify graphically the topology and the size of the trajectory in the dynamic controller for appropriate training.

Return to Text

# dynamic backpropagation

If we read the control literature we will see that instead of BPTT the preferred method to adapt a mixture of neural networks and dynamic systems is dynamical backpropagation. Dynamic backpropagation is a blend of RTRL with static backpropagation. The parameters of the ANN are adapted with backpropagation but the sensitivities are passed forward among sub-modules using the old idea of RTRL.

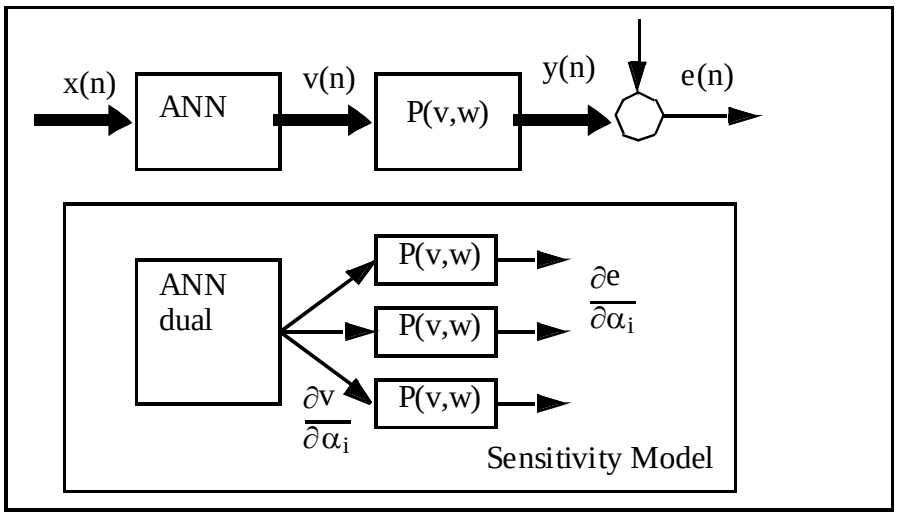
The advantage of this technique is that provides adaptation sample by sample, i.e. it is

intrinsically on-line. The disadvantage is that it is computationally demanding and it depends upon the particular arrangement of sub-systems. As we will see below there is a proliferation of blocks needed to pass sensitivity forward, basically one per parameter. While in BPTT the gradients are all local to the topology so it can be applied in the same manner irrespective of the topological configuration of the system. However BPTT is not local in time, which means that the adaptation must be done for a trajectory.

Let us present here the ideas of dynamic backpropagation as applied to the representation 1 of Figure 11. For this representation we need to find the parameter set of the ANN, here represented by $\alpha_i$, but since the output of the net v(n) is not directly accessible, one has to include the system P(v,w) in the computation of the error. Assume that the error at the output is $e(n) = d(n) - y(n)$ . Since y is the output of the cascade, we have by the chain rule

$$\frac{\partial e(n)}{\partial \alpha_i} = P(v,w)\frac{\partial v(n)}{\partial \alpha_i}$$

We know already how to compute $\frac{\partial v(n)}{\partial \alpha_i}$ for every time tick using backpropagation. But every time we change one of the parameters there is going to be a corresponding change in the output error (for the present time and for future time since P(v,w) is dynamic). This change can be computed if the sensitivity of the ANN is passed forward through a copy of P(v,w) as in the Figure




Notice that we need in general as many copies of P(v,w) as parameters in the ANN (one system will generate a partial derivative $\frac{\partial e(n)}{\partial \alpha_i}$). This is were the computational complexity gets out of hand quickly. But notice that we can update the parameters of the ANN for EACH time step which is an advantage.

If we want to work with any of the other representations this simple sensitivity model has to be appropriately modified, but it is possible in all cases to update the weights at each step.

Return to text

# derivation of energy function

For the discrete time Hopfield network considered here, it was shown that the function

$$H(y) = -\frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N} w_{ij} y_i y_j - \sum_{j=1}^{N} b_j x_j + \sum_{j=1}^{N} G(y_j)$$

$$G(y_j) = \int_0^{y_j} f(y)dy$$

is a Lyapunov function for the network provided that the weight matrix is symmetric and the nonlinearity is sufficiently steep. Notice that if the bias are zero the second term

disappears, and if the nonlinearity approaches the step function the last term approaches a constant.

The existence of a Lyapunov function for a given dynamical system guarantees stability (in the sense of Lyapunov). A Lyapunov function is a scalar function of the state with the following properties:

- is continuous and has a continuous first partial derivative over the domain,
- is strictly positive except at the equilibrium point,
- is zero at the equilibrium point
- approaches infinite at infinite
- and has a first difference that is strictly negative in the domain except the equilibrium point.

A quadratic function of the state is a Lyapunov functions for linear systems. For nonlinear systems the form of the Lyapunov function is unknown and normally is difficult to find. Hence the importance of Hopfield work.

H(y) obeys trivially all the conditions except for the last one. The proof shows that when the Hopfield network evolves according to its dynamics, H either stays the same or decreases proving the strictly negative condition. See Hopfield for the complete proof.

Return to Text

# fully recurrent

networks which have arbitrary feedback connections from input-to hidden-to output PEs.

# TLRN

have short-term memory structures anywhere in the network (i.e. either input, hidden or output PEs) but they still have a general feedfoward toplogy.

# trajectory

trajectory learning is the most general case of training a dynamical systems which specifies the desired response during a time segment.

# fixed point

is a training regime for dynamical systems where the input - desired response pair is specified for all time. This means that we are using a dynamical system as a static system (i.e. as a MLP).

# Hopfield

John Hopfield is a theoretical biologist who studied the properties of a fully recurrent neural network using the ideas of dynamics, which lead to the powerful concept of computational energy . See the paper Neural networks and physical systems with emergent collective computational abilities, Proc. Natl. Acad. Sc. USA, 79, 2554-2558, 1982.

# Eq.1

$$y(n+1) = \mu_1 y(n) + \mu_2 x(n)$$

# Eq.3

$$\frac{\partial}{\partial \mu_1} y(n+1) = y(n) \frac{\partial}{\mu_1} \mu_1 + \mu_1 \frac{\partial}{\partial \mu_1} \Big( y(n) \Big)$$

# unfolding

is a method to create from a recurrent network an equivalent feedforward network. This is possible only for a finite time span.

## Eq.6

$$y_2(n) = w_1 f\left[\mu y_1(n-1) + x(n)\right]$$

## Eq.8

$$L = \left\{\mu, w_1, y_1(0), y_2(0), y_3(1), y_4(1), \ldots, y_{2T-1}(T), y_{2T}(T)\right\}$$

## Eq.9

$$\frac{\partial J}{\partial y\ (n)} = \frac{\partial^d J}{\partial y_i(n)} + \sum_{\tau>n}\sum_{j>i} \frac{\partial J}{\partial y_j(\tau)} \frac{\partial^d}{\partial y_i(n)} y_j(\tau)$$

$$\frac{\partial J}{\partial w_{ij}} = \sum_n \sum_k \frac{\partial J}{\partial y_k(n)} \frac{\partial^d}{\partial w_{ij}} y_k(n)$$

## Eq.10

$$\frac{\partial J}{\partial y_1(n)} = 0 + \mu \frac{\partial J}{\partial y_1(n+1)} + w_1 f'\left(net_2(n)\right) \frac{\partial J}{\partial y_2(n)}$$

## Eq.12

$$\frac{\partial J}{\partial \mu} = \sum_n \frac{\partial J}{\partial y_1(n)} y_1(n-1) = \sum_n \delta_1(n) y_1(n-1)$$

## Eq.4

$$J = \sum_{n=0}^{T} J_n = \sum_n \sum_m e^2{}_m(n)$$

## Eq.15

$$\frac{\partial J}{\partial w_{ij}} = -\sum_n \sum_m \varepsilon_m(n) \frac{\partial}{\partial w_{ij}} y_m(n) \qquad i, j = 1, \ldots, N$$

## Eq.11

$$\frac{\partial J}{\partial w_1} = \sum_n \frac{\partial J}{\partial y_2(n)} f'\left(net_2(n)\right) y_1(n) = \sum_n \delta_2(n) y_1(n)$$

## Eq.24

$$y_j(n+1) = f\left( \sum_j w_{ij} x_i(n) + b_j + \left(1 - \mu\right) y_j(n) \right) \qquad j \neq i$$

## Eq.5

$$J = \sum_m e^2{}_m$$

## attractor

is the trajectory set that defines the limit point of the dynamics.

## dKfl

decoupled Kalman filtering learning and multistream learning have been recently proposed by Feldkamp and co-workers with very promising results. See

## Freeman

Walter Freeman, Tutorial on Neurobiology: from single neurons to brain chaos, Int. J. of Bifurcation and Chaos, 2, 451-482, 1992.

See also the book “Mass Activation of the Nervous System”, Academic Press, 1975.

# Luis Almeida

A learning rule for asynchronous perceptrons with feedback in a combinatorial environment, 1st IEEE Int. Conf. Neural Networks, vol 2, 609-618, 1987.

## Eq.46

$$y(n) = \sum_{k=0}^{K} w_k x_k(n)$$
$$x_k(n) = (1-\mu)x_k(n-1) + \mu x_{k-1}(n-1) \qquad k = 1,\ldots,K$$

## Eq. 48

$$\alpha_0(n) = 0$$
$$\alpha_k(n) = (1-\mu)\alpha_k(n-1) + \mu\alpha_{k-1}(n-1) + [x_{k-1}(n-1) - x_k(n-1)]$$

## Eq.22

$$y(n+1) = f\left(u(n) + wy(n)\right)$$

## Eq.25

$$net(n+1) = \gamma net(n) + u(n)$$
$$y(n+1) = f\left(net(n+1)\right)$$

## Eq.31

$$y_i(n+1) = \mathrm{sgn}\left(\sum_{j=1}^{N} w_{ij} y_j(n) - b_i\right) \qquad i = 1,\ldots N$$

## Eq.34

$$H(y) = -\frac{1}{2}\sum_i \sum_j w_{ij} y_i y_j$$

## Eq.33

$$w_{ij} = \frac{1}{N}\sum_{p=1}^{P} x^p{}_i x^p{}_j$$

## Eq.38

$$y_i(n+1) = (1-\mu) y_i(n) + f\left(\sum_j w_{ij} y_j(n) + b_i\right) + I_i(n) \qquad \begin{matrix} i = 1,\ldots N \\ j \neq i \end{matrix}$$

## Eq.45

$$net_{ij}(n) = \sum_{k=o}^{K} w_{ijk} x_{jk}(n)$$

## Lee Giles

Proc. IEEE Workshop Neural Networks for Signal Processing VII, pp34- 43, 1997, IEEE Press.

## Eq.35

$$\alpha^k{}_{ij} = \frac{\partial}{\partial net_k} f\left(net_k\right) \frac{\partial}{\partial w_{ij}} net_k = f'\left(net_k\right)\left[\delta_{ik} y_j\right]$$

# Eq.32

$$\frac{\partial J}{\partial y_i} = \frac{\partial^d J}{\partial y_i} + \sum_{i<j} \frac{\partial J}{\partial y_j} \frac{\partial^d y_j}{\partial y_i}$$

# Narendra

Narendra K., and Parthasarathy K., “Identification and control of dynamical systems”,

IEEE Trans. neural networks 1(1):4-27, 1990.

# Wan

(type popup definition text here)

# Eq.14

$$\begin{array}{ll} forward \quad map & y(n) = \mu y(n-1) + x(n) \\ backward \quad map & e(n) = \mu e(n+1) + \delta_2(n) \end{array}$$

# Eq.31

$$L = \left[ \left\{ w_{ij} \right\}, y_1, \ldots y_N \right]$$

# Bengio

time dependency

# Feldkamp

truncated backprop

# Index