CSE5301 Neuro-Fuzzy Computing

Tutorial/Assignment 2: Adaline and Multilayer Perceptron

About this tutorial

The objective of this tutorial is to study:

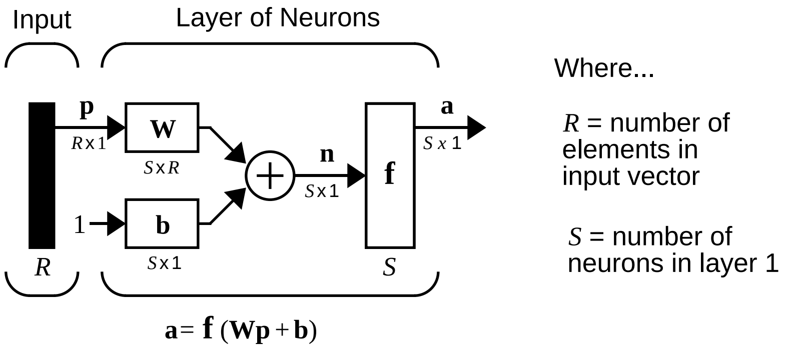
- The Neural Network Toolbox for MATLAB.
- Linear neural networks (Adalines) and related learning algorithms.
- Multilayer perceptrons and related learning algorithms.

2.1 Neural Network Toolbox

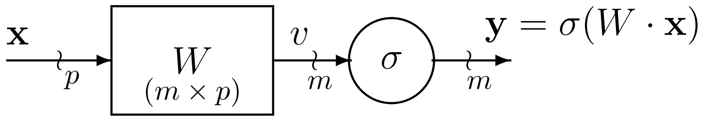
This section is an extract from *Neural Network Toolbox User's Guide* that is available in the pdf format in `$CSE5301/nnet.pdf`

2.1.1 Notation

Notation used in the *Neural Network Toolbox User's Guide* differs from that used in Lecture Notes. Graphically, the networks in the User's Guide are presented using a block-diagram notation.
A layer of neuron is presented in the User's Guide in the following way:



Equivalently, in the Lecture Notes notation a neuron is represented by the following block-diagram:



Note and record differences/similarities in notations.
Before we start the next part it is sensible to be aware of existence of **cells** and **structures** in MATLAB. Try:

`help struct` and `help cell`

2.1.2 Creating a Network (newff)

- To create a feedforward neural network (MLP) object in an **interactive way** you can use the command **nntool** to open the Neural Network Manager.

  You can then import or define data and train your network in the GUI or export your network to the command line for training.

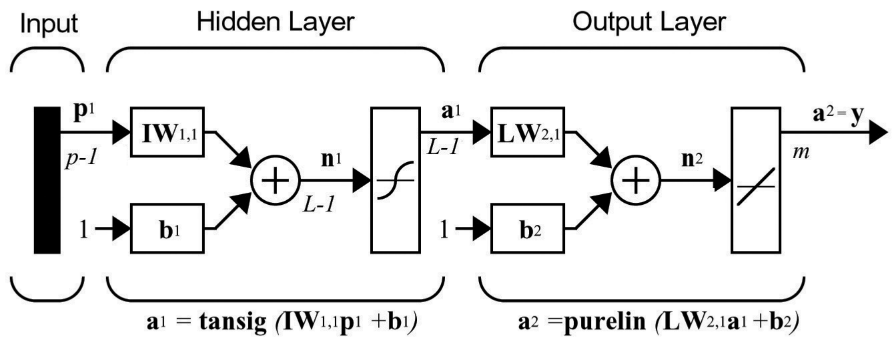
- To crate a feedforward neural network **from the command line** use the the function **newff**:

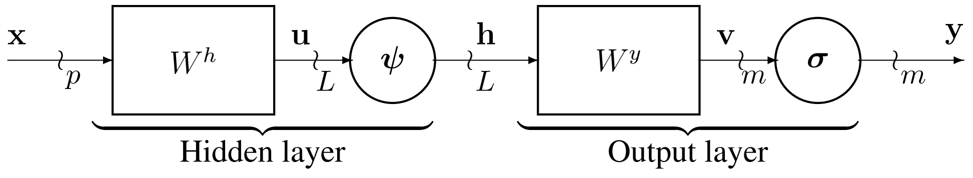
$$\text{net} = \textbf{newff}(\text{MnMx}, [\text{S}_1\ \text{S}_2 \ldots \text{S}_K], \{\text{TF}_1\ \text{TF}_2 \ldots \text{TF}_K\}, \text{BTF}, \text{BLF}, \text{PF})$$

where	MnMx	$(p-1 \times 2)$ matrix of min and max values for $\mathbf{x}$ input vectors.
	S$_i$	Size of ith layer, for K layers.
	TF$_i$	Transfer function of ith layer, default = **tansig**.
	BTF	Backpropagation network training function, default = **traingdx**.
	BLF	Backpropagation weight/bias learning function, default = **learngdm**.
	PF	Performance function, default = **mse**.

- Note that the dimensionality of the input space p is defined indirectly through the size of the matrix MnMx.
- The function **nntool** returns an K-layer feed-forward neural network network ready to be trained.
- The transfer functions TF$_i$ can be any differentiable transfer function such as **tansig** (hyperbolic tangent, tanh), **logsig** (unipolar sigmoidal function), or **purelin** (linear function).
- The training function BTF can be any of the backpropagation training functions such as **trainlm**, **trainbfg**, **trainrp**, **traingd**, etc. Details are discussed in the Lecture Notes.

Example: Consider a two-layer feed-forward network





with
inputs: $p = 4$, ($\mathbf{x} = [x_1, x_2, x_3, 1]$). The range of input values is specified as follows:
$x_1 \in [-1 \ldots 2]$, $x_2 \in [0 \ldots 5]$, $x_3 \in [-3 \ldots 3]$.
hidden layer: $L = 6$, ($\mathbf{h} = [h_1, \ldots, h_5, 1]$). TF = **tansig**
output layer: $m = 2$, ($\mathbf{y} = [y_1, y_2]$), TF = **purelin**.

The training function is to be **traingd**.
The following command defines the network `nnet1`:

```
p = 4; L = 6; m = 2 ;
XR = [-1 2; 0 5; -3 3] ;
nnet1 = newff(XR,[L-1,m],{'tansig','purelin'},'traingd') ;
```

This command **creates** the network object **nnet1** and also **initializes** the weights and biases of the network using the default Nguyen-Widrow algorithm (**initnw**). Therefore the network is ready for training.

2.1.3 Network structure and parameters

For the above example we can examine the structure of the created network **nnet1** and its all initial parameters.

- Typing in `nnet1` on the command line gives the structure of the top level neural network object. The extract of the structure is as follows:

```
Neural Network object:
       ...
subobject structures:
        inputs: {1x1 cell} of inputs
        layers: {2x1 cell} of layers
       outputs: {1x2 cell} containing 1 output
       targets: {1x2 cell} containing 1 target
        biases: {2x1 cell} containing 2 biases
  inputWeights: {2x1 cell} containing 1 input weight
  layerWeights: {2x2 cell} containing 1 layer weight
           ...
  weight and bias values:
            IW: {2x1 cell} containing 1 input weight matrix
            LW: {2x2 cell} containing 1 layer weight matrix
             b: {2x1 cell} containing 2 bias vectors
```

- The range matrix MnMx is hidden in
  `nnet1.inputs{1}.range` (Try it!)

- Parameters of two layers are stored in
  `nnet1.layers{1}` and in
  `nnet1.layers{1}` respectively.

- Initial values of the hidden weight matrix W^h $(L-1 \times p)$ (called "input matrix") are give in:

```
 nnet1.IW{1}                nnet1.b{1}
   1.3465  0.4704  0.1727   -4.2433
  -1.0747 -0.1044  0.5835    1.9954
   0.2627 -0.7105  0.5187    1.6448
  -0.0559  0.7689  0.4749   -3.0912
   1.2225 -0.1037 -0.5057    2.0419
```

- Similarly, initial values of the output weight matrix W^y $(m \times L)$ (called "hidden matrix") are give in:

```
nnet1.LW{2,1}                              nnet1.b{2}
  -0.1886  0.8338  0.7873 -0.2943 -0.9803  -0.5945
   0.8709 -0.1795 -0.8842  0.6263 -0.7222  -0.6026
```

- Examine other parameters of the network **nnet1**

2.1.4 Re-Initializing Weights

- If you need re-initialization, or change to the default initialization algorithm you use the command **init**:
- The function **init** takes a network object as input and returns a network object with all weights and biases initialized.
- This function is invoked by the **newff** command and uses the Nguyen-Widrow algorithm as the default initialization.
- If, for example, we would like to re-initialise the weights and biases in the first layer to random values using the **rands** function, we would issue the following commands:

```
nnet1.layers{1}.initFcn = 'initwb';
nnet1.inputWeights{1,1}.initFcn = 'rands';
nnet1.biases{1,1}.initFcn = 'rands';
nnet1.biases{2,1}.initFcn = 'rands';
nnet1 = init(nnet1);
```

2.1.5 Simulation (sim)

- The function **sim** simulates a network, that is it calculates outputs for given inputs:
- The function **sim** takes the network input X, and the network object `net`, and returns the network outputs Y.

$$Y = \sigma(W^y \cdot \sigma(W^h \cdot X))$$

- Here is how **sim** can be used to simulate the network **nnet1** we created above ($p - 1 = 3$ and $m = 2$) for a single input pattern:

```
X =
   0.5000      Y = sim(nnet1, X)
   2.0000         0.5628
   1.0000         1.2168
```

  (If you try these commands, your output may be different, depending on the state of your random number generator when the network was initialized.)

- Below, **sim** is called to calculate the outputs for a set of three input vectors $N = 3$.

```
X =
  -0.5000    1.0000    1.5000
   4.0000    2.0000    3.0000
  -2.0000    1.0000    2.0000
Y = sim(nnet1, X)
   1.5924    0.5821    0.5193
  -0.7313    0.1693    0.2691
```

2.1.6 Incremental Training (adapt)

- In incremental training the weights and biases of the network are updated each time an input is presented to the network.
- The function **adapt** is used to train networks in the incremental (pattern) mode. Use `help adapt` for the description of the function
- The function **adapt** takes the network object and the inputs and the targets from the training set, and returns the trained network object and the outputs and errors of the network for the final weights and biases.
- Inputs and targets can be presented either as **cell arrays**:

```
X = {[1;2] [2;1] [2;3] [3;1]};
D = {4 5 7 7};
```

  or matrices:

```
X = [[1;2] [2;1] [2;3] [3;1]];
D = [4 5 7 7];
```

- The cell array format is most convenient to be used for networks with multiple inputs and outputs, and allows sequences of inputs to be presented:
- The matrix format can be used if only one time step is to be simulated, so that the sequences of inputs cannot be described.
- There are two basic functions available for the incremental training: **learngd** and **learngdm**. Refer to the manual for details.

2.1.7 Batch Training (train)

- In batch mode the weights and biases of the network are updated only after the entire training set has been applied to the network.
- The batch training is invoked using the function **train**.
- All advanced training algorithms are used in the batch mode.
- The format of data/signal is the same as for the incremental learning.

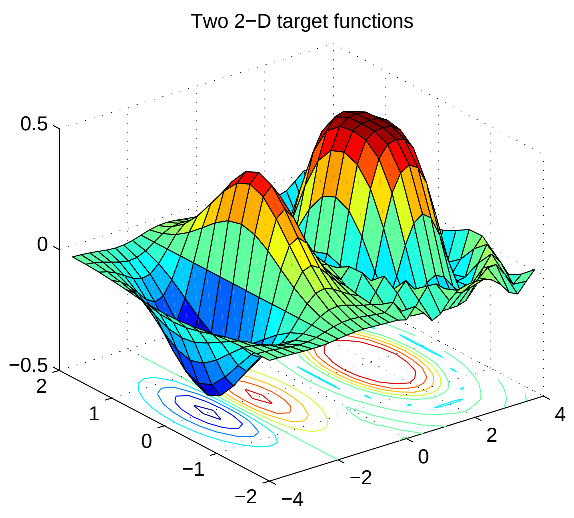
2.1.8 Function approximation — Example

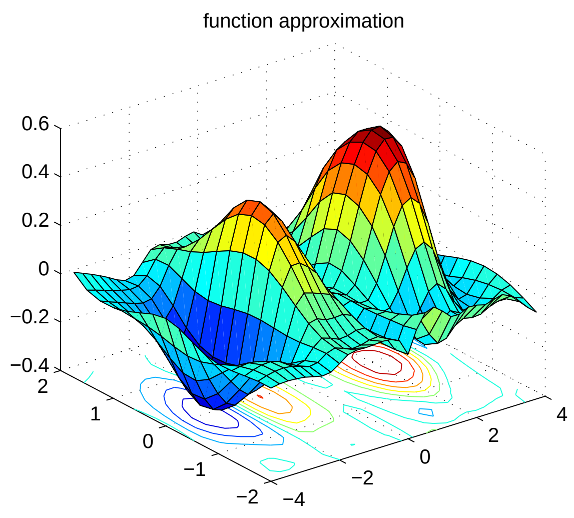
In this example we use the Levenberg-Marquardt algorithm to approximate two functions of two variables.

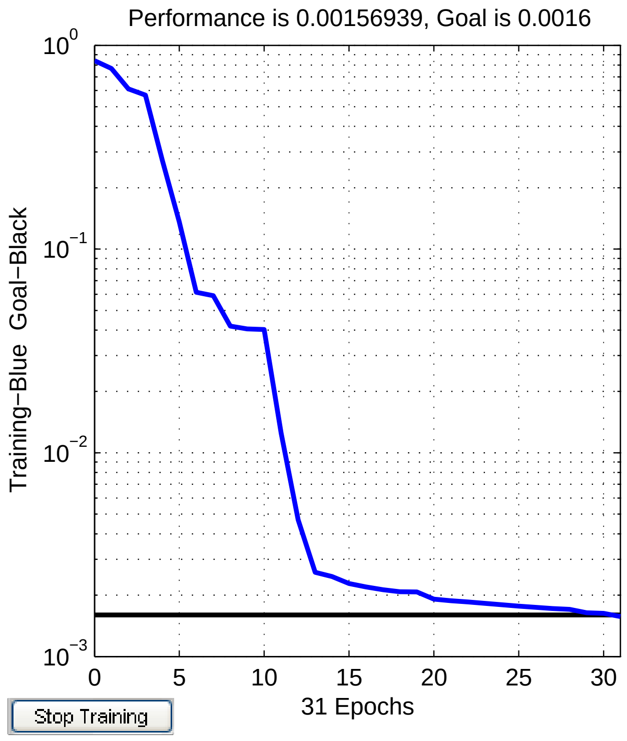
```
%  fap2dLM.m
%           28 March 2005
% Approximation of two 2D functions using the Levenberg-Marquardt
%            algorithm
clear
% Generation of training points
na = 16 ;  N = na^2; nn = 0:na-1;
X1 = nn*4/na - 2;
[X1 X2] = meshgrid(X1);
R  = -(X1.^2 + X2.^2 +0.00001);
D1 = X1 .* exp(R);  D = (D1(:))';
D2 = 0.25*sin(2*R)./R ;  D = [D ; (D2(:))'];
Y = zeros(size(D)) ;
X = [ X1(:)'; X2(:)'];
figure(1), % myfigA(1, 11, 10)
surfc([X1-2 X1+2], [X2 X2], [D1 D2]),
title('Two 2-D target functions'),
grid on, drawnow
% print(1, '-depsc2', 'p2LM1')

% network specification
```

```
p = 2 ; % Number of inputs excluding bias
L = 12; % Number of hidden signals excluding bias
m = 2 ; % Number of outputs
MnMx = [-2 2; -2 2] ;
   Nnet = newff(MnMx, [L, m]) ; % Network creation
   Nnet.trainParam.epochs = 50;
   Nnet.trainParam.goal = 0.0016;
   Nnet = train(Nnet, X, D) ;  % Training
 grid on, hf = gcf ;
 set(hf, 'PaperUnits', 'centimeters', 'PaperPosition', [0 0 10 11] );
 % print(hf, '-depsc2', 'p2LM3')
   Y = sim(Nnet , X) ;         % Validation
D1(:)=Y(1,:)'; D2(:)=Y(2,:)';
% Two 2-D approximated functions are plotted side by side
figure(2), % myfigA(2, 11, 10)
surfc([X1-2 X1+2], [X2 X2], [D1 D2])
title('function approximation')
grid on, drawnow
% print(2, '-depsc2', 'p2LM2')
```

The results should be similar to those in the following figures:







Exercise 2. 1
Repeat the above function approximation exercise replacing the Levenberg-Marquardt algorithm with a conjugate gradient algorithm, **traincgp**.

Run it for two different values of the target error (goal) parameter.
You might also need to increase the maximum number of epochs to achieve satisfactory results.
□

2.2 Linear Networks — Adaline and its applications

Exercise 2. 2
Study the following demo files which demonstrate properties and applications of linear neural networks:

```
  $CSE5301/Mtlb/adln1.m, $NNET/nndemos/demolin1...8.m.
```
□

Exercise 2. 3
Consider a problem of approximation of a collection of points by a **regression line**.
In this case the dimensionality of the augmented input space is $p = 2$.
The objective is to find the regression line: $y = a \cdot x + b$, that is, to find the parameters $\{a, b\}$ specifying the approximation line, from the collection of points $\{\mathbf{x}(n), d(n)\}$.

1. Re-write the Normal (Wiener-Hopf) equation directly for two unknown weights, w_1, w_2 and points, $\{x_1(n), x_2(n) = 1, d(n)\}$ for $n = 1, \ldots, N$.

   Derive equations for $a = w_1, b = w_2$ as functions of the training points.

   In the derived expressions identify the following statistical quantities: the mean, $\bar{x}$ the second moment $\bar{x^2}$ of $\mathbf{x}$, the mean $\bar{d}$ of d, and the correlation c of d and x.

2. Illustrate using MATLAB the problem of fitting the regression line into a collection of points from a $\{x, y\}$ plane.

   Compare the results obtained using the following three methods:

   (a) Direct solution of the Wiener-Hopf equation as derived above,
   (b) The LMS learning law,
   (c) The Sequential Regression algorithm.

□

Exercise 2. 4
Study the following MATLAB scripts which demonstrate applications of linear neural networks in adaptive signal processing:

```
  $CSE5301/Mtlb/adlpr.m, adsid.m, adlnc.m
  $NNET/nndemos/applin1...4.m
```
□

Exercise 2. 5
Modify the following scripts

1. `adlpr.m` — adaptive prediction,
2. `adsid.m` — adaptive system identification,
3. `adlnc.m` — adaptive noise cancellation

Modification should include:

- replacement of the LMS learning law by the sequential regression (SR) algorithm,
- usage of the **amplitude and frequency modulated** sinusoidal input signal in all three cases. Modify parameters of the signal and filters involved.

□

2.3 Multilayer Perceptron — Back-Propagation

Exercise 2. 6
Study the following demo files which demonstrate properties and applications of multilayer perceptrons and backpropagation learning law:

```
$NNET/nnd12sd1, nnd12sd2, nnd12vl, nnd12cg, nnd12m
```

□

Exercise 2. 7
Write a function approximation script similar to `$CSE5301/Mtlb/fap2D.m` with the following modifications:

1. The function to be approximated, $d = f(x)$, is of the following form:

   $$d = 0.02x(x^2 - 13x + 48)\cos(2x) + 0.2\text{noise} \ , \quad \text{for } x \in (0, \ldots, 9)$$

   Note that to plot such a function in MATLAB you can use:

   ```
   x = 0:dx:9 ;
   d = 0.02*polyval([1 -13 48 0],x).*cos(2*x)+0.2*rand(1,N);
   plot(x, d), grid
   ```

2. Use the **pattern update** mode in learning
3. Use the Nguyen-Widrow initialisation. You can use the script from `$CSE5301/Mtlb/nwini.m` described in Lecture Notes in sec. 5.4

□

Exercise 2. 8
Repeat the above exercise using the Neural Network Toolbox as in sec. 2.1.8.

Use the **conjugate gradient** and **Levenberg-Marquardt** learning laws and compare their efficiency.
□

Exercise 2. 9
For the above neural network plot **two** error surfaces, $J(w_a, w_b)$, where (w_a, w_b) is a pair of weights of your choice from the hidden and output layers.
□

Exercise 2. 10
Use a two-layer perceptron and a selected backpropagation learning law (conjugate gradient or Levenberg-Marquardt) to design an XOR gate. Aim at the minimal total number of synapses.

Once the learning is finished, replace the output activation function with a unipolar step function and verify that the perceptron really acts as an XOR gate.

Sketch the detailed structure of the obtained XOR perceptron.
□

2.4 An image coding algorithm using a two-layer perceptron

Exercise 2. 11
Implement in MATLAB an image coding algorithm using a multi-layer perceptron and an advanced learning algorithm as discussed in Lecture Notes.

- Demo images can be found in

  `$MATLAB/toolbox/matlab/demos/*.mat`

  To examine the demo images run:

  ```
  load file_name
  whos
  image(X),
  colormap(map), title(caption)
  imageext
  ```

- For initial experimentation use a 60×80 pixel sub-image from any image available in MATLAB.
- The block-matrix conversion functions `blkM2vc.m, vc2blkM.m` are in `$CSE5301/Mtlb`
- Use either the Levenberg-Marquardt or a conjugate gradient algorithm.
- Determine the speed of training and the SNR.

□

Submission

In **your report/submission** include answers/solutions to all exercises. Include

- Brief comments regarding the demo files,
- Relevant derivations, equations, scripts and plots with suitable comments.

Note that in the recent versions of MATLAB from the editor window you can invoke the **publish** command:
File ⟶ Publish To HTML
It creates the directory html with all the results in it.