Unit-8

Recurrent Neural Network

Introduction to RNN

- Recurrent Neural Network(RNN) are a type of neural Network where the output from previous step are fed as input to the current step.
- In traditional neural networks, all the inputs and outputs are independent of each other, but in cases like when it is required to predict the next word of a sentence, the previous words are required and hence there is a need to remember the previous words.

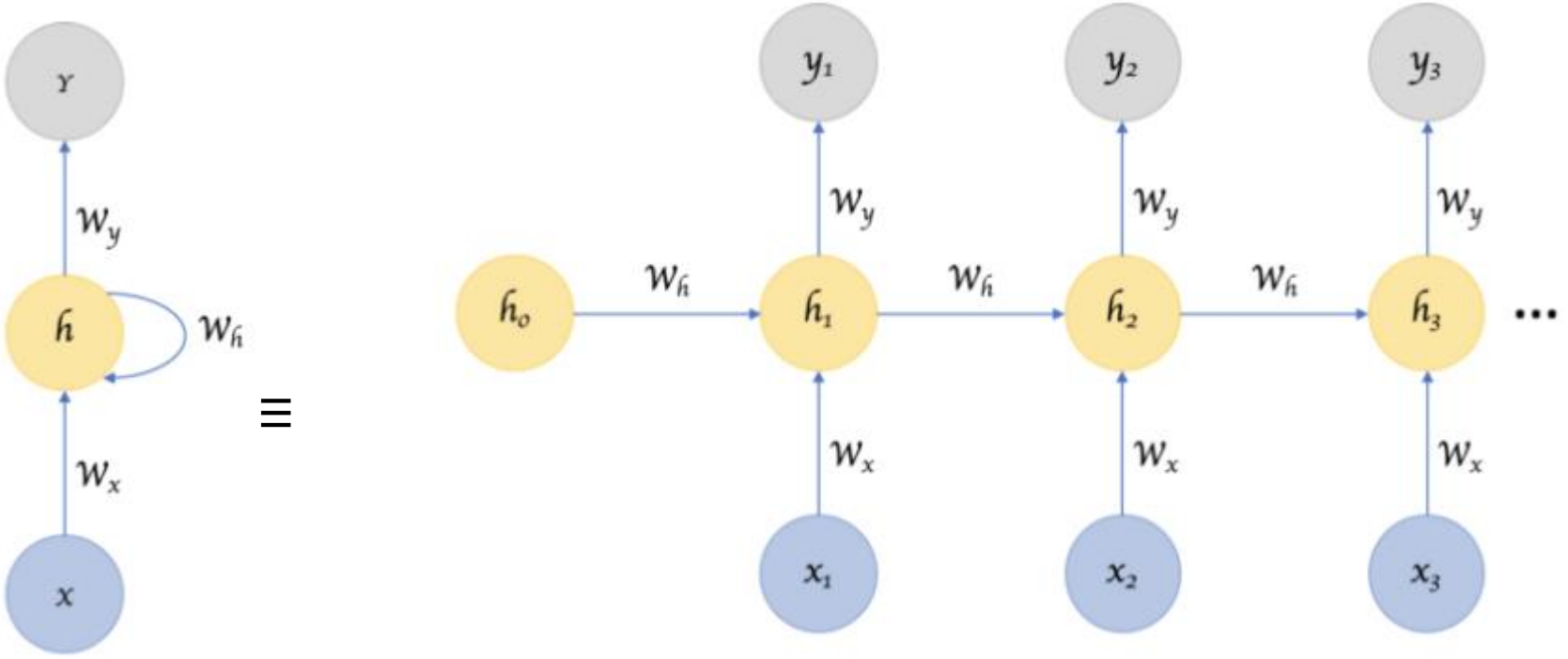
Introduction to RNN

- Thus RNN came into existence, which solved this issue with the help of a Hidden Layer. The main and most important feature of RNN is Hidden state, which remembers some information about a sequence.
- RNN have a **"memory"** which remembers all information about what has been calculated. It uses the same parameters for each input as it performs the same task on all the inputs or hidden states to produce the output.

Introduction to RNN

- Thus, RNN converts the independent activations into dependent activations by providing the same weights and biases to all the layers, thus reducing the complexity of increasing parameters and memorizing each previous outputs by giving each output as input to the next hidden layer (see right part of the figure in next slide).
- Hence layers of neural network in right side can be joined together such that the weights and bias of all the hidden layers is the same, into a single recurrent layer.

Introduction to RNN



Computation of hidden state

$$h_t = f(w_h h_{t-1} + w_x x_t)$$

Computation of output

$$y_t = f(w_y h_t)$$

Introduction to RNN

Applications of RNN

- Time series prediction
- Sequence prediction
- Natural Language Processing
- Speech processing

Forward Propagation in RNN

Example: Consider the word 'dogs'. Show forward propagation of RNN with three nodes in hidden layer to predict letter **'s'** given the letters **'d'**, **'o'**, and **'g'**. Assume that hidden layer activation function is Tanh and activation in output layer is softmax.

Solution

One hot encoding of the input: input vocabulary{d, o, g, s}

d	o	g	s
1	0	0	0
0	1	0	0
0	0	1	0
0	0	0	1

Forward Propagation in RNN

We know that:

$$h_t = f(w_h h_{t-1} + w_x x_t) = \tanh(w_h h_{t-1} + w_x x_t) \qquad Note: Soft\max(x) = \frac{e^x}{\sum_{i=1}^{n} e^{x_i}}$$

$$y_t = f(w_y h_t) = soft\max(w_y h_t)$$

Here, dimension of w_h is d × d, dimension of w_x is d × k, and dimension of w_y is k × d. Where, d is number of hidden nodes, and k is dimension of input vector.

Assume that:

$$w_h = \begin{bmatrix} 0.1 & 0.5 & 0.1 \\ 0.5 & 0.9 & 0.3 \\ 0.3 & 0.2 & 0.1 \end{bmatrix} \qquad w_x = \begin{bmatrix} 0.6 & 0.8 & 0.4 & 0.8 \\ 0.2 & 0.9 & 0.8 & 0.7 \\ 0.9 & 0.8 & 0.1 & 0.2 \end{bmatrix} \qquad w_y = \begin{bmatrix} 0.9 & 0.8 & 0.3 \\ 0.2 & 0.3 & 0.4 \\ 0.6 & 0.9 & 0.1 \\ 0.5 & 0.0 & 0.3 \end{bmatrix}$$

Forward Propagation in RNN

Thus, (for input **'d'**)

$$h_t = \tanh\left(\begin{bmatrix} 0.1 & 0.5 & 0.1 \\ 0.5 & 0.9 & 0.3 \\ 0.3 & 0.2 & 0.1 \end{bmatrix}\begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 0.6 & 0.8 & 0.4 & 0.8 \\ 0.2 & 0.9 & 0.8 & 0.7 \\ 0.9 & 0.8 & 0.1 & 0.2 \end{bmatrix}\begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}\right) = \tanh\left(\begin{bmatrix} 0.6 \\ 0.2 \\ 0.9 \end{bmatrix}\right) = \begin{bmatrix} 0.54 \\ 0.20 \\ 0.72 \end{bmatrix}$$

$$y_t = soft\max\left(\begin{bmatrix} 0.9 & 0.8 & 0.3 \\ 0.2 & 0.3 & 0.4 \\ 0.6 & 0.9 & 0.1 \\ 0.5 & 0.0 & 0.3 \end{bmatrix}\begin{bmatrix} 0.54 \\ 0.20 \\ 0.72 \end{bmatrix}\right) = soft\max\left(\begin{bmatrix} 0.86 \\ 0.45 \\ 0.57 \\ 0.48 \end{bmatrix}\right) = \begin{bmatrix} 0.32 \\ 0.21 \\ 0.24 \\ 0.22 \end{bmatrix}$$

At t=1, RNN predicts the letter "d" given the input "d". This doesn't make sense but it's ok because we've used untrained random weights.

Forward Propagation in RNN

Example: Consider the input series {0.2,0.4,0.6, 0.8}. Show forward propagation of RNN with two nodes in hidden layer to predict next term in the series. Assume that activation function of hidden layer is sigmoid and activation in output layer is linear.

Solution {Input: 0.2 **}**

$$h_t = \tanh\left(\begin{bmatrix} 0.1 & 0.5 \\ 0.5 & 0.9 \end{bmatrix}\begin{bmatrix} 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 0.6 \\ 0.2 \end{bmatrix}\begin{bmatrix} 0.2 \end{bmatrix}\right) = \sigma\left(\begin{bmatrix} 0.12 \\ 0.04 \end{bmatrix}\right) = \begin{bmatrix} 0.53 \\ 0.51 \end{bmatrix}$$

$$y_t = \begin{bmatrix} 0.4 & 0.7 \end{bmatrix}\begin{bmatrix} 0.53 \\ 0.51 \end{bmatrix} = 0.57$$

RNN Architectures

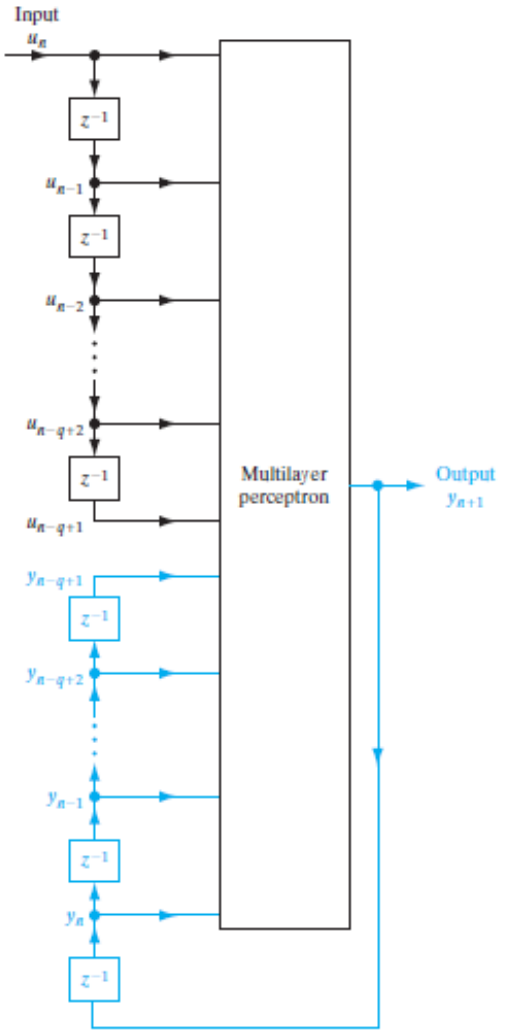
- The architectural layout of a recurrent network takes many different form. Four commonly used RNN architectures are:
  - **Input-Output Recurrent Model**
  - **State Space Model**
  - **Recurrent Multilayer perceptron**
  - **Second Order Network**

RNN Architectures

Input-Output Recurrent Model

- The model has a single input that is applied to a tapped delay- line memory of *q units. It has a single output that is fed back to the input via another* tapped-delay-line memory of *q units.*
- The contents of these two tapped-delay-line memories are used to feed the input layer of the multilayer perceptron.

$$y_{n+1} = F(y_n, \ldots\ldots, y_{n-q+1}, u_n, \ldots\ldots, u_{n-q+1})$$



RNN Architectures

State-Space Model

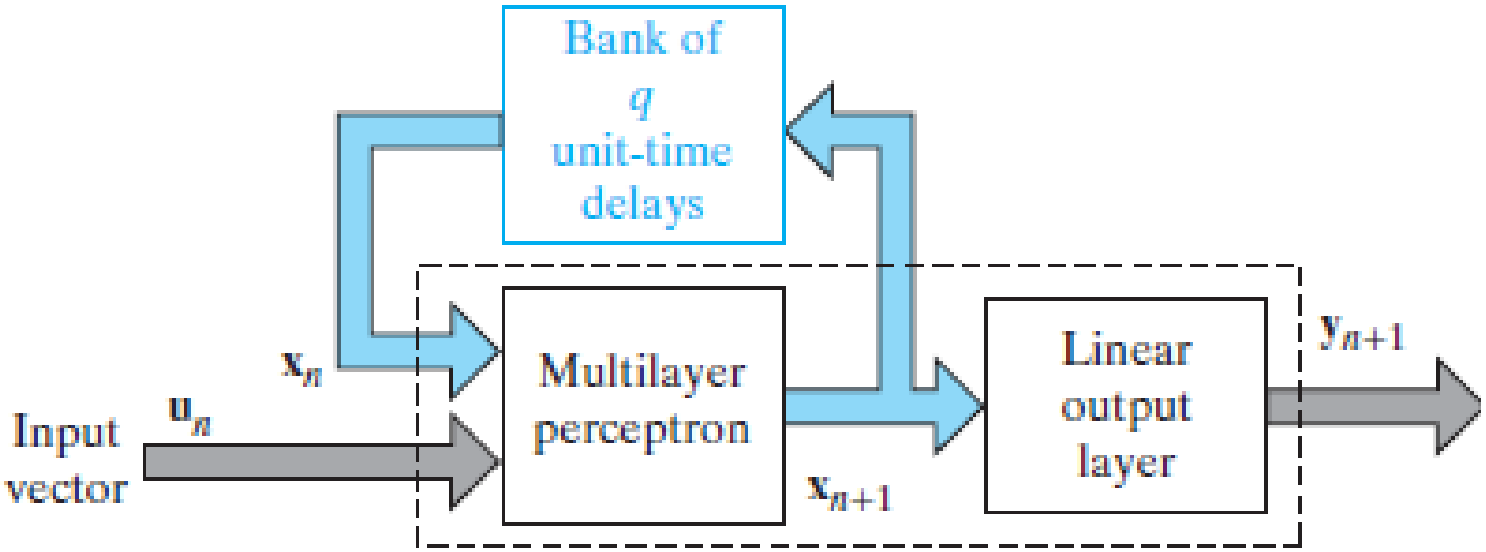
- The hidden neurons define the *state of the network. The output of the hidden layer is fed back to the input* layer via a bank of unit-time delays. The input layer consists of a concatenation of feedback nodes and source nodes.
- Dynamic behavior of the model can be described by the following pair of equations:

$$x_{n+1} = f(x_n, u_n)$$

$$y_n = wx_n$$

where, w is weight matrix

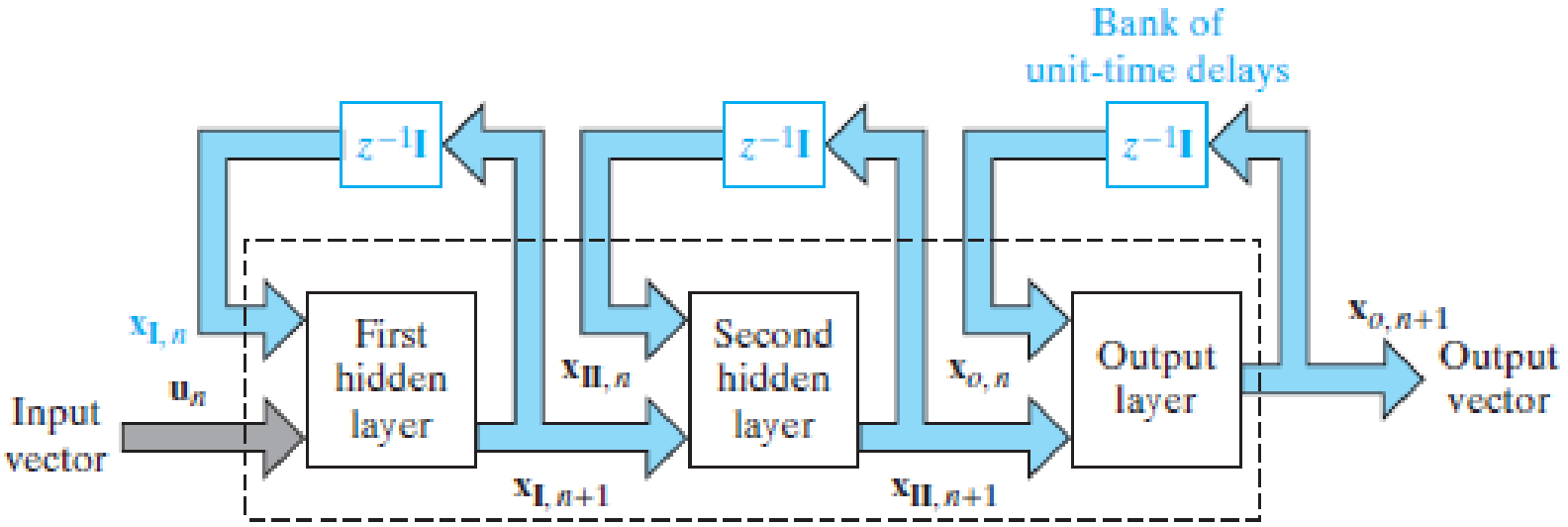
RNN Architectures



RNN Architectures

Recurrent Multilayer Perceptron (RMLP)

- RMLP has one or more hidden layers. Each computation layer of an RMLP has feedback around it.



RNN Architectures

Recurrent Multilayer Perceptron (RMLP)

- Behavior of RMLP can be represented by following system of equations.

$$x_{1,n+1} = \phi_1(x_{1,n}, u_n)$$

$$x_{2,n+1} = \phi_2(x_{2,n}, x_{1,n+1})$$

$$x_{o,n+1} = \phi_o(x_{o,n}, x_{2,n+1})$$

where, ϕ is activation function of the layer

RNN Architectures

Second order network

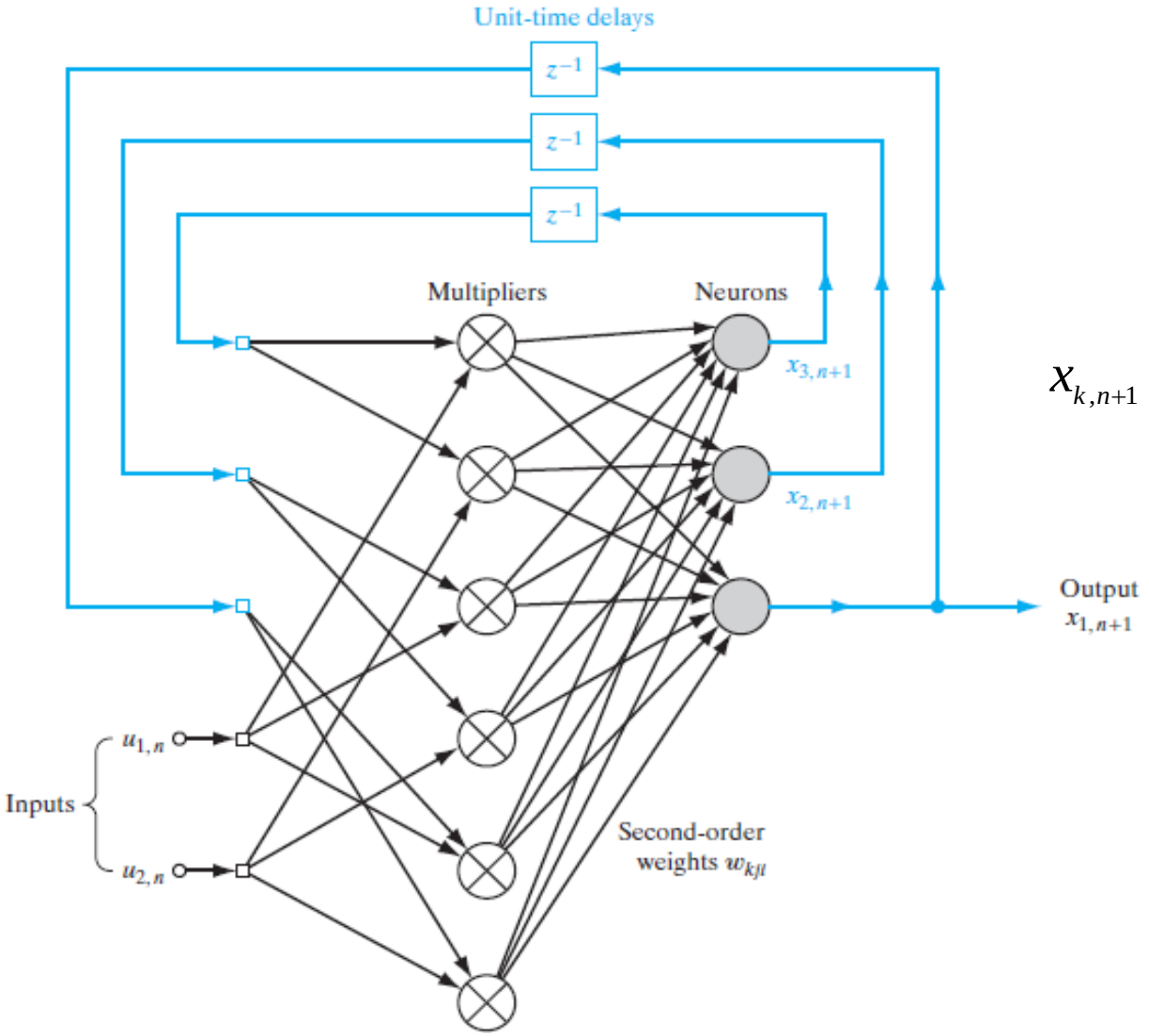
- The second-order neuron *k uses a single* weight w_{kij} that connects it to the input nodes *i and j.*
- The induced local field v_k *of* neuron k is combined using multiplications as shown in the equation below.

$$v_k = \sum_i \sum_j w_{kij} x_i u_j$$

where, u is source input and x is feedback

- The RNN that is formed on the basis of second order neuron is called second order recurrent network

RNN Architectures



$$x_{k,n+1} = \phi(v_{k,n}) = \frac{1}{1+\exp(-v_{k,n})}$$

Universal Approximation Theorem

- Universal approximation theorem states that "*any non-linear dynamical system can be approximated to any accuracy by a recurrent neural network, with no restrictions on the compactness of the state space, provided that the network has enough sigmoidal hidden units*".
- This means hidden neurons of RNN must be equipped with sigmoidal activation functions as given below:

$$\varphi(x) = \frac{1 - e^{-2x}}{1 + e^{-2x}} \quad \text{or} \quad \varphi(x) = \frac{1}{1 + e^{-x}}$$

Controllability and Observability

- A recurrent neural network is said to be controllable if an initial state is steerable to any desired state within a finite number of time steps.
- A recurrent network is said to be observable if the state of the network can be determined from a finite set of input/output measurements.

Computational Power of recurrent Networks

- Computational power of recurrent neural network is based on following theorem.
- Theorem: *All Turing machines may be simulated by fully connected recurrent networks built on neurons with sigmoidal activation functions* (**(Siegelmann and Sontag, 1991).**
- They show that, for every computable function, there exists a finite RNN (of the form described above) that can compute it. They do this by showing that it's possible to use a RNN to explicitly simulate a pushdown automaton with two stacks.

Computational Power of recurrent Networks

- This is another model that's computationally equivalent to a Turing machine. Any computable function can be computed by a Turing machine. Any Turing machine can be simulated by a pushdown automaton with two stacks.
- Any pushdown automaton with two stacks can be simulated by a RNN. Therefore, any computable function can be computed by a RNN.

Learning Algorithms

- Training algorithms in RNN can be divided into two categories:
- *Epochwise Training*
- *Continuous Training*
- In epochwise training the recurrent network uses a temporal sequence of input–target pairs and starts running from some initial state until it reaches a new state, at which point the training is stopped and the network is reset to an initial state for the next epoch.

Learning Algorithms

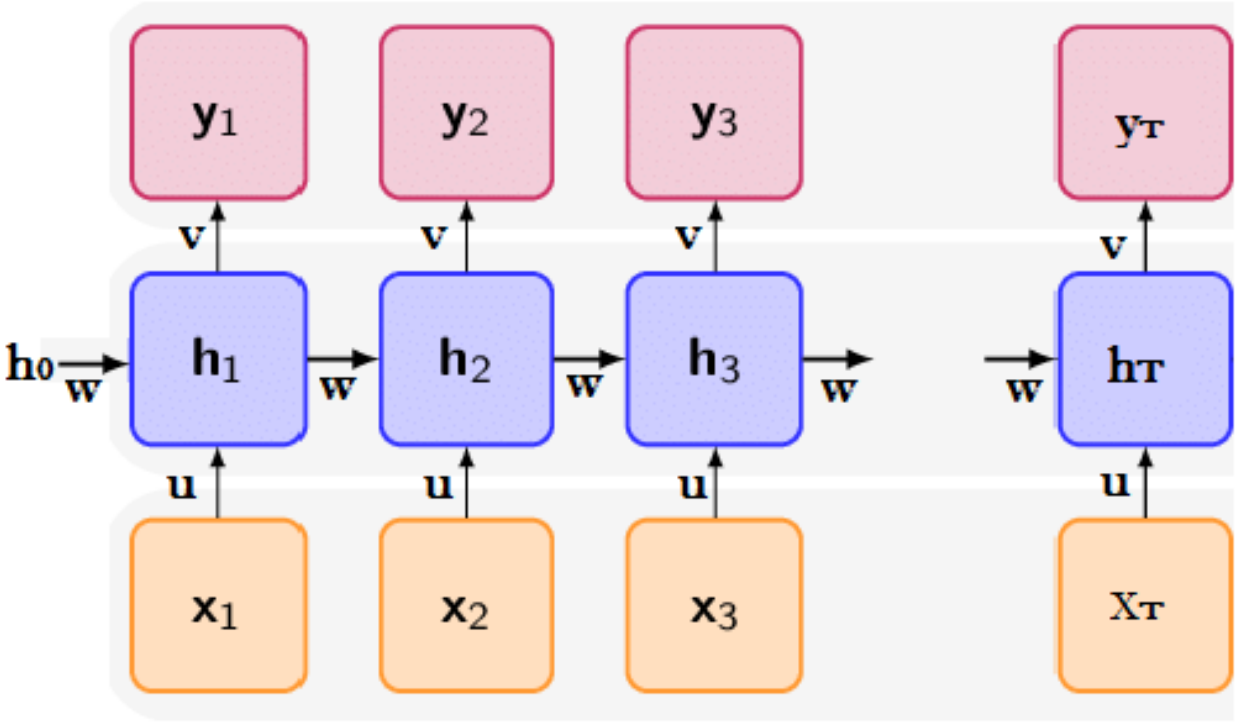
- The initial state doesn't have to be the same for each epoch of training. Rather, what is important is for the initial state for the new epoch to be different from the state reached by the network at the end of the previous epoch.
- Although an epoch in the training of a multilayer perceptron involves the entire training sample of input–target pairs, an epoch in the training of a recurrent neural network involves a group of temporally consecutive input–target pairs.

Learning Algorithms

- Continuous training is suitable for situations where there are no reset states available and on-line learning is required. The network learns while signal processing is being performed by the network.

Backpropagation Through Time (BPTT)

- Unrolling of RNN with T layers looks like below, where w, u, v are weigh matrices.



Backpropagation Through Time (BPTT)

- Let L is total loss in the RNN. Then the weights w, u, v are updated as below.

$$w = w - \alpha \frac{\partial L}{\partial w}$$

$$v = v - \alpha \frac{\partial L}{\partial v}$$

$$u = u - \alpha \frac{\partial L}{\partial u}$$

- Thus, we need to compute $\partial L/\partial w, \partial L/\partial v, \partial L/\partial u$ for updating weights.

Backpropagation Through Time (BPTT)

- Let L_t is the loss at t^{th} step of RNN. Then total loss of the RNN is given by.

$$L = \sum_{t=1}^{T} L_t$$

- Loss function can be different according to need. Basically, cross-entropy or least square loss function is used. Normally, Cross-entropy loss function is used for classification problems whereas least square loss function is used for regression problems.

Backpropagation Through Time (BPTT)

- Recall that computation of hidden state and output in RNN is performed as below:

Computation of hidden state *Computation of output*

$h_t = f(wh_{t-1} + ux_t)$ $y_t = g(vh_t)$

- Recall that, we need to compute $\partial L/\partial w, \partial L/\partial v, \partial L/\partial u$ BPTT. Assume that t=3

$h_3 = f(wh_2 + ux_3)$ $y_3 = g(vh_3)$

- Also, assume that

$$L_3 = \frac{1}{2}(d_3 - y_3)^2$$

,where d is desired output and y is predicted output

Backpropagation Through Time (BPTT)

- Now,

$$\frac{\partial L_3}{\partial v} = \frac{\partial L_3}{\partial y_3}\frac{\partial y_3}{\partial v} = -(d_3 - y_3)h_3$$

$$\frac{\partial L_3}{\partial w} = \frac{\partial L_3}{\partial y_3}\frac{\partial y_3}{\partial h_3}\frac{\partial h_3}{\partial w} = -(d_3 - y_3)vf'(z_3)\frac{\partial z_3}{\partial w}$$

$$= -(d_3 - y_3)vf'(z_3)\left[h_2 + w\frac{\partial h_2}{\partial w}\right]$$

- Here, $z_3 = wh_2 + ux_3$
- In the above relation derivative of h_2 needs to be computed recursively in terms of h_1.

Backpropagation Through Time (BPTT)

- Finally,

$$\frac{\partial L_3}{\partial u} = \frac{\partial L_3}{\partial y_3}\frac{\partial y_3}{\partial h_3}\frac{\partial h_3}{\partial u} = -(d_3 - y_3)vf'(z_3)\frac{\partial z_3}{\partial u}$$

$$= -(d_3 - y_3)vf'(z_3)\left[x_3 + w\frac{\partial h_2}{\partial u}\right]$$

- Again, in the above relation, derivative of h_2 needs to be computed recursively in terms of h_1.
- Since the derivatives $\partial L/\partial v$, and $\partial L/\partial u$ needs to be calculated recursively by back propagating through time the algorithm is named as BPTT.

Backpropagation Through Time (BPTT)

Algorithm

1. For i=1 to T Compute y_i and L_i as below

   $y_i = g(vh_i)$, *where v is weight matrix*

   $L_i = \frac{1}{2}(d_i - y_i)^2$, *where d is desired output and y is predicted output*
2. For i=1 to T Compute following gradients

   $\partial L_i/\partial w, \partial L_i/\partial v, \partial L_i/\partial u$
3. Find total gradient by summing all local gradients
4. Update weights as below

$$w = w - \alpha \frac{\partial L}{\partial w} \qquad v = v - \alpha \frac{\partial L}{\partial v} \qquad u = u - \alpha \frac{\partial L}{\partial u}$$

Vanishing Gradient and Truncated BPTT

- Two main problems associated with Recurrent Neural networks are:
  1. Gradient calculations either vanish or explode
  2. Gradient calculations are expensive
- Solution to exploding gradient is gradient clipping and solution to vanishing gradient problem is to use alternate architectures like gated recurrent unit (GRU) network and Long Short-term Memory (LSTM) network. Both, GRU and LSTM are variants of RNN architecture.
- Solution to second problem is to used truncated BPTT algorithm

Vanishing Gradient and Truncated BPTT

- RNNs might have a difficulty in learning long range dependencies. For instance if we have a sentence like “The man who ate my pizza has purple hair”. In this case, the description purple hair is for the man and not for the pizza. So this is a long dependency.
- In BPTT algorithm, we need to compute gradient of loss to update weights.
- In order to compute loss we need to use recursive computation.
- For instance following two gradients can be computed as below.

Vanishing Gradient and Truncated BPTT

$$\frac{\partial L_t}{\partial w} = -(d_t - y_t)vf'(z_t)\left[h_{t-1} + w\frac{\partial h_{t-1}}{\partial w}\right]$$

$$\frac{\partial L_t}{\partial u} = -(d_t - y_t)vf'(z_t)\left[x_t + h_t + w\frac{\partial h_t}{\partial u}\right]$$

- In the above equations gradient of h_{t-1} needs to be computed recursively in terms of h_{t-2} . Thus, the weight matrix w is multiplied by many times while computing the gradients.

Vanishing Gradient and Truncated BPTT

- If w is small, we experience a vanishing gradient problem. And if w is large, we experience an exploding gradient problem.
- The lower the gradient is, the harder it is for the network to update the weights and the longer it takes to get to the final result.
- Exploding gradients are a problem where large error gradients accumulate and result in very large updates to neural network model weights during training. This has the effect of our model being unstable and unable to learn from your training data.

Vanishing Gradient and Truncated BPTT

- In BPTT algorithm we need to back propagate through entire sequence in order to compute gradient of local loss. Thus, computation of gradient of loss is time consuming process.
- The practical solution is limit the backpropagation to a maximum of window of M steps. The forward pass should still be performed over the entire sequence, but the backward pass is truncated to windows of size M.
- This modified version of BPTT algorithm is called Truncated BPTT or TBPTT.

Vanishing Gradient and Truncated BPTT

- The TBPTT algorithm can be summarized as below:
  1. Present a sequence of k1 time steps of input and output pairs to the network.
  2. calculate and accumulate errors across k2 time steps.
  3. update weights.
  4. Repeat
- As we can clearly see that you need two parameters namely k1 and k2 for implementing TBPTT. K1 is the number of forward pass time steps between updates. This influences how fast or slow will be the training and the frequency of the weight updates.

Vanishing Gradient and Truncated BPTT

- On the other hand, k2 is the number of time steps which apply to BPTT. It should be large enough to capture the temporal structure in the problem for the network to learn.
- But the problem with TBPTT is that the network can't learn long dependencies as in BPTT because of limit on flow of gradient due to truncation.

Real Time Recurrent Learning (RTRL)

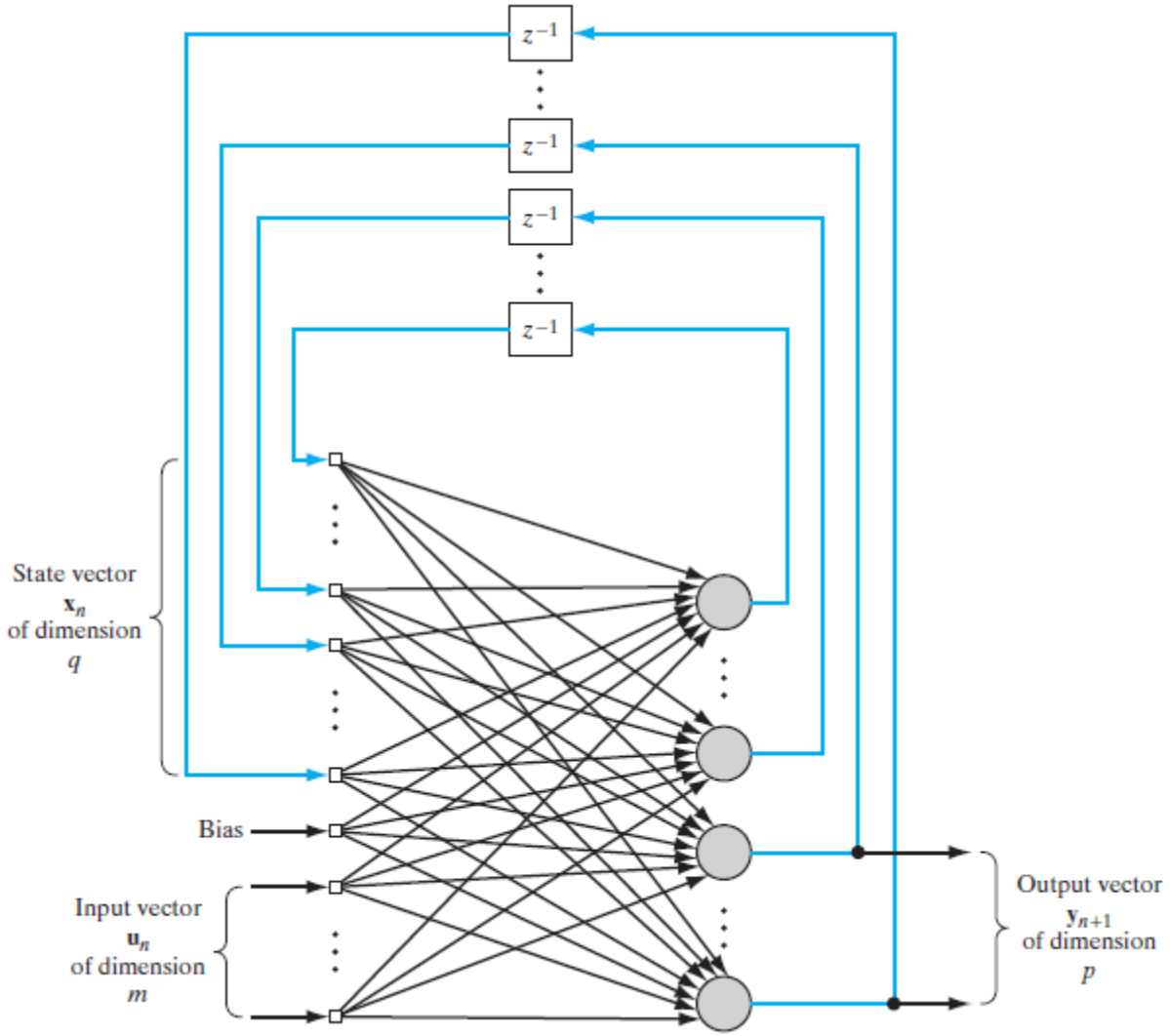
- Basically, BPTT and RTRL involve the propagation of derivatives, one in the backward direction and the other in the forward direction.
- BPTT requires less computation than RTRL does, but the memory space required by BPTT increases fast as the length of a sequence of consecutive input–target response pairs increases.
- BPTT is better for off-line training, and RTRL is more suitable for on-line continuous training.

Real Time Recurrent Learning (RTRL)

Summarized RTRL Algorithm

- **Assumptions**: *m is dimensionality of input space, q is dimensionality of state space, and p is dimensionality of output space*.
- The RTRL algorithm presented here considers fully connected RNN as given in the figure of next slide.

Real Time Recurrent Learning (RTRL)



Real Time Recurrent Learning (RTRL)

- Initialize synaptic weights randomly to small values
- Initialize state vector x(0)=0
- For j=1 to q: Initialize $\partial_{j,0} = 0$
- For n=0 to N
  - Compute error vector as: $e_n = d_n - w_c x_n$
  - Compute Change in weight as:
    $\Delta w_{j,n} = \alpha w_c \partial_{j,n} e_n$, *where* $\partial_{j,n}$ *is matrix of partial derivatives*
  - Compute partial derivatives for next step as:
    $\partial_{j,n+1} = \Phi(w_a \partial_{j,n} - U_{j,n})$, *where* Φ *is diagonal matrix*

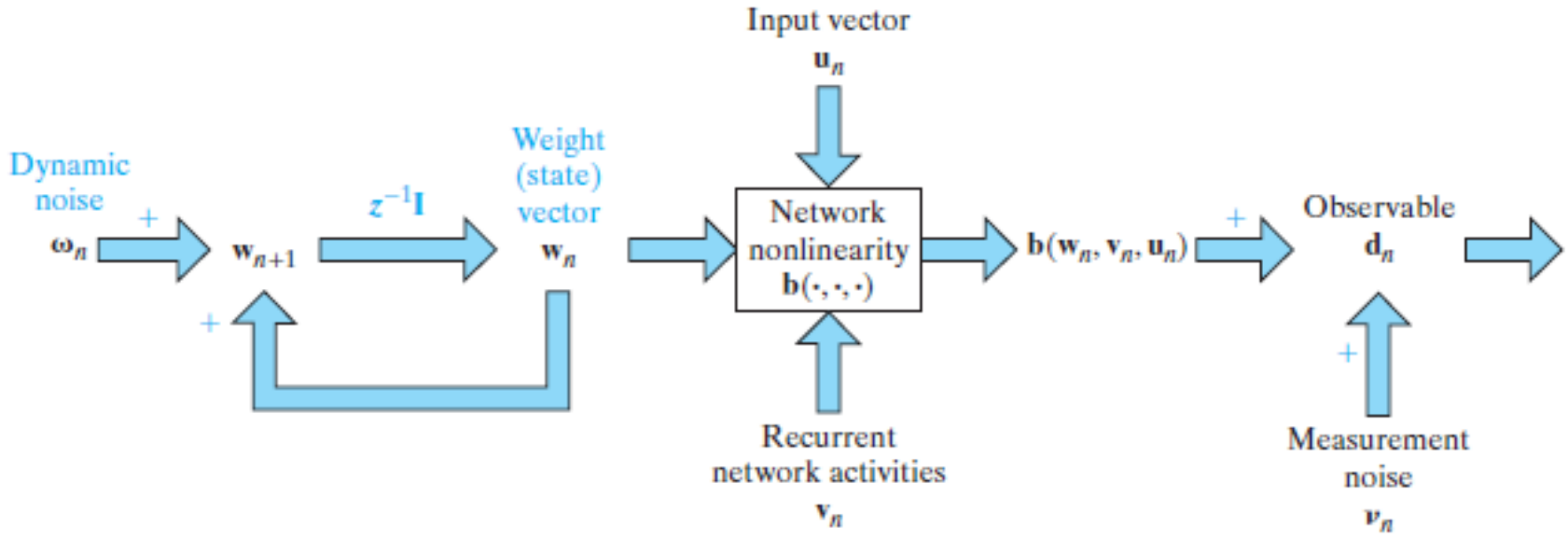
    whose j^{th} *diagonal is p.d. of actication function of its* argument *and*

    $U_{j,n}$ *is matrix whose* j^{th} *row is* $\begin{bmatrix} x_n \\ u_n \end{bmatrix}^T$ *and all other rows are zero*

Supervised Learning Framework for RNN using Nonlinear Sequential State Estimators

- Consider a recurrent network built around a multilayer perceptron with s synaptic weights and p output nodes.
- With n denoting a time-step in the supervised training of the network, let the vector $\mathbf{w}_n$ **denote the entire set of synaptic** weights in the network computed at time-step n.
- With sequential state estimation in mind, the state-space model of the network under training is defined by the following pair of models.

Supervised Learning Framework for RNN using Nonlinear Sequential State Estimators



Supervised Learning Framework for RNN using Nonlinear Sequential State Estimators

- **System Model**: System model is defined by the following random walk equation.

$$w_{n+1} = w_n + \omega_n$$

- The *dynamic noise* ω_n *is a white Gaussian noise* which is purposely included in the system model to *anneal the supervised training* of the network over time.
- In the early stages of the training, ω_n is large in order to encourage the supervised-learning algorithm to escape local minima, and then it is gradually reduced to small value.

Supervised Learning Framework for RNN using Nonlinear Sequential State Estimators

- **Measurement Model**: This model is described by the following equation.

$$d_n = f(w_n, v_n, u_n) + \upsilon_n$$

- In the above equation,
- d_n is observable
- v_n is internal state
- u_n is input signal
- υ_n is measurement noise and
- f is non-linearity of MLP from input to output

Supervised Learning Framework for RNN using Nonlinear Sequential State Estimators

- There are two different context in which notion of sate is considered in supervised training of RNN.
- **Externally Adjustable State**: This refers to adjustments applied to the network's weights through supervised training.
- **Internally Adjustable State**: This is represented by the internal state of the recurrent node. Input signal, dynamic noise, and feedback play role in evolution of hidden state over time.

Adaptaptivity Considerations

- An interesting property of a recurrent neural network, observed after the network has been trained in a supervised manner, is the *adaptive behavior*.
- This phenomenon occurs despite the fact that the synaptic weights in the network have been fixed. The root of this adaptive behavior may be traced to a fundamental theorem, which is stated as follows:

Adaptaptivity Considerations

- *"Consider a recurrent neural network embedded in a stochastic environment with relatively small variability in its statistical behavior. Provided that that the underlying probability distribution of the environment is fully represented in the supervised-training sample supplied to the network, it is possible for the network to adapt to the relatively small statistical variations in the environment without any further on-line adjustments being made to the synaptic weights of the network."*

Adaptaptivity Considerations

- This fundamental theorem is valid *only for recurrent networks. We say so because the* dynamic state of a recurrent network actually acts as a "short-term memory" that carries an estimate or statistic of the uncertain environment for adaptation, in which the network is embodied.
- However, it is not expected that it will work as effectively as a truly adaptive neural network, where provision is made for automatic on-line weight adjustments if the environment exhibits large statistical variability.

Adaptaptivity Considerations

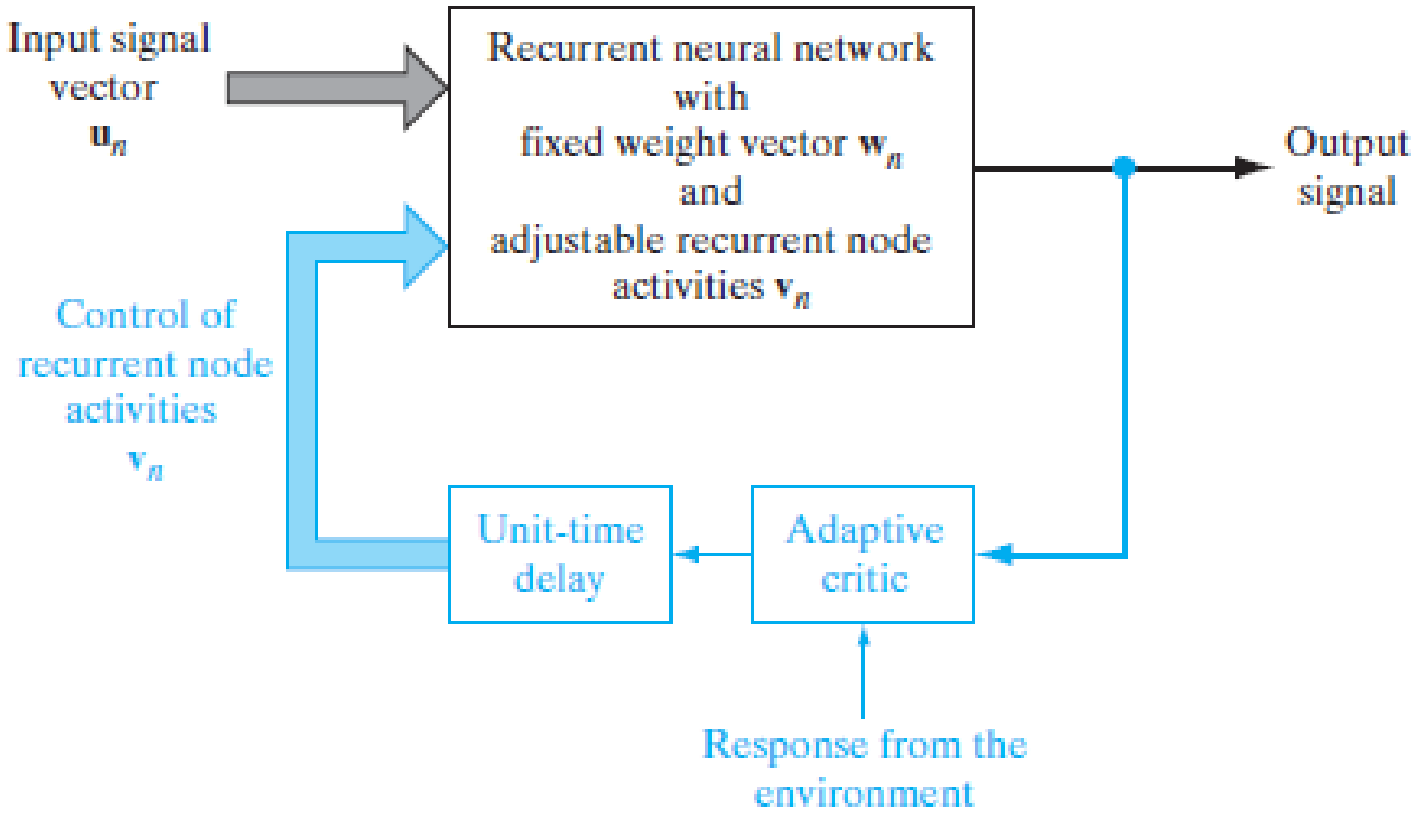


Figure: Block diagram illustrating the control of hidden state