12 Nonlinear Sequence Processing

Objective

In this chapter we extend the concepts from the Linear Sequence Processing chapter. The networks covered here are also dynamic. As in the previous chapter, they incorporate memory by using tapped delay lines. Some of the networks are related to linear FIR networks – they are strictly feedforward. Other networks have feedback and are related to linear IIR networks. Some of the networks in this chapter are of the input/output type and use tapped delay lines. Other networks are state space and use a single vector delay.

Theory and Examples

The architectures of nonlinear sequence processing networks are based on the linear sequence processing networks of the previous chapter. We will begin by showing how the FIR networks can be extended to nonlinear networks that have a finite memory. Then we will move on to recurrent networks, which can have either an input/output structure or a state space structure.

This chapter will also discuss how nonlinear dynamic networks are trained. This topic was covered in some depth in Chapter 14 of NND2. In this chapter we concentrate on how the backpropagation-through-time algorithm can be implemented by *unfolding* the network, which is the technique used in TensorFlow and PyTorch.

Extending Linear Models to Nonlinear Models

Focused Time Delay Neural Network

The simplest nonlinear dynamic network can be obtained by extending the linear FIR network of Figure 11.9. For the linear case, the TDL leads into a single linear layer. For the nonlinear case, we can simply replace the single linear layer with a multilayer network. This is sometimes referred to as a *focused time delay neural network* (FTDNN). The term focused refers to the fact that the TDL is focused at the input to the network. A two-layer version is shown in Figure 12.1.

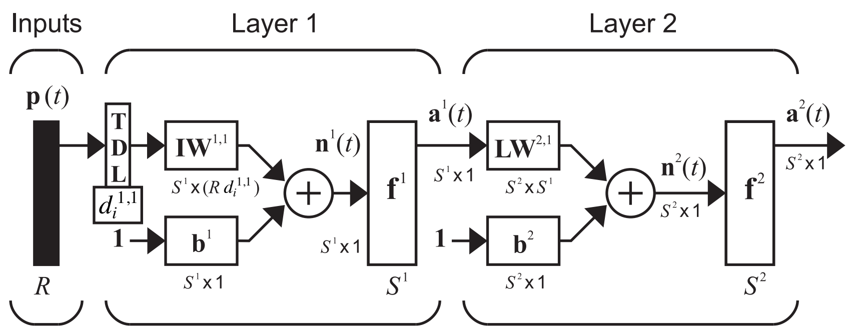


Figure 12.1: Focused Time Delay Neural Network

An advantage of this network is that its gradients can be computed with standard static backpropagation. The inputs are all available before training and can be stacked together to create the input to the weight $\mathbf{IW}^{1,1}$. Also, because this network has no feedback, computations can be done in parallel.

Distributed Time Delay Neural Network

A variation on the FTDNN is the *distributed time delay neural network* (DTDNN). (See [Waibel et al., 2013].) This is also a feedforward network, but the TDLs are distributed throughout the network. There is a TDL between each layer, as shown in Figure 12.2 for a two-layer version of the DTDNN.

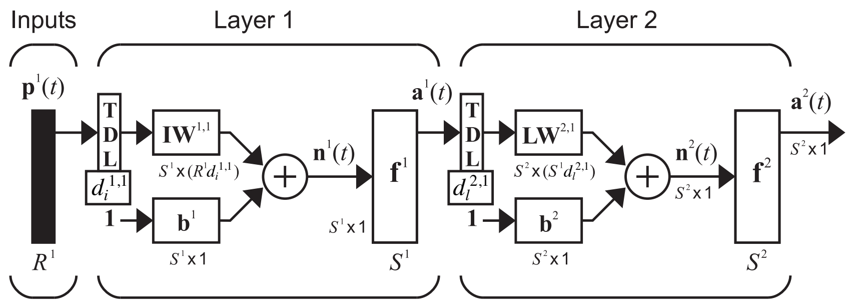


Figure 12.2: Distributed Time Delay Neural Network

This network was initially used for phoneme detection, but has also been used for other applications. It has sometimes been referred to as a 1D convolution network. As we saw in the previous chapter, the single layer FIR network of Figure 11.9 is effectively performing a convolution between the elements of the weight and the input, because the elements in the weight make up the system impulse response, as shown in Eq. 11.12.

The gradients for the DTDNN cannot be computed with the standard backpropagation algorithm. You can use the dynamic backpropagation algorithms described in Chapter 14 of NND2.

The Gamma Memory Network

The FTDNN and the DTDNN have finite memory. For the FTDNN the length of memory is the maximum delay of the TDL. For the DTDNN, it is the sum of the maximum delays of the TDLs in each layer. To have infinite memory, we need feedback. One network that introduced feedback in a simple way was the *focused gamma memory network* (FGMN). (See [Hu and Hwang, 2018].) To build the FGNN, we need to introduce the gamma element, as shown in Figure 12.3. We can arrange the gamma elements into a tapped gamma memory line (TGML), similar to the TDL. This is shown in Figure 12.4. For the FGMN we can put the TGML at the input of a multilayer network, as shown in Figure 12.5.

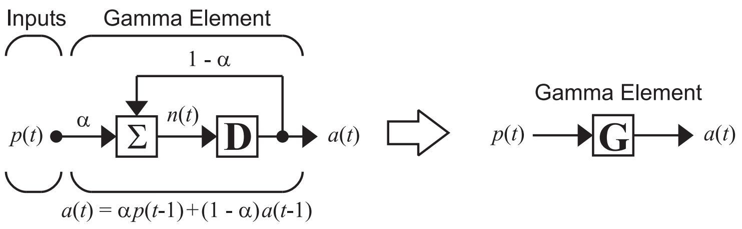


Figure 12.3: Gamma Element

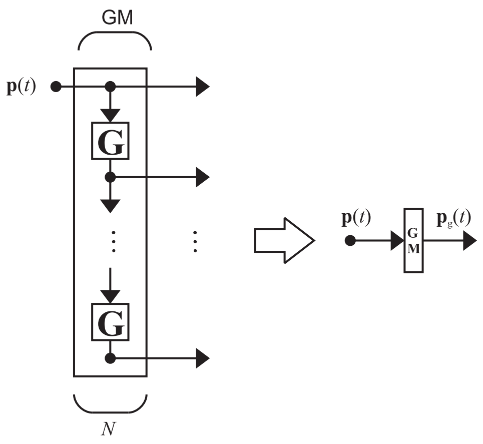


Figure 12.4: Tapped Gamma Memory Line

The FGMN uses the simplest type of recurrent connections, which allows the network to have infinite memory. The effective memory is adjusted during training by the parameter α in the gamma element:

$$a(t) = \alpha p(t-1) = (1-\alpha)a(t-1) \quad (12.1)$$

The gamma element is a low pass filter, or what is sometimes called a *leaky integrator*. Each single gamma element is an IIR system, with impulse response

$$h(t) = \alpha(1-\alpha)^{t-1}, t > 0 \quad (12.2)$$

When $\alpha = 1$ the gamma acts as a simple delay of one time step. As α decreases, the effective memory length increases. During training, the network adjusts its memory length to fit the problem. This network is typically used for adaptive filtering.

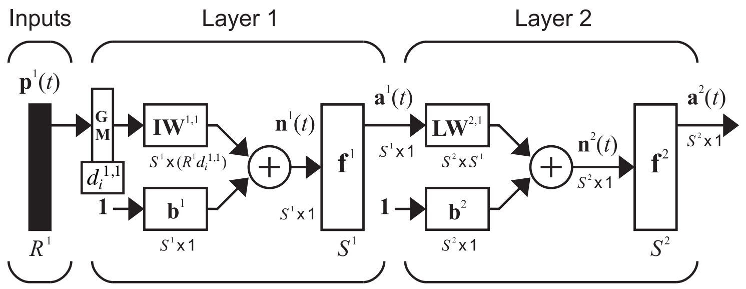


Figure 12.5: Focused Gamma Memory Network

Nonlinear Autoregressive with Exogneous Inputs (NARX)

There is a nonlinear recurrent network that descends directly from the IIR input/output network of Figure 11.13. It is called the nonlinear autoregressive with exogenous inputs, or *NARX* network, which is shown in Figure 12.6.

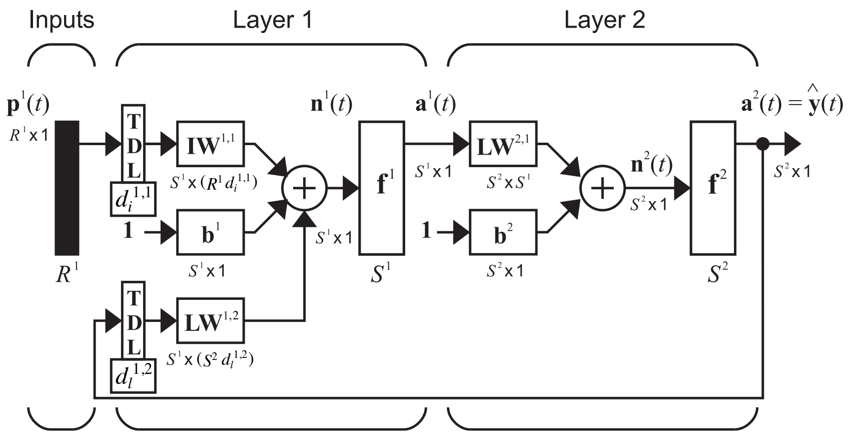


Figure 12.6: NARX Network

The term *autoregressive* comes from the fact that $a(t)$ is regressed on previous values of the same series, as shown in the difference equation in Eq. 12.3. The term *exogenous* refers to the external inputs $p(t)$ in the equation.

$$\begin{aligned} a(t) =& f(a(t-1), a(t-2), \cdots, a(t-n_a), \\ & p(t-1), p(t-2), \cdots, p(t-n_p)) \end{aligned} \tag{12.3}$$

Although the NARX network is recurrent (has feedback connections), it is possible to compute its gradient with the standard backpropagation algorithm by using a trick, which is illustrated in Figure 12.7. If we are training the network output $\mathbf{a}(t)$ to match a target prediction $\mathbf{y}(t)$, then we can think of the network output as the estimate of $\mathbf{y}(t)$, which we could label $\hat{\mathbf{y}}(t)$. After the network has been trained, $\hat{\mathbf{y}}(t)$ should be a close match to $\mathbf{y}(t)$. This gives us the idea to replace $\hat{\mathbf{y}}(t)$ with $\mathbf{y}(t)$ in the feedback TDL. This effectively opens the feedback loop, and we can train it in the same was as we can train the FTDDN – as a static network. The NARX network with feedback has been called a *parallel* architecture, and the NARX network with the feedback opened has been called a *series-parallel* architecture, as illustrated in Figure 12.7.

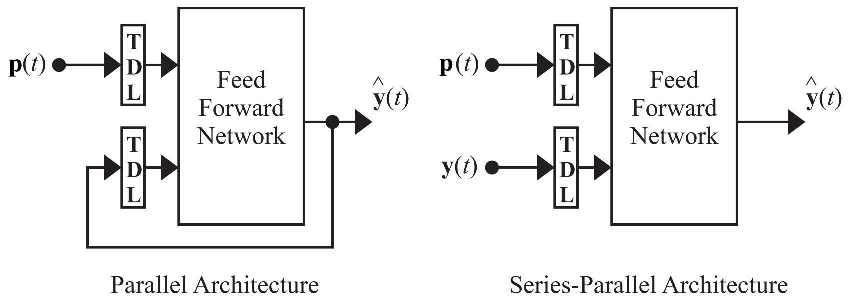


Figure 12.7: Parallel and Series-Parallel Architectures

The two main applications for NARX networks are system control/identification (modeling dynamic systems) [Jafari and Hagan, 2018] and multi-step prediction [Kelley and Hagan, 2024].

Recurrent Neural Network (RNN)

The term *recurrent* generally refers to a feedback operation. Recurrent connections can occur in a variety of ways. For example, the NARX network in Figure 12.6 has a recurrent connection. A more complex example of recurrent connections is shown in Figure 7.4. Historically, however, many have tended to give the name *Recurrent Neural Network* (RNN) to a network with one feedback connection with a single delay, as shown in Figure 12.8. (Sometimes the output layer is not included.) Equations 12.4 and 12.5 define the network operation.

$$\mathbf{a}^1(t) = \mathbf{tansig}\left(\mathbf{IW}^{1,1}\mathbf{p}(t) + \mathbf{LW}^{1,1}\mathbf{a}^1(t-1) + \mathbf{b}^1\right) \tag{12.4}$$

$$\mathbf{a}^2(t) = \mathbf{LW}^{2,1}\mathbf{a}^1(t) + \mathbf{b}^2 \tag{12.5}$$

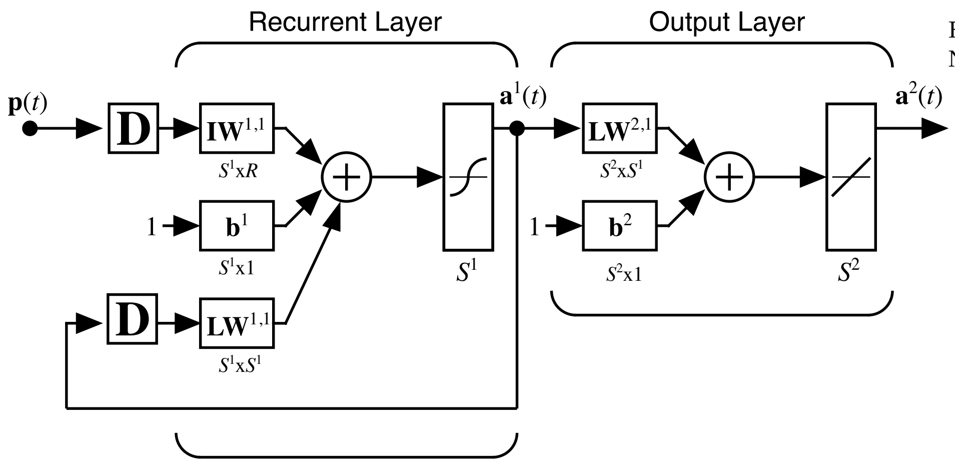


Figure 12.8: Recurrent Neural Network

If we compare this RNN with Figure 11.15, we can see that it is just a nonlinear version of the linear state space model. In fact, most of the networks we have seen in this section are nonlinear versions of the linear networks in Chapter 11. The FTDNN is a nonlinear version of the FIR network of Figure 11.9. The NARX network is a nonlinear version of the IIR network of Figure 11.13.

Dynamic Backpropagation

Chapter 14 of NND2 covers in detail how gradients can be computed for dynamic networks. In this section we focus on the method used in TensorFlow and PyTorch, which is a particular implementation of the *backpropagation-through-time* algorithm. To illustrate the algorithm, we will use the basic RNN, without the output layer, as shown in Figure 12.9.

As discussed in Chapter 14 of NND2, there are two basic approaches to computing the gradients for dynamic networks. One, which is shown in Eqs. 12.6 and 12.7, is called *real time recurrent learning* RTRL.

$$\frac{\partial F}{\partial \mathbf{x}} = \sum_{t=1}^{Q} \left[\frac{\partial \mathbf{a}(t)}{\partial \mathbf{x}^T}\right]^T \times \frac{\partial^e F}{\partial \mathbf{a}(t)} \tag{12.6}$$

$$\frac{\partial \mathbf{a}(t)}{\partial \mathbf{x}^T} = \frac{\partial^e \mathbf{a}(t)}{\partial \mathbf{x}^T} + \frac{\partial^e \mathbf{a}(t)}{\partial \mathbf{a}(t-1)^T}\frac{\partial \mathbf{a}(t-1)}{\partial \mathbf{x}^T} \tag{12.7}$$

This is RTRL for the network in Figure 12.9. The RTRL algorithm for arbitrary LDDNs is given on page 14-17 of NND2.

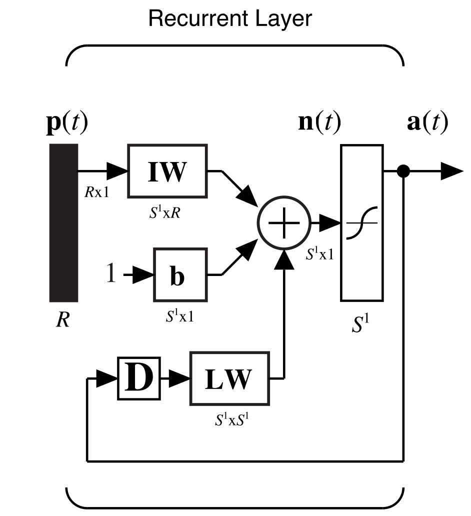


Figure 12.9: Basic Recurrent Neural Network

Both of these equations represent implementations of the chain rule. The term **x** represents all of the weights and biases in the network. The superscript *e* represents an explicit derivative, not accounting for indirect effects through time. The explicit derivative in Eq. 12.6 is the explicit derivative of the loss function with respect to the network output $\mathbf{a}(t)$. If $\mathbf{a}(t)$ does not appear explicitly in the loss function, then this derivative will be zero.

The other derivative in Eq. 12.6 is the complete derivative of the network output at time t with respect to the network weights and biases. This derivative must account for the indirect effect, for example, that **LW** would have on $\mathbf{a}(3)$ because **LW** directly affects $\mathbf{a}(2)$, which then affects $\mathbf{a}(3)$ through **LW** at the next time step. This derivative must be calculated recursively, as shown in Eq. 12.7. For RTRL this recursion moves forward in time.

The other general approach to gradient calculation for dynamic networks is called *backpropagation through time* (BPTT). The equations for this method are given in Eqs. 12.8 and 12.9.

$$\frac{\partial F}{\partial \mathbf{x}} = \sum_{t=1}^{Q} \left[\frac{\partial^e \mathbf{a}(t)}{\partial \mathbf{x}^T} \right]^T \times \frac{\partial F}{\partial \mathbf{a}(t)} \tag{12.8}$$

$$\frac{\partial F}{\partial \mathbf{a}(t)} = \frac{\partial^e F}{\partial \mathbf{a}(t)} + \frac{\partial^e \mathbf{a}(t+1)}{\partial \mathbf{a}(t)^T} \times \frac{\partial F}{\partial \mathbf{a}(t+1)} \tag{12.9}$$

This is BPTT for the network in Figure 12.9. The BPTT algorithm for arbitrary LDDNs is given on page 14-26 of NND2.

Note that there are two main differences between BPTT and RTRL. First, in RTRL the recurrence equation moves forward in time, whereas it moves backward in time for BPTT. Second, in RTRL the complete derivative that is updated recursively is $\partial \mathbf{a}(t)/\partial \mathbf{x}^T$, whereas in BPTT $\partial F/\partial \mathbf{a}(t)$ is updated.

In Eq. 12.9, the first term represents the direct effect that $\mathbf{a}(t)$ has on $F(\mathbf{x})$. The second term represents the indirect effect. This is caused because $\mathbf{a}(t)$ contributes to $\mathbf{a}(t+1)$.

It is easiest to follow the steps of BPTT by working through a few steps of the algorithm. Let's set the number of time points to $Q = 2$, and let the loss function be the sum squared error at the last time point:

$$F(\mathbf{x}) = (\mathbf{t} - \mathbf{a}(2))^T(\mathbf{t} - \mathbf{a}(2)) = \mathbf{e}^T\mathbf{e} \qquad (12.10)$$

We can't begin the BPTT algorithm before computing the network output for all time points.

We begin with Eq. 12.9, in which we recursively update $\partial F/\partial \mathbf{a}(t)$. This needs to be initialized at the last time point. Since the loss function doesn't contain any terms beyond $t = 2$, we can set derivatives beyond that point to zero.

$$\frac{\partial F}{\partial \mathbf{a}(t)} = \mathbf{0},\ t > 2 \qquad (12.11)$$

Now iterate Eq. 12.9 from $t = 3$ to $t = 2$ and then to $t = 1$.

$$\frac{\partial F}{\partial \mathbf{a}(2)} = \cancelto{-2\mathbf{e}}{\frac{\partial^e F}{\partial \mathbf{a}(2)}} + \frac{\partial^e \mathbf{a}(3)}{\partial \mathbf{a}(2)^T} \times \cancelto{0}{\frac{\partial F}{\partial \mathbf{a}(3)}} = -2\mathbf{e} \qquad (12.12)$$

$$\frac{\partial F}{\partial \mathbf{a}(1)} = \cancelto{0}{\frac{\partial^e F}{\partial \mathbf{a}(1)}} + \frac{\partial^e \mathbf{a}(2)}{\partial \mathbf{a}(1)^T} \times \cancelto{-2\mathbf{e}}{\frac{\partial F}{\partial \mathbf{a}(2)}} = [\mathbf{LW}]^T\, \dot{\mathbf{F}}(\mathbf{n}(2))^T\,[-2\mathbf{e}] \qquad (12.13)$$

The first term in Eq. 12.12 is the explicit derivative of the loss with respect to $\mathbf{a}(2)$. Since $\mathbf{a}(2)$ appears explicitly in $F(\mathbf{x})$, this term is not zero. Also, since $\mathbf{a}(t)$ for $t > 2$ does not appear in the loss, $\mathbf{a}(2)$ cannot have any indirect effects on $F(\mathbf{x})$. Therefore the explicit derivative is the same as the total derivative.

For Eq. 12.13 the explicit derivative of $F(\mathbf{x})$ is zero, since $\mathbf{a}(1)$ does not appear explicitly in $F(\mathbf{x})$. For the explicit derivative of $\mathbf{a}(2)$ with respect to $\mathbf{a}(1)$, this just involves standard static backpropagation. (Compare with Eq. 3.49.)

Next we need to implement Eq. 12.8, which computes the derivative of the loss with respect to the weights. Let's consider the particular weight $lw_{i,j}$.

$$\frac{\partial F}{\partial lw_{i,j}} = \sum_{t=1}^{2} \left[\frac{\partial^e \mathbf{a}(t)}{\partial lw_{i,j}}\right]^T \times \frac{\partial F}{\partial \mathbf{a}(t)} \tag{12.14}$$

Consider one term in this sum.

$$\frac{\partial^e \mathbf{a}^T(t)}{\partial lw_{i,j}} \frac{\partial F}{\partial \mathbf{a}(t)} = \frac{\partial^e \mathbf{n}^T(t)}{\partial lw_{i,j}} \frac{\partial^e \mathbf{a}^T(t)}{\partial \mathbf{n}(t)} \frac{\partial F}{\partial \mathbf{a}(t)} \tag{12.15}$$

The first term on the right hand side of Eq. 12.15 can be written

$$\frac{\partial^e \mathbf{n}(t)}{\partial lw_{i,j}} = \boldsymbol{\epsilon}_i a_j(t-1) \tag{12.16}$$

since $lw_{i,j}$ connects $a_j(t-1)$ to $n_i(t)$.

The second term on the right hand side of Eq. 12.15 is the derivative of the activation function.

$$\frac{\partial^e \mathbf{a}^T(t)}{\partial \mathbf{n}(t)} = \dot{\mathbf{F}}^T(\mathbf{n}(t)) \tag{12.17}$$

Substituting these two terms back into Eq. 12.15 we have

$$\frac{\partial^e \mathbf{a}^T(t)}{\partial lw_{i,j}} \frac{\partial F}{\partial \mathbf{a}(t)} = a_j(t-1)\dot{f}(n_i(t))\frac{\partial F}{\partial a_i(t)} \tag{12.18}$$

This is the gradient for one element of the weight. We can represent the gradient for the full weight matrix and substitute into Eq. 12.14 as follows.

$$\frac{\partial F}{\partial \mathbf{LW}} = \sum_{t=1}^{2} \dot{\mathbf{F}}^T(\mathbf{n}(t))\frac{\partial F}{\partial \mathbf{a}(t)}\mathbf{a}^T(t-1) \tag{12.19}$$

$$= \dot{\mathbf{F}}^T(\mathbf{n}(1))\frac{\partial F}{\partial \mathbf{a}(1)}\mathbf{a}^T(0) + \dot{\mathbf{F}}^T(\mathbf{n}(2))\frac{\partial F}{\partial \mathbf{a}(2)}\mathbf{a}^T(1) \tag{12.20}$$

Now we can substitute the previous terms that we calculated in Eqs. 12.12 and 12.13.

$$\begin{aligned}\frac{\partial F}{\partial \mathbf{LW}} =& \dot{\mathbf{F}}^T(\mathbf{n}(1))\mathbf{LW}^T\dot{\mathbf{F}}^T(\mathbf{n}(2))\left[-2\mathbf{e}\right]\mathbf{a}^T(0) \\ &+ \dot{\mathbf{F}}^T(\mathbf{n}(2))\left[-2\mathbf{e}\right]\mathbf{a}^T(1)\end{aligned} \tag{12.21}$$

For certain types of architectures, it is possible to implement the BPTT algorithm in a different way, using standard static backpropagation. This can be done by *unrolling* the network in time. Consider again Figure 12.9. Notice that the output of the network at time step t becomes the input to the network at time step $t+1$. Figure 12.10 illustrates this idea.

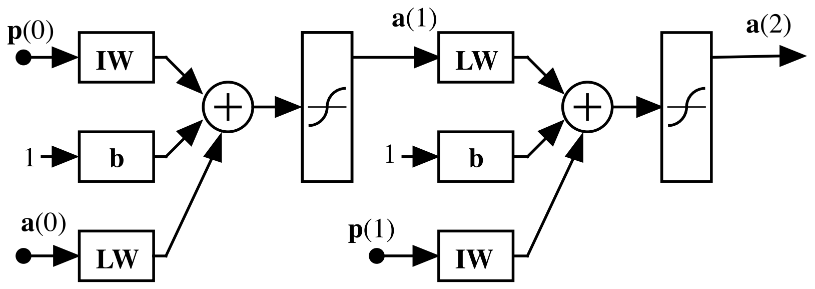


Figure 12.10: Unrolled Network

This is a static network, but the final output for this network will be the same as for the original recurrent network. We can therefore use static backpropagation to compute the gradient. It is true that the same weights and biases appear multiple times in the unrolled network, but we can just add together the gradients from each appearance.

By unrolling, we can see that any recurrent network, no matter how many layers, can always be a *deep* network.

First, recall the static backpropagation equations from Eqs. 3.39, 3.49 and 3.53.

$$\mathbf{s}^M = \dot{\mathbf{F}}^{\mathbf{M}}\left(\mathbf{n}^M\right)^T[-2\mathbf{e}] \tag{12.22}$$

$$\mathbf{s}^m = \dot{\mathbf{F}}^m(\mathbf{n}^m)^T(\mathbf{W}^{m+1})^T\mathbf{s}^{m+1} \tag{12.23}$$

$$\frac{\partial \hat{F}(\mathbf{x})}{\partial w^m_{i,j}} = s^m_i a^{m-1}_j \tag{12.24}$$

$$\frac{\partial F(\mathbf{x})}{\partial \mathbf{W}^m} = \mathbf{s}^m\left(\mathbf{a}^{m-1}\right)^T \tag{12.25}$$

Begin by initializing the sensitivity at the last time step.

$$\mathbf{s}(2) = \dot{\mathbf{F}}\left(\mathbf{n}(2)\right)^T[-2\mathbf{e}] \tag{12.26}$$

Now backpropagate one step.

$$\mathbf{s}(1) = \dot{\mathbf{F}}(\mathbf{n}(1))^T\mathbf{LW}^T\mathbf{s}(2) \tag{12.27}$$

$$= \dot{\mathbf{F}}(\mathbf{n}(1))^T\mathbf{LW}^T\dot{\mathbf{F}}\left(\mathbf{n}(2)\right)^T[-2\mathbf{e}] \tag{12.28}$$

Now we compute the gradient of the weight for each time step, according to Eq. 12.25.

$$\left.\frac{\partial F(\mathbf{x})}{\partial \mathbf{LW}}\right|_{t=2} = \mathbf{s}(2)\mathbf{a}(1)^T = \dot{\mathbf{F}}\left(\mathbf{n}(2)\right)^T[-2\mathbf{e}]\,\mathbf{a}(1)^T \tag{12.29}$$

$$\begin{aligned}\left.\frac{\partial F(\mathbf{x})}{\partial \mathbf{LW}}\right|_{t=1} &= \mathbf{s}(1)\mathbf{a}(0)^T \\ &= \dot{\mathbf{F}}(\mathbf{n}(1))^T\mathbf{LW}^T\dot{\mathbf{F}}\left(\mathbf{n}(2)\right)^T[-2\mathbf{e}]\,\mathbf{a}(0)^T\end{aligned} \tag{12.30}$$

Since the weights at the two time steps are actually the same, we add the gradients from each to get the total.

$$\begin{aligned}\left.\frac{\partial F(\mathbf{x})}{\partial \mathbf{LW}}\right|_{total} &= \dot{\mathbf{F}}(\mathbf{n}(1))^T\mathbf{LW}^T\dot{\mathbf{F}}\left(\mathbf{n}(2)\right)^T[-2\mathbf{e}]\,\mathbf{a}(0)^T \\ &+ \dot{\mathbf{F}}\left(\mathbf{n}(2)\right)^T[-2\mathbf{e}]\,\mathbf{a}(1)^T\end{aligned} \tag{12.31}$$

Comparing Eq. 12.31 with Eq. 12.21 we can see that we get the same answer by unrolling the network as we did from the earlier BPTT algorithm. This unrolling technique is currently used in TensorFlow and PyTorch, and it is especially suited to recurrent networks that have a state space form. It is not as well suited to input/output type networks which include TDLs rather than single delays. This is addressed in the Solved Problems and Exercises.

Learning Long Term Dependencies

There are some sequence processing problems where we would like to predict responses that are significantly delayed from the corresponding stimulus. For example, a chess game can be lost 12 moves after the critical mistake. When performing a translation, the words in a previous paragraph might provide important context.

Our network structure must enable the possibility of connecting items that are far removed in time. This means that the network must have long term memory. We have seen in the previous chapter that linear recurrent networks will have long term memory only if the system poles are close to the unit circle. This means that long memory systems can flirt with instability.

As an example, consider the linear recurrent network in Figure 12.11, and let $lw_{1,1} = 0.5$ and $iw_{1,1} = 1$. The impulse response for this network is

$$h(t) = 0.5^{(t-1)},\ t > 0 \tag{12.32}$$

as shown in Figure 12.12.

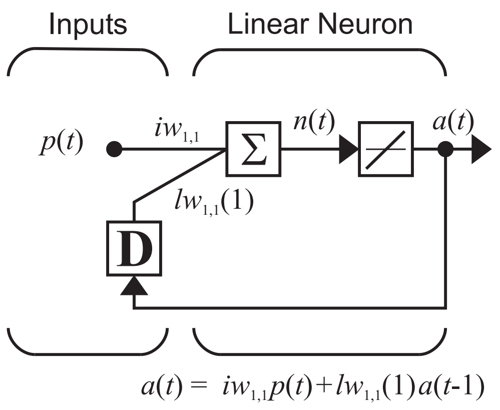


Figure 12.11: Linear Recurrent Network

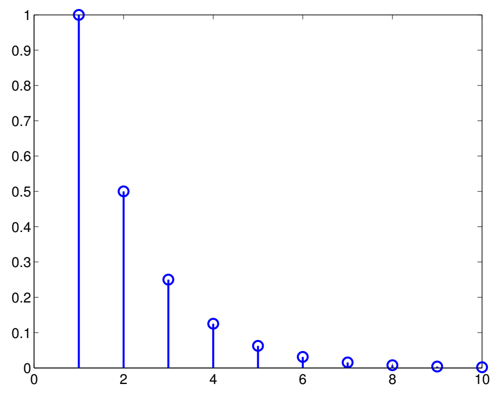


Figure 12.12: Impulse Reponse

As we saw in the last chapter, this system has a pole at 0.5. This produced the $0.5^{(t-1)}$ impulse response. When the poles are all inside the unit circle, the network is stable, and the impulse response will decay with time. In addition, the gradients for this network will also decay with time. Using Eq. 12.9, the update in the BPTT algorithm is

$$\frac{\partial F}{\partial a(t)} = \frac{\partial^e F}{\partial a(t)} + lw_{1,1}\frac{\partial F}{\partial a(t+1)} \tag{12.33}$$

If the error only occurs at the 10th time point, we have

$$\frac{\partial F}{\partial a(t)} = 0.5^{(10-t)}\frac{\partial^e F}{\partial a(10)} \tag{12.34}$$

Therefore, if the network is stable, the derivatives will decay as you go back in time. Because the derivatives are small, it will take a long time to learn long term dependencies.

This effect is enhanced when using a saturating nonlinear activation function. Consider the network in Figure 12.13. The BPTT update would now be

$$\frac{\partial F}{\partial a(t)} = \frac{\partial^e F}{\partial a(t)} + \dot{f}(n(t))lw_{1,1}\frac{\partial F}{\partial a(t+1)}, \left|\dot{f}(n(t))\right| \leq 1 \qquad (12.35)$$

where the derivative of the activation function will be less than one and will cause the gradient to decay even more quickly.

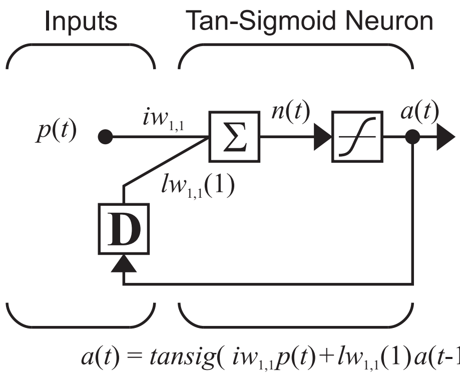


Figure 12.13: Nonlinear Recurrent Network

It is possible to have a recurrent network that can remember forever. For example, consider the parity network in Figure 12.14, with the following weights

To experiment with parity network, use the Deep Learning Demonstration Parity Network (**`dl12par`**).

$$\mathbf{IW}^{1,1} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}, \mathbf{LW}^{1,1} = \begin{bmatrix} 1 & -1 \\ 1 & -1 \end{bmatrix},$$

$$\mathbf{LW}^{2,1} = \begin{bmatrix} 1 & -1 \end{bmatrix}, \mathbf{b}^1 = \begin{bmatrix} -0.5 \\ -1.5 \end{bmatrix}, \mathbf{b}^2 = \begin{bmatrix} -0.5 \end{bmatrix} \qquad (12.36)$$

This network computes the parity of a sequence of bits. It can keep track of the parity, no matter how many bits have been presented.

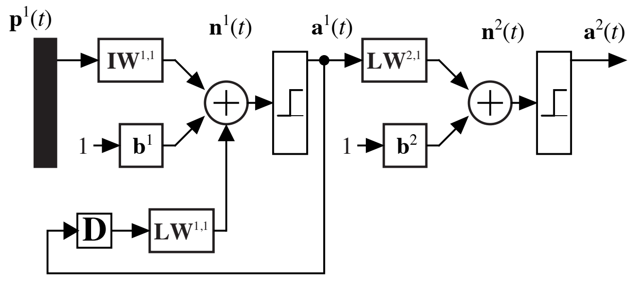


Figure 12.14: Parity Network

The Long Short Term Memory (LSTM) Network

Long term memories are network weights, and short term memories are layer outputs. We need a network that has long short term memories. In recurrent networks, as the weights change during training, the length of the short term memory will change.

If the initial weights produce a network without long short term memory, it will be difficult to increase it. This is because the gradient will become small, as shown in the previous section. On the other hand, if the initial weights produce long short term memory (poles close to the unit circle in the linear case), then instabilities can occur.

The *Long Short Term Memory* (LSTM) was designed to achieve long short term memory while maintaining stability. LSTM is a recurrent network with some special components, which we describe next. [Hochreiter and Schmidhuber, 1997]

Constant Error Carousel (CEC)

Consider the recurrent network in Figure 12.15. To maintain a long memory, we would like the feedback matrix $\mathbf{LW}^{1,1}$ to have some eigenvalues very close to one, but if they become larger than one, the system will become unstable. These eigenvalues need to stay close to one during training, or the gradients will vanish. Also, we want the derivative of the activation function to remain constant. A solution would be to set $\mathbf{LW}^{1,1} = \mathbf{I}$ and to use a linear transfer function. The result is called a *constant error carousel* (CEC).

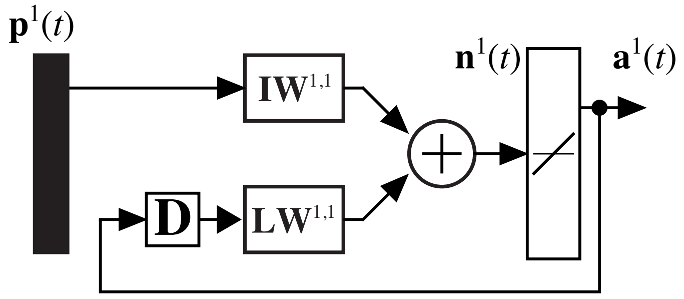
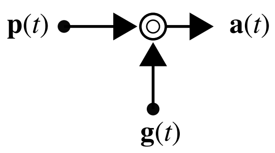


Figure 12.15: Circular Error Carousel

Gating

The problem with using the CEC is that we don't want to indiscriminately remember everything. To remember selectively we introduce several *gates* (switches). In the gate, an input signal $\mathbf{p}(t)$ is multiplied (Hadamard product) by a gating signal $\mathbf{g}(t)$ that has values between zero and one. The gate turns the input signal on or off.



$$\mathbf{a}(t) = \mathbf{g}(t) \circ \mathbf{p}(t) \tag{12.37}$$

Figure 12.16 shows a CEC with gating signals in the input and feedback paths. The gates are turned off when we want to forget and are turned on when we want to remember.

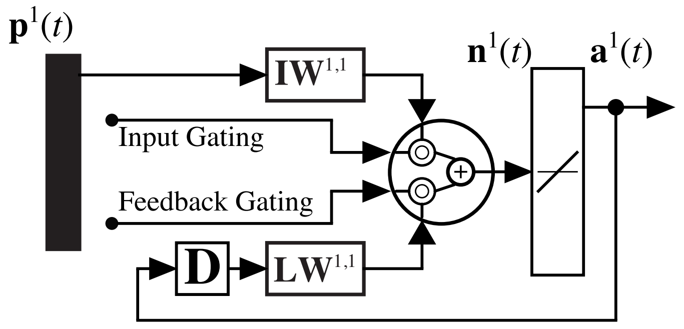


Figure 12.16: Gated CEC

To experiment with gating operations, use the Deep Learning Demonstration CEC Gating (**`dl12gat`**).

Full LSTM

In all, we will have three gates: 1) the input gate allows selective inputs into the CEC, 2) the feedback (or forget) gate will clear the CEC, and 3) an output gate will allow selective outputs from the CEC. Each gate will be a layer with inputs from the gated outputs and the network input. The result is called Long Short Term Memory (LSTM) and is shown in Figure 12.17.

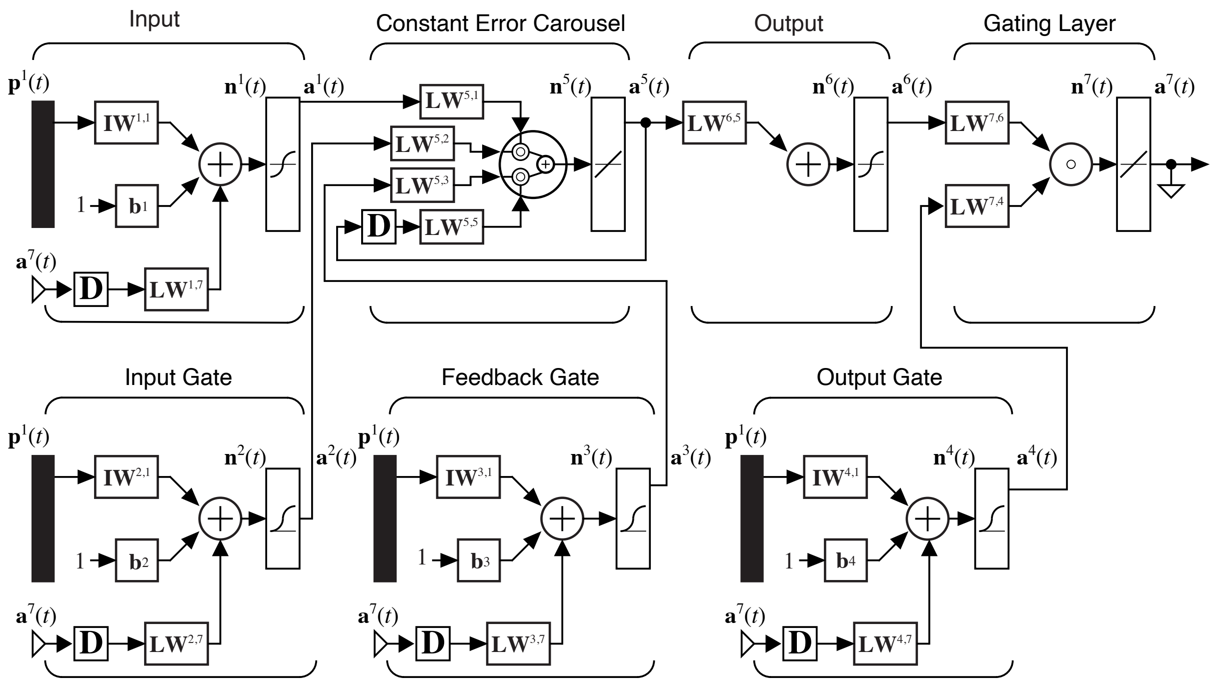


Figure 12.17: Complete LSTM

Although the network looks complex, if we step back, we can see the structure of a basic RNN, as shown in Figure 12.9. The input $\mathbf{p}^1(t)$ is multiplied by the weight $\mathbf{IW}^{1,1}$, and the output $\mathbf{a}^7(t)$ (compare with $\mathbf{a}^1(t)$ in Figure 12.9) is multiplied by the weight $\mathbf{LW}^{1,7}$ (compare with $\mathbf{LW}^{1,1}$ in Figure 12.9). The CEC, with its gating mechanisms is inserted between the first layer and the output of the basic RNN. This enables the network to have long short term memory.

The weights in the CEC are all fixed to the identity matrix, and they are not trained. The output and gating layer weights are also fixed to the identity matrix. It has been shown that the best results are obtained by initializing the feedback (forget) gate bias, $\mathbf{b}^3$, to all ones or larger values. This turns the gate on for the beginning of training. Other weights and biases are set to small random values.

The output of the gating layer generally connects to another layer or a multilayer network with a softmax activation function. Also, multiple LSTMs can be cascaded together.

The LSTM is trained with standard gradient-based algorithms, as with other deep networks. The original LSTM paper used RTRL for computing the gradient, although the currrent frameworks use BPTT. An approximate gradient is sometimes used, in which the derivatives are only propagated in time through the CEC delays. Only static derivatives are computed for the remaining terms.

Variations on the LSTM

There have been a number of variations on the LSTM. The original LSTM did not have the feedback gate, but it is commonly used now. Some variations feedback the CEC output $\mathbf{a}^5(t)$ to the gate layers. These are called *peephole* connections. [Jozefowicz et al., 2015] provides an experimental comparison of many of the variations.

A popular variation is the *Gated Recurrent Unit* (GRU). Figure 12.18 illustrates this network. The differencing layer subtracts the delayed network output $\mathbf{a}^6(t)$ from the processed input $\mathbf{a}^1(t)$ – the weight $\mathbf{LW}^{5,1}$ is set to $\mathbf{I}$, and the weight $\mathbf{LW}^{5,6}$ is set to $-\mathbf{I}$. They are not trained. The gating layer weights are also set to the identity matrix.

The delayed network output is gated by the reset gate, before it is fed into the input layer. This resets the memory. The overall output can be considered a weighted average of a candidate potential output $\mathbf{a}^1(t)$ and the previous output $\mathbf{a}^6(t-1)$:

$$\begin{aligned}\mathbf{a}^6(t) &= \mathbf{a}^4(t) \times [\mathbf{a}^1(t) - \mathbf{a}^6(t-1)] + \mathbf{a}^6(t-1) \\ &= \mathbf{a}^4(t) \times \mathbf{a}^1(t) + [1 - \mathbf{a}^4(t)] \times \mathbf{a}^6(t-1)\end{aligned}$$

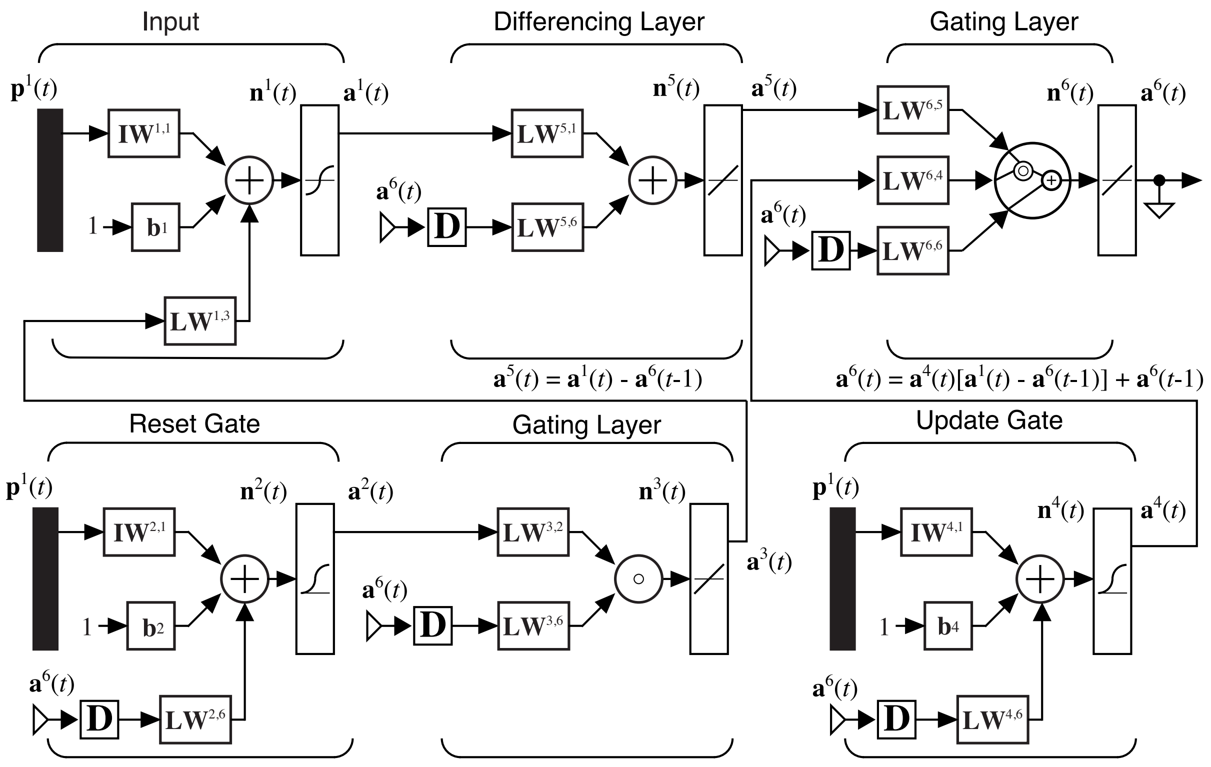


Figure 12.18: GRU Network

Epilogue

This chapter has shown how the linear sequence processing networks of the previous chapter can be extended for nonlinear processing. There are networks that directly descend from linear FIR systems, like the Focused Time Delay neural network. Other networks are related to the linear IIR input-output systems, like the NARX network. Finally, the linear state space models have lead to the RNN, LSTM and GRU.

The next chapter also considers nonlinear sequence processing. It begins with the sequence-to-sequence model, which leads to the concept of attention and then the transformer network, on which most current large language models are based.

Further Reading

[Hochreiter and Schmidhuber, 1997] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997

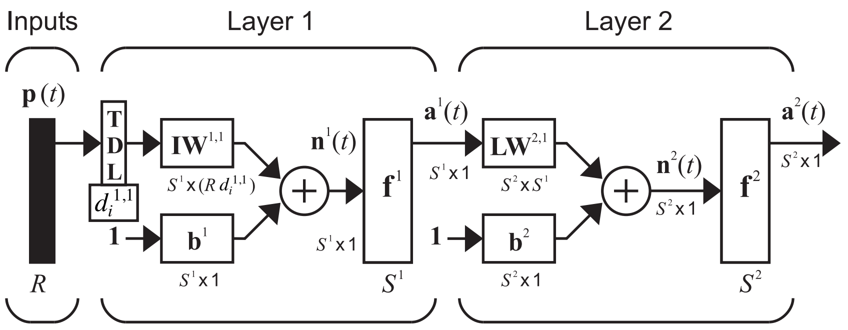
This is the paper that introduced the LSTM network. Although it was written in the 1990s, it found major application after 2015.

[Jozefowicz et al., 2015] Rafal Jozefowicz, Wojciech Zaremba, and Ilya Sutskever. An empirical exploration of recurrent network architectures. In *International conference on machine learning*, pages 2342–2350. PMLR, 2015

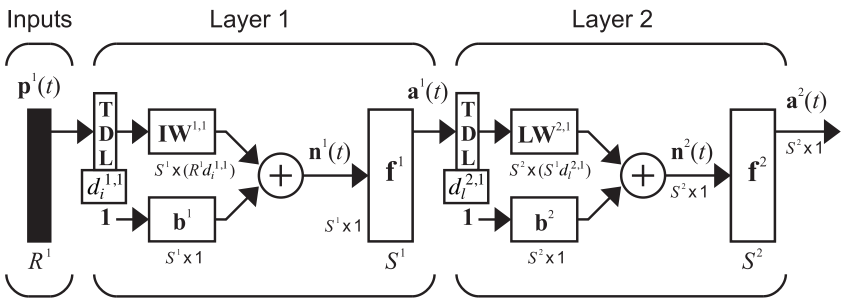
This paper tests a variety of LSTM-like networks across a number of experiments.

Summary of Results

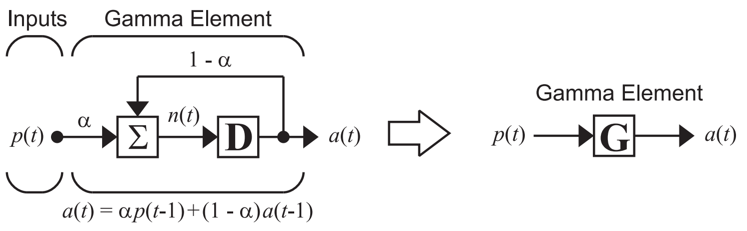
Focused Time Delay Neural Network



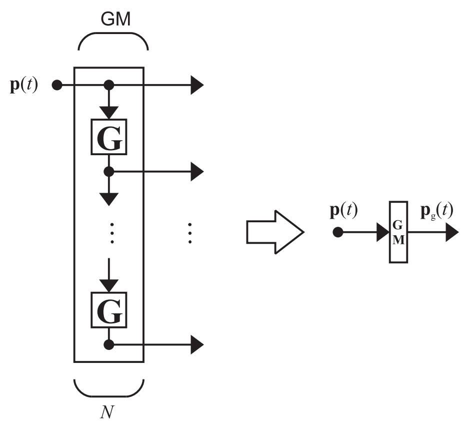
Distributed Time Delay Neural Network



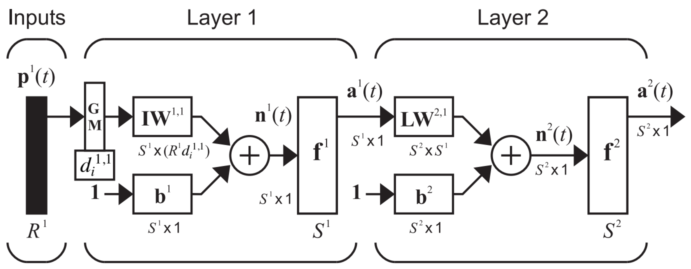
Gamma Element



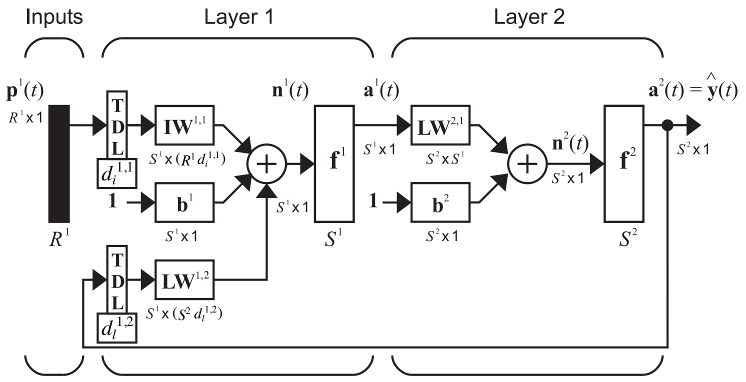
Gamma Memory Line



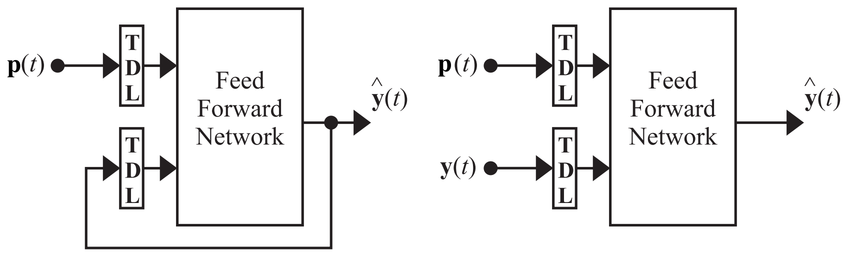
Focused Gamma Network



NARX Network

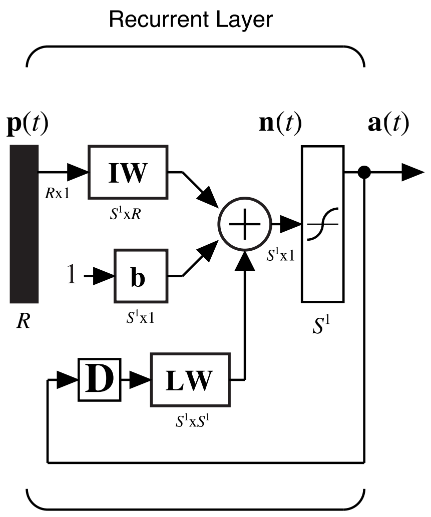


Parallel and Series-Parallel Architectures



Parallel Architecture Series-Parallel Architecture

Recurrent Neural Network



Real Time Recurrent Learning (RTRL)

$$\frac{\partial F}{\partial \mathbf{x}} = \sum_{t=1}^{Q} \left[\frac{\partial \mathbf{a}(t)}{\partial \mathbf{x}^T}\right]^T \times \frac{\partial^e F}{\partial \mathbf{a}(t)}$$

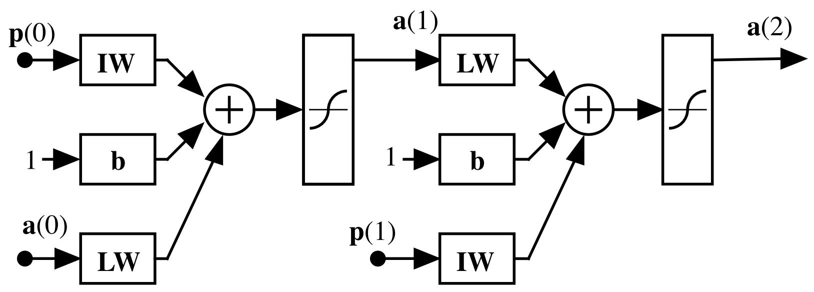
$$\frac{\partial \mathbf{a}(t)}{\partial \mathbf{x}^T} = \frac{\partial^e \mathbf{a}(t)}{\partial \mathbf{x}^T} + \frac{\partial^e \mathbf{a}(t)}{\partial \mathbf{a}(t-1)^T}\frac{\partial \mathbf{a}(t-1)}{\partial \mathbf{x}^T}$$

Backpropagation Through Time (BPTT)

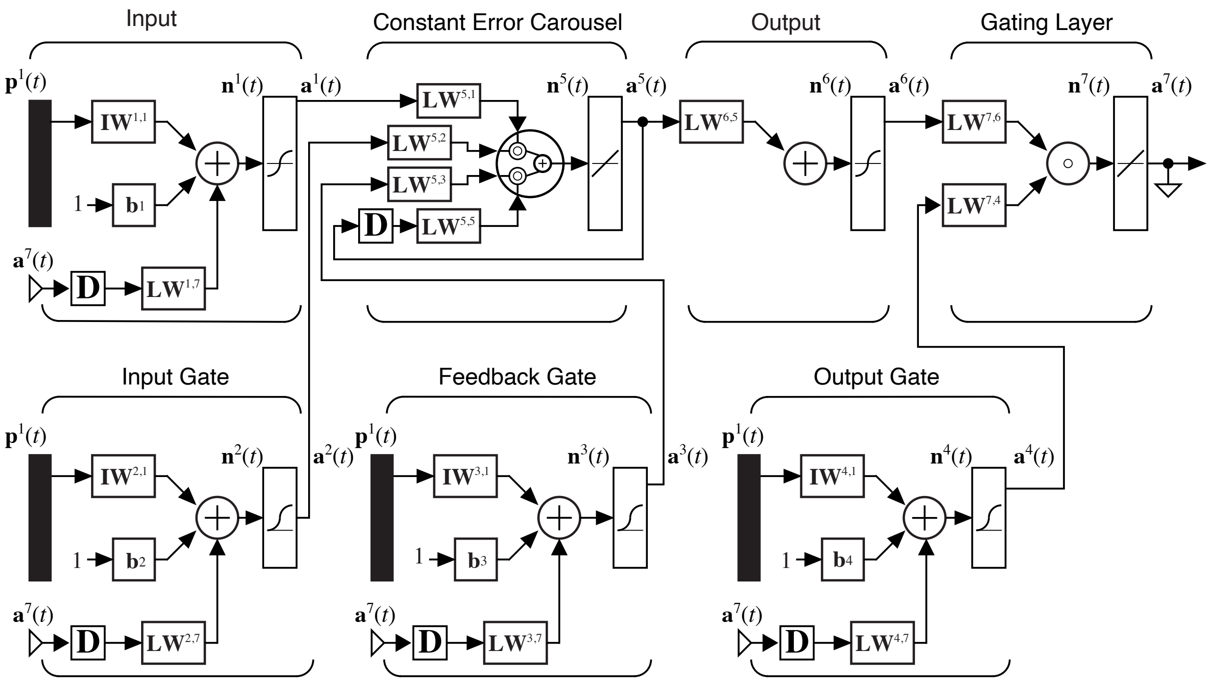
$$\frac{\partial F}{\partial \mathbf{x}} = \sum_{t=1}^{Q} \left[\frac{\partial^e \mathbf{a}(t)}{\partial \mathbf{x}^T}\right]^T \times \frac{\partial F}{\partial \mathbf{a}(t)}$$

$$\frac{\partial F}{\partial \mathbf{a}(t)} = \frac{\partial^e F}{\partial \mathbf{a}(t)} + \frac{\partial^e \mathbf{a}(t+1)}{\partial \mathbf{a}(t)^T} \times \frac{\partial F}{\partial \mathbf{a}(t+1)}$$

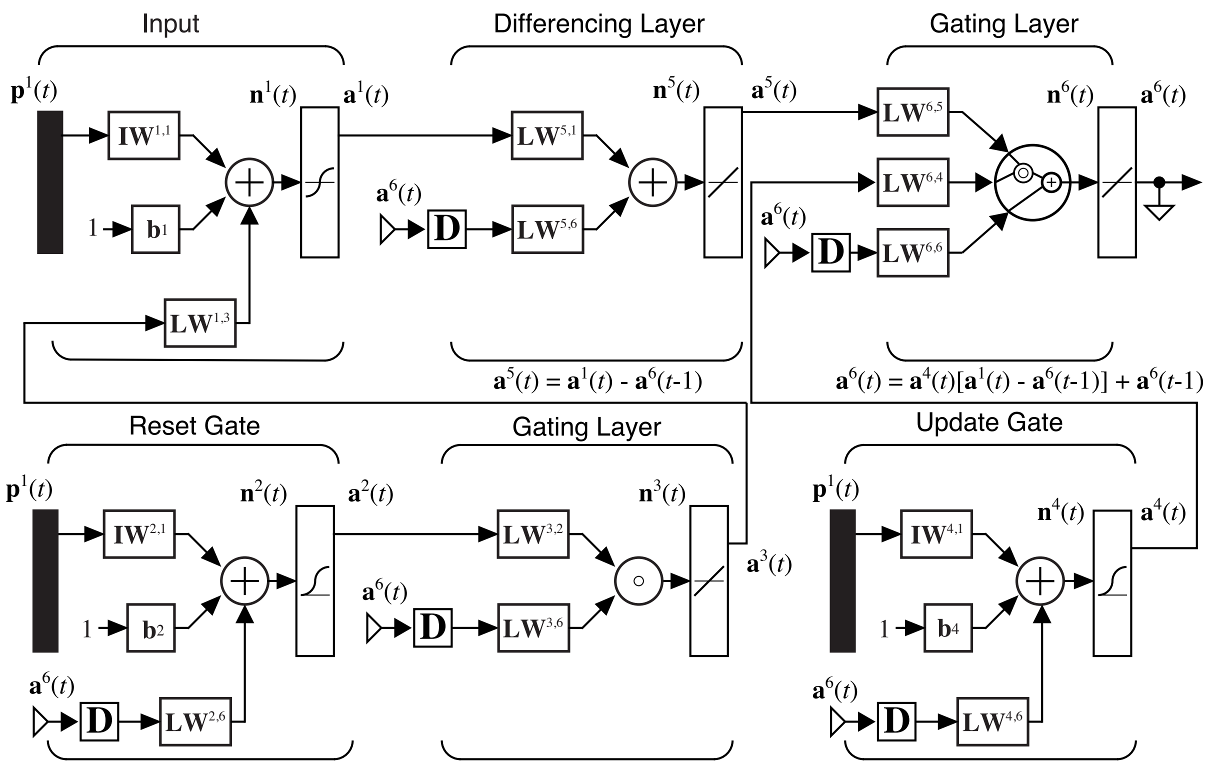
Unrolled RNN



Long Short Term Memory (LSTM) Network



Gated Recurrent Unit (GRU) Network



Solved Problems

P12.1 **Consider the FTDNN network of Figure 12.1. This has the form of an LDDN, as discussed in Chapter 7. Write out the equations of operation of this network, using the form of Eqs. 7.6 and 7.7.**

The general LDDN equations of operation are

$$\begin{aligned}\mathbf{n}^m(t) =& \sum_{l \in L_m^f} \sum_{d \in DL_{m,l}} \mathbf{LW}^{m,l}(d)\mathbf{a}^l(t-d)\\ &+ \sum_{l \in I_m} \sum_{d \in DI_{m,l}} \mathbf{IW}^{m,l}(d)\mathbf{p}^l(t-d) + \mathbf{b}^m\\ \mathbf{a}^m(t) =& \mathbf{f}^m(\mathbf{n}^m(t))\end{aligned}$$

The FTDNN network of Figure 12.1 has two layers. The first layer has a TDL input from $\mathbf{p}^1(t)$. The second layer has an input from $\mathbf{a}^1(t)$ with no delays. The first step is to identify the sets I_1, L_2^f and $DI_{1,1}$. The TDL in the first layer has length $d_i^{1,1}$. Assuming that the delays go from 1 to $d_i^{1,1}$, we can write:

$$I_1 = \{1\}, L_2^f = \{1\}, DI_{1,1} = \left\{1, 2, \cdots, d_i^{1,1}\right\}$$

We can then write the equations

$$\begin{aligned}\mathbf{n}^1(t) =& \sum_{d=1}^{d_i^{1,1}} \mathbf{IW}^{1,1}(d)\mathbf{p}^1(t-d) + \mathbf{b}^1\\ \mathbf{a}^1(t) =& \mathbf{f}^1(\mathbf{n}^1(t))\\ \mathbf{n}^2(t) =& \mathbf{LW}^{2,1}\mathbf{a}^1(t-d) + \mathbf{b}^2\\ \mathbf{a}^2(t) =& \mathbf{f}^2(\mathbf{n}^2(t))\end{aligned}$$

The net input in the first layer could also be written using augmented matrices as follows:

$$\mathbf{n}^1(t) = \left[\mathbf{IW}^{1,1}(1) \quad \mathbf{IW}^{1,1}(2) \quad \cdots \quad \mathbf{IW}^{1,1}(d_i^{1,1})\right] \begin{bmatrix} \mathbf{p}^1(t-1) \\ \mathbf{p}^1(t-2) \\ \vdots \\ \mathbf{p}^1(t-d_i^{1,1}) \end{bmatrix} + \mathbf{b}^1$$

P12.2 **We have a NARX network with two delays in the input TDL and two delays in the feedback TDL. The weights, biases and input sequence are provided below. The first layer activation is ReLU, and the second layer activation is linear. Perform two iterations (time steps) of the network.**

$$\mathbf{IW}^{1,1}(1) = \begin{bmatrix} 1 & -1 \\ -2 & 2 \end{bmatrix}, \mathbf{IW}^{1,1}(2) = \begin{bmatrix} 3 & 1 \\ 1 & -3 \end{bmatrix}$$

$$\mathbf{LW}^{1,2}(1) = \begin{bmatrix} 1 & -2 \\ 2 & 1 \end{bmatrix}, \mathbf{LW}^{1,2}(2) = \begin{bmatrix} 1 & 3 \\ -3 & 1 \end{bmatrix}, \mathbf{b}^1 = \begin{bmatrix} 1 \\ -2 \end{bmatrix}$$

$$\mathbf{LW}^{2,1}(0) = \begin{bmatrix} 2 & -1 \\ 1 & -2 \end{bmatrix}, \mathbf{b}^2 = \begin{bmatrix} -2 \\ 1 \end{bmatrix}$$

$$\mathbf{p}(t) = \left\{ \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 2 \\ 1 \end{bmatrix}, \begin{bmatrix} 1 \\ 2 \end{bmatrix}, \begin{bmatrix} 2 \\ 3 \end{bmatrix} \right\}$$

For dynamic networks, the first step is to fill the TDLs. If the network was being trained, and if we had target outputs, we could fill the feedback TDL with target values – we could replace previous values of $\mathbf{a}^2$ with previous targets. Since we don't have targets, we need to initialize the feedback TDL with zeros. We could also fill the input TDL with zeros, if we assume that the inputs were zero before the provided inputs. To demonstrate how the filling would work, we will fill the input TDL with the first two inputs.

If we use the LDDN formulation of Eq. 7.6, we can compute the net input for the first layer for the first time step.

$$\begin{aligned} \mathbf{n}^1(2) = &\mathbf{LW}^{1,2}(1)\mathbf{a}^2(1) + \mathbf{LW}^{1,2}(2)\mathbf{a}^2(0) \\ &+ \mathbf{IW}^{1,1}(1)\mathbf{p}^1(1) + \mathbf{IW}^{1,1}(2)\mathbf{p}^1(0) + \mathbf{b}^1 \end{aligned}$$

$$\mathbf{n}^1(2) = \begin{bmatrix} 1 & -2 \\ 2 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 1 & 3 \\ -3 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

$$+ \begin{bmatrix} 1 & -1 \\ -2 & 2 \end{bmatrix} \begin{bmatrix} 2 \\ 1 \end{bmatrix} + \begin{bmatrix} 3 & 1 \\ 1 & -3 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} + \begin{bmatrix} 1 \\ -2 \end{bmatrix} = \begin{bmatrix} 5 \\ -4 \end{bmatrix}$$

Then, using Eq. 7.7, we can compute the first layer output.

$$\mathbf{a}^1(2) = \mathbf{relu}(\mathbf{n}^1(2))$$
$$\mathbf{a}^1(2) = \mathbf{relu}\left(\begin{bmatrix} 5 \\ -4 \end{bmatrix}\right) = \begin{bmatrix} 5 \\ 0 \end{bmatrix}$$

For the second layer there is no TDL, so the computation is simpler.

$$\mathbf{n}^2(2) = \mathbf{LW}^{2,1}(0)\mathbf{a}^1(2) + \mathbf{b}^2$$
$$\mathbf{n}^2(2) = \begin{bmatrix} 2 & -1 \\ 1 & -2 \end{bmatrix} \begin{bmatrix} 5 \\ 0 \end{bmatrix} + \begin{bmatrix} -2 \\ 1 \end{bmatrix} = \begin{bmatrix} 8 \\ 6 \end{bmatrix}$$

Since the activation for the last layer is linear, the output is the same as the net input.

$$\mathbf{a}^2(2) = \mathbf{n}^2(2) = \begin{bmatrix} 8 \\ 6 \end{bmatrix}$$

For the next time step, we repeat the process, beginning with the net input for the first layer. The input and feedback TDLs are shifted from the previous time step.

$$\mathbf{n}^1(3) = \begin{bmatrix} 1 & -2 \\ 2 & 1 \end{bmatrix} \begin{bmatrix} 8 \\ 6 \end{bmatrix} + \begin{bmatrix} 1 & 3 \\ -3 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$
$$+ \begin{bmatrix} 1 & -1 \\ -2 & 2 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \end{bmatrix} + \begin{bmatrix} 3 & 1 \\ 1 & -3 \end{bmatrix} \begin{bmatrix} 2 \\ 1 \end{bmatrix} + \begin{bmatrix} 1 \\ -2 \end{bmatrix} = \begin{bmatrix} 9 \\ 21 \end{bmatrix}$$
$$\mathbf{a}^1(3) = \mathbf{relu}\left(\begin{bmatrix} 9 \\ 21 \end{bmatrix}\right) = \begin{bmatrix} 9 \\ 21 \end{bmatrix}$$

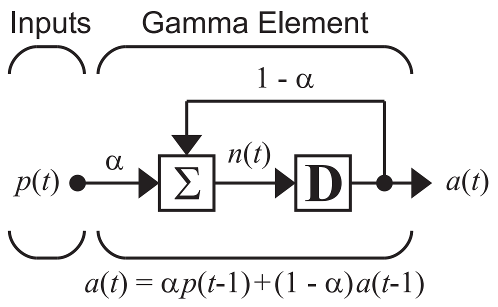
For the second layer at the next time step, we have

$$\mathbf{n}^2(3) = \begin{bmatrix} 2 & -1 \\ 1 & -2 \end{bmatrix} \begin{bmatrix} 9 \\ 21 \end{bmatrix} + \begin{bmatrix} -2 \\ 1 \end{bmatrix} = \begin{bmatrix} -5 \\ 32 \end{bmatrix}$$
$$\mathbf{a}^2(3) = \mathbf{n}^2(3) = \begin{bmatrix} -5 \\ 32 \end{bmatrix}$$

Rather than using Eqs. 7.6, we can also use augmented matrices, as shown in the previous problem. For example, for the net input calculation for the first layer in the first time step, we could have written the equation as follows (ignoring the feedback term, which is zero).

$$\mathbf{n}^1(2) = \begin{bmatrix} \mathbf{IW}^{1,1}(1) & \mathbf{IW}^{1,1}(2) \end{bmatrix} \begin{bmatrix} \mathbf{p}^1(1) \\ \mathbf{p}^1(0) \end{bmatrix} + \mathbf{b}^1$$
$$= \begin{bmatrix} 1 & -1 & 3 & 1 \\ -2 & 2 & 1 & -3 \end{bmatrix} \begin{bmatrix} 2 \\ 1 \\ 1 \\ 0 \end{bmatrix} + \begin{bmatrix} 1 \\ -2 \end{bmatrix} = \begin{bmatrix} 5 \\ -4 \end{bmatrix}$$

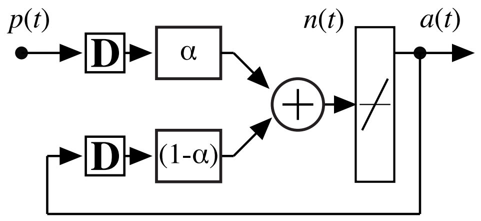
P12.3 Consider a single gamma element, as shown below.



i. Assume that this is the complete network, and convert this diagram into the standard LDDN form.

ii. Write out the BPTT equations to compute the gradient of the loss with respect to the gamma parameter α. Assume that the loss is sum squared error.

i. For the LDDN, each layer could have input and layer TDLs, a summing junction and an activation function. The gamma element can be adjusted to this form if we move the delay to the other side of the summing junction. The input weight would be α, and the layer weight would be $(1 - \alpha)$. The activation function would be linear. The converted network is shown below.



ii. Let's repeat the BPTT equations (Eqs. 12.8 and 12.9) here:

$$\frac{\partial F}{\partial \mathbf{x}} = \sum_{t=1}^{Q} \left[\frac{\partial^e \mathbf{a}(t)}{\partial \mathbf{x}^T} \right]^T \times \frac{\partial F}{\partial \mathbf{a}(t)}$$
$$\frac{\partial F}{\partial \mathbf{a}(t)} = \frac{\partial^e F}{\partial \mathbf{a}(t)} + \frac{\partial^e \mathbf{a}(t+1)}{\partial \mathbf{a}(t)^T} \times \frac{\partial F}{\partial \mathbf{a}(t+1)}$$

(Note that these equation represent BPTT only for networks with a single feedback delay, which fits the gamma element.)

Let's begin by finding the explicit derivatives first. The network operation equation is

$$a(t) = \alpha p(t-1) + (1-\alpha)a(t-1)$$

From this and the loss function, we can write the three explicit derivatives.

$$\frac{\partial^e a(t+1)}{\partial a(t)} = \frac{\partial^e a(t)}{\partial a(t-1)} = (1-\alpha)$$
$$\frac{\partial^e a(t)}{\partial x} = \frac{\partial^e a(t)}{\partial \alpha} = p(t-1) - a(t-1)$$
$$\frac{\partial^e F}{\partial a(t)} = -2e(t)$$

Now we can write out the BPTT equations.

$$\frac{\partial F}{\partial a(t)} = -2e(t) + (1-\alpha)\frac{\partial F}{\partial a(t+1)}, \text{ initialized with } \frac{\partial F}{\partial a(Q+1)} = 0$$
$$\frac{\partial F}{\partial \alpha} = \sum_{t=1}^{Q} [p(t-1) - a(t-1)] \times \frac{\partial F}{\partial a(t)}$$

Exercises

E12.1 Consider the DTDNN network of Figure 12.2. This has the form of an LDDN, as discussed in Chapter 7. Write out the equations of operation of this network, using the form of Eqs. 7.6 and 7.7.

E12.2 Consider the NARX network of Figure 12.6. This has the form of an LDDN. Write out the equations of operation of this network, using the form of Eqs. 7.6 and 7.7.

E12.3 Suppose that we have the NARX network of Figure 12.6, with delays of 1 and 2 in the input TDL and delays of 1, 2 and 3 in the feedback TDL. The training data is given below. Using the series-parallel arrangement of Figure 12.7, arrange the data and the network so that you can compute the gradient for training the network using standard static backpropagation. (Some of the ideas in Solved Problem P.12.1 could be useful here.)

t	0	1	2	3	4	5	6	7	8	9
Input	9	18	13	1	4	6	17	5	3	15
Target	0	9	24	27	14	7	5	20	17	12

E12.4 Suppose we have a two-layer FGMN, with two taps on the gamma memory. The first layer of the network has two neurons and a ReLU activation function. The second layer has one neuron and a linear activation function. Draw the network diagram in the form of an LDDN. (Hint. Use ideas from Solved Problem P.12.3, and make a layer for each gamma element.)

E12.5 Consider the RNN in Figure 12.8, which contains an output layer.

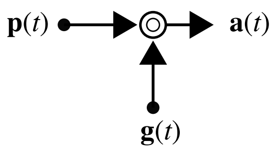
i. Unroll the network for two time steps.

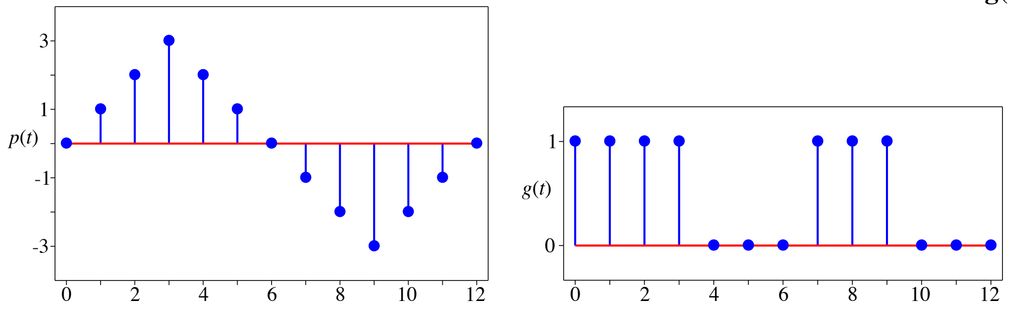
ii. Write out the backpropagation equations for updating the weight, assuming that there is a target output at Layer 2 for both time steps.

iii. Perform Lab 1, in which you will implement your backpropagation equations for this network. You can find it at `https://github.com/NNDesignDeepLearning/NNDesignDeepLearning/tree/master/NNDesignDeepLearning/12.NonlinearSequenceProcessingChapter/Code/Labs`

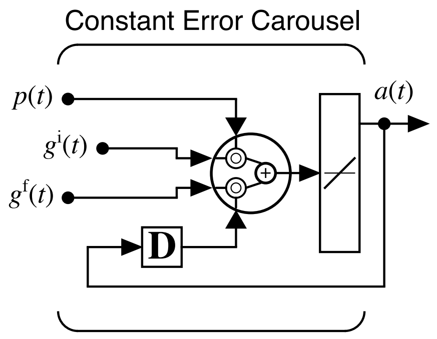
.

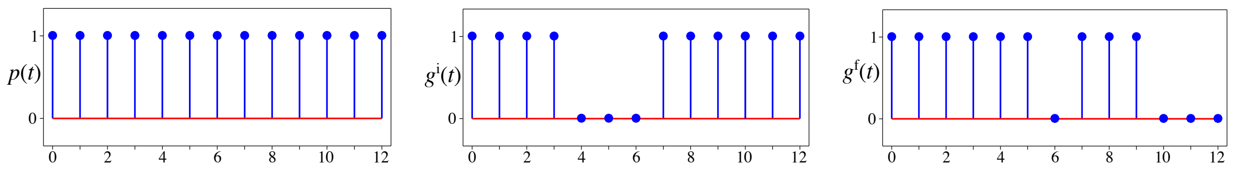
E12.6 Consider the gate shown in the margin and the input signal and gating signals in the figures below. Plot the output of the gate, $a(t)$.





E12.7 Consider the scalar CEC diagram and the input signal $p(t)$, input gating signal $g^{\mathrm{i}}(t)$ and feedback gating signal $g^{\mathrm{f}}(t)$ in the figures below. Plot the network output $a(t)$. Explain how each gate affects network operation.





E12.8 We have a NARX network with two delays in the input TDL and two delays in the feedback TDL. Unfold the network for two time steps. Discuss the difficulty in unfolding a NARX network (or other networks with TDLs) in comparison with unfolding RNNs.