

Artificial Neural Networks

The Tutorial

With MATLAB

Contents

1. PERCEPTRON	3
1.1. CLASSIFICATION WITH A 2-INPUT PERCEPTRON.....	3
1.2. CLASSIFICATION WITH A 3-INPUT PERCEPTRON.....	5
1.3. CLASSIFICATION WITH A 2-NEURON PERCEPTRON.....	6
1.4. CLASSIFICATION WITH A 2-LAYER PERCEPTRON	7
2. LINEAR NETWORKS.....	9
2.1. PATTERN ASSOCIATION WITH A LINEAR NEURON	9
2.2. TRAINING A LINEAR LAYER.....	11
2.3. ADAPTIVE LINEAR LAYER	13
2.4. LINEAR PREDICTION.....	14
2.5. ADAPTIVE LINEAR PREDICTION	15
3. BACKPROPAGATION NETWORKS.....	17
3.1. PATTERN ASSOCIATION WITH A LINEAR NEURON	17

1. Perceptron

1.1. Classification with a 2-input perceptron.

SIMUP - Simulates a perceptron layer.

TRAINP - Trains a perceptron layer with perceptron rule.

Using the above functions a 2-input hard limit neuron is trained to classify 4 input vectors into two categories.

DEFINING A CLASSIFICATION PROBLEM

A row vector P defines four 2-element input vectors:

```
P = [-0.5 -0.5 +0.3 +0.0;  
     -0.5 +0.5 -0.5 +1.0];
```

A row vector T defines the vector's target categories.

```
T = [1 1 0 0];
```

PLOTTING THE VECTORS TO CLASSIFY

We can plot these vectors with PLOTPV:

```
plotpv(P,T);
```

The perceptron must properly classify the 4 input vectors in P into the two categories defined by T.

DEFINE THE PERCEPTRON

Perceptrons have HARDLIM neurons. These neurons are capable of separating an input space with a straight line into two categories (0 and 1).

INITP generates initial weights and biases for our neuron:

```
[W,b] = initp(P,T)
```

INITP - Initializes a perceptron layer.

```
[W,B] = INITP(P,T)
```

P - RxQ matrix of input vectors.

T - SxQ matrix of target outputs.

Returns weights and biases.

INITIAL PERCEPTRON CLASSIFICATION

The input vectors can be replotted...

```
plotpv(P,T)
```

...with the neuron's initial attempt at classification.

plotpc(W,b)

The neuron probably does not yet make a good classification! Fear not...we are going to train it.

TRAINING THE PERCEPTRON

TRAINP trains perceptrons to classify input vectors.

TRAINP returns new weights and biases that will form a better classifier. It also returns the number of epochs the perceptron was trained and the perceptron's errors throughout training.

```
[W,b,epochs,errors] = trainp(W,b,P,T,-1);
```

TRAINP Train perceptron layer with perceptron rule.

```
[W,B,TE,TR] = TRAINP(W,B,P,T,TP)
```

W - SxR weight matrix.

B - Sx1 bias vector.

P - RxQ matrix of input vectors.

T - SxQ matrix of target vectors.

TP - Training parameters (optional).

Returns:

W - New weight matrix.

B - New bias vector.

TE - Trained epochs.

TR - Training record: errors in row vector.

Training parameters are:

TP(1) - Epochs between updating display, default = 1.

TP(2) - Maximum number of epochs to train, default = 100.

Missing parameters and NaN's are replaced with defaults.

If TP(1) is negative, and a 1-input neuron is being trained the input vectors and classification line are plotted instead of the network error.

PLOTTING THE ERROR CURVE

Here the errors are plotted with respect to training epochs:

```
ploterr(errors);
```

USING THE CLASSIFIER

We can now classify any vector using SIMUP.

Lets try an input vector of [-0.5; 0.5]:

```
p = [-0.5; 0.5];
```

```
a = simup(p,W,b)
```

SIMUP Simulate perceptron layer.

SIMUP(P,W,B)

P - R×Q matrix of input (column) vectors.

W - S×R weight matrix.

B - S×1 bias (column) vector.

Returns outputs of the perceptron layer.

Now, use SIMUP yourself to test whether [0.3; -0.5] is correctly classified as 0.

1.2. Classification with a 3-input perceptron

Using the above functions a 3-input hard limit neuron is trained to classify 8 input vectors into two categories.

DEFINING A CLASSIFICATION PROBLEM

A matrix P defines eight 3-element input (column) vectors:

```
P = [-1 +1 -1 +1 -1 +1 -1 +1 ;  
      -1 -1 +1 +1 -1 -1 +1 +1 ;  
      -1 -1 -1 -1 +1 +1 +1 +1 ] ;
```

A row vector T defines the vector's target categories.

```
T = [0 1 0 0 1 1 0 1] ;
```

PLOTTING THE VECTORS TO CLASSIFY

We can plot these vectors with PLOTPV:

```
plotpv(P,T) ;
```

The perceptron must properly classify the 4 input vectors in P into the two categories defined by T.

DEFINE THE PERCEPTRON

```
[W,b] = initp(P,T)
```

INITIAL PERCEPTRON CLASSIFICATION

The input vectors can be replotted...

```
plotpv(P,T)
```

...with the neuron's initial attempt at classification.

```
plotpc(W,b)
```

The neuron probably does not yet make a good classification! Fear not...we are going to train it.

TRAINING THE PERCEPTRON

```
[W,b,epochs,errors] = trainp(W,b,P,T,-1);
```

PLOTTING THE ERROR CURVE

Here the errors are plotted with respect to training epochs:

```
ploterr(errors);
```

USING THE CLASSIFIER

We can now classify any vector using SIMUP. Lets try an input vector of [0.7; 1.2; -0.2]:

```
p = [0.7; 1.2; -0.2];  
a = simup(p,W,b)
```

Now, use SIMUP to see if [-1; +1; -1] is properly classified as a 0.

1.3. Classification with a 2-neuron perceptron

Using the above functions a layer of 2 hard limit neurons is trained to classify 10 input vectors into 4 categories.

DEFINING A CLASSIFICATION PROBLEM

A matrix P defines ten 2-element input (column) vectors:

```
P = [+0.1 +0.7 +0.8 +0.8 +1.0 +0.3 +0.0 -0.3 -0.5 -1.5; ...  
      +1.2 +1.8 +1.6 +0.6 +0.8 +0.5 +0.2 +0.8 -1.5 -1.3];
```

A matrix T defines the categories with target (column) vectors.

```
T = [1 1 1 0 0 1 1 1 0 0;  
      0 0 0 0 0 1 1 1 1 1];
```

PLOTTING THE VECTORS TO CLASSIFY

```
plotpv(P,T);
```

The perceptron must properly classify the 4 input vectors in P into the two categories defined by T.

DEFINE THE PERCEPTRON

A perceptron layer with two neurons is able to separate the input space into 4 different categories.

```
[W,b] = initp(P,T)
```

INITIAL PERCEPTRON CLASSIFICATION

The input vectors can be replotted...

```
plotpv(P,T)
```

...with the neuron's initial attempt at classification.

```
plotpc(W,b)
```

The neuron probably does not yet make a good classification! Fear not...we are going to train it.

TRAINING THE PERCEPTRON

```
[W,b,epochs,errors] = trainp(W,b,P,T,-1);
```

PLOTTING THE ERROR CURVE

Here the errors are plotted with respect to training epochs:

```
ploterr(errors);
```

USING THE CLASSIFIER

We can now classify any vector we like using SIMUP. Lets try an input vector of [0.7; 1.2]:

```
p = [0.7; 1.2];  
a = simup(p,W,b)
```

Now, use SIMUP to see if [0.1; 1.2] is properly classified as [1; 0].

1.4. Classification with a 2-layer perceptron

Using the above functions a two-layer perceptron can often classify non-linearly separable input vectors.

The first layer acts as a non-linear preprocessor for the second layer. The second layer is trained as usual.

DEFINING A CLASSIFICATION PROBLEM

A matrix P defines ten 2-element input (column) vectors:

```
P = [-0.5 -0.5 +0.3 -0.1 -0.8;  
      -0.5 +0.5 -0.5 +1.0 +0.0];
```

A matrix T defines the categories with target (column) vectors.

```
T = [1 1 0 0 0];
```

PLOTTING THE VECTORS TO CLASSIFY

```
plotpv(P,T);
```

The perceptron must properly classify the input 5 vectors in P into the 2 categories defined by T.

Because the vectors are not linearly separable (you cannot draw a line between x's and o's) a single layer perceptron cannot classify them properly. We will try using a two-layer perceptron to classify them.

DEFINE THE PERCEPTRON

To maximize the chance that the preprocessing layer finds a linearly separable representation for the input vectors, it needs a lot of neurons. We will try 20.

```
s1 = 20;
```

INITP generates initial weights and biases for our network:

```
[w1,b1] = initp(P,S1);   Preprocessing layer  
[w2,b2] = initp(S1,T);   Learning layer
```

TRAINING THE PERCEPTRON

TRAINP trains perceptrons to classify input vectors.

The first layer is used to preprocess the input vectors:

```
A1 = simup(P,W1,b1);
```

TRAINP is the used to train the second layer to classify the preprocessed input vectors A1.

```
[w2,b2,epochs,errors] = trainp(w2,b2,A1,T,-1);
```

PLOTTING THE ERROR CURVE

Here the errors are plotted with respect to training epochs:

```
ploterr(errors);
```

If the hidden (first) layer preprocessed the original non-linearly separable input vectors into new linearly separable vectors, then the perceptron will have 0 error. If the error never reached 0, it means a new preprocessing layer should be created (perhaps with more neurons). I.e. try running this script again.

USING THE CLASSIFIER

IF the classifier WORKED we can now classify any vector we like using SIMUP. Lets try an input vector of [0.7; 1.2]:

```
p = [0.7; 1.2];  
a1 = simup(p,W1,b1);   Preprocess the vector  
a2 = simup(a1,W2,b2)   Classify the vector
```

2. Linear networks

2.1. Pattern association with a linear neuron

Using the above functions a linear neuron is designed to respond to specific inputs with target outputs.

DEFINING A PATTERN ASSOCIATION PROBLEM

P defines two 1-element input patterns (column vectors):

```
P = [1.0 -1.2];
```

T defines the associated 1-element targets (column vectors):

```
T = [0.5 1.0];
```

PLOTTING THE ERROR SURFACE AND CONTOUR

ERRSURF calculates errors for a neuron with a range of possible weight and bias values. PLOTES plots this error surface with a contour plot underneath.

```
w_range = -1:0.1:1;
```

```
b_range = -1:0.1:1;
```

```
ES = errsurf(P,T,w_range,b_range,'purelin');
```

ERRSURF(P,T,WV,BV,F)

P - 1xQ matrix of input vectors.

T - 1xQ matrix of target vectors.

WV - Row vector of values of W.

BV - Row vector of values of B.

F - Transfer function (string).

Returns a matrix of error values over WV and BV.

```
plotes(w_range,b_range,ES);
```

PLOTES(WV,BV,ES,V)

WV - 1xN row vector of values of W.

BV - 1xM row vector of values of B.

ES - MxN matrix of error vectors.

V - View, default = [-37.5, 30].

Plots error surface with contour underneath.

Calculate the error surface ES with ERRSURF.

The best weight and bias values are those that result in the lowest point on the error surface.

DESIGN THE NETWORK

The function SOLVELIN will find the weight and bias that result in the minimum error:

```
[w,b] = solvelin(P,T)
```

SOLVELIN Design linear network.

```
[W,B] = SOLVELIN(P,T)  
P - RxQ matrix of Q input vectors.  
T - SxQ matrix of Q target vectors.  
Returns:  
W - SxR weight matrix.  
B - Sx1 bias vector.
```

CALCULATING THE NETWORK ERROR

SIMULIN is used to simulate the neuron for inputs P.

```
A = simulin(P,w,b) ;
```

SIMULIN Simulate linear layer.

```
SIMULIN(P,W,B)  
P - RxQ Matrix of input (column) vectors.  
W - SxR Weight matrix of the layer.  
B - Sx1 Bias (column) vector of the layer.  
Returns outputs of the perceptron layer.
```

We can then calculate the neurons errors.

```
E = T - A;
```

SUMSQR adds up the squared errors.

```
SSE = sumsqr(E)
```

PLOT SOLUTION ON ERROR SURFACE

PLOTES replots the error surface.

```
plotes(w_range,b_range,ES) ;
```

PLOTEP plots the "position" of the network using the weight and bias values returned by SOLVELIN.

```
plotep(w,b,SSE)
```

As can be seen from the plot, SOLVELIN found the minimum error solution.

USING THE PATTERN ASSOCIATOR

We can now test the associator with one of the original inputs, -1.2, and see if it returns the target, 1.0.

```
p = -1.2;
```

```
a = simulin(p,w,b)
```

Use SIMLIN to check that the neurons response to 1.0 is 0.5.

2.2. Training a linear layer

INITLIN - Initializes a linear layer.

TRAINWH - Trains a linear layer with Widrow-Hoff rule.

SIMULIN - Simulates a linear layer.

Using the above functions a linear layer is trained to respond to specific inputs with target outputs.

DEFINING A PATTERN ASSOCIATION PROBLEM

P defines four 3-element input patterns (column vectors):

```
P = [+1.0 +1.5 +1.2 -0.3  
      -1.0 +2.0 +3.0 -0.5  
      +2.0 +1.0 -1.6 +0.9];
```

T defines associated 4-element targets (column vectors):

```
T = [+0.5 +3.0 -2.2 +1.4  
      +1.1 -1.2 +1.7 -0.4  
      +3.0 +0.2 -1.8 -0.4  
      -1.0 +0.1 -1.0 +0.6];
```

DEFINE THE NETWORK

INITLIN generates initial weights and biases for our neuron:

```
[W,b] = initlin(P,T);
```

TRAINING THE NETWORK

TRAINWH uses the Widrow-Hoff rule to train PURELIN networks.

```
me = 400;      Maximum number of epochs to train.
```

```
eg = 0.001;   Sum-squared error goal.
```

```
[W,b,epochs,errors] = trainwh(W,b,P,T,[NaN me eg NaN]);
```

TRAINWH Train linear layer with Widrow-Hoff rule.

[W,B,TE,TR] = TRAINWH(W,B,P,T,TP)

W - SxR weight matrix.

B - Sx1 bias vector.

P - RxQ matrix of input vectors.

T - SxQ matrix of target vectors.

TP - Training parameters (optional).

Returns:

W - new weight matrix

B - new weights & biases.

TE - the actual number of epochs trained.

TR - training record: [row of errors]

Training parameters are:

TP(1) - Epochs between updating display, default = 25.

TP(2) - Maximum number of epochs to train, default = 100.

TP(3) - Sum-squared error goal, default = 0.02.

TP(4) - Learning rate, default found with MAXLINLR.

Missing parameters and NaN's are replaced with defaults.

The plot shows the final error met the error goal.

PLOTTING INDIVIDUAL ERRORS

BARERR creates a bar plot of errors associated with

barerr(T-simulin(P,W,b))

BARERR Plot bar chart of errors.

BARERR(E)

E - SxQ matrix of error vectors.

Plots bar chart of the squared errors in each column.

Note that while the sum of these squared errors is less than our error goal, the individual errors are not the same.

USING THE PATTERN ASSOCIATOR

We can now test the associator with one of the original input vectors [1; -1; 2], and see if it returns the appropriate target vector [0.5; 1.1; 3; -1].

```
p = [1; -1; 2];  
a = simulin(p,W,b)
```

Use **SIMULIN** to check that the neuron response to [1.5; 2; 1] is the target response [3; -1.2; 0.2; 0.1].

2.3. Adaptive linear layer

INITLIN - Initializes a linear layer.

ADAPTWH - Trains a linear layer with Widrow-Hoff rule.

Using the above functions a linear neuron is allowed to adapt so that, given one signal, it can predict a second signal.

DEFINING A WAVE FORM

TIME defines the time steps of this simulation.

```
time = 1:0.0025:5;
```

P defines a signal over these time steps:

```
P = sin(sin(time).*time*10);
```

T is a signal which is linearly related to P:

```
T = P * 2 + 2;
```

PLOTTING THE SIGNALS

Here is how the two signals are plotted:

```
plot(time,P,time,T,'--')
title('Input and Target Signals')
xlabel('Time')
ylabel('Input ____ Target _ _')
```

DEFINE THE NETWORK

```
[w,b] = initlin(P,T)
```

ADAPTING THE LINEAR NEURON

ADAPTWH simulates adaptive linear neurons. It takes initial weights and biases, an input signal, and a target signal, and filters the signal adaptively. The output signal and the error signal are returned, along with new weights and biases.

```
lr = 0.01;      Learning rate.
```

```
[a,e,w,b] = adaptwh(w,b,P,T,lr);
```

PLOTTING THE OUTPUT SIGNAL

Here the output signal of the linear neuron is plotted with the target signal.

```
plot(time,a,time,T,'--')
title('Output and Target Signals')
xlabel('Time')
ylabel('Output ____ Target _ _')
```

It does not take the adaptive neuron long to figure out how to generate the target signal.

PLOTTING THE ERROR SIGNAL

A plot of the difference between the neurons output signal and the target shows how well the adaptive neurons works.

```
plot(time,e)
hold on
plot([min(time) max(time)], [0 0], 'r')
hold off
title('Error Signal')
xlabel('Time')
ylabel('Error')
```

2.4. Linear prediction

SOLVELIN - Solves for a linear layer.

SIMULIN - Simulates a linear layer.

Using the above functions a linear neuron is designed to predict the next value in a signal, given the last five values of the signal.

DEFINING A WAVE FORM

TIME defines the time steps of this simulation.

```
time = 0:0.025:5;    from 0 to 6 seconds
```

T defines the signal in time to be predicted:

```
T = sin(time*4*pi);
```

The input P to the network is the last five values of the signal T:

```
P = delaysig(T,1,5);
```

DELAYSIG Create delayed signal matrix from signal matrix.

DELAYSIG(X,D)

X - SxT matrix with S-element column vectors for T timesteps.

D - Maximum delay.

Returns signal X delayed by 0, 1, ..., and D2 timesteps.

DELAYSIG(X,D1,D2)

X - SxT matrix with S-element column vectors for T timesteps.

D1 - Minimum delay.

D2 - Maximum delay.

Returns signal X delayed by D1, D1+1, ..., and D2 timesteps.

The signal X can be a row vector of values, or a matrix of (column) vectors.

PLOTTING THE SIGNALS

Here is a plot of the signal to be predicted:

```
plot(time,T)
xlabel('Time','Target Signal','Signal to be Predicted')
```

SOLVELIN solves for weights and biases which will let the linear neuron model the system.

```
[w,b] = solvelin(P,T)
```

TESTING THE PREDICTOR

SIMULIN simulates the linear neuron, which attempts to predict the next value in the signal at each timestep.

```
a = simulin(P,w,b);
```

The output signal is plotted with the targets.

```
plot(time,a,time,T,'+')
xlabel('Time','Output _ Target +','Output and Target Signals')
```

The linear neuron does a good job, doesn't it?

Error is the difference between output and target signals.

```
e = T-a;
```

This error can be plotted.

```
plot(time,e)
hold on
plot([min(time) max(time)], [0 0],':r')
hold off
xlabel('Time','Error','Error Signal')
```

Notice how small the error is!

2.5. Adaptive linear prediction

INITLIN - Initializes a linear layer.

ADAPTWH - Trains a linear layer with Widrow-Hoff rule.

Using the above functions a linear neuron is adaptively trained to predict the next value in a signal, given the last five values of the signal.

The linear neuron is able to adapt to changes in the signal it is trying to predict.

DEFINING A WAVE FORM

TIME1 and TIME2 define two segments of time.

```
time1 = 0:0.05:4;      from 0 to 4 seconds  
time2 = 4.05:0.024:6; from 4 to 6 seconds
```

TIME defines all the time steps of this simulation.

```
time = [time1 time2];  from 0 to 6 seconds
```

T defines a signal which changes frequency once:

```
T = [sin(time1*4*pi) sin(time2*8*pi)];
```

The input P to the network is the last five values of the target signal:

```
P = delaysig(T,1,5);
```

PLOTTING THE SIGNALS

Here is a plot of the signal to be predicted:

```
plot(time,T)  
xlabel('Time','Target Signal','Signal to be Predicted')
```

DEFINE THE NETWORK

INITLIN generates initial weights and biases for our neuron:

```
[w,b] = initlin(P,T)
```

ADAPTING THE LINEAR NEURON

ADAPTWH simulates adaptive linear neurons. It takes initial weights and biases, an input signal, and a target signal, and filters the signal adaptively. The output signal and the error signal are returned, along with new weights and biases.

We will use a learning rate of 0.1.

```
lr = 0.1;  
[a,e,w,b] = adaptwh(w,b,P,T,lr);
```

ADAPTWH Adapt linear layer with Widrow-Hoff rule.

```
[A,E,W,B] = ADAPTWH(W,B,P,T,lr)
```

W - $S \times R$ weight matrix.

B - $S \times 1$ bias vector.

P - $R \times Q$ matrix of input vectors.

T - $S \times Q$ matrix of target vectors.

lr - Learning rate (optional, default = 0.1).

Returns:

A - output of adaptive linear filter.

E - error of adaptive linear filter.

W - new weight matrix

B - new weights & biases.

PLOTTING THE OUTPUT SIGNAL

Here the output signal of the linear neuron is plotted with the target signal.

```
plot(time,a,time,T,'--')
xlabel('Time','Output ____ Target __','Output and Target Signals')
```

It does not take the adaptive neuron long to figure out how to generate the target signal.

A plot of the difference between the neurons output signal and the target signal shows how well the adaptive neuron works.

```
plot(time,e,[min(time) max(time)],[0 0],':r')
xlabel('Time','Error','Error Signal')
```

3. Backpropagation networks

3.1. *Pattern association with a linear neuron*

INITFF - Initializes a feed-forward network.

TRAINBP - Trains a feed-forward network with backpropagation.

SIMUFF - Simulates a feed-forward network.

Using the above functions a neuron is trained to respond to specific inputs with target outputs.

DEFINING A VECTOR ASSOCIATION PROBLEM

P defines two 1-element input vectors (column vectors):

```
P = [-3.0 +2.0];
```

T defines the associated 1-element targets (column vectors):

```
T = [+0.4 +0.8];
```

PLOTTING THE ERROR SURFACE AND CONTOUR

ERRSURF calculates errors for a neuron with a range of possible weight and bias values. PLOTES plots this error surface with a contour plot underneath.

```
wv = -4:0.4:4;
bv = -4:0.4:4;
es = errsrf(P,T,wv,bv,'logsig');
plotes(wv,bv,es,[60 30]);
```

The best weight and bias values are those that result in the lowest point on the error surface.

DESIGN THE NETWORK

INITFF is used to initialize the weights and biases for the LOGSIG neuron.

```
[w,b] = initff(P,T,'logsig')
```

INITFF Initialize feed-forward network up to 3 layers.

`[W1,B1,...] = INITFF(P,S1,'F1',...,Sn,'Fn')`

P - Rx2 matrix of input vectors.

Si - Size of ith layer.

Fi - Transfer function of the ith layer (string).

Returns:

Wi - Weight matrix of the ith layer.

Bi - Bias (column) vector of the ith layer.

TRAINING THE NETWORK

TBP1 uses backpropagation to train 1-layer networks.

`df = 5;` Frequency of progress displays (in epochs).

`me = 100;` Maximum number of epochs to train.

`eg = 0.01;` Sum-squared error goal.

`lr = 2;` Learning rate.

`[w,b,ep,tr] = tbp1(w,b,'logsig',P,T,[df me eg lr],wv,bv,es,[60 30]);`

TBP1 Train 1-layer feed-forward network w/backpropagation.

`[W,B,TE,TR] = TBP1(W,B,F,P,T,TP)`

W - SxR weight matrix.

B - Sx1 bias vector.

F - Transfer function (string).

P - RxQ matrix of input vectors.

T - SxQ matrix of target vectors.

TP - Training parameters (optional).

Returns:

W - new weights.

B - new biases.

TE - the actual number of epochs trained.

TR - training record: [row of errors]

Training parameters are:

TP(1) - Epochs between updating display, default = 25.

TP(2) - Maximum number of epochs to train, default = 1000.

TP(3) - Sum-squared error goal, default = 0.02.

TP(4) - Learning rate, 0.01.

Missing parameters and NaN's are replaced with defaults.

TRAINBP has returned new weight and bias values, the number of epochs trained EP, and a record of training errors TR.

PLOTTING THE ERROR CURVE

Here the errors are plotted with respect to training epochs:

`ploterr(tr,eg);`

USING THE PATTERN ASSOCIATOR

We can now test the associator with one of the original inputs, -3, and see if it returns the target, 0.4.

```
p = -1.2;  
a = simuff(p,w,b,'logsig')
```

Training to a lower error goal would reduce this error.

Use SIMUP to check the neuron for an input of 2.0. The target response is 0.8.