EE5904/ME5404: Lecture Four Multi-layer Perceptron: Design and Training Issues

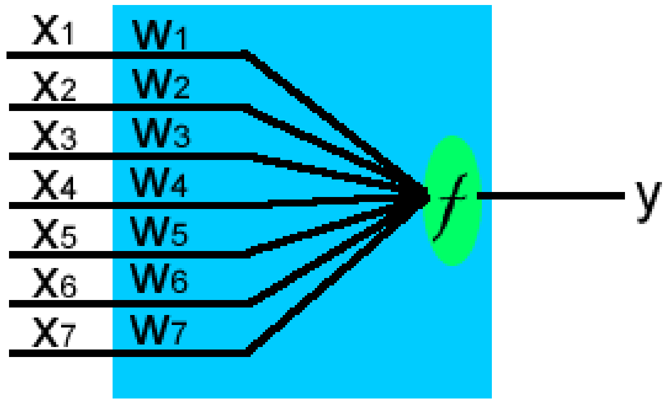
Xiang Cheng

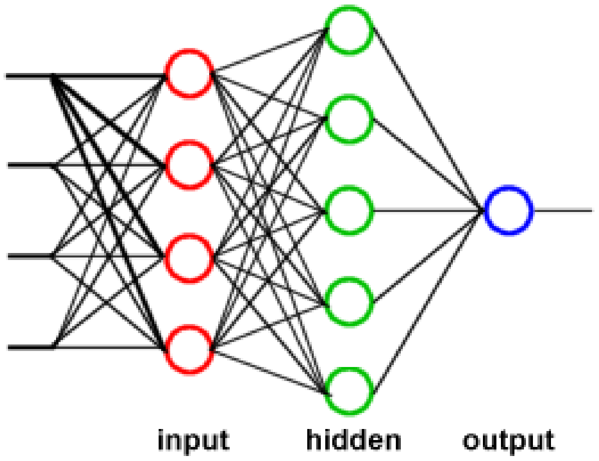
Associate Professor

Department of Electrical & Computer Engineering
The National University of Singapore

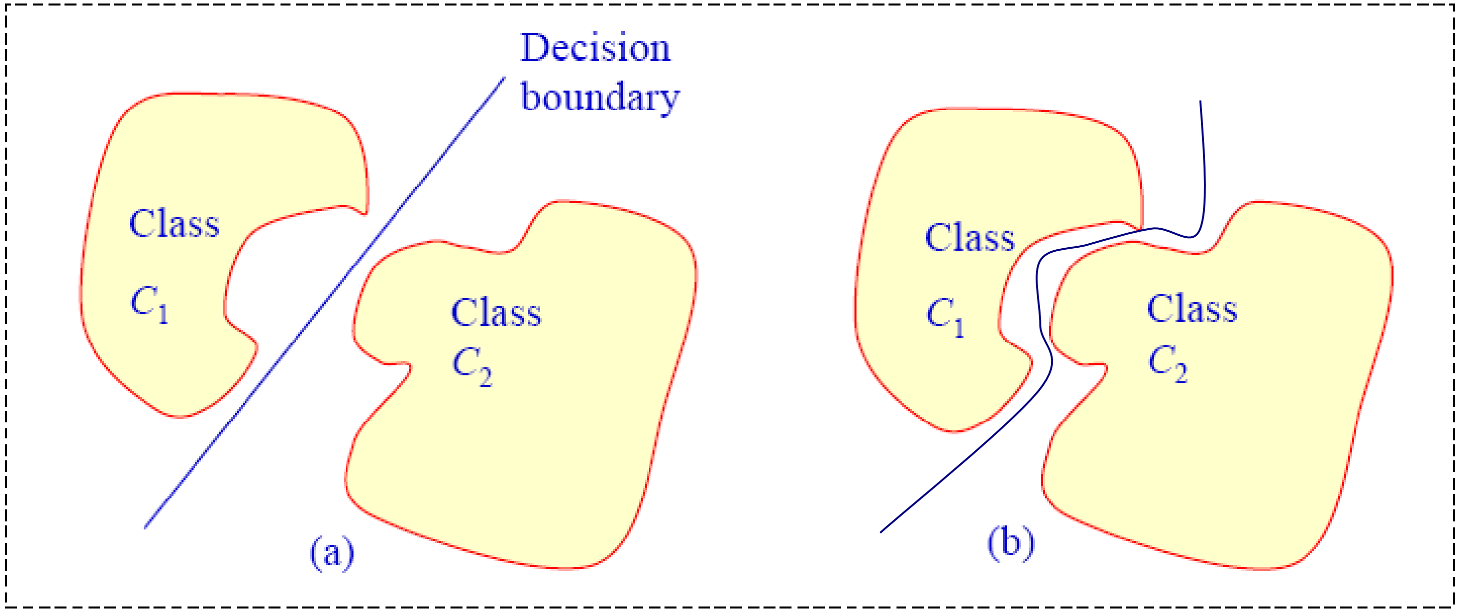
Phone: 65166210 Office: Block E4-08-07
Email: elexc@nus.edu.sg

Single Layer Perceptron v.s. Multi-layer Perceptrons



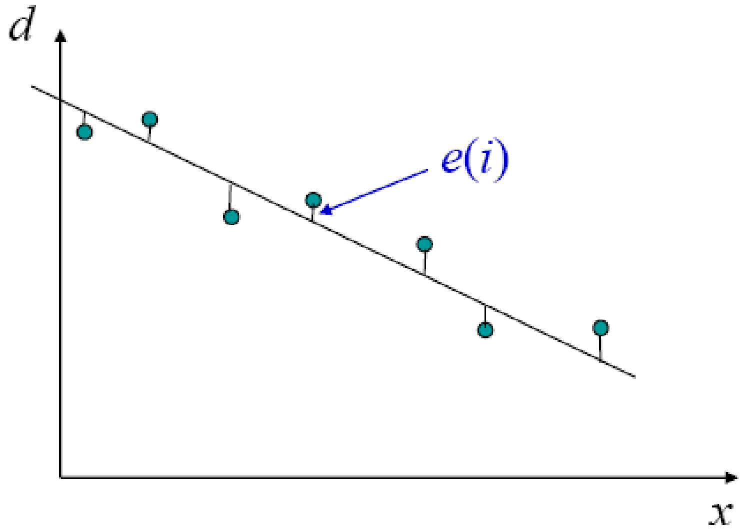


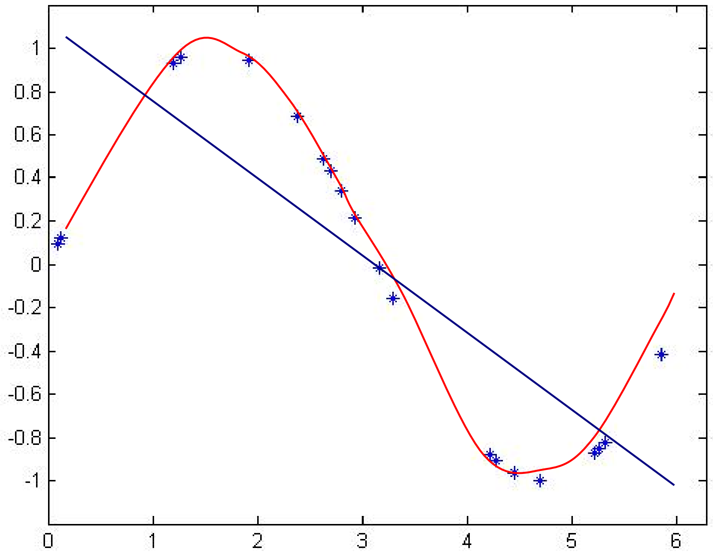
Pattern Recognition Problem:



Single Layer Perceptron v.s. Multi-layer Perceptrons

Regression Problem:





Multi-layer Perceptrons can approximate any bounded continuous functions!

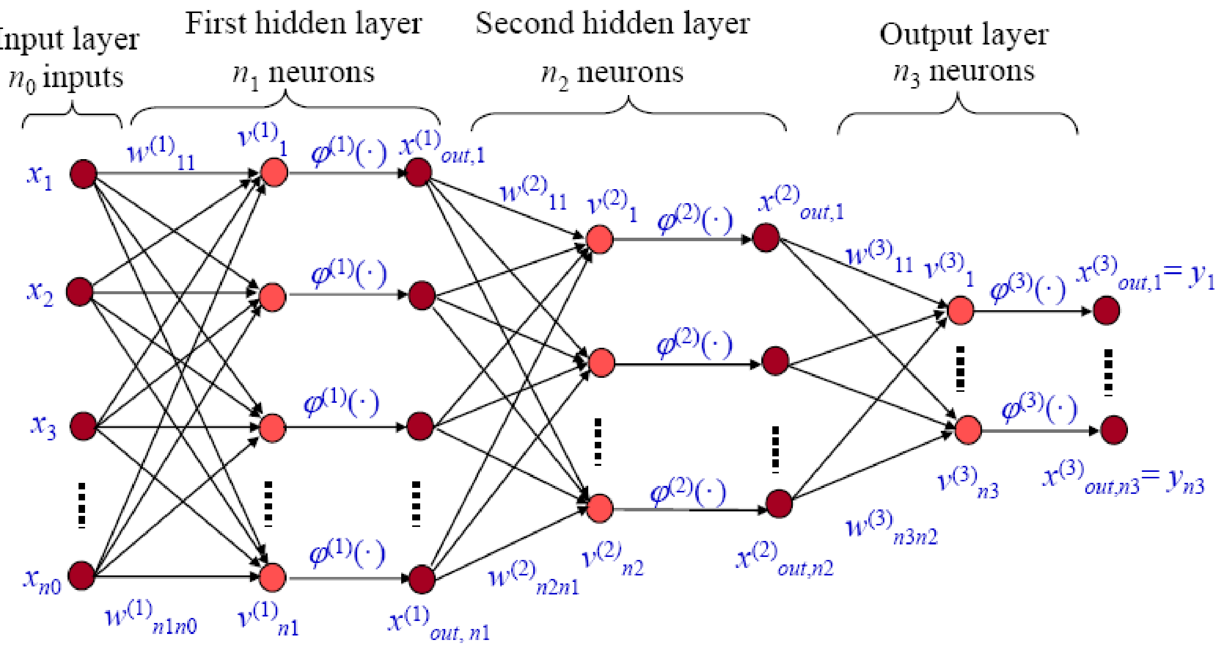
The learning algorithms are based upon the steepest descent (gradient descent) method:

$$w(k+1) = w(k) - \eta g(k)$$

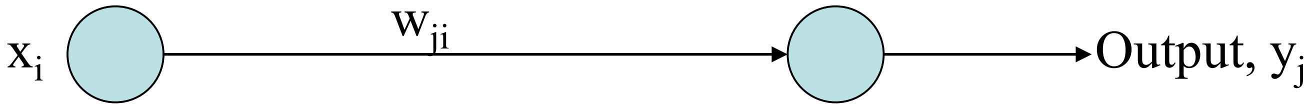
$$w(k+1) = w(k) + \eta \underbrace{e(k)}_{\text{Output Error}} \underbrace{x(k)}_{\text{Input Signal}}$$

$$w_{ji}^{(s)}(k+1) = w_{ji}^{(s)}(k) + \overbrace{\eta^{(s)} \underbrace{\delta_j^{(s)}}_{\text{Output Error}} \underbrace{x_{out,i}^{(s-1)}}_{\text{Input Signal}}}^{\Delta w_{ji}^{(s)}}$$

The learning of the Multi-layer perceptron



$$w_{ji}^{(s)}(n+1) = w_{ji}^{(s)}(n) + \eta\delta_j^{(s)}(n)x_{out,i}^{(s-1)}(n)$$

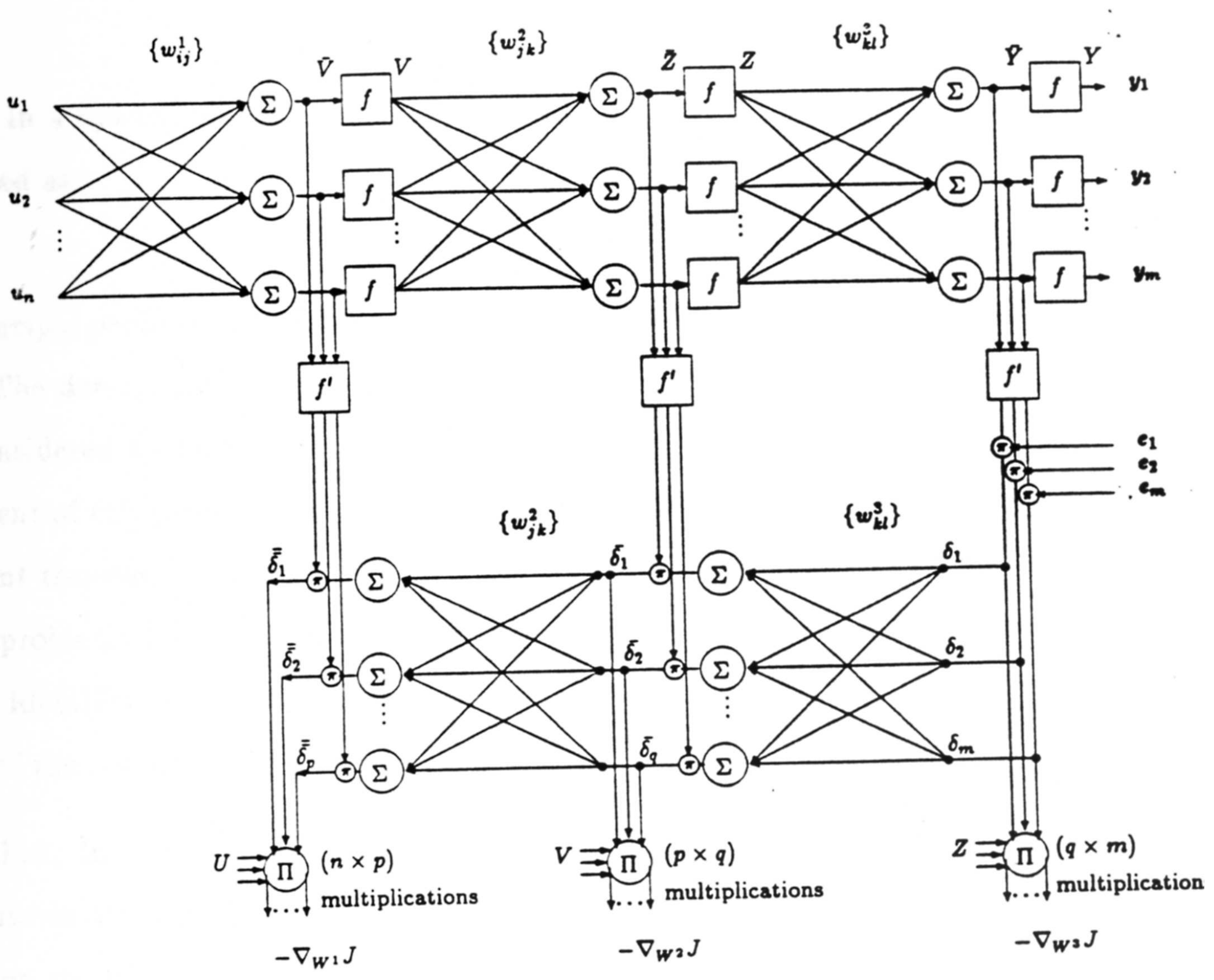


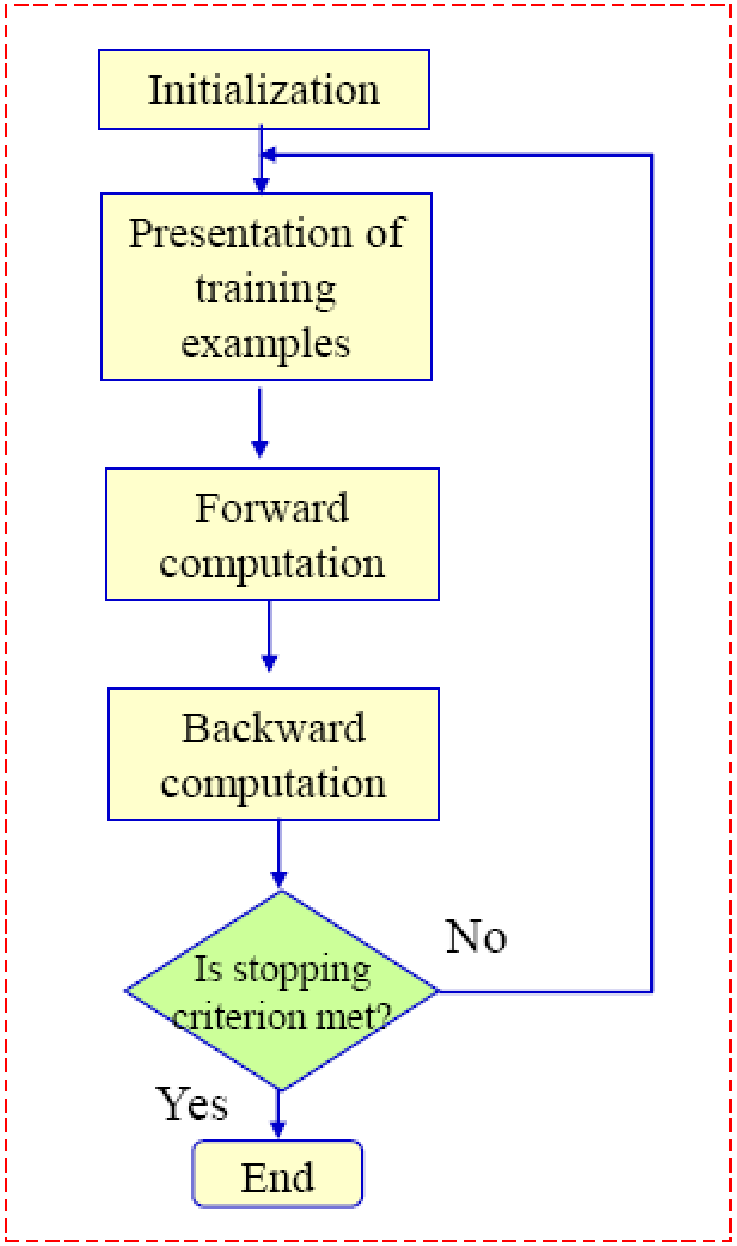
The adjustment of the synaptic weight only depends upon the information of the input neuron and the output neuron, and nothing else.

The output of the neuron depends upon all the connected neurons!

The adjustment of the synaptic weight is proportional to both the input signal and the output error.

Signal-flow graphic representation of BP


$\{w^1_{ij}\}$
$\bar{V}$ V
$\{w^2_{jk}\}$
$\bar{Z}$ Z
$\{w^3_{kl}\}$
$\bar{Y}$ Y
u_1 u_2 u_n
y_1 y_2 y_m
f f' Σ π Π
e_1 e_2 e_m
$\bar{\bar{\delta}}_1$ $\bar{\bar{\delta}}_2$ $\bar{\bar{\delta}}_p$
$\bar{\delta}_1$ $\bar{\delta}_2$ $\bar{\delta}_q$
δ_1 δ_2 δ_m
U $(n \times p)$ multiplications
V $(p \times q)$ multiplications
Z $(q \times m)$ multiplication
$-\nabla_{W^1} J$
$-\nabla_{W^2} J$
$-\nabla_{W^3} J$



Multilayer perceptron with backpropagation is very popular because it is conceptually simple, computationally efficient. And because it often works!

We are assured by mathematics that a good solution always exists. But how to find it?

BP can be very slow particular for MLP where the cost surface is typical non-quadratic, non-convex, and high dimensional with many local minima and/or flat regions. There is no formula to guarantee that

(1) The network will converge to a good solution
(2) Convergence is fast or
(3) Convergence even occurs at all.

How to get it to work?

Many design and training issues to consider:

Do you want to use the data one by one (called sequential mode), or feed them all to the network in one step (called batch mode) ?

How many hidden layers?

How many hidden neurons in each layer?

How to choose the activation function?

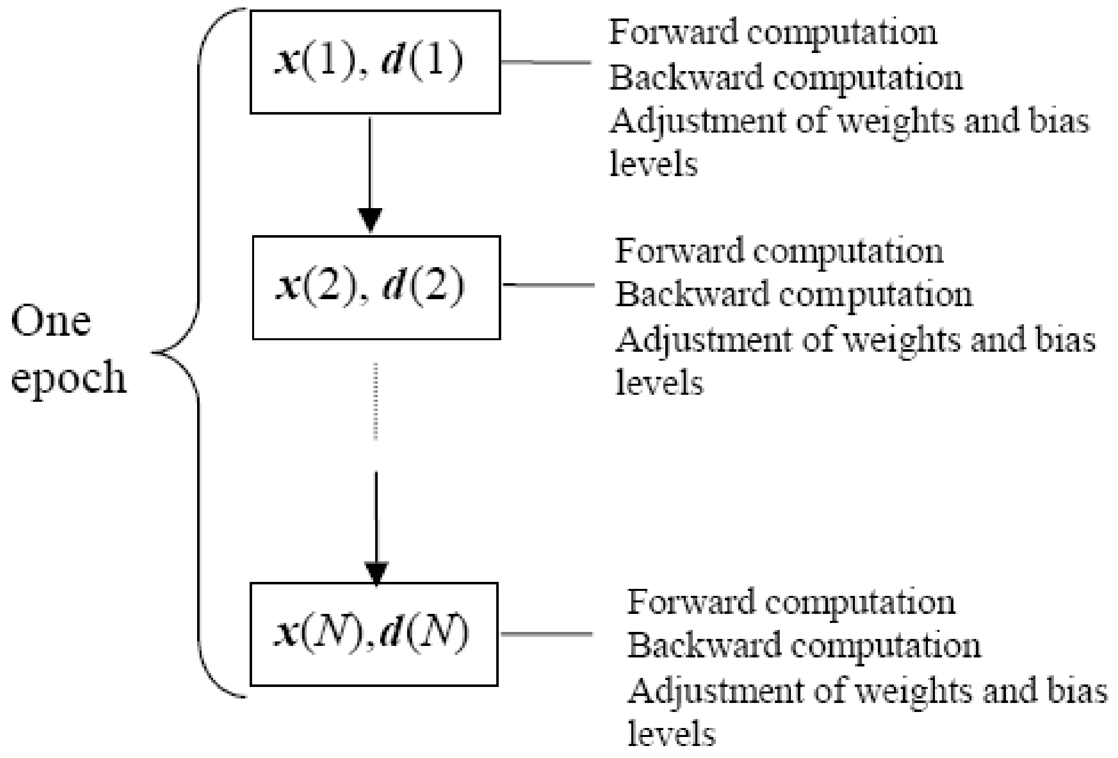
How to pre-process the input data?

How to avoid over-fitting?

BP Algorithm - Sequential and Batch Modes of Training

Two modes of back-propagation learning:

- Sequential (online, pattern, stochastic) mode: Weight updating is performed after each presentation of the training example. Consider N training examples:



$$\Delta w_{ij}(1) = -\eta \frac{\partial E(1)}{\partial w_{ij}(1)}$$

$$\Delta w_{ij}(2) = -\eta \frac{\partial E(2)}{\partial w_{ij}(2)}$$

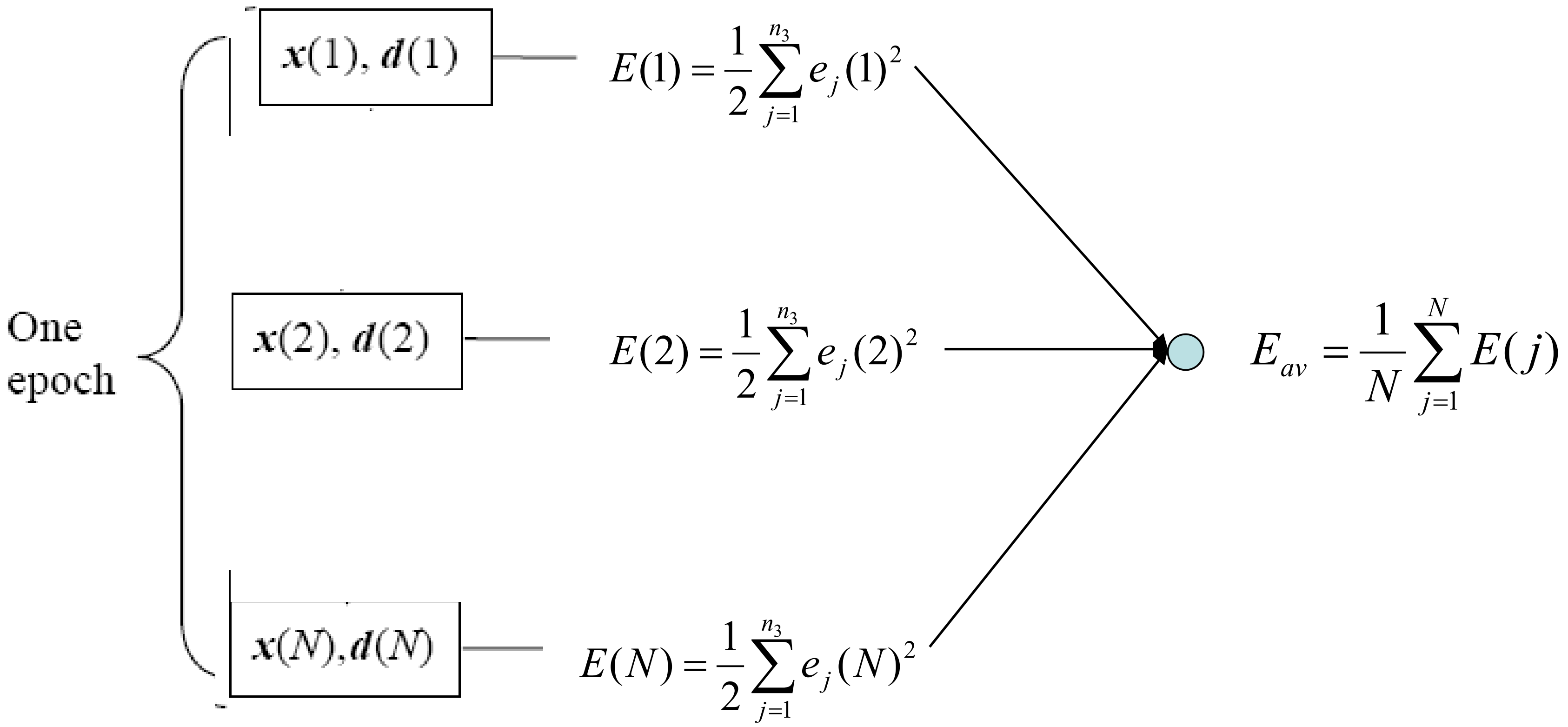
$$\Delta w_{ij}(N) = -\eta \frac{\partial E(N)}{\partial w_{ij}(N)}$$

- **<u>Did we use the sequential mode to derive the BP algorithm last week?</u>**

Yes. This is the mode we used to derive the BP!

BP Algorithm - Sequential and Batch Modes of Training

Batch Mode: Update weight after the presentation of all the training examples.



$$E(1)=\frac{1}{2}\sum_{j=1}^{n_3}e_j(1)^2$$

$$E(2)=\frac{1}{2}\sum_{j=1}^{n_3}e_j(2)^2$$

$$E(N)=\frac{1}{2}\sum_{j=1}^{n_3}e_j(N)^2$$

$$E_{av}=\frac{1}{N}\sum_{j=1}^{N}E(j)$$

Steepest Descent: $\Delta w_{ij}=-\eta\frac{\partial E_{av}}{\partial w_{ij}}=-\eta\frac{1}{N}\sum_{k=1}^{N}\frac{\partial E(k)}{\partial w_{ij}}$

$\frac{\partial E(k)}{\partial w_{ij}}$ Can be easily calculated by BP derived early for individual samples

<u>How many adjustments of the weights are done in one epoch?</u>

Only once. For sequential learning, N times in one epoch!

Which one is better?

Let's take a deeper look at the cost functions and weight adjustments:

To make it simple, let's assume that there is only one output.

Sequential mode $E(k)=\frac{1}{2}e(k)^2 \Longrightarrow \Delta w_{ij}(k)=-\eta\frac{\partial E(k)}{\partial w_{ij}}$

Sequential learning is trying to decrease the error for the individual sample at each step!

Batch mode: $E_{av}=\frac{1}{N}\sum_{j=1}^{N}E(j) \Longrightarrow \Delta w_{ij}=-\eta\frac{\partial E_{av}}{\partial w_{ij}}=-\eta\frac{1}{N}\sum_{k=1}^{N}\frac{\partial E(k)}{\partial w_{ij}}$

Batch learning is trying to decrease the average errors for all the samples at every step!

Which error should we care more in judging the performance of the MLP?
The individual error or the average errors for the whole group?

Of course, the final judgment will solely depend upon the average error for the whole group!

So theoretically speaking, the gradient $\frac{\partial E_{av}}{\partial w_{ij}}=\frac{1}{N}\sum_{k=1}^{N}\frac{\partial E(k)}{\partial w_{ij}}$ is more useful!

$\frac{\partial E(k)}{\partial w_{ij}}$ by the sequential learning is just the estimate of the true gradient, $\frac{\partial E_{av}}{\partial w_{ij}}=\frac{1}{N}\sum_{k=1}^{N}\frac{\partial E(k)}{\partial w_{ij}}$

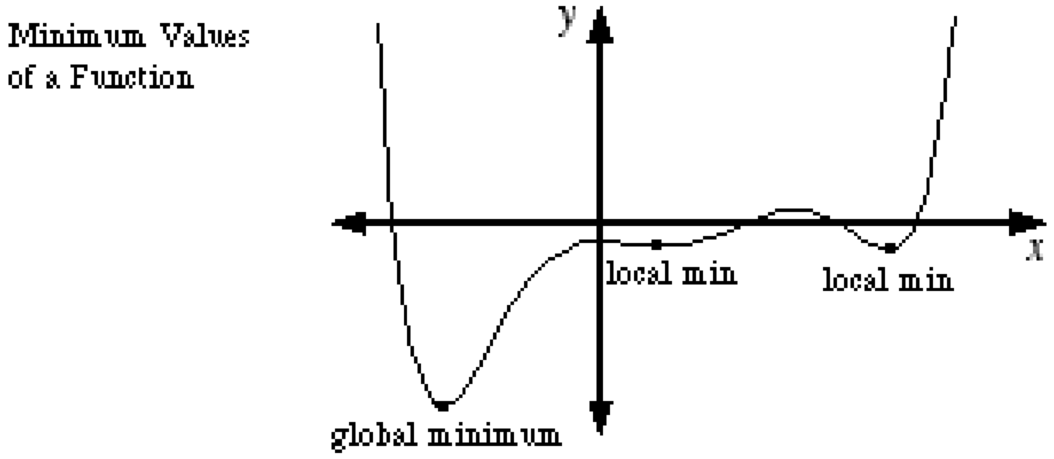
Is it a good estimation using just one sample? No! It is a very noisy estimate!

Now are you ready to tell which method is better? Batch mode or sequential mode?

Sequential learning is generally the preferred one! What a surprise!

Main reason: Sequential learning often results in better solutions.

Nonlinear networks often have multiple local minima. Batch learning can easily get trapped into the local minimum.



How to deal with this problem? We discussed a number of strategies last time.

One way is to add some noise to force it to jump out of the local minimum.

Sequential learning can deal with this problem better than batch mode! Why?

In sequential learning, the estimate of the true gradient is noisy, the weights may not move precisely down the gradient at each iteration. This "noise" can be advantageous because it may help the network to jump out of the local minimum, and move into a deeper (therefore better) local minimum.

Despite the advantage of sequential learning, there are still reasons why one might consider using batch learning.

The batch learning always converges to a minimum (either global or local).

<u>How about the sequential learning? Does it always converge?</u>

The noise, which is so critical for finding better local minima also prevents full convergence to the minimum.

Instead of converging to the exact minimum, the convergence stalls out due to the weight fluctuations. The variance of the fluctuations around the local minimum is proportional to the learning rate.

$$w_{ji}^{(s)}(k+1) = w_{ji}^{(s)}(k) + \underbrace{\eta^{(s)}\delta_j^{(s)}x_{out,i}^{(s-1)}}_{\Delta w_{ji}^{(s)}}$$

<u>How to reduce the fluctuations for sequential learning?</u>

One way is to decrease the learning rate gradually (c/t).

Another way to remove noise is to use "mini-batches", that is, start with a small batch size and increase the size as training proceeds.

Another huge advantage of batch training is that one is able to use second order methods to speed up the learning process. That is the biggest advantage for batch mode.

What are the second order methods? (4.16 in the textbook)

Training a neural network can be formulated as a nonlinear optimization problem.

Steepest (Gradient) Descent Method: $w(k+1) = w(k) - \eta g(k)$

Steepest Descent method is the easiest one to implement. However it is notoriously slow!

Consider the famous example of

Rosenbrock's valley,

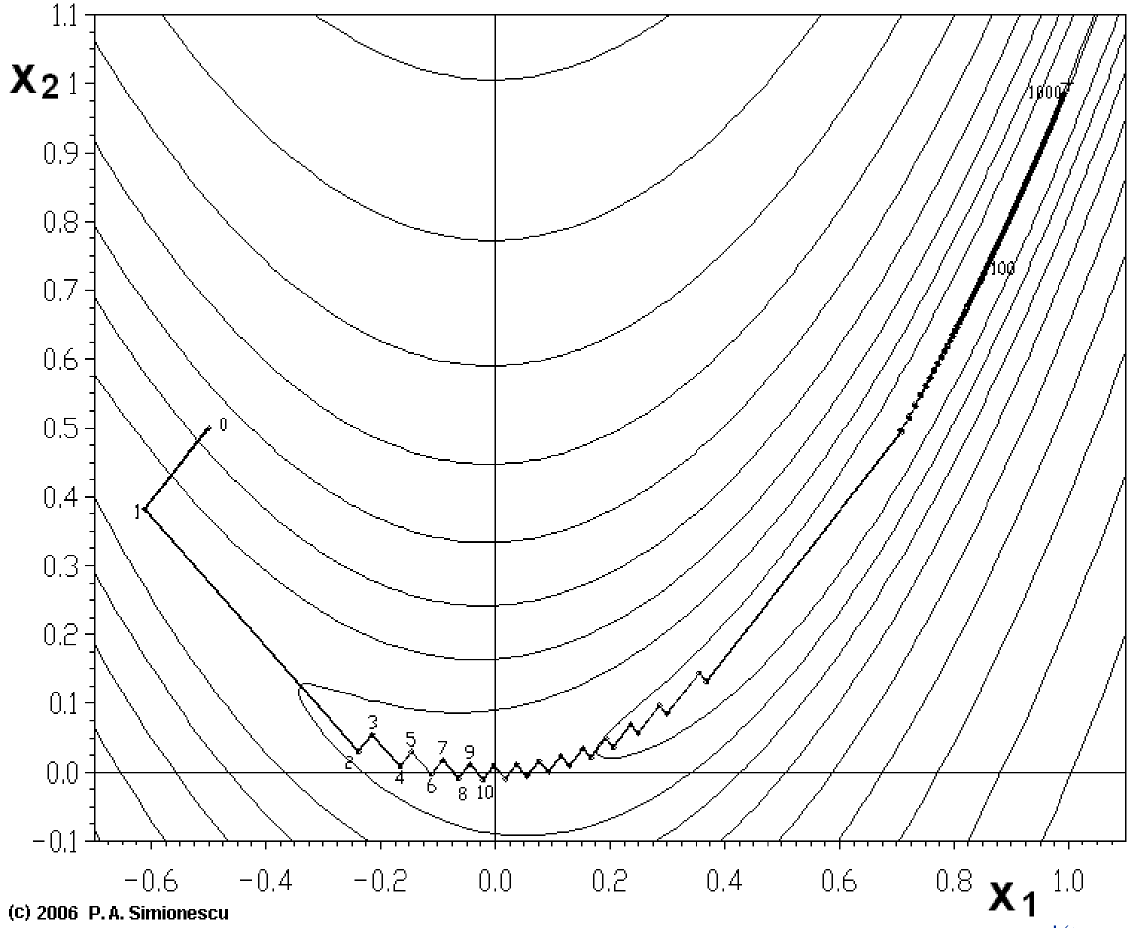
$$f(x,y) = (1-x)^2 + 100(y-x^2)^2$$

It has a global minimum at $(x,y) = (1,1)$ where $f(x,y) = 0$. The global minimum is inside a long, narrow, parabolic shaped flat valley.

Let's see what would happen when gradient descent is applied.

It is very slow!

To speed up the convergence, people proposed many second order methods.



What are the second order methods? (4.16 in the textbook)

Second order methods (using second order derivatives) are much faster than gradient descent!

When we derived the steepest descent algorithm, we only used the first order Taylor series.

To derive the second order method, we need to use second order Taylor Series

$$E(w(n+1)) = E(w(n)+\Delta w(n)) = E(w(n)) + g^T(n)\Delta w(n) + \frac{1}{2}\Delta w^T(n)H(n)\Delta w(n)$$
$$+\text{(third- and higher-order terms)}$$

Gradient vector: $g^T(n) = \dfrac{\partial E(w)}{\partial w}$ Hessian matrix: $H(n) = \dfrac{\partial^2 E(w)}{\partial w^2}$

Let's ignore the higher-order terms, and consider the following question:

How to choose $\Delta w(n)$ such that the cost at the next step, E(w(n+1)), is minimized?

$$\frac{\partial E(w(n+1))}{\partial \Delta w(n)} = g^T(n) + \Delta w^T(n)H(n) = 0$$

$$H^T(n)\Delta w(n) = -g(n) \quad \xrightarrow{H^T(n) = H(n)} \quad \Delta w(n) = -H^{-1}(n)g(n)$$

This is so called the Newton's method.

It can converge to the minimum in one step if the cost function is quadratic!

It is faster at the high cost of computing Hessian matrix H(n)!

How to do it without directly computing H(n)?

Many types of second order methods:

Newton and Gauss-Newton algorithms, Levenberg-Marquardt algorithms

conjugate-gradient algorithms quasi-Newton algorithms

There is no single best method for nonlinear optimization. You need to choose a method based on the characteristics of the problem to be solved.

Three general types of algorithms have been found to be effective for most practical purposes:

•For a small number of weights, stabilized Newton and Gauss-Newton algorithms, including various Levenberg-Marquardt algorithms, are efficient. The memory required by these algorithms is proportional to the square of the number of weights.

•For a moderate number of weights, various quasi-Newton algorithms are efficient. The memory required by these algorithms is proportional to the square of the number of weights.

•For a large number of weights, various conjugate-gradient algorithms are efficient. The memory required by these algorithms is proportional to the number of weights.

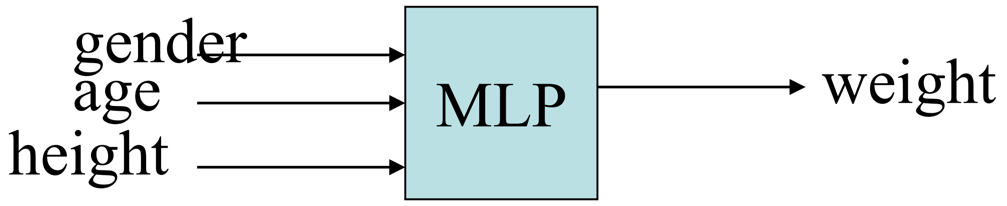
Levenberg-Marquardt algorithm is available in the deep learning toolbox in MATLAB, called “trainlm”. It is found to be very efficient in many applications.

If you run into “out of memory” problem using “trainlm”, then you can switch to “traincgf” --- MATLAB command for conjugate-gradient method.

Normalizing the Inputs

Why do we need to normalize the input variables?

Consider the following regression example: we try to build a map which uses the person's age, gender and height to predict the average weight.



Note that the ranges of the three input variables are very different:

Gender: 1 or -1, 1 or 0 Age: 0--120 Height: 0---2.5m

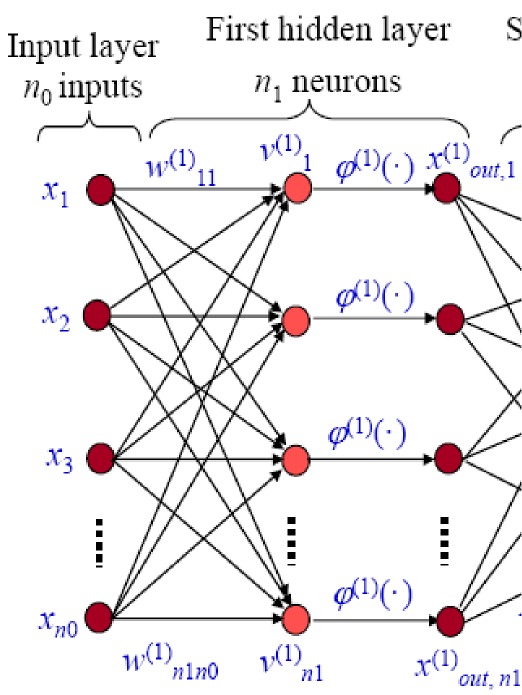
Does the MLP "understand" that the first input indicates the "gender", and the second is the "age"?

The neural network would treat all the input variables equally as simple numbers!
If the input is [1 30 2], the neural network would recognize that the second input value is much higher than others.
But in reality, which variable is HUGE in this input example of [1 30 2] ? 2 meters!

Therefore, it is important to normalize the input variables such that they have similar ranges!

We need to bring all the variables to the similar ranges to avoid any misunderstanding by the neural network.

Consider the situation that you want to normalize all the inputs to the same range of either [0,1] or [-1, 1].



Does it make any difference among these two choices for MLP?

We need to check out the effect of the inputs on the training algorithm.

$$w_{ji}^{(s)}(k+1) = w_{ji}^{(s)}(k) + \underbrace{\eta^{(s)}\delta_j^{(s)}x_{out,i}^{(s-1)}}_{\Delta w_{ji}^{(s)}}$$

What would happen if all the inputs are normalized to the range of [0,1]

Let's consider all the synaptic weight adjustments in the first hidden layer that connect to jth hidden neuron, then

$$\Delta w_{ji}^{(1)}(k) = \eta\delta_j^{(1)}(k)x_i(k), i = 1,2,\cdots,n_0$$

Is there any thing special about the algebraic sign of the change $\Delta w_{ji}^{(1)}(k)$ if all $x_i(k) \geq 0$?

All of the updates of weights that feed into a node will have the same sign. As a result, these weights can only all decrease or increase together for a given input pattern.

Is this efficient for updating the weights?

It is better to give more freedom to the weight adjustments!

$$\Delta w_{ji}^{(1)}(k) = \eta\delta_j^{(1)}(k)x_i(k), i = 1,2,\cdots,n_0$$

In general, any shift of the average input away from zero will bias the updates in a particular direction and thus slow down learning.

Therefore it is good to shift the inputs so that the average over the training set is close to zero.

Convergence is also faster if they are scaled so that all have about the similar size.

Can you find out the reason by looking at the learning algorithm?

The bigger the input, the larger the change!

Scaling speeds learning because it helps to balance out the rate at which the weights connected to the input nodes learn.

How to normalize the input variables?

There are many ways. One possibility is

$$\bar{x}_i = \frac{\sum_{n=1}^{N} x_i(n)}{N} \qquad \sigma = \sqrt{\frac{\sum_{n=1}^{N} (x_i(n) - \bar{x}_i)^2}{N}}$$

$$x_i'(n) = \frac{(x_i(n) - \bar{x}_i)}{\sigma}$$

The resulting variable would have zero mean and unit variance.

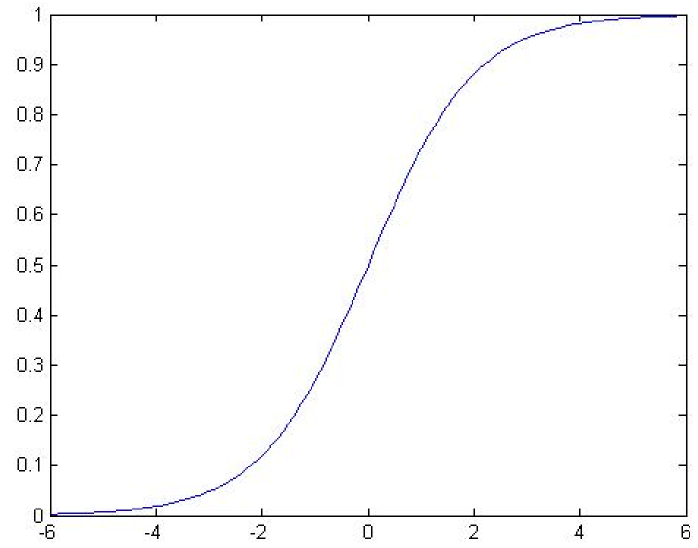
The normalization is done for each variable separately!

Do we always need to normalize the inputs?

Of course it is not necessary if all the input variables have the same physical meaning and the ranges are already close to [-1 1].

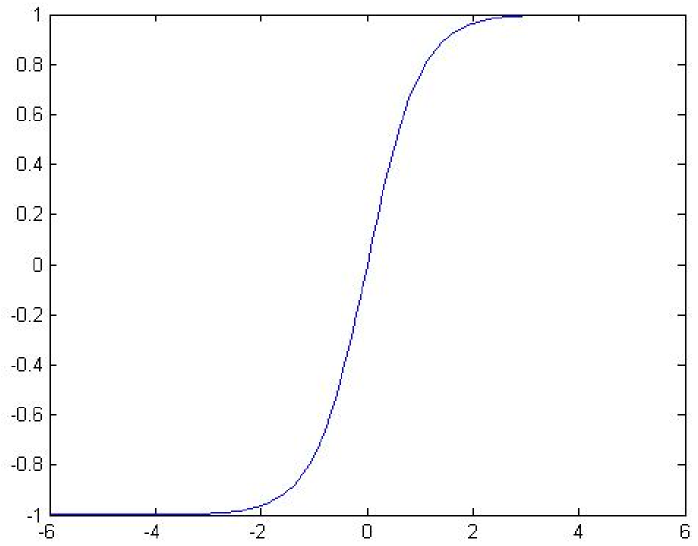
How to choose the activation functions in the hidden neurons?

There are two types of sigmoid functions provided in MATLAB.



Logistic function:

$$\log sig(x) = \frac{1}{1+e^{-x}}$$



Hyperbolic Tangent function:

$$\tan sig(x) = \frac{2}{1+e^{-2x}} - 1$$

Which one should be preferred? And Why?

$$w_{ji}^{(s)}(k+1) = w_{ji}^{(s)}(k) + \underbrace{\eta^{(s)}\delta_j^{(s)}x_{out,i}^{(s-1)}}_{\Delta w_{ji}^{(s)}}$$

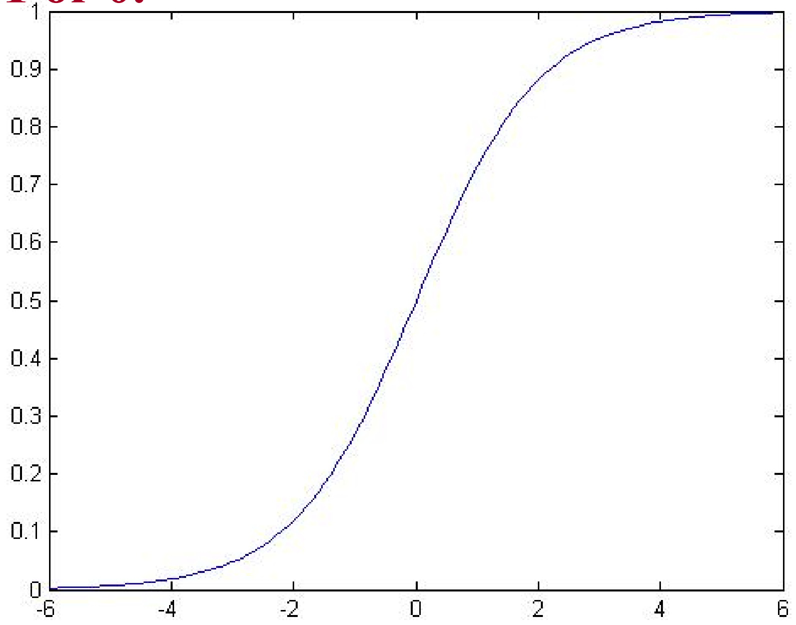
Tansig is better! The previous heuristic should be applied at all hidden layers which means that we want the average of the outputs of a node to be close to zero because these outputs are the inputs to the next layer.

How to choose the activation functions in the output neuron?

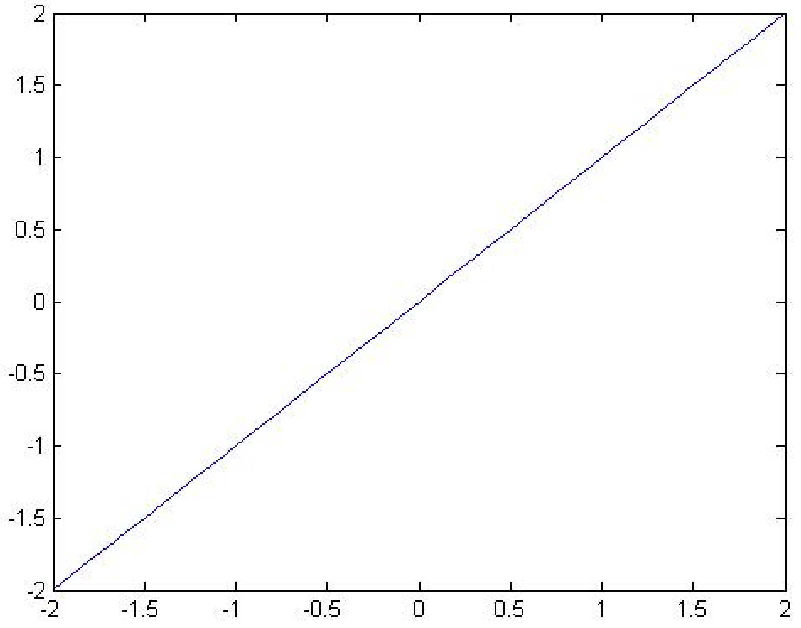
Pattern Recognition

What is the desired output ?

1 or 0.



Regression Problem



$$\log sig(x) = \frac{1}{1+e^{-x}}$$

$$purelin(x) = x$$

In the regression problem, the outputs may not necessarily lie in the range of (0, 1) Or (-1, 1). Using linear neuron at the output layer would be more flexible.

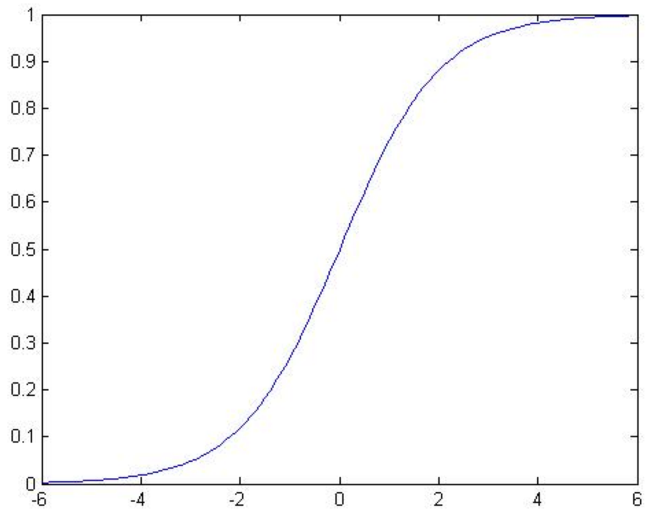
We can still use logsig or tansig if the outputs are normalized properly.

How to Choose the Target Values for Pattern Recognition ?

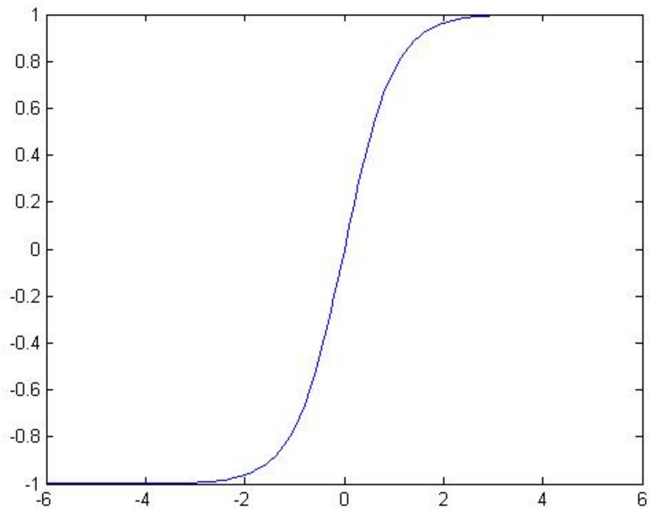
Consider the speech recognition problem in the last lecture, what did we choose the target values for the two person's pronunciation of "hello"?

[1,0] for one person, and [0,1] for another.

In classification problems, target values are typically binary (e.g. {-1,+1}, {1,0}). It is natural to set the target values at the values of the sigmoid's asymptotes. 1 and 0 for logsig, 1 and -1 for tansig. **Right?**



Logistic function:



Hyperbolic Tangent function:

But there is a little problem!

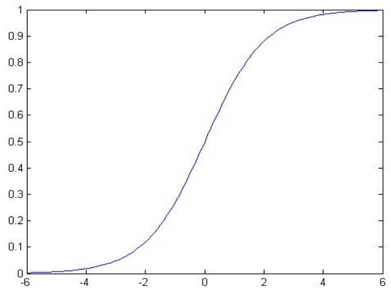
Can the target value of 1 or 0 be ever reached exactly for logistic function?

How to Choose the Target Values for Pattern Recognition ?

The training process will try to drive the output as close as possible to the target values, which can only be achieved asymptotically.

Therefore, the magnitude of the induced local fields $v_j^{(3)}(n) = \sum_{i=1}^{n_2} w_{ji}^{(3)}(n) x_{out,i}^{(2)}(n)$

in the output layer will be driven to bigger and bigger values in the training process.



How about the magnitudes of the weights in the output layer?

The weights are driven to larger and larger values.

How about the derivative of the activation function evaluated at large values?

Very close to zero. Therefore it will slow down the learning!

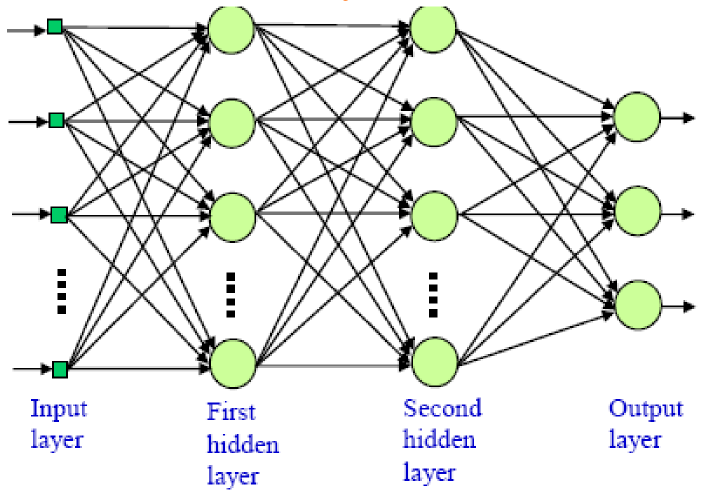
There is also another problem. When an input pattern falls near a decision boundary the output class is uncertain. Ideally this should be reflected in the network by an output value that is in the middle (around 0.5). However, large weights tend to force all outputs to the tails of the sigmoid (either 1 or 0) regardless of the uncertainty.

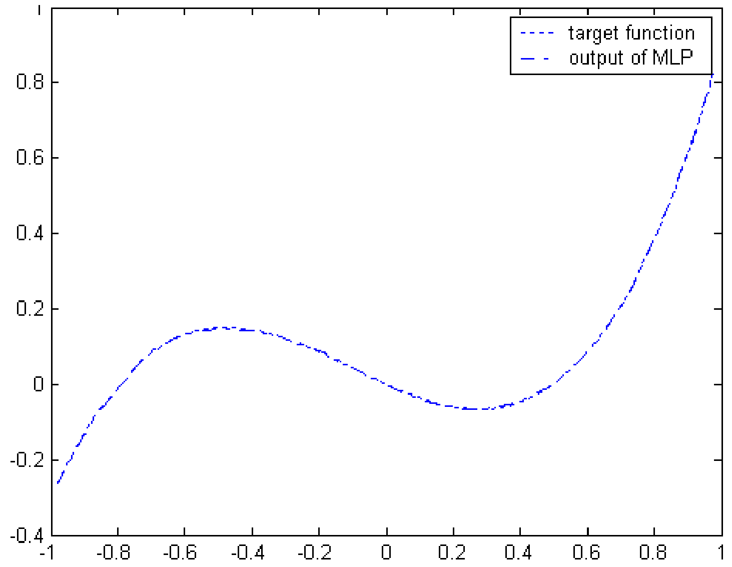
A solution to these problems is to set the target values to be within the range of the sigmoid, rather than at the asymptotic values. Setting the target values to the point of the maximum second derivative on the sigmoid is usually the best way.

For tansig function in MATLAB, the recommended target values are [-0.6, 0.6]

For logsig function in MATLAB, the recommended target values are [0.2, 0.8]

How many hidden neurons, and how many hidden layers?





<u>What are the possible ways?</u>

Trial and Error

Network growing:

Start with a small MLP, and then add a new neuron or a new layer only when we are unable to meet the design specification.

Network pruning:

Start with a large MLP, and then prune it in a systematic fashion. Please refer to section 4.14 in the textbook.

<u>Is there any simpler way rather than these time-consuming techniques?</u>

<u>For instance, can we tell the minimal number of hidden neurons by simply looking at the geometrical shape of the target function?</u>

In the rest of the lecture, we are going to discuss some research results on how to select the number of neurons and layers for MLP.

These results are not covered by the textbook (2009) yet. Please refer to the following three papers (which will be provided to you as supplementary material later) for further details:

C. Xiang, S.Q. Ding and T. H. Lee, "Geometrical Interpretation and Architecture Selection of MLP," *IEEE Trans. Neural Networks,* vol. 16, no. 1, pp.84-96, 2005.

E.J. Teoh, K.C. Tan and C. Xiang, "Estimating the Number of Hidden Neurons in a Feedforward Network using the Singular Value Decomposition," *IEEE Trans. Neural Networks,* vol. 17, no. 6, pp. 1623-1629, 2006.

S.Q. Ding and C. Xiang, "Overfitting Problem: A New Perspective from the Geometrical Interpretation of MLP". in Design and Application of Hybrid Intelligent Systems, ed. A. Abraham, M. Koppen and K. Franke (2003): 50-57. Holland: IOS Press. (The Third International Conference on Hybrid Intelligent Systems, 14 - 17 Dec 2003, Melbourne, Australia)

Break

- Visions of The Future 4

Instead of looking at the neural network as a black box, let's ask

A Deeper Question

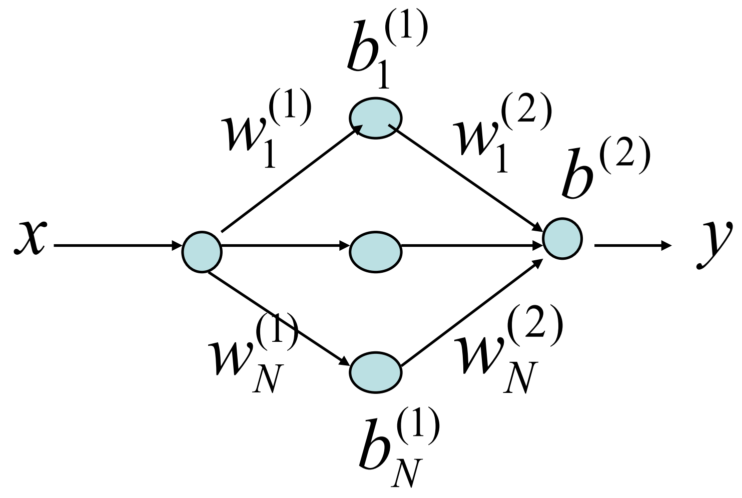
What are the geometrical meanings of

--- the number of neurons?
--- the weights?
--- the biases?

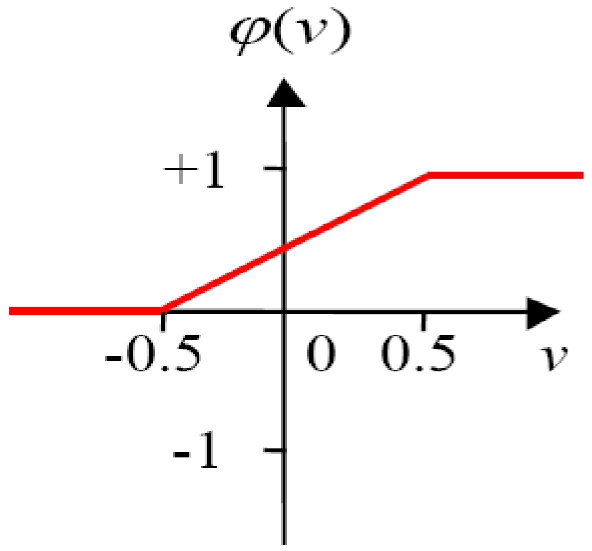
Once the geometrical (not biological) meanings are understood, then it is easy to design the network properly.

1-N-1 MLP with one hidden layer

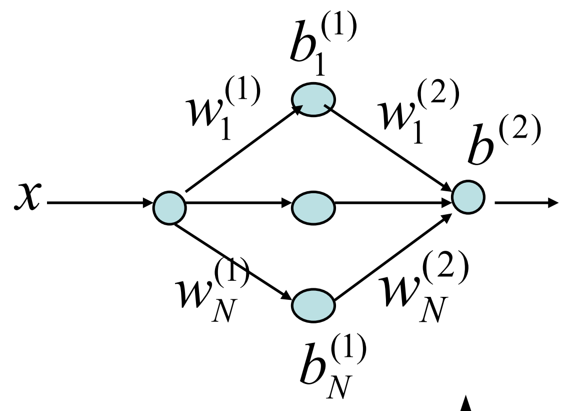
Structure



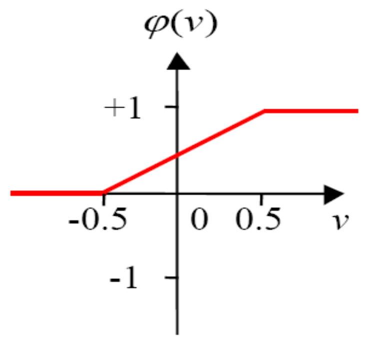
Activation Function in the hidden neurons:
Piecewise linear function



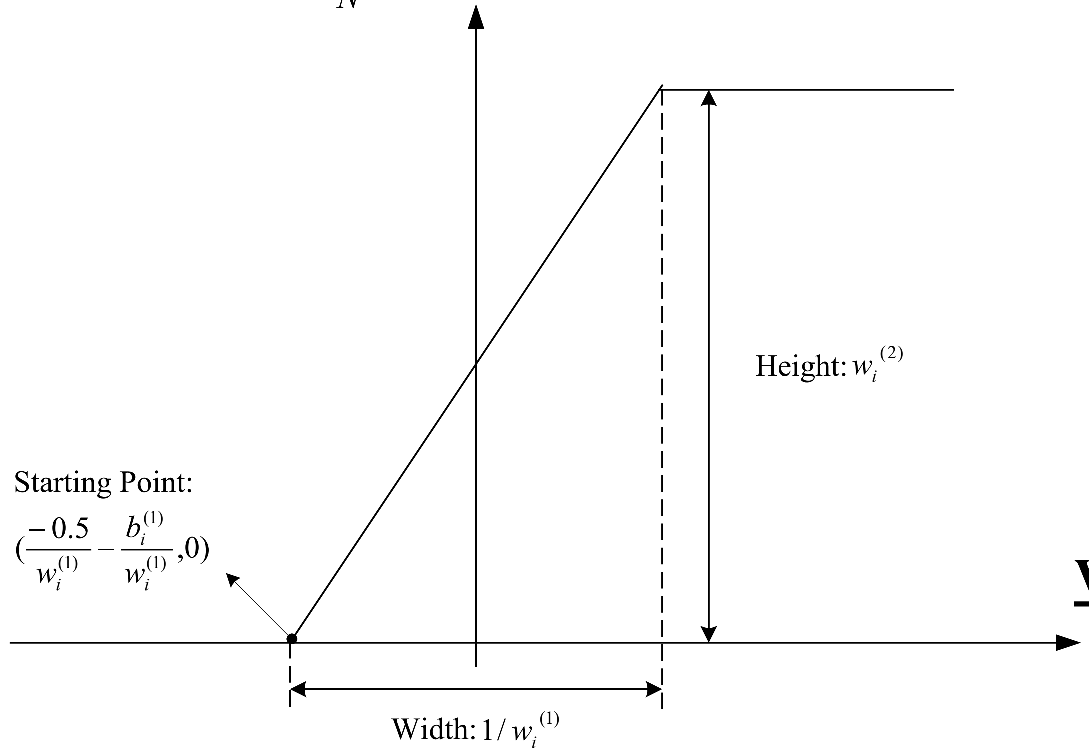
The Geometrical Interpretation



$$y(x)=\sum_{i=1}^{N} w_i^{(2)}\varphi(w_i^{(1)}x+b_i^{(1)})+b^{(2)}$$



Let's plot out each term: $w_i^{(2)}\varphi(w_i^{(1)}x+b_i^{(1)})$



Neurons ------ Building Blocks

What is the height of the building block?

$w_i^{(2)}$ Weights in the output layer

What is the width of the building block?

$\frac{1}{w_i^{(1)}}$ Weights in the hidden layer

Where is the center of the building block?

$\frac{-b_i^{(1)}}{w_i^{(1)}}$ Biases and weights in the hidden layer

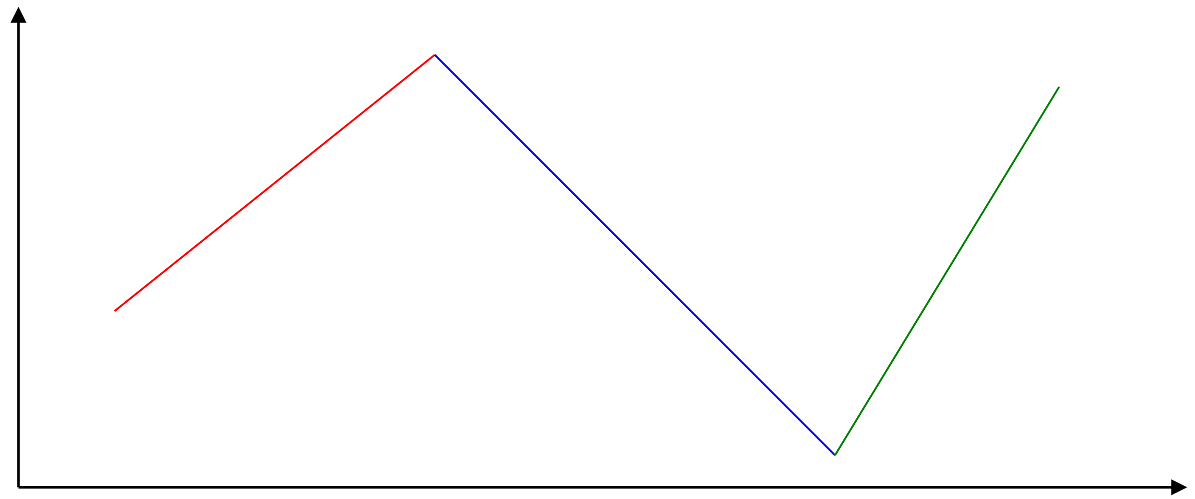
What is the slope of the building block?

$w_i^{(1)}w_i^{(2)}$ Production of the weights!

How about the meaning of the bias in the output neuron ?

$b^{(2)}$ is simply a shift of the output!

How many hidden neurons are needed to approximate a target function which is piecewise linear?

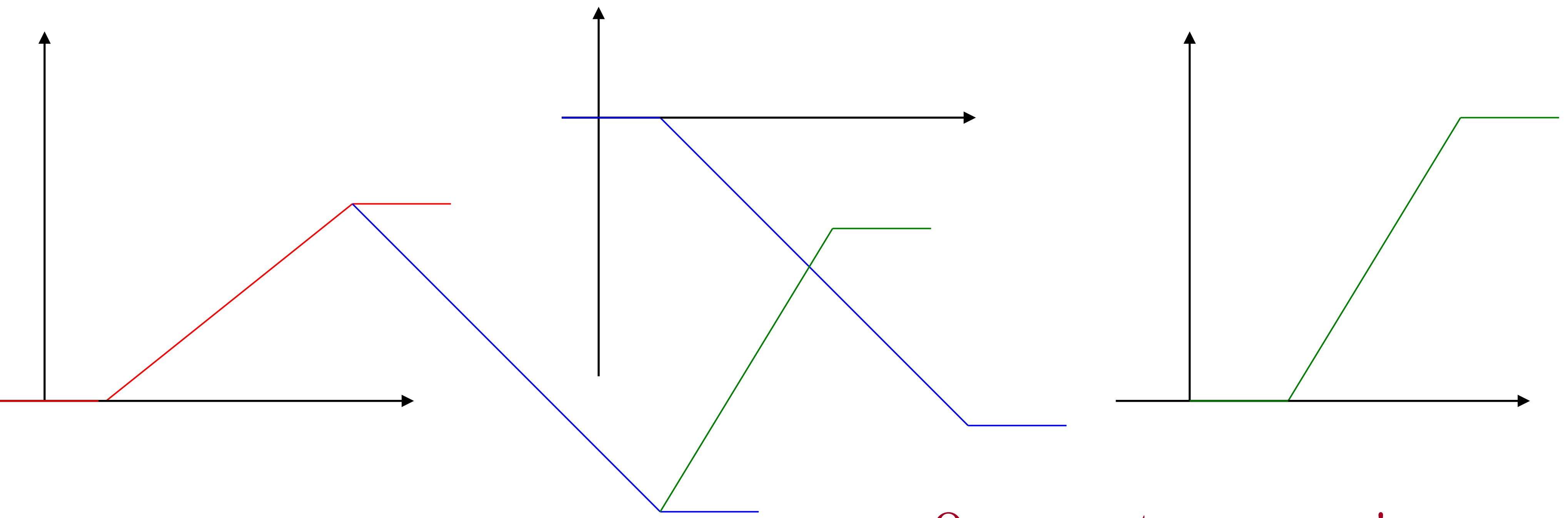


How many building blocks are needed to construct this target function?

The first segment

The second segment

The third segment



One segment—one neuron!

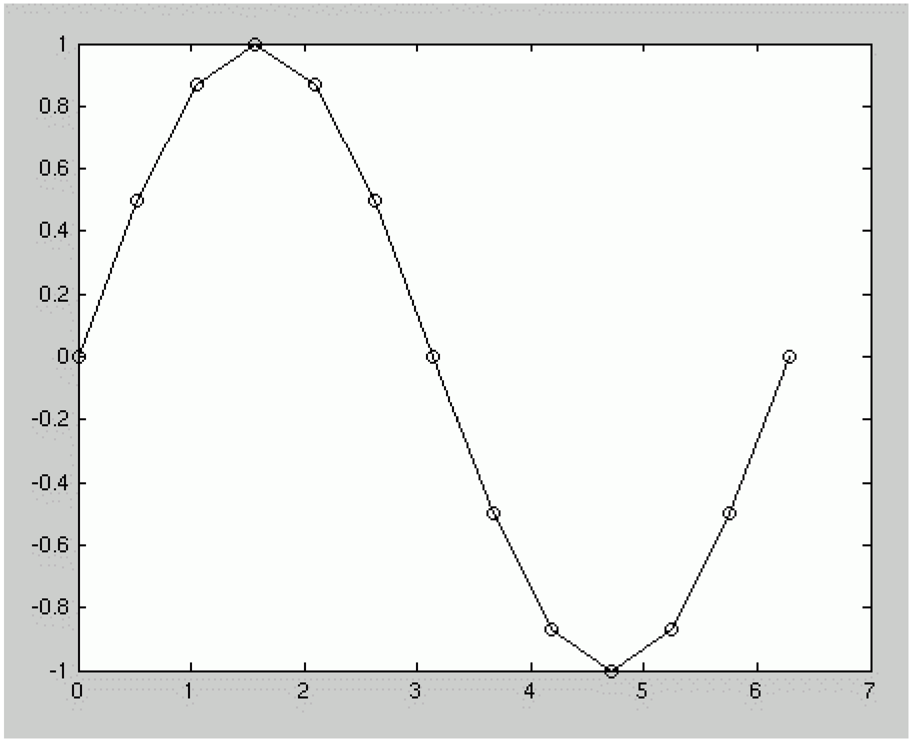
How many neurons are needed for approximating piecewise linear function?

The minimal number corresponds to the number of line segments!

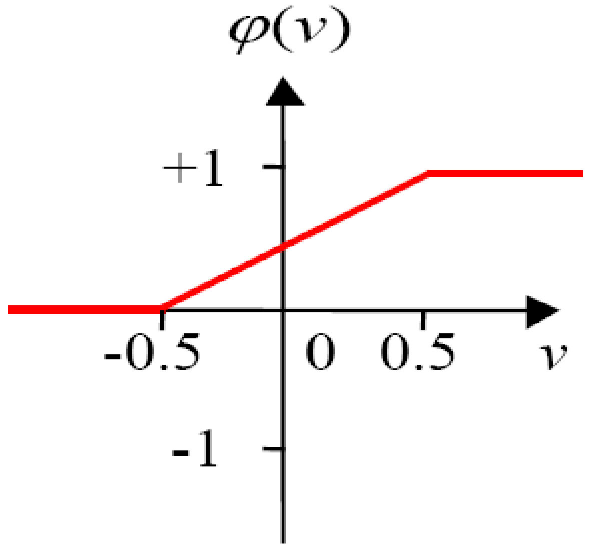
But how about functions that are smooth instead of piecewise linear?

Can any continuous function be approximated by piecewise linear function?

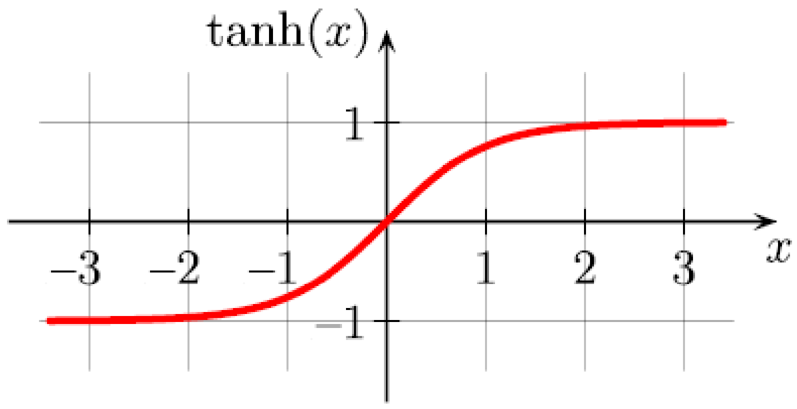
Sure.



Of course, we are going to use sigmoid function instead of piecewise linear function.



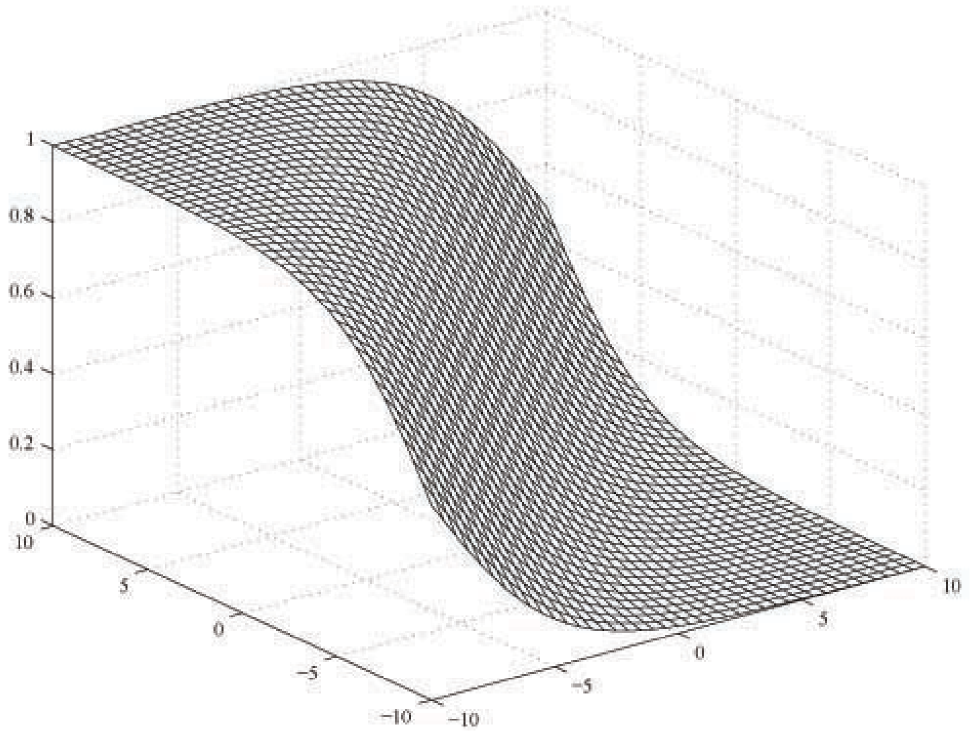
$\Rightarrow$



The sigmoid function is even better since it is close to the piecewise linear function in the middle, but also smoothens out the discontinuity!

What does the building block look like in two-dimensional problem ?

Piece-wise planes!



How many hidden neurons ?

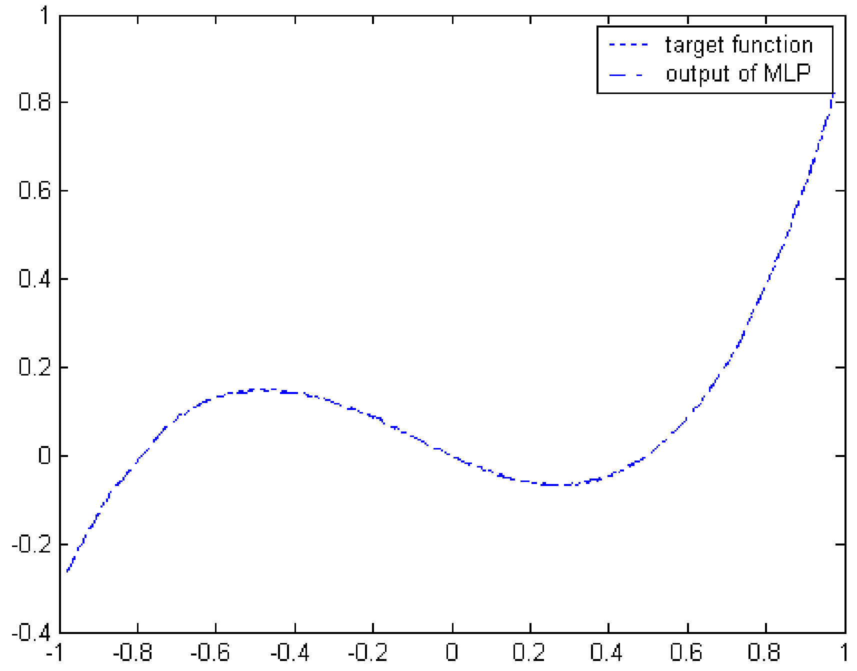
<u>Guideline:</u> Estimate the minimal number of line segments (or hyperplanes in high dimensional cases) that can construct the basic geometrical shape of the target function, and use this number as the first trial for the number of hidden neurons of the two-layered MLP.

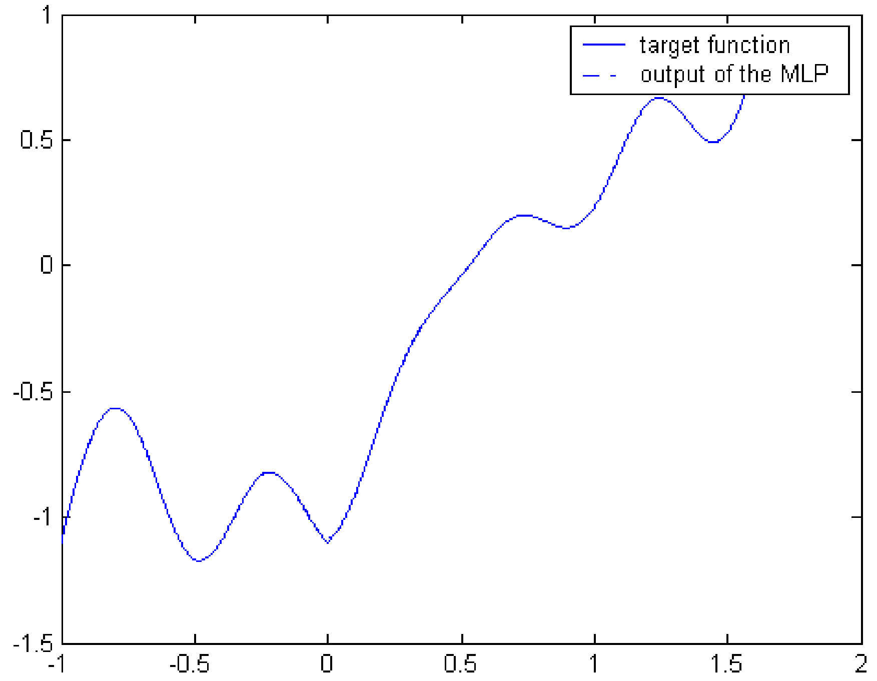
If this number is not good enough, then it provides a good starting point for further growing or pruning the network.

One Dimensional Examples

$$f(x) = x^3 + 0.3x^2 - 0.4x, x \in [-1,1]$$

$$y = 0.5\sin(\pi x)^3 - \frac{2}{x^3+2} - 0.1\cos(4\pi x) + |x|, \quad x \in [-1,1.6]$$





<u>How many lines do we need to construct the basic shape?</u>

1-3-1

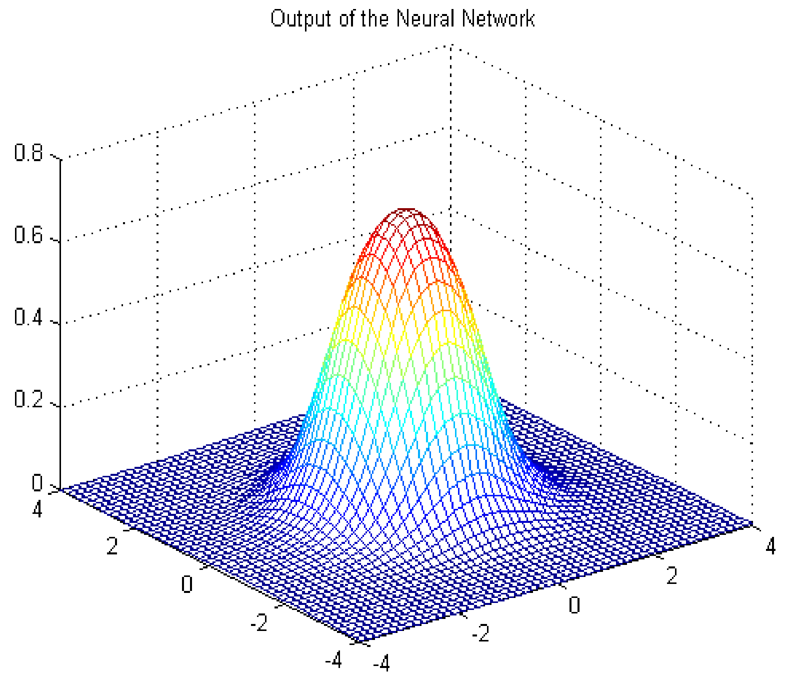
<u>How many lines do we need to construct the basic shape?</u>

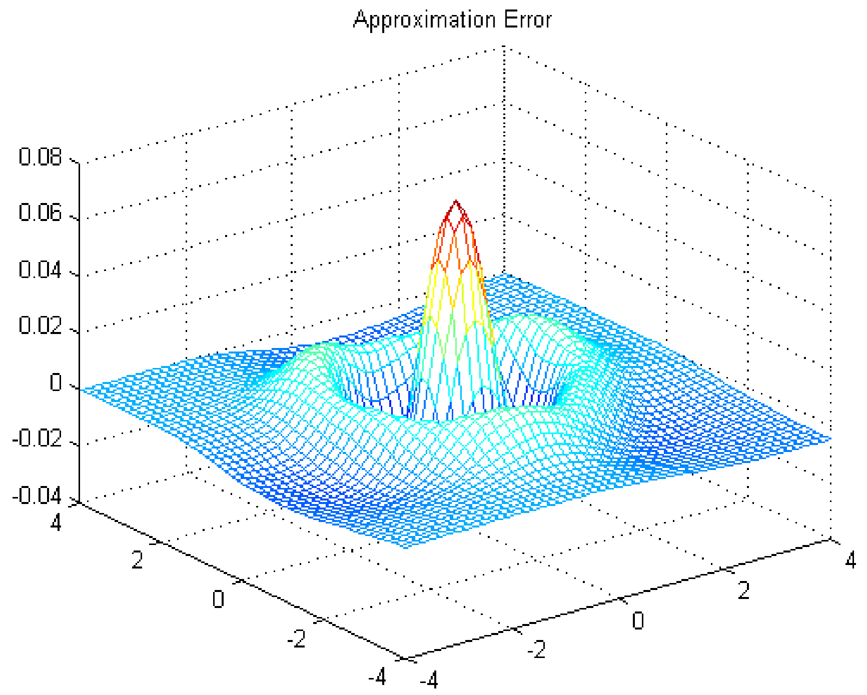
1-9-1

Back

```
%Matlab code
clc
clear all;
%% sampling points in the domain of [-1,1]
x=-1:0.02:1;
%% generating training data, and the desired outputs
y=x.^3 + 0.3 * x.^2 -0.4 * x;
 %% specify the structure and learning algorithm for MLP
net = feedforwardnet(3,'trainlm');
net.layers{1}.transferFcn = 'tansig';
net.layers{2}.transferFcn = 'purelin';
net = configure(net,x,y);
net.trainparam.lr=0.01;
net.trainparam.epochs=10000;
net.trainparam.goal=1e-8;
net.divideParam.trainRatio=1.0;
net.divideParam.valRatio=0.0;
net.divideParam.testRatio=0.0;
%% Train the MLP
[net,tr]=train(net,x,y);
 %% Test the MLP, net_output is the output of the MLP, ytest is the desired output.
xtest=-1:0.01:1;
ytest=xtest.^3 + 0.3 * xtest.^2 -0.4 * xtest;
net_output=sim(net,xtest);
%% Plot out the test results
plot(xtest,ytest,'b+');
hold on;
plot(xtest,net_output,'r-');
hold off
```

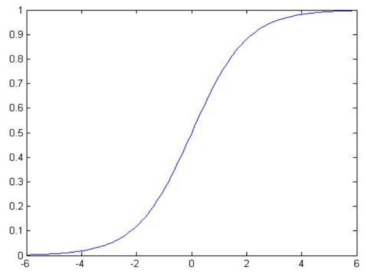
$$f(x,y)=\frac{5}{2\pi}e^{-\frac{x^2+y^2}{2}}, \quad x,y\in[-4,4]$$





How many planes do we need to construct the basic shape of the Gaussian function?

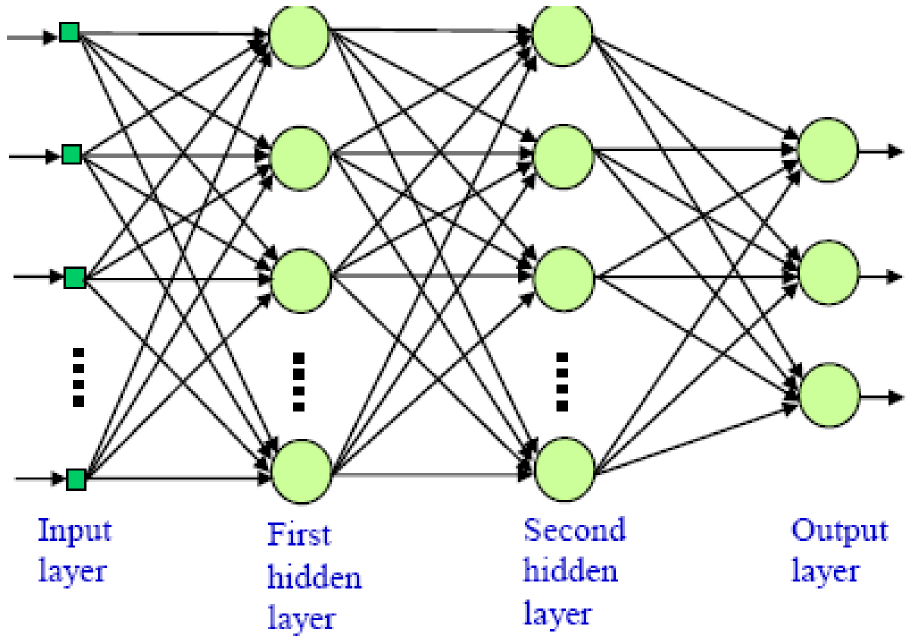
Three. The minimal structure is indeed 2-3-1 if the activation function of the output neuron is logsig. The sigmoid function has the property of flattening things outside its focused domain, which is particularly suitable for Gaussian hill problem.



What would happen if the activation function in the output neuron is linear?

The minimal structure is then 2-20-1 in order to get a good approximation.

MLP with two or more hidden layers



<u>In the universal approximation theorem discussed in the last lecture, how many hidden layers are needed to approximate any bounded continuous function?</u>

$$F(x_1,\dots,x_{m_0}) = \sum_{i=1}^{m_1} \alpha_i \varphi\left(\sum_{j=1}^{m_0} w_{ij}x_j + b_i\right)$$

Only one hidden layer is needed.

<u>Is there any advantage in adding more layers?</u>

One or two hidden Layers ?

One dimensional example:

MLPs	Total Parameters
1-9-1	28
1-3-3-1	22

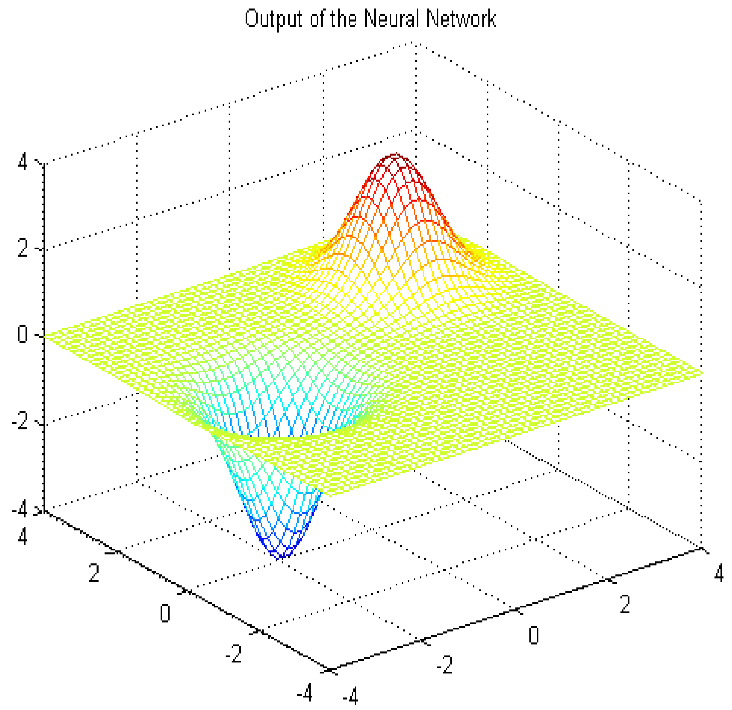
What is the advantage of introducing more hidden layers?

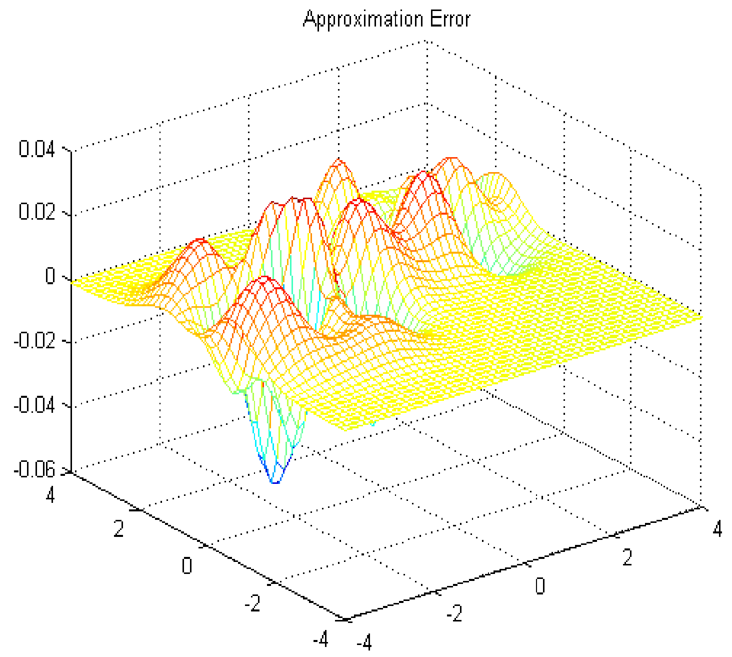
The total number of free parameters can be slightly decreased!

However, adding more layers may make the MLP more prone to local minima traps because of its more complicate structure.

The Famous Hill and Valley Example

$$f(x,y) = 3\,e^{-((x-2)^2+(y-2)^2)} - 4\,e^{-((x+2)^2+y^2)}, \quad x, y \in [-4, 4]$$





MLP with one hidden layer: 2-30-1 $\Rightarrow$ # of weights: 3*30+31=121

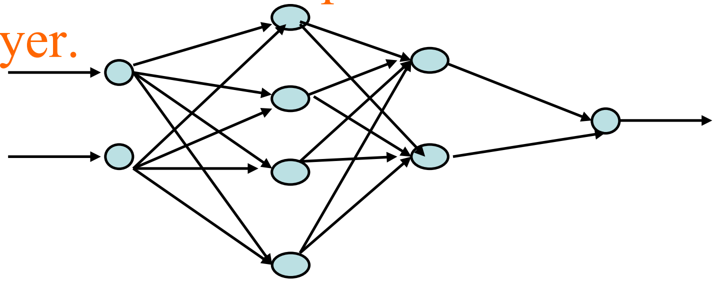
MLP with two hidden layers: 2-?-2-1 $\Rightarrow$ # of weights: 3*4+5*2+3=25

2-4-2-1

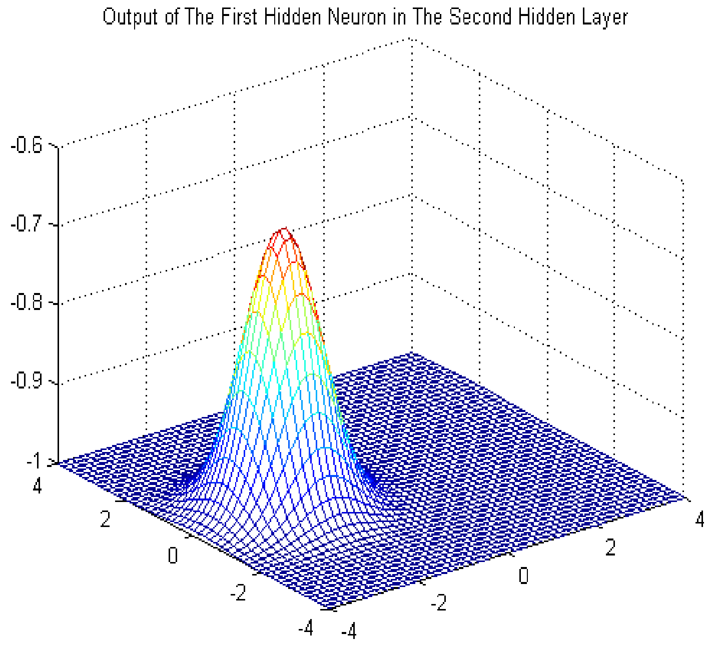
Why does it make such a big difference?

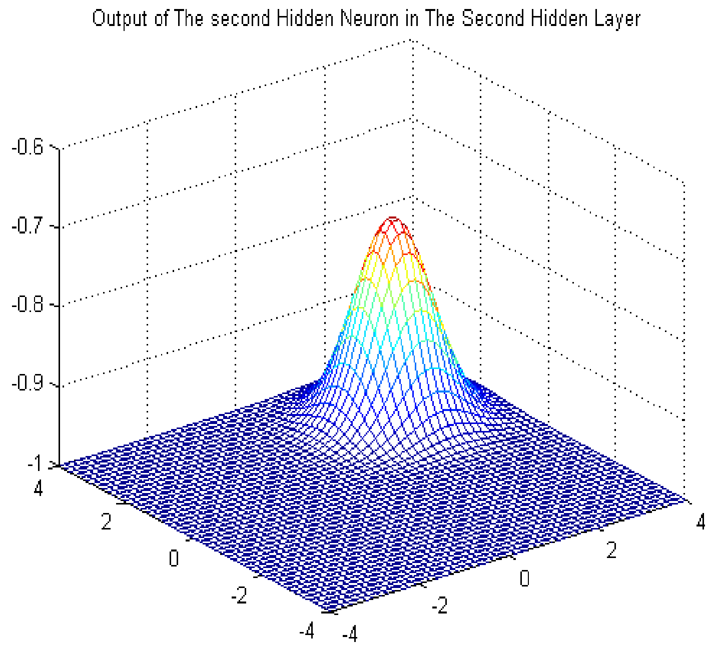
Why do we use two neurons at the second hidden layer?

The answer will be clear if we plot out the outputs of the hidden neurons in the second hidden layer.



<u>What would they look like?</u>



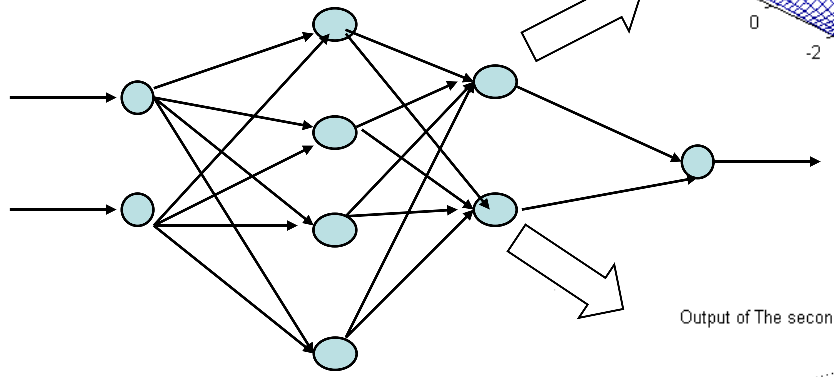


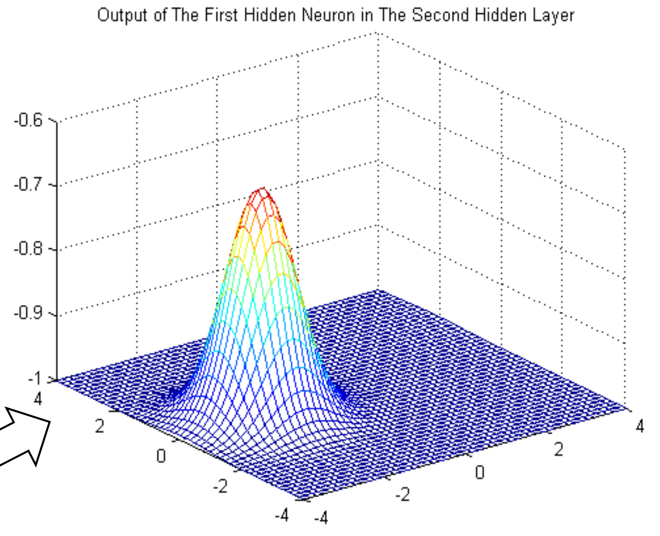
The outputs of the hidden neurons correspond to two Gaussian hills.

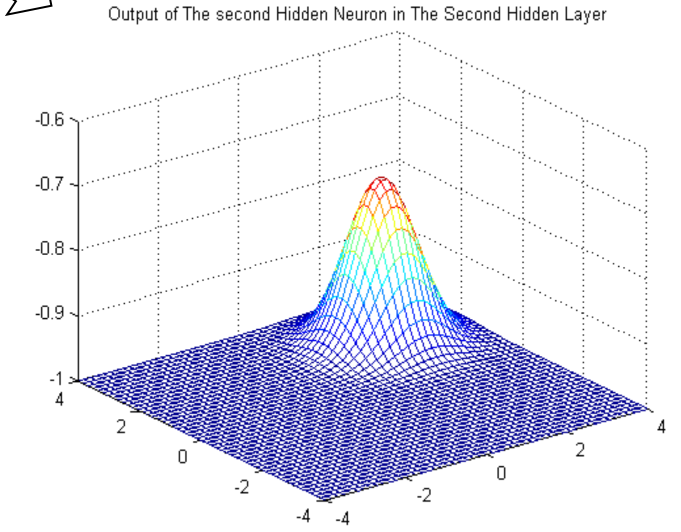
<u>Is it possible to combine two Gaussian hills into a hill and valley?</u>

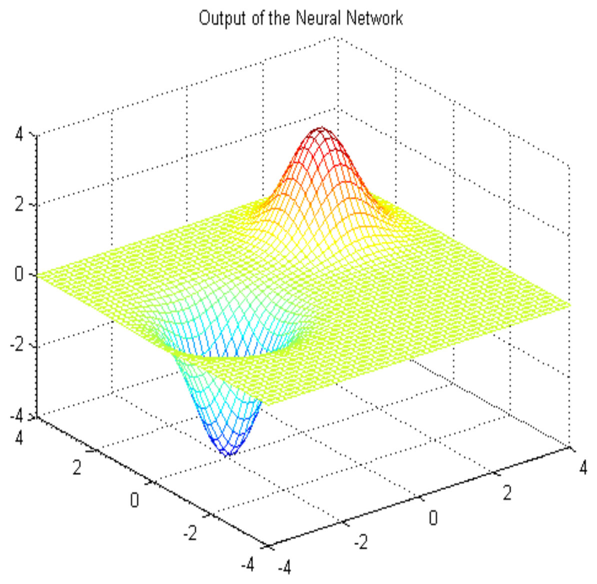
It can be done by very simple linear combination, which corresponds to the output of the final output neuron!

2-4-2-1



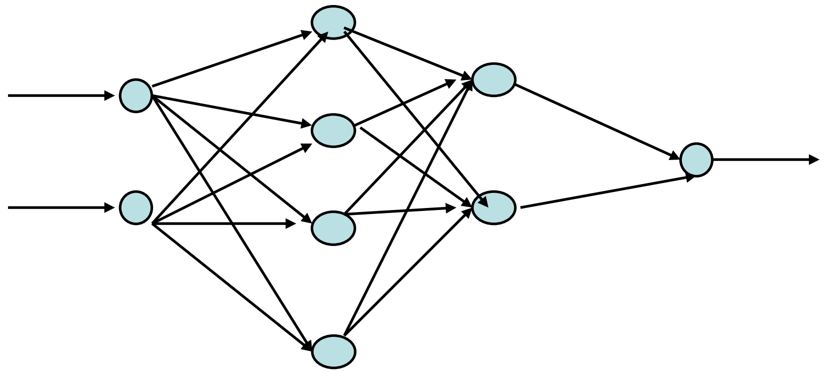

Output of The First Hidden Neuron in The Second Hidden Layer
-0.6
-0.7
-0.8
-0.9
-1
4
2
0
-2
-4
-4
-2
0
2
4





The task of approximating hill and valley is decomposed into two simpler tasks!

MLP with two hidden layers can be simply considered as linear combinations of multiple MLPs with one hidden layer.



The number of the neurons in the second hidden layer corresponds to the number of the sub-functions that would combine into the target function.

One or Two Hidden Layers?

If the target function can be easily decomposed into simple sub-functions, then it is better to use two hidden layers.

Summary

How many neurons ?

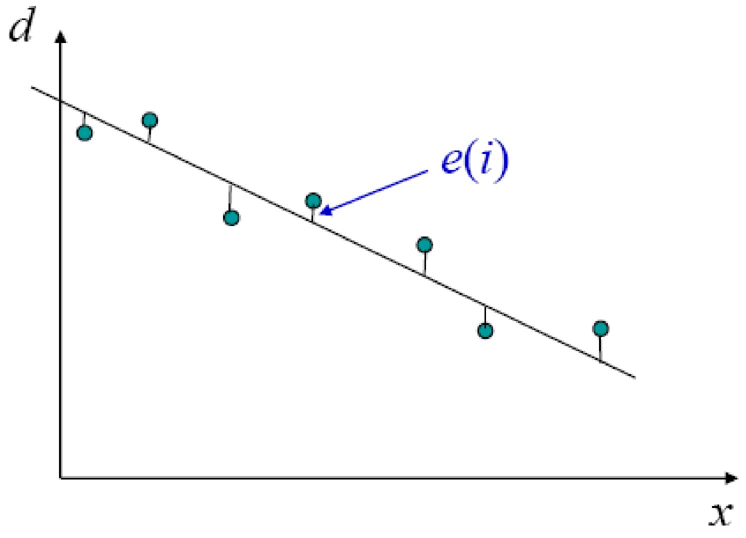
Guideline: Estimate the minimal number of line segments (or hyperplanes in high dimensional cases) that can construct the basic geometrical shape of the target function, and use this number as the first trial for the number of hidden neurons.

How many layers ?

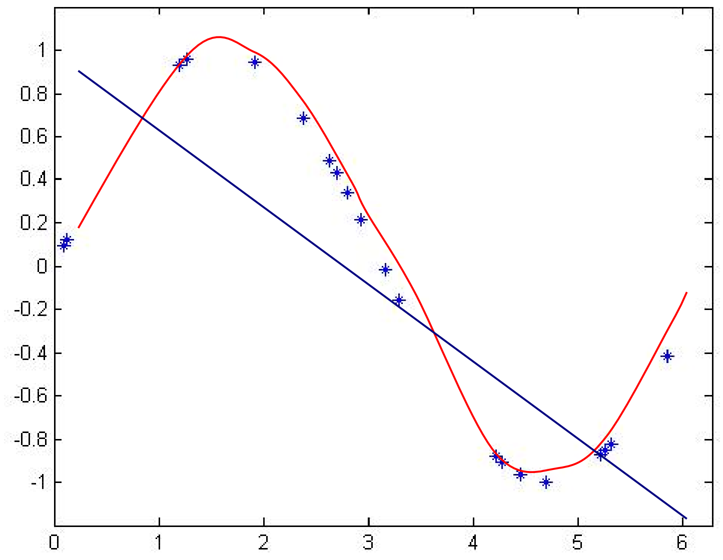
In many applications, one hidden layer proves sufficient. If the target function can be decomposed into simple sub-functions, then it is better to use two hidden layers.

In the next, we will see how we can apply this guideline to deal with a very important issue in regression problem.

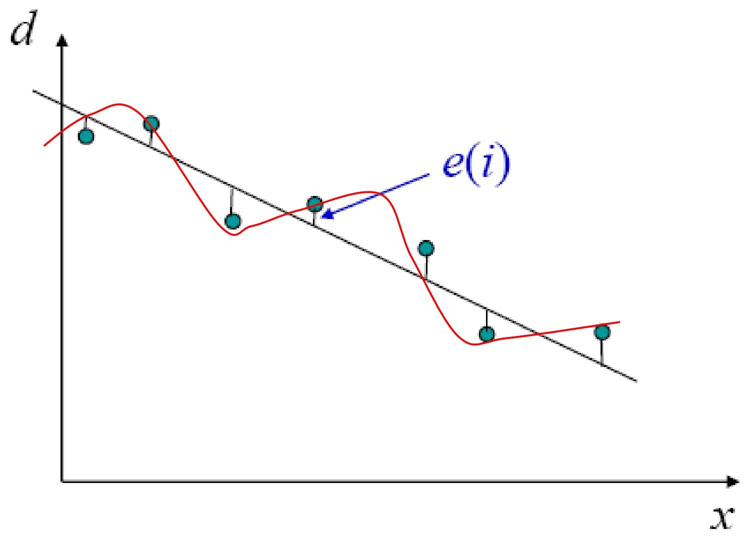
Single Layer Perceptron v.s. Multi-layer Perceptrons

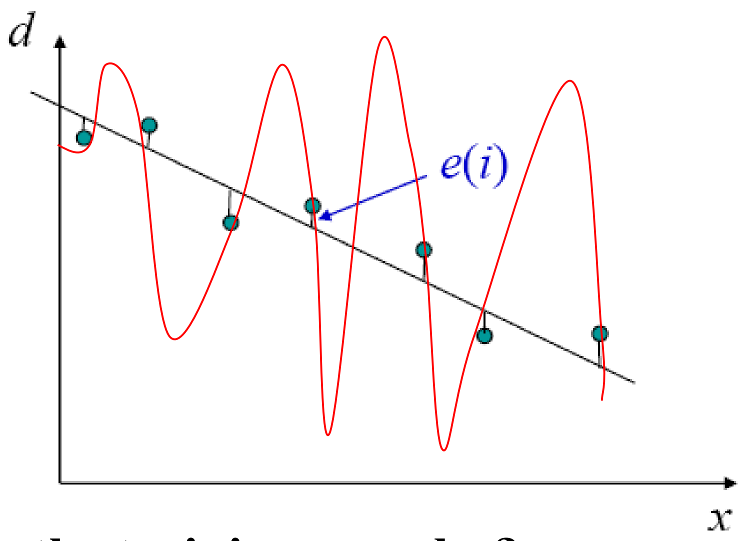


Underfitting
with single layer



What would happen if we apply MLP to the example in the left ?





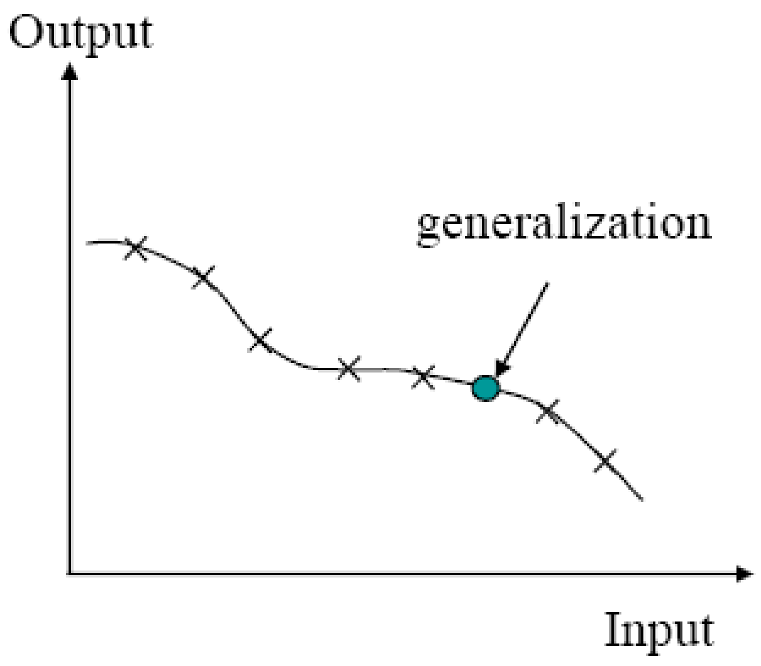
Which one is the best in terms of the errors for the training samples?
Which one is the worst in terms of the errors for the training samples?
Which one is the best fitting?

The one whose training error is the largest!

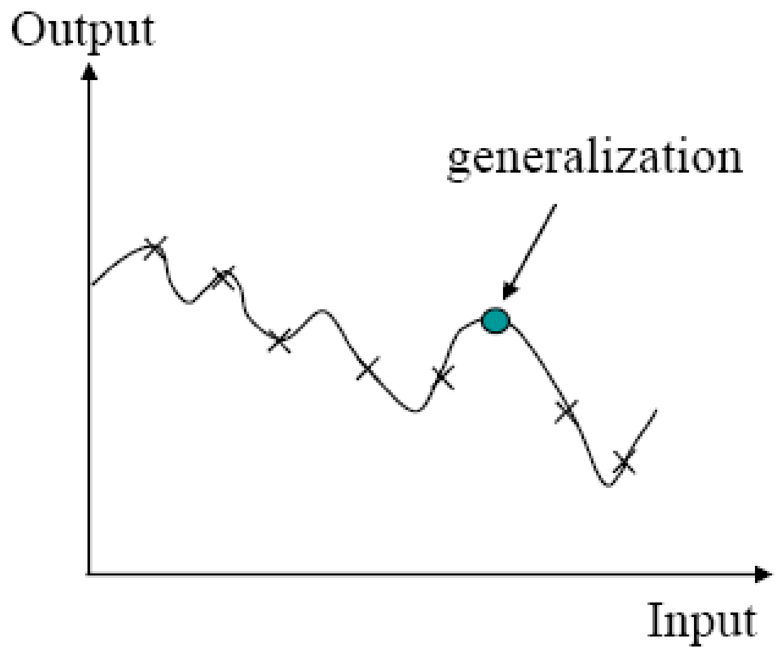
Generalization

A network is said to generalize well when the input-output mapping computed by the network is correct (or near so) for test data never used in creating or training the network.

X – Training data



Good generalization – properly fitted data



Poor generalization – overfitted data

What are the factors that can influence generalization?

- Size and accuracy of training set;
- Architecture of neural network---number of free parameters.
- Training process;

In practice, for good generalization, we could have the size of the training set, N, satisfy the empirical condition,

$$N = O\left(\frac{W}{\varepsilon}\right)$$

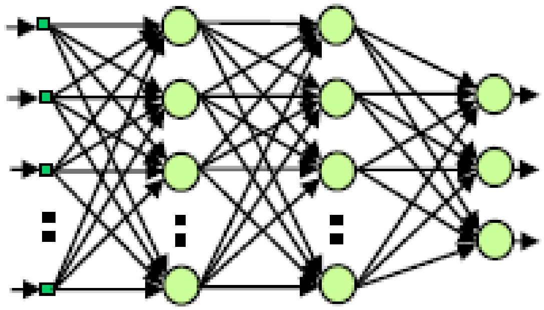
where W is the total number of free parameters (i.e., synaptic weights and biases) in the networks, and ε denotes the fraction of classification errors permitted on test data, and $O(\cdot)$ denotes the order of quantity enclosed within. E.g., with an error of 10% the number of training examples needed should be about 10 times the number of free parameters in the network.

In many applications, the number of samples is not enough to satisfy above condition.

What are the possible ways to overcome the over-fitting problem?

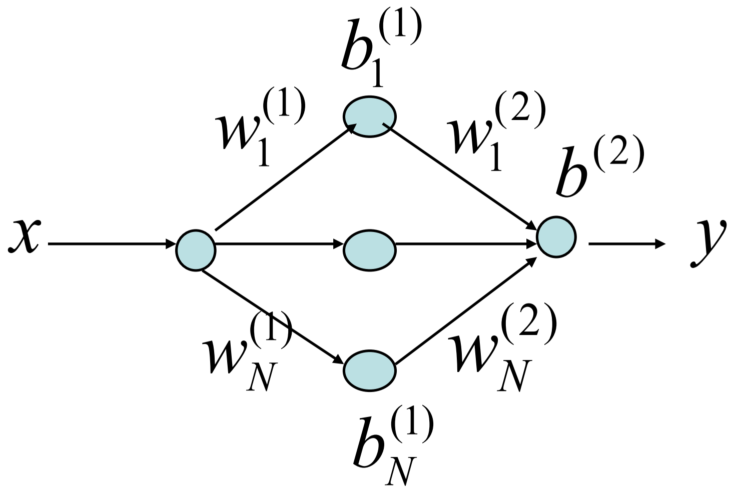
One way is to limit the number of hidden units.

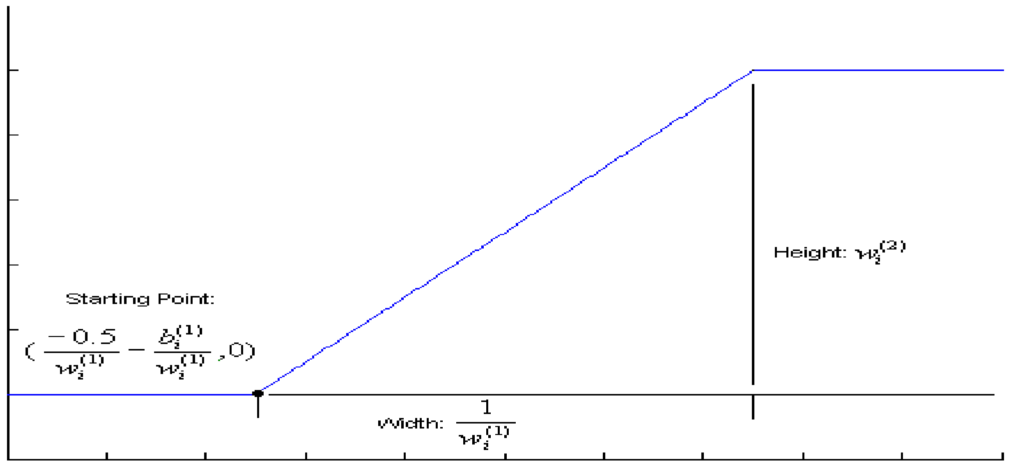
It is important to seek the minimal structure of the MLP!



The Geometrical Interpretation

$$y(x) = \sum_{i=1}^{N} w_i^{(2)} \varphi(w_i^{(1)} x + b_i^{(1)}) + b^{(2)}$$





Neurons		$\Rightarrow$	**Building Blocks**
Weights	$w_i^{(1)}$	$\Rightarrow$	**Width**
Weights	$w_i^{(2)}$	$\Rightarrow$	**Height**
Bias	$b_i^{(1)}$	$\Rightarrow$	**Position**
Bias	$b^{(2)}$	$\Rightarrow$	**Offset**
$w_i^{(1)} w_i^{(2)}$		$\Rightarrow$	**Slope**

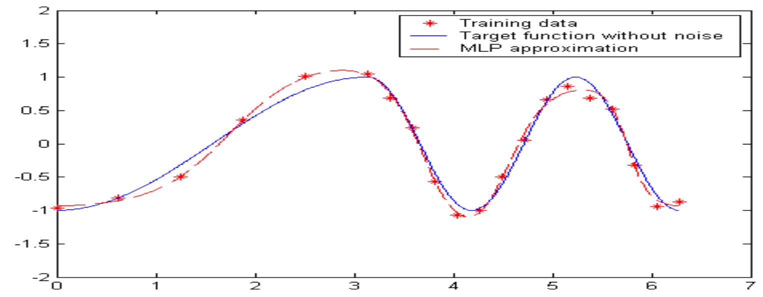
Design Guideline

Estimate the minimal number of line segments (or hyperplanes in high dimensional cases) that can construct the basic geometrical shape of the target function, and use this number as the first trial for the number of hidden neurons of the MLP.

Over-fitting Example

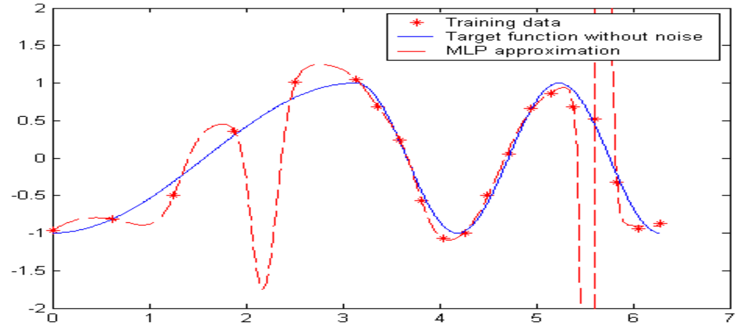
$$y = \begin{cases} -\cos(x) + v & 0 \le x < \pi \\ \cos(3(x-\pi)) + v & \pi \le x \le 2\pi \end{cases}$$

The noise *v* is uniformly distributed within [–0.25,0.25].



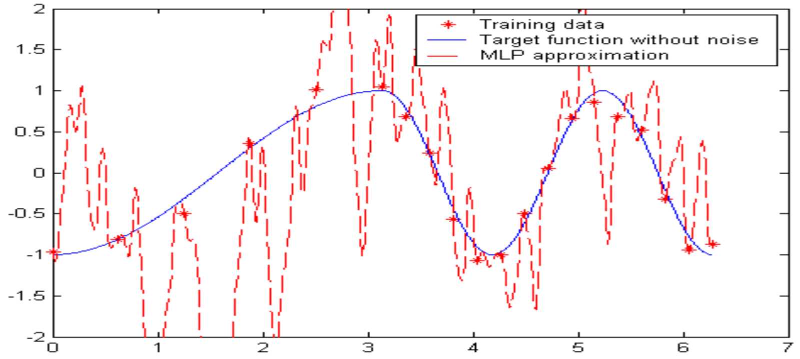
How many lines do we need to construct the basic shape?

1-4-1



What would happen if we use 1-10-1 MLP?

Over-fitting happens!



What would happen if we use 1-100-1 MLP?

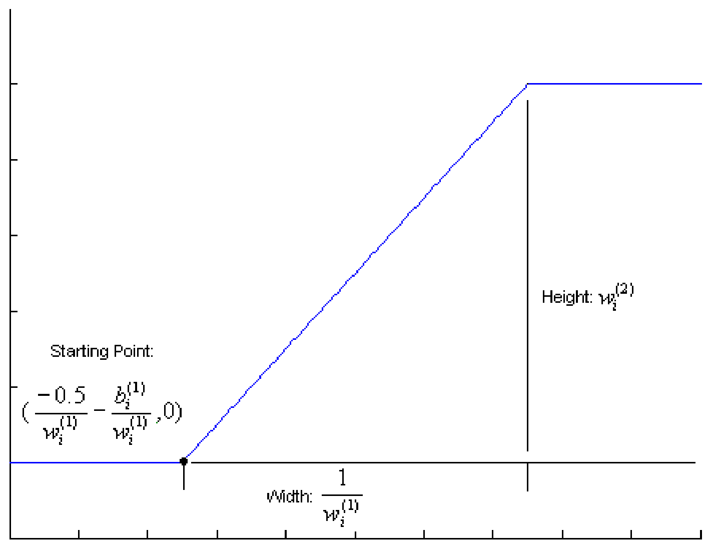
The bigger the MLP, the worse the performance.

The size of MLP is critical!

Bartlett, NIPS 1997

“For valid generalization, the size of the weights is more important than the size of the network.”

Isn’t it surprising? Why is the size of the weights important?
Now I want you to use the geometrical interpretation of the MLP to explain this!
How does the size of the weights affect the shape of the building block?



Neurons		$\Rightarrow$	**Building Blocks**
Weights	$w_i^{(1)}$	$\Rightarrow$	**Width**
Weights	$w_i^{(2)}$	$\Rightarrow$	**Height**
Bias	$b_i^{(1)}$	$\Rightarrow$	**Position**
Bias	$b^{(2)}$	$\Rightarrow$	**Offset**
$w_i^{(1)} w_i^{(2)}$		$\Rightarrow$	**Slope**

A simple explanation:

The smaller the weights, the gentler the slope of each building-block and hence the smoother the shape of the overall function.

Regularization Methods

Cost function: $F = E_D + \lambda E_w$

$\underbrace{E_D}_{\boxed{\text{training error}}}$ $\underbrace{\lambda E_w}_{\boxed{\text{cost on weights}}}$

Weight decay:

$$E_w = \sum_{i=1}^{N}[(w_i^{(1)})^2 + (w_i^{(2)})^2 + (b_i^{(1)})^2] + (b^{(2)})^2$$

Weight elimination:

A normalized version of weight decay.

The most recent one: $$E_w = \sum_{i=1}^{N}(w_i^{(2)} w_i^{(1)})^2$$

Why is this recent cost function better than the one for weight decay?

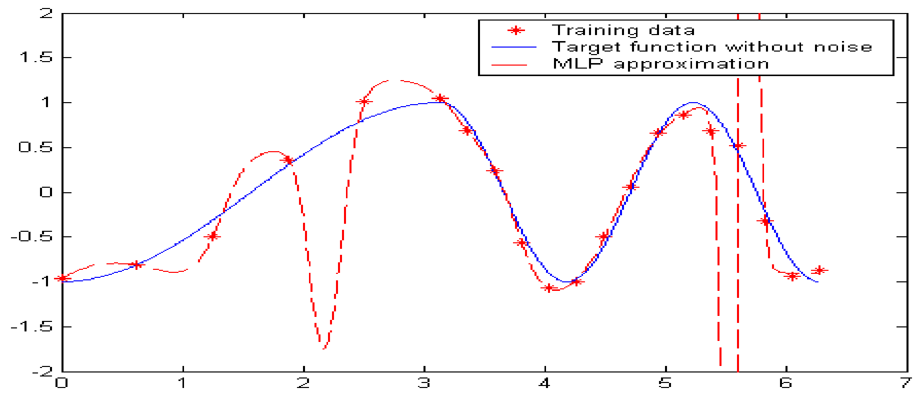
The smoothness of the building blocks are completely determined by the slopes, $w_i^{(2)} w_i^{(1)}$

It has nothing to do with the biases!

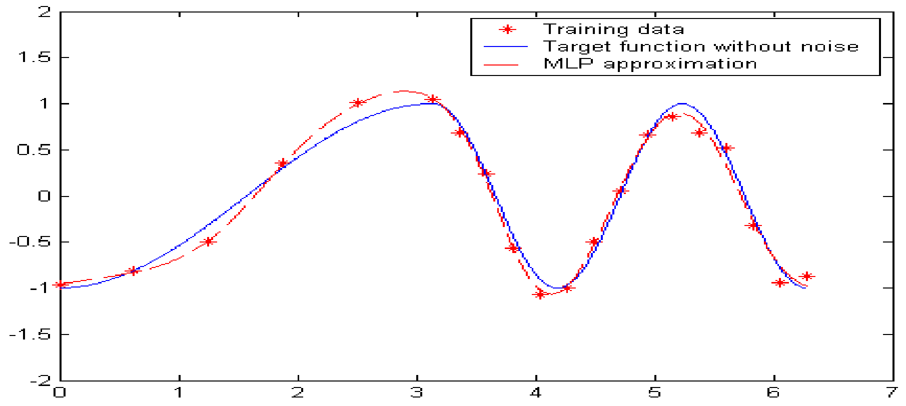
The regularization algorithm for MLP is implemented by MATLAB neural network toolbox by the training algorithm called "trainbr".

A successful example

1-10-1 trainlm
Without regularization



1-10-1 trainbr
With regularization

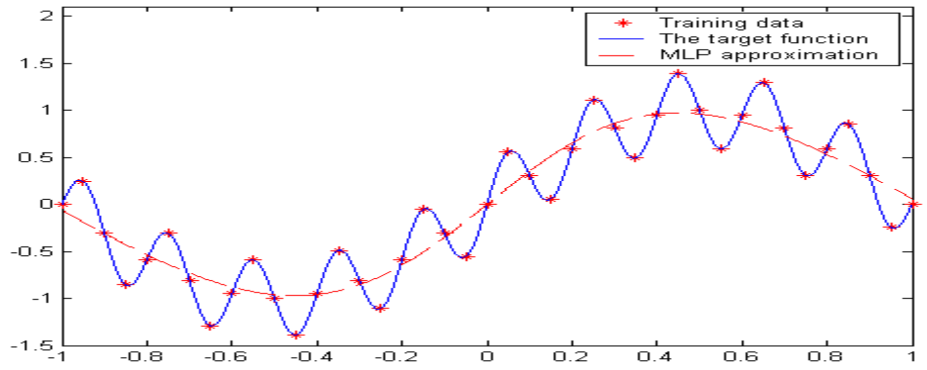


A potential problem with regularization: Is it noise or signal?

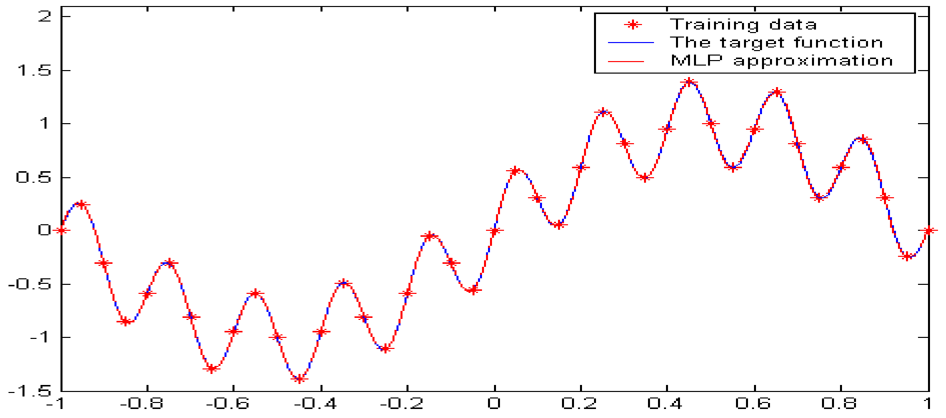
The training data are generated using the following function:

$$y = \sin(\pi x) + 0.2\sin(10\pi x)$$

1-21-1 trainbr
With regularization



1-21-1 trainlm
Without regularization



Which one is better?

Regularization algorithm essentially filters out the high frequency part of the signal.

But what if the high frequency part contains some useful information, not only noise?

In summary, how to avoid over-fitting?

1. Choose the minimal structure
2. Use regularization techniques

How to choose minimal structure?

Guideline: Estimate the minimal number of line segments (or hyperplanes in high dimensional cases) that can construct the basic geometrical shape of the target function, and use this number as the first trial for the number of hidden neurons.

Open Question:

Is it simple to estimate the number of hyper-planes in high-dimensional surface?

Very difficult!

This guideline is useless if the dimension of the input space is high!

In the following we will introduce one approach to determine the proper number of neurons for high dimensional problems.

Why are we interested in the minimal structure of MLP?

Besides the objective of avoiding over-fitting, there is another deeper reason:

Occam's Razor

A principle attributed to the English logician, William of Ockham (1288 – 1348) .

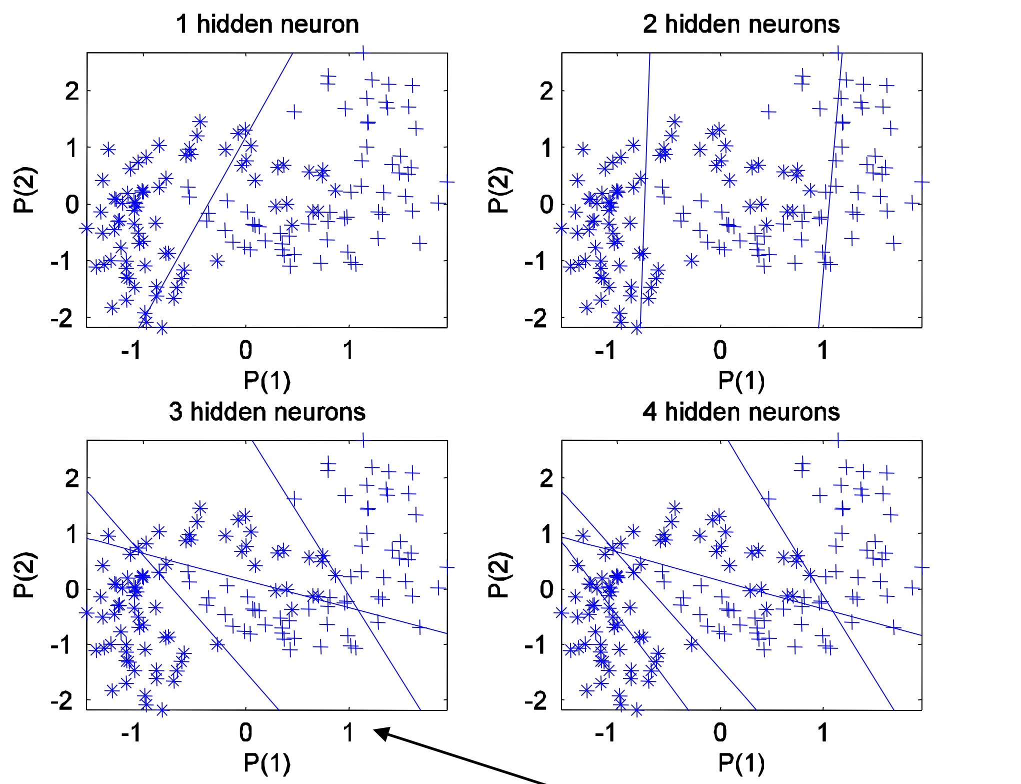
Principle of parsimony

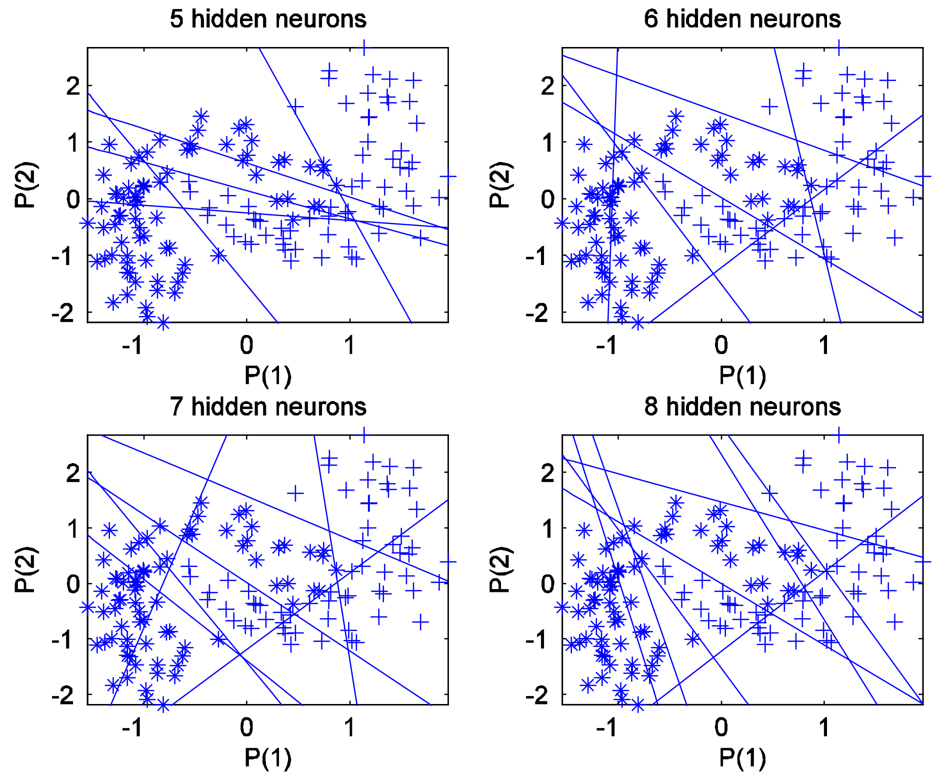
"When deciding between two models which make equivalent predictions, choose the simpler one."

Neural network

"What is the simplest or smallest neural network that can model the training data given the absence of higher-level information?"

Illustrative Example





3 Hidden Neurons

Singular Value Decomposition (SVD)

Matrix factorization

Commonly found in statistical, signal processing approaches, subspace modeling

$$\mathbf{H}_{Nxn} = \mathbf{U}_{NxN} \Sigma_{Nxn} \mathbf{V}^{T}_{nxn}$$

$$\Sigma = \begin{bmatrix} \begin{bmatrix} \sigma_1 & 0 & \cdots & 0 \\ 0 & \sigma_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \sigma_k \end{bmatrix} & 0 \\ 0 & 0 \end{bmatrix}$$

$\sigma_1 \geq \sigma_2 \geq \ldots \geq \sigma_k > 0$ are called the singular values of the matrix H_{Nxn}

U and V are orthogonal matrices:

$U^{-1} = U^T$ $V^{-1} = V^T$

$$\mathbf{H}_{Nxn} = \mathbf{U}_{NxN}\, \Sigma_{Nxn}\, \mathbf{V}^{\mathbf{T}}_{nxn}$$

$$\Sigma = \begin{bmatrix} \begin{bmatrix} \sigma_1 & 0 & \cdots & 0 \\ 0 & \sigma_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \sigma_k \end{bmatrix} & 0 \\ 0 & 0 \end{bmatrix}$$

H_{Nxn} **: *N x n* rectangular matrix**

<u>What is the rank of the matrix H?</u>

Rank(H_{Nxn}) ***=k***

Numerical rank

Number of linearly independent columns or rows

But sometimes, the numerical rank may not truly reflect the actual number of independent vectors.

Actual (Effective) rank

Effective rank indicates the actual number of independent columns and rows.

SVD can be used to determine the effective rank:
How close are the singular values to zero?
The small ones can be ignored in getting the effective rank.

Numerical Rank vs. Effective Rank

Let's compare the following two matrices

$$A = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \end{bmatrix} \quad \text{and} \quad B = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & 1 & 0.01 \end{bmatrix}$$

What is the numerical rank of A and B? 3!

There is a big difference between the linear independency of the rows of A and B?

The second and third rows of B are almost linearly dependent! The third row is almost redundant!

Numerical rank does not provide insight into structure and linear independency.

Let's see if the SVD can reveal this special property of B:

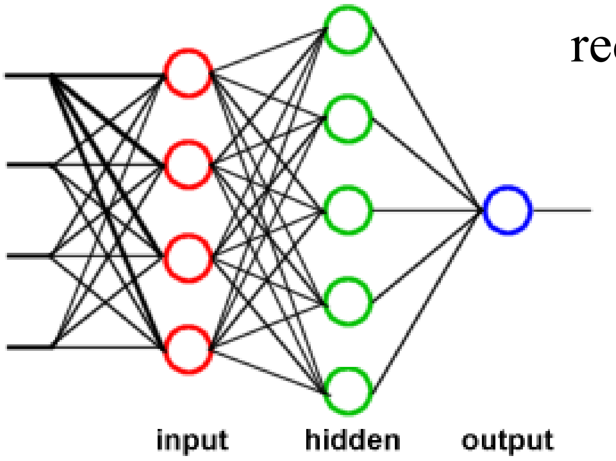
Use MATLAB code svd(A) and svd(B) will result in

svd(A)=[2.247 0.8019 0.5550] svd(B)=[2.1358 0.6622 0.0071]

Effective rank of A: 3 Effective rank of B: 2

Effective Rank of Hidden Neurons

There are n hidden neurons in the network. Some of them may be redundant. We can use the training data to find out.



***Training Set: N* training samples.**

***Each sample is a M-dimensional vector→M* input variables**

Let's calculate the outputs of the hidden neurons for each training sample input vector x(k):

$$h_{ki} = f_i\left(\sum_{j=1}^{M} w_{ij} x_j(k) + b_i\right) \in H$$

<u>**What is the dimension of the matrix H?**</u>

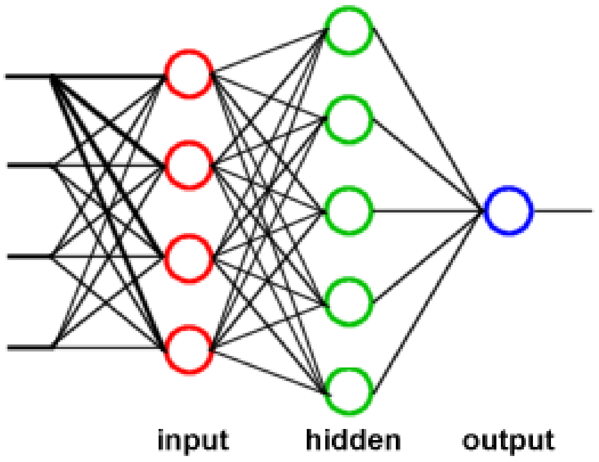
N* *training examples and* *n **hidden neurons** **--→** H_{Nxn}

H_{Nxn} is a matrix describing the outputs of the hidden neurons for all the training samples.

Each row vector -→ the outputs of the hidden neurons for individual training samples.

If there are too many hidden neurons, then there must be some redundant information in the matrix H, i.e. some columns are almost linearly dependent!

Effective Rank of Hidden Neurons



***Training Set: N* training samples.**

***Each sample is a M-dimensional vector→M* input variables**

The outputs of the hidden neurons for each training sample input vector x(k):

$$h_{ki} = f_i(\sum_{j=1}^{M} w_{ij} x_j(k) + b_i) \in H$$

H_{Nxn} is a matrix describing the outputs of the hidden neurons for all the training samples.

How to determine whether the outputs of some neurons in H are redundant or not?

Use SVD to get the effective rank of H!

How small is "small"? What is the cutoff value?

Subjective, problem-dependent

Need a criteria, threshold to define *'small'* values

Pruning criterion

The singular values $\sigma_1 \geq \sigma_2 \geq \ldots \geq \sigma_n \geq 0$

Quantify relevant hidden layer neurons

Relative, not ***absolute*** magnitudes of singular values are important

<u>There are many possibilities. Can you suggest one?</u>

Normalize the values such that the total is one.

The most common choices are the following two:

$$(1) \quad \min_k \{ (\textstyle\sum_{i=1}^{k} \sigma_i / \sum_{i=1}^{n} \sigma_i) \geq \gamma \}$$

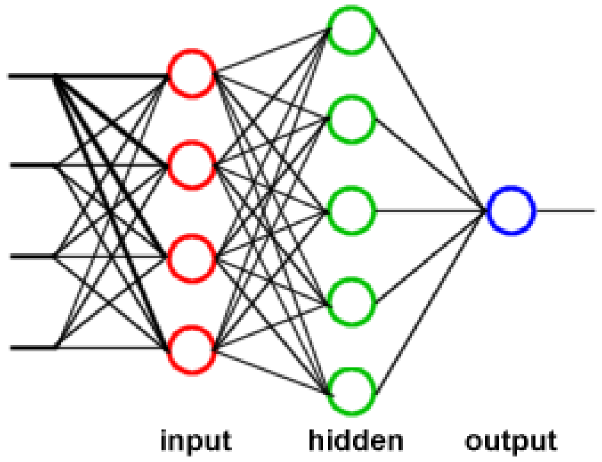
$$(2) \quad \min_k \{ (\textstyle\sum_{i=1}^{k} \sigma_i^2 / \sum_{i=1}^{n} \sigma_i^2) \geq \gamma \}$$

The threshold γ is usually chosen as 95% or 99%.

Algorithm

Train the MLP on the *N* training samples (X_{NxM}) with sufficiently large number of *n* hidden neurons until convergence.

Calculate the outputs of the hidden neurons for all the samples:



$$h_{ki} = f_i\left(\sum_{j=1}^{M} w_{ij} x_j(k) + b_i\right) \in H$$

Perform the SVD on the hidden layer activations, H_{Nxn},

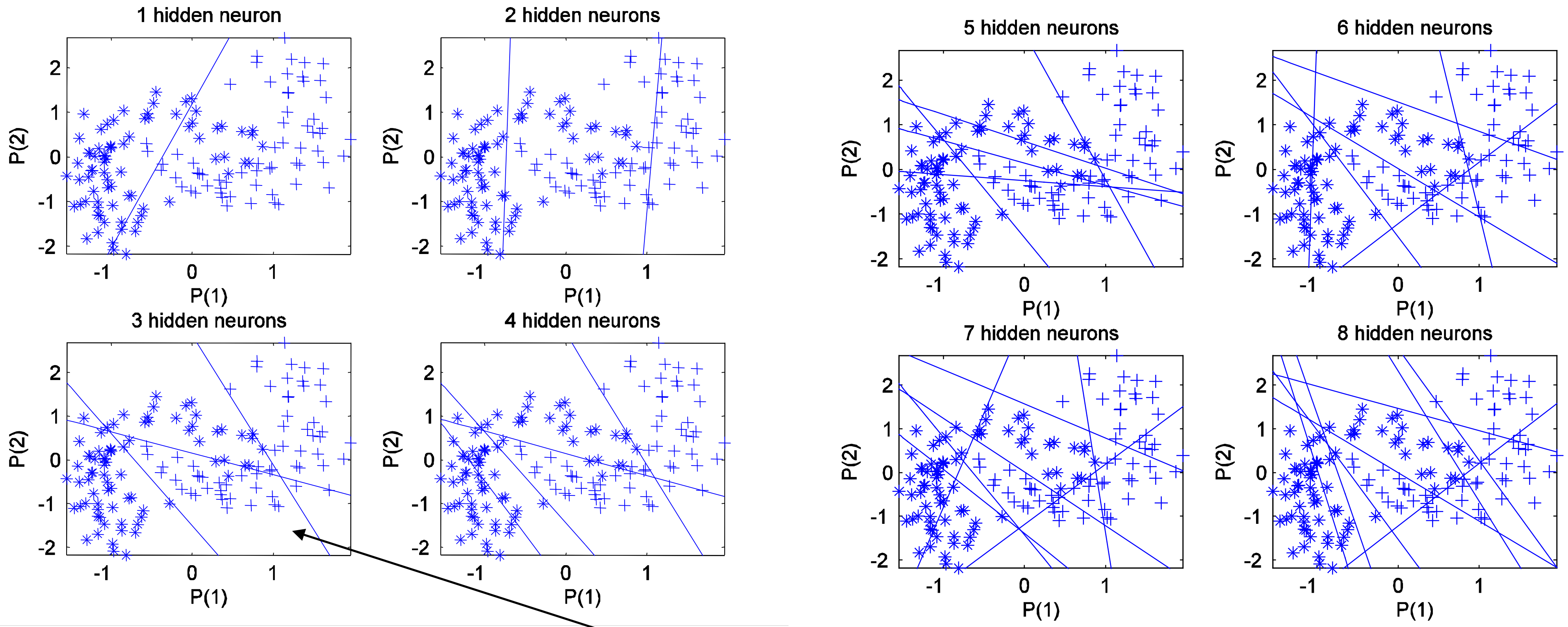
MATLAB: svd(H) $\Longrightarrow$ The singular values $\sigma_1 \geq \sigma_2 \geq \ldots \geq \sigma_n \geq 0$

Select a suitable *pruning criterion* and appropriate *threshold* to obtain *k*

Retrain the new (reduced) network with *k* hidden neurons ($k \leq n$)

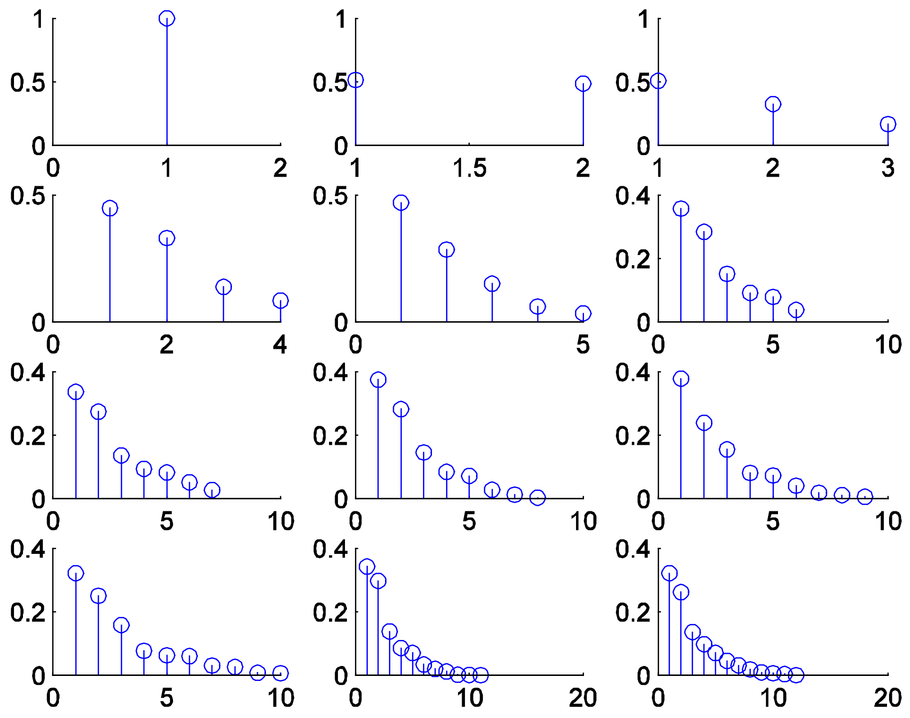
For complex problems, you may need to repeat this procedure until convergence.

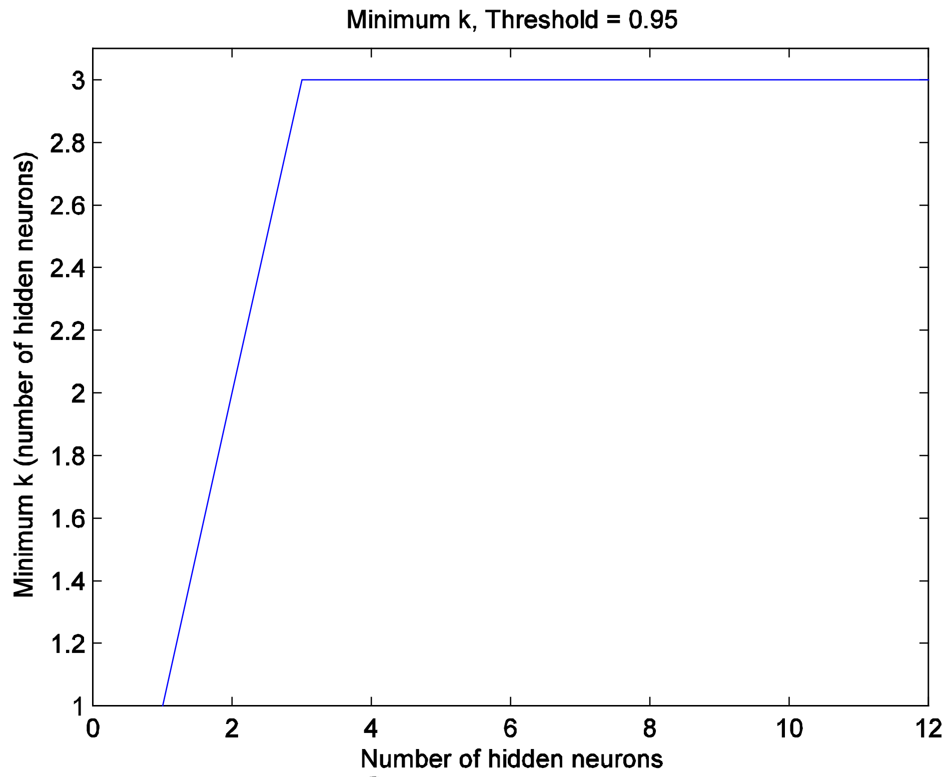
Illustrative Example



3 Hidden Neurons

Illustrative Example (1b)





$$\min_k\{\left(\sum_{i=1}^{k}\sigma_i^2 / \sum_{i=1}^{n}\sigma_i^2\right) \geq \gamma\}$$

Vary 1-12 Hidden Neurons

Threshold = 95% of total spectral energy of singular values

Summary

SVD for estimating number of hidden neurons

Singular values act as an estimate of linear independency of the outputs of the hidden neurons.

We need to find an appropriate criteria & threshold, which is problem dependent.

It is simple and well-grounded in mathematical analysis.

How to design and train the neural network?

How many hidden layers?

Normally one hidden layer is enough. Two hidden layers may be better if the target function can be clearly decomposed into sub-functions.

How many hidden neurons?

If the geometrical shape can be perceived, then use the minimal number of line segments (or hyper-planes) as the starting point.

For higher dimensional problem, start with a large network, then use SVD to determine the effective number of hidden neurons.

How to choose activation function in the hidden layers?

Hyperbolic tangent (tansig) is the preferred one for all the hidden neurons.

How to choose activation functions in the output neurons?

Logsig for pattern recogntion, purelin for regression problem.

How to design and train the neural network?

How to pre-process the input data?

Normalize all the input variables to the same range such that the mean is zero.

When to use sequential learning? And when to use batch learning ?

Batch learning with second order algorithm (such as trainlm in MATLAB) is usually faster, but prone to local minima.

Sequential learning can produce better solution, in particular for large database with lots of redundant samples. But it is slow.

How to deal with over-fitting?

You either identify the minimal structure, or use regularization (trainbr in MATLAB).

One application using MLP: Face Recognition

How do you code the face image?

mxn pixels in the image: a mxn matrix

You need to concatenate it into a vector!

Do you need to normalize the inputs?

Normalization will help!

What are the output units?

The number of outputs correspond to the number of classes.

Identity recognition: n (number of persons)

Expression recognition: 6 (basic expressions)

Glasses-Wearing recognition: With/without Glasses 2

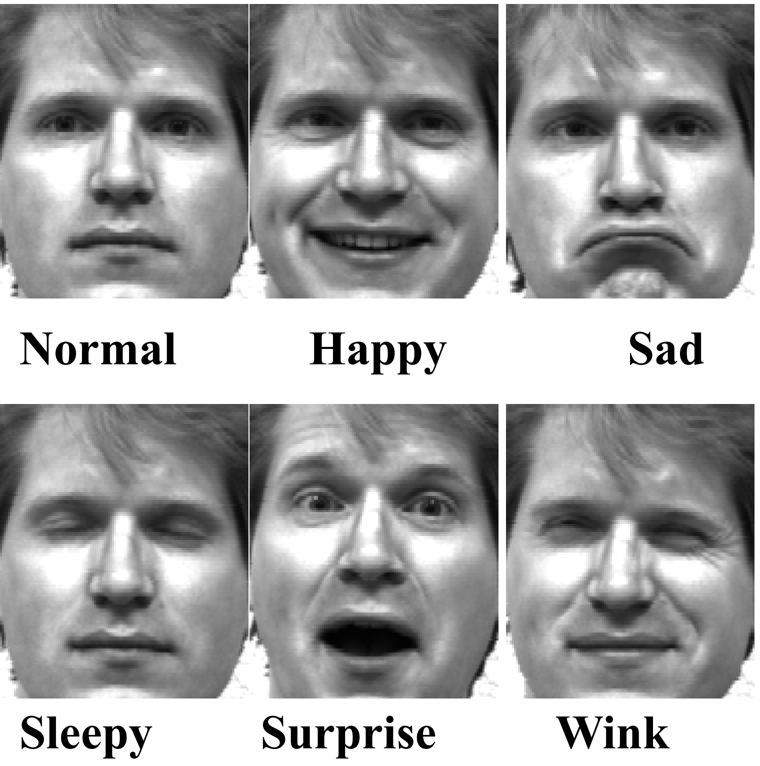
Can you use only one output unit for two-class problem?

How to choose the activation function for output units?

Logistic function: Logsig

How many hidden neurons?

How many hidden layers?



National University of Singapore

Q & A…

THANK YOU !