Multilayer Perceptron

Chapter Intended Learning Outcomes:

(i) Understand multilayer perceptron and its properties

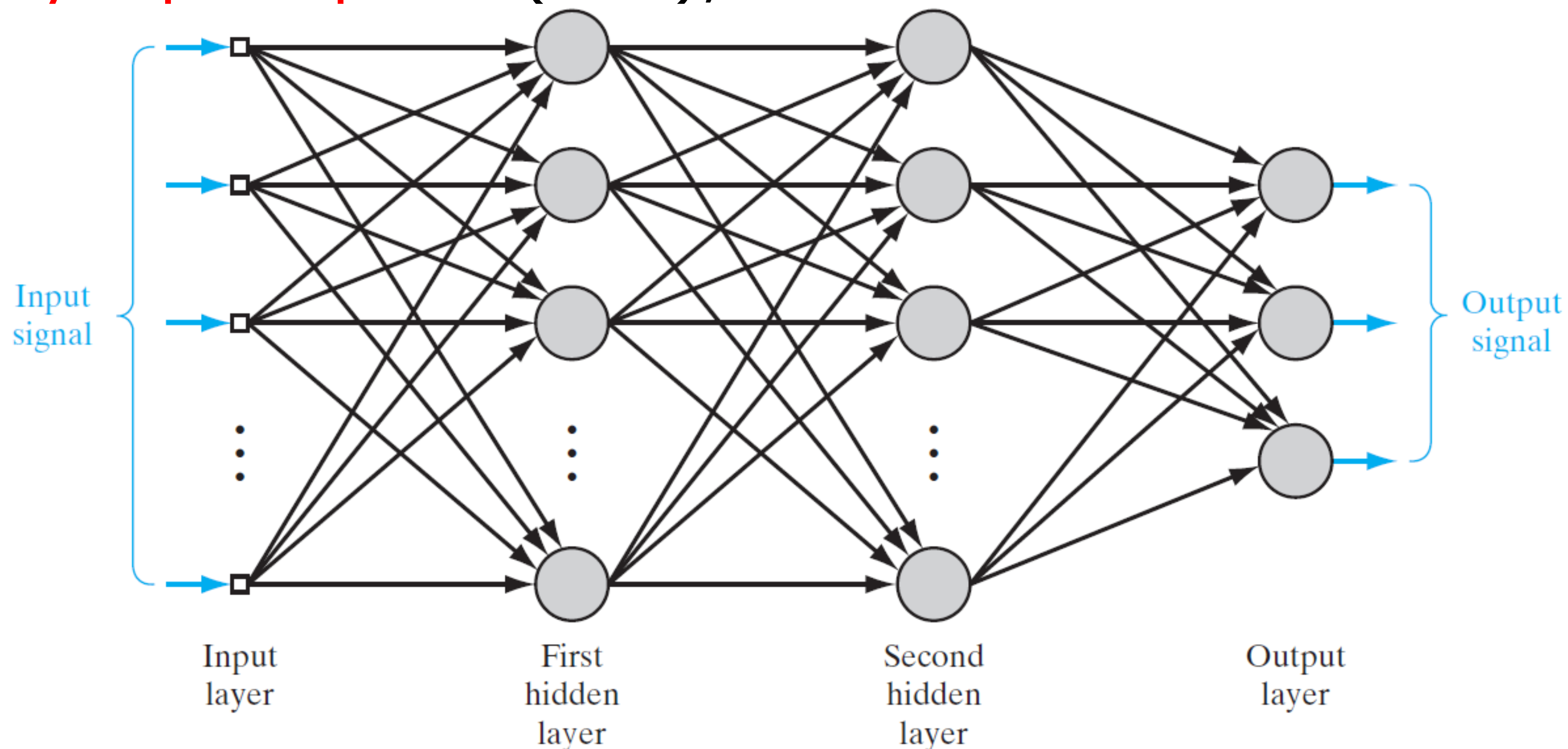
(ii) Understand back propagation algorithm for training multilayer perceptron

(iii) Investigate underfitting and overfitting as well as bias-variance tradeoff in machine learning

Structure of Multilayer Perception

The perceptron and ADALINE have only one artificial neuron, and basically can only perform binary classification in the linearly separable case.

Multiple neurons with multiple layers, referred to as multilayer perceptron (MLP), can overcome these limitations.



This is also referred to as feedforward neural network (NN). Here, “feedforward” means that signals flow from the input layer to the output layer in a forward direction.

It consists of:

- Input layer: Contains the inputs including the bias as in perceptron. No computation is performed.
- Hidden layer(s): There is at least one hidden layer with hidden neurons. Each hidden neuron is a perceptron, performing linear combination and then activation.
- Output layer: Contains output neurons performing computation as in perceptron.

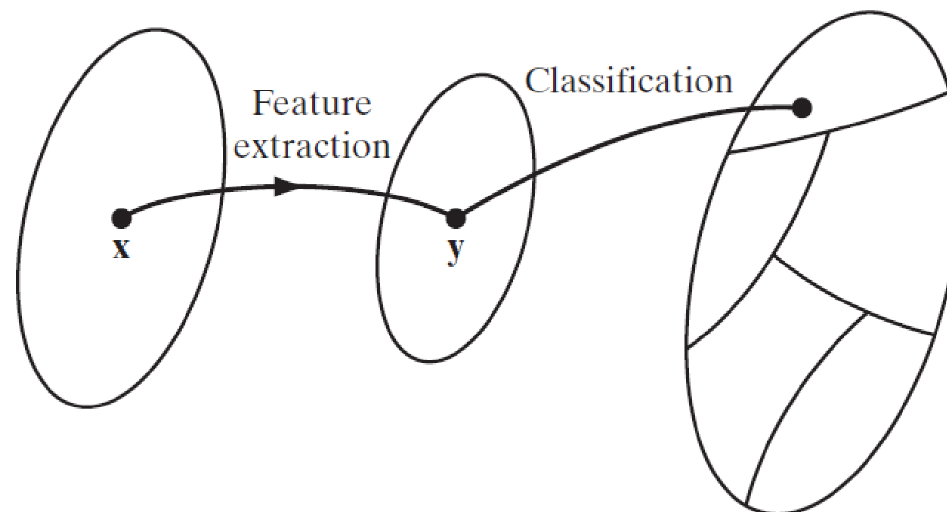
We can call the example network a 2-hidden-layer MLP or NN.

The network is fully connected, i.e., a neuron in any layer is connected to all the neurons or nodes in the previous layer.

If there are no hidden layers and only one output neuron using sign activation function, NN reduces to perceptron.

The hidden neurons act as feature detectors by performing nonlinear transformation on the input data into a feature space such that different classes may be more easily separated than using the input data space. As perceptron has no hidden neuron, no feature space is formed.

Instead of directly using the input in the input layer, features can be used when there is additional feature extraction step.



Typical Activation Functions in Neural Networks

Activation function is a nonlinear and differentiable function.

Logistic function:

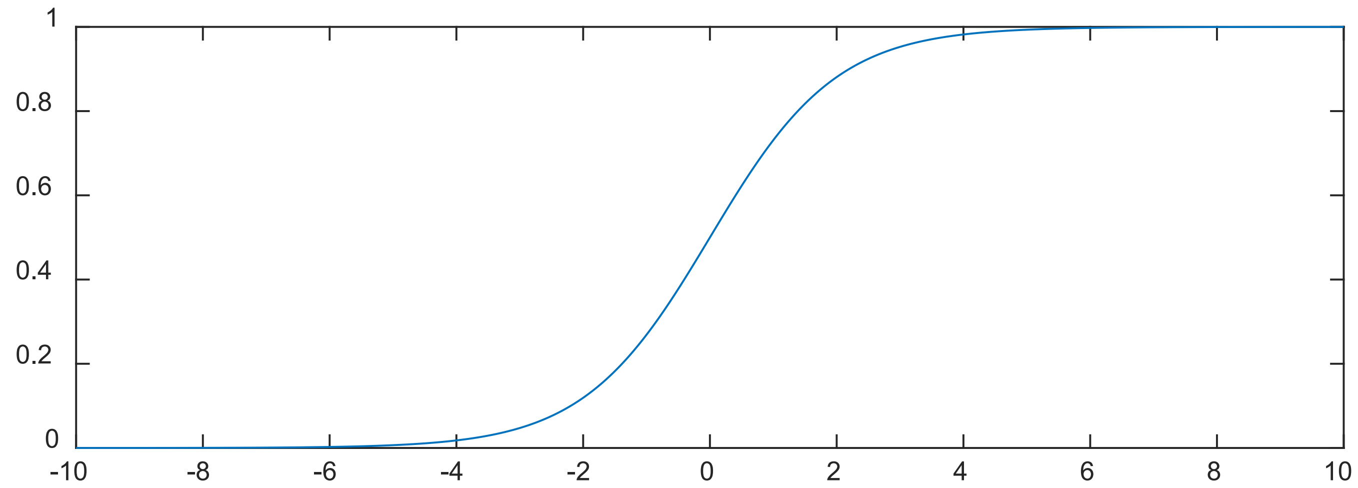
$$y = \varphi(v) = \frac{1}{1+e^{-av}}, \quad a > 0 \tag{1}$$

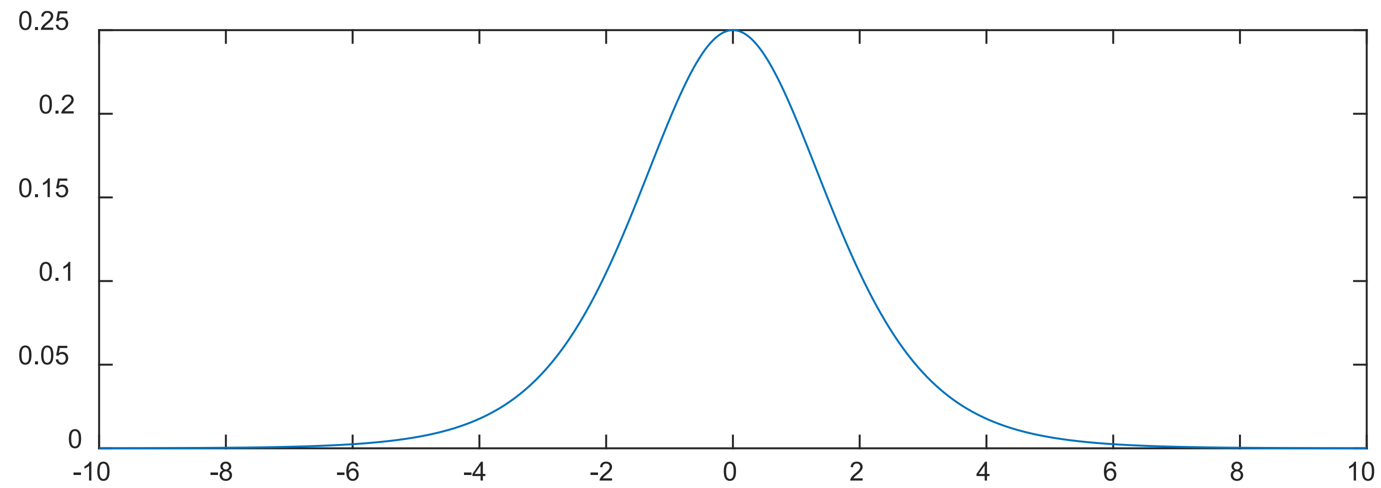
```
>> syms f a v
>> f=1/(1+exp(-a*v));
>> diff(f,v)
ans = (a*exp(-a*v))/(exp(-a*v) + 1)^2
```

$$\varphi'(v) = \frac{ae^{-av}}{(1+e^{-av})^2} = a \cdot \frac{1}{1+e^{-av}} \cdot \frac{e^{-av}}{1+e^{-av}} = ay(1-y) \tag{2}$$

The simplest setting corresponds to $a = 1$.

At $a=1$:





Hyperbolic tangent function:

$$y = \varphi(v) = a\tanh(bv), \quad a > 0,\ b > 0 \tag{3}$$

which can be considered as a modified logistic function.

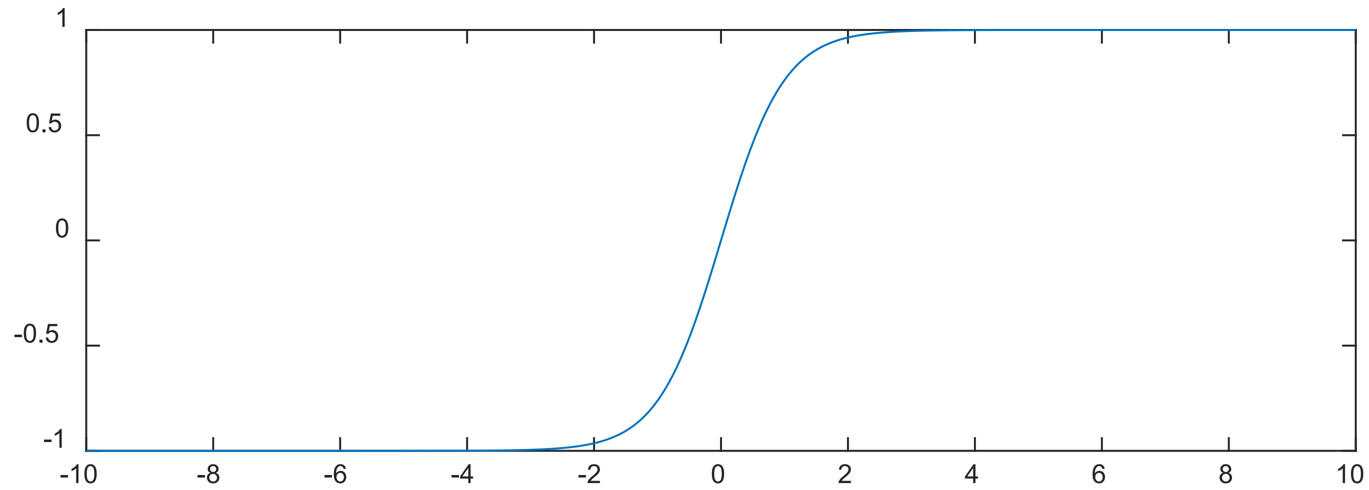
```
>> syms f a b v
>> f=a*tanh(b*v);
>> diff(f,v)
ans = -a*b*(tanh(b*v)^2 - 1)
```

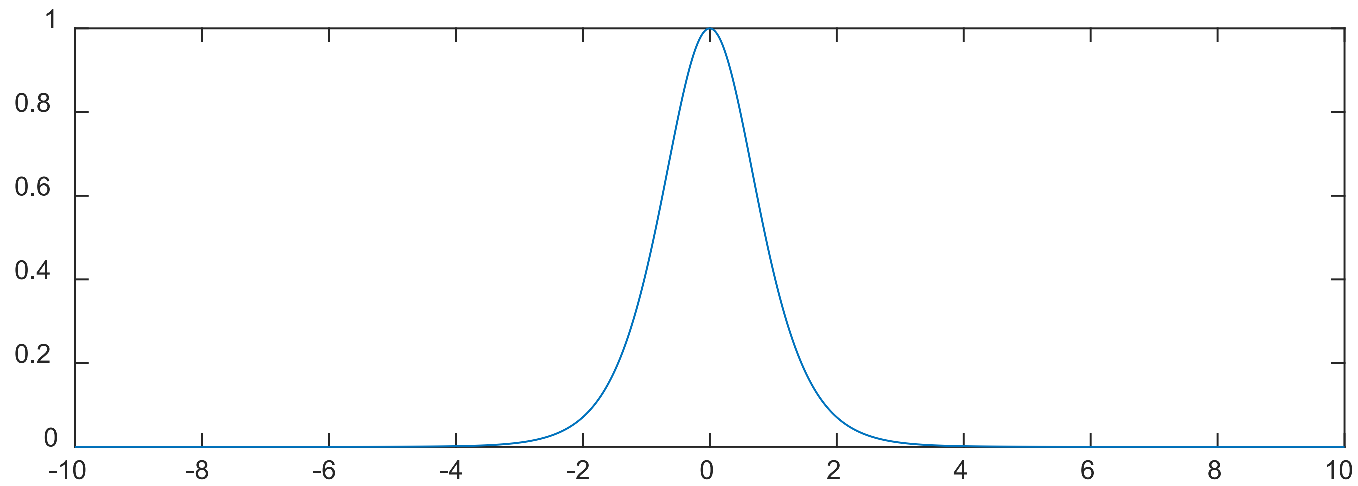
$$\varphi'(v) = ab[1 - \tanh^2(bv)] = \frac{b}{a}[a^2 - a^2\tanh^2(bv)] = \frac{b}{a}(a-y)(a+y) \tag{4}$$

The simplest setting corresponds to $a = b = 1$:

$$\varphi(v) = \frac{e^v - e^{-v}}{e^v + e^{-v}} = \frac{1 - e^{-2v}}{1 + e^{-2v}}, \quad \varphi'(v) = [1 - \tanh^2(v)] = (1-y)(1+y) \tag{5}$$

At $a = b = 1$:





Rectified linear function:

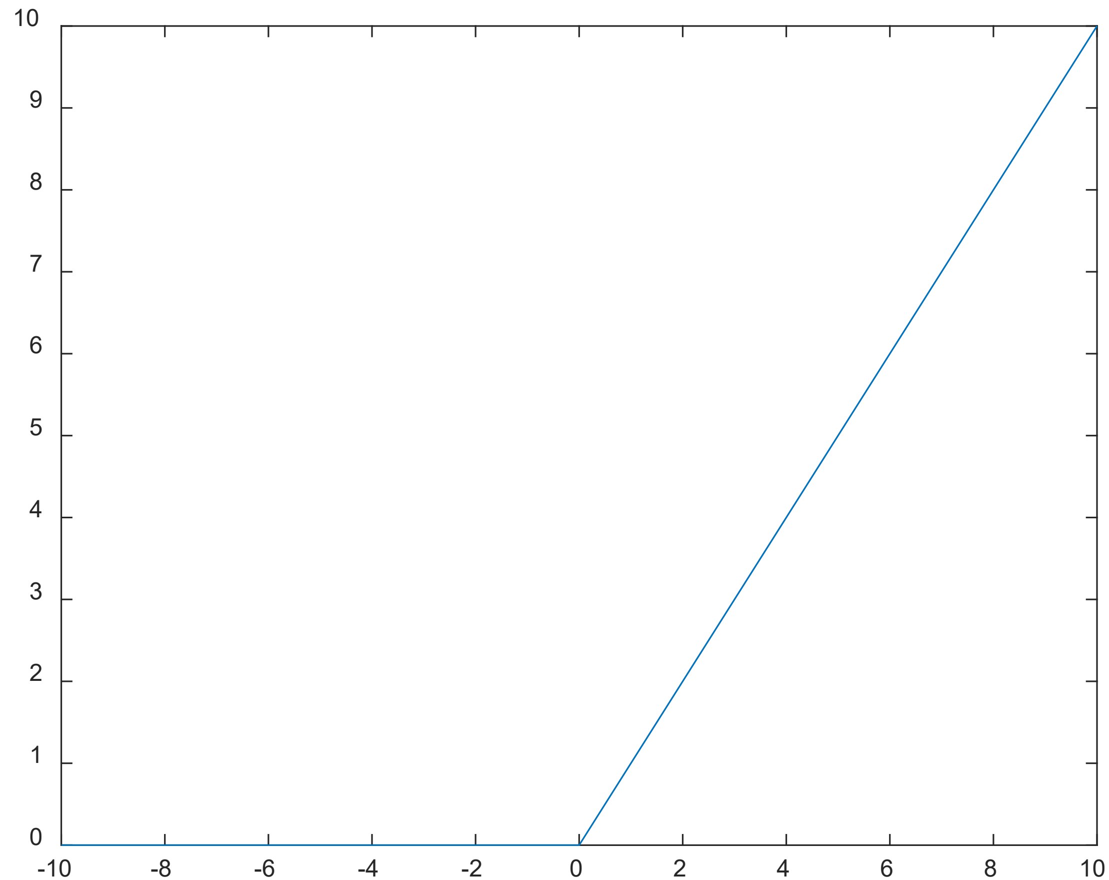
$$y = \varphi(v) = \max(0, v) = \begin{cases} v, & v \geq 0 \\ 0, & v < 0 \end{cases} \quad (6)$$

It is also known as ReLU, which stands for rectified linear unit.

$$\varphi'(v) = \begin{cases} 1, & v > 0 \\ 0, & v < 0 \\ [0,1], & v = 0 \end{cases} \quad (7)$$

Note that at $v = 0$, there is no unique derivative because at $v \to 0^+$, the slope is 1 while that of $v \to 0^-$ is 0. We may ignore this case because $v = 0$ is unlikely to occur.

```
>> v=-10:0.01:10;
>> f=max(v,0);
>> plot(v,f)
```



The activation functions and their derivatives should be easy to compute to facilitate the output computation and weight determination.

Back Propgation Algorithm

As in single neuron case, we need to determine the weights in all hidden and output layers, denoted by $\boldsymbol{w}$, for the classification model.

Suppose we have training input-output dataset $\{\boldsymbol{x}(n), \boldsymbol{d}(n)\}_{n=1}^{N}$.

A cost function $J(\boldsymbol{w})$ is first needed, from which $\boldsymbol{w}$ is determined via its optimization.

We start with the batch mode and establish an error function $e_j(n)$ using the nth training pair at the output neuron j:

$$e_j(n) = d_j(n) - y_j(n)$$

where $y_j(n)$ is the actual NN output and we may write $y_j(n) = f_j(\boldsymbol{x}(n), \boldsymbol{w})$ such that f_j is nonlinear.

According to least squares (LS) criterion, the cost function based on the nth training pair at all output neurons is:

$$J_n(\boldsymbol{w}) = \frac{1}{2}\sum_j e_j^2(n) = \frac{1}{2}\sum_j (d_j(n) - y_j(n))^2 \tag{8}$$

Together with all N training data, the overall cost function is:

$$J(\boldsymbol{w}) = \sum_{n=1}^{N} J_n(\boldsymbol{w}) = \frac{1}{2}\sum_{n=1}^{N}\sum_j e_j^2(n) = \frac{1}{2}\sum_{n=1}^{N}\sum_j (d_j(n) - y_j(n))^2 \tag{9}$$

and the LS estimate is:

$$\hat{\boldsymbol{w}} = \arg\min_{\boldsymbol{w}} J(\boldsymbol{w})$$

However, it is difficult to perform weight computation even for minimizing $J_n(\boldsymbol{w})$ as it is a nonlinear regression problem. It is because the weights in a hidden layer will undergo multiple nonlinear operations, and $J_n(\boldsymbol{w})$ has multiple minima.

Standard solution is back propagation (BP) algorithm which aims to minimize $J_n(\boldsymbol{w})$ in online mode via gradient descent.

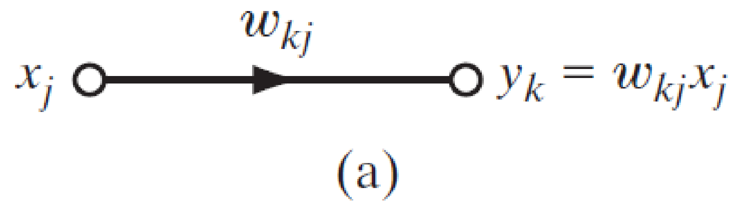
For each epoch, we randomize the order in the training set $\{\boldsymbol{x}(n), \boldsymbol{d}(n)\}_{n=1}^{N}$, and then update all weights at each n via minimizing $J_1(\boldsymbol{w})$, $J_2(\boldsymbol{w})$, …, $J_N(\boldsymbol{w})$, in a sequential manner.

At the end of each epoch, we compute $J(\hat{\boldsymbol{w}})$ and the algorithm terminates after a sufficient number of epochs.

At the output layer, the updating rule in minimizing $e_j^2(n)$ is similar to that of perceptron.

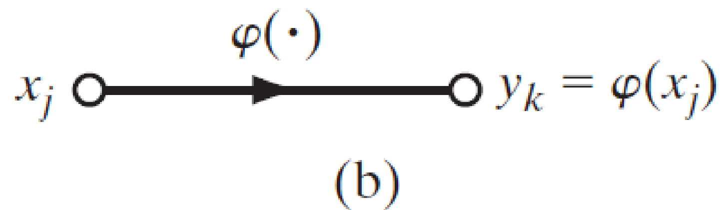
While in updating hidden layer neurons, there are no desired signals like $\boldsymbol{d}(n)$, and propagation of $e_j^2(n)$ is involved.

To ease presentation, signal flow graph which consists of only nodes and directed branches, is introduced:



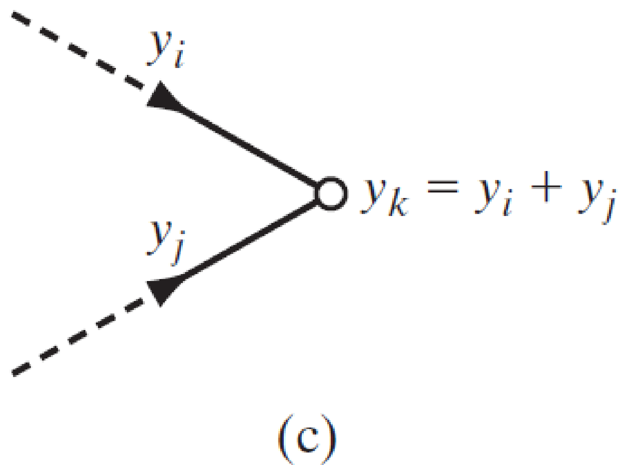
(a)

Output node signal is equal to the branch gain times input node signal.



(b)

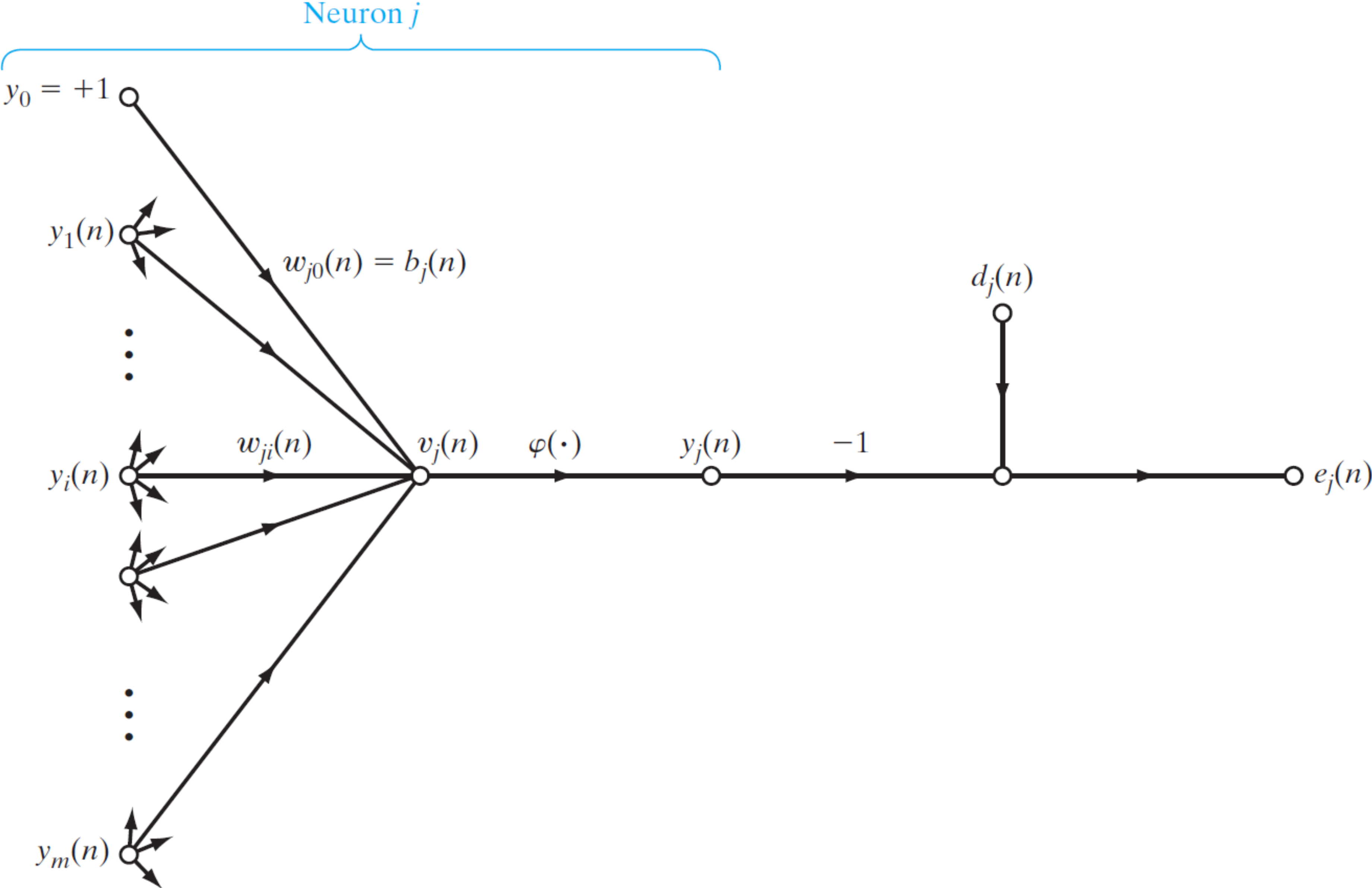
Branch gain can mean a function.



(c)

Signal at a node of a flow graph is equal to the sum of the signals from all branches connecting to the node.

Consider output neuron j:



Looking from right to left:

$$e_j(n) = d_j(n) - y_j(n)$$

$$y_j(n) = \varphi(v_j(n))$$

$$v_j(n) = \sum_{i=0}^{m} w_{ji}(n) y_i(n) \qquad (10)$$

Like perceptron, bias $w_{j0}(n)$ is absorbed in (10) and difference is that the input corresponds to hidden layer with $\{y_i(n)\}$.

Applying the idea of LMS algorithm with chain rule:

$$\begin{aligned}\frac{\partial J_n(\boldsymbol{w})}{\partial w_{ji}(n)} &= \frac{\partial J_n(\boldsymbol{w})}{\partial e_j(n)} \cdot \frac{\partial e_j(n)}{\partial y_j(n)} \cdot \frac{\partial y_j(n)}{\partial v_j(n)} \cdot \frac{\partial v_j(n)}{\partial w_{ji}(n)} \\ &= e_j(n) \cdot -1 \cdot \varphi_j'(v_j(n)) \cdot y_i(n) \\ &= -e_j(n)\varphi_j'(v_j(n)) y_i(n) \qquad (11)\end{aligned}$$

The iterative procedure is called delta rule:

$$w_{ji}(n+1) = w_{ji}(n) + \Delta w_{ji}(n), \quad \Delta w_{ji}(n) = -\eta \frac{\partial J_n(\boldsymbol{w})}{\partial w_{ji}(n)} \qquad (12)$$

where $\eta > 0$ is the learning rate parameter. Defining local gradient $\delta_j(n)$:

$$\delta_j(n) = \frac{\partial J_n(\boldsymbol{w})}{\partial v_j(n)} = \frac{\partial J_n(\boldsymbol{w})}{\partial e_j(n)} \cdot \frac{\partial e_j(n)}{\partial y_j(n)} \cdot \frac{\partial y_j(n)}{\partial v_j(n)} = -e_j(n)\varphi_j'(v_j(n)) \qquad (13)$$

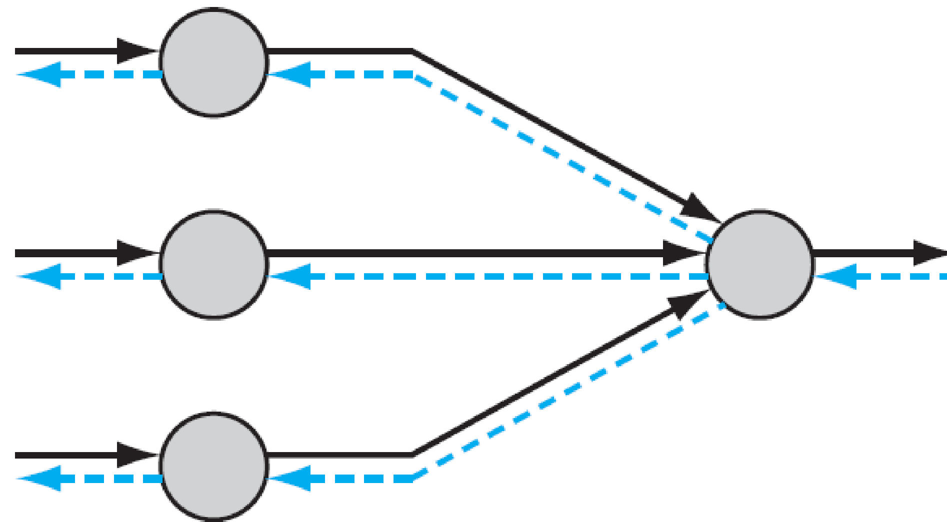
We can write:

$$\Delta w_{ji}(n) = \eta \delta_j(n) y_i(n) \qquad (14)$$

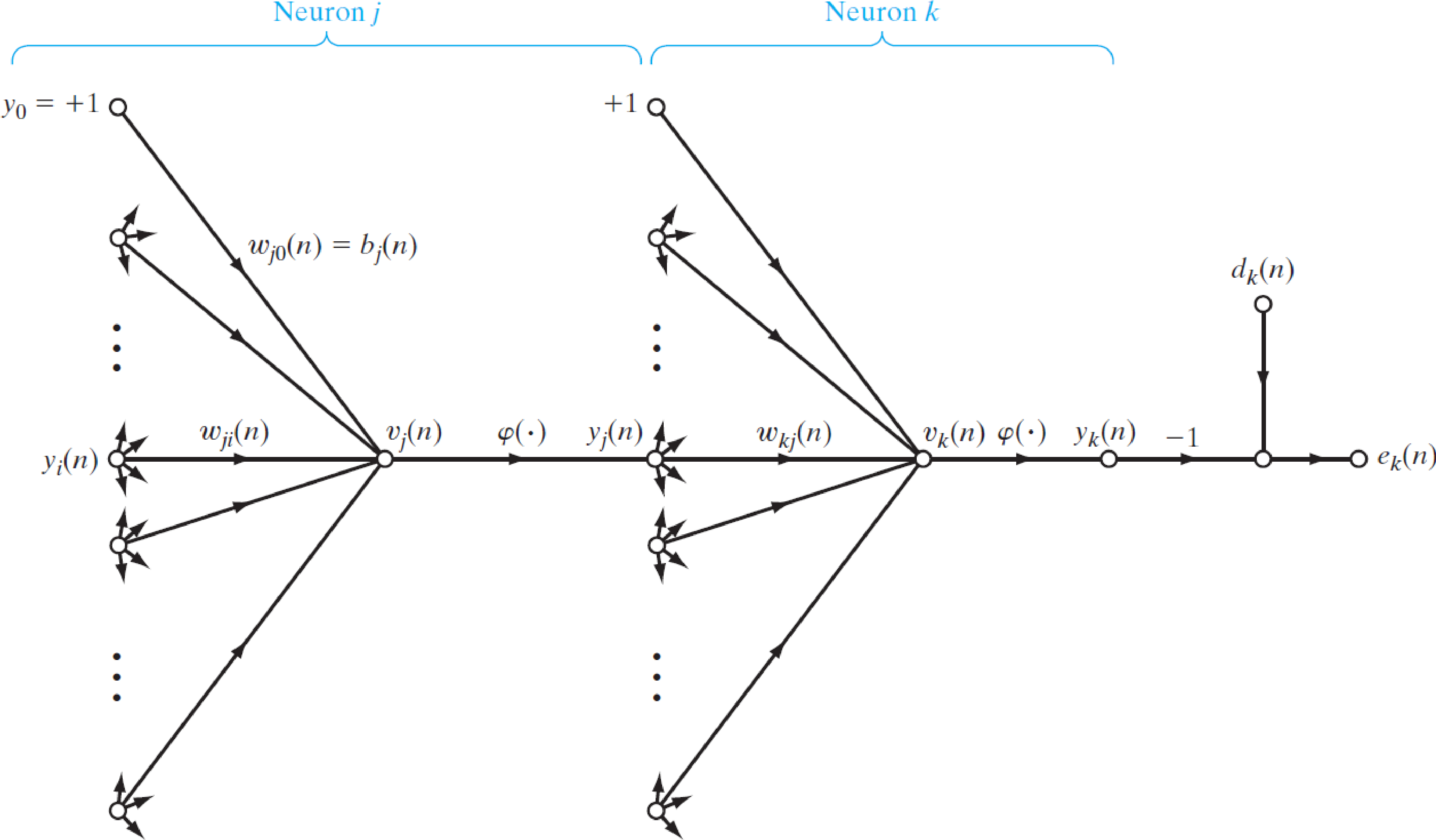
$\{w_{ji}(n+1)\}$ or $\boldsymbol{w}(n+1)$ will then be used to compute $\{y_i(n+1)\}$ $\{y_j(n+1)\}$, and $e_j(n+1)$ with $\{\boldsymbol{x}(n+1), \boldsymbol{d}(n+1)\}$ in a feedforward mode.

That is, for each input-output pair, there are two passes of computation:

- Forward pass: compute outputs at all hidden and output layers
- Backward pass: update weights by passing the error from the output layer to first hidden layer.



Consider hidden neuron j:



To avoid confusion, index k is now used in output layer. Now we write:

$$J_n(\boldsymbol{w}) = \frac{1}{2}\sum_k e_k^2(n), \quad e_k(n) = d_k(n) - y_k(n) = d_k(n) - \varphi_k(v_k(n)) \quad (15)$$

The correction term $\Delta w_{ji}(n)$ in the delta rule still has the form as in (14) but the local gradient becomes:

$$\delta_j(n) = \frac{\partial J_n(\boldsymbol{w})}{\partial v_j(n)} = \frac{\partial J_n(\boldsymbol{w})}{\partial y_j(n)} \cdot \frac{\partial y_j(n)}{\partial v_j(n)} = \sum_k e_k(n)\frac{\partial e_k(n)}{\partial y_j(n)}\varphi_j'(v_j(n)) \quad (16)$$

To find $\partial e_k(n)/\partial y_j(n)$, we need to relate with $v_k(n)$:

$$v_k(n) = \sum_{j=0}^{m} w_{kj}(n)y_j(n) \Rightarrow \frac{\partial v_k(n)}{\partial y_j(n)} = w_{kj}(n)$$

Employing (15) and applying chain rule again:

$$\frac{\partial e_k(n)}{\partial y_j(n)} = \frac{\partial e_k(n)}{\partial v_k(n)} \cdot \frac{\partial v_k(n)}{\partial y_j(n)} = -\varphi_k'(v_k(n))w_{kj}(n) \tag{17}$$

With the use of (13) and (17), (16) becomes:

$$\begin{aligned}
\delta_j(n) &= \sum_k e_k(n)\frac{\partial e_k(n)}{\partial y_j(n)}\varphi_j'(v_j(n)) \\
&= \sum_k e_k(n) \cdot -\varphi_k'(v_k(n))w_{kj}(n) \cdot \varphi_j'(v_j(n)) \\
&= \varphi_j'(v_j(n)) \sum_k [-e_k(n)\varphi_k'(v_k(n))]w_{kj}(n) \\
&= \varphi_j'(v_j(n)) \sum_k \delta_k(n)w_{kj}(n)
\end{aligned} \tag{18}$$

To summarize, $\Delta w_{ji}(n)$ is computed as:

$$\Delta w_{ji}(n) = \eta \delta_j(n) y_i(n)$$

where the local gradient is given by (13) for output node and (18) for hidden node.

To specify $y_i(n)$ at the input and output layers, respectively, we can write:

$$y_i(n) = x_i(n)$$

$$y_i(n) = o_i(n)$$

Assuming logistic activation function and using (2) and (13), the local gradient at output node is:

$$\begin{aligned} \delta_j(n) &= -e_j(n)\varphi_j'(v_j(n)) = -[d_j(n) - o_j(n)] \cdot a o_j(n)[1 - o_j(n)] \\ &= -a[d_j(n) - o_j(n)] o_j(n)[1 - o_j(n)] \end{aligned}$$

While the local gradient at hidden node is

$$\delta_j(n) = \varphi_j'(v_j(n)) \sum_k \delta_k(n) w_{kj}(n)$$
$$= a y_j(n)[1 - y_j(n)] \sum_k \delta_k(n) w_{kj}(n)$$

From (12), the performance of the delta rule depends on η which has a tradeoff between convergence rate and stability. In practice, a large η is adopted at the beginning to achieve fast convergence while using a small η upon convergence. Note that we can set different values of η at different layers.

(12) can also be improved by adding momentum term:

$$\Delta w_{ji}(n) = \alpha \Delta w_{ji}(n-1) + \eta \delta_j(n) y_i(n), \quad 1 > \alpha \geq 0 \qquad (19)$$

which is also called generalized delta rule, and α is the momentum constant.

Solving (19) yields:

$$\Delta w_{ji}(n) = \eta \sum_{t=0}^{n} \alpha^{n-t} \delta_j(t) y_i(t) = -\eta \sum_{t=0}^{n} \alpha^{n-t} \frac{\partial J_t(\boldsymbol{w})}{\partial w_{ji}(t)}$$

which is an exponentially weighed sum of gradients.

The improvement might be explained:

- Large amount of adjustment when the gradient $\partial J_t(\boldsymbol{w})/\partial w_{ji}(t)$ has the same sign for consecutive updates, leading to convergence acceleration.
- Small amount of adjustment when the gradient $\partial J_t(\boldsymbol{w})/\partial w_{ji}(t)$ has the opposite signs for consecutive updates, leading to a stabilizing effect.

Compared with (12), the momentum term may also prevent learning process from terminating in a shallow local minimum.

For a NN with L layer, the BP can be summarized as:

1. Randomly initialize all weights $\{\Delta w_{ji}(0)\}$.
2. For each epoch, randomize the order in $\{\boldsymbol{x}(n), \boldsymbol{d}(n)\}_{n=1}^{N}$, and we may consider $n = 1, \cdots, N, N+1, \cdots, 2N, ...$ as time index.
3. For each n, we perform
   (i) Forward computation:
   Compute signals at layer l using outputs at layer $l-1$

$$v_j^{(l)}(n) = \sum_{i=0}^{m} w_{ji}^{(l)}(n) y_i^{(l-1)}(n), \quad y_j^{(0)} = x_j(n), \quad y_j^{(L)} = o_j(n)$$

$$y_j^{(l)}(n) = \varphi(v_j^{(l)}(n))$$

   Compute error signals at output layer

$$e_j(n) = d_j(n) - o_j(n)$$

(ii) Backward computation:
Compute local gradients

$$\delta_j(n) = \begin{cases} -e_j^{(L)}(n)\varphi_j'(v_j^{(L)}(n)), & \text{output layer} \\ \varphi_j'(v_j^{(l)}(n))\sum_k \delta_k^{(l+1)}(n)w_{kj}^{(l+1)}(n), & \text{hidden layer} \end{cases}$$

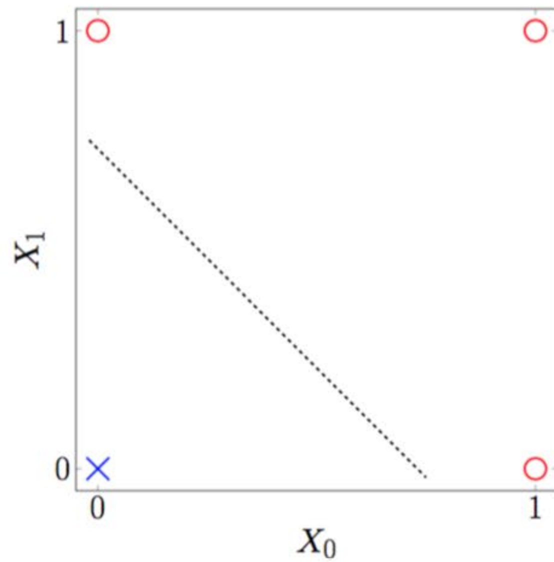
Update weights

$$w_{ji}^{(l)}(n+1) = w_{ji}^{(l)}(n) + \alpha\Delta w_{ji}^{(l)}(n-1) + \eta\delta_j^{(l)}(n)y_i^{(l-1)}(n)$$

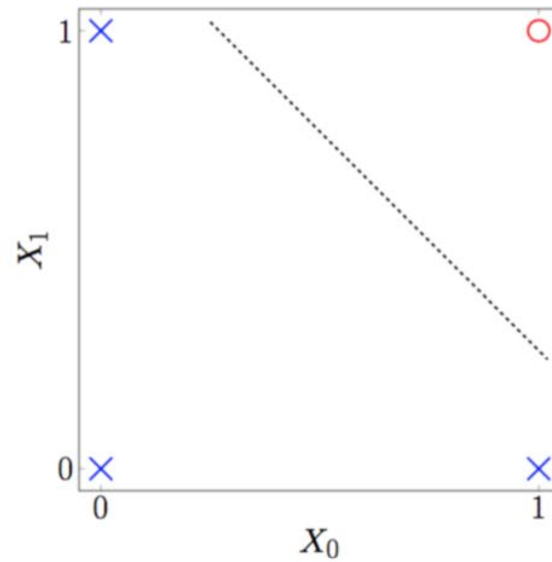
4. Repeat Step 3 until a stopping criterion is reached.

At each epoch, an input-output pair is picked at random in performing the gradient update, which is also referred to as stochastic gradient descent (SGD).

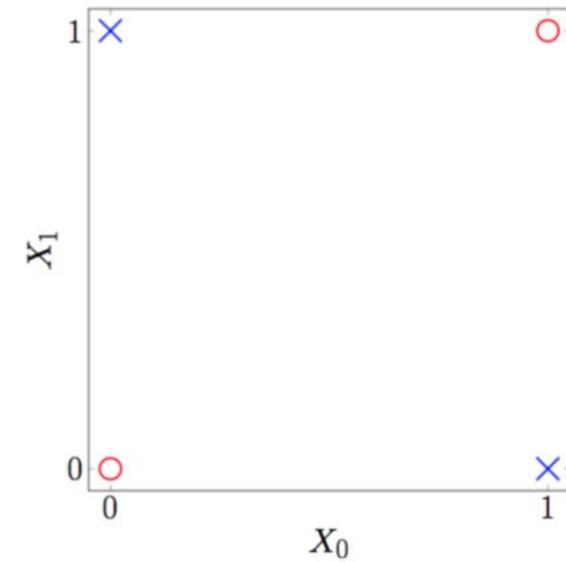
Apart from SGD, mini-batch gradient descent is also popular where a subset of training data is involved per iteration.



(a) OR



(b) AND



(c) XOR

Now we apply NN to implement the nonlinearly separable XOR operator:

$$\begin{aligned} 0 \oplus 0 = 0 \\ 1 \oplus 1 = 0 \\ 0 \oplus 1 = 1 \\ 1 \oplus 0 = 1 \end{aligned}$$

Perform XOR function without momentum

stdBPXOR.m

```
in_dim=2;
out_dim=1;
x=[-1 1 -1 1;-1 -1 1 1];
t=[1 -1 -1 1];
no_h=5;
net = feedforwardnet(no_h,'traingd');
net = configure(net,x,t);
rand('seed',1997);
IW = 1*(rand(no_h,in_dim)-0.5);
b1 = 1*(rand(no_h,1)-0.5);
LW = 1*(rand(out_dim,no_h)-0.5);
b2 = 1*(rand(out_dim,1)-0.5);
net.IW{1,1} = IW;
net.b{1,1} = b1;
net.LW{2,1} = LW;
net.b{2,1} = b2;
net.layers{1}.transferFcn = 'tansig';
net.layers{2}.transferFcn ='tansig';
net.trainParam.epochs = 1000;
net.trainParam.show = 50;
net.trainParam.lr = 0.5;
net.trainParam.goal = 0;
net.trainParam.min_grad=0;
net.divideFcn = '';
[net,tr] = train(net,x,t);
a=net(x)
```

Neural Network Training (nntraint...

Neural Network

Input 2 → Hidden (W, b) 5 → Output (W, b) 1 → Output 1

Algorithms

Training: Gradient Descent (traingd)
Performance: Mean Squared Error (mse)
Calculations: MEX

Progress

Epoch:	0	1000 iterations	1000
Time:		0:00:00	
Performance:	1.05	0.000406	0.00
Gradient:	0.371	0.000932	0.00

Plots

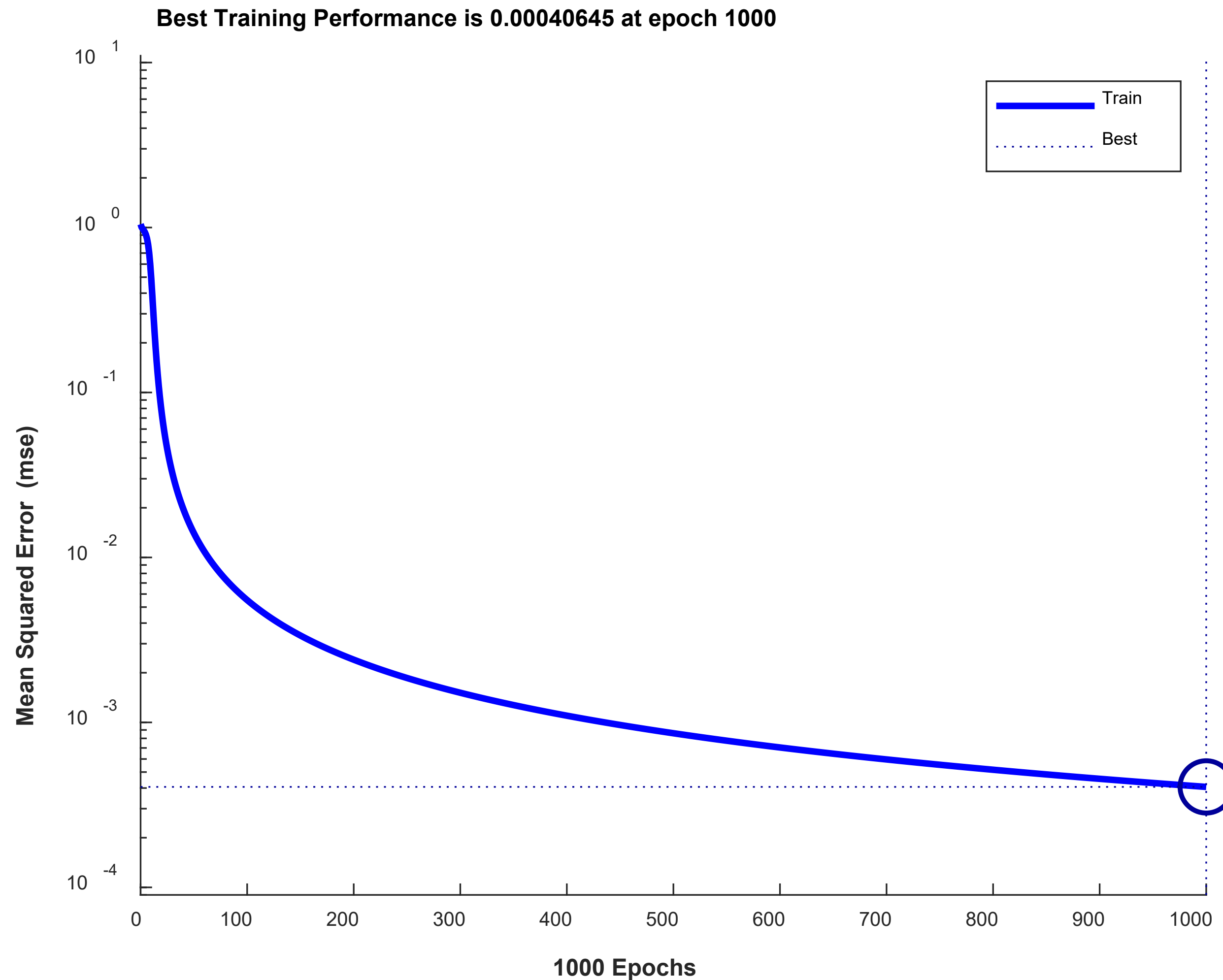
Performance (plotperform)
Training State (plottrainstate)
Error Histogram (ploterrhist)
Regression (plotregression)

Plot Interval: 1 epochs

Opening Training State Plot

Stop Training Cancel

```
>> stdBPXOR
a =       0.9773    -0.9826    -0.9829    0.9773
```



https://www.mathworks.com/help/deeplearning/ref/traingd.html

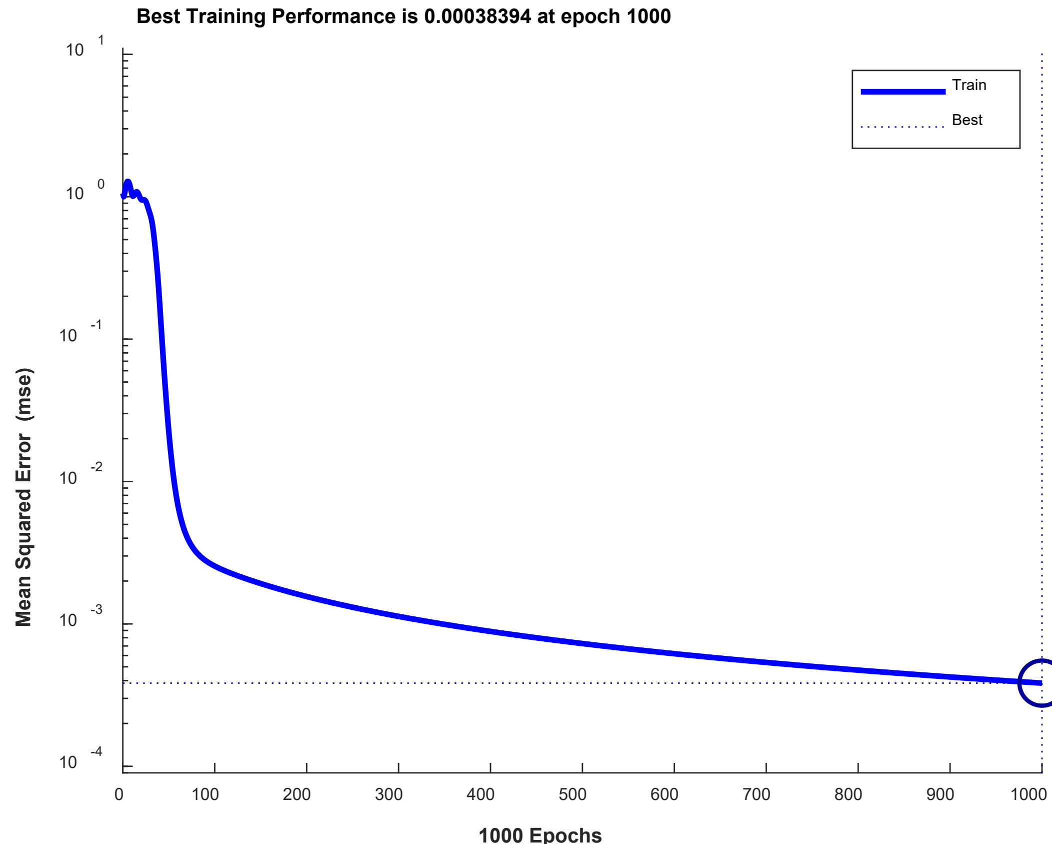
```
dX = lr * dperf/dX
```

Perform XOR function with momentum

stdBPXOR.m | stdmomBPXOR.m

```
1 - in_dim=2;
2 - out_dim=1;
3 - x=[-1 1 -1 1;-1 -1 1 1];
4 - t=[1 -1 -1 1];
5 - no_h=4;
6 - net = feedforwardnet(no_h,'traingdm');
7 - net = configure(net,x,t);
8 - rand('seed',1997);
9 - IW = 1*(rand(no_h,in_dim)-0.5);
10 - b1 = 1*(rand(no_h,1)-0.5);
11 - LW = 1*(rand(out_dim,no_h)-0.5);
12 - b2 = 1*(rand(out_dim,1)-0.5);
13 - net.IW{1,1} = IW;
14 - net.b{1,1} = b1;
15 - net.LW{2,1} = LW;
16 - net.b{2,1} = b2;
17 - net.layers{1}.transferFcn = 'tansig';
18 - net.layers{2}.transferFcn ='tansig';
19 - net.trainParam.epochs = 1000;
20 - net.trainParam.show = 50;
21 - net.trainParam.lr = 0.5;
22 - net.trainParam.mc = 0.9;
23 - net.trainParam.goal = 0;
24 - net.trainParam.min_grad=0;
25 - net.divideFcn = '';
26 - [net,tr] = train(net,x,t);
27 - a=net(x)
```

```
>> stdmomBPXOR
a =  0.9774   -0.9831   -0.9845    0.9777
```



https://www.mathworks.com/help/deeplearning/ref/traingdm.html

```
dX = mc*dXprev + lr*(1-mc)*dperf/dX
```

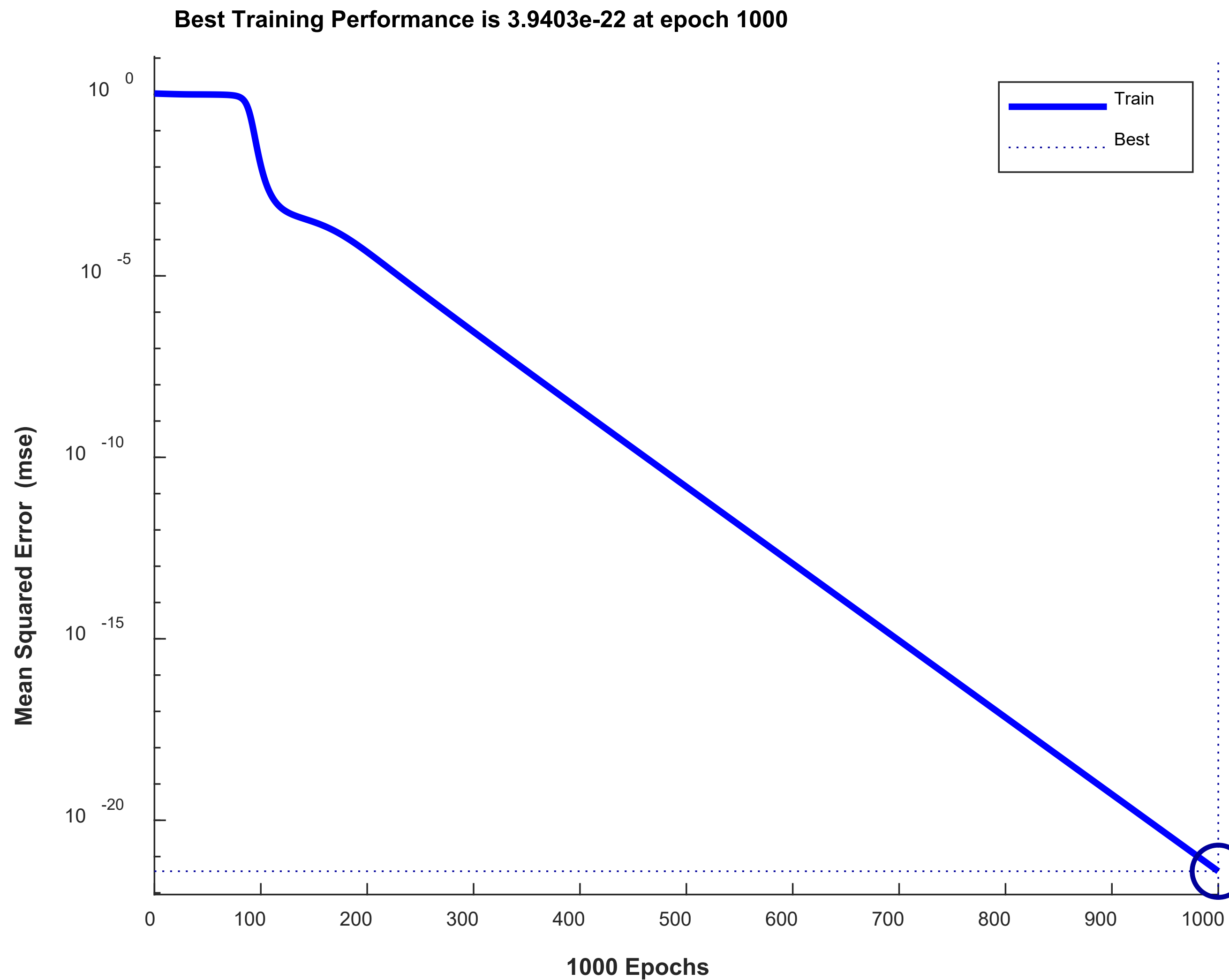
`mc` is set between 0 (no momentum) and values close to 1 (lots of momentum).

Perform XOR function with momentum and adaptive η

stdBPXOR.m | stdmomBPXOR.m | gdmxBPXOR.m

```
 2 -   out_dim=1;
 3 -   x=[-1 1 -1 1;-1 -1 1 1];
 4 -   t=[1 -1 -1 1];
 5 -   no_h=4;
 6 -   net = feedforwardnet(no_h,'traingdx');
 7 -   net = configure(net,x,t);
 8 -   rand('seed',1997);
 9 -   IW = 1*(rand(no_h,in_dim)-0.5);
10 -   b1 = 1*(rand(no_h,1)-0.5);
11 -   LW = 1*(rand(out_dim,no_h)-0.5);
12 -   b2 = 1*(rand(out_dim,1)-0.5);
13 -   net.IW{1,1} = IW;
14 -   net.b{1,1} = b1;
15 -   net.LW{2,1} = LW;
16 -   net.b{2,1} = b2;
17 -   net.layers{1}.transferFcn = 'tansig';
18 -   net.layers{2}.transferFcn ='tansig';
19 -   net.trainParam.epochs = 1000;
20 -   net.trainParam.show = 50;
21 -   net.trainParam.lr = 0.01;
22 -   net.trainParam.lr_inc=1.05;
23 -   net.trainParam.lr_dec=0.7;
24 -   net.trainParam.max_perf_inc=1.04;
25 -   net.trainParam.mc = 0.9;
26 -   net.trainParam.goal = 0;
27 -   net.trainParam.min_grad=0;
28 -   net.divideFcn = '';
29 -   [net,tr] = train(net,x,t);
30 -   a=net(x)
```

```
>> gdmxBPXOR
a =  1.0000   -1.0000   -1.0000    1.0000
```



https://www.mathworks.com/help/deeplearning/ref/traingdx.html

For each epoch I, if performance decreases toward the goal, i.e.,

$$J^{(I+1)}(\boldsymbol{w}) < J^{(I)}(\boldsymbol{w})$$

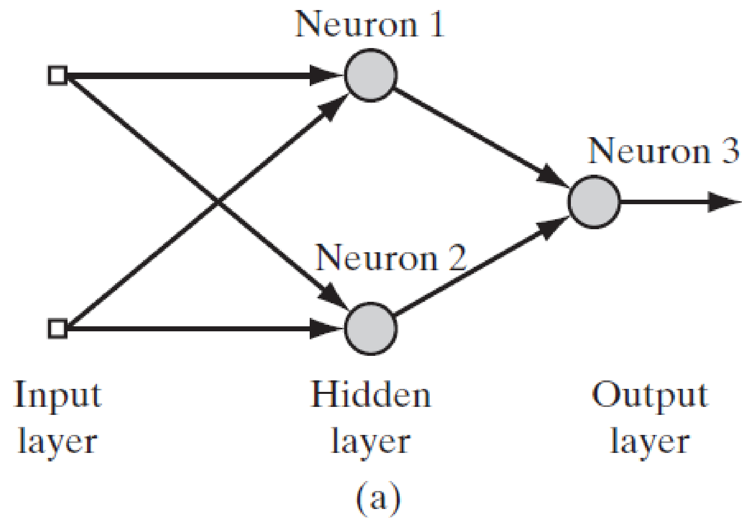
indicating that the mean square error (MSE) is gradually decreasing, then η is increased by the factor `lr_inc`.

If performance increases by more than the factor `max_perf_inc`, i.e.,

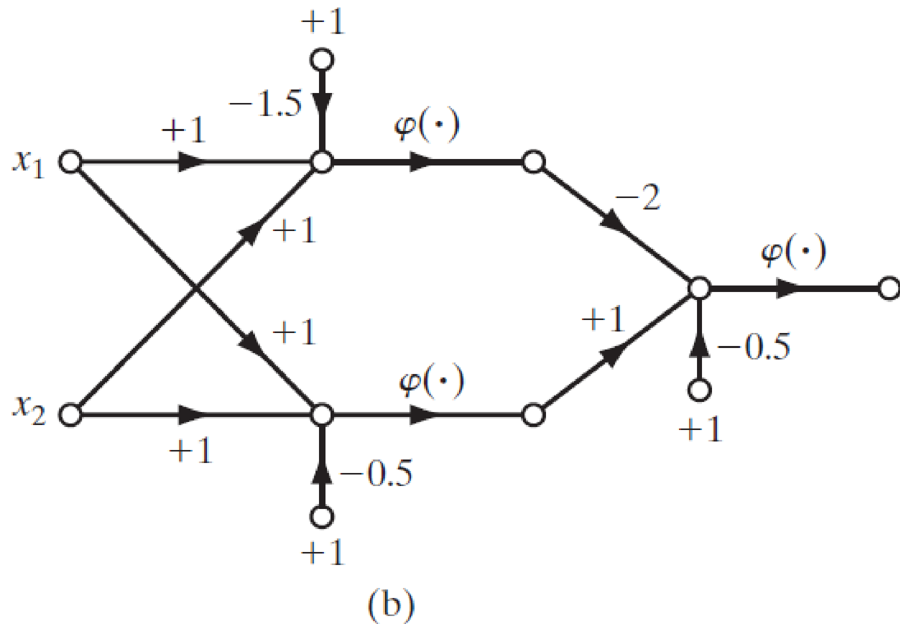
$$J^{(I+1)}(\boldsymbol{w}) > J^{(I)}(\boldsymbol{w}) * \texttt{max_perf_inc}$$

indicating the MSE is oscillating, then η is adjusted by the factor `lr_dec` and the change that increased the performance is not made, i.e., we keep the weights at epoch I for epoch $I+2$.

In fact, 2 hidden neurons are sufficient for XOR, e.g.,



(a)



(b)

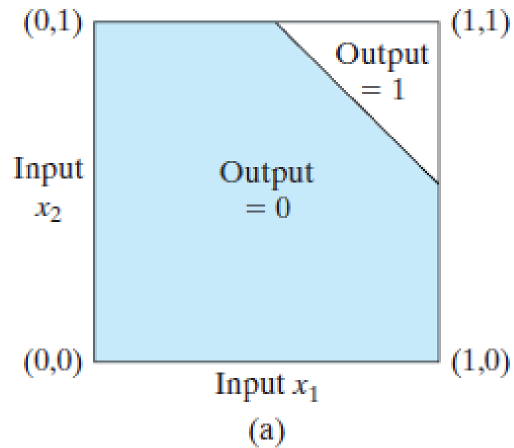
$$v_1 = x_1 + x_2 - 1.5$$
$$y_1 = \text{sign}(v_1)$$

$$v_2 = x_1 + x_2 - 0.5$$
$$y_2 = \text{sign}(v_2)$$

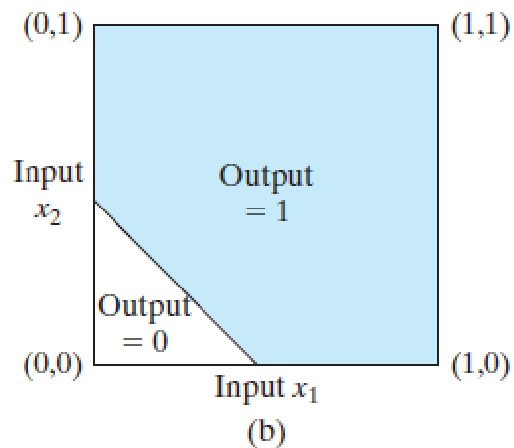
$$o = \text{sign}(-2y_1 + y_2 - 0.5)$$

$$(x_1, x_2) = (0, 0) \Rightarrow$$
$$v_1 = -1.5,\ \ y_1 = 0$$
$$v_2 = -0.5,\ \ y_2 = 0$$
$$o = \text{sign}(0 + 0 - 0.5) = 0$$

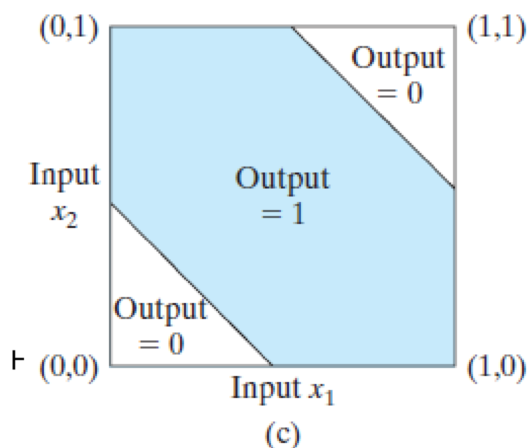
$$(x_1, x_2) = (1, 1) \Rightarrow$$
$$v_1 = 0.5,\ \ y_1 = 1$$
$$v_2 = 1.5,\ \ y_2 = 1$$
$$o = \text{sign}(-2 + 1 - 0.5) = 0$$



(a)



(b)



(c)

Bottom and top hidden neurons have positive and negative connections to output neuron, respectively.

$(x_1, x_2) = (0, 0)$:
When both hidden neurons are off, output neuron remains off.

$(x_1, x_2) = (1, 1)$:
When both hidden neurons are on, output neuron still remains off because top hidden neuron has larger negative weight.

$(x_1, x_2) = (0, 1)$ or $(1, 0)$:
When the top hidden neuron is off and bottom hidden neuron is on, output neuron is on because of the positive weight in bottom hidden neuron.

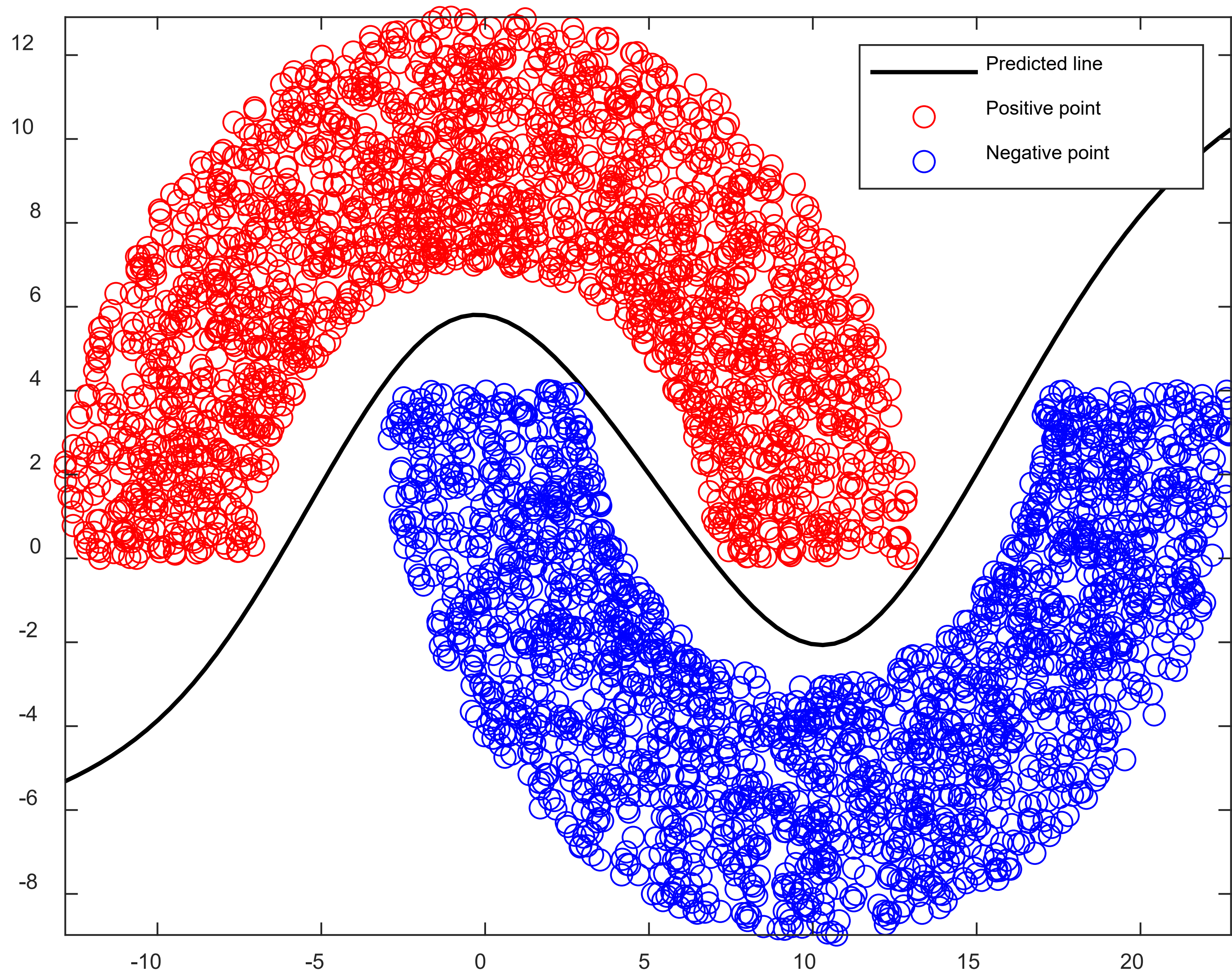
Can you use other weights to achieve the XOR operator?
Can you see the function of the bottom hidden neuron?
Can you see the function of the top hidden neuron?

We can return to the double-moon example with nonlinearly separable cases:

Size of input layer = 2
Size of hidden layer = 20
Size of output layer =1
Training data = 2000 per class
Hyperbolic tangent activation function: $y = \varphi(v) = \tanh(v)$
Learning rate $\eta = 0.01$
Mean removal and normalization are first applied on the input data
Decision boundary is obtained by `contour` command, corresponding to output = 0 ($o = -1$ or $o = 1$)

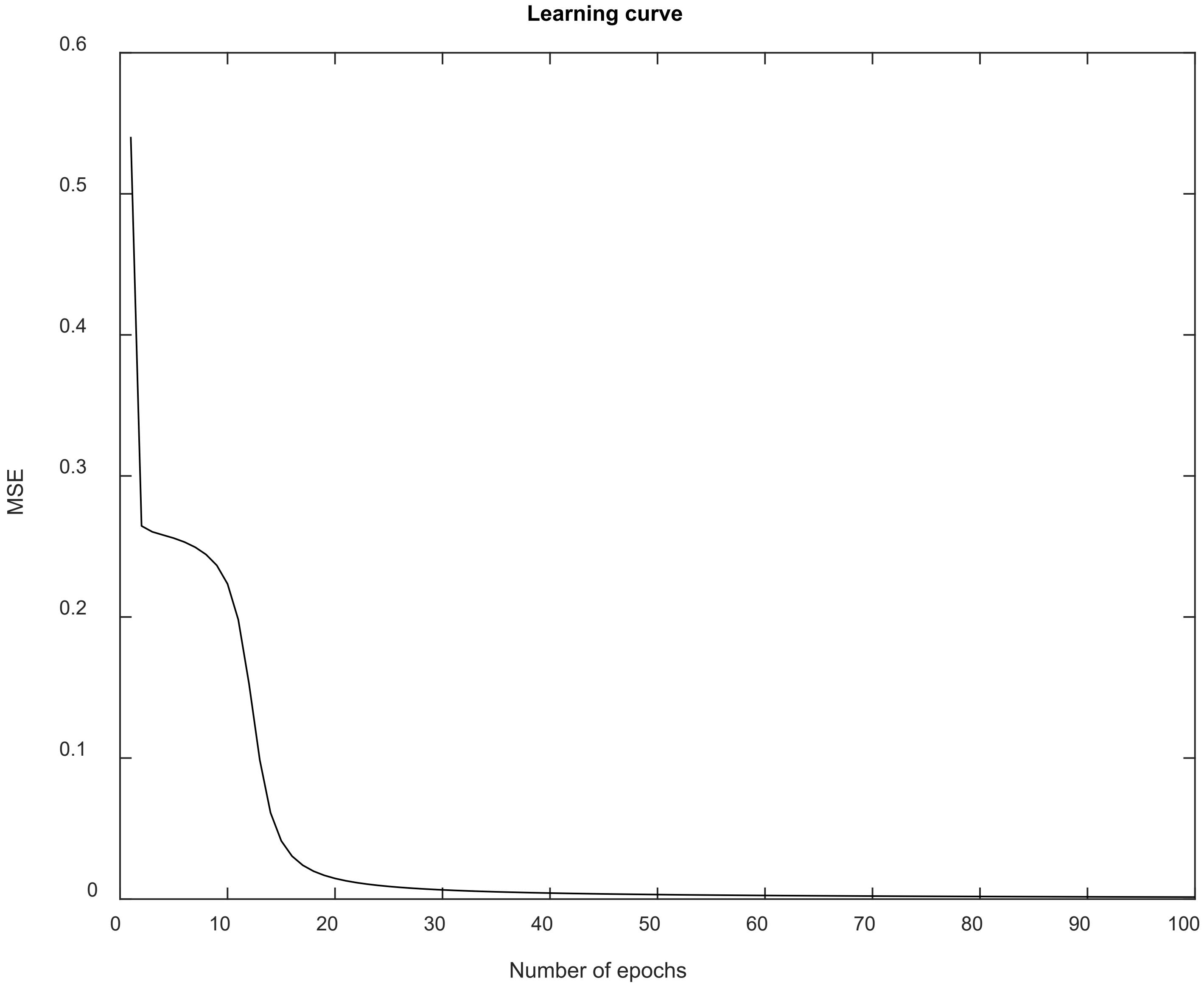
stdBPXOR.m | stdmomBPXOR.m | gdmxBPXOR.m | mlp.m

```
clear
close all

%% Generate dataset
r = 10;
w = 6;
d = -4;
N = 2000;
seed = randn(1);
display = 0;
[X,y,Xt] = generate_two_moons(r,w,d,N,seed,display);
shuffle_seq = randperm(2*N);
data = X(shuffle_seq,:);
labels = y(shuffle_seq,:);

%% Preprocess the input data: remove mean and normalize
mean_data = mean(data);
normal_mean_data = data - repmat(mean_data,size(mean_data,1),1);
max_data = max(abs(normal_mean_data));
input_data = normal_mean_data./max_data;

%% Neural network setting
num_input = 2;      % number of input nodes
num_hd = 20;        % number of hidden nodes
num_output = 1;     % number of output node
num_Epoch = 100;     % number of epochs
learningRate  = 0.01;
```

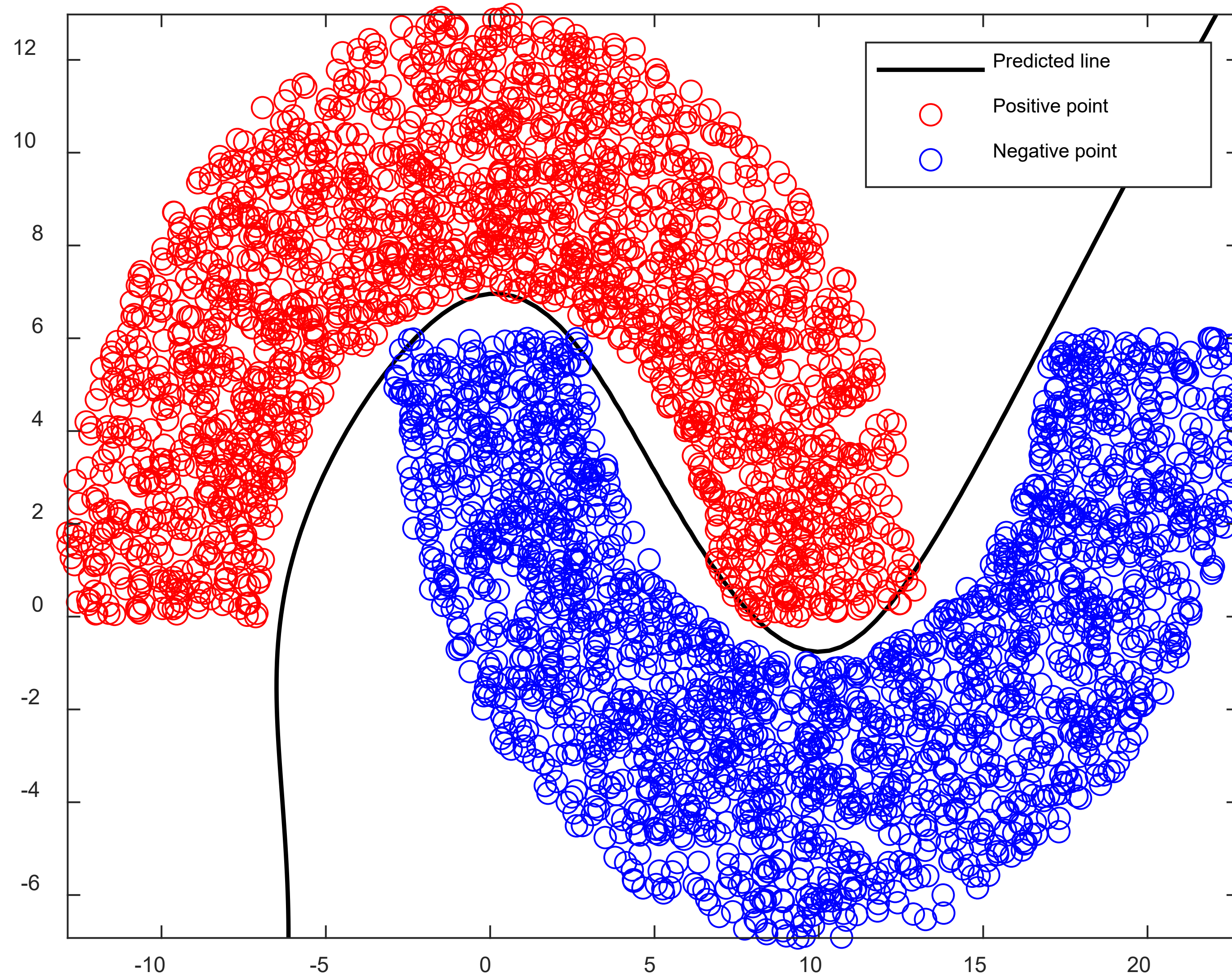


Predicted line
Positive point
Negative point
12
10
8
6
4
2
0
-2
-4
-6
-8
-10
-5
0
5
10
15
20

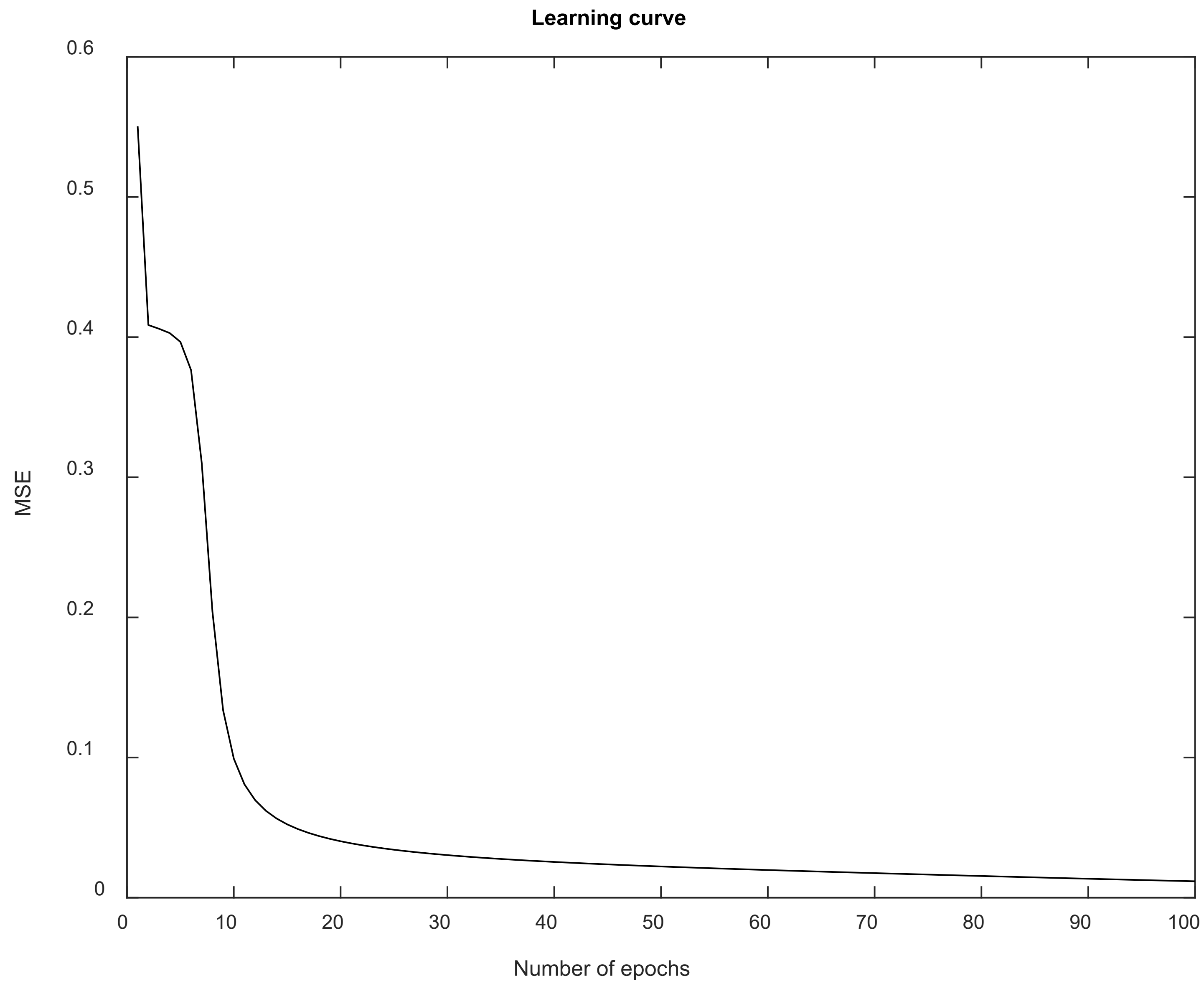
Perfect classification for training data:



$d = -6$



A few misclassification cases:



Testing and Validation

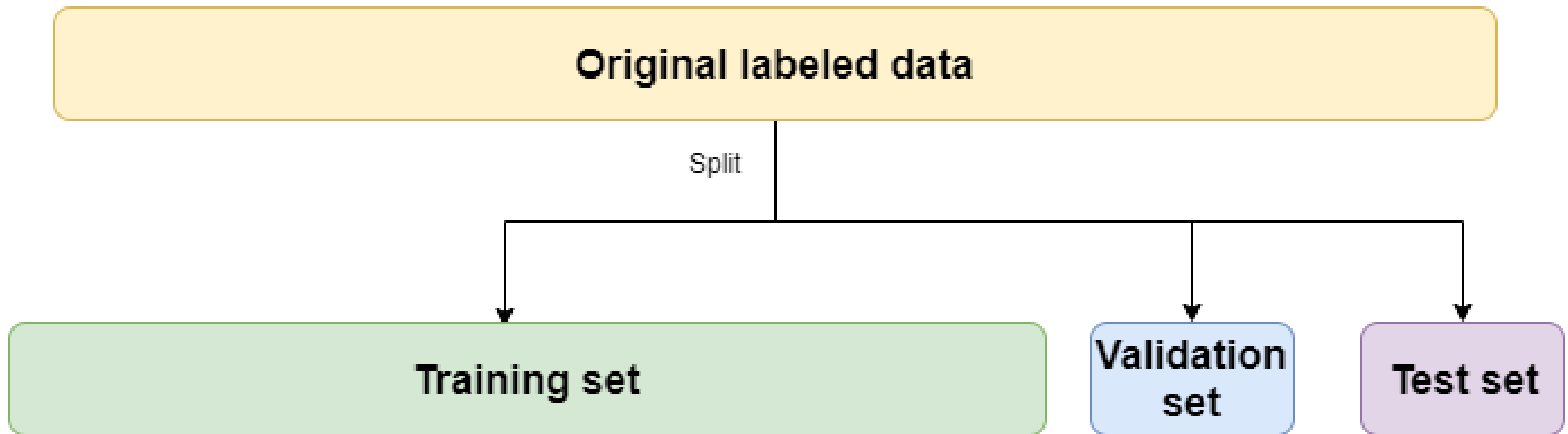
Unlike XOR or other logic functions, in many practical scenarios, the inputs are of any values like double-moon data.

In many supervised learning algorithms including the BP, we can achieve accurate classification results for training data.

However, an algorithm that will give perfect results using the known data is not useful. In fact, the algorithm is expected to perform well for unseen data.

To determine the appropriate weights and evaluate the performance of unseen data, we can divide the original data into training, validation and test sets.

Overfitting refers to the scenario when the algorithm performs very well in the training phase or the training MSE is very small while it performs poor for unseen data.



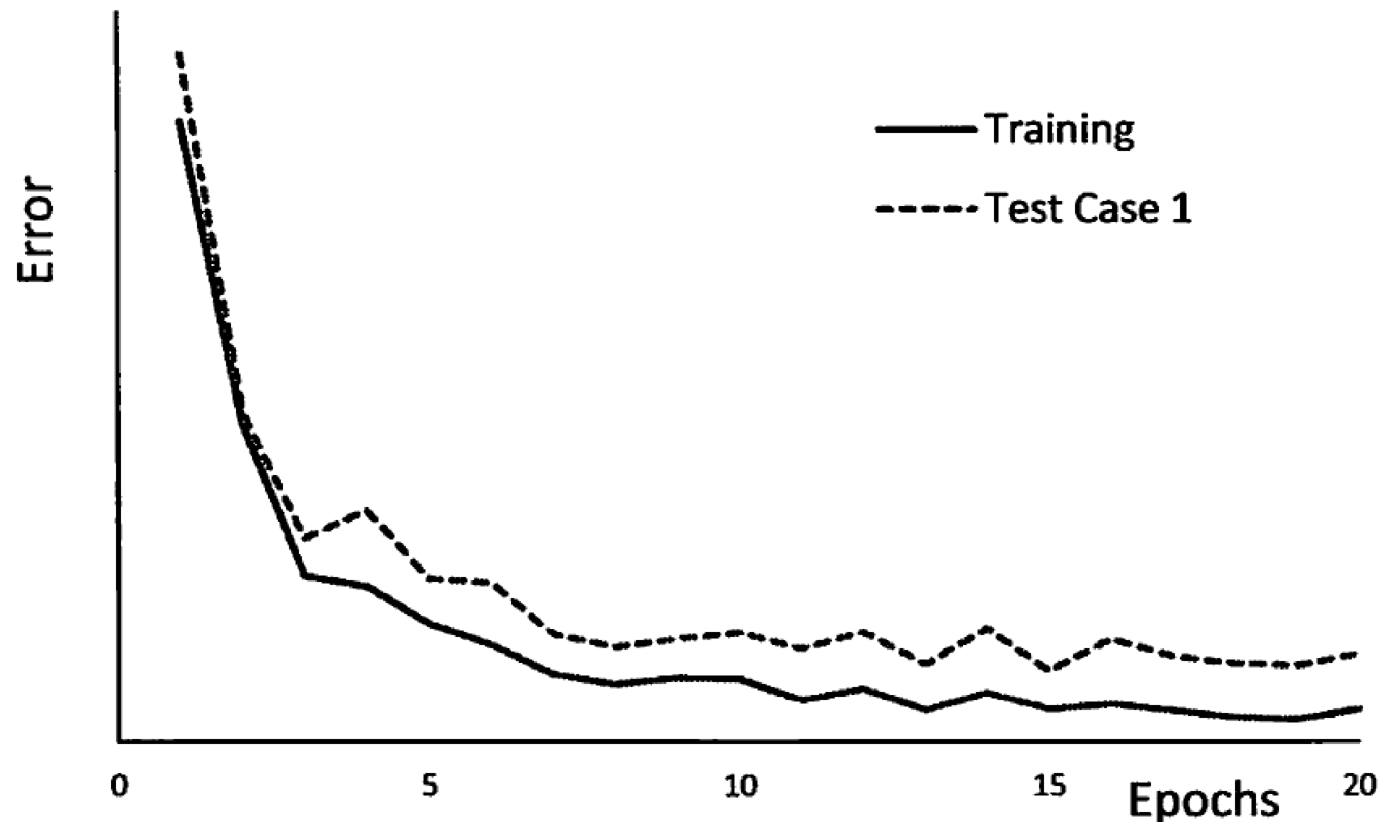
Source: https://www.brainstobytes.com/test-training-and-validation-sets/

Validation set aims to find the best model or weights from various trained models for unseen data

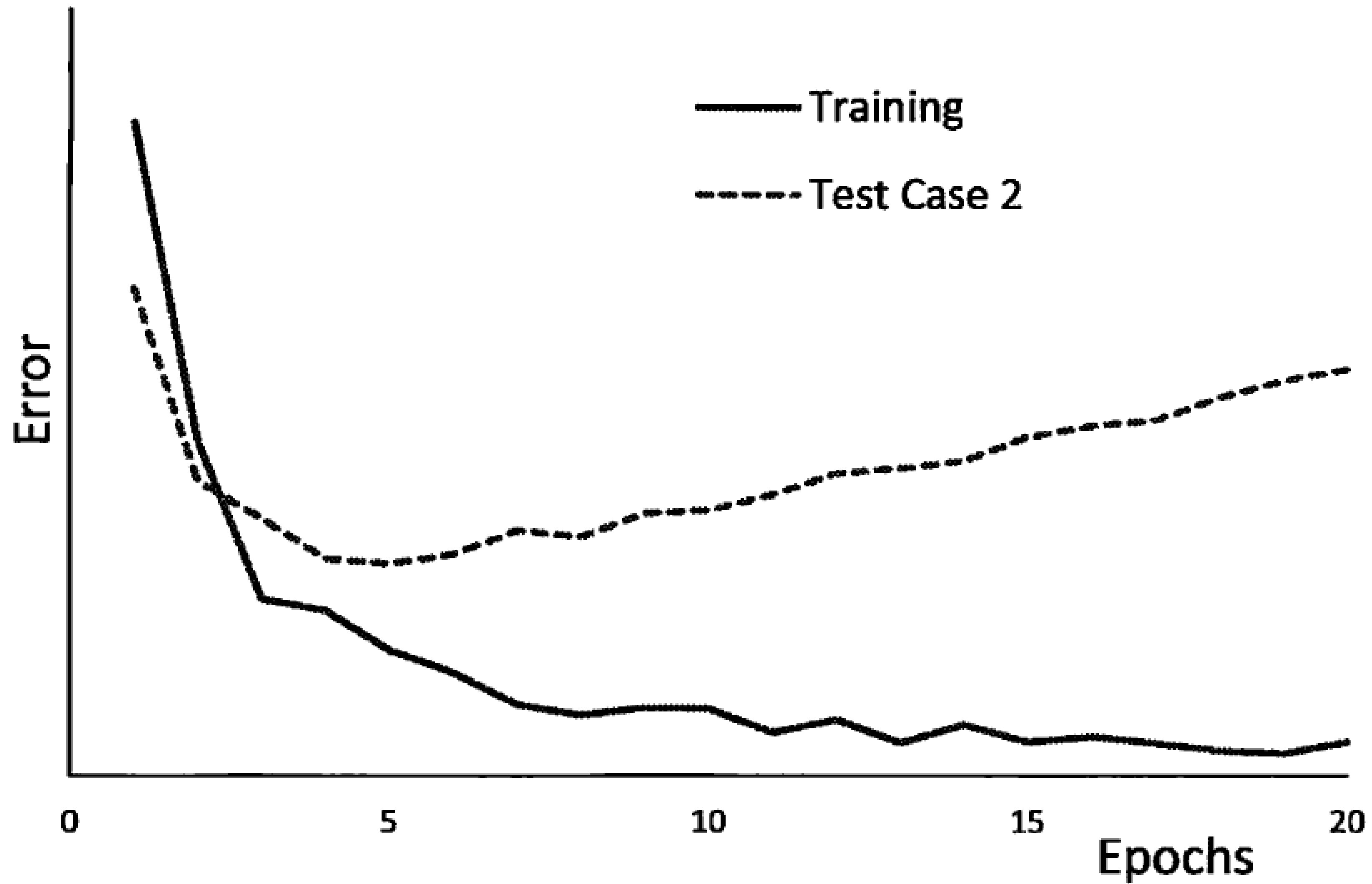
Based on the model determined in the validation stage, we use test set as unseen data to check for performance such as classification error.

Early-Stopping Method

Overfitting can happen if the algorithm has seen the training data too many times, i.e., there have been too many epochs.

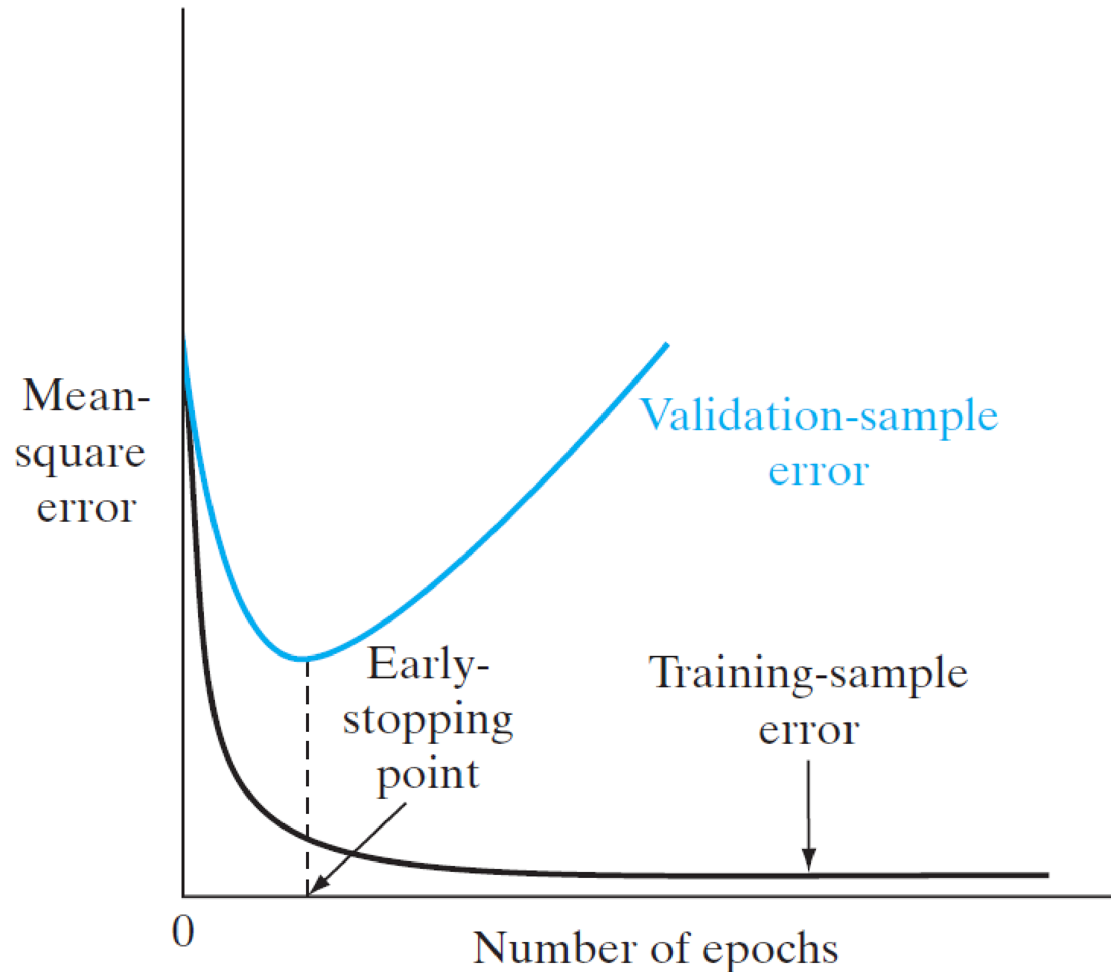


No overfitting as both training and test data have same trend but we can stop the training error becomes steady.



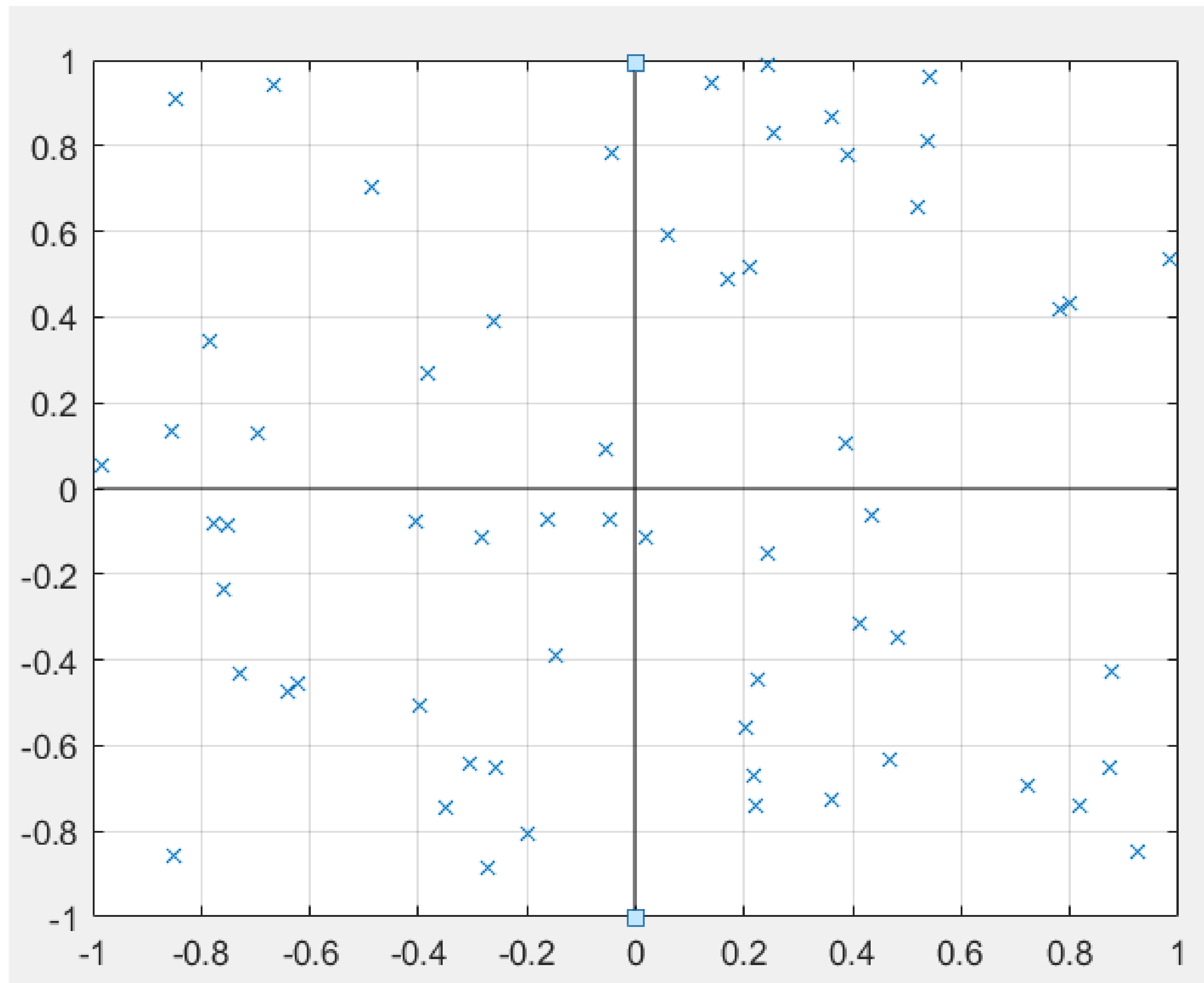
Overfitting occurs as the error in the test data increases with more epochs.

To avoid overfitting, early-stopping method can be used with a validation dataset.



We use weights at early-stopping point for the model from which classification accuracy is determined with test set.

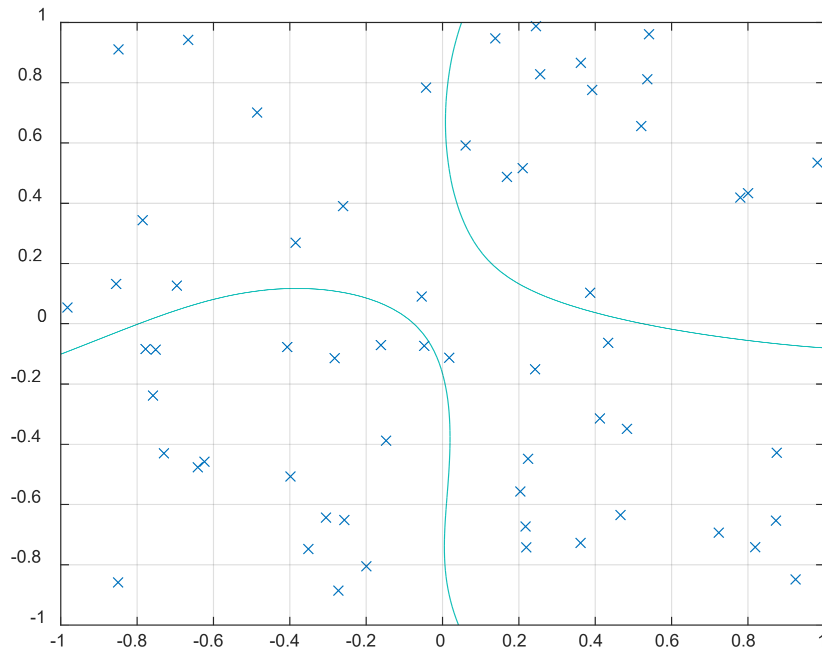
Consider a generalized XOR function with input $\boldsymbol{x} = [x_1 \ x_2]^T$ where $x_1, x_2 \in [-1, 1]$, and output $y = \text{sign}(x_1 x_2) \in \{1, -1\}$.



Without early-stopping:

BPgeneralXORwithoutearlystopping.m

```
in_dim=2;
out_dim=1;
n=120;
rand('seed',777);
x=2*(rand(2,n)-0.5);
trIdx=[1:60];
testIdx=[61:120];
figure(1)
t=sign(x(1,:).*x(2,:));
plot(x(1,trIdx),x(2,trIdx),'x');
grid on
no_h=20;


net = feedforwardnet(no_h,'traingdx');
net = configure(net,x,t);
rand('seed',1997);

IW = 1*(rand(no_h,in_dim)-0.5);
b1 = 1*(rand(no_h,1)-0.5);
LW = 1*(rand(out_dim,no_h)-0.5);
b2 = 1*(rand(out_dim,1)-0.5);

net.IW{1,1} = IW;
net.b{1,1} = b1;
net.LW{2,1} = LW;
net.b{2,1} = b2;

net.layers{1}.transferFcn = 'tansig';
net.layers{2}.transferFcn ='tansig';

net.trainParam.epochs = 200;
net.trainParam.show = 10;
net.trainParam.lr = 0.01;
net.trainParam.lr_inc=1.05;
net.trainParam.lr_dec=0.7;
net.trainParam.max_perf_inc=1.04;
net.trainParam.mc=0.9;
net.trainParam.goal = 0;
net.trainParam.min_grad=0;
net.trainParam.max_fail=1000;
net.trainParam.showWindow = 1;
%net.trainParam.showCommandLine = true ;
net.divideFcn='divideind';
net.divideParam.trainInd = trIdx;
net.divideParam.testInd = testIdx;
%net.divideParam.valInd = vaIdx;
%net.performParam.regularization=0;
[net,tr] = train(net,x,t);

x1 = linspace(-1,1);
 x2 = linspace(-1,1);
 [X1,X2] = meshgrid(x1,x2);
 XX1=X1(:)';
 XX2=X2(:)';
 XX=[XX1;XX2];
 ZZ=net(XX)';
  Z=reshape(ZZ,100,100);
  figure
  contour(X1,X2,Z,[0 0]);
  hold on
  plot(x(1,trIdx),x(2,trIdx),'x');
  grid on
```



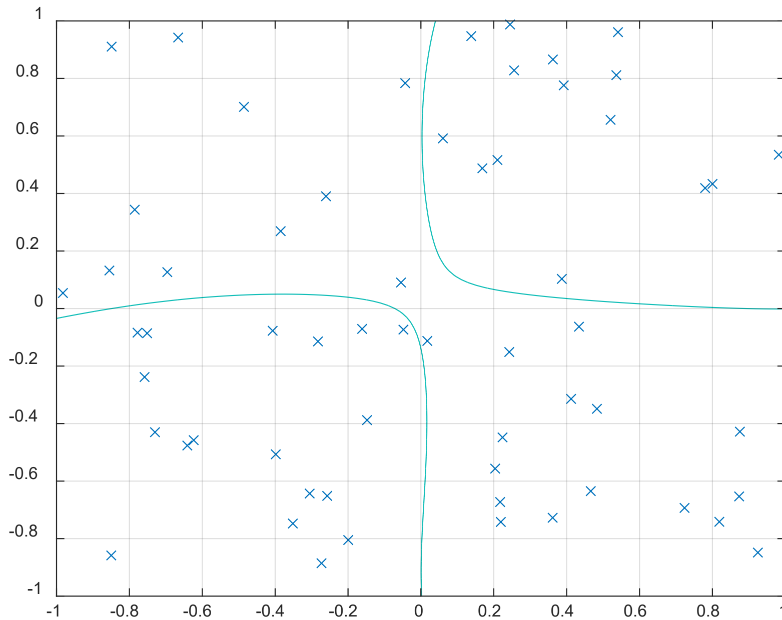
We use the weights at 200th epoch, corresponding to the best training performance, to produce the classification boundary.

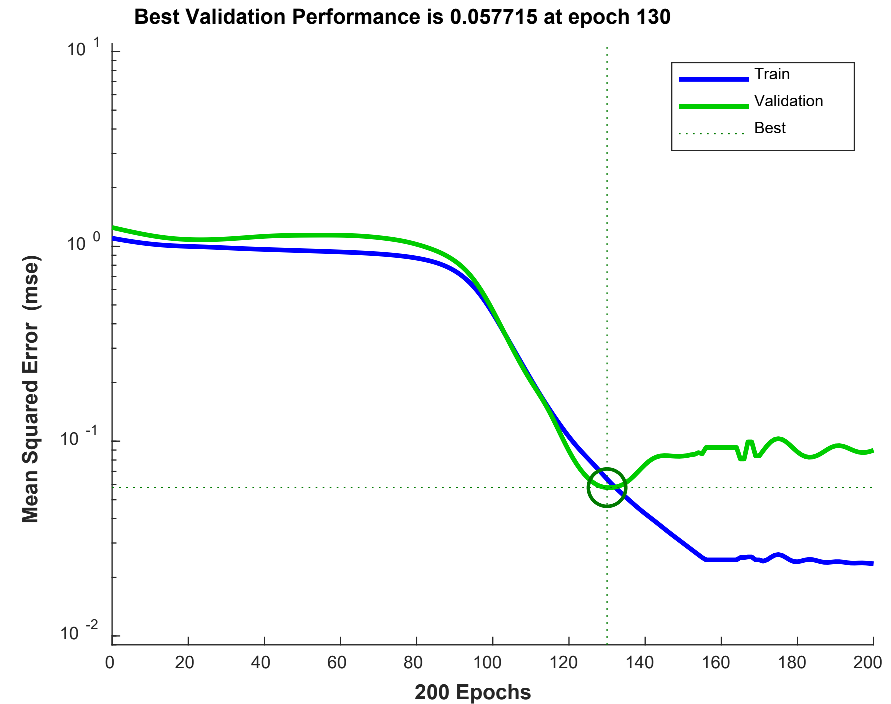
Note that in practice, we may not compute the error of validation set per epoch to save computation, e.g., compute every 5 epochs.

With early-stopping:

```
%net.divideParam.testInd = testIdx;
net.divideParam.valInd = valIdx;
```

```
43      %net.divideParam.testInd = testIdx;
44 –    net.divideParam.valInd = valIdx;
```





Now we use the weights at the epoch such that the validation set has the minimum error.

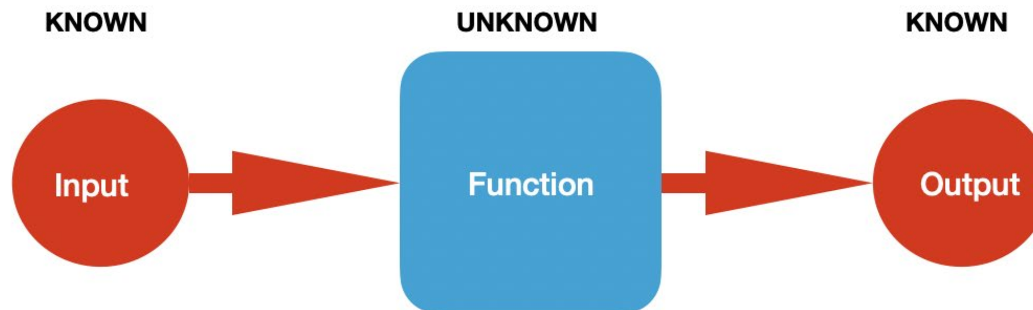
The produced decision boundary is closer to the ideal one.

Universal Approximation Theorem

Another application in supervised learning is function approximation.

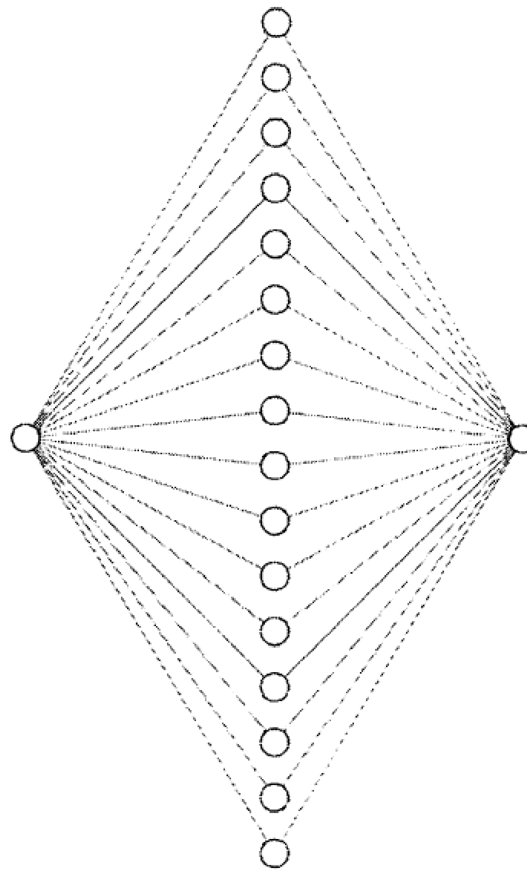
Assume the underlying relation is $y = f(x)$ where f is unknown, we want to estimate f based on a set of input and output, $\{x_i, y_i\}_{i=1}^N$.

Note that the estimated f, denoted by $\hat{f}$ does not have a closed-form expression like tangent, sine and sinc.



Source: Supervised, Unsupervised, And Semi-Supervised Learning With Real-Life Usecase (enjoyalgorithms.com)

Universal approximation theorem states that under quite weak conditions, any continuous function can be approximated with a single-layer NN to any degree of accuracy. That is, better approximation can be achieved with more hidden nodes.



Consider using a single-layer NN to approximate a sinc function:

$$y = f(x) = \mathrm{sinc}(x) = \frac{\sin(\pi x)}{\pi x}$$

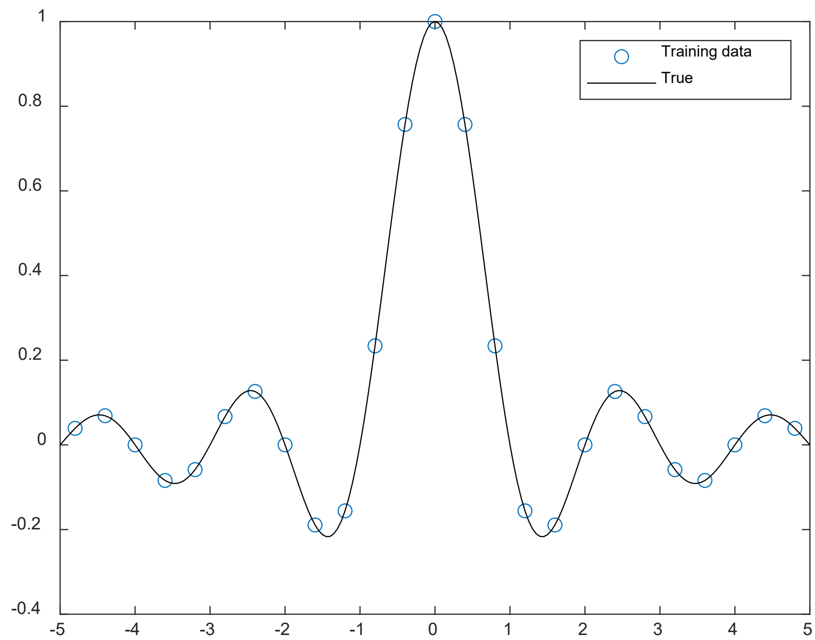
BPSincwithearlystopping.m

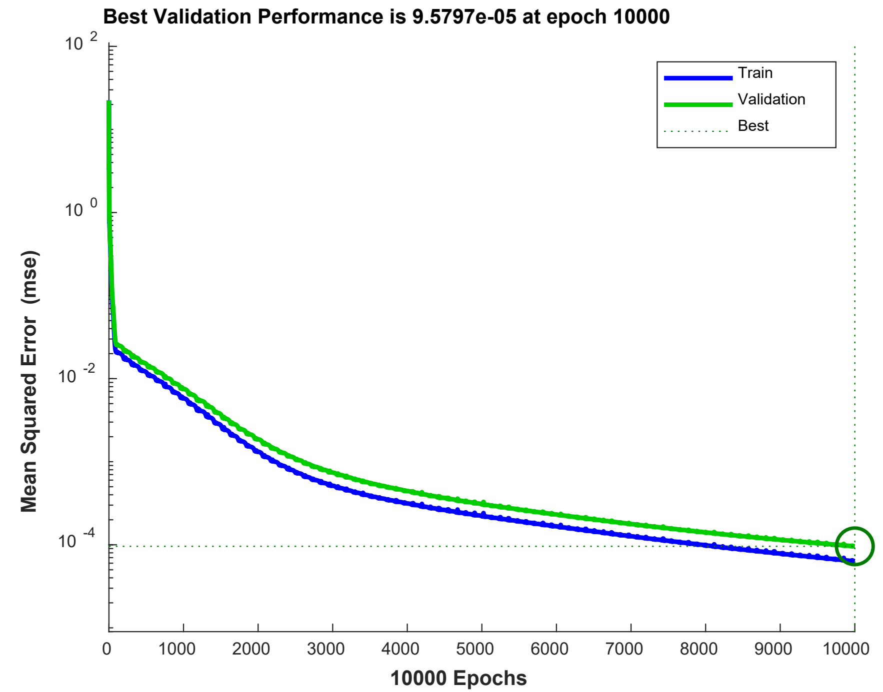
```
1  in_dim=1;
2  out_dim=1;
3  rand('seed',777);
4  randn('seed',689);
5  x_train=-4.8:.4:4.8
6  x_test=-5:.05:5;
7  n_test=length(x_test);
8  n_train=length(x_train);
9  x_test=sort(x_test);
10 y_train=sinc(x_train)+0.0*(rand(1,n_train)-0.5);
11 y_test=sinc(x_test);
12 
13 x=[x_train x_test];
14 t=[y_train y_test];
15 
16 figure(1)
17 hold off
18 plot(x_train,y_train,'o');
19 hold on
20 plot(x_test,y_test,'k-');
21 
22 trIdx=[1:n_train];
23 vaIdx=[n_train:n_train+n_test];
24 
25 no_h=100;
26 net = feedforwardnet(no_h,'traingdx');
27 net = configure(net,x,t);
```

Number of hidden nodes =100

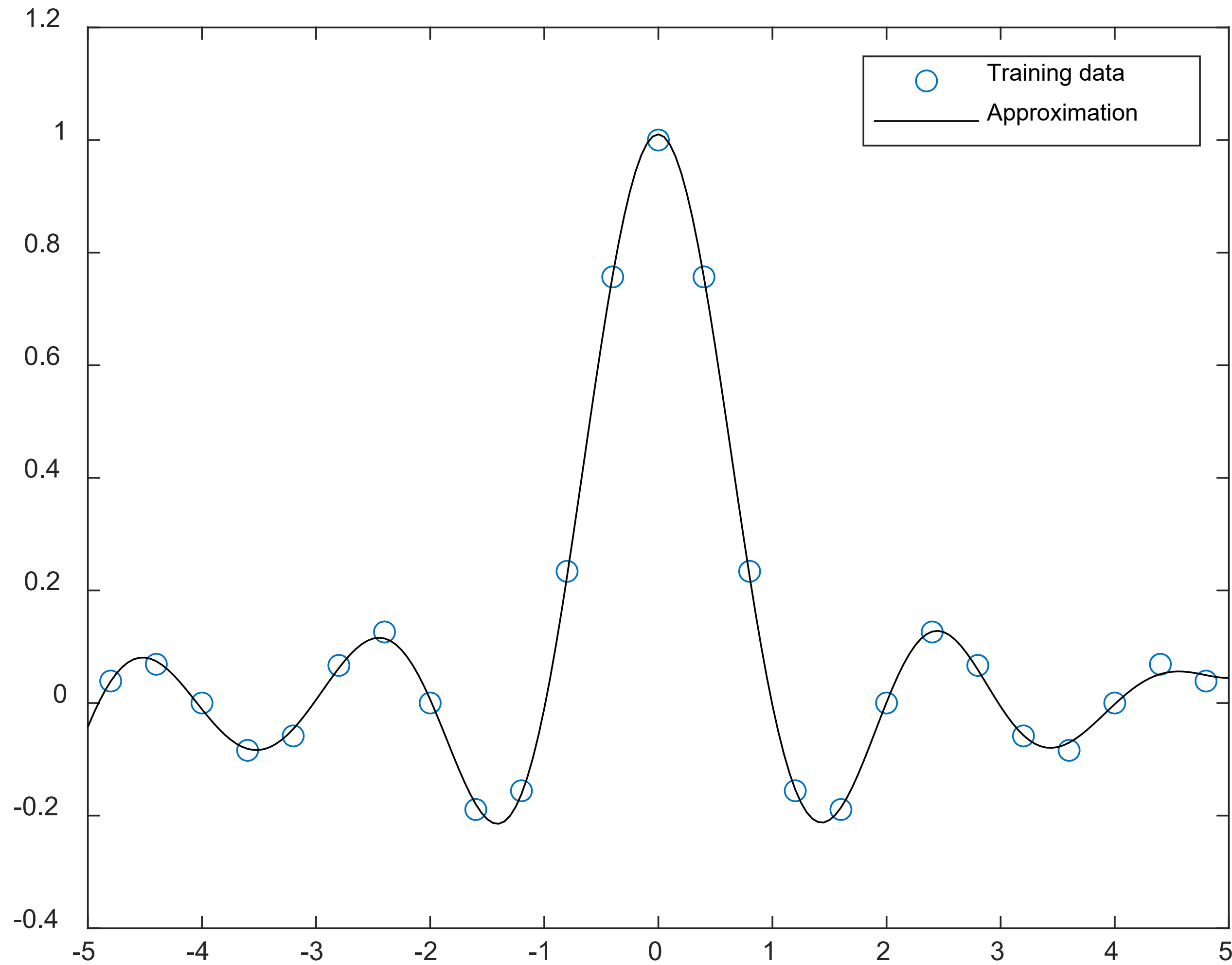
25 training input-output data with $x_i = -4.8,\ -4.4, \cdots, 4.8$ and noise-free y_i

Test input $x_i = -5,\ -4.95, \cdots, 5$



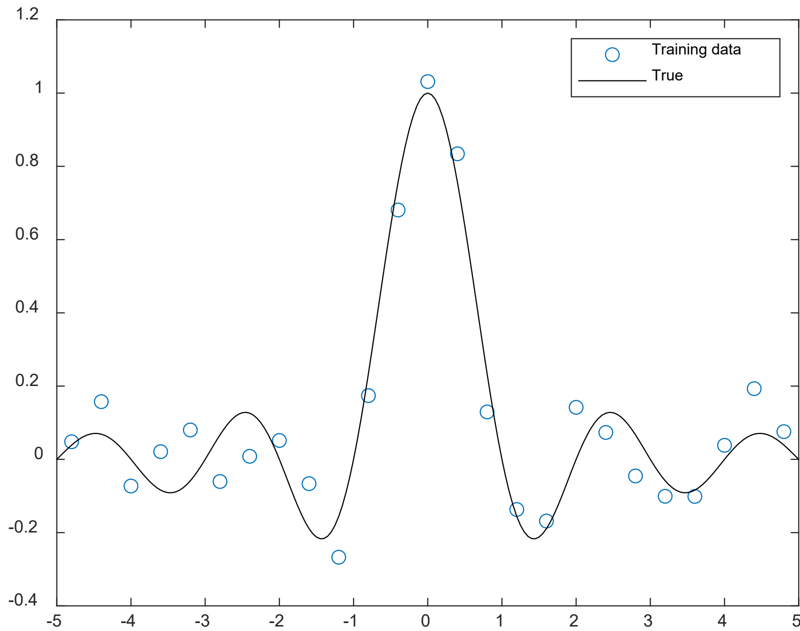


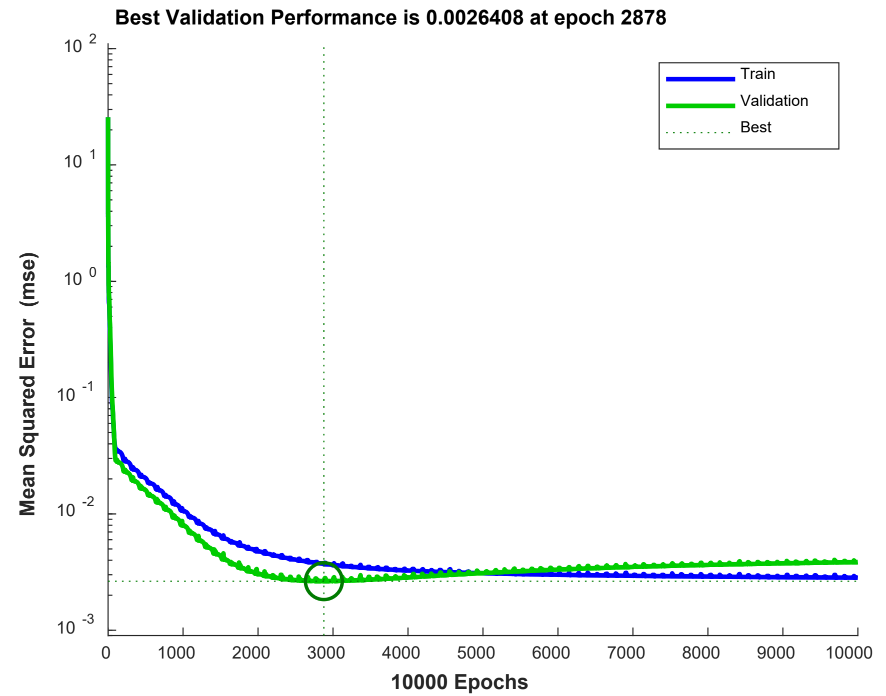
No overfitting is observed for noise-free case.



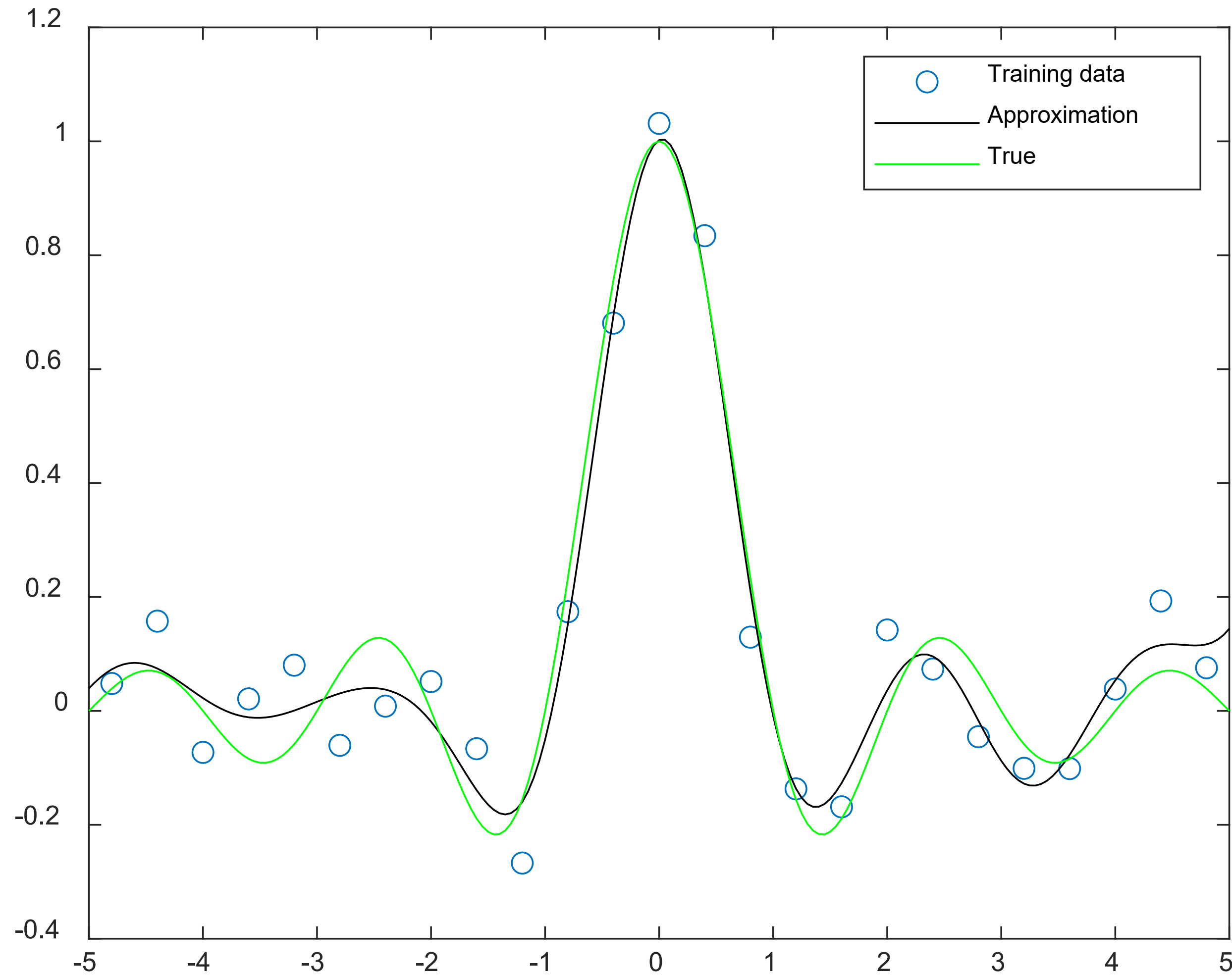
Accurate approximation is observed for $x_i = -5, \ -4.95, \cdots, 5$, except for the extrapolation points.

With noisy output:





Overfitting is observed for noisy case.



Even with weights from the early-stopping point, accurate function approximation cannot be achieved.

Bias and Variance Tradeoff

Consider function approximation in the presence of noise:

$$y = f(x) + e$$

where e is zero-mean noise with variance $\mathbb{E}\{e^2\} = \epsilon^2$, and is independent of $f(x)$.

Applying supervised learning using a set of training data $\{x_i, y_i\}_{i=1}^N$, an approximation function $\hat{f}(x)$ is obtained.

Consider x', which is not in the training set, the estimated function value based on the trained model is then $\hat{f}(x')$.

The error using the trained model is:

$$\delta = \hat{f}(x') - y' = \hat{f}(x') - f(x') - e \quad (20)$$

As different training datasets produce different $\hat{f}(x)$, $\hat{f}(x)$ can be viewed as a random variable. The MSE of δ is then:

$$
\begin{aligned}
\mathbb{E}\{\delta^2\} &= \mathbb{E}\left\{\left(\hat{f}(x') - f(x') - e\right)^2\right\} \\
&= \mathbb{E}\left\{\left(\left[\hat{f}(x') - \mathbb{E}\{\hat{f}(x')\}\right] + \left[\mathbb{E}\{\hat{f}(x')\} - f(x')\right] - e\right)^2\right\} \\
&= \mathrm{var}(\hat{f}(x')) + \left[\mathbb{E}\{\hat{f}(x')\} - f(x')\right]^2 + \epsilon^2 \qquad (21)
\end{aligned}
$$

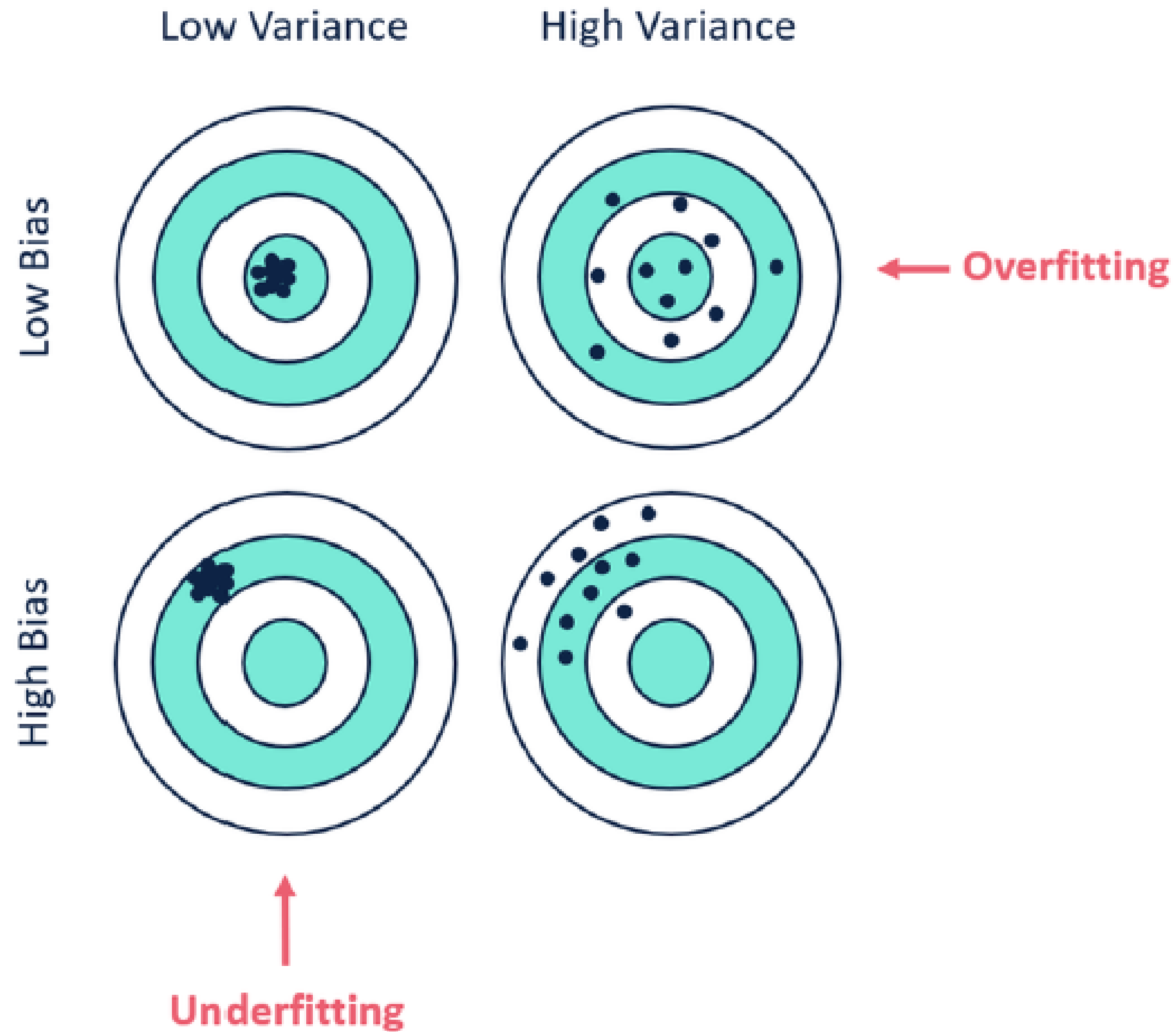
where

$$
\mathrm{var}(\hat{f}(x')) = \mathbb{E}\left\{\left[\hat{f}(x') - \mathbb{E}\{\hat{f}(x')\}\right]^2\right\}
$$

The first, second, and third terms of (21) are variance, bias, and noise floor, respectively.

While we have no control on ϵ^2, a balance between bias and variance can be attained.

Ideally, we want low bias and low variance.

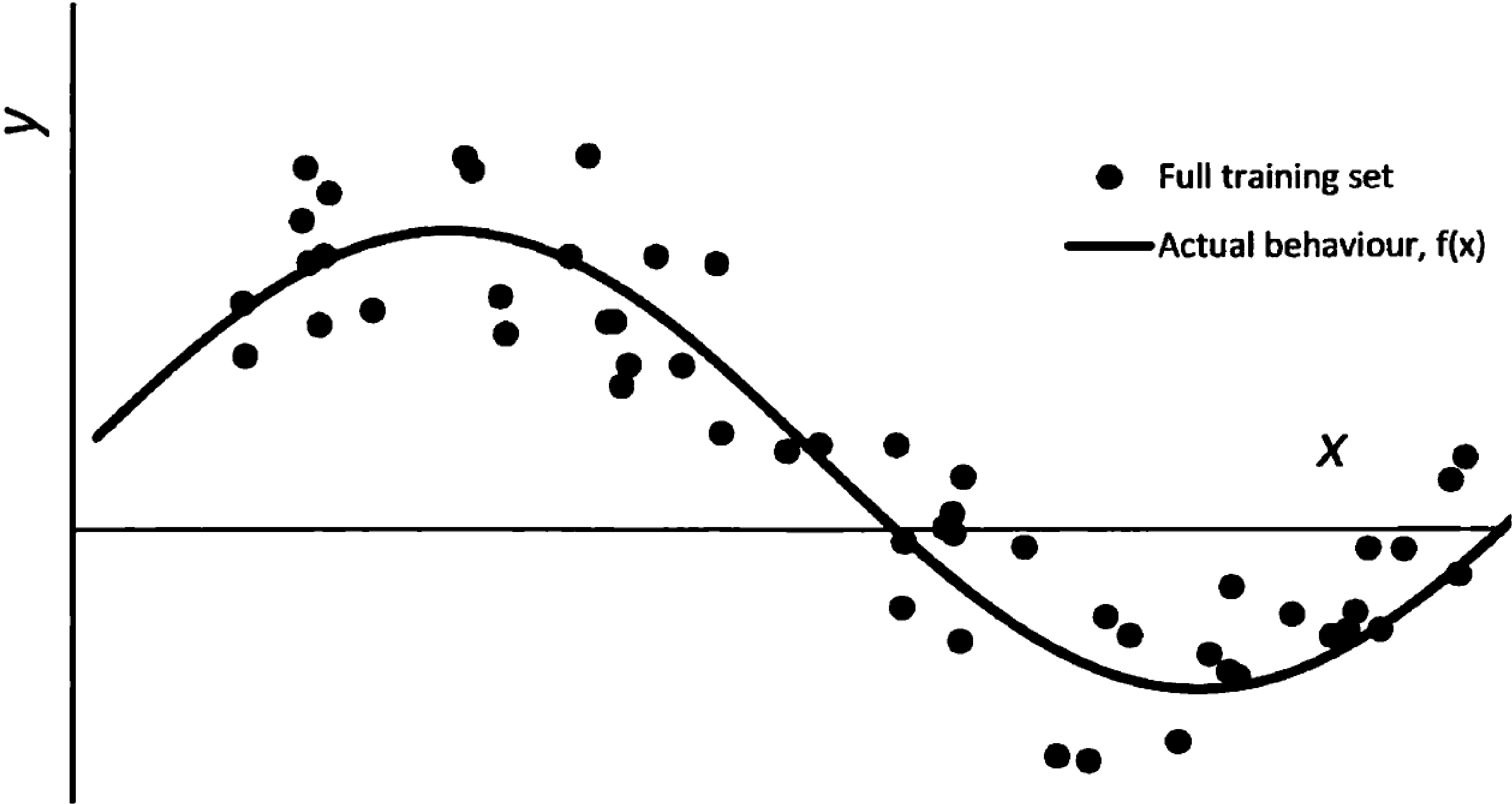


Overfitting occurs when we train our algorithm too well on the training data, possibly with too many parameters for fitting, leading to high variance.

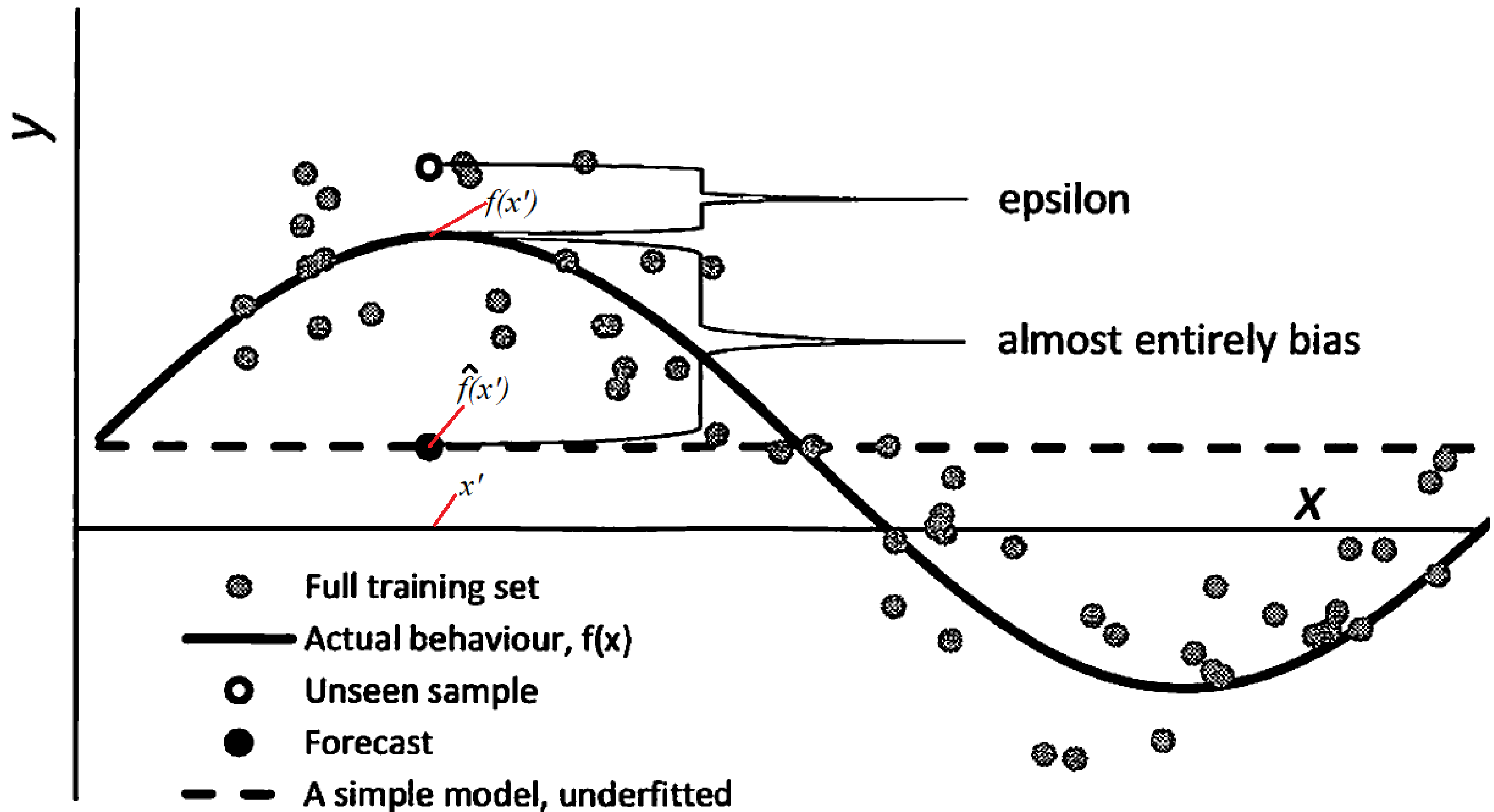
An analogy: a 2-year old boy playing with his first jigsaw puzzle. After many attempts, he finally succeeds. Thereafter he finds it much easier to solve the puzzle. But has he really learned how to do jigsaw puzzles or has he just memorized the one picture?

Underfitting occurs when the algorithm is too simple, possibly with too few parameters for fitting, or not sufficiently trained, leading to high bias.

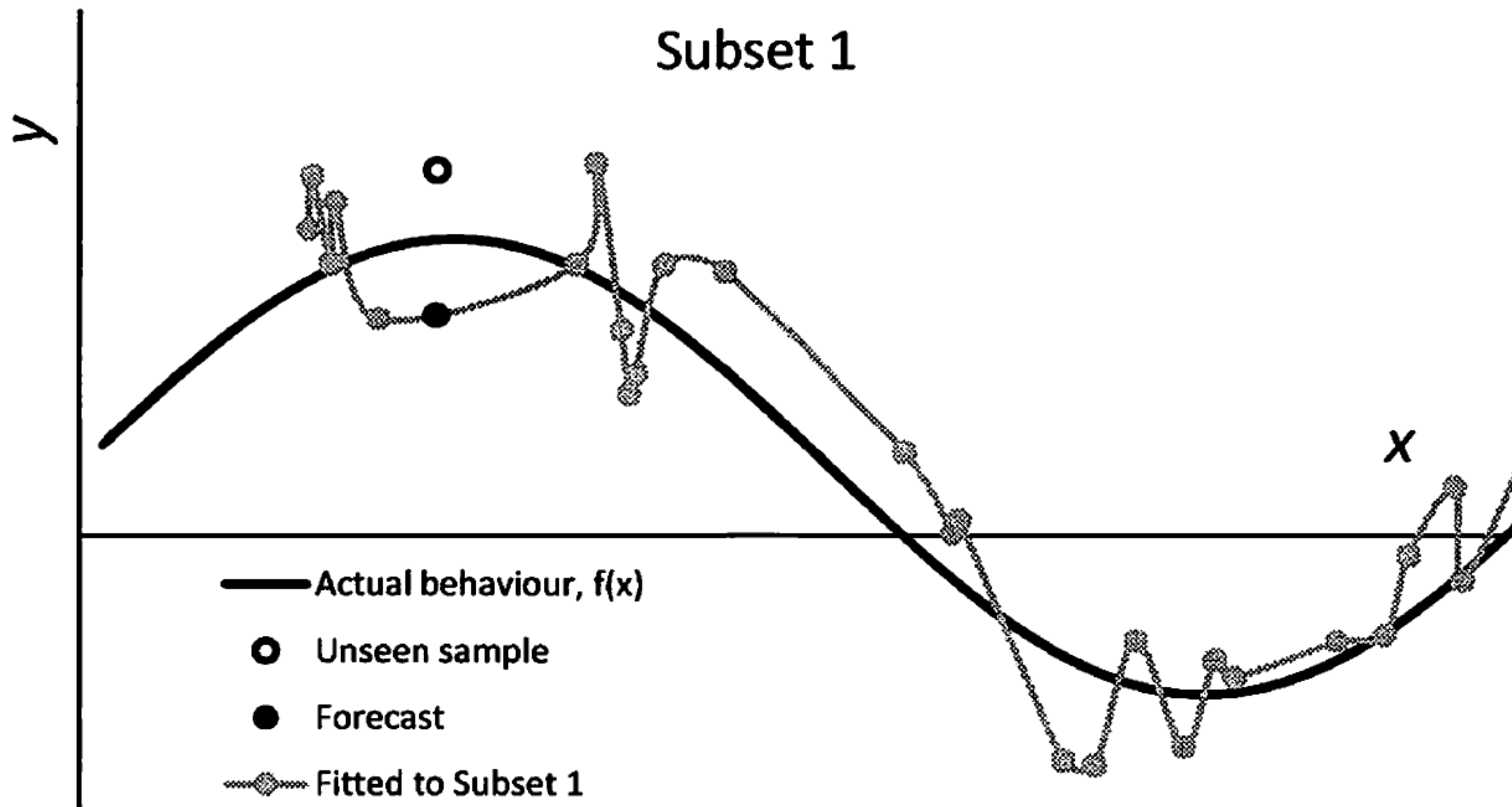
Consider a function approximation example again:



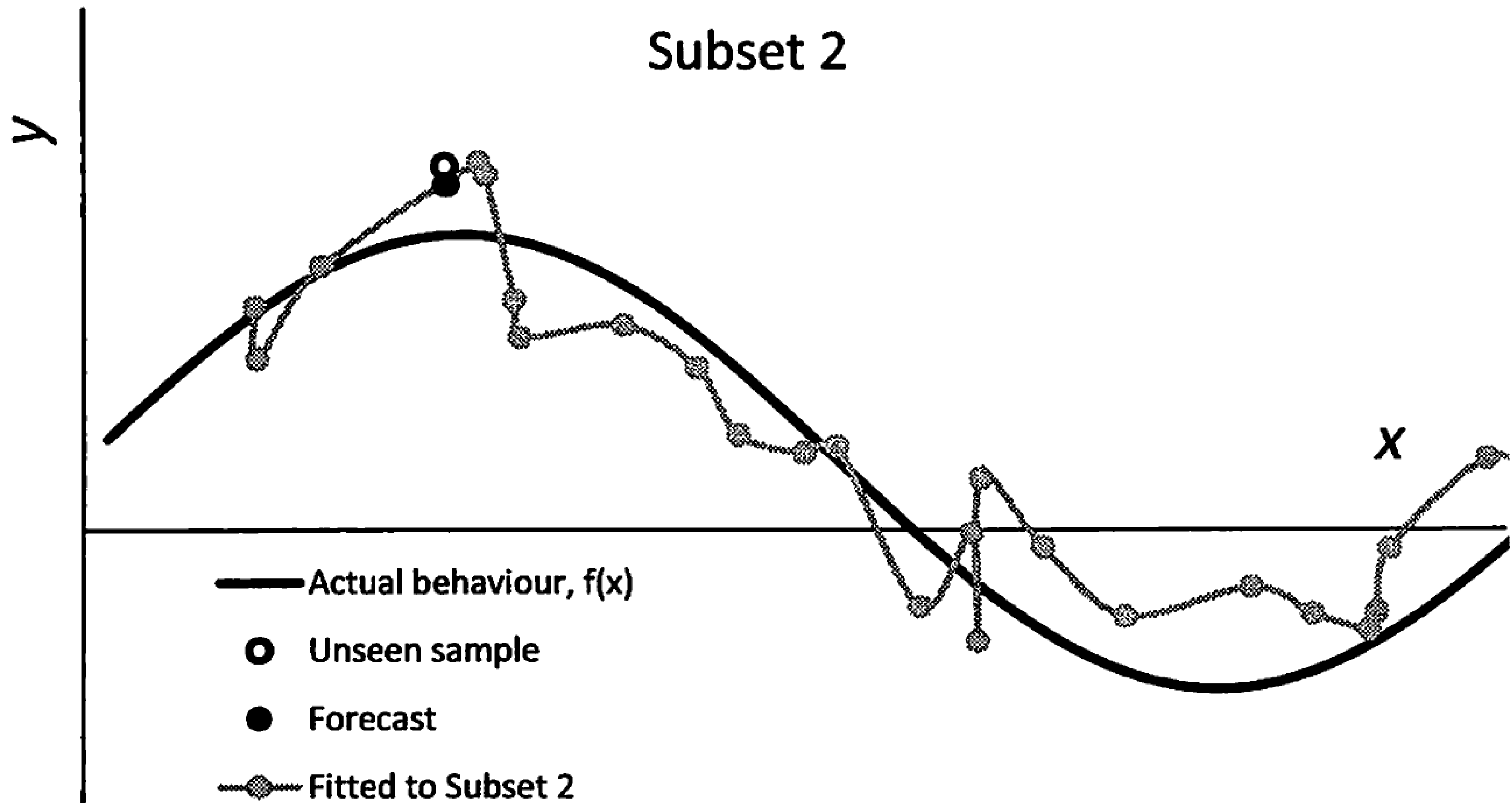
Large bias for a simple model which is a straight line:



Perfect fit with a training subset:

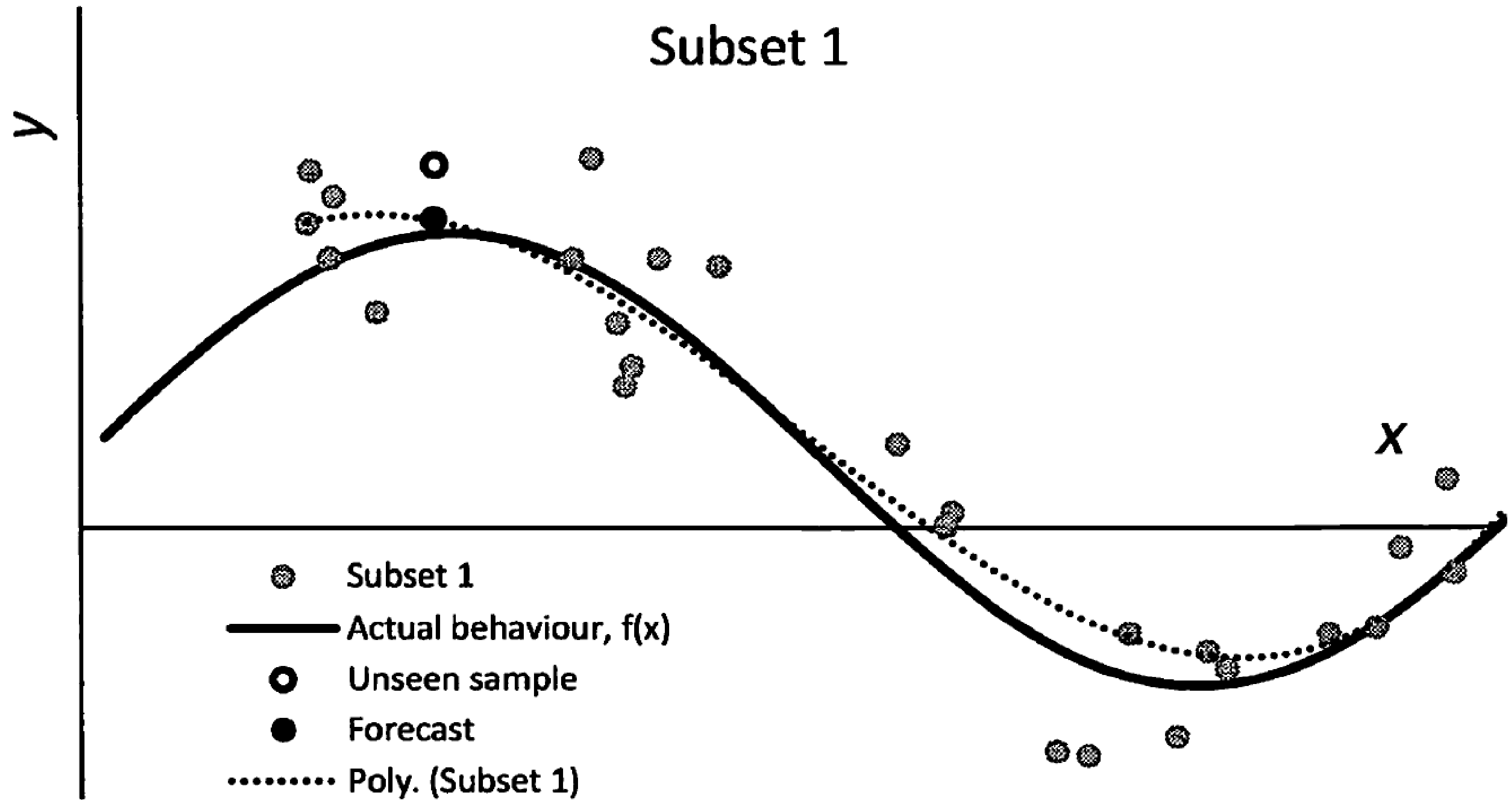


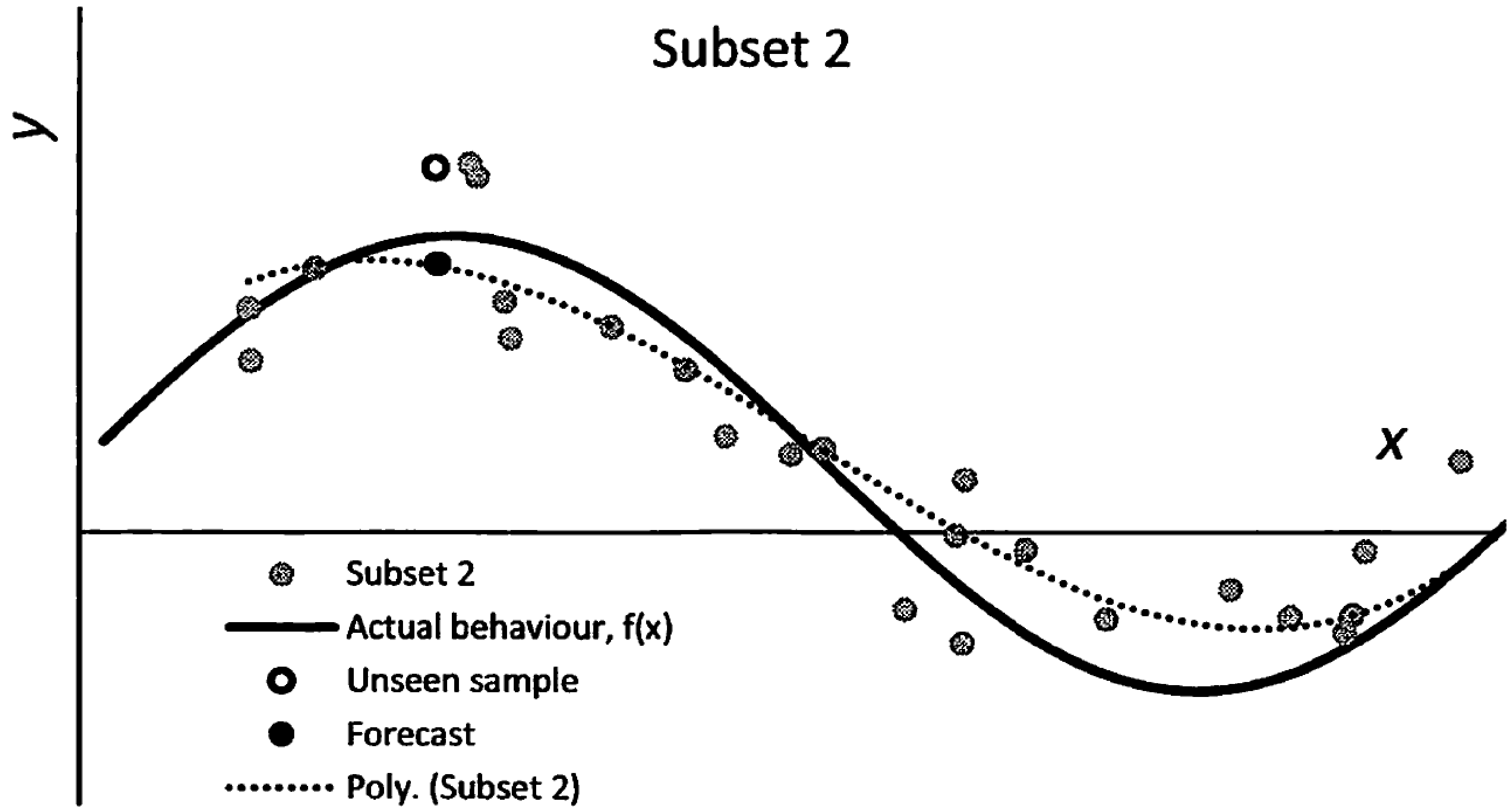
Perfect fit with another training subset:



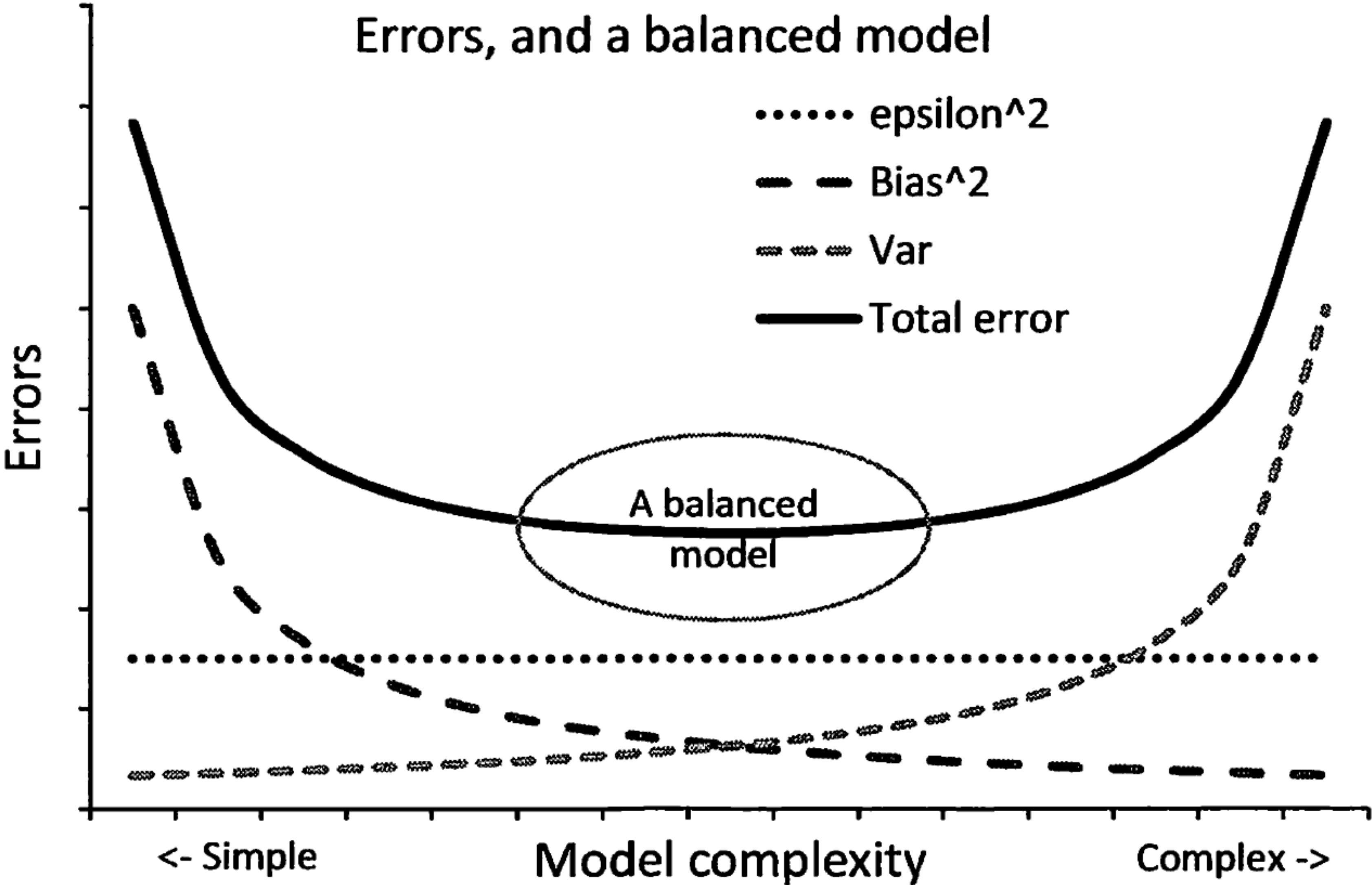
We see large variations between 2 realizations of $\hat{f}(x')$, indicating large variance.

A good fit can be achieved for a model which is not too simple or too complex, e.g., a cubic polynomial:




Subset 2
y
x
Subset 2
Actual behaviour, f(x)
Unseen sample
Forecast
Poly. (Subset 2)

A balanced model is:



Weight-Decay Regularization

To avoid overfitting due to too many parameters, regularization can be applied.

The overall cost function of (9) to be minimized becomes:

$$J(\boldsymbol{w}) = \frac{1}{2}\sum_{n=1}^{N}\sum_{j}(d_j(n) - y_j(n))^2 + \frac{\lambda}{2}\|\boldsymbol{w}\|_2^2, \quad \|\boldsymbol{w}\|_2^2 = \sum_{j}\sum_{i} w_{ji}^2 \quad (22)$$

where $\lambda > 0$ is the regularization parameter containing the prior information of $\boldsymbol{w}$.

Here, the idea of regularization is to reduce magnitudes of $\{w_{ji}\}$ so that their impacts on the model is reduced.

The BP algorithm can be easily modified because:

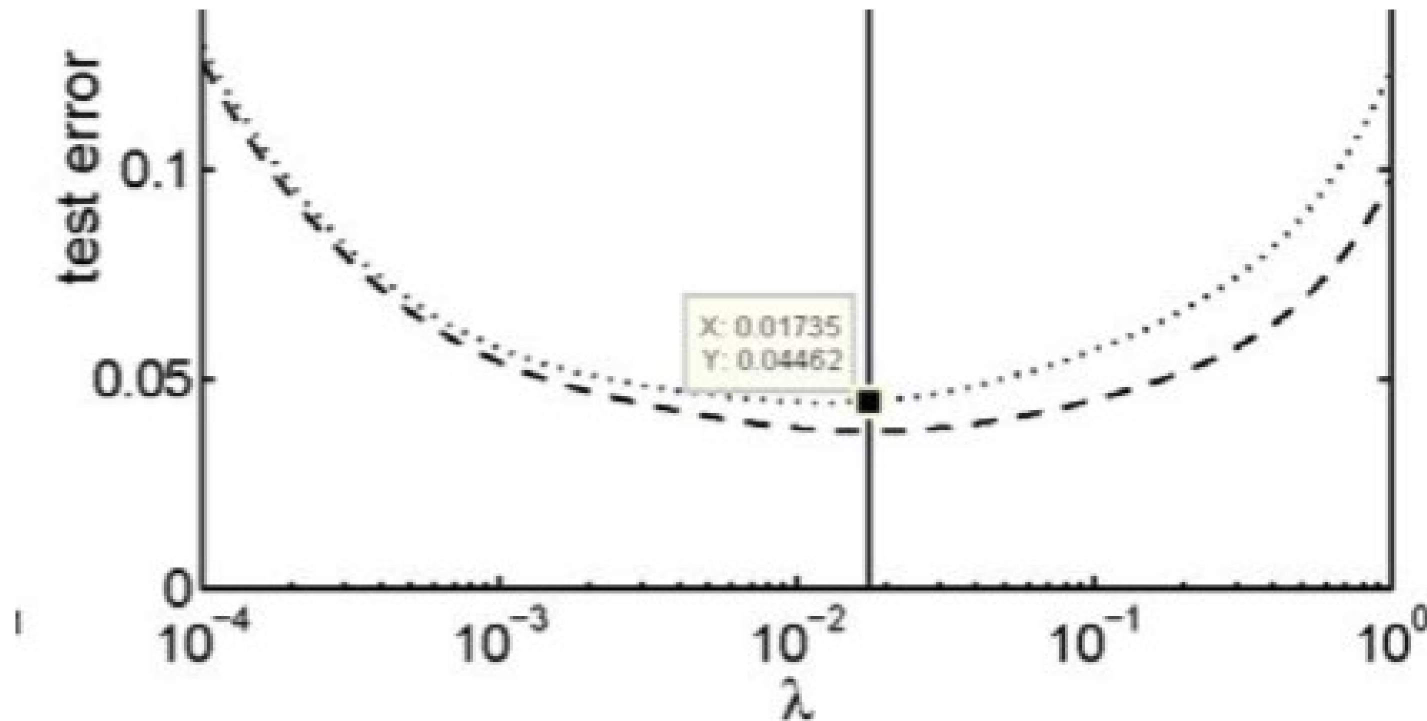
$$\frac{\partial \|\boldsymbol{w}\|_2^2}{\partial w_{ji}} = 2w_{ji}$$

Hence $w_{ji}(n)$ in (12) is reduced, referred to as weight decay:

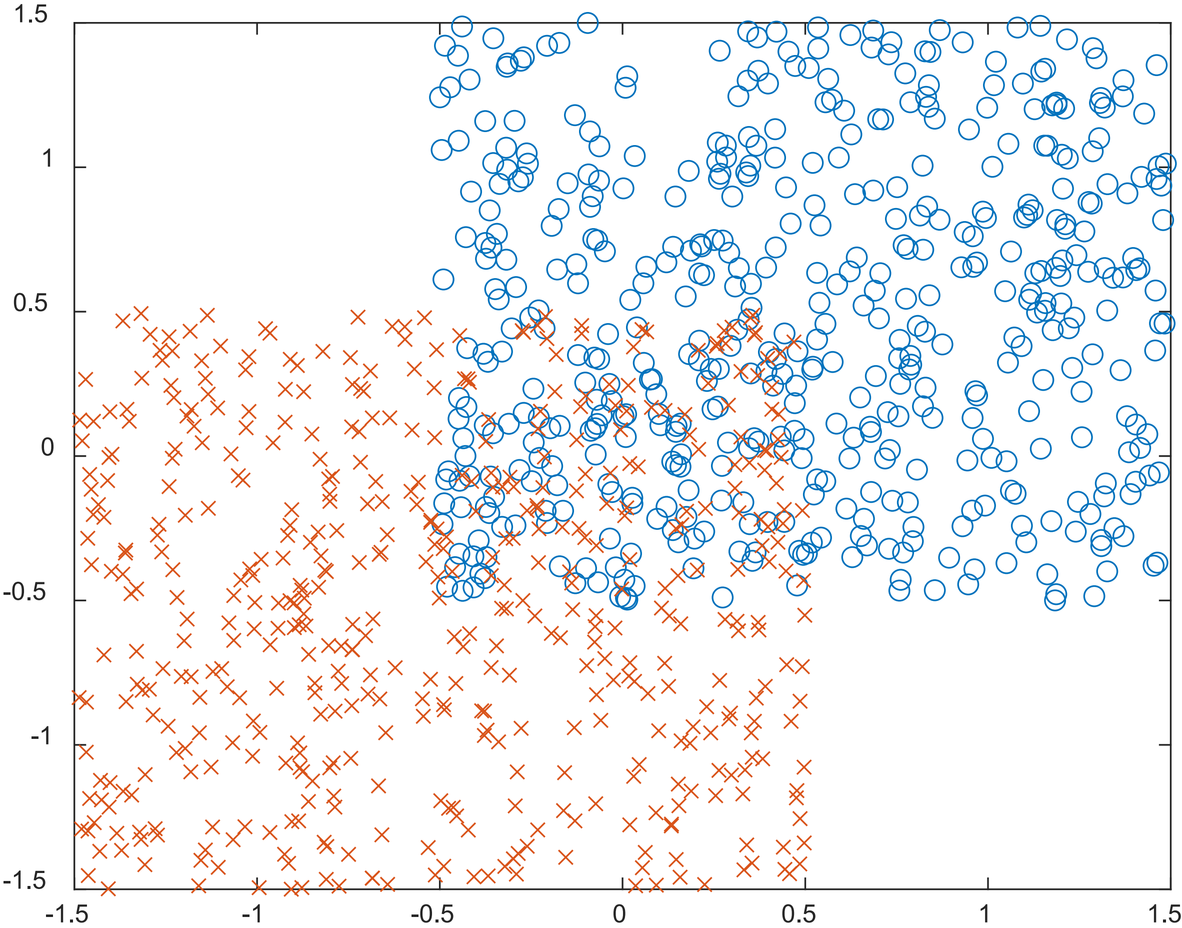
$$w_{ji}(n+1) = w_{ji}(n) + \Delta w_{ji}(n), \quad \Delta w_{ji}(n) = -\eta \frac{\partial J_n(\boldsymbol{w})}{\partial w_{ji}(n)}$$

To find λ, we search on a logarithmic scale instead of linear scale and use a test set on a trained model with different λ.

```
>> lam=[-4:.25:1]';
>> lam=10.^lam

lam =

   0.000100000000000
   0.000177827941004
   0.000316227766017
   0.000562341325190
   0.001000000000000
   0.001778279410039
   0.003162277660168
   0.005623413251903
   0.010000000000000
   0.017782794100389
   0.031622776601684
   0.056234132519035
   0.100000000000000
   0.177827941003892
   0.316227766016838
   0.562341325190349
   1.000000000000000
   1.778279410038923
   3.162277660168380
   5.623413251903491
  10.000000000000000
```



Consider classification between 2 classes of data:



Without regularization:

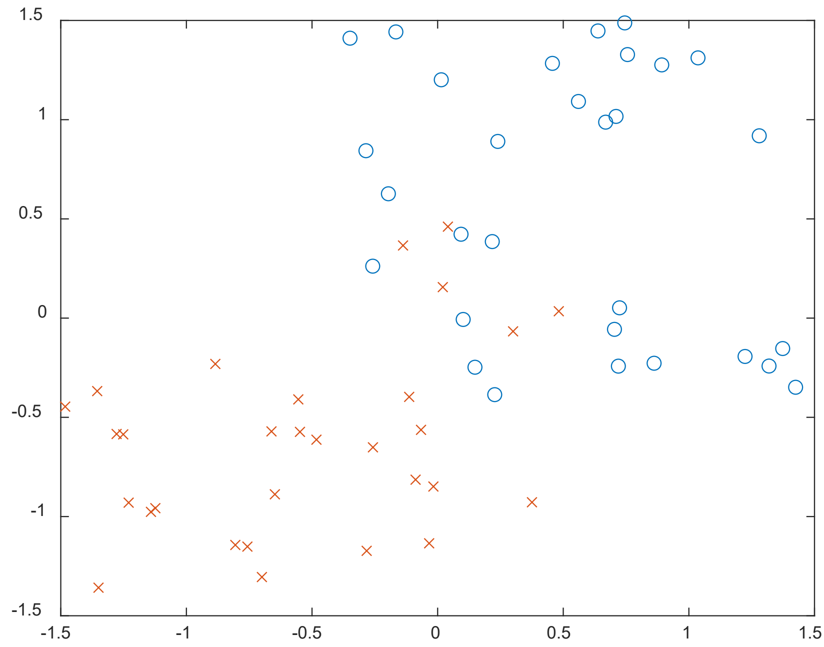
BPgenera_weight_decay.m

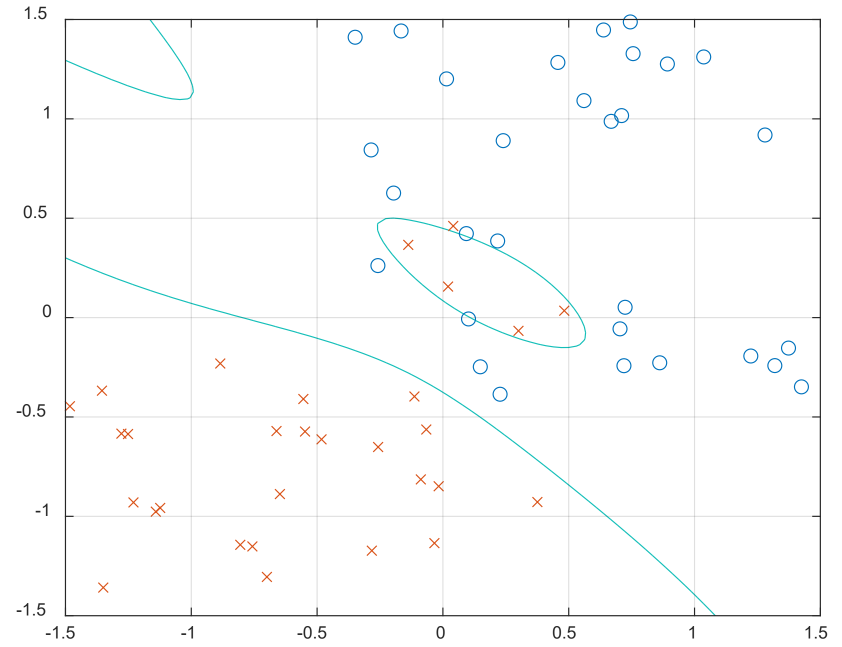
```
1 -     in_dim=2;
2 -     out_dim=1;
3 -     n=60;
4 -     rand('seed',777);
5 -     kx1=2*(rand(2,n/2)-0.5)+[0.5 0.5]';
6 -     t1=ones(1,n/2);
7 -     kx2=2*(rand(2,n/2)-0.5)-[0.5 0.5]';
8 -     t2=-ones(1,n/2);
9 -     x=[kx1 kx2];
10 -    t=[t1 t2];
11 -    figure(1)
12 -    plot(kx1(1,:),kx1(2,:),'o');
13 -    hold on
14 -    plot(kx2(1,:),kx2(2,:),'x');
15 -    trIdx=[1:n];
16
17 -    n_test=1000;
18 -    rand('seed',689);
19 -    kx1_test=2*(rand(2,n_test/2)-0.5)+[0.5 0.5]';
20 -    t1_test=ones(1,n_test/2);
21 -    kx2_test=2*(rand(2,n_test/2)-0.5)-[0.5 0.5]';
22 -    t2_test=-ones(1,n_test/2);
23 -    x_test=[kx1_test kx2_test];
24 -    t_test=[t1_test t2_test];
25 -    figure(2)
26 -    plot(kx1_test(1,:),kx1_test(2,:),'o');
27 -    hold on
28 -    plot(kx2_test(1,:),kx2_test(2,:),'x');
```

BPgenera_weight_decay.m

```
32 -    no_h=30;
33 -    net = feedforwardnet(no_h,'traingdx');
34 -    net = configure(net,x,t);
35 -    rand('seed',1997);
36 -    IW = 1*(rand(no_h,in_dim)-0.5);
37 -    b1 = 1*(rand(no_h,1)-0.5);
38 -    LW = 1*(rand(out_dim,no_h)-0.5);
39 -    b2 = 1*(rand(out_dim,1)-0.5);
40 -    net.IW{1,1} = IW;
41 -    net.b{1,1} = b1;
42 -    net.LW{2,1} = LW;
43 -    net.b{2,1} = b2;
44
45 -    net.layers{1}.transferFcn = 'tansig';
46 -    net.layers{2}.transferFcn ='tansig';
47 -    net.trainParam.epochs = 1000;
48 -    net.trainParam.show = 5;
49 -    net.trainParam.lr = 0.01;
50 -    net.trainParam.lr_inc=1.05;
51 -    net.trainParam.lr_dec=0.7;
52 -    net.trainParam.max_perf_inc=1.04;
53 -    net.trainParam.mc=0.9;
54 -    net.trainParam.goal = 0;
55 -    net.trainParam.min_grad=0;
56 -    net.trainParam.max_fail=1000;
57 -    net.divideFcn='divideind';
58 -    net.divideParam.trainInd = trIdx;
59 -    net.performParam.regularization=0;
```

Too complicated model:

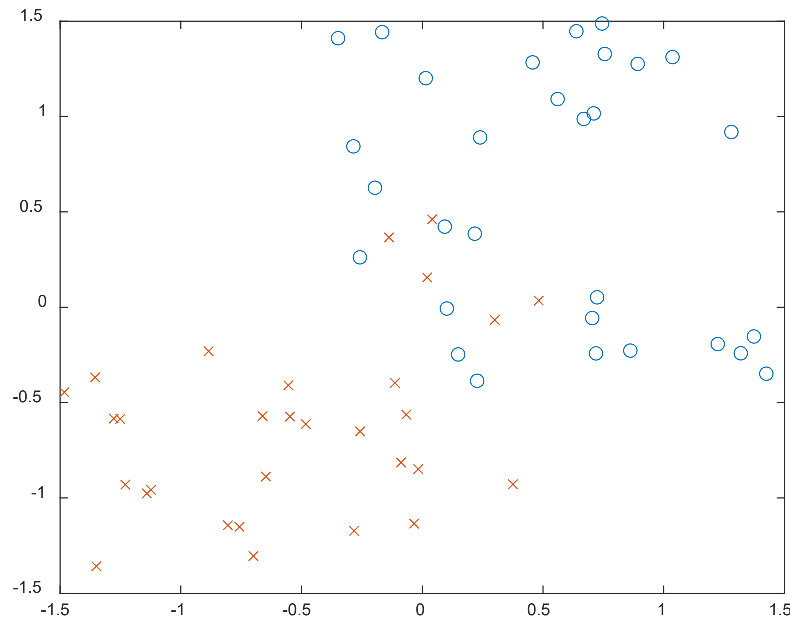


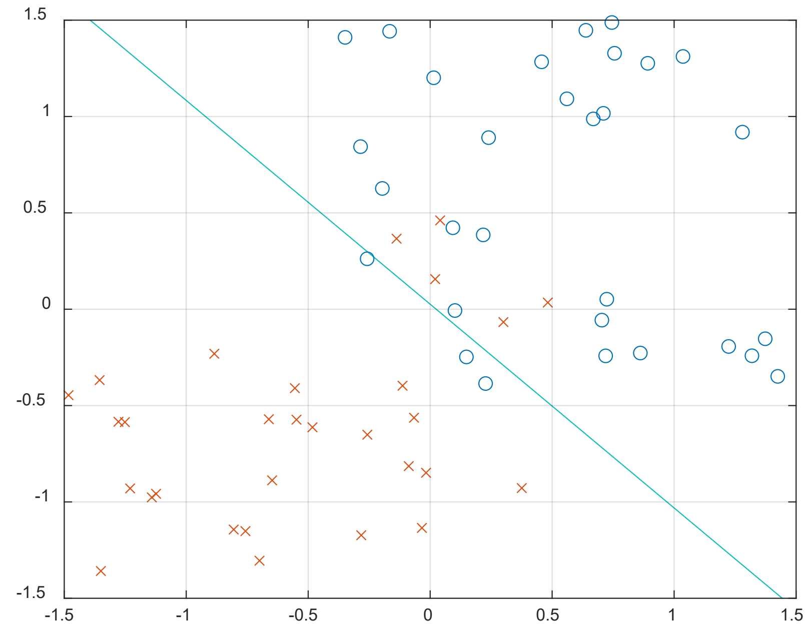


With regularization:

```
net.performParam.regularization=0.5;
```

Simpler model with satisfactory classification performance:





References:

1. S. Haykin, *Neural Networks and Learning Machines*, Prentice Hall, 2009
2. P. Wilmott, *Machine Learning: An Applied Mathematics Introduction*, Panda Ohana Publishing, 2019
3. J. Dawani, *Hands-On Mathematics for Deep Learning: Build a Solid Mathematical Foundation for Training Efficient Deep Neural Networks*, Packt Publishing, 2020
4. https://www.mathworks.com/products/deep-learning.html