Appendix B

MATLAB® Neural Network Toolbox Function Description

In this appendix, we are describing some of the very important functions provided by the MATLAB Neural Network Toolbox that are used in various experiments in this thesis. The functions described below show their syntax, algorithm and interfaces to the user programs.

B.1 Categorical list of some useful Functions:

The Toolbox allows us to create and use many kinds of functions. There exist a large number of in-built functions that give us a great deal of control over the algorithms used to initialize, simulate, and train; and allow adaption for our networks to the solution of a particular type of problem.

Here, a categorical list of some important functions are given below (function name with brief description) that could be useful to the context of our problem solution [24].

Graphical Interface Function

- nnntool - Neural Network Tool: Graphical User Interface.

Learning Functions

- learngd - Gradient descent weight/bias learning function.

Net Input Functions

- netprod - Product net input function.
- netsum - Sum net input function.

Network Functions

- backprop - Backpropagation networks

Network Initialization Function

- initlay - Layer-by-layer network initialization function.

Network Use Functions

- adapt - Allow a neural network to adapt.
- disp - Display a neural network's properties.
- display - Display a neural network variable's name and properties.
- init - Initialize a neural network.
- sim - Simulate a neural network.
- train - Train a neural network.
New Networks Functions

- network - Create a custom neural network.
- newff - Create a feed-forward backpropagation network.

Performance Functions

- mse - Mean squared error performance function.
- sse - Sum squared error performance function.

Plotting Functions

- plotperf - Plot network performance.

Training Functions

- trainb - Batch training with weight and bias learning rules.
- traingd - Gradient descent backpropagation.
- traingda - Gradient descent with adaptive learning rate backpropagation.
- traingdm - Gradient descent with momentum backpropagation.
- traingdx - Gradient descent with momentum & adaptive learning rate backprop.
- trains - Sequential order incremental update.

Transfer Functions

- compet - Competitive transfer function.
- logsig - Log sigmoid transfer function.

Vector Functions

- minmax - Ranges of matrix rows.
B.2 Description of important Functions used in the Thesis:

- **NEWFF Function**:

The *newff* function is used to create a feed-forward backpropagation network.

**Syntax:**

```matlab
net = newff(pr, s, tf, btf, blf, pf)
```

More precisely we can write the syntax as,

```matlab
net = newff(PR, [S1 S2...SN], {TF1 TF2...TFN}, BTF, BLF, PF)
```

**Description:**

NEWFF(PR, [S1 S2...SN], {TF1 TF2...TFN}, BTF, BLF, PF) takes the following arguments as input:

- **PR** – a R×2 matrix of min and max values for R input elements.
  - or, we can directly use the function `maxmin` to retrieve such values from a vector P. For e.g., `maxmin(P)`
- **Si** – Size of *ith* layer, for N/ layers.
- **TFi** – Transfer function of *ith* layer.
- **BTF** – Backpropagation network training function.
- **BLF** – Backpropagation weight/bias learning function.
- **PF** – Performance function.

and the function returns an N layer feed-forward backprop network.

- The **Transfer functions** TFi can be any differentiable transfer function such as `tansig`, `logsig`, or `purelin`. Default function is `tansig`.
- The **Training function** BTF can be any of the backprop training functions such as `trainlm`, `trainbfg`, `trainrp`, `traingd`, etc. Default function is `trainlm`.
- The **Learning function** BLF can be either of the backpropagation learning functions such as `learngd`, or `learngdm`. Default function is `learngdm`.
- The **Performance function** PF can be any of the differentiable performance functions such as `sse`, `mse` or `msereg`. Default function is `mse`.

We may not pass all the parameter values to the *newff* function while first invoking it, instead we can explicitly set some values to the parameters latter in the program as follows:

```matlab
% Assigning the particular Performance function
net.performFcn = 'sse'; % similar to passing the argument for pf
% Assigning the particular Training function
net.trainFcn = 'logsig'; % similar to passing the argument for btf
```
Algorithm:

- Feed-forward networks consist of N layers using the dotprod weight function, netsum net input function, and the specified transfer functions (say logsig). The dotprod is the dot product weight function that applies weights to an input to get weighted inputs. The netsum is a net input function that calculate a layer's net input by combining its weighted inputs and biases.
- The first layer has weights coming from the input. Each subsequent layer has a weight coming from the previous layer. All layers have biases. The last layer is the network output.
- Each layer's weights and biases are initialized with the function initnw. It is a layer initialization function which initializes a layer's weights and biases according to the Nguyen-Widrow initialization algorithm.
- Adaption is done with trains (net.adaptFcn = 'trains'), which updates weights with the specified learning function (such as learngd, or learnngd).
- Training is done with the specified training function (such as trainlm, trainbfg, trainrp, traingd, etc.).
- Performance is measured according to the specified performance function (such as sse, or mse).
- The trains function trains a network with weight and bias learning rules with sequential order incremental updates. The sequence of inputs is presented to the network with updates occurring after each time step. Function trains is not called directly. Instead it is called by function adapt for the network whose net.adaptFcn property is set to 'trains', or by train for network's whose net.trainFcn property is set to 'trains' [24].

Example: (In the context of our thesis)

```plaintext
S1 = 5;                                             % No of neurons in the hidden layer
[R,Q] = size(digits);                                % Size of the Input vector is 35x10
[S2,Q] = size(targets);                              % Size of the Output vector is 10x10
P = digits;                                          % Input vector is assigned to P
T = targets;

% The two-layer network is created with newff.
net = newff(minmax(P),[S1 S2],{'logsig' 'logsig'},'traingdx');
```

The above codes create a two-layer feed-forward network. The network's input ranges from 0 to 1, since we are taking binary data for each digit (for graycycle image, floating-point data is used). The first layer has five logsig neurons; the second layer has ten logsig neuron. The trainlm network training function (default) is to be used.

Here the network is trained for 300 epochs (approximate).

```plaintext
net.trainParam.epochs = 300;
```
net = train(net, P, T);
Y = sim(net, P);

TRAIN Function:
The `train` function is used to train a neural network. This function trains a neural network \textit{net} according to \texttt{net.trainFcn} and \texttt{net.trainParam}.

Syntax:
Any one of the three different types syntax can used:

1. \texttt{net = train(net, P, T)}
2. \texttt{[net, TR] = train(net, P, T)} % sufficient to invoke the function.
3. \texttt{[NET, TR, Y, E, Pf, Af] = train(NET, P, T, Pi, Ai, VV, TV)} % more precise

Description:
\texttt{TRAIN(NET, P, T, Pi, Ai)} takes the following arguments as input:

- \texttt{NET} - Network.
- \texttt{P} - Network inputs.
- \texttt{T} - Network targets (optional), default = \texttt{zeros}.
- \texttt{Pi} - Initial input delay conditions (optional), default = \texttt{zeros}.
- \texttt{Ai} - Initial layer delay conditions, default = \texttt{zeros}.
- \texttt{VV} - Structure of validation vectors (optional), default = \texttt{[]}.
- \texttt{TV} - Structure of test vectors (optional), default = \texttt{[]}.

and the function returns the following arguments:

- \texttt{NET} - New network.
- \texttt{TR} - Training record (epoch and perf).
- \texttt{Y} - Network outputs.
- \texttt{E} - Network errors.
- \texttt{Pf} - Final input delay conditions (optional).
- \texttt{Af} - Final layer delay conditions.

\texttt{TRAIN}'s signal arguments can have two formats: cell array or matrix.

Algorithm:
- The \textit{train} function trains a neural network \textit{net} by calling the function indicated by \texttt{net.trainFcn}, using the training parameter values indicated by \texttt{net.trainParam}.
- Typically one epoch of training is defined as a single presentation of all input vectors to the network. The network is then updated according to the results of all those presentations.
- Training occurs until a maximum number of epochs occur, the performance goal is met, or any other stopping condition of the function \texttt{net.trainFcn} occurs.
• Some training functions depart from this norm by presenting only one input vector (or sequence) each epoch. An input vector (or sequence) is chosen randomly each epoch from concurrent input vectors (or sequences). In this case, the network uses a random order incremental training with learning function TRAINR, and receives the network returned by using NEWC and NEWSOM [24].

Example: (In the context of our thesis)

```matlab
S1 = 5; % No of neurons in the hidden layer
[R,Q] = size(digits); % Size of the Input vector is 35x10
[S2,Q] = size(targets); % Size of the Output vector is 10x10
P = digits; % Input vector is assigned to P
T = targets;
% The two-layer network is created with newff:
net = newff(minmax(P),[S1 S2],{'logsig' 'logsig'},'traingdx');
```

Here the network is trained for 300 epochs to a error goal of 0.1, and then resimulated.

```matlab
net.trainParam.epochs = 300;
net.trainParam.goal = 0.1;
net = train(net, P, T);
Y = sim(net, P);
```

\[
\text{TRAINGDX Function}:
\]

The \text{traingdx} function is a network training function that updates weight and bias values according to gradient descent momentum and an adaptive learning rate backpropagation. Gradient descent is the process of making changes to weights and biases, where the changes are proportional to the derivatives of network error with respect to those weights and biases. This is done to minimize network error.

\textbf{Syntax}:

\[
[\text{net, tr, Ac, El}] = \text{traingdx}(\text{net, Pd, Tl, Ai, Q, TS, VV, TV})
\]

\textbf{Description}:

- \text{TRAINGDX(NET, Pd, Tl, Ai, Q, TS, VV, TV)} takes the following arguments as input:
  - \text{NET} — Neural network.
  - \text{Pd} — Delayed input vectors.
  - \text{Tl} — Layer target vectors.
  - \text{Ai} — Initial input delay conditions.
  - \text{Q} — Batch size.
  - \text{TS} — Time steps.
  - \text{VV} — Empty matrix [ ] or structure of validation vectors.
TV – Empty matrix [ ] or structure of test vectors.

and the function returns the following arguments:

NET – Trained network.

TR – Training record of various values over each epoch:

- TR.epoch – Epoch number.
- TR.vperf – Validation performance.
- TR.tperf – Test performance.
- TR.lr – Adaptive learning rate.

Ac – Collective layer outputs for last epoch.

El – Layer errors for last epoch.

Training occurs according to the TRAINGDX's training parameters shown here with their default values:

<table>
<thead>
<tr>
<th>training parameters</th>
<th>default values</th>
</tr>
</thead>
<tbody>
<tr>
<td>net.trainParam.epochs</td>
<td>10 Maximum number of epochs to train</td>
</tr>
<tr>
<td>net.trainParam.goal</td>
<td>0 Performance goal</td>
</tr>
<tr>
<td>net.trainParam.lr</td>
<td>0.01 Learning rate</td>
</tr>
<tr>
<td>net.trainParam.lr_inc</td>
<td>1.05 Ratio to increase learning rate</td>
</tr>
<tr>
<td>net.trainParam.lr_dec</td>
<td>0.7 Ratio to decrease learning rate</td>
</tr>
<tr>
<td>net.trainParam.max_fail</td>
<td>5 Maximum validation failures</td>
</tr>
<tr>
<td>net.trainParam.max_perf_inc</td>
<td>1.04 Maximum performance increase</td>
</tr>
<tr>
<td>net.trainParam.mc</td>
<td>0.9 Momentum constant.</td>
</tr>
<tr>
<td>net.trainParam.min_grad</td>
<td>1e-10 Minimum performance gradient</td>
</tr>
<tr>
<td>net.trainParam.show</td>
<td>25 Epochs between displays (NaN for no displays)</td>
</tr>
<tr>
<td>net.trainParam.time</td>
<td>inf Maximum time to train in seconds</td>
</tr>
</tbody>
</table>

net = trainParam;

Dimensions for these variables are:

- Pd - No×Ni×TS cell array, each element P{i,j,ts} is a DijxQ matrix.
- TI - Ni×TS cell array, each element P{i,ts} is a VixQ matrix.
- Ai - Ni×LD cell array, each element Ai{i,k} is a SxQ matrix.

Where,

Ni = net.numInputs
N1 = net.numLayers
LD = net.numLayerDelays
Ri = net.inputs{i}.size
Si = net.layers{i}.size
\( V_i = \text{net.targets}\{i\}.\text{size} \)

\( D_{ij} = R_i \times \text{length(net,inputWeights}\{i,j\}.\text{delays}) \)

**Network Use:**

We can create a standard network that uses *traingdx* with *newff*, *newcf*, or *newelm*.

To prepare a custom network to be trained with *traingdx*, we do the following:

1. Set *net.trainFcn* to 'traingdx'.
   This will set *net.trainParam* to *traingdx*’s default parameters.
2. Set *net.trainParam* properties to desired values.

In either case, calling *train* with the resulting network will train the network with *traingdx*.

**Algorithm:**

- The *traingdx* function can train any network as long as its weight, net input, and transfer functions have derivative functions.
- Backpropagation is used to calculate derivatives of performance *perf* (performance function) with respect to the weight and bias variables \( X \). Each variable is adjusted according to the gradient descent with momentum.
  \[
  dX = mc \times dX_{\text{prev}} + lr \times mc \times d\text{perf} / dX
  \]
  where \( dX_{\text{prev}} \) is the previous change to the weight or bias.
- For each epoch, if performance decreases toward the goal, then the learning rate is increased by the factor \( lr\_\text{inc} \). If performance increases by more than the factor \( \text{max}\_\text{perf}\_\text{inc} \), the learning rate is adjusted by the factor \( lr\_\text{dec} \) and the change, which increased the performance, is not made [24].

**Stopping Criteria:**

Training stops when any of these conditions occur:

1. The maximum number of *epochs* (repetitions) is reached.
2. The maximum amount of *time* has been exceeded.
3. Performance has been minimized to the *goal*.
4. The performance gradient falls below \( \text{mingrad} \).
5. Validation performance has increase more than \( \text{max}\_\text{fail} \) times since the last time it decreased (when using validation).

**SIM Function**

The *sim* function is used to simulate a neural network.

**Syntax:**

\[
[Y, Pf, Af, E, perf] = \text{sim}(\text{net, P, Pi, Ai, T})
\]

**Description:**

\([Y, Pf, Af, E, perf] = \text{SIM(}\text{net, P, Pi, Ai, T})\) takes the following arguments as input:
NET - Network.
P - Network inputs.
Pi - Initial input delay conditions (optional), default = zeros.
Ai - Initial layer delay conditions (optional), default = zeros.
T - Network targets, default = zeros.

and the function returns the following arguments:

Y - Network outputs.
Pf - Final input delay conditions (optional).
Af - Final layer delay conditions (optional).
E - Network errors.
perf - Network performance.

Some of the aforementioned arguments are optional such as Pi, Ai, Pf, and Af. They need only to be used for networks that have input or layer delays.

SIM's signal arguments can have two formats: cell array or matrix.

Examples:

S1 = 5; % No of neurons in the hidden layer
[R,Q] = size(digits); % Size of the Input vector is 35x10
[S2,Q] = size(targets); % Size of the Output vector is 10x10
P = digits; % Input vector is assigned to P
T = targets;

% The two-layer network is created with newff.
net = newff(minmax(P),[S1 S2],{'logsig' 'logsig'},'traingdx');

Here the network is trained for 300 epochs to a error goal of 0.1, and then resimulated.

net.trainParam.epochs = 300;
net.trainParam.goal = 0.1;
net = train(net, P, T);
Y = sim(net, P);

Algorithm:

- SIM uses the following properties to simulate a network net.
  net.numInputs,
  net.numLayers
  net.outputConnect,
  net.biasConnect
  net.inputConnect,
  net.layerConnect
• The following properties determine the network's weight and bias values, and the number of delays associated with each weight:
  net.inputWeights{ij}.value
  net.layerWeights{i,j}.value
  net.layers{i}.value
  net.inputWeights{ij}.delays
  net.layerWeights{i,j}.delays

• The following function properties indicate how SIM applies weight and bias values to inputs to get each layer's output:
  net.inputWeights{ij}.weightFcn
  net.layerWeights{i,j}.weightFcn
  net.layers{i}.netInputFcn
  net.layers{i}.transferFcn

➤ **COMPET Function**: 

*Compet* is a competitive transfer function that calculates a layer's output from its net input.

**Syntax:**

A = compet(N)

**Description:**

COMPET(N) takes one input argument,

N - SxQ matrix of net input (column) vectors.

and returns output vectors with 1 where each net input vector has its maximum value, and 0 elsewhere.

**Examples:**

Here we define a net input vector N, calculate the output, and plot both with bar graphs.

n = [0; 1; -0.5; 0.5];

a = compet(n);

subplot(2,1,1), bar(n), ylabel('n')

subplot(2,1,2), bar(a), ylabel('a')

➤ **MINMAX Function**: 

*Minmax* is a function that gives the ranges of matrix rows.

**Syntax:**

pr = minmax(p)

**Description:**

MINMAX(P) takes one argument,
and returns the $R \times 2$ matrix $PR$ of minimum and maximum values for each row of $P$.

**Example 1:**

\[ p = \begin{bmatrix} 0 & 1 & 2 \\ -1 & -2 & -0.5 \end{bmatrix} \]

\[ pr = \text{minmax}(p) \]

Actually, it invokes the two separate functions as follows:

\[ pr = [\text{min}(p, []) \, \text{max}(p, [])] \]

---

**SIZE Function**

Size is a function that gives the size of the matrix.

**Syntax:**

\[ D = \text{size}(X) \]
\[ [M, N] = \text{size}(X) \]
\[ M = \text{size}(X, \text{DIM}) \]

**Description:**

- \[ D = \text{size}(X) ; \] % \[ D = [M, N] \] contains the number of rows and columns in the matrix.
  - For $M \times N$ matrix $X$, \text{size}(X) returns the two-element row vector.
  - For N-D arrays, \text{size}(X) returns a 1-by-N vector of dimension lengths. Trailing singleton dimensions are ignored.
- \[ [M, N] = \text{size}(X) \] returns the number of rows and columns in separate output variables. \[ [M_1, M_2, M_3, \ldots, MN] = \text{size}(X) \] returns the length of the first $N$ dimensions of $X$.
- \[ M = \text{size}(X, \text{DIM}) \] returns the length of the dimension specified by the scalar DIM. For example, \text{size}(X, 1) returns the number of rows.

**Example 1:**

The size is output as a single vector:

\[ d = \text{size}(\text{rand}(2,3,4)) \]
\[ d = \begin{bmatrix} 2 & 3 & 4 \end{bmatrix} \]

**Example 2:**

The size of each dimension is assigned to a separate variable:

\[ [m, n, p] = \text{size}(\text{rand}(2,3,4)) \]
\[ m = 2 \]
\[ n = 3 \]
\[ p = 4 \]