Temporal Processing Using Feedforward Networks

Dr. Ratna Babu Chinnam

Industrial & Systems Engineering

Wayne State University

Primary Source: Simon Haykin (2009)

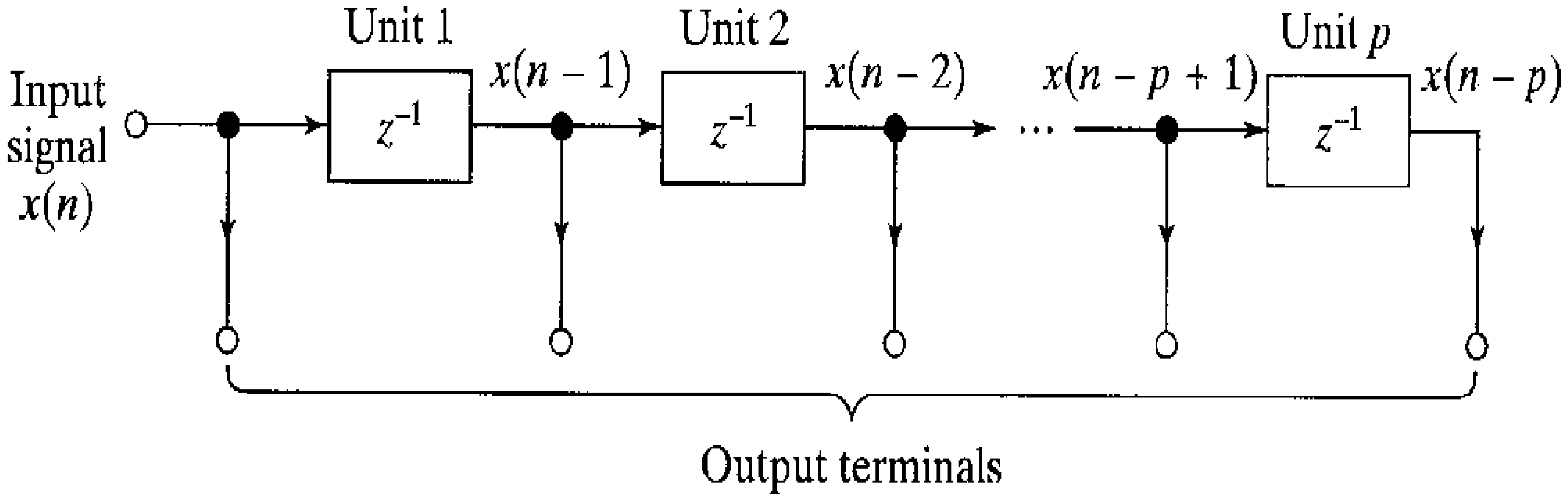
Temporal Processing Using FFNs

- "Time" enables network to follow variations in *nonstationary processes*
  - Examples: market demand, equipment degradation, stock markets
- Time allows a "static" network (e.g., MLP) to possess "*dynamic*" properties
- For a neural network to be "dynamic", it must be given "*memory*"
- How do we build time into the operation of a neural network?
  - *Implicit Representation: Our interest!*
    - Example: Signal can be "sampled" over time and this temporal sample forms part of the input
  - *Explicit Representation:* Time becomes an additional input variable
    - Not effective unless signal exhibits a distinct and consistent signature with time
- Memory can be "short-term" and "long-term", depending on retention time
  - Long-term memory is built into FFNs through supervised learning
  - It is short-term memory that makes the network dynamic
- One can build short-term memory into FFNs through the use of *time delays*
  - Can be implemented at synaptic level (*Distributed TLFN*)
  - Can be implemented at input layer (*Focussed TLFN*)
- Networks are still trained using error correction methods

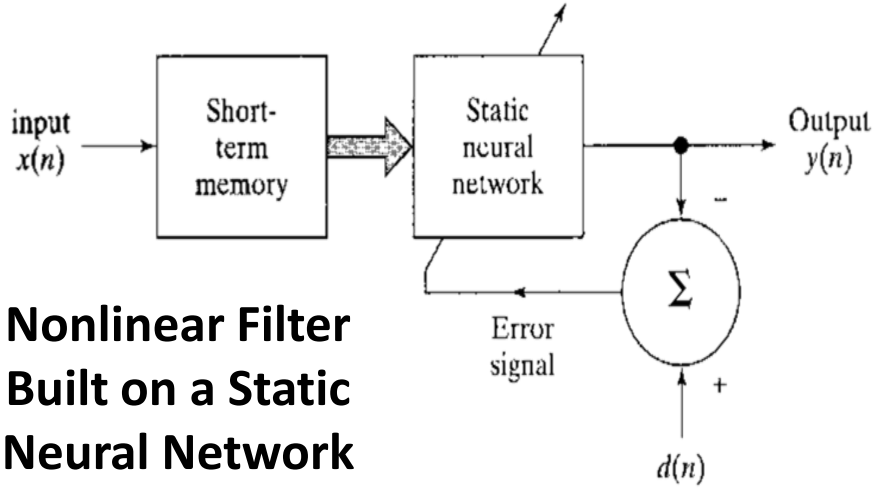
Short-Term Memory Structure

- By embedding memory into the structure of a static network, network output becomes a function of time
  - Static network accounts for nonlinearity
  - Short-term memory accounts for time
- An approach to introducing short-term memory:

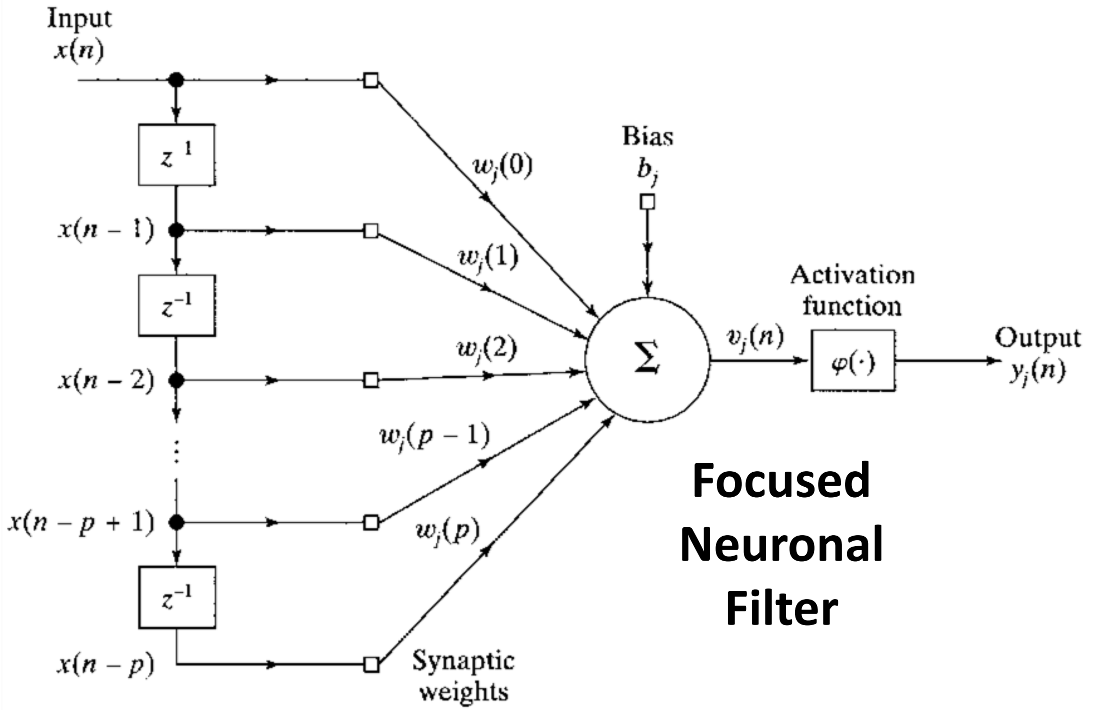
Ordinary Tapped Delay Line Memory of Order p



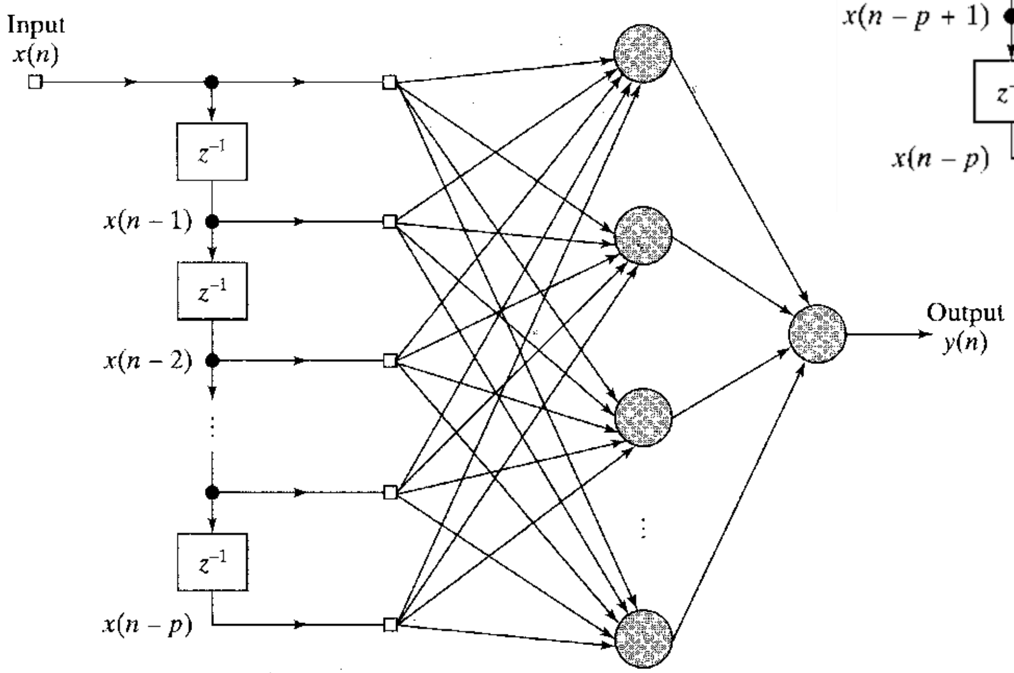
Focused Time-Lagged Feedforward Network (TLFN)



Nonlinear Filter Built on a Static Neural Network



Focused Neuronal Filter



Focused TLFN with Omitted Bias Levels

MLP with SPC Dataset:

We introduced memory by doing exactly this!

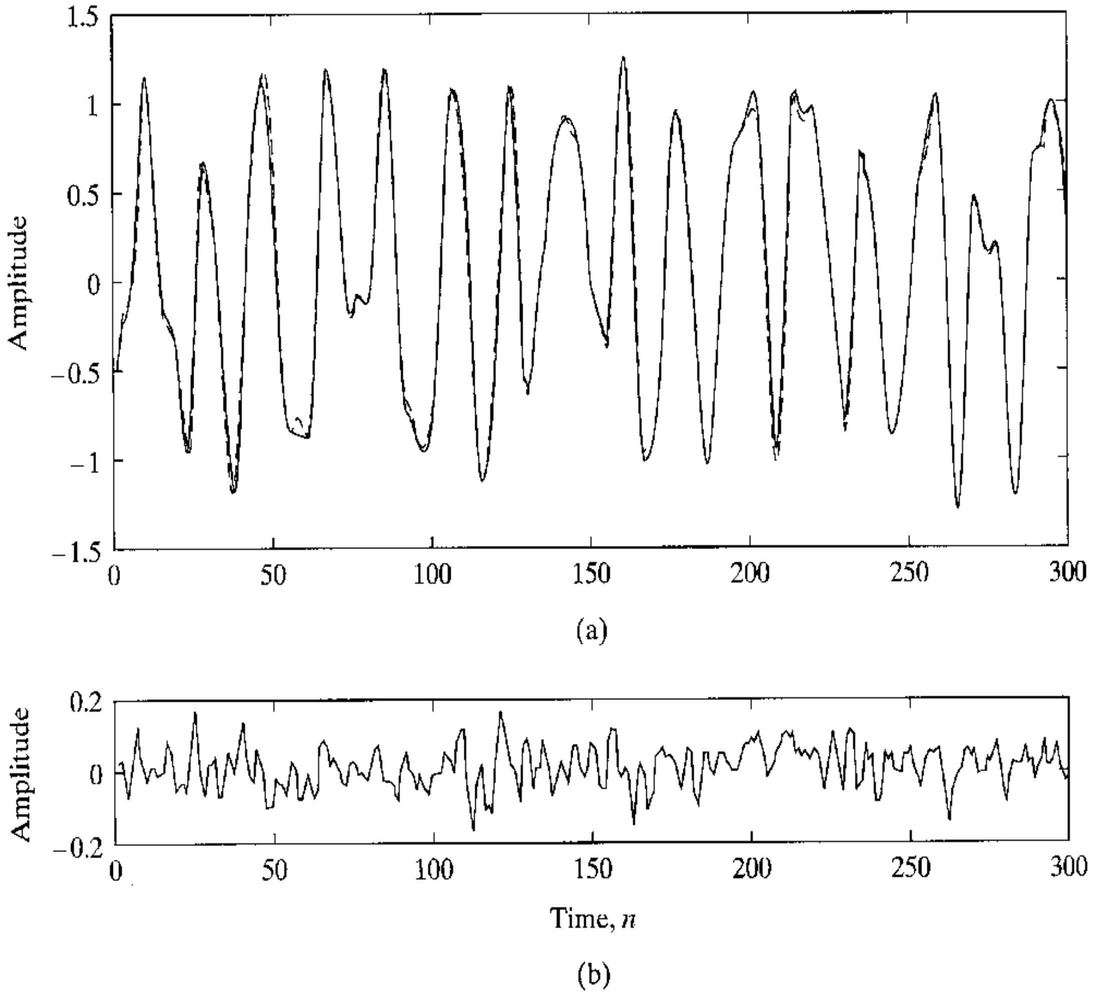
Computer Experiment - Focused TLFN

One-step ahead time-series forecasting of a *frequency modulated* signal:

$$x(n) = \sin(n + \sin(n^2)) \quad n = 0,1,2,\cdots$$

Parameters of Focused TLFN:

Order of tapped delay line memory, p:	20
Hidden layer, m_1:	10 neurons
Activation function of hidden neurons:	logistic
Output layer:	1 neuron
Activation function of output neuron:	linear
Learning rate:	0.01
Momentum:	None

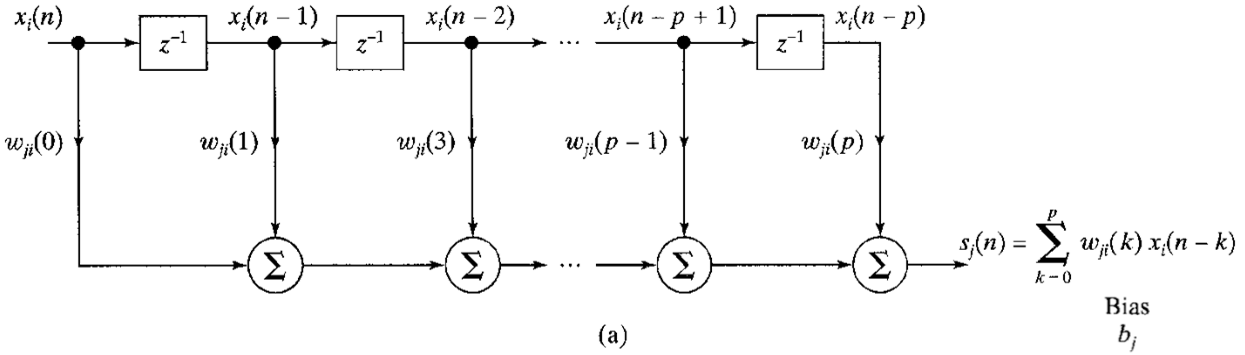


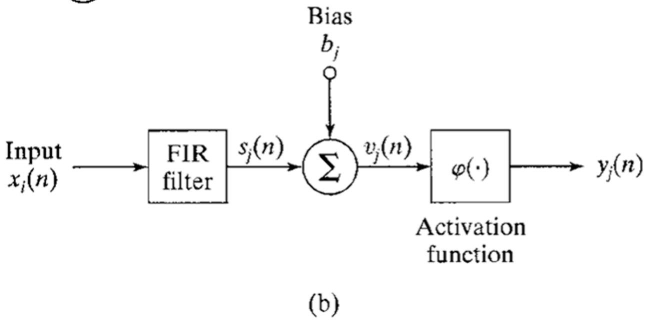
When trained using a Decoupled Extended Kalman Filter (DEKF), Errors further reduced by 90%!

(a) Actual and Predicted (dashed) waveforms.
(b) Waveform of Prediction Error

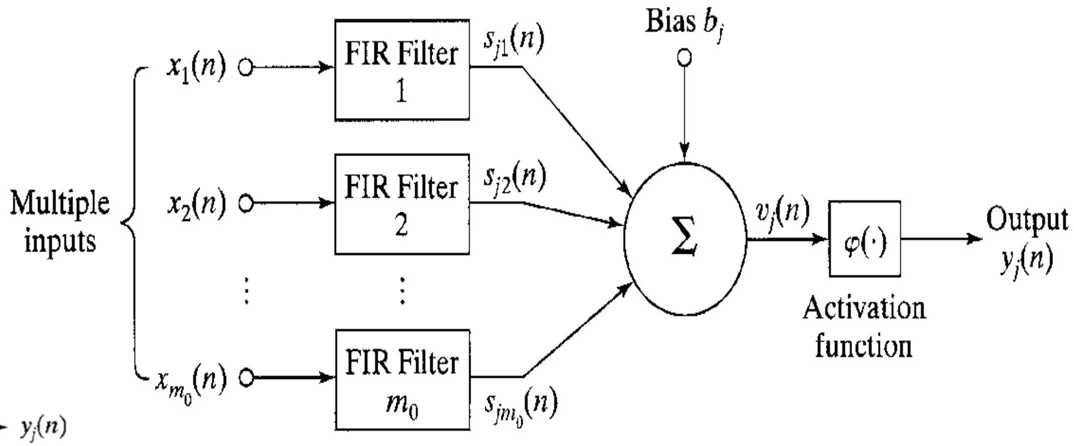
Distributed TLFN

- Universal myopic mapping algorithm, mathematical justification for Focused TLFNs, is limited to maps that are “shift invariant”
  - Implication of shift invariance is that Focused TLFNs are only suitable for use in stationary (i.e., time-invariant) environments
- One can potentially overcome limitation by using a “distributed” TLFN (Eric Wan)
  - Implicit influence of time is distributed throughout network
- Construction of distributed TLFNs is normally based on finite-duration impulse response (FIR) filter as the spatio-temporal model of a neuron



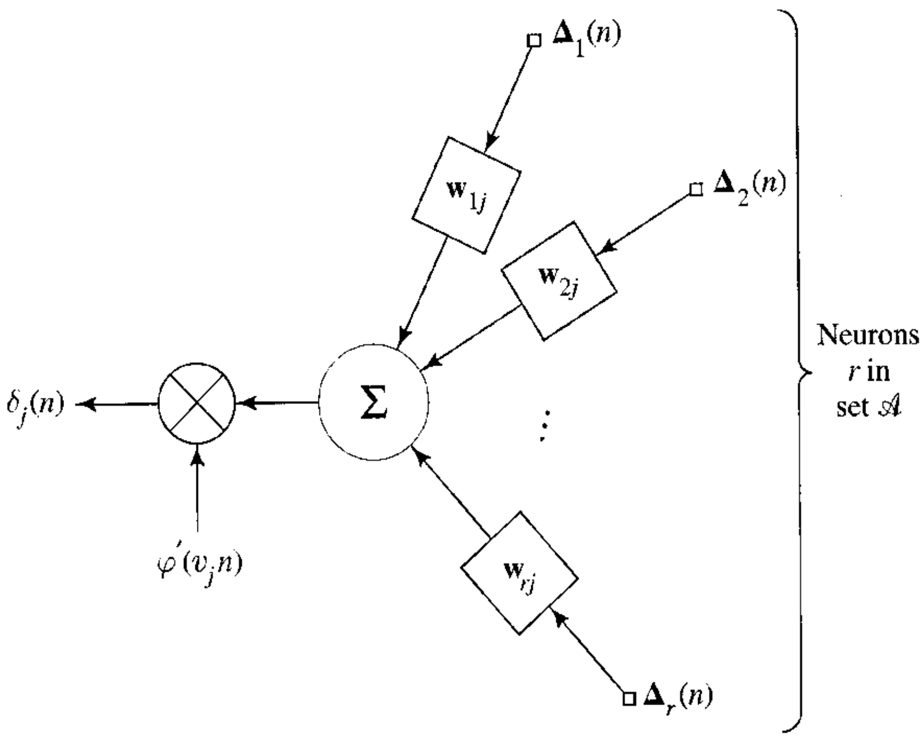


(a) FIR Filter
(b) Neuron as a Nonlinear FIR Filter



Multiple Input Neuronal Filter

Back-Propagation for Distributed TLFN



Back-Propagation of Local Gradients Through a Distributed TLFN

1. Propagate the input signal through the network in the forward direction, layer by layer. Determine the error signal $e_j(n)$ for neuron j in the output layer by subtracting its actual output from the corresponding desired response. Also record the state vector for each synapse in the network.
2. For neuron j in the output layer compute

$$\delta_j(n) = e_j(n)\varphi_j'(n)$$

$$\mathbf{w}_{ji}(n+1) = \mathbf{w}_{ji}(n) + \eta\delta_j(n)\mathbf{x}_i(n)$$

   where $\mathbf{x}_i(n)$ is the state of synapse i of a hidden neuron connected to output neuron j.
3. For neuron j in a hidden layer, compute

$$\delta_j(n-lp) = \varphi'(v_j(n-lp)) \sum_{r\in\mathcal{A}} \Delta_r^T(n-lp)\mathbf{w}_{rj}$$

$$\mathbf{w}_{ji}(n+1) = \mathbf{w}_{ji}(n) + \eta\delta_j(n-lp)\mathbf{x}_i(n-lp)$$

   where p is the order of each synaptic FIR filter, and the index l identifies the hidden layer in question. Specifically, for networks with multiple hidden layers, $l = 1$ corresponds to one layer back from the output layer, $l = 2$ corresponds to two layers back from the output layer, and so on.

Summary of Temporal Back-Propagation Algorithm

SOURCE: Wan, E. A. (1990). Temporal backpropagation for FIR neural networks. *IJCNN International Joint Conference on Neural Networks* (pp. 575-580). IEEE. | PDF

Implementing Focused TLFNs in Matlab

TIMEDELAYNET | LINK

- Syntax: `timedelaynet(inputDelays,hiddenSizes,trainFcn)`
- Arguments:

inputDelays	Row vector of increasing 0 or positive delays (default = 1:2)
hiddenSizes	Row vector of one or more hidden layer sizes (default = 10)
trainFcn	Training function (default = 'trainlm')

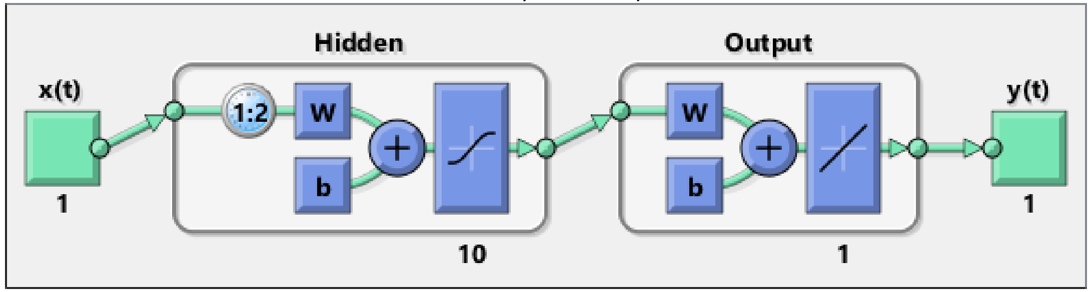
Example

- Partition training set. Use `Xnew` to do prediction in closed loop mode later.

```
[X,T] = simpleseries_dataset;
Xnew = X(81:100);
X = X(1:80);
T = T(1:80);
```

- Train a time delay network, and simulate it on first 80 observations.

```
net = timedelaynet(1:2,10);
[Xs,Xi,Ai,Ts] = preparets(net,X,T);
net = train(net,Xs,Ts,Xi,Ai);
view(net)
```



- Calculate network performance:

```
[Y,Xf,Af] = net(Xs,Xi,Ai);
perf = perform(net,Ts,Y);
```

- Run prediction for 20 timesteps ahead in closed loop mode.

```
[netc,Xic,Aic] = closeloop(net,Xf,Af);
y2 = netc(Xnew,Xic,Aic);
```

Implementing Distributed TLFNs in Matlab

WAYNE STATE UNIVERSITY

DISTDELAYNET | LINK

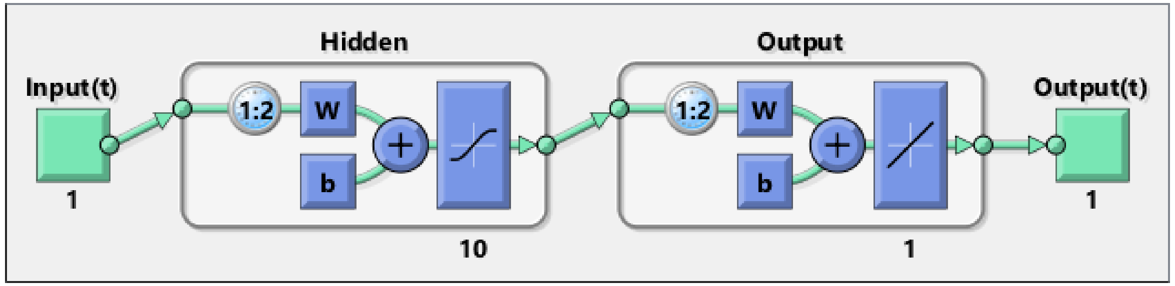
- Syntax: `distdelaynet(delays,hiddenSizes,trainFcn)`
- Arguments:

delays	Row vector of increasing 0 or positive delays (default = 1:2)
hiddenSizes	Row vector of one or more hidden layer sizes (default = 10)
trainFcn	Training function (default = 'trainlm')

Example

```
[X,T] = simpleseries_dataset;
net = distdelaynet({1:2,1:2},10);
[Xs,Xi,Ai,Ts] = preparets(net,X,T);
net = train(net,Xs,Ts,Xi,Ai);
```

```
view(net)
```




```
Y = net(Xs,Xi,Ai);
perf = perform(net,Y,Ts)
perf =
    0.0323
```

Python Implementations:

Check Out Several Options on GitHub for adding TDNN Layers (including Pytorch options)

Link