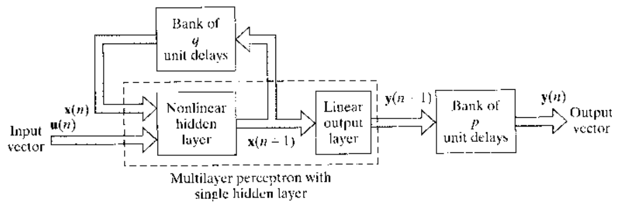
DYNAMICALLY DRIVEN RECURRENT NETWORKS

Primary Source: Haykin, 2009

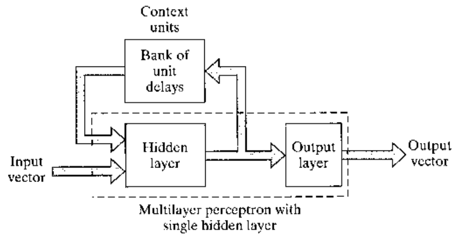
- Recurrent networks are networks with one or more feedback loops
- Feedback can take a variety of forms leading to a rich repertoire of architectural layouts
- There are at least two functional uses for recurrent networks:
  - Associative Memories
  - Input-Output Mapping Networks
- The feed back loops in a recurrent network make the network respond *temporally* to an externally applied input signal, introducing the ability to map dynamic systems
- An issue of primary concern in the study of recurrent networks is that of *stability*
- Recurrent networks offer an alternative to dynamically driven feedforward networks (such as focused TLFNs and distributed TLFNs)
  - Whereas TLFNs employ FIR (finite impulse response) filters for memory, recurrent networks in leverage IIR (infinite impulse response) filters for their performance
- Recurrent networks (because of their use of feedback) may actually fare better in applications such as nonlinear prediction and modeling, speech processing, plant control, and automotive engine diagnostics
  - Feedback enables recurrent networks to acquire state representations
- Use of feedback in recurrent networks also has the potential of reducing the memory requirement significantly (over distributed TLFNs for example), resulting in compact networks [Haykin, 1999]
  - Can lead to more compact networks in terms of number of synaptic weight parameters

RECURRENT NETWORK ARCHITECTURES

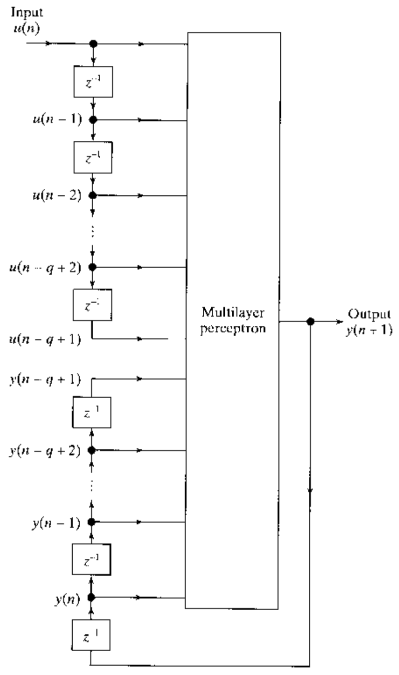
- Four specific recurrent network architectures that incorporate a *static* MLP (or parts thereof) and exploit the nonlinear mapping capability of the MLP are:
  - Nonlinear Autoregressive Exogenous Inputs (NARX) Model
    - Nonlinear Autoregressive Inputs (NAR) Model is a variant
  - State-Space Model
  - Elman Network or Simple Recurrent Network (SRN)
  - Recurrent Multilayer Perceptron (RMLP)



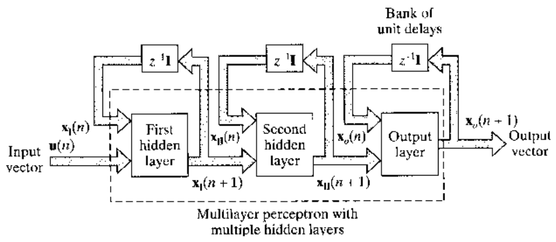
State-Space Model



Elman Network or Simple Recurrent Network



Nonlinear Autoregressive Exogenous Inputs (NARX) Model

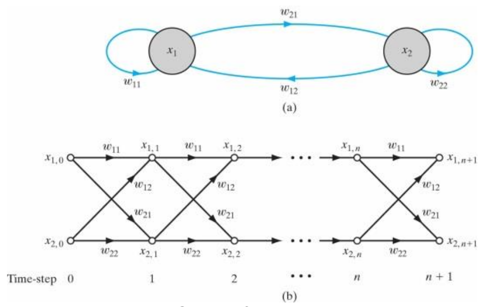


Recurrent Multilayer Perceptron

LEARNING ALGORITHMS

- Recurrent networks can be trained in two modes (Williams and Zipser, 1995):
  - ➢ *Epochwise Training*
    - For a given epoch, the network 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 a different initial state for the next epoch
    - In the current terminology, the epoch for the recurrent network corresponds to one training pattern for the ordinary multilayer perceptron
  - ➢ *Continuous Training*
    - This method is suitable for situations where there are no reset states available and/or on-line learning is required
- Two gradient descent learning algorithms popular for training recurrent networks include: Back-propagation Through Time (BPTT) Algorithm and Real-Time Recurrent Learning (RTRL) Algorithm
  - BPTT requires less computation but more memory than RTRL. Hence, BPTT is more suitable for off-line learning and RTRL for online learning.

Back-propagation Through Time (BPTT) Algorithm



(a) Architectural Graph of 2-Neuron Recurrent Network.
(b) Signal-flow Graph of Network Unfolded in Time.

Once unfolded, train the network using some form of a gradient descent technique!

- BPTT is an extension of the standard back-propagation algorithm
- It is derived by *unfolding* the temporal operation of the network into a layered feedforward network
  - Note that the same synaptic weights are *repeating* themselves across the network
- The network topology grows by one layer for every time-step
- Two options for training: 1) Epochwise BPTT and 2) Truncated BPTT

Epochwise Back-Propagation Through Time

- Let us suppose that the dataset is made up of several independent epochs (e.g., dataset from each cutting tool)
- Let n_0 and n_1 denote the start and end times of the epoch
- Let the cost function for the epoch be defined as:

$$\xi_{\text{Total}} = \frac{1}{2}\sum_{n=n_0}^{n_1}\sum_{j\in\mathcal{H}} e_{j,n}^2$$

where $\mathcal{H}$ is the set of neuron indices j for which desired responses are specified

- Algorithm involves two passes:
  - First, there is a *single forward pass* over the entire epoch data through the network for the interval (n_0, n_1). Complete record of input data, network state, outputs/desired responses are all saved in memory
  - A *single backward pass* over this past record is performed to compute the values of the local gradients

$$\delta_{j,n} = -\frac{\partial \xi_{\text{Total}}}{\partial v_{j,n}} \quad \forall j \text{ and } n_0 < n \le n_1$$

$$\delta_{j,n} = \begin{cases} \varphi'(v_{j,n})e_{j,n} & \text{for } n = n_1 \\ \varphi'(v_{j,n})\left[e_{j,n} + \sum_{k\in\mathcal{H}} w_{kj}\delta_{k,n+1}\right] & \text{for } n_0 < n < n_1 \end{cases}$$

  - This is repeated starting from the last layer (time n_1) and working back to time n_0
  - Once the back propagation is performed back to time $n_0 + 1$, synaptic weights are adjusted:

$$\Delta w_{ji} = -\eta\frac{\partial \xi_{\text{Total}}}{\partial w_{ji}} = \eta \sum_{n=n_0+1}^{n_1} \delta_{j,n}\, x_{i,n-1}$$

Truncated Back-Propagation Through Time

- Truncates the unfolding to a finite depth h to reduce memory requirement
- The adjustments to weights are made on a continuous basis (there is not a full forward pass followed by a backward pass), similar to pattern mode of learning with ordinary back-propagation algorithm
- Employs instantaneous value of the sum of squared errors:

$$\xi_n = \frac{1}{2}\sum_{j\in\mathcal{H}} e_{j,n}^2$$

- This leads to the following local gradient expression:

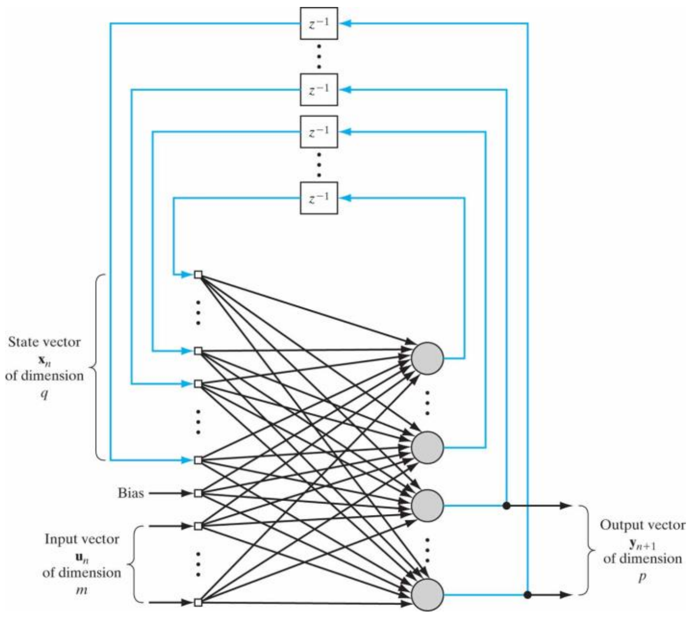
$$\delta_{j,l} = \begin{cases} \varphi'(v_{j,l})e_{j,l} & \text{for } l = n \\ \varphi'(v_{j,n})\left[\cancel{e_{j,n}} + \sum_{k\in\mathcal{H}} w_{kj,l}\delta_{k,l+1}\right] & \text{for } n-h < l < n \end{cases}$$

- Weight updates are as follows:

$$\Delta w_{ji,n} = \eta \sum_{l=n-h+1}^{n} \delta_{j,l}\, x_{i,l-1}$$

Real-Time Recurrent Learning (RTRL) Algorithm

- Adjustments are made to synaptic weights in real time (while the network is performing its function)
- It is derived here for the *state-space* model


z^{-1}
z^{-1}
z^{-1}
z^{-1}
State vector $\mathbf{x}_n$ of dimension q
Bias
Input vector $\mathbf{u}_n$ of dimension m
Output vector $\mathbf{y}_{n+1}$ of dimension p

Fully Connected Recurrent Network for Formulation of the RTRL Algorithm. (Feedback Connections in Blue)

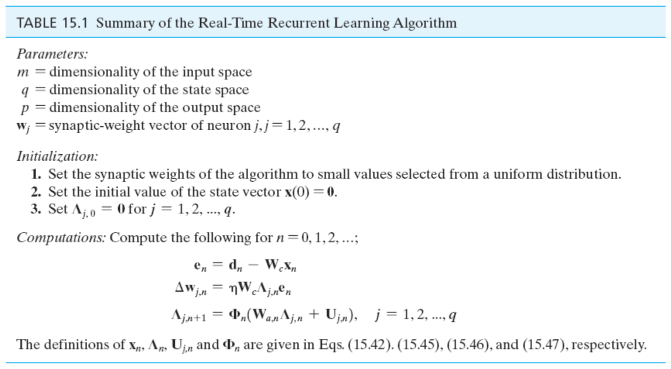


TABLE 15.1 Summary of the Real-Time Recurrent Learning Algorithm

Parameters:
m = dimensionality of the input space
q = dimensionality of the state space
p = dimensionality of the output space
$\mathbf{w}_j$ = synaptic-weight vector of neuron j, $j = 1, 2, \ldots, q$

Initialization:
1. Set the synaptic weights of the algorithm to small values selected from a uniform distribution.
2. Set the initial value of the state vector $\mathbf{x}(0) = \mathbf{0}$.
3. Set $\mathbf{\Lambda}_{j,0} = \mathbf{0}$ for $j = 1, 2, \ldots, q$.

Computations: Compute the following for $n = 0, 1, 2, \ldots$;

$$\mathbf{e}_n = \mathbf{d}_n - \mathbf{W}_c\mathbf{x}_n$$
$$\Delta\mathbf{w}_{j,n} = \eta\mathbf{W}_c\mathbf{\Lambda}_{j,n}\mathbf{e}_n$$
$$\mathbf{\Lambda}_{j,n+1} = \mathbf{\Phi}_n(\mathbf{W}_{a,n}\mathbf{\Lambda}_{j,n} + \mathbf{U}_{j,n}), \quad j = 1, 2, \ldots, q$$

The definitions of $\mathbf{x}_n$, $\mathbf{\Lambda}_n$, $\mathbf{U}_{j,n}$ and $\mathbf{\Phi}_n$ are given in Eqs. (15.42). (15.45), (15.46), and (15.47), respectively.

- RTRL suffers heavily from vanishing-gradients problem
- Second-order methods perform better over steepest descent methods

- The above first-order gradient descent methods can be accelerated by exploiting Kalman filter theory, which utilizes information contained in the training data more effectively

TABLE 15.2 Summary of the EKF algorithm for Supervised Training of the RMLP

Training sample:

$\mathcal{T} = \{\mathbf{u}_n, \mathbf{d}_n\}_{n=1}^{N}$

where $\mathbf{u}_n$ is the input vector applied to the RMLP and $\mathbf{d}_n$ is the corresponding desired response.

RMLP and Kalman filter: parameters and variables

$\mathbf{b}(\cdot,\cdot,\cdot)$	:	vector-valued measurement function
$\mathbf{B}$	:	linearized measurement matrix
$\mathbf{w}_n$	:	weight vector at time-step n
$\hat{\mathbf{w}}_{n\vert n-1}$	:	predicted estimate of the weight vector
$\hat{\mathbf{w}}_{n\vert n}$	:	filtered estimate of the weight vector
$\mathbf{v}_n$	:	vector of recurrent node activities in the RMLP
$\mathbf{y}_n$	:	output vector of the RMLP produced in response to the input vector $\mathbf{u}_n$
$\mathbf{Q}_\omega$	:	covariance matrix of the dynamic noise $\boldsymbol{\omega}_n$
$\mathbf{Q}_\nu$	:	covariance matrix of the measurement noise $\boldsymbol{\nu}_n$
$\mathbf{G}_n$	:	Kalman gain
$\mathbf{P}_{n\vert n-1}$	:	prediction-error covariance matrix
$\mathbf{P}_{n\vert n}$	:	filtering-error covariance matrix

Computation:

For $n = 1, 2, \ldots,$ compute the following:

$$\mathbf{G}_n = \mathbf{P}_{n|n-1}\mathbf{B}_n^T[\mathbf{B}_n\mathbf{P}_{n|n-1}\mathbf{B}_n^T + \mathbf{Q}_{\nu,n}]^{-1}$$
$$\boldsymbol{\alpha}_n = \mathbf{d}_n - \mathbf{b}_n(\hat{\mathbf{w}}_{n|n-1}, \mathbf{v}_n, \mathbf{u}_n)$$
$$\hat{\mathbf{w}}_{n|n} = \hat{\mathbf{w}}_{n|n-1} + \mathbf{G}_n\boldsymbol{\alpha}_n$$
$$\hat{\mathbf{w}}_{n+1|n} = \hat{\mathbf{w}}_{n|n}$$
$$\mathbf{P}_{n|n} = \mathbf{P}_{n|n-1} - \mathbf{G}_n\mathbf{B}_n\mathbf{P}_{n|n-1}$$
$$\mathbf{P}_{n+1|n} = \mathbf{P}_{n|n} + \mathbf{Q}_{\omega,n}$$

Initilization:

$\hat{\mathbf{w}}_{1|0} = \mathbb{E}[\mathbf{w}_1]$

$\mathbf{P}_{1|0} = \delta^{-1}\mathbf{I}$, where δ is a small positive constant and $\mathbf{I}$ is the identity matrix

COMPUTER EXPERIMENT – RMLP AND FOCUSSED TLFN

One-step ahead time series forecasting of the following frequency modulated signal:

$$x(n) = \sin(n + \sin(n^2)), \qquad n = 0, 1, 2, \ldots$$

Parameters of RMLP Network:

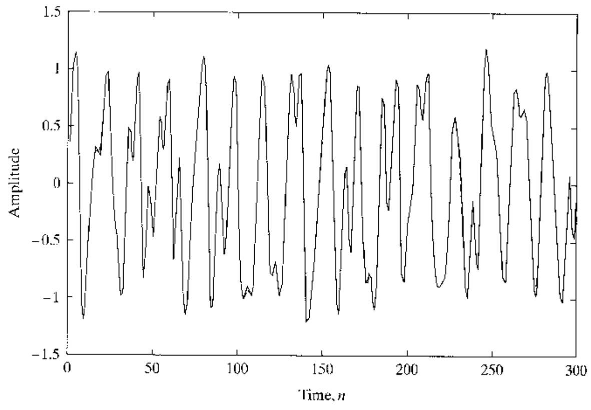
- One input node
- First hidden layer of 10 neurons
- One linear output neuron

Parameters of the Focussed TLFN:

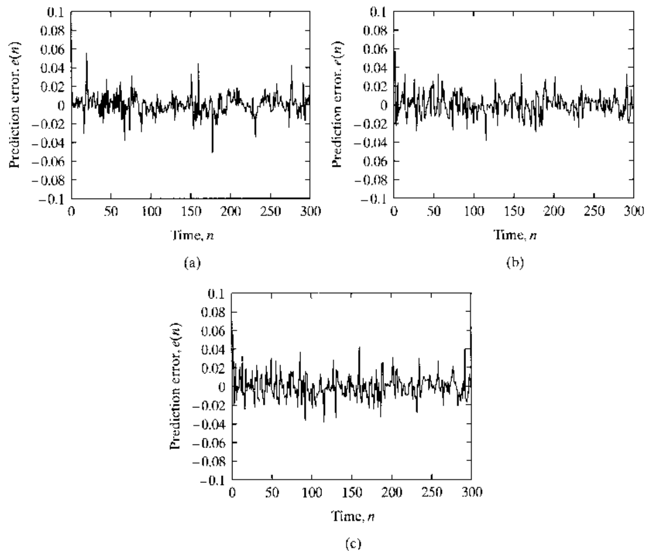
- One input node with 20 taps
- First hidden layer of 10 neurons
- One linear output neuron

Note: RMLP has slightly more synaptic weights than the focused TLFN, but half the memory (10 recurrent nodes versus 20 taps)

Results:



Superposition of Actual (continuous) and Predicted (dashed) Waveforms for the RMLP Trained Using RTRL and DEKF.



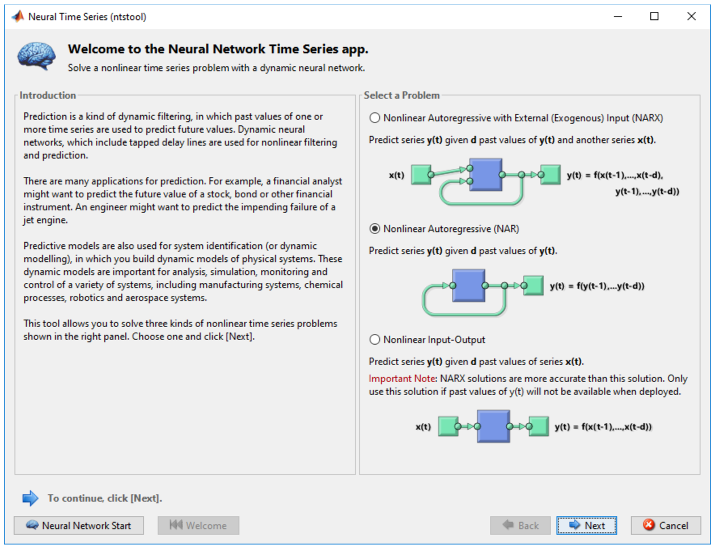
Prediction Error Waveforms for Three Different Simulations: (a) RMLP with RTRL and DEKF Training, Error Variance = 1.1839 x 10^{-4}. (b) Focused TLFN with GEKF Training, Error Variance = 1.3351 x 10^{-4}. (c) Focused TLFN with DEKF Training, Error Variance = 1.5871 x 10^{-4}.

% TIME-SERIES PREDICTION USING MATLAB NEURALNET TOOLBOX

Solve time-series problems using dynamic neural networks, including networks with feedback

Apps

Neural Net Time Series	Solve a nonlinear time series problem by training a dynamic neural network


Neural Time Series (ntstool)
x(t)
y(t) = f(x(t-1),...,x(t-d), y(t-1),...,y(t-d))
y(t) = f(y(t-1),...y(t-d))
y(t) = f(x(t-1),...,x(t-d))
Neural Network Start
Welcome
Back
Next
Cancel

Welcome to the Neural Network Time Series app.

Solve a nonlinear time series problem with a dynamic neural network.

Introduction

Prediction is a kind of dynamic filtering, in which past values of one or more time series are used to predict future values. Dynamic neural networks, which include tapped delay lines are used for nonlinear filtering and prediction.

There are many applications for prediction. For example, a financial analyst might want to predict the future value of a stock, bond or other financial instrument. An engineer might want to predict the impending failure of a jet engine.

Predictive models are also used for system identification (or dynamic modelling), in which you build dynamic models of physical systems. These dynamic models are important for analysis, simulation, monitoring and control of a variety of systems, including manufacturing systems, chemical processes, robotics and aerospace systems.

This tool allows you to solve three kinds of nonlinear time series problems shown in the right panel. Choose one and click [Next].

Select a Problem

○ Nonlinear Autoregressive with External (Exogenous) Input (NARX)

Predict series **y(t)** given **d** past values of **y(t)** and another series **x(t)**.

● Nonlinear Autoregressive (NAR)

Predict series **y(t)** given **d** past values of **y(t)**.

○ Nonlinear Input-Output

Predict series **y(t)** given **d** past values of series **x(t)**.

Important Note: NARX solutions are more accurate than this solution. Only use this solution if past values of y(t) will not be available when deployed.

To continue, click [Next].

% OVERVIEW
% Input-output time series problems consist of predicting the next value
% of one time-series given another time-series. Past values of both series
% (for best accuracy), or only one of the series (for a simpler system)
% may be used to predict the target series.

% SAMPLE DATASET
% simplenarx_dataset -- Simple time-series prediction dataset
% This dataset can be used to demonstrate how a neural network can be trained to make predictions.

```
% LOAD DATASET
load simplenarx_dataset.MAT
% Loads these two variables:
% simplenarxInputs - a 1x100 cell array of scalar values representing a
100 timestep time-series.
% simplenarxTargets - a 1x100 cell array of scalar values representing a
100 timestep time-series to be predicted.

% Load the inputs and targets into variables of your own choosing
[X,T] = simplenarx_dataset;

% CREATE AND TRAIN "NARXNET"
% Here is how to design a neural network that predicts the target series
from using past values of inputs as
% well as past values of outputs. See "narxnet" for more details.

net = narxnet(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)
plotresponse(Ts,Y)

% CREATE AND TRAIN "TIMEDELAYNET"
% Here is how to design a neural network that predicts the target series
from only using past values of inputs. % See "timedelaynet" for details.

net = timedelaynet(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)
plotresponse(Ts,Y)

% CREATE AND TRAIN "NARNET" - Nonlinear AutoRegressive Network
% Here is how to design a neural network that predicts the targets series
only using past values of the target
% series. See narnet for details.

net = narnet(1:2,10);
[Xs,Xi,Ai,Ts] = preparets(net,{},{},T);
net = train(net,X,T,Xi,Ai);
view(net)
Y = net(Xs,Xi,Ai)
plotresponse(Ts,Y)

% CREATE AND TRAIN "LAYRECNET" – Recurrent
Network
% See "layrecnet" for more details.
% [X,T] = simpleseries_dataset;
net = layrecnet(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)

%See also: ntstool, narxnet, timedelaynet, narnet, preparets, nndatasets
```

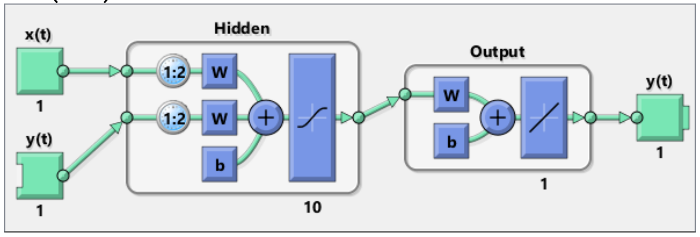
Multistep Neural Network Prediction

Set Up in Open-Loop Mode

Dynamic networks with feedback, such as `narxnet` and `narnet` neural networks, can be transformed between open-loop and closed-loop modes with the functions `openloop` and `closeloop`. Closed-loop networks make multistep predictions. In other words they continue to predict when external feedback is missing, by using internal feedback.

Here a neural network is trained to model the magnetic levitation system and simulated in the default open-loop mode.

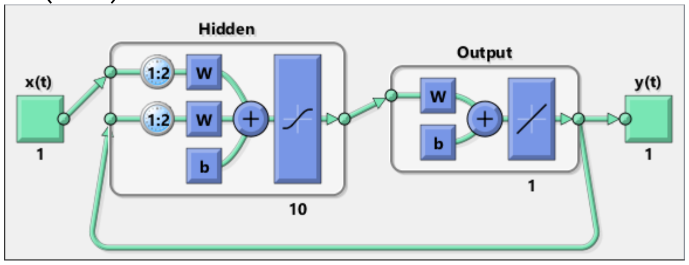
```
[X,T] = maglev_dataset;
net = narxnet(1:2,1:2,10);
[x,xi,ai,t] = preparets(net,X,{},T);
net = train(net,x,t,xi,ai);
y = net(x,xi,ai);
view(net)
```



Multistep Closed-Loop Prediction from Initial Conditions

A neural network can also be simulated only in closed-loop form, so that given an external input series and initial conditions, the neural network performs as many predictions as the input series has time steps.

```
netc = closeloop(net);
view(netc)
```



Here the training data is used to define the inputs `x`, and the initial input and layer delay states, `xi` and `ai`, but they can be defined to make multiple predictions for any input series and initial states.

```
[x,xi,ai,t] = preparets(netc,X,{},T);
yc = netc(x,xi,ai);
```

Multistep Closed-Loop Prediction Following Known Sequence

It can also be useful to simulate a trained neural network up the present with all the known values of a time-series in open-loop mode, then switch to closed-loop mode to continue the simulation for as many predictions into the future as are desired.

Just as openloop and closeloop can be used to transform between open- and closed-loop neural networks, they can convert the state of open- and closed-loop networks. Here are the full interfaces for these functions.

```
        [open_net,open_xi,open_ai] =
        openloop(closed_net,closed_xi,closed_ai);

        [closed_net,closed_xi,closed_ai] =
        closeloop(open_net,open_xi,open_ai);
```

Consider the case where you might have a record of the Maglev's behavior for 20 time steps, and you want to predict ahead for 20 more time steps.

First, define the first 20 steps of inputs and targets, representing the 20 time steps where the known output is defined by the targets `t`. With the next 20 time steps of the input are defined, use the network to predict the 20 outputs using each of its predictions feedback to help the network perform the next prediction.

```
x1 = x(1:20);
t1 = t(1:20);
x2 = x(21:40);
```

The open-loop neural network is then simulated on this data.

```
[x,xi,ai,t] = preparets(net,x1,{},t1);
[y1,xf,af] = net(x,xi,ai);
```

Now the final input and layer states returned by the network are converted to closed-loop form along with the network. The final input states `xf` and layer states `af` of the open-loop network become the initial input states `xi` and layer states `ai` of the closed-loop network.

```
[netc,xi,ai] = closeloop(net,xf,af);
```

Typically use preparets to define initial input and layer states. Since these have already been obtained from the end of the open-loop simulation, you do not need `preparets` to continue with the 20 step predictions of the closed-loop network.

```
[y2,xf,af] = netc(x2,xi,ai);
```

Note that you can set `x2` to different sequences of inputs to test different scenarios for however many time steps you would like to make predictions. For example, to predict the magnetic levitation system's behavior if 10 random inputs are used:

```
x2 = num2cell(rand(1,10));
[y2,xf,af] = netc(x2,xi,ai);
```

Following Closed-Loop Simulation with Open-Loop Simulation

If after simulating the network in closed-loop form, you can continue the simulation from there in open-loop form. Here the closed-loop state is converted back to open-loop state. (You do not have to convert the network back to open-loop form as you already have the original open-loop network.)

```
[~,xi,ai] = openloop(netc,xf,af);
```

Now you can define continuations of the external input and open-loop feedback, and simulate the open-loop network.

```
x3 = num2cell(rand(2,10));
y3 = net(x3,xi,ai);
```

In this way, you can switch simulation between open-loop and closed-loop manners. One application for this is making time-series predictions of a sensor, where the last sensor value is usually known, allowing open-loop prediction of the next step. But on some occasions the sensor reading is not available, or known to be erroneous, requiring a closed-loop prediction step. The predictions can alternate between open-loop and closed-loop form, depending on the availability of the last step's sensor reading.

LAYRECNET

Layer recurrent neural network

Syntax

```
    layrecnet(layerDelays,hiddenSizes,trainFcn)
```

Description

Layer recurrent neural networks are similar to feedforward networks, except that each layer has a recurrent connection with a tap delay associated with it. This allows the network to have an infinite dynamic response to time series input data. This network is similar to the time delay (timedelaynet) and distributed delay (distdelaynet) neural networks, which have finite input responses.

`layrecnet(layerDelays,hiddenSizes,trainFcn)` takes these arguments,

`layerDelays`	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'`)

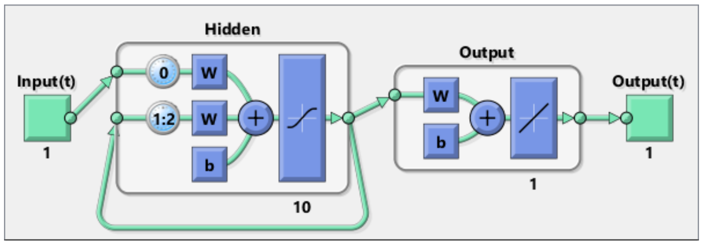
and returns a layer recurrent neural network.

Examples

Recurrent Neural Network

Use a layer recurrent neural network to solve a simple time series problem.

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

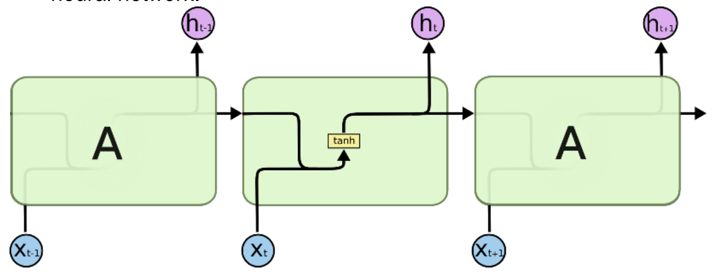


```
Y = net(Xs,Xi,Ai);
perf = perform(net,Y,Ts)
perf =
   6.1239e-11
```

Long Short-Term Memory (LSTM) Networks[*]

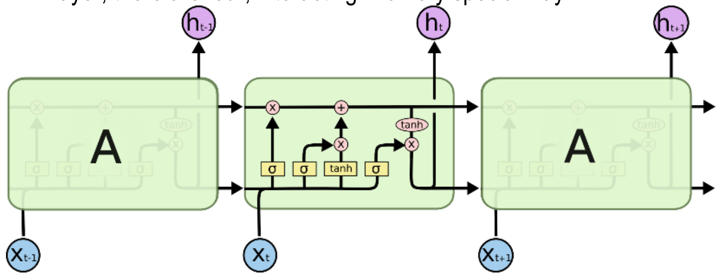
- Humans don't start their thinking from scratch every second; our thoughts have persistence.
  - Example: If you are reading a novel about a vacation to France, entry into France might be discussed in the first chapter but we remember that as we read the entire novel ("**context**" is **maintained naturally**).
  - Traditional neural networks cannot do this, and it seems like a major shortcoming
  - While general recurrent networks attempt this, they do not manage it explicitly
  - Another name for this is ***stability-plasticity dilemma***:
  - Well-known constraint for artificial and biological neural systems
  - Basic idea is that learning requires **plasticity for the integration of new knowledge**, but also **stability to prevent the forgetting of previous knowledge**.
- In the last few years, there have been incredible success applying RNNs to a variety of problems: speech recognition, language modeling, translation, image captioning…
- Check out the discussion of the amazing feats one can achieve with RNNs to Andrej Karpathy's excellent blog post, The Unreasonable Effectiveness of Recurrent Neural Networks.
- Essential to these successes is the use of "LSTMs," which works, for many tasks, much better than the standard version.
  - Almost all exciting results based on RNNs are achieved with them. – Colah (2015)
- Consider trying to predict the last word in the text "I grew up in France… I speak fluent *French*."
  - Recent information suggests that the next word is probably the name of a language, but if we want to narrow down which language, we need the context of France, from further back.
  - It is entirely possible for the gap between the relevant information and the point where it is needed to become very large.
  - Unfortunately, as that gap grows, RNNs become unable to learn to connect the information.
- An LSTM network is a ***type of recurrent neural network*** (RNN) that can learn long-term dependencies between time steps of sequence data, **seeking better "context" management**
- **LSTMs are explicitly designed to avoid the long-term dependency problem.**
  - **Remembering information for prolonged periods of time is practically their default behavior**, not something they struggle to learn!

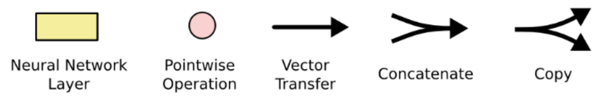
- **All RNNs have the form of a chain of repeating modules** of neural network.



Repeating module in a standard RNN contains a simple single layer

- **LSTMs also have this chain like structure**, but **repeating module has a different structure.** Instead of having a single neural network layer, there are four, interacting in a very special way.



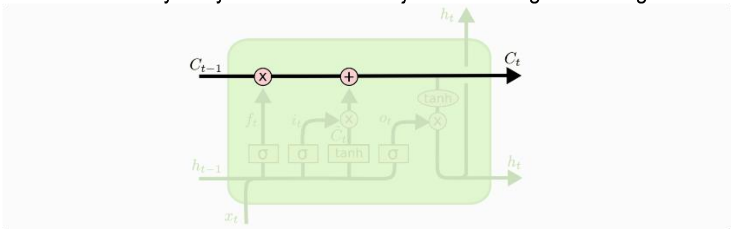


Repeating module in an LSTM contains four interacting layers

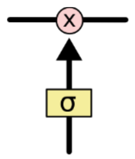
- Pink circles represent pointwise operations, like vector addition, while the yellow boxes are learned neural network layers. Lines merging denote concatenation, while a line forking denotes its content being copied and the copies going to different locations.

Core Idea Behind LSTMs

- **Key to LSTMs is the "cell" state (maintaining context)**, the horizontal line running through the top of the diagram.
  - **Cell state is kind of like a conveyor belt.** It runs straight down the entire chain, with only some minor linear interactions. It's very easy for information to just flow along it unchanged.



- **LSTM does have the ability to remove or add information to the cell state, carefully regulated by structures called gates.**
- **Gates are a way to optionally let information through.** They are composed out of a sigmoid neural net layer and a pointwise multiplication operation.

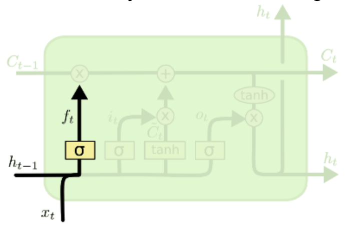


  - **Sigmoid layer outputs numbers between zero and one, describing how much of each component should be let through.** A value of zero means "let nothing through," while a value of one means "let everything through!"
- An LSTM has three of these gates, to protect and control the cell state.

[*] Primary Source: Colah's Blog: Understanding LSTM Networks LINK

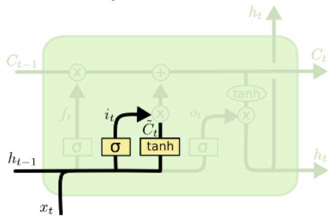
Step-by-Step LSTM Walk Through

- **First step in LSTM is to decide what information to throw away from the cell state.**
  - Decision is made by a sigmoid layer called the "**forget gate layer**."
  - It looks at h_{t-1} and x_t, and outputs a number between 0 and 1 for each number in the cell state C_{t-1}.
  - 1 represents "completely keep this" while a 0 represents "completely get rid of this."
  - Example: In the context of the language model example of trying to predict the next word based on all the previous ones, the cell state might include the gender of the present subject, so that the correct pronouns can be used. When we see a new subject, we want to forget the gender of the old subject.



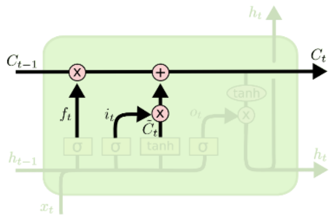
$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \ + \ b_f\right)$$

- **Next step is to decide what new information to store in the cell state and has two parts.**
  - First, a sigmoid layer called the "**input gate layer**" decides which values we will update.
  - Next, a $tanh$ layer creates a vector of **new candidate values**, $\tilde{C}_t$, that could be added to the state. In the next step, we will combine these two to create an update to the state.
  - Example: In the example of our language model, we would want to add the gender of the new subject to the cell state, to replace the old one we are forgetting.



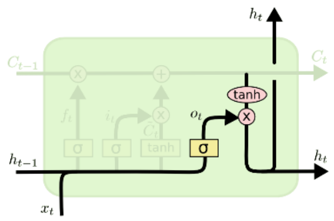
$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \ + \ b_i\right)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \ + \ b_C)$$

- **It is now time to update the old cell state**, C_{t-1}, into the new cell state C_t. The previous steps already decided what to do, we just need to actually do it.
  - We multiply the old state by f_t, forgetting the things we decided to forget earlier.
  - Then we add $i_t * \tilde{C}_t$. This is the new candidate values, scaled by how much we decided to update each state value.
  - In the case of the language model, this is where we would actually drop the information about the old subject's gender and add the new information.



$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

- Finally, we need to decide what we are going to output.
  - This output will be based on our cell state but will be a filtered version.
  - First, we run a sigmoid layer which decides what parts of the cell state we are going to output.
  - Then, we put the cell state through $tanh$ (to push the values to be between −1 and 1) and multiply it by the output of the sigmoid gate, so that we only output the parts we decided to.
  - For the language model example, since it just saw a subject, it might want to output information relevant to a verb, in case that's what is coming next. For example, it might output whether the subject is singular or plural, so that we know what form a verb should be conjugated into if that's what follows next.



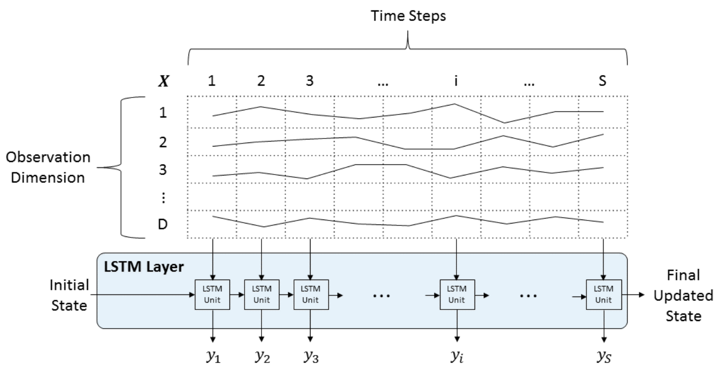
$$o_t = \sigma\left(W_o \ [h_{t-1}, x_t] \ + \ b_o\right)$$
$$h_t = o_t * \tanh\left(C_t\right)$$

Variants on Long Short-Term Memory

- What we have described so far is a pretty normal LSTM. But not all LSTMs are the same as the above. In fact, it seems like almost every paper involving LSTMs uses a slightly different version. The differences are minor, but it's worth mentioning some of them.

Implementing LSTM Networks in Matlab

- **Layers in an LSTM Network**
  - *Sequence Input Layer*
    - A sequence input layer inputs sequence data to the network.
    - In Matlab, one can create a sequence input layer using sequenceInputLayer.
  - *LSTM Layer*
    - LSTM layer is a RNN layer that enables support for time series/sequence data.
    - Layer performs additive interactions, which can help improve gradient flow over long sequences during training.
    - Suitable for situations where there are no reset states and/or on-line learning is required.
    - In Matlab, one can create an LSTM layer using lstmLayer.
- **LSTM Layer Architecture**
  - Figure illustrates flow of a time series X with D features of length S through an LSTM layer.



  - The first LSTM unit takes the initial network state and the first time step of the sequence to make a prediction, and outputs the updated network state to the next LSTM unit.

  - Each LSTM unit takes the updated network state from the previous unit and outputs a prediction and a new updated network state.
- **LSTM Network State**
  - LSTM networks can remember the state of the network between predictions.
  - The network state is useful for when you do not have the complete time series in advance, or if you want to make multiple predictions on a long time series.
  - In Matlab, to predict and classify on parts of a time series and update the network state, you can use predictAndUpdateState and classifyAndUpdateState. To reset the network state between predictions, use resetState.

MATLAB EXAMPLE OF LSTM: RECOGNIZE SPEAKER FROM SPEECH

Try in Matlab: Run this command in Matlab Command Window

```
openExample('nnet/ClassifySequenceDataUsingLSTMNetworksExa
mple')
```

- Example uses the Japanese Vowels data set as described by Kudp *et al.*[1].
- Trains an LSTM network to recognize the speaker given time series data representing two Japanese vowels spoken in succession.
- Training data contains time series data for nine speakers.
- Each sequence has 12 features and varies in length.
- The data set contains 270 training observations and 370 test observations.

Load Sequence Data

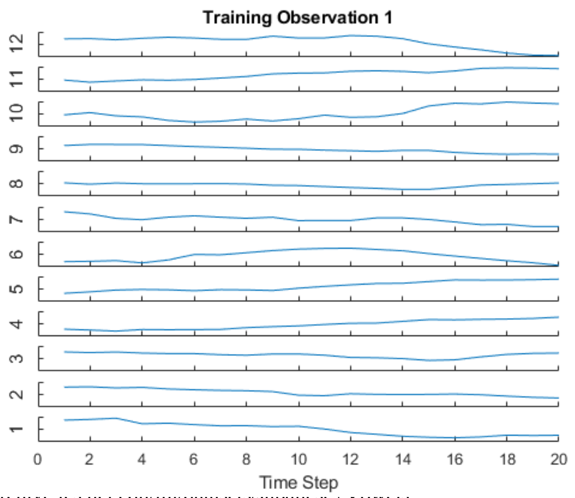
- Load the Japanese Vowels training data:
  - X is a cell array containing 270 sequences of dimension 12 of varying length.
  - Y is a categorical vector of labels "1","2",...,"9", which correspond to the nine speakers.

```
load JapaneseVowelsTrain
```

- View the first five observations. The entries in X are matrices with 12 rows (one row for each feature) and varying number of columns (one column for each time step).

```
X(1:5)
ans = 5×1 cell array
    {12×20 double}
    {12×26 double}
    {12×22 double}
    {12×20 double}
    {12×21 double}
```

- Visualize the first time series in a plot. Each subplot corresponds to a feature.



* M
Pa
11
https://archive.ics.uci.edu/ml/datasets/Japanese+Vowels

```
figure
for i = 1:12
    subplot(12,1,13-i)
    plot(X{1}(i,:));
    ylabel(i)
    xticklabels('')
    yticklabels('')
    box off
end
title("Training Observation 1")
subplot(12,1,12)
xticklabels('auto')
xlabel("Time Step")
```

Sort Data by Sequence Length

- During training, the software splits the training data into mini-batches and pads or truncates the sequences so that they have the same length. Too much padding or discarding of data can have a negative impact on the network performance.
- To prevent the training process from adding too much padding, you can sort the training data by sequence length, and choose a mini-batch size so that sequences in a mini-batch have a similar length.
- Get the sequence lengths for each observation.

```
numObservations = numel(X);
for i=1:numObservations
    sequence = X{i};
    sequenceLengths(i) = size(sequence,2);
end
```

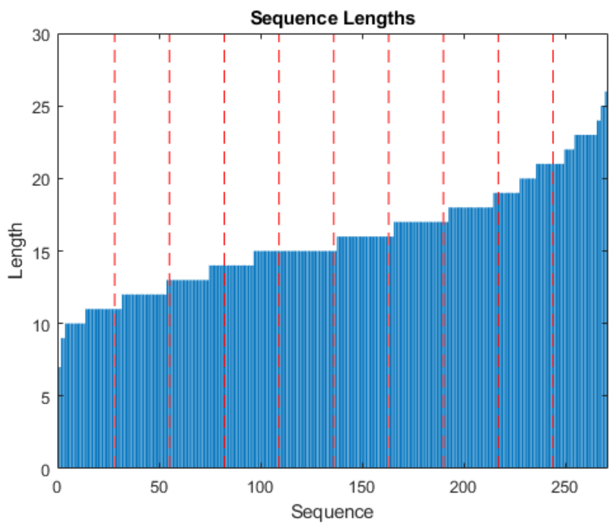
- Sort the data by sequence length.

```
[sequenceLengths,idx] =
           sort(sequenceLengths);
X = X(idx);
Y = Y(idx);
```

- View the sorted sequence lengths in a bar chart.

```
figure
bar(sequenceLengths)
ylim([0 30])
xlabel("Sequence")
ylabel("Length")
title("Sequence Lengths")
```

- Choose a mini-batch size of 27 to divide the training data evenly



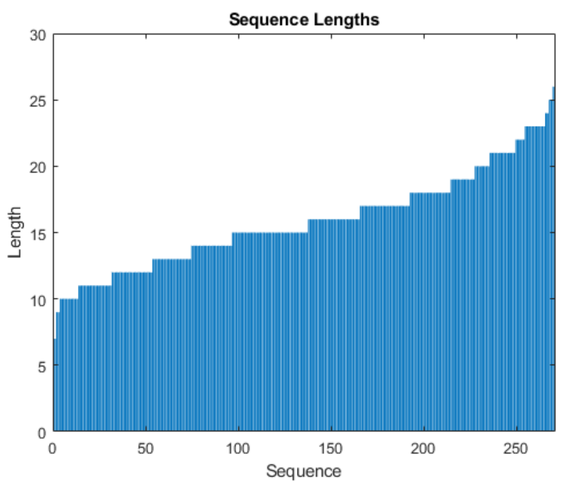
and reduce the amount of padding in the mini-batches.
- Determine the mini-batch boundaries.

```
miniBatchSize = 27;
miniBatchLocations =
miniBatchSize+1:miniBatchSize:numObservations;
XLocations = repmat(miniBatchLocations,[2 1]);
YLocations = repmat([0;30],[1 9]);
```

- Plot the mini-batch boundaries with the sequence lengths. The plot shows how sequences are divided into mini-batches.

```
hold on
line(XLocations,YLocations, ...
      'Color','r', ...
```

```
    'LineStyle','--')
```



Define LSTM Network Architecture

- Define the LSTM network architecture. Specify the input size to be sequences of size 12 (the dimension of the input data). Specify an LSTM layer with an output size of 100, and output the last element of the sequence. Finally, specify nine classes by including a fully connected layer of size 9, followed by a softmax layer and a classification layer.

```
inputSize = 12;
outputSize = 100;
outputMode = 'last';
numClasses = 9;

layers = [ ...
    sequenceInputLayer(inputSize)
    lstmLayer(outputSize,'OutputMode',outputMode)
    fullyConnectedLayer(numClasses)
    softmaxLayer
    classificationLayer]
layers =
  5x1 Layer array with layers:

     1   ''   Sequence Input          Sequence input with
12 dimensions
     2   ''   LSTM                    LSTM with 100 hidden
units
     3   ''   Fully Connected         9 fully connected
layer
     4   ''   Softmax                 softmax
     5   ''   Classification Output   crossentropyex
```

- Now, specify the training options. Choose a mini-batch size of 27 to reduce the amount of padding in the mini-batches. Set the maximum number of epochs to 150, and specify to not shuffle the data.

```
maxEpochs = 150;
miniBatchSize = 27;
shuffle = 'never';

options = trainingOptions('sgdm', ...
    'MaxEpochs',maxEpochs, ...
    'MiniBatchSize',miniBatchSize, ...
    'Shuffle', shuffle);
```

Train LSTM Network

- Train the LSTM network with the specified training options by using trainNetwork.

```
net = trainNetwork(X,Y,layers,options);
Training on single GPU.
|=========================================================================================|
|     Epoch    |   Iteration  | Time Elapsed |  Mini-batch  |  Mini-batch  | Base Learning|
|              |              |  (seconds)   |     Loss     |   Accuracy   |     Rate     |
|=========================================================================================|
|            1 |            1 |         0.37 |       2.1973 |        0.00% |       0.0100 |
|           10 |          100 |         2.12 |       2.2042 |        0.00% |       0.0100 |
|           20 |          200 |         3.69 |       2.1652 |       14.81% |       0.0100 |
|           30 |          300 |         5.20 |       1.5640 |        3.70% |       0.0100 |
|           40 |          400 |         6.70 |       0.8349 |       62.96% |       0.0100 |
|           50 |          500 |         8.19 |       0.6772 |       74.07% |       0.0100 |
|           60 |          600 |         9.66 |       0.3584 |       88.89% |       0.0100 |
|           70 |          700 |        11.18 |       0.2477 |       88.89% |       0.0100 |
|           80 |          800 |        12.77 |       0.0551 |      100.00% |       0.0100 |
|           90 |          900 |        14.36 |       0.0581 |      100.00% |       0.0100 |
|          100 |         1000 |        15.89 |       0.0272 |      100.00% |       0.0100 |
|          110 |         1100 |        17.42 |       0.0163 |      100.00% |       0.0100 |
|          120 |         1200 |        18.94 |       0.0113 |      100.00% |       0.0100 |
|          130 |         1300 |        20.48 |       0.0126 |      100.00% |       0.0100 |
|          140 |         1400 |        22.12 |       0.0172 |      100.00% |       0.0100 |
|          150 |         1500 |        23.69 |       0.0088 |      100.00% |       0.0100 |
|=========================================================================================|
```

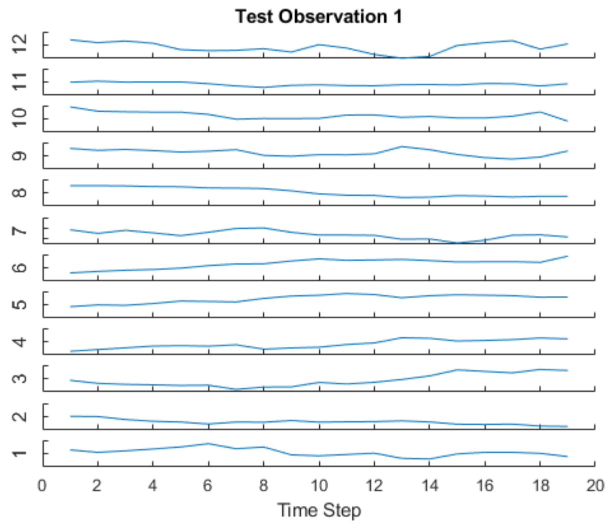
Test LSTM Network

- Load the test set and classify the sequences into speakers.
- Load the Japanese Vowels test data. XTest is a cell array containing 370 sequences of dimension 12 of varying length. YTest is a categorical vector of labels "1","2",..."9", which correspond to the nine speakers.

```
load JapaneseVowelsTest
XTest(1:3)
ans = 3×1 cell array
    {12×19 double}
    {12×17 double}
    {12×19 double}
```

- Visualize the first time series in a plot. Each subplot corresponds to a feature.

```
figure
for i = 1:12
    subplot(12,1,13-i)
    plot(XTest{1}(i,:))
    ylabel(i)
    xticklabels('')
    yticklabels('')
    box off
end
title("Test Observation 1")
subplot(12,1,12)
xticklabels('auto')
xlabel("Time Step")
```



- The LSTM network net was trained using mini-batches of sequences of similar length. Ensure that the test data is organized in the same way. Sort the test data by sequence length.

```
numObservationsTest = numel(XTest);
for i=1:numObservationsTest
    sequence = XTest{i};
    sequenceLengthsTest(i) = size(sequence,2);
end
[sequenceLengthsTest,idx] = sort(sequenceLengthsTest);
XTest = XTest(idx);
YTest = YTest(idx);
```

- Classify the test data. To reduce the amount of padding introduced by the classification process, set the mini-batch size to 27.

```
miniBatchSize = 27;
YPred = classify(net,XTest, ...
    'MiniBatchSize',miniBatchSize);
```

- Calculate the classification accuracy of the predictions.

```
acc = sum(YPred == YTest)./numel(YTest)
acc = 0.9432
```

IMPLEMENTING LSTMS USING TENSORFLOW: Tutorial with Code by Rowel Atiena