UF UNIVERSITY of FLORIDA
The Foundation for The Gator Nation

Neural Network Model

Artificial neural network

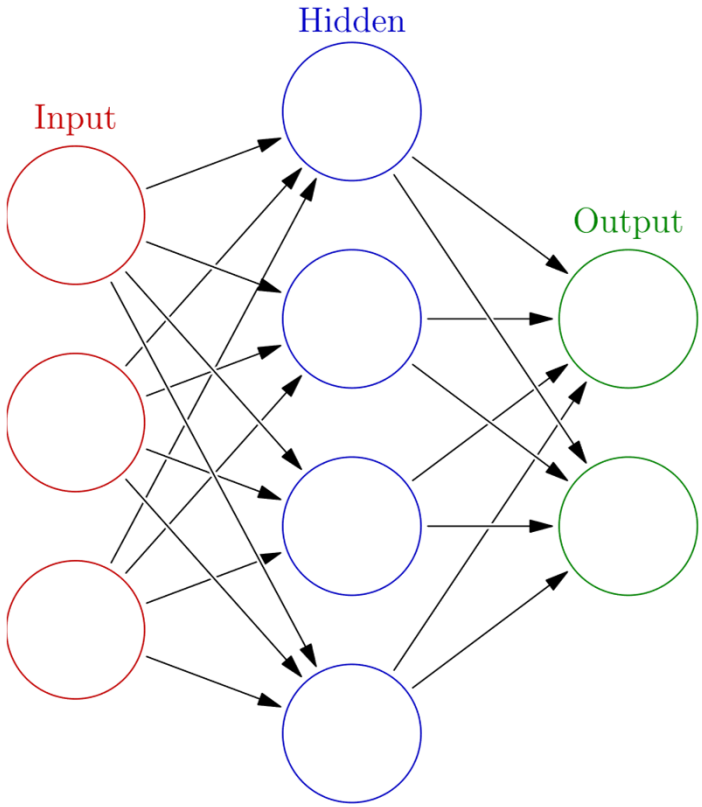
- Computing systems vaguely inspired by the biological neural networks that constitute animal brains
  - The data structures and functionality of neural nets are designed to simulate associative memory
  - Learn by processing input and output data (samples)
  - After learn by enough data, neural nets can predict results from inputs
- Relationship between the number and diversity of samples processed by a neural net and the prediction accuracy
- Fancy description, but basically neural network is a kind of surrogate model
  - Determine (train) unknown parameters using samples and predict

Artificial neural network *cont.*

- based on a collection of connected units or nodes called artificial neurons
  - loosely model the neurons in a biological brain
  - Each connection can transmit a signal to other neurons
  - A neuron receives a signal then processes it and sends the signal to the neurons connected to it
- Implementation
  - "signal" at a connection is a real number, and the output of each neuron is computed by a nonlinear function of the sum of its inputs
  - Neurons typically have a weight that adjusts as learning proceeds
  - Input layer (inputs), hidden layer (not shown), and output layer (prediction)

Deep learning

- part of a broader family of machine learning methods based on artificial neural networks with representation learning
  - Learning can be supervised, semi-supervised or unsupervised
- The adjective "deep" comes from the use of multiple layers in the network
  - Deep learning is a modern variation which is concerned with an unbounded number of layers of bounded size
  - permits practical application and optimized implementation



History of neural network

- first introduced in 1943 by Warren McCulloch, a neurophysiologist, and a mathematician Walter Pitts
- In 1949, Donald Hebb published a book, “The Organization of Behavior,” where he formulated the classical Hebbian rule, which is proved to be the basis of almost all neural learning procedures
- John Hopfield invented associative neural network in 1982, which is now more commonly known as Hopfield Network
- backpropagation algorithm, a generalization of Widrow-Hoff learning algorithm, was invented by Rumelhart, Hinton and Williams → renaissance of research in neural network

Types of NN

- Feedforward NN (Svozil et al. 1997): conveys the information in one direction
- Radial basis function network (Orr 1996): uses the Gaussian radial basis function
- Recurrent NN (Bodén 2001): local feedback connections between the input and outputs, and so forth
- Fuzzy-neural (Liu and Li 2004), wavelet (He et al. 2004), associative-memory (Bicciato et al. 2001), modular (Happel and Murre 1994) and hybrid (Zhang and Yuen 2013, Rovithakis et al. 2004) neural networks

Feedforward NN model

- Neurons or nodes: basic processing units in NN
  - they can be grouped into different layers such as input layer, output layer and hidden layer
- Input layer: receiving input data from outside
- Output layer: send processed data out of the NN (prediction)
- Hidden layer: between input and output layers, no interaction with outside (not shown to users)
- Feedforward NN
  - information only moves in forward direction from input layer, through hidden layer and to output layer
  - no feedback or flow within layers

Illustration of feedforward NN model

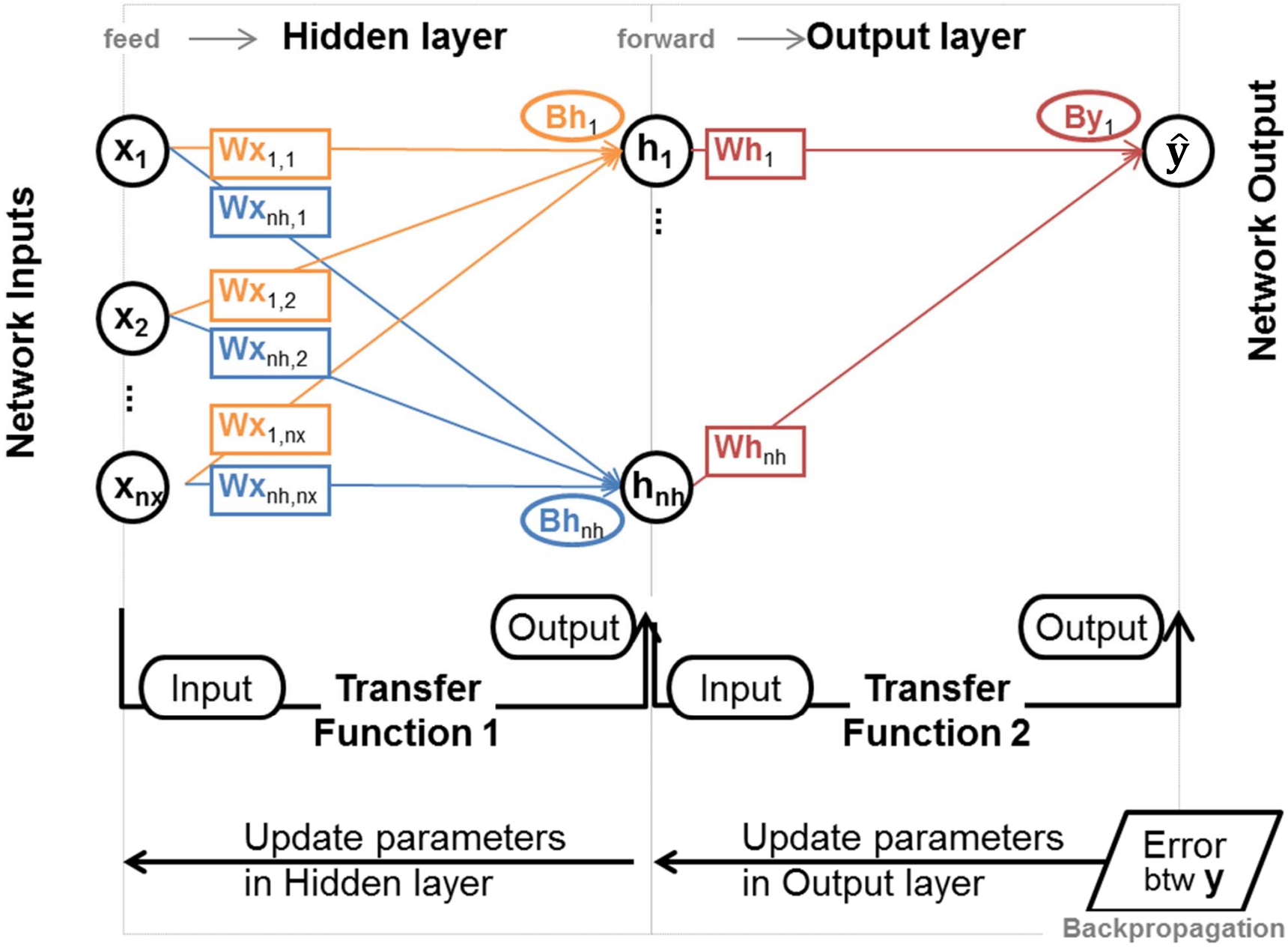


Illustration of feedforward NN model

- Feedforward: information flows from input layer to hidden layer to output layer
- Parameters
  - Weight: interconnection parameter between different nodes, w_x, w_h
  - Bias: threshold value added after weighted sum of inputs from previous layer, b_h, b_z
- Transfer (activation) function
  - Integrate the linear combination of information from the previous layers using weights and bias

Training process

- finding the optimal parameters, weights and biases, such that the network model accurately represents the relationship between inputs and outputs
- backpropagation: propagates the error between training data and the network outputs backward
  - optimization process to find the best weights and biases to minimize the mean square error between network predictions and training data
  - parameters (gradient or Jacobian) for hidden weights and output bias (output layer) are updated first
  - then, parameters for input weights and hidden biases (hidden layer) are updated next based on the updated parameters in the output layer

Feedforward mechanism

- Relationship between input nodes $\mathbf{x}$ and hidden nodes $\mathbf{h}$:

$$\mathbf{h} = d_h\left(\mathbf{W}_x\mathbf{x} + \mathbf{b}_h\right)$$

  - $\mathbf{W}_x$: $(n_h \times n_x)$ input weight matrix
  - $\mathbf{b}_h$: $(n_h \times 1)$ hidden bias vector
  - $d_h(\,)$:transfer function in the hidden layer

- Relationship between hidden nodes $\mathbf{h}$ and output nodes $\hat{\mathbf{y}}$:

$$\hat{\mathbf{y}} = d_z\left(\mathbf{W}_h\mathbf{h} + \mathbf{b}_z\right)$$

  - $\mathbf{W}_h$: $(n_z \times n_h)$ hidden weight matrix
  - $\mathbf{b}_z$: $(n_z \times 1)$ output bias vector
  - $d_z(\,)$: transfer function in the output layer

We will consider $n_z = 1$ case only

Training process

- First, the network structure must be decided
  - The number of input nodes, hidden nodes, and output nodes
- Training process determines all weights, $\mathbf{W}_x$ and $\mathbf{W}_h$, and all biases, $\mathbf{b}_h$ and $\mathbf{b}_z$
- Training data: $(\mathbf{x}_i, y_i), i = 1, \cdots, n_y$
- Minimize the error between $\hat{y}(\mathbf{x}_i)$ and y_i
  - This is similar to regression
  - In general, need to solve optimization problem for nonlinear transfer functions

Training process *cont.*

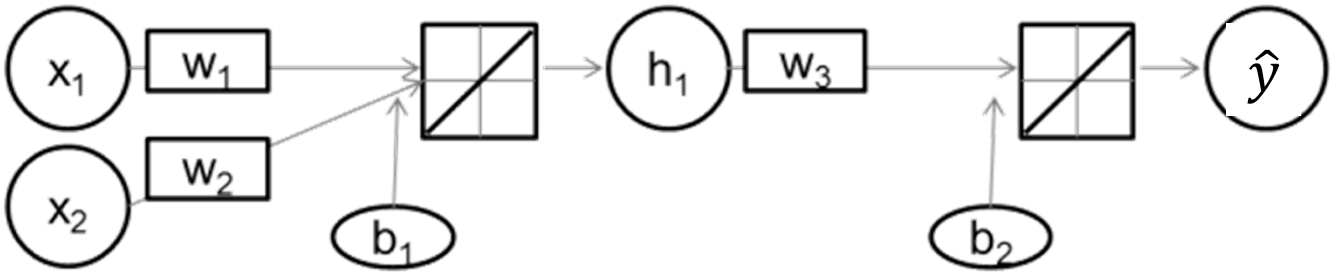
- Apply training data to $\mathbf{h} = d_h\left(\mathbf{W}_x\mathbf{x} + \mathbf{b}_h\right)$ $\hat{y} = d_z\left(\mathbf{W}_h\mathbf{h} + b_z\right)$

$$\mathbf{H} = d_h\left(\mathbf{W}_x\mathbf{X} + \mathbf{B}_h\right),\ \hat{\mathbf{y}} = d_z\left(\mathbf{w}_h\mathbf{H} + \mathbf{b}_z\right)$$

  - $\mathbf{X}\ (n_x \times n_y)$: matrix of training input data
  - $\mathbf{B}_h\ (n_h \times n_y)$: hidden bias matrix
  - $\mathbf{H}\ (n_h \times n_y)$: matrix of hidden layer outputs
  - $\hat{\mathbf{y}}\ (1 \times n_y)$: vector of network simulation outputs
  - $\mathbf{w}_h\ (1 \times n_h)$: hidden weight row vector
  - $\mathbf{b}_z\ (1 \times n_y)$: output bias vector

Ex) FFNN with linear transfer function

- Two input nodes, one hidden node, linear transfer function $d_h(x) = x$ and $d_z(x) = x$ $(n_x = 2, n_h = 1, n_z = 1)$



$$\mathbf{W}_x = \begin{bmatrix} w_1 & w_2 \end{bmatrix},\ \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix},\ \mathbf{b}_h = b_1,\ \mathbf{W}_h = w_3,\ \mathbf{b}_z = b_2$$

- Hidden layer $\mathbf{h} = \mathbf{W}_x\mathbf{x} + \mathbf{b}_h = w_1x_1 + w_2x_2 + b_1$
- Output layer $\hat{\mathbf{y}} = \mathbf{W}_h\mathbf{h} + \mathbf{b}_z$

$$= w_3\left(w_1x_1 + w_2x_2 + b_1\right) + b_2$$

$$= \left(w_1w_3x_1 + w_2w_3x_2\right) + w_3b_1 + b_2$$

- Equivalent one-layer network model

$$\hat{\mathbf{y}} = w_1^*x_1 + w_2^*x_2 + b_1^*,\ w_1w_3 = w_1^*,\ w_2w_3 = w_2^*,\ w_3b_1 + b_2 = b_1^*$$

Training process with pure linear transfer function

- When $d_h(x) = x$ and $d_z(x) = x$

$$\hat{\mathbf{y}} = \mathbf{w}_h(\mathbf{W}_x\mathbf{X} + \mathbf{B}_h) + \mathbf{b}_z = (\mathbf{w}_h\mathbf{W}_x)\mathbf{X} + (\mathbf{w}_h\mathbf{B}_h + \mathbf{b}_z)$$

  - One layer NN model with $(\mathbf{w}_h\mathbf{W}_x)_{1\times n_x}$ being input weights and $(\mathbf{w_h}\mathbf{B}_h + \mathbf{b}_z)_{1\times n_y}$ being the output bias

  - Multi-layer NN model with pure linear transfer function can be converted into a single-layer NN model

Input data in time-domain NN

- We consider a dynamic response of a system. Input = time, output = QoI
- Data
  - $\mathbf{t} = \{t_1, t_2, \cdots, t_{n_y}, t_{n_y+1}, t_{n_y+2}\}$
  - $\mathbf{y} = \{y_1, y_2, \cdots, y_{n_y}, y_{n_y+1}, y_{n_y+2}\}$
- Use 3 previous y data as input ($n_x = 3$) for predicting $\hat{y}_k$
  - $\mathbf{x} = \{y_{k-1}, y_{k-2}, y_{k-3}\}^{\mathrm{T}}$
- Input matrix

$$\mathbf{X} = \begin{bmatrix} y_1 & y_2 & \cdots & y_{n_y} \\ y_2 & y_3 & \cdots & y_{n_y+1} \\ y_3 & y_4 & \cdots & y_{n_y+2} \end{bmatrix}_{n_x \times n_y}$$

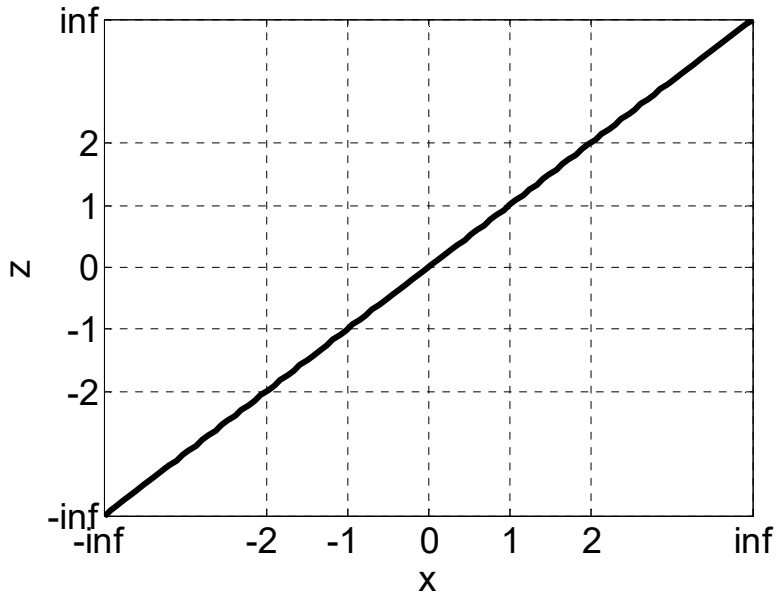
Output vector

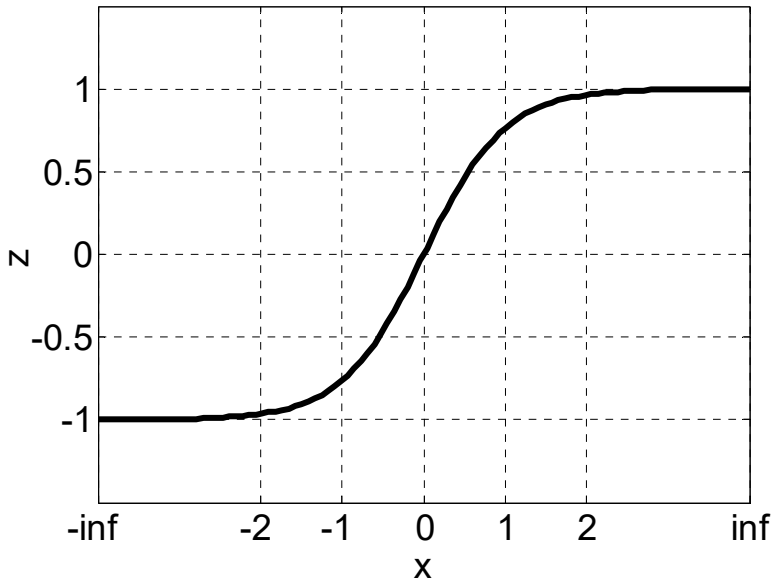
$$\hat{\mathbf{y}} = \{\hat{y}_4, \hat{y}_5, \ldots, \hat{y}_{n_y+3}\}$$

$$\mathbf{y} = \{y_4, y_5, \ldots, y_{n_y+3}\}$$

Transfer functions

- characterize the relationship between two adjacent layers
- should be differentiable: need for backpropagation training process
- Pure linear function: $\hat{y} = x$
- Sigmoid function: $\hat{y} = \frac{1-e^{-2x}}{1+e^{-2x}}$



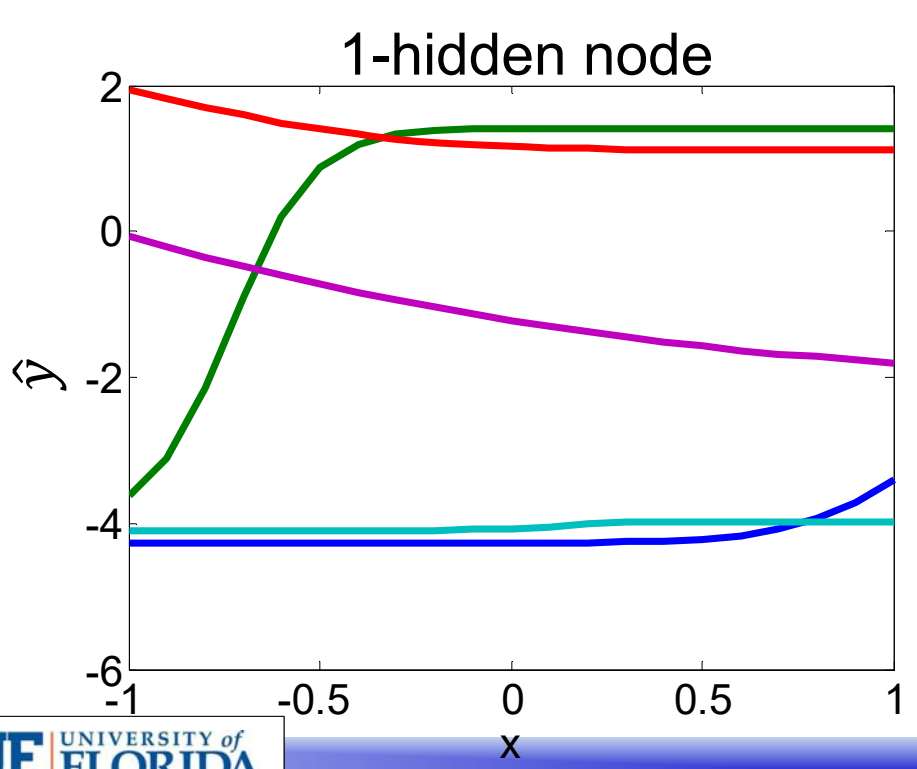


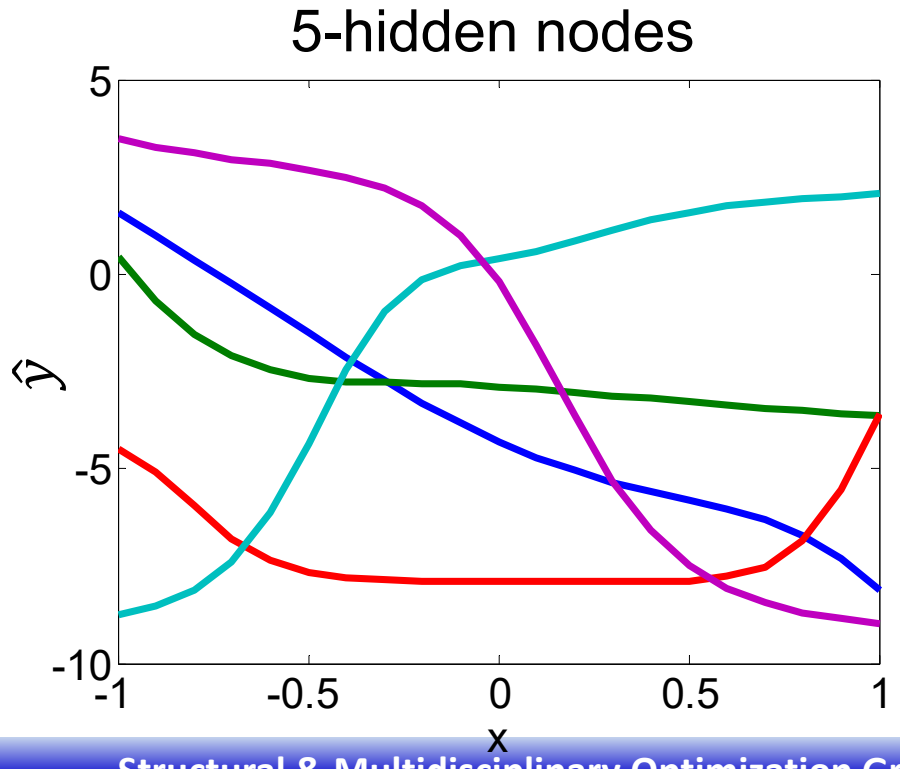
Ex) Complexity of NN model

- 2-layer NN model: 1 input node, 1 (or 5) hidden node, and 1 output node

$$\mathbf{h} = d_h\left(\mathbf{W}_x\mathbf{x} + \mathbf{b}_h\right) \Rightarrow \hat{\mathbf{y}} = d_z\left(\mathbf{W}_h\mathbf{h} + \mathbf{b}_z\right)$$

  - Generate random weights and biases from $\sim \mathrm{U}(-5,5)$
  - Repeat 5 times:





Backpropagation process

– a training method to determine weights and biases through a learning/optimization algorithm by backward propagation of the errors between the training data and the network outputs

– layer-by-layer updating

1. Setting initial values of parameters (weights & biases)
   – Usually random number in $[-1, 1]$, or fixed value of 1
2. Feedforward to simulation outputs $\hat{y}_i$ using n_y inputs
3. Mean-squared-error (MSE)

$$\text{MSE} = \frac{1}{n_y}\sum_{i=1}^{n_y}[y_i - \hat{y}_i(\mathbf{W}_x, \mathbf{B}_h, \mathbf{w}_h, \mathbf{b}_z)]^2$$

Backpropagation process *cont.*

4. Using optimization algorithm and gradients of MSE, calculate $\Delta\mathbf{w}_h$ and $\Delta\mathbf{b}_z$ of the output layer
   - Gradient descent or Levenberg-Marquardt (LM) algorithm
5. Calculate $\Delta\mathbf{W}_x$ and $\Delta\mathbf{B}_h$ of the hidden layer
   - $\Delta\mathbf{W}_x$ and $\Delta\mathbf{B}_h$ depend on $\Delta\mathbf{w}_h$ and $\Delta\mathbf{b}_z$
6. Update the weights and biases of NN model
7. Repeat Steps 1 ~ 6 until satisfying a stopping criterion

Matlab functions for feedforward NN

- define a network model → set up configurations → train the network model → prediction with different inputs
- `feedforwardnet`: creates a two-layer network model with user-defined number of hidden nodes
- `configure`: Optional step to normalize input and output data and assigns them to each node
- `train`: trains the network model with the LM backpropagation for training method and the MSE for the error function
- `sim`: simulates/predicts the network outputs at new inputs based on the identified weights and biases during the training process

Stopping criterion based on validation error

- Randomly divide data: training (70%), validation (15%), test (15%)
- use training set to update the weights and biases based on their error
- The error of the training set keeps decreasing, and the error in the validation set also decreases: the training process goes well
- The error in the validation set starts increasing after some phase of training process, i.e., good performance at the training set but bad performance at the validation set (new points, prediction points): the network model overfits the training data
- Therefore, the training process stops when the validation error starts increasing
- The test data sets are not used during the training process but used for testing the prediction accuracy with trained parameters

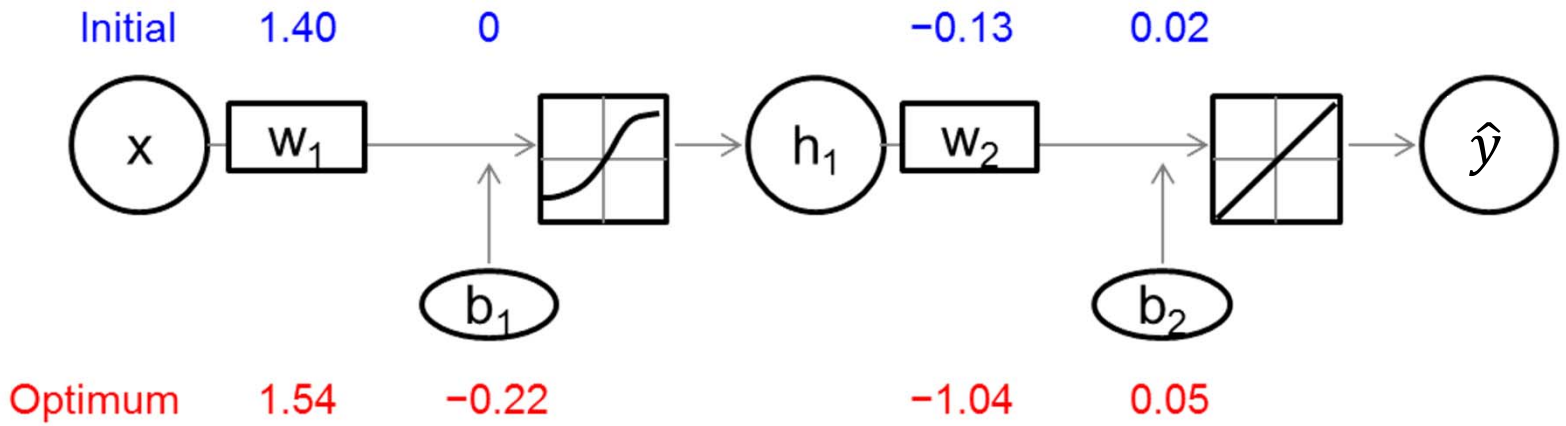
Ex) Feedforward NN model

- Predict y at $x_{\text{new}} \in [-5, 25]$ using one hidden-node NN

$$\mathbf{x}\ (=\mathbf{t}) = \begin{bmatrix} 0 & 5 & 10 & 15 & 20 \end{bmatrix}^{\mathrm{T}}, \quad \mathbf{y} = \begin{bmatrix} 1 & 0.99 & 0.99 & 0.94 & 0.95 \end{bmatrix}^{\mathrm{T}}$$

- Matlab code

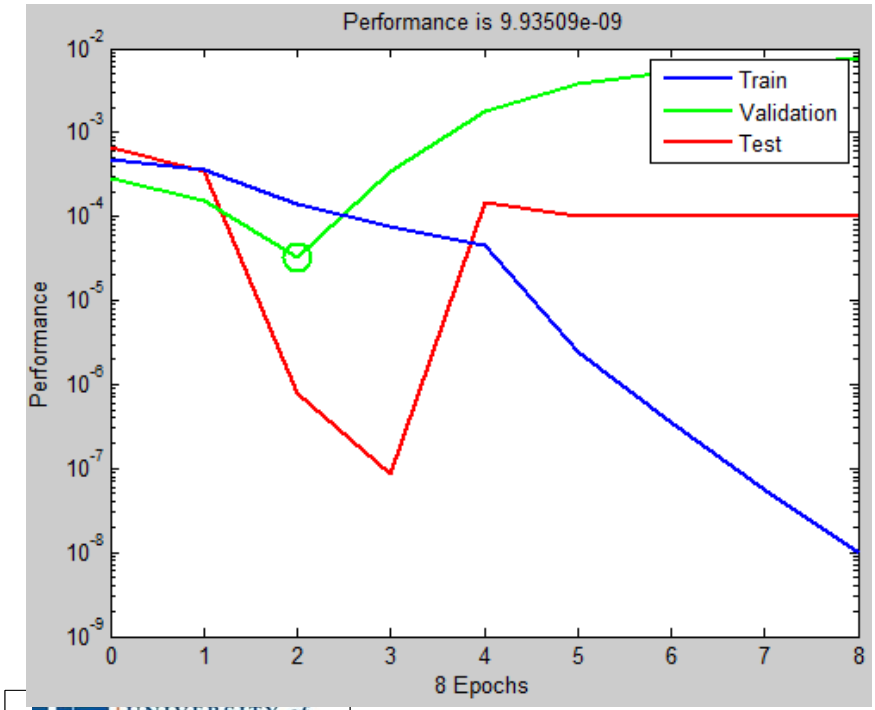
```
x=[0    5   10   15   20];        %[1 x 5] training input data
y=[1 0.99 0.99 0.94 0.95];       %[1 x 5] training output data
nh=1;                                    % num. of hidden node
net=feedforwardnet(nh);      % create two-layer network model
net=configure(net,x,y);                  % this is optional
[netModel,trainRecor]=train(net,x,y); % train the model 'net'
xNew=-5:0.1:25;                          % new input points
z=sim(netModel,xNew);                   % simulation results
```

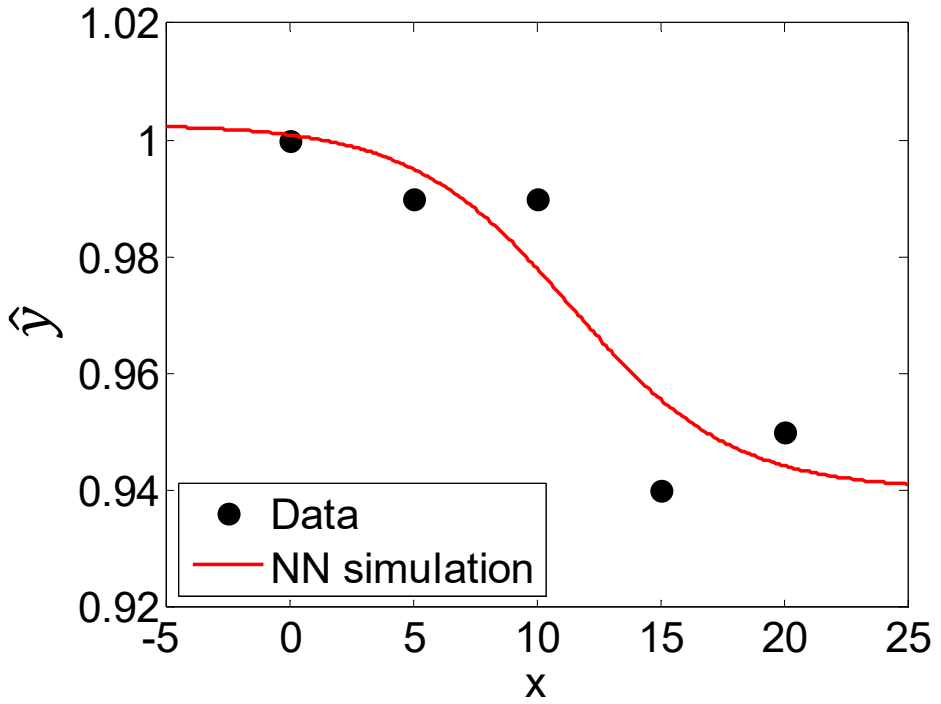


Ex) Feedforward NN model *cont.*

- # of training data (3) < # of parameters (4): not accurate
- **epoch**: one complete presentation of the data set to be learned to a **learning machine**

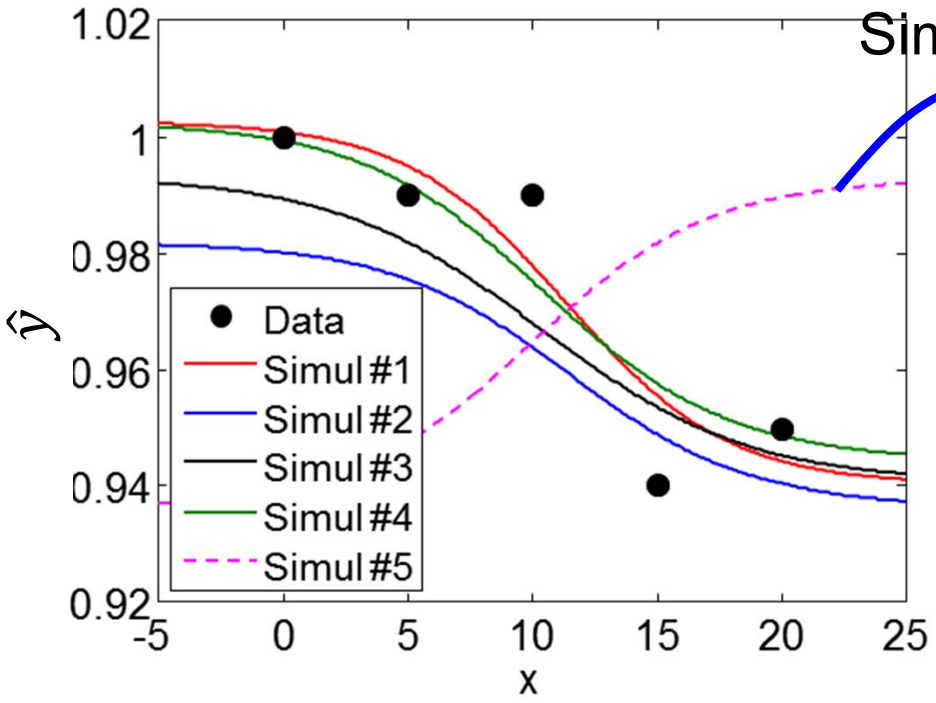
```
bestE=trainRecor.best_epoch;
bestP=trainRecor.best_vperf;
hold on; plot(bestE,bestP,'o');
```



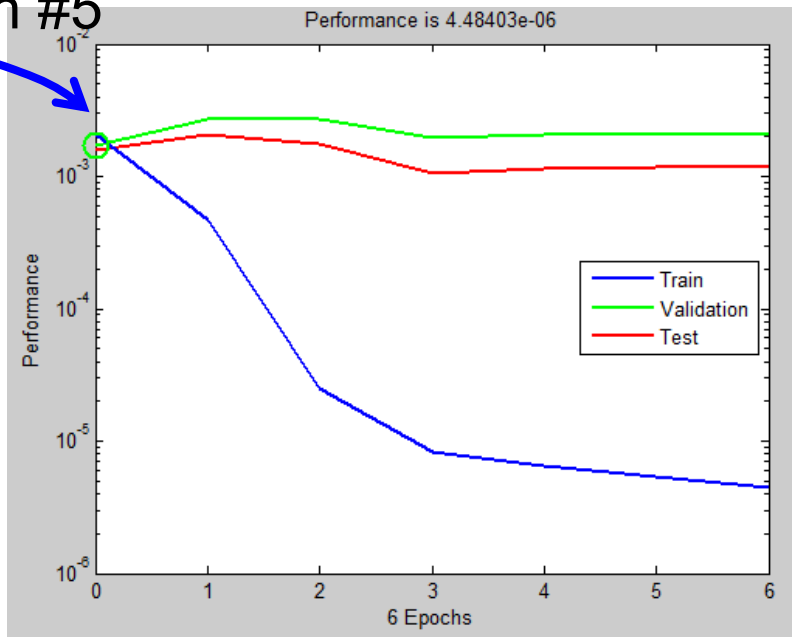


Ex) Feedforward NN model *cont.*

- Uncertainty
  - the initial values of parameters are assigned differently. Also, the train, validation, and test sets change when the training is re-started. These two are the sources of uncertainty in the neural network process and make different results

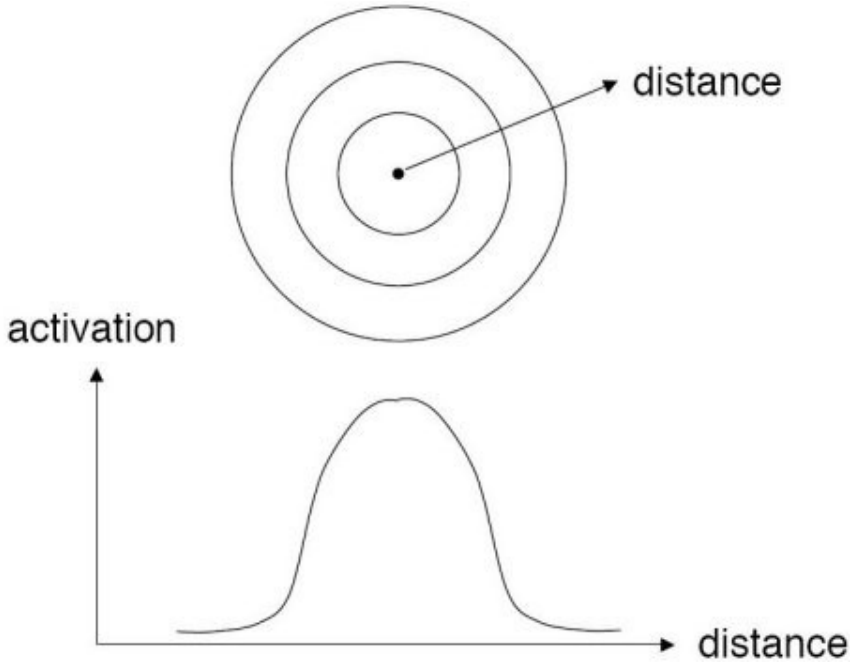


Simulation #5



UF UNIVERSITY of FLORIDA
The Foundation for The Gator Nation

Radial Basis Neural Network



Radial basis function (RBF)

- Real-valued function whose value depends only on the distance from its origin to the evaluation point

$$\phi_i(\mathbf{x}) = \phi(\|\mathbf{x} - \mathbf{c}_i\|)$$

$\mathbf{c}_i$: center

  - It is also called radial kernel function

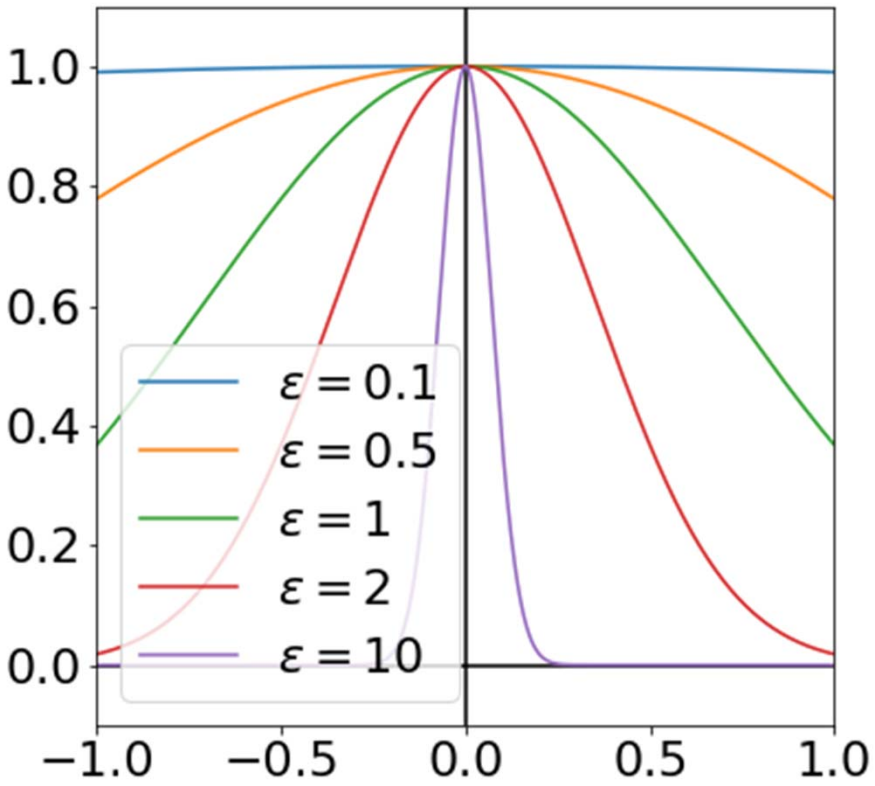
- Use $r = \|\mathbf{x} - \mathbf{c}_i\|$ as a distance and ϵ as a spread parameter

  - Gaussian $\phi(r) = e^{-\varepsilon r^2}$
  - Multiquadric $\phi(r) = \sqrt{1 + \varepsilon r^2}$
  - Inverse quadratic $\phi(r) = \dfrac{1}{1 + \varepsilon r^2}$

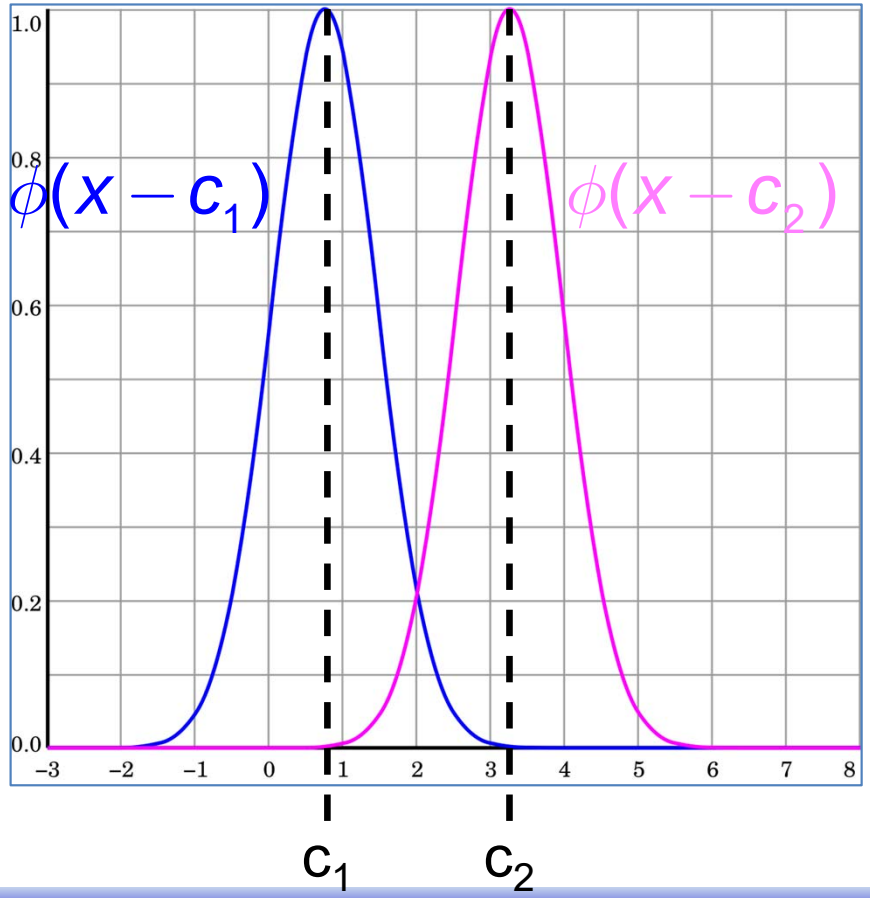
Parameters in radial basis function

- Spread parameter ϵ determines the influence range of the radial basis function
- Center $\mathbf{c}_i$ shifts the basis function

Effect of spread parameter



Effect of center



Approximation using radial basis functions

- Radial basis function (RBF) model
  - Linear combination of radial basis functions

$$\hat{y}(x) = \sum_{i=1}^{N_{\text{RBF}}} b_i \phi_i(x)$$

  - Model parameters: coefficient b_i, center c_i, spread parameter ϵ_i
  - When the centers and spread parameters are fixed, linear regression can be used to determine unknown coefficients b_i

Ex) Matlab radial basis function

- Radial basis as a transfer function

```
x = -3:.1:3; a = radbas(x);
plot(x,a); title('Radial Basis Function');
xlabel('X'); ylabel('\phi (x) ');
```

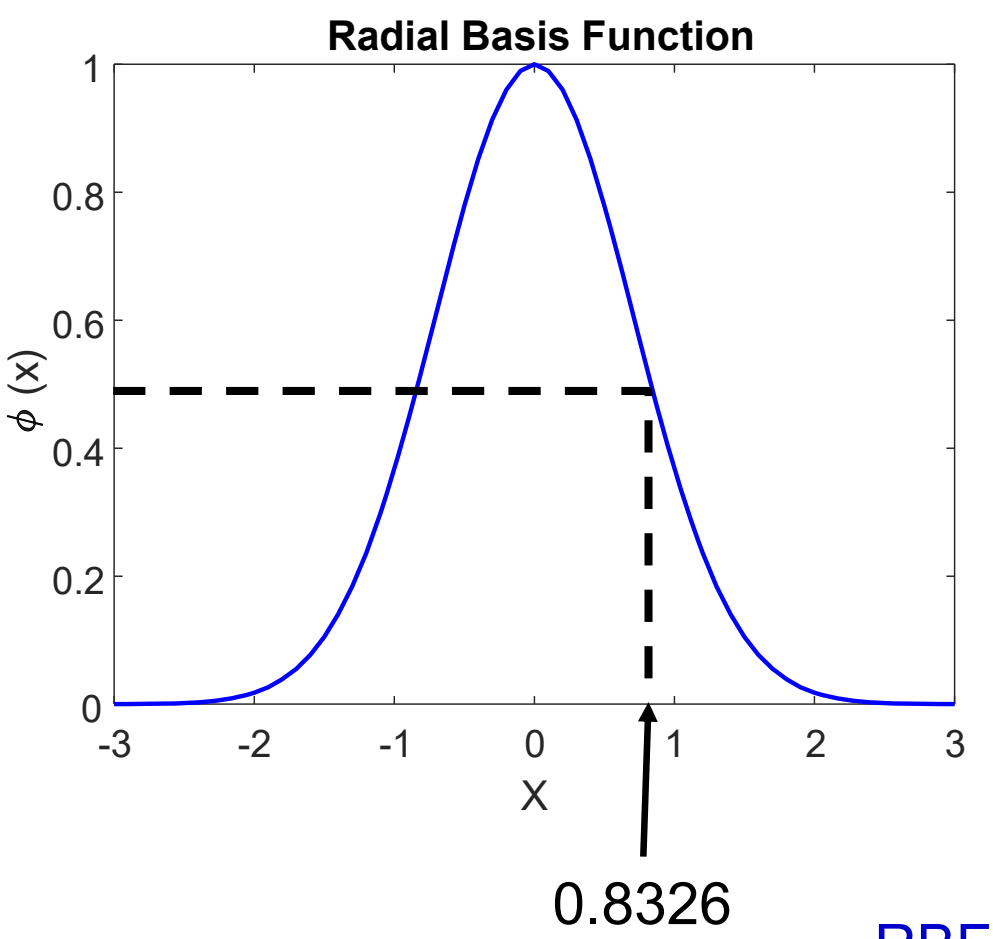
- Weight = center position, bias = width of RBF (spread)

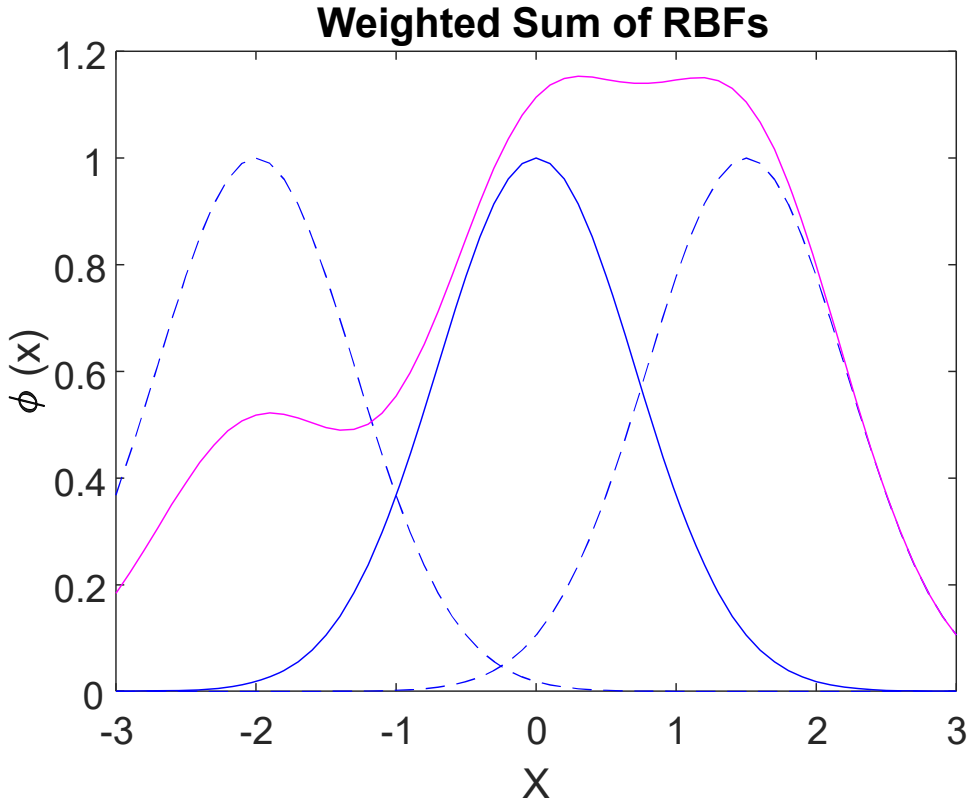
$$\phi(x,w,\varepsilon) = e^{-\varepsilon\|x-w\|^2}$$

- Linear combination of 3 RBFs

```
a2 = radbas(x-1.5); a3 = radbas(x+2);
a4 = a + a2*1 + a3*0.5;
plot(x,a,'b-',x,a2,'b--',x,a3,'b--',x,a4,'m-')
title('Weighted Sum of RBFs');
xlabel('X'); ylabel('\phi (x)');
```

Ex) Matlab radial basis function *cont.*

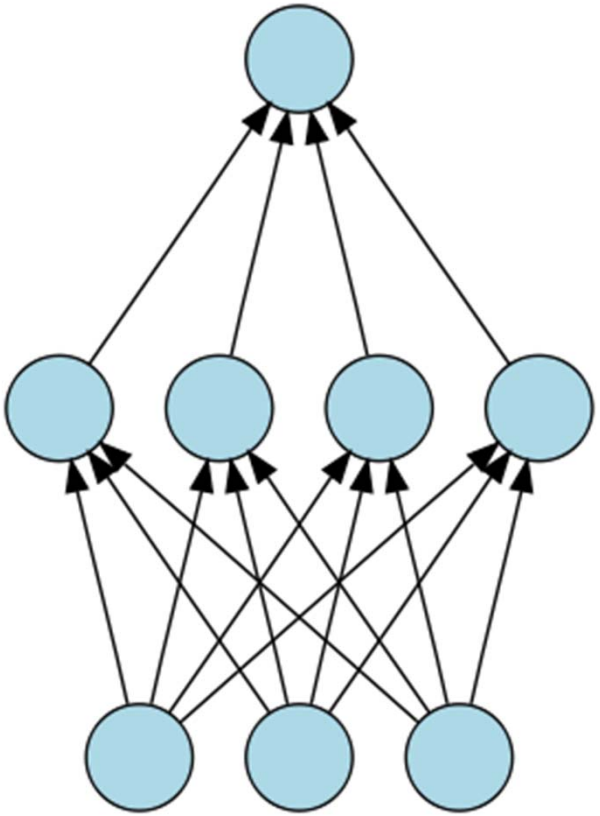




RBF = 0.5 at distance = 0.8326

Radial basis neural network (RBNN) model

- NN model with input, hidden, and output layer
  - Hidden nodes have RBF as a transfer function
  - Use the weight as a center point and bias as a spread



$$\phi(x, w, \varepsilon) = e^{-\varepsilon \|x-w\|^2}$$

Matlab radial basis neural network newrbe

- Exact design (newrbe)
  - produce a network with zero error on training vectors (interpolation)
  - `net = newrbe(x,y,spread)`
  - Generate same number of hidden nodes with the input samples with the weights being the location of input samples
  - User provides bias (spread): $\epsilon = 0.8326/\text{spread}$
  - Determine output weights (n_y) and output bias (1) using linear regression

$$\hat{y} = b_z + \sum_{i=1}^{n_y} w_{hi}\phi_i(x)$$

- Regression error vector
  - with n_y samples $(\mathbf{x}_i, y_i)$

$$\mathbf{e} = \mathbf{y} - \mathbf{X} \cdot \mathbf{w}_h - \mathbf{b}_z$$

$$e_j = y_j - \sum_{i=1}^{n_y} w_{hi} \phi_i(\mathbf{x}_j) - b_z$$

Evaluation of basis at sample points

  - In matrix notation

$$\begin{Bmatrix} e_1 \\ e_2 \\ \vdots \\ e_{n_y} \end{Bmatrix} = \begin{Bmatrix} y_1 \\ y_2 \\ \vdots \\ y_{n_y} \end{Bmatrix} - \begin{bmatrix} \phi_1(\mathbf{x}_1) & \phi_2(\mathbf{x}_1) & \cdots & \phi_{n_y}(\mathbf{x}_1) & 1 \\ \phi_1(\mathbf{x}_2) & \phi_2(\mathbf{x}_2) & \cdots & \phi_{n_y}(\mathbf{x}_2) & 1 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ \phi_1(\mathbf{x}_{n_y}) & \phi_2(\mathbf{x}_{n_y}) & \cdots & \phi_{n_y}(\mathbf{x}_{n_y}) & 1 \end{bmatrix} \begin{Bmatrix} w_{h1} \\ w_{h2} \\ \vdots \\ w_{n_y} \\ b_z \end{Bmatrix}$$

$\mathbf{X}: n_y \times (n_y + 1)$ design matrix

Regression for RBNN *cont*.

- Minimum of function = zero derivative

$$\frac{\mathrm{d}}{\mathrm{d}\mathbf{b}}(\mathbf{e}^{\mathrm{T}}\mathbf{e}) = -2\mathbf{X}^{\mathrm{T}}\mathbf{y} + 2\mathbf{X}^{\mathrm{T}}\mathbf{X}\mathbf{w} = 0$$

$$\Rightarrow \boxed{\mathbf{X}^{\mathrm{T}}\mathbf{X}\mathbf{w} = \mathbf{X}^{\mathrm{T}}\mathbf{y}}$$

Equation of linear regression
Normal equation

$$\mathbf{w} = \underbrace{(\mathbf{X}^{\mathrm{T}}\mathbf{X})^{-1}}\mathbf{X}^{\mathrm{T}}\mathbf{y}$$

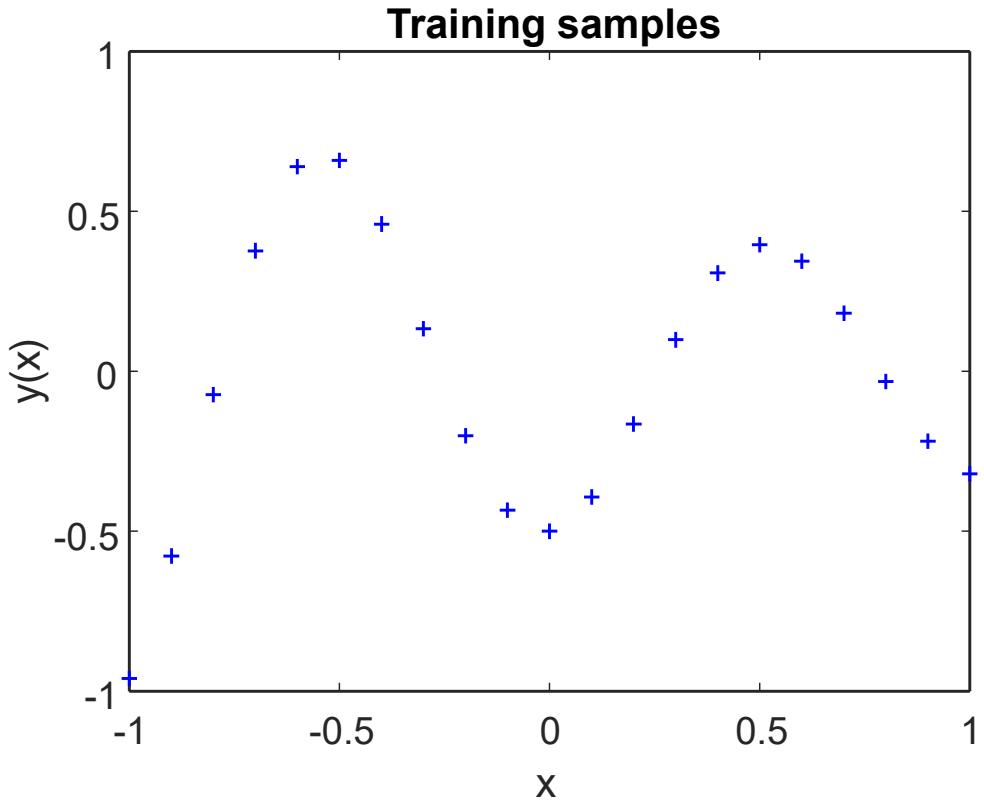
$n_y \times n_y$ matrix: ill-conditioned for large n_y

– Remedy: Solve $\mathbf{X}\mathbf{w} = \mathbf{y}$ using QR decomposition

Ex) Matlab radial basis approximation

- 21 training samples

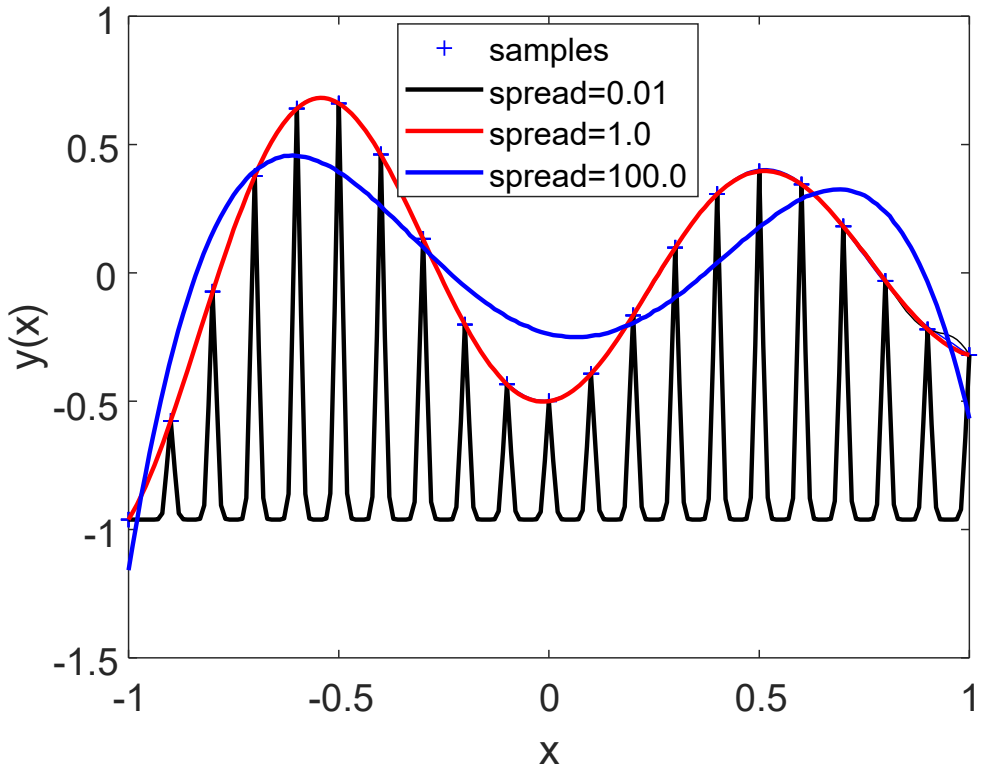
```
x = -1:.1:1;
y = [-.9602 -.5770 -.0729  .3771  .6405  .6600  .4609 ...
      .1336 -.2013 -.4344 -.5000 -.3930 -.1647  .0988 ...
      .3072  .3960  .3449  .1816 -.0312 -.2189 -.3201];
plot(x,y, '+'); title('Training samples');
xlabel('x'); ylabel('y(x)');
```



Ex) Effect of spread (exact design newrbe)

- Three different spreads (0.01, 1.0, 100.0)

```
sc = 0.01; net1 = newrbe(x,y,sc); Y1 = net1(X);
sc = 1.0; net2 = newrbe(x,y,sc); Y2 = net2(X);
sc = 100.0; net3 = newrbe(x,y,sc); Y3 = net3(X);
plot(x,y,'+b',X,Y1,'-k',X,Y2,'-r',X,Y3,'-b'); hold on;
xlabel('x'); ylabel('y(x)');
legend({'samples','spread=0.01','spread=1.0','spread=100.0'});
```



Even if newrbe is designed to interpolate all samples, it may not pass samples when the spread is too large

When the spread is too small, most points predict the minimum sample values, except for actual sample points

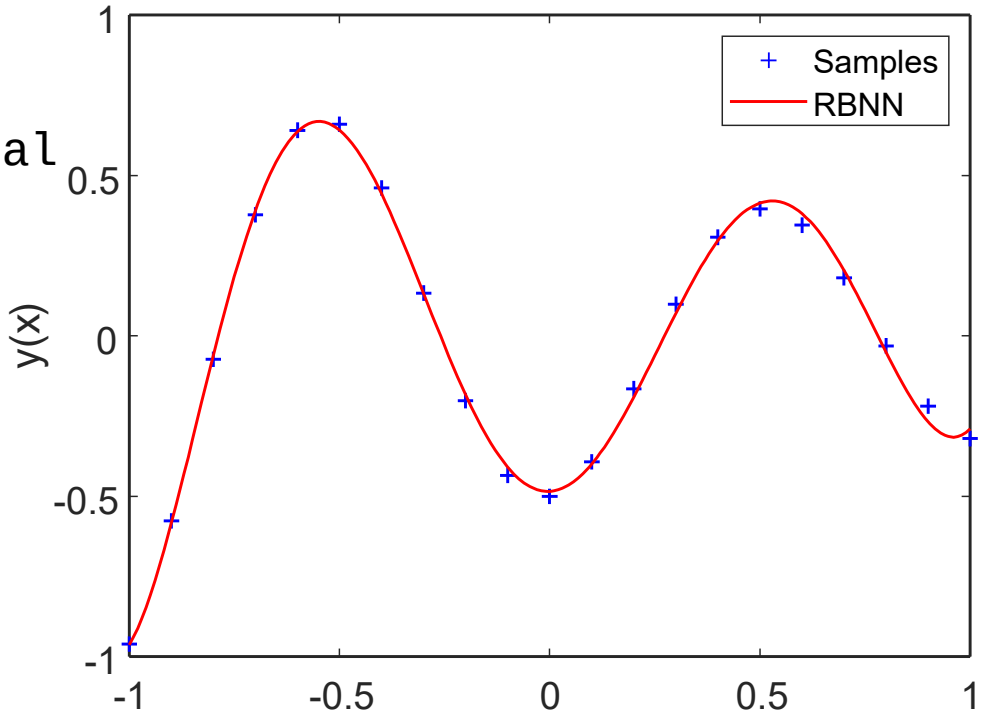
Matlab radial basis neural network newrb

- Efficient design (newrb)
  - iteratively creates a radial basis network one neuron at a time
  - `net = newrb(x,y,goal,spread)`
  - Neurons are added until MSE become less than GOAL or the max # of neurons has been reached
  - Iteration i, $\mathbf{x}_i$ that lowers MSE the most is used to create a neuron
- Compared with NN, RBNN requires more neurons
  - RBF are local, while linear or sigmoid transfer functions are global
  - But RBNN training is more efficient

Ex) Matlab RBNN approximation *cont.*

- RBNN fit

```
eg = 0.02; % sum-squared error goal
sc = 1;    % spread constant
net = newrb(x,y,eg,sc);
X = -1:.01:1; Y = net(X);
plot(x,y,'+b',X,Y,'-r'); hold on;
xlabel('x'); ylabel('y(x)');
legend({'Samples','RBNN'});
```



- How to choose spread?
  - larger than the distance between adjacent input samples
  - smaller than the distance across the whole input space

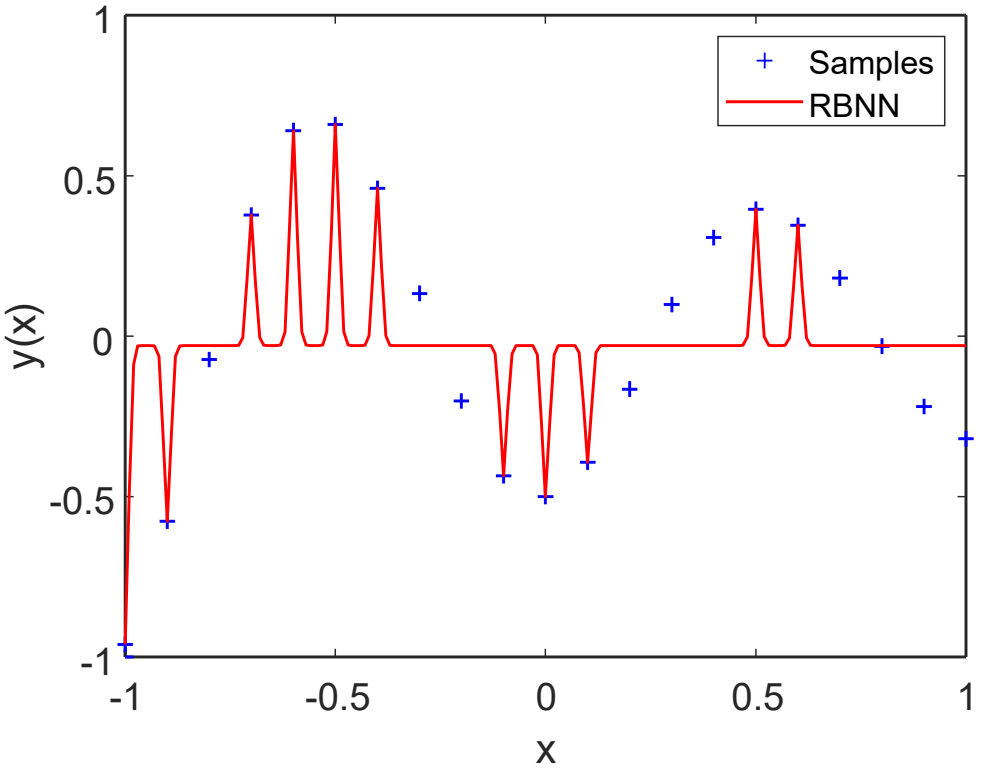
$$\phi(x,w,b) = e^{-b\|x-w\|^2}$$

Ex) Matlab RBNN approximation *cont.*

- Repeat with spread = 0.01

```
sc = 0.01;     % spread constant
```

  - Too small spread constant (underlapping neurons)
  - No two radial basis neurons cover the same point



Ex) Matlab RBNN approximation *cont.*

- Repeat with spread = 100

```
sc = 100;     % spread constant
```

  - Too large spread constant (overlapping neurons)
  - All radial basis neurons are 1 at any point