Machine Learning
Baruch College
Lecture 2

Miguel A. Castro

Today We'll Cover:

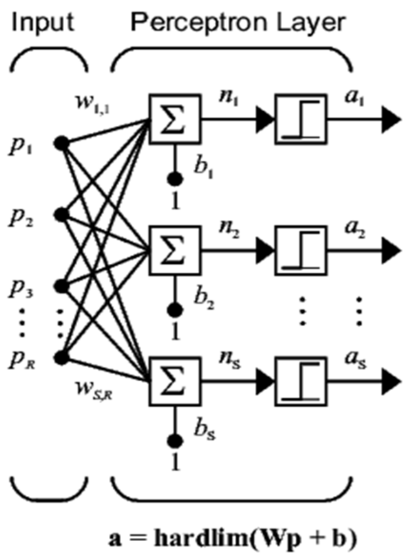
- Review of Last Lecture
- Simulations I owed you
- Continue Neural Networks

Last Time…

- Class Syllabus
  - I didn't realize there's No Class September 29
  - One less HW Assignment (3 instead of 4)
- Importance/Usefulness of Machine Learning
- Brief History of ML in the Context of AI
- Supervised vs. Unsupervised Learning
- Neural Networks
  - Important and useful example of ML
  - Originally models of the biological brain
  - Useful in their own right
  - Starting point for ML

Neural Networks (Review from Last Time)

- Perceptron
  - Hard Limit Activation Functions (thought to emulate neurons)
- Single-Layer Perceptron



R inputs

S outputs

$\mathbf{W}$ is an SxR matrix of weights

$\mathbf{b}$ is an S vector of biases

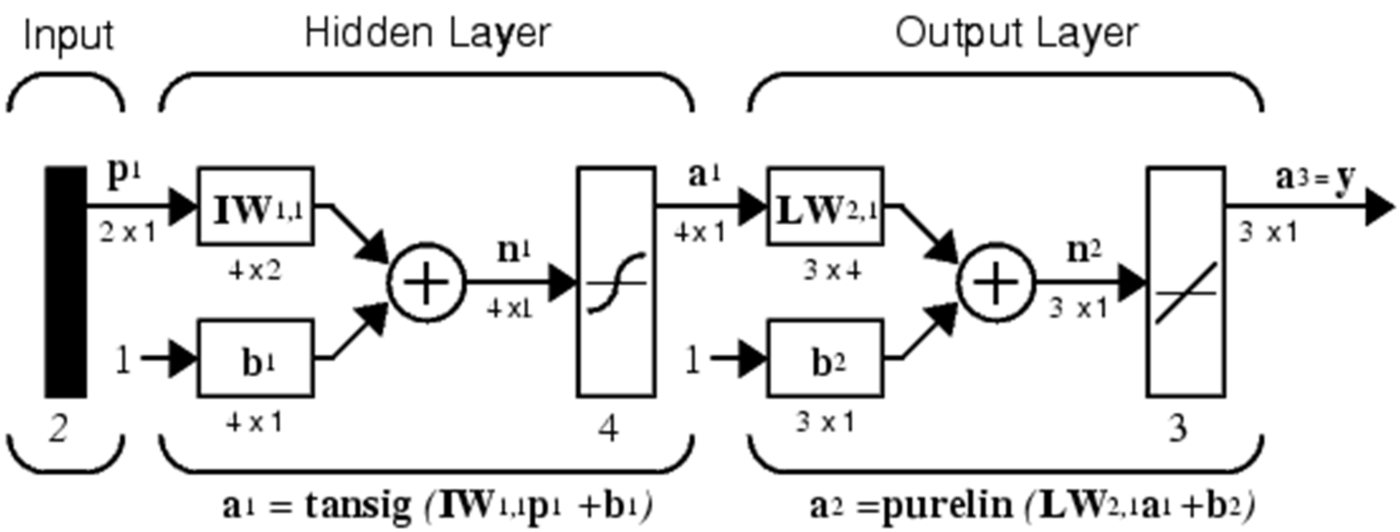
- Perceptron Learning Rule
- Correctly Classifies AND, OR functions, but *not* XOR
- Can only handle *Linearly Separable* Classes

Neural Networks (Review from Last Time)

- Multi-Layer Perceptron (MLP)
  - Can be constructed to solve XOR (in a contrived way)
  - Can solve *Linearly Non-Separable* problem; ***but…***
  - No analog of Perceptron Learning Rule for Multiple Layers
  - Need differentiable Activation Functions to produce Learning Rule
- ADALINEs/MADALINEs
  - Linear Activation Functions (differentiable → *Delta Learning Rule*)
  - Delta Learning Rule can be automated for multiple layers
  - Linear Activations still only work for *Linearly Separable* Problems

Neural Networks (Review from Last Time)

- Wish List for General-Purpose Approximator is to Combine:
  - Multiple Layers (we learned from Perceptrons)
  - Nonlinear Activations (we learned from ADALINEs)
  - Automated Learning Rule ➔ Smooth (Differentiable) Activations
- FeedForward Neural Networks



  - *Multi-Layer*
  - Sigmoidal or other *non-linear* but *smooth* Activations
  - Learning Rule: ***Backpropagation*** (generalization of the Delta Learning Rule)
  - Hornik *et.al.* (1989): This is a ***Universal Approximator***

Neural Networks (Review from Last Time)

- FeedForward Neural Networks:

  - Also known as MLPs with non-linear activations, we won't use that name here (reserve for Hard-Limit Activation Perceptrons)

  - A *<u>Single Hidden-Layer</u>* of Non-Linear Activations is sufficient for universal approximation (White *et al*.)

  - Workhorse is Sigmoidal Hidden Layer and Linear Output Layer

- Backpropagation

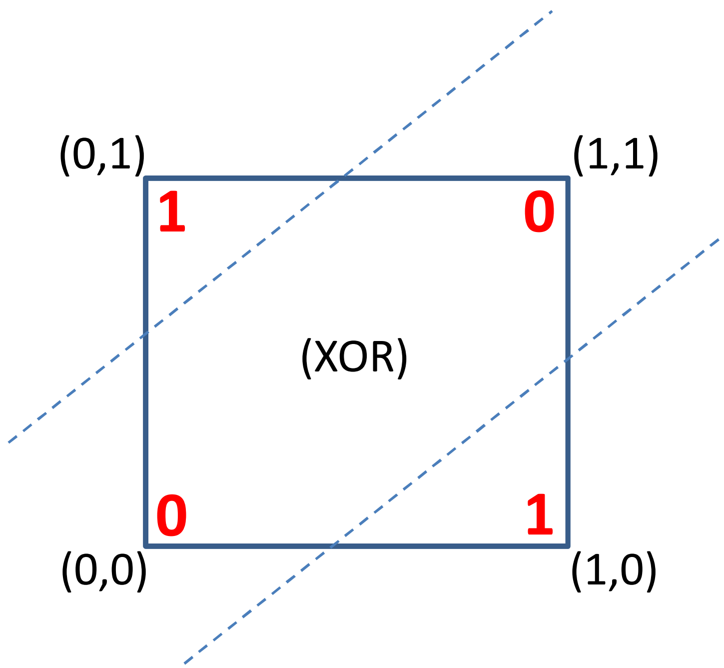
  - Generalization of the Delta Learning Rule

  - Perform a gradient descent by starting at the output and propagating the error back to the input weights using the chain rule of differentiation:

$$\Delta w_{hj} = -\eta \frac{\partial E}{\partial w_{hj}} = -\eta \frac{\partial E}{\partial y_i} \frac{\partial y_i}{\partial a_h} \frac{\partial a_h}{\partial w_{hj}} \dots$$

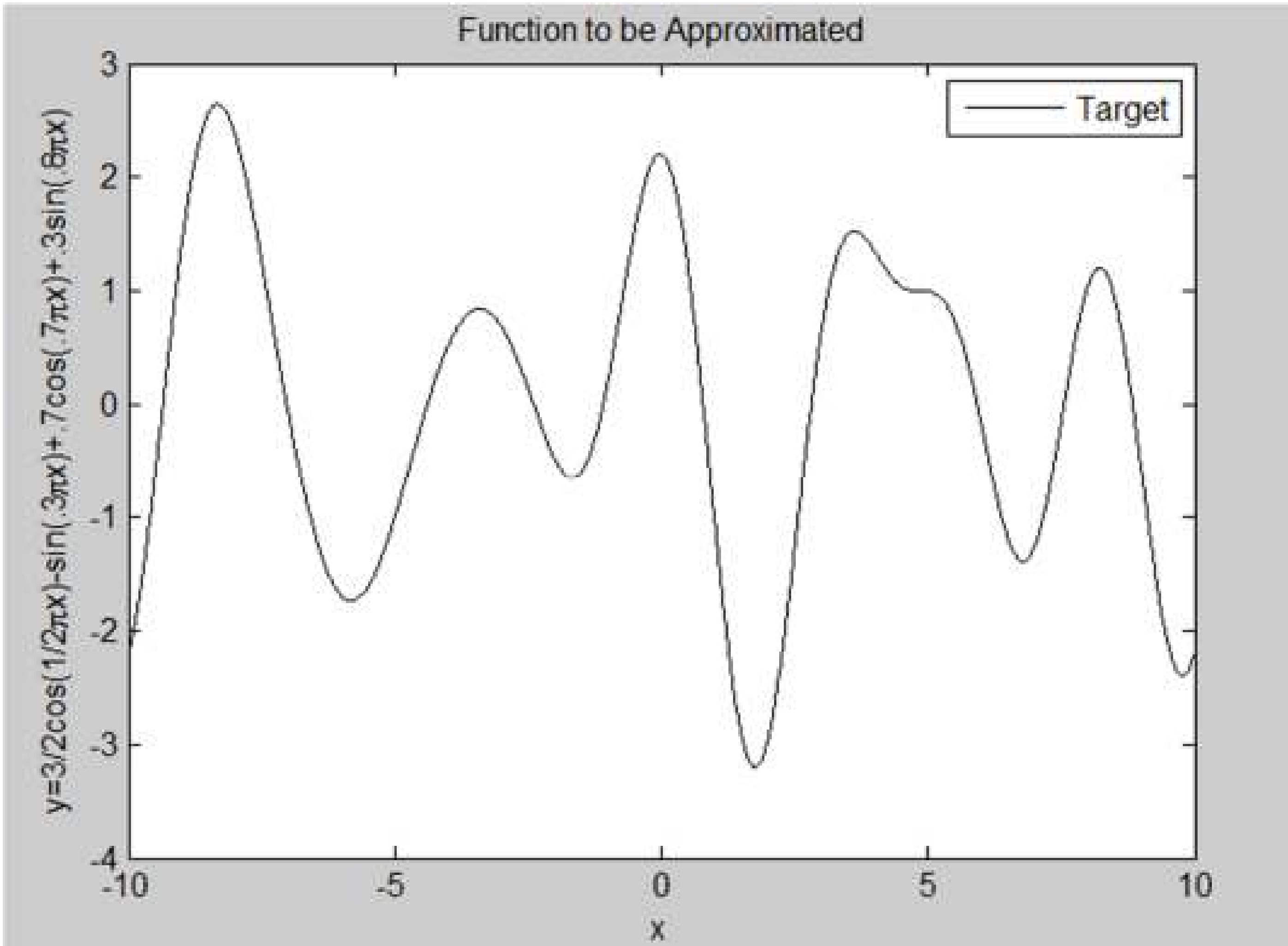
Demos from Last Time…

- FFN with single hidden layer of sigmoidal activations and linear output can classify XOR, where Perceptron Flounders.

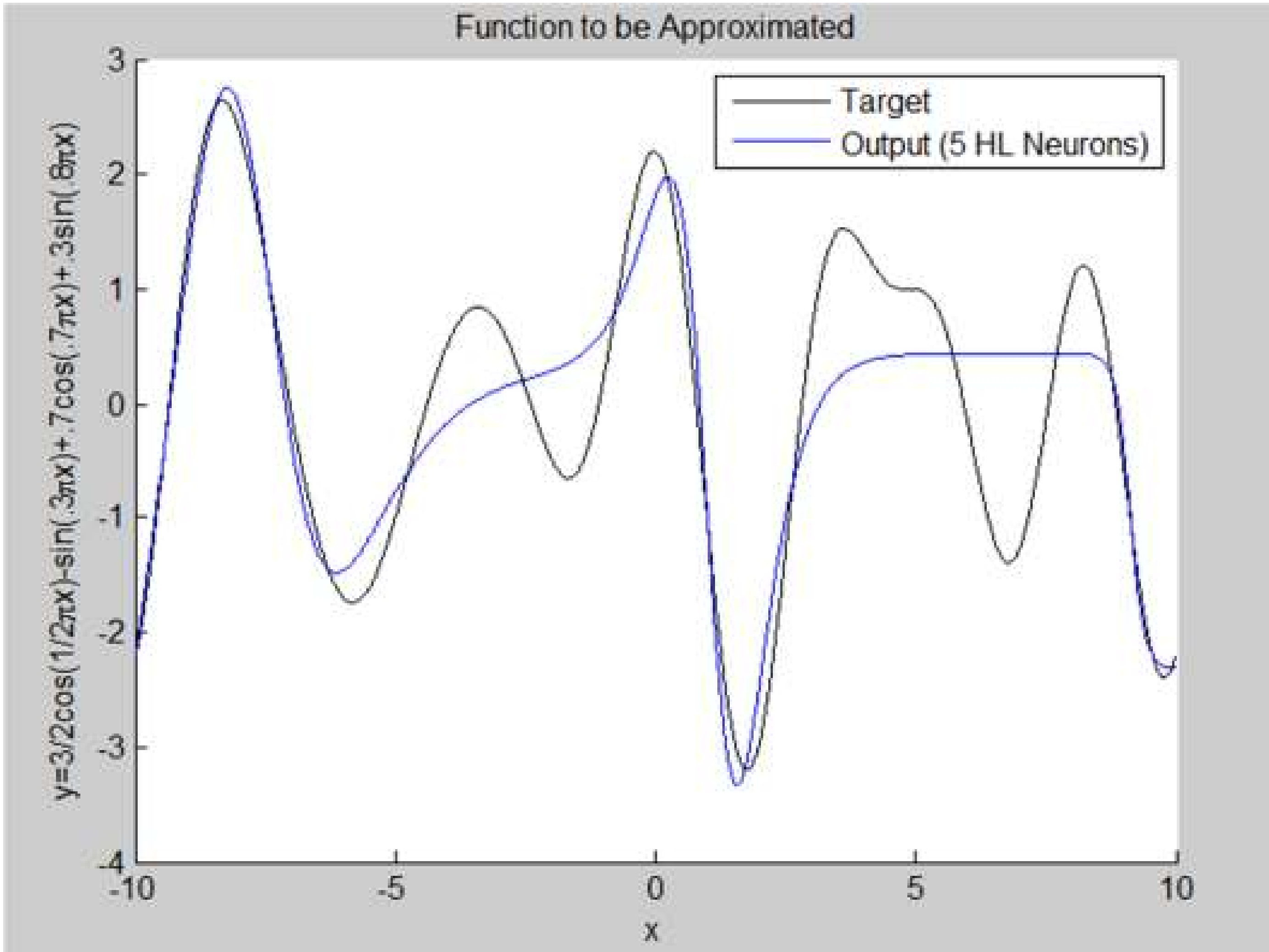


- Can classify any ***<u>Linearly Non-Separable</u>*** Problem, even more difficult ones in higher dimensions, or with more than 2 classes…

Function Approximation

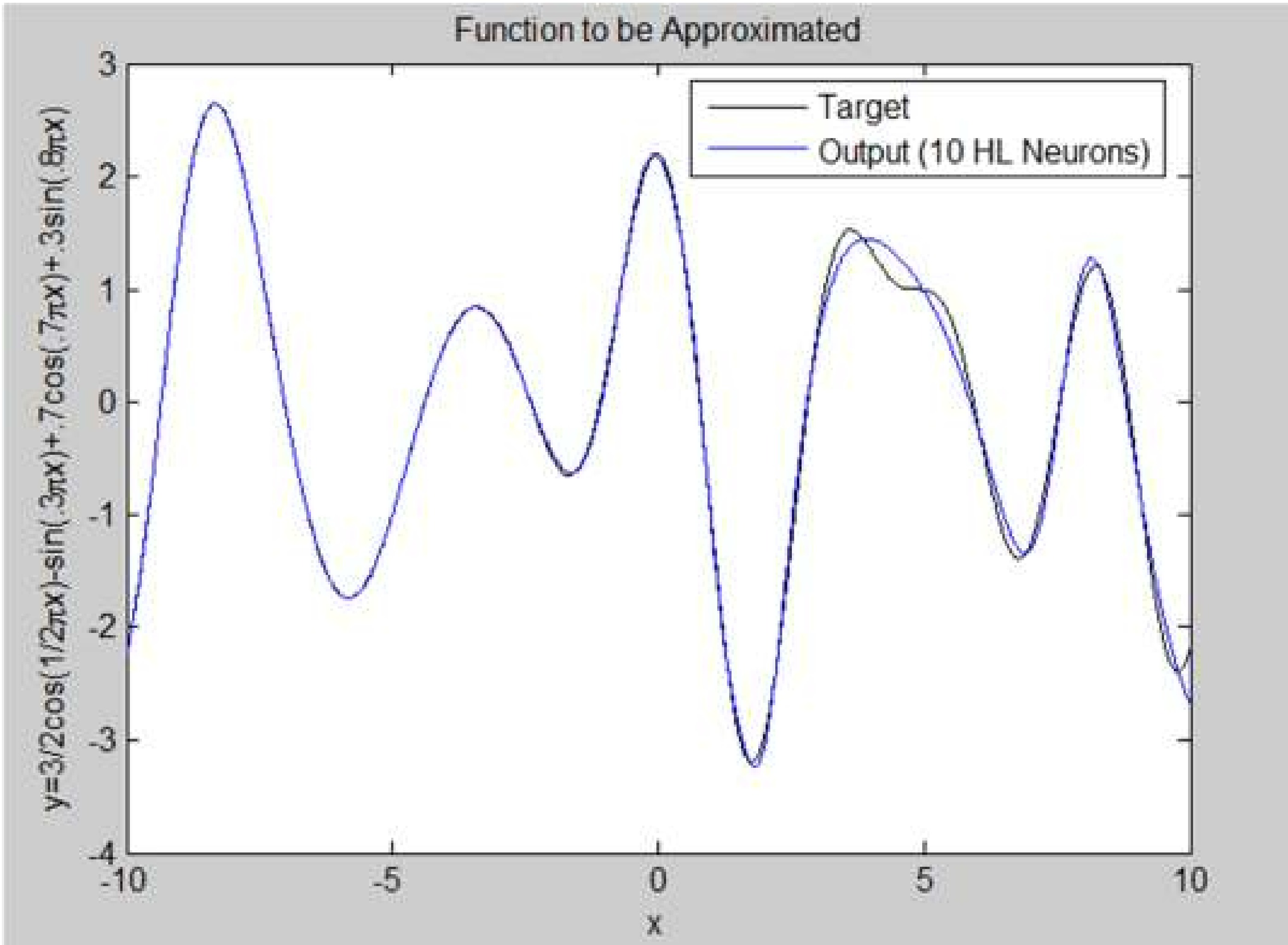


Function Approximation (FNN, 5 HN)



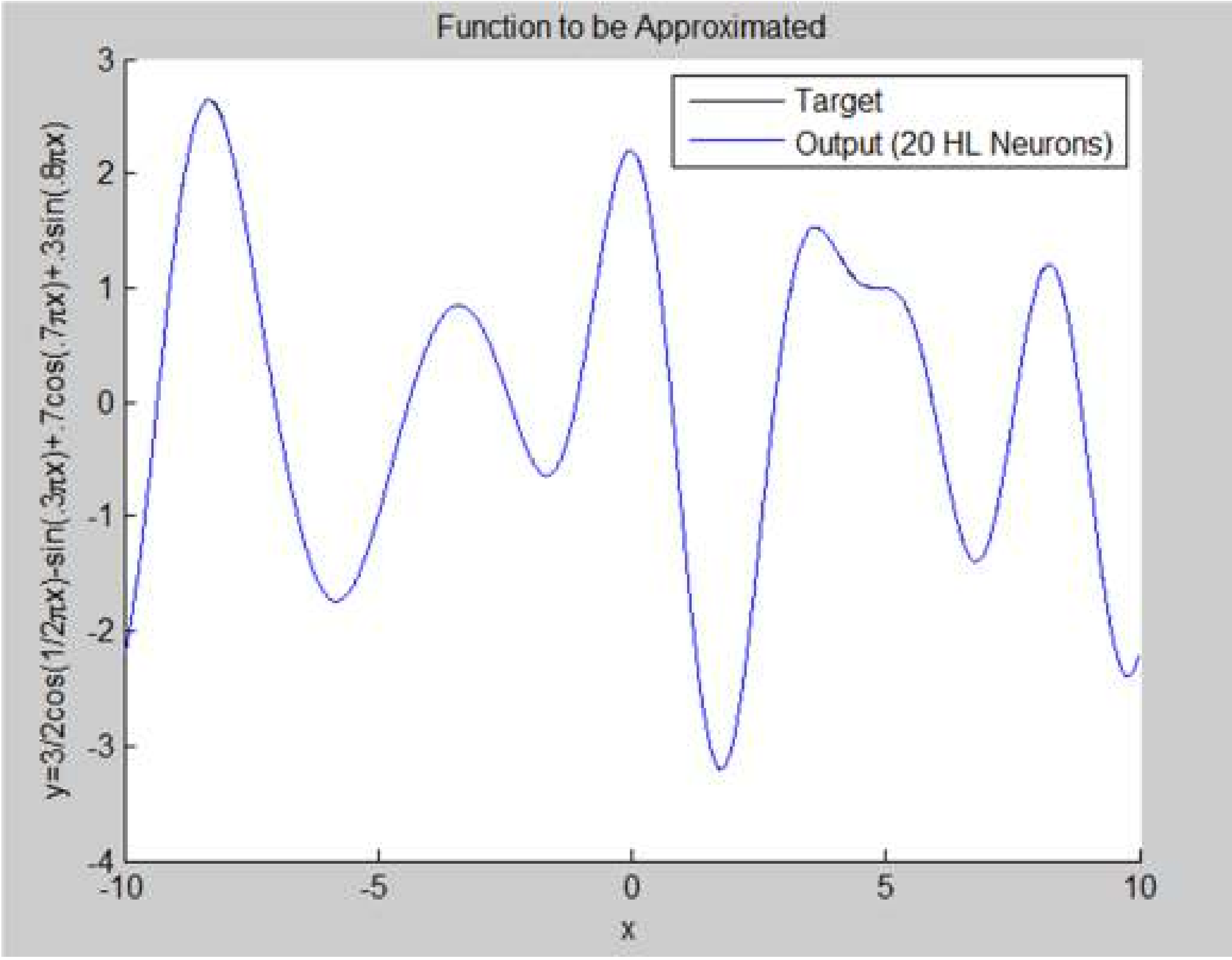
- Pretty Good

Function Approximation (FNN, 10 HN)



- Better

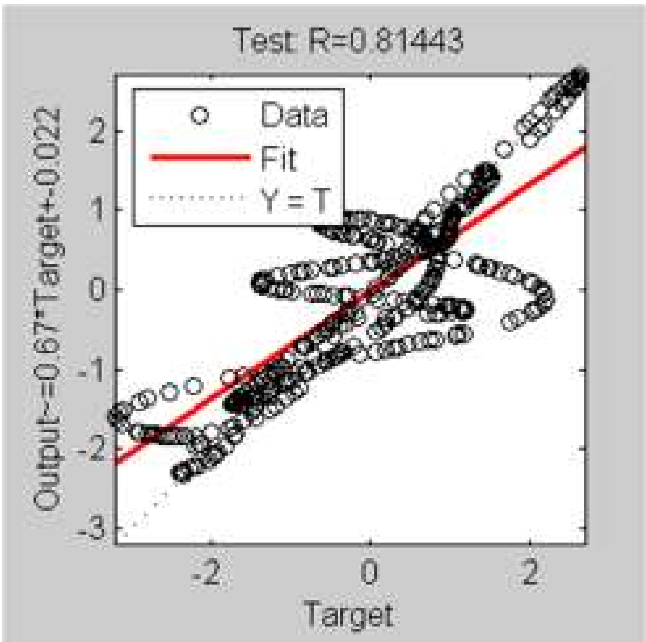
Function Approximation (FNN, 20 HN)



- Spot on!

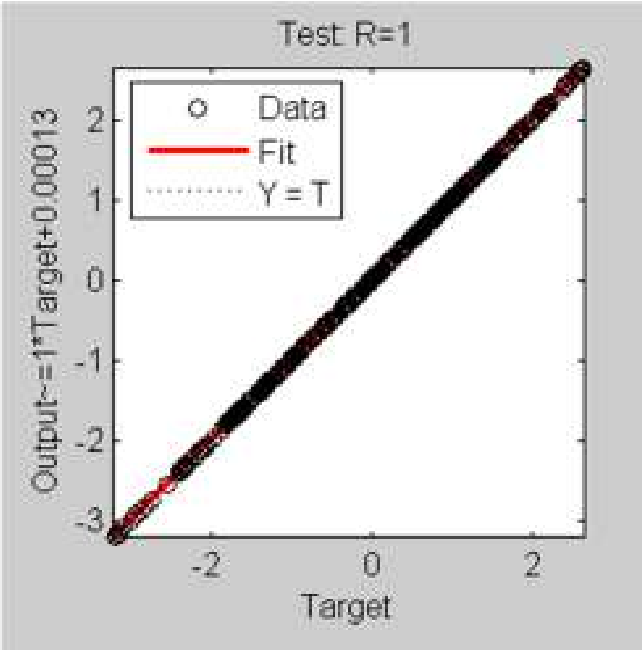
Function Approximation: Evaluation

- We don't have to rely on eyeballing
- Regress Output vs. Target to get stats:



5 Hidden-Layer Neurons

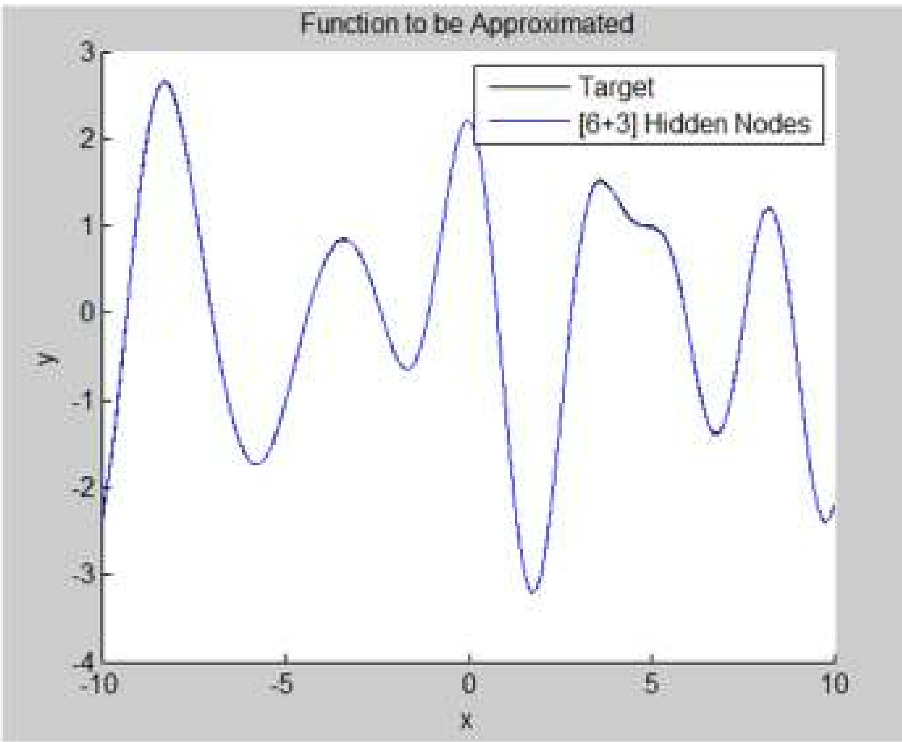
Function Approximation: Evaluation

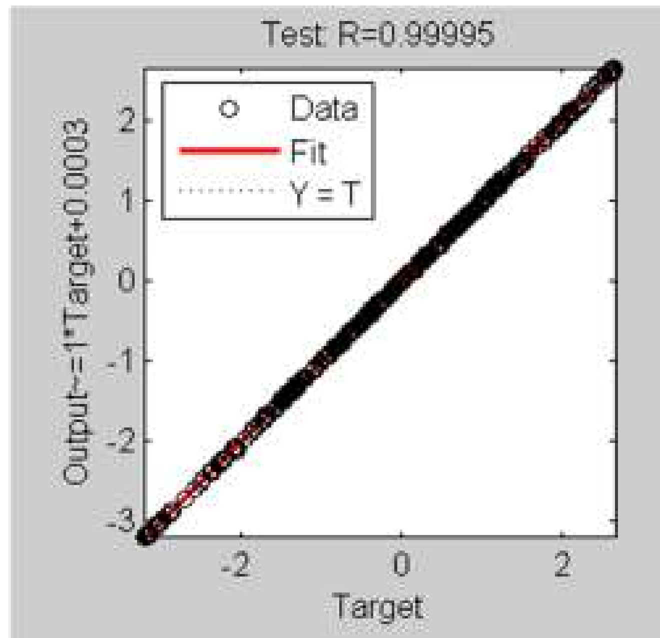


20 Hidden-Layer Neurons

Single vs. Multiple Hidden Layers

- We found that a single hidden layer with 20 nodes (neurons) worked extremely well.
- Now Consider two hidden layers with [6+3] nodes:



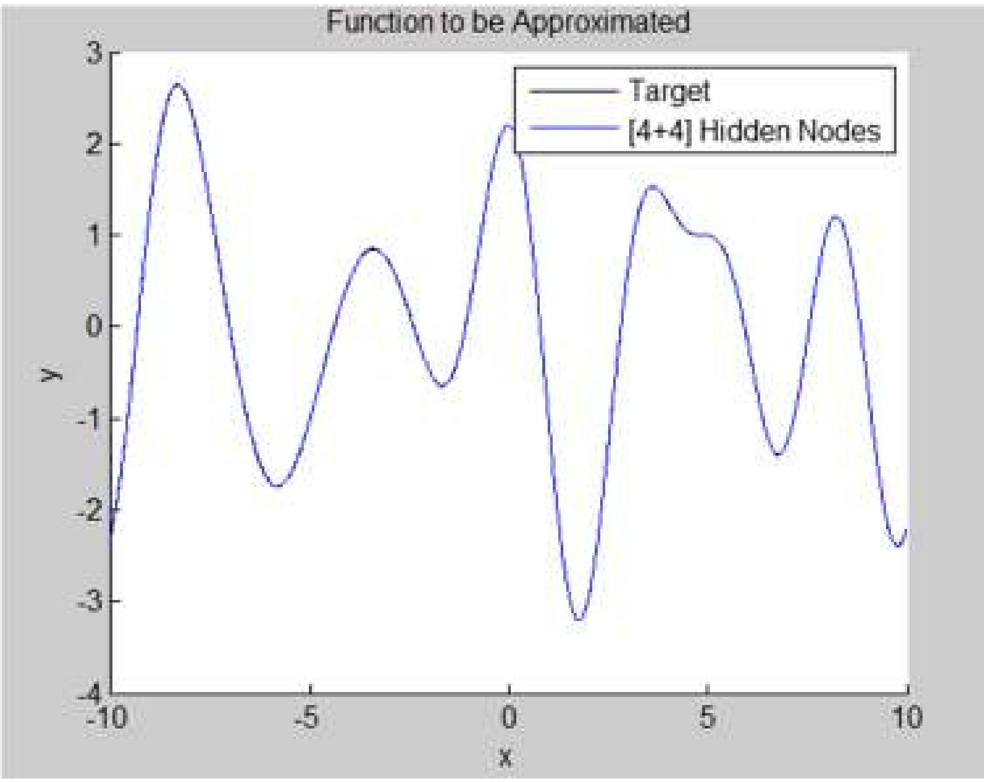


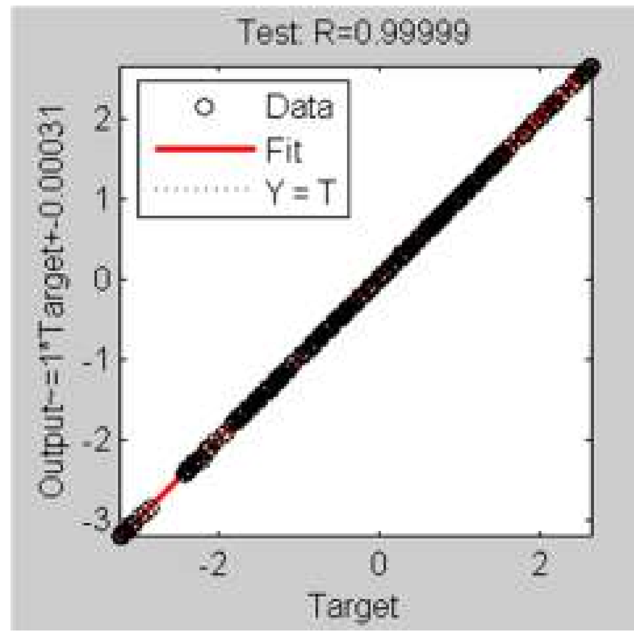
[6+3] Hidden Nodes

- Works extremely well also.

Single vs. Multiple Hidden Layers

- Now Consider two hidden layers with [4+4] nodes:





[4+4] Hidden Nodes

- Works extremely well also.

Single vs. Multiple Hidden Layers

- Let's take a closer look.
- Are Neural Networks Parametric or Nonparametric techniques?
- They can be treated as Nonparametric:

$$Y_i = f(\mathbf{X}_i) + \varepsilon_i,$$

- Where $f \in \mathcal{F}$, some class of functions.
- The only requirement is for $\mathcal{F}$ to be sufficiently rich, but we're not really interested in f itself.
- So, to the extent that NNs are "black boxes" they behave as if they were Nonparametric. However…

Single vs. Multiple Hidden Layers

- NNs have internal parameters: the connection strengths (weights).
- In this regard they are Parametric, and suffer from the need for parsimony exhibited by Parametric models.
- Consider a FFNN with I Inputs, J Hidden Layers, and O outputs, with H_j nodes in the j^{th} Hidden Layer.
- The number of parameters (weights) is just:

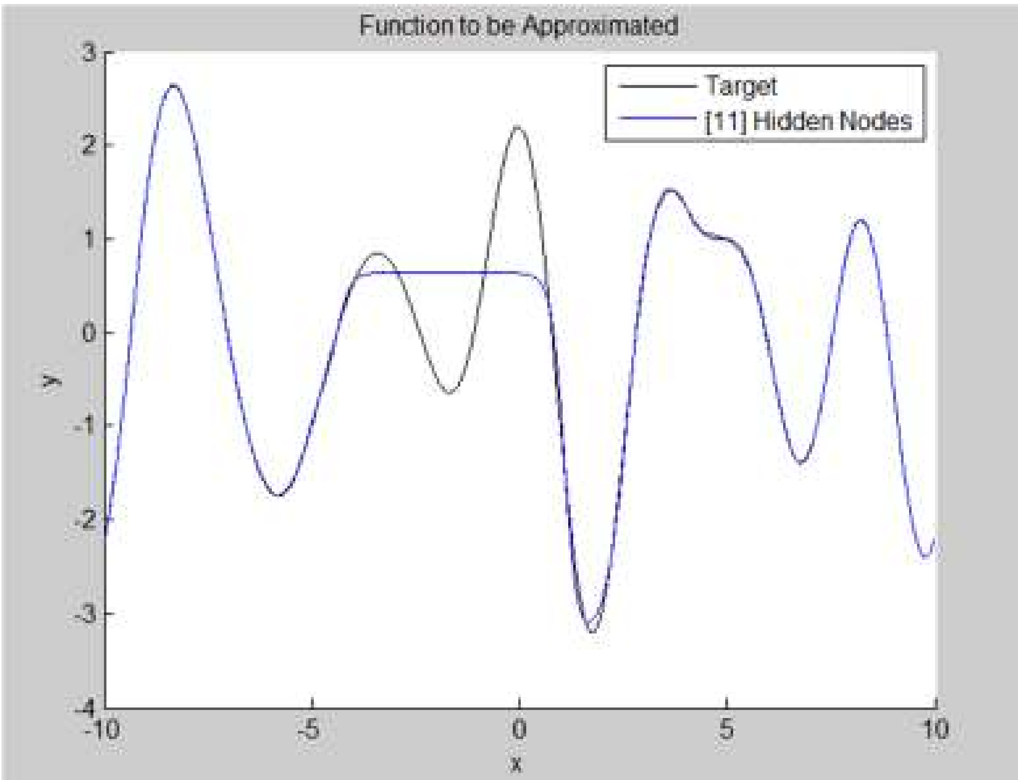
$$\mathcal{N} = (I+1)H_1 + \sum_{\substack{j=1 \\ (J>1)}}^{J-1} \left(H_j + 1\right) H_{j+1} + \left(H_J + 1\right)_{.}$$

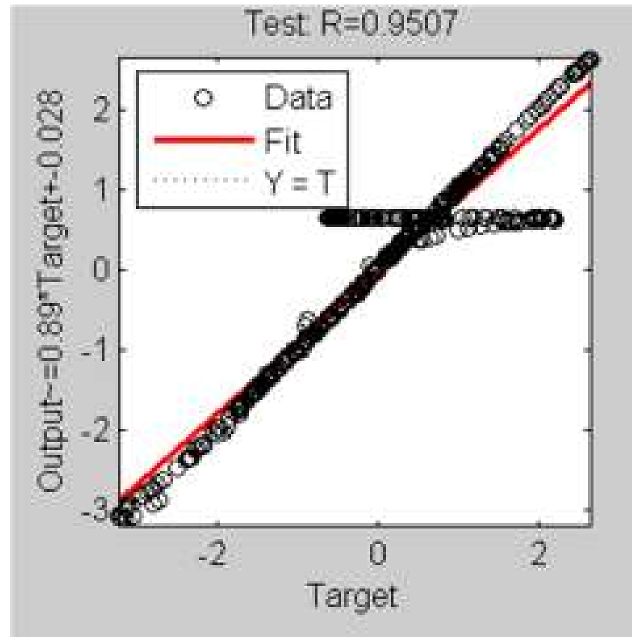
Single vs. Multiple Hidden Layers

- According to this, the [20] architecture above (single hidden layer with 20 nodes) had $\mathcal{N} = 61$, while the [6+3] architecture (two hidden layers with 6 and 3 nodes, respectively) had $\mathcal{N} = 37$, and the [4+4] architecture had $\mathcal{N} = 33$.
- All of these architectures worked quite well with correlations very close to 1 (full approximation).
- However, they have very different numbers of degrees of freedom (weights).
- On the other hand, the [10] architecture had $\mathcal{N} = 31$. It didn't work quite as well, but had a similar $\mathcal{N}$ to the double-layer architectures.
- Notice that if we had used an [11] architecture, we would have $\mathcal{N} = 34$, which is close to the $\mathcal{N}$ for [4,4]. Let's try this…

Single vs. Multiple Hidden Layers

- Now Consider [11] architecture (one HL with 11 nodes):



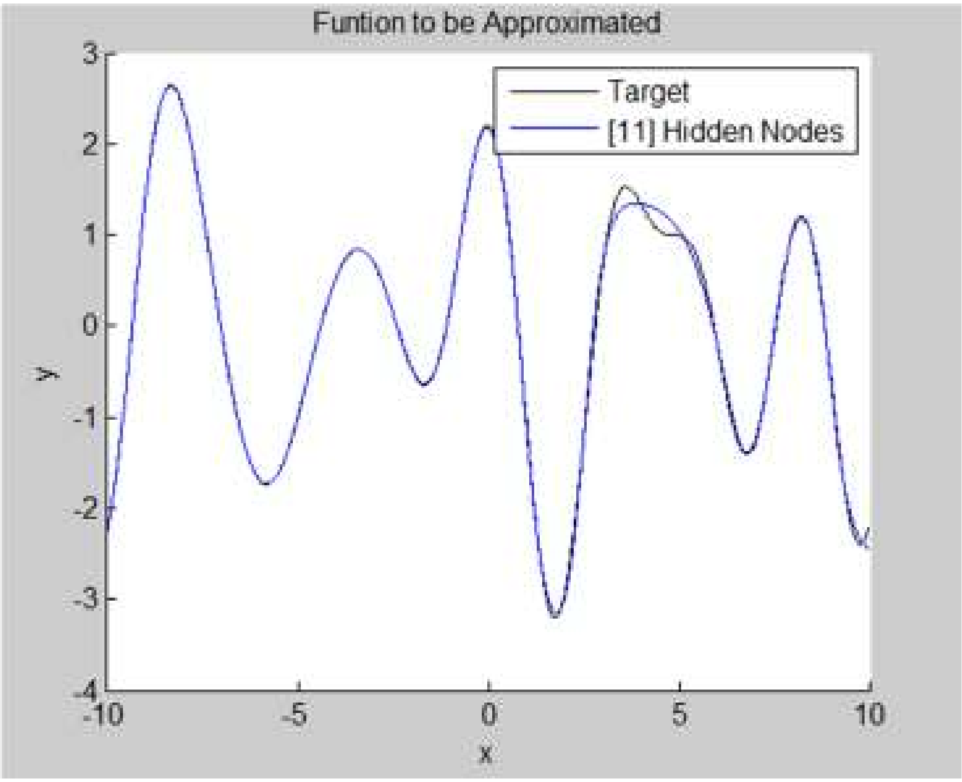


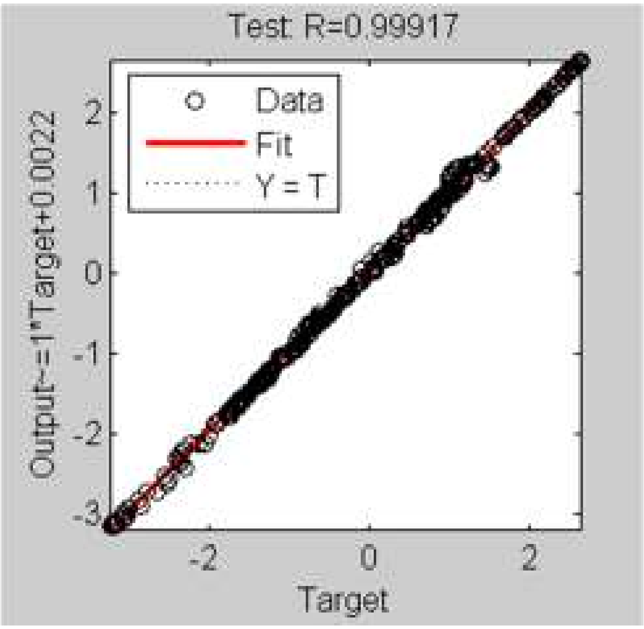
[11] Hidden Nodes

- Not quite as good as [4+4] despite similar number of weights. Why?

Single vs. Multiple Hidden Layers

- Either:
  - Multiple HLs yield more parsimonious architecture, or
  - Stuck in some local minimum.
- Retried several times starting from different initial conditions and found similar performance until hit on a better trained network:





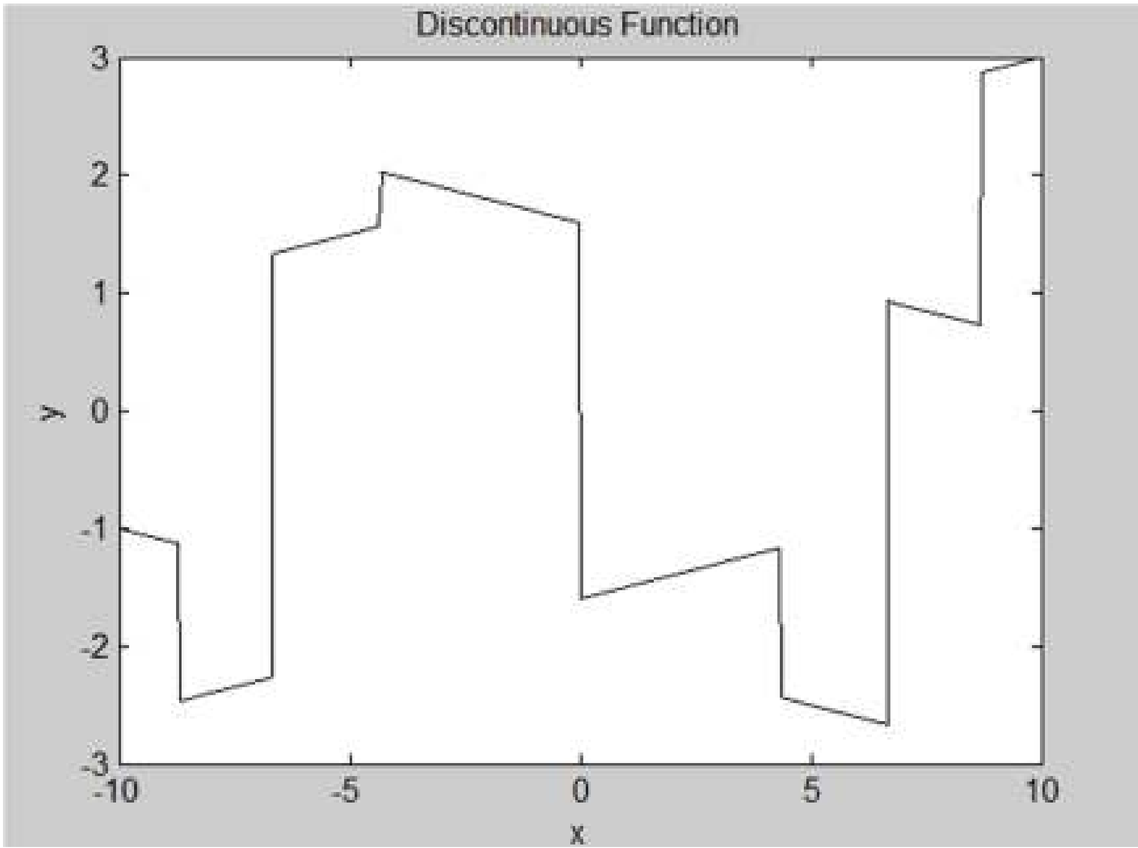
[11] Hidden Nodes

Single vs. Multiple Hidden Layers

- The [4+4] and [11] architectures yielded similar results (with [4+4] being slightly better, but had to try several times with the [11] architecture.
- This offers some empirical evidence that multiple-hidden-layer architectures are equivalent or slightly better than single-hidden-layer architectures (*after adjusting for the overall number of degrees of freedom*), but that perhaps multiple-hidden-layer architectures are more stable.
- A good research project would be to show this (theoretically or empirically) for a general class of functions. Empirically one would have to do many runs and report average performance (a Monte Carlo type study).

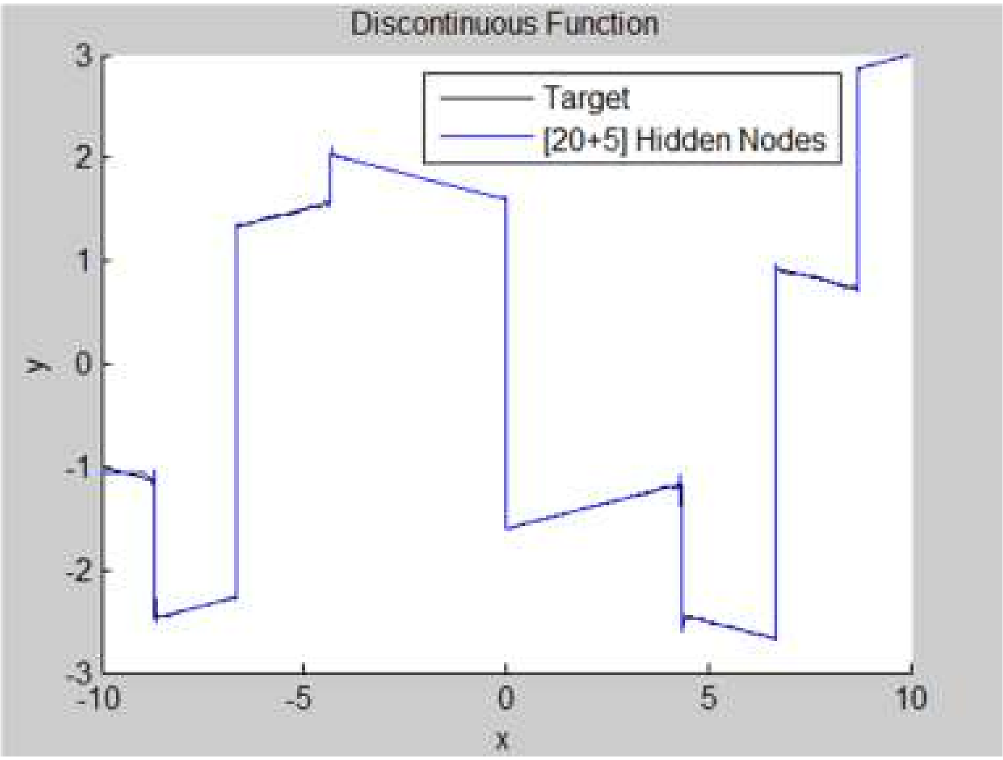
Discontinuous Functions

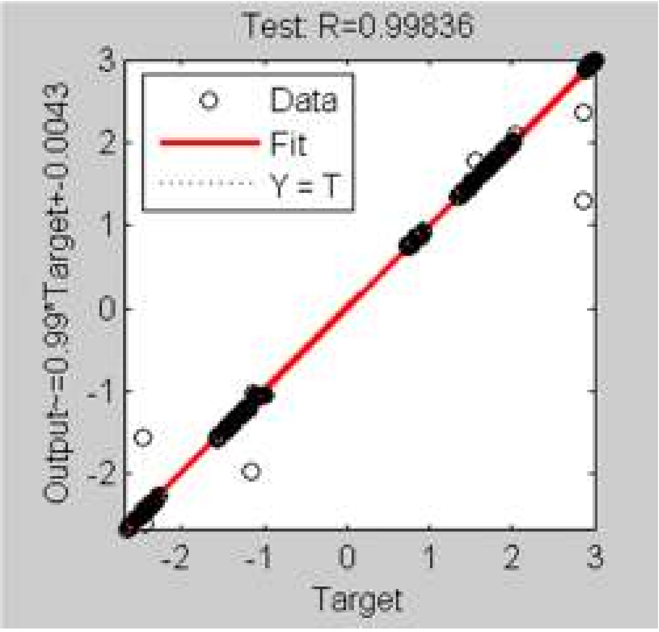
- FFNNs should be able to approximate continuous functions.
- Consider:



Discontinuous Functions

- A [20+5] architecture approximates a discontinuous function very well.
- Other architectures are also possible.





[20+5] Hidden Nodes

Noise

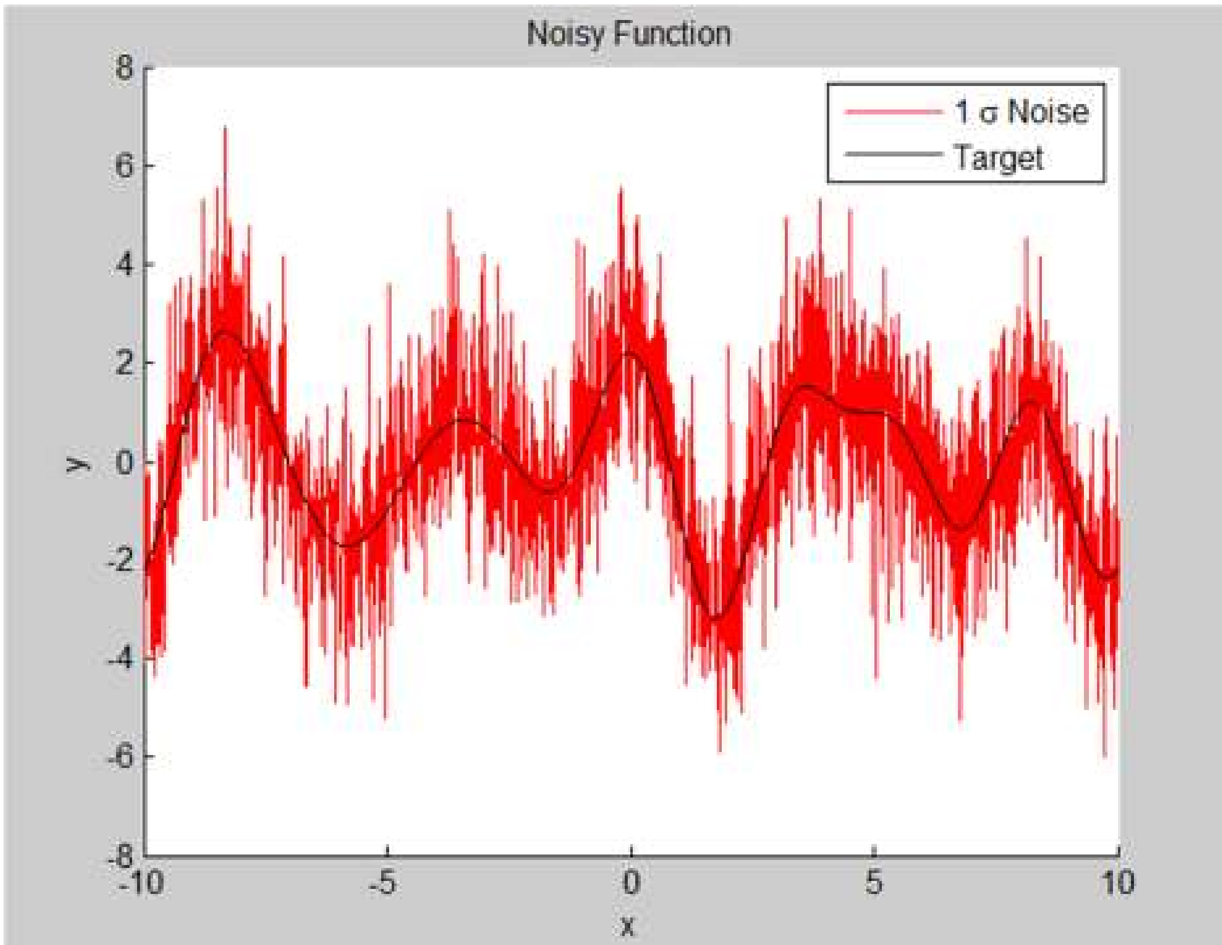
- We have seen that any function, even highly-nonlinear and even discontinuous ones can be approximated arbitrarily closely if we have enough hidden-layer nodes (*i.e.*, enough internal parameters or weights).
- In real-world applications most target functions are corrupted by a significant amount of noise (particularly with financial data).
- Consider corrupting our test function with Gaussian Noise:

$$Y_i^* = Y_i + \varepsilon_i,$$

where $\varepsilon_i \sim \mathcal{N}(0, n\sigma_Y)$, and n is the *__Noise Strength__* ($1/n$ is the *__Signal-to-Noise__* ratio).

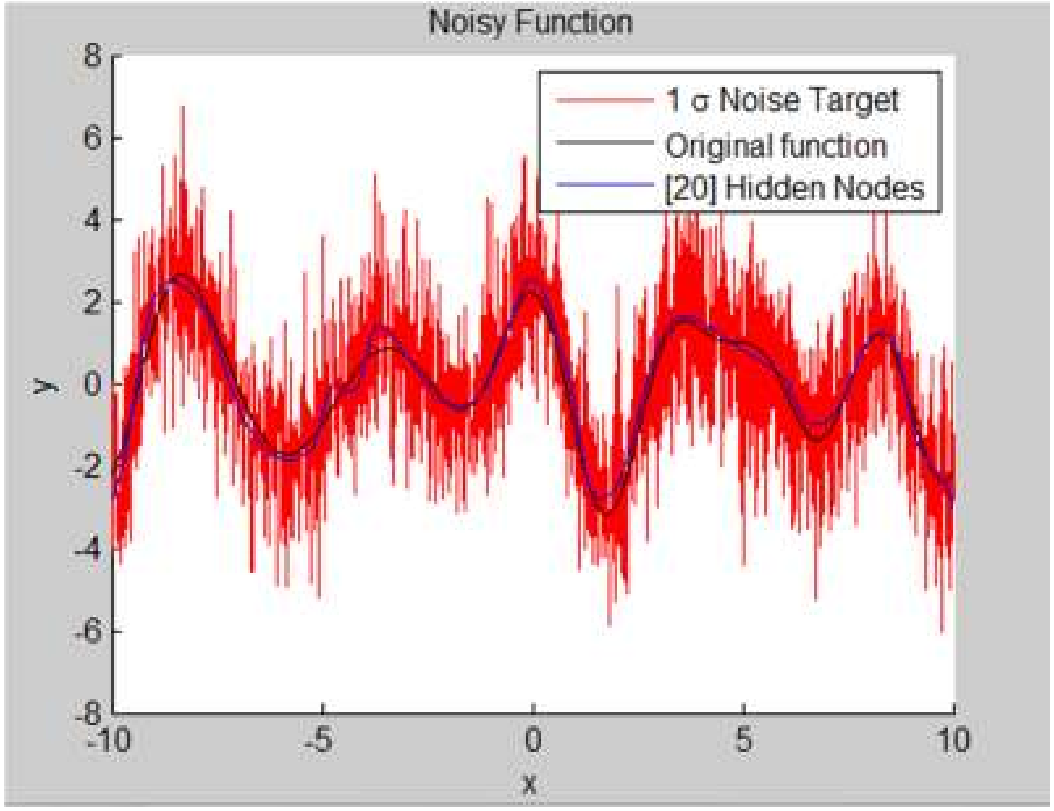
Noise

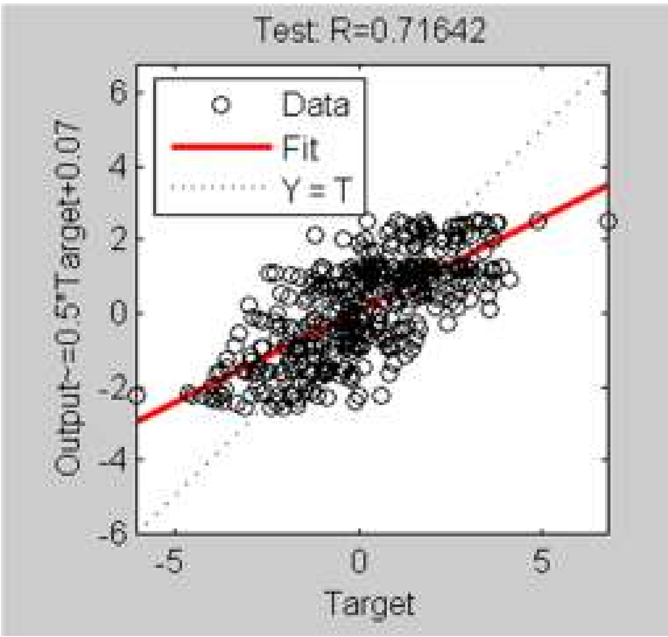
- Noise Strength of 1 Standard Dev:



Noise

- A [20] FFNN fitted on the 1 Std Dev Noisy Data (red) approximates the original function (black) quite well.
- Regression curve is now more scattered, but has reached the theoretically maximum performance…





[20] Hidden Nodes

Theoretically Maximum Performance

- We saw before that for non-noisy targets the R^2 can reach 1.
- For noisy targets the situation changes and the R^2 will be less than 1.
- We can, in fact, derive an expression for the theoretically maximum R^2 for a given level of noise.

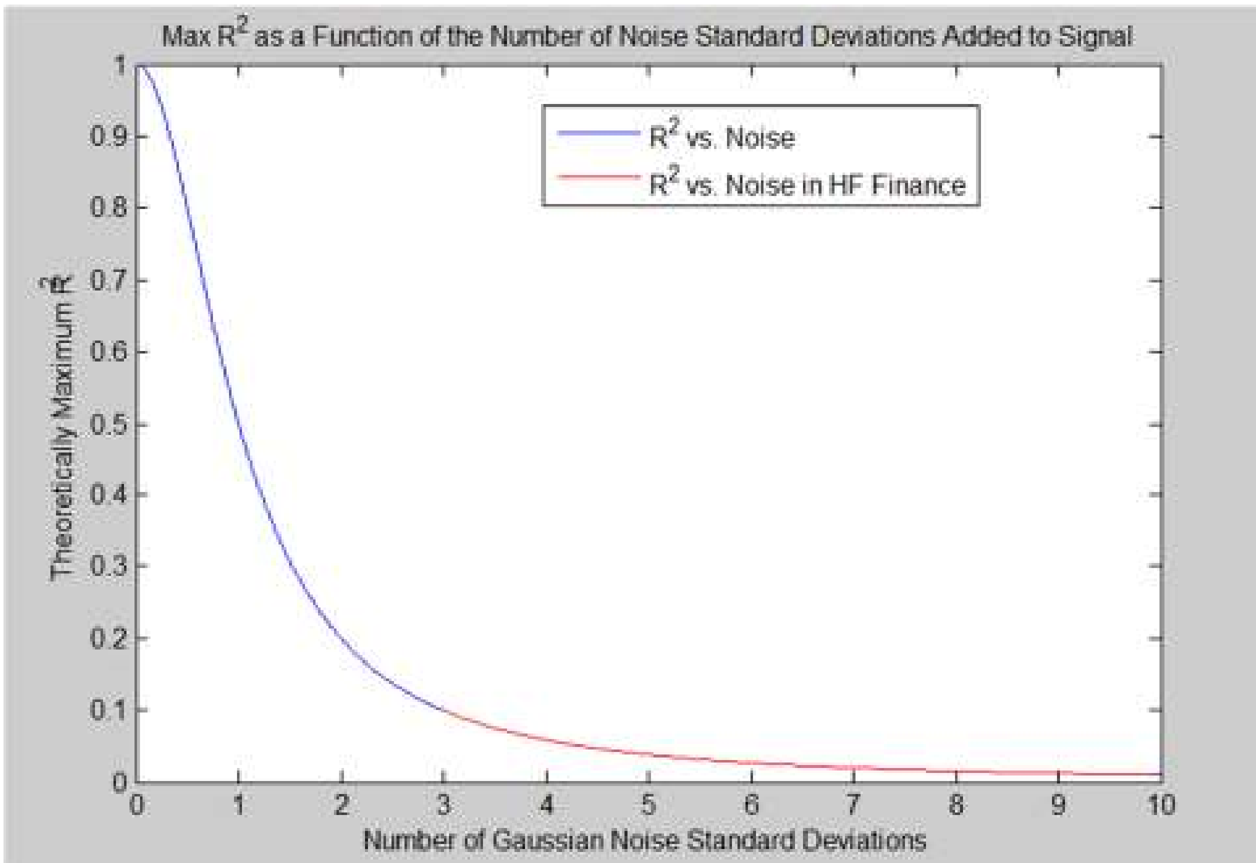
$$maxR^2 = 1 - \frac{\sum_i \left(Y_i{}^* - Y_i\right)^2}{\sum_i \left(Y_i{}^* - \bar{Y}_i\right)^2},$$

from which it follows that $maxR^2 = 1 - \frac{n^2}{1+n^2}$.

- For n=1, the maximum R^2 is 0.5, which is precisely what was reached by the [20] network above (R=.71).

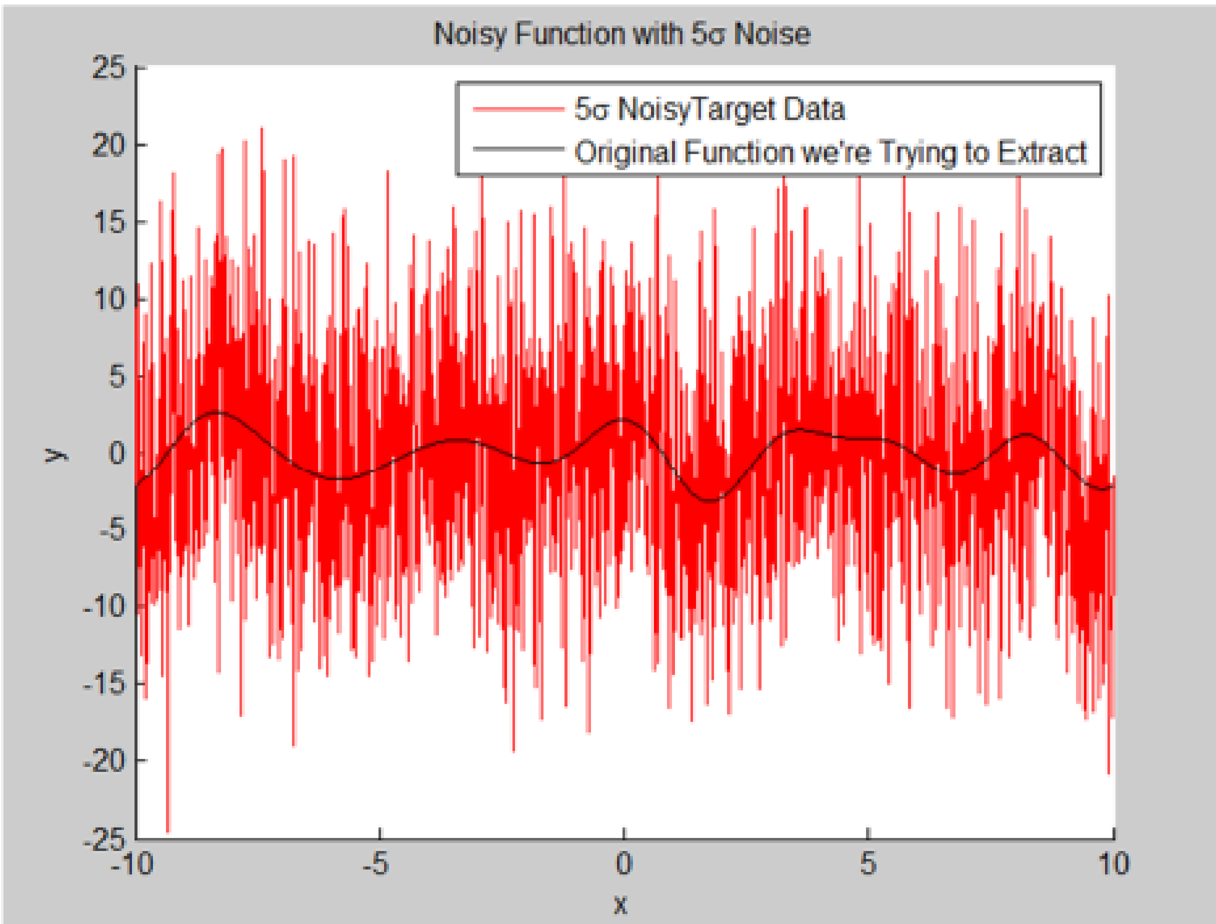
Theoretically Maximum Performance

- In HF Finance R^2s typically hover in the single digits.
- This means that, assuming our models capture most of the R^2, the signal-to-noise ratios $(1/n)$ are around 0.1-0.3 .



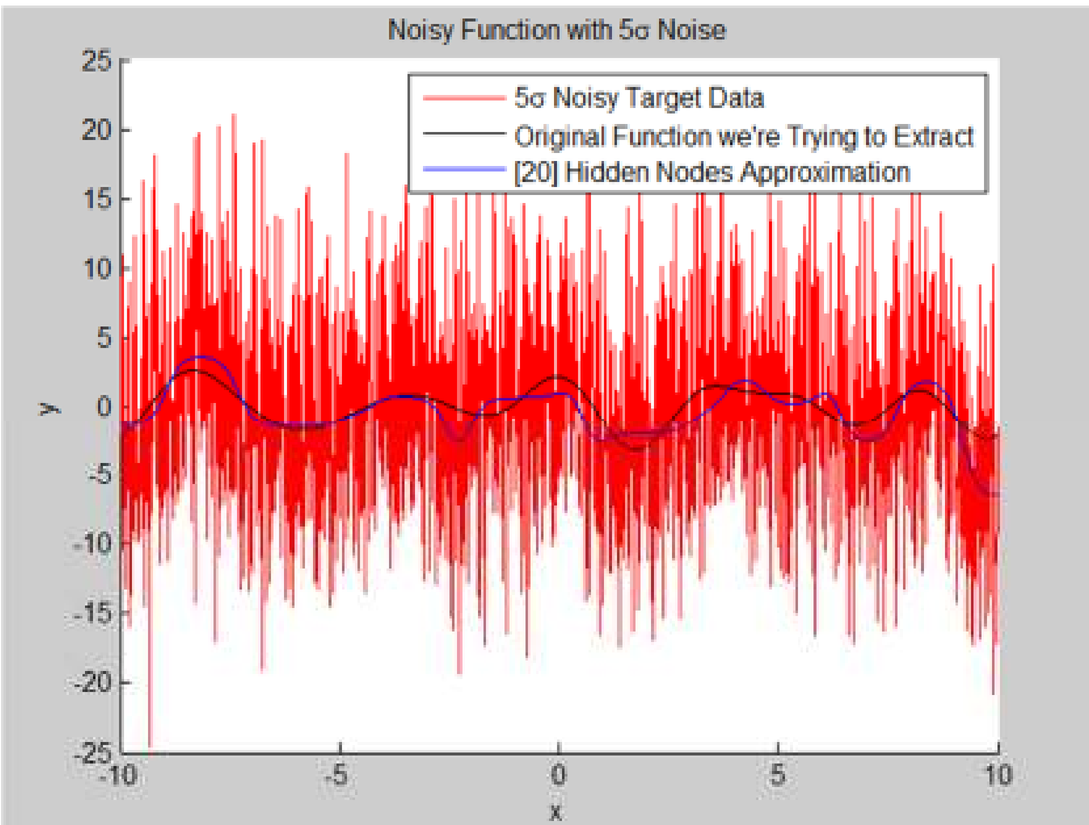
Noise

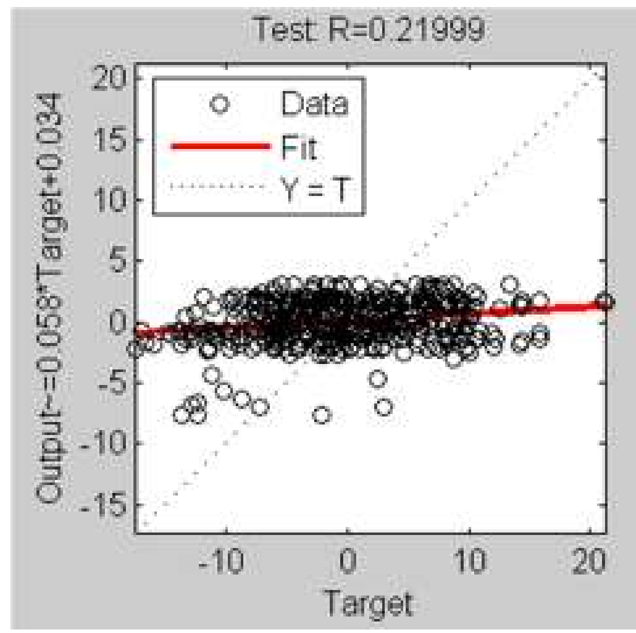
- A more realistic noise strength is n=5 (5σ).
- Theoretically Maximum $R^2 = 0.039$.



Noise

- Noise strength is n=5 (5σ).
- Theoretically Maximum $R^2 = 0.039$ ($maxR = 0.20$).
- Slight overfit with [20] Hidden Nodes or artifact of data size?

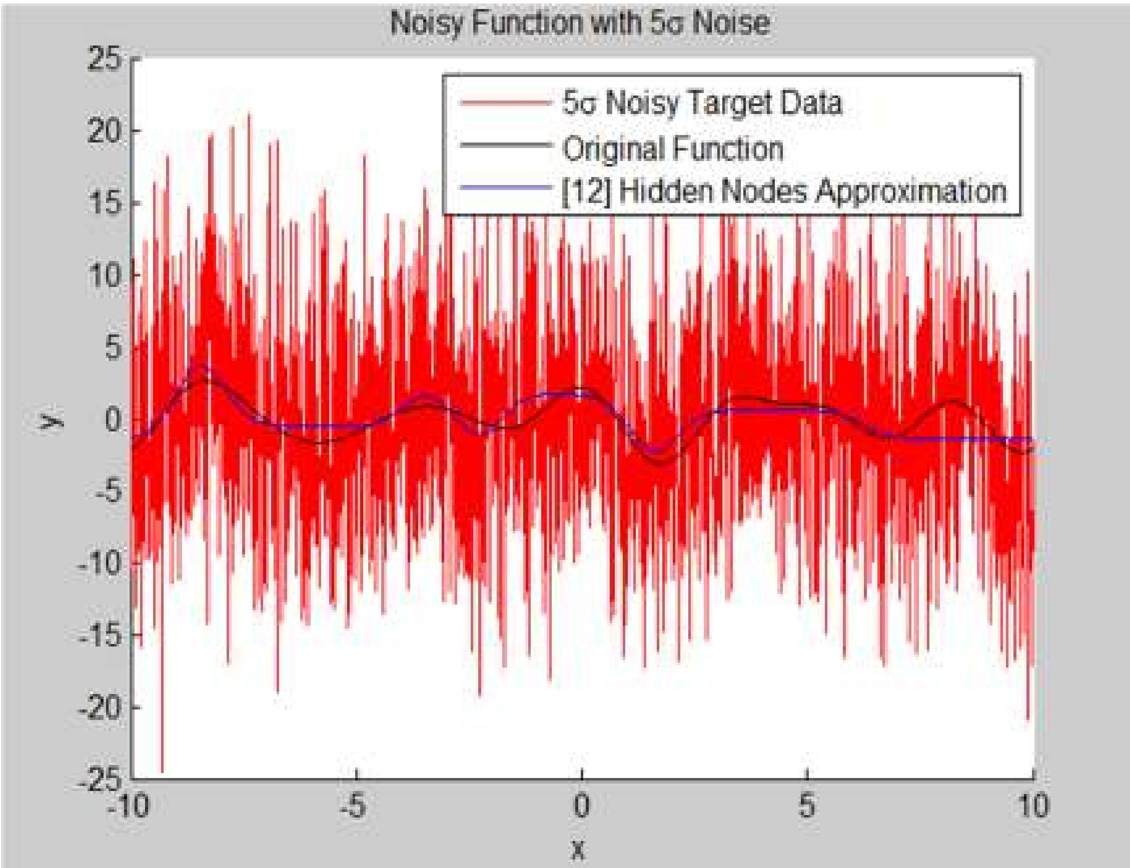


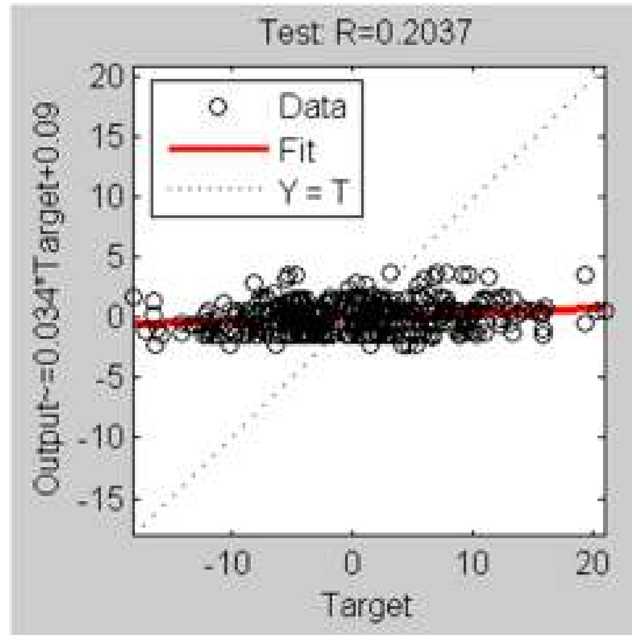


[20] Hidden Nodes

Noise

- Noise strength is n=5 (5σ).
- Theoretically Maximum $R^2 = 0.039$ $(maxR = 0.20)$.
- [12] Hidden Nodes maybe slightly better:





[12] Hidden Nodes

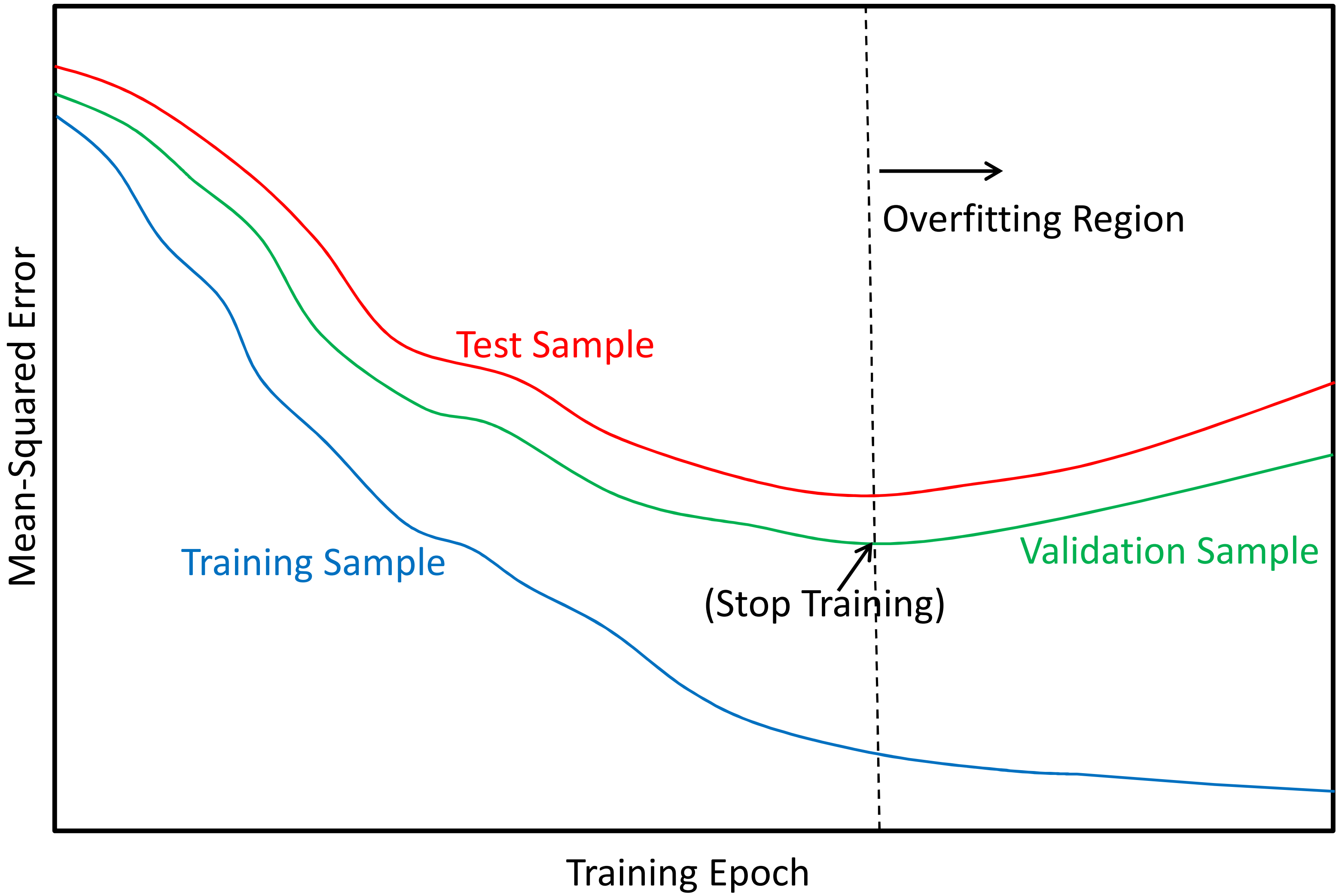
Overfitting

- Compounded by too many degrees of freedom (parameters, weights), too few data points.
- Leads to poor *Generalization*, the model's ability to perform well when presented with data that it has never "seen" before (i.e., data that was not used during the fitting or training of the model).
- Balancing act: The model should be as complex as needed, but no more (*Parsimony*).
- We don't have the original function (black curve above) available.
- One way to get around overfitting is to start with a relatively large model (number of hidden nodes), split the data into *Training* and *Validation* samples, and use the validation sample to stop the training. This is the method of *Early Stopping* or *Cross Validation*.
- Model performance stats such as R^2 are then reported on yet another data sample called the *Testing* or *Hold-Out* sample, which was not used for either training or stopping training.
- Note that Validation and Testing are often used interchangeably in nomenclature, but they are different.

(Aside: Data Mining)…

- In fields outside of Finance "*Data Mining*" is a perfectly respectable phrase referring to the analysis and extraction of information out of (typically large) corpuses of data.
- In Finance (particularly HFT), Data Mining is a "dirty word", and it refers to finding patterns that do not persist in the data (unlike in other fields such as bioinformatics, physics, marketing, etc.).
- This is likely due to the small predictable signals present in market data since any predictable patterns are typically arbitraged away quickly.
- Typically one needs a great deal of data to make robust predictions, but there is a tradeoff between amount of data and how far back to go in the data (staleness) to dig for patterns.
- For example, in a random Head/Tail data set: HTHTHHTHHT one may be tempted to find a pattern such as "Heads are more likely after a Tail," which is an artifact of the smallness of the sample.

Early Stopping



- Stop Training when Validation Error reaches a minimum.
- For a large model, Training Error usually continues to decrease past this point.

Regularization

- Early Stopping doesn't directly modify the FFNN architecture.
- Early Stopping doesn't use all data for training.
- *Regularization:* Avoids overfitting by penalizing complexity.
- Pruning, Penalty for Complexity, Bayesian Methods.

Regularization: Pruning

- Remove weights that are "small":
  - If $|w_{ij}| < \delta$ then set $w_{ij} \triangleq 0$;
  - Retrain Network;
  - Repeat until no more "small" weights.
- Jiggle weights and remove those who have "small" impact on the output (or error)—*Sensitivity*:
  - If $\left|\frac{\partial output}{\partial w_{ij}}\right| < \delta$ then set $w_{ij} \triangleq 0$;
  - Retrain Network
  - Repeat until no more "small" Sensitivities left.

Regularization: Pruning

- Pruning Regularization Methods work reasonably well at producing parsimonious methods.
- By setting weights to zero, they adjust the Network’s architecture.
- We can re-train with fewer nodes to reflect the number of weights left after pruning.
- Can be used in combination with Early Stopping (if have enough data).
- Drawbacks:
  - Reliance on “small” parameter δ.
  - Computationally intensive/Time consuming.

Regularization: Penalty

- Penalty function so far has been MSE:

$$\mathcal{F}_W = E = \frac{1}{N}\sum_i \tfrac{1}{2}(t_i - a_i)^2.$$

- Introduce a term to directly penalize complexity by penalizing weight magnitudes:

$$\mathcal{F}_W = (1-\lambda)\frac{1}{N}\sum_i \tfrac{1}{2}(t_i - a_i)^2 + \lambda \sum_{ij} \tfrac{1}{2}{w_{ij}}^2\,.$$

Regularization: Penalty

- Penalty Regularization works reasonably well.
- Can be used with the other methods discussed above (Early Stopping and Pruning).
- Main drawback is its reliance on weight-complexity penalty knob λ.

Annealing Methods

- Start with a “large” architecture.
- From a set of weights (“*state*” of the system) in the current iteration, $\boldsymbol{w}(m)$, obtain new weights $\boldsymbol{w}(m+1)$ (the new state) by a preferred method (either with or without regularization as above).
- Compute the new penalty function (the “*Free Energy*” which we’re trying to minimize): $\mathcal{F}_W(m+1)$.
- Transition to the new state (*i.e.*, accept the new weights) with probability

$$P \propto exp\left\{-\frac{\mathcal{F}_W(m+1)-\mathcal{F}_W(m)}{T(m)}\right\},$$

where $T(m)$ is the “*Temperature*” which is decayed slowly.

Annealing Methods (cont.)

- Do not transition to the new state (*i.e.*, *reject* the new weights) with probability $1-P$, and corrupt the previous weights $\boldsymbol{w}(m)$ with a small amount of noise dependent on the current Temperature:

$$\boldsymbol{w}(m) \leftarrow \boldsymbol{w}(m) + \varepsilon\big(0, \sigma\big(T(m)\big)\big).$$

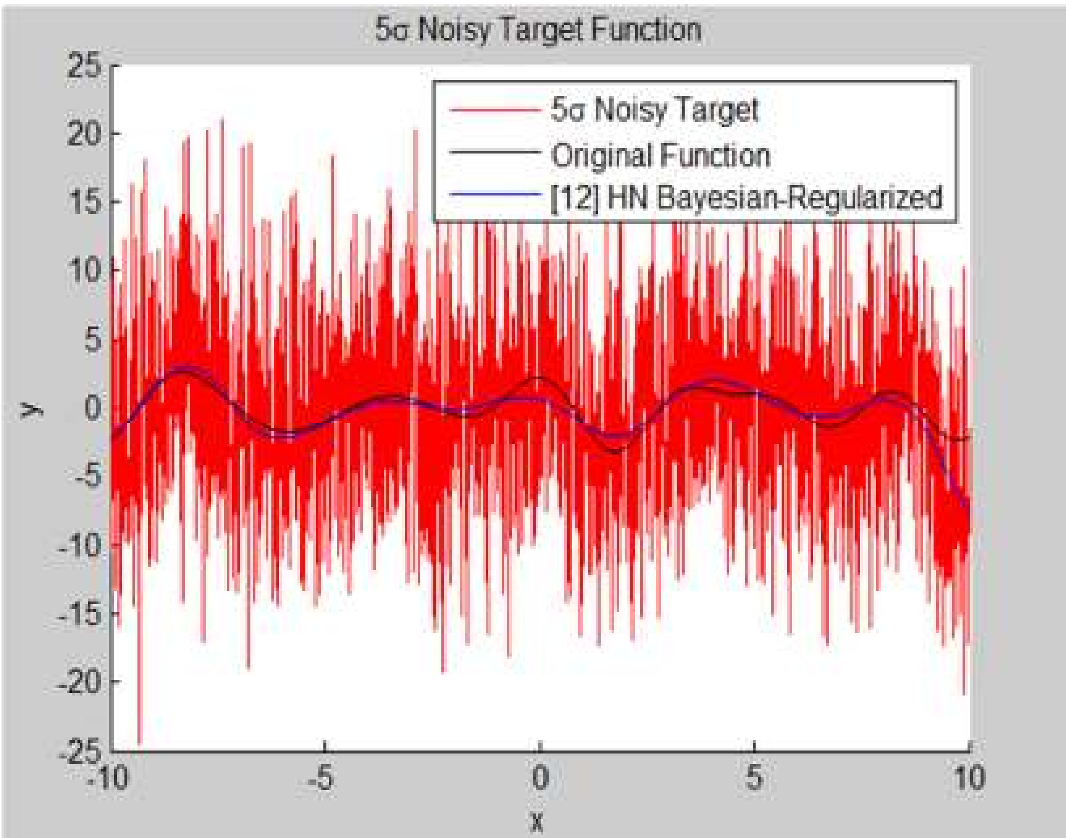
- Decay the Temperature (*i.e.*, the Noise Variance) slowly only if transition to the new state.
- Repeat until Temperature or Penalty is “small”.
- Inspired by metallurgical or ceramic annealing process (slow cooling to form stable alloys/crystals).
- A close cousin to Genetic Algorithms (another variant used with NNs)
- Process converges in probability to global optimum.
- Works well, but:
  - It is slow and computationally intensive.
  - Works only for small to medium size problems.
  - Depends on a host of *ad hoc* parameters.

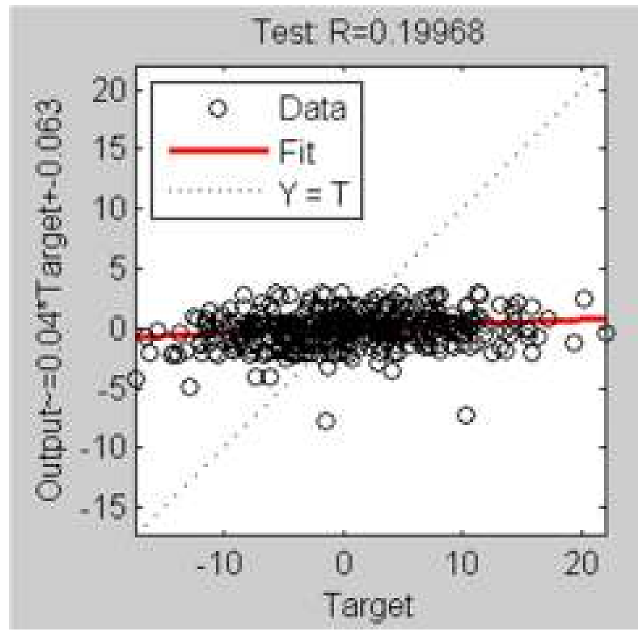
Bayesian Methods

- The previous Regularization and Early-Stopping methods have either a host of *ad hoc* parameters needing heuristics to specify (*i.e.*, lacking objective methods for specifying them), or diminish the overall data size by setting aside Test samples, or are computationally intensive, or all of the above.
- <u>*Bayesian Regularization*</u> is a method that:
  - Introduces objective criteria for regularizing parameters;
  - Introduces objective criteria for comparing among models (including non-neural network approaches);
  - Does not decrease the data size;
  - Bonus: Obtains the <u>*Effective Number of Degrees of Freedom*</u> (weights).
  - This can be used to re-train a new network with a smaller architecture that matches the Effective Number of Weights found above, or to check if a larger network leads to the same Effective Number of Weights, meaning it's unnecessary to go larger.

Bayesian Methods

- Bayesian Regularization produces smoother fit:

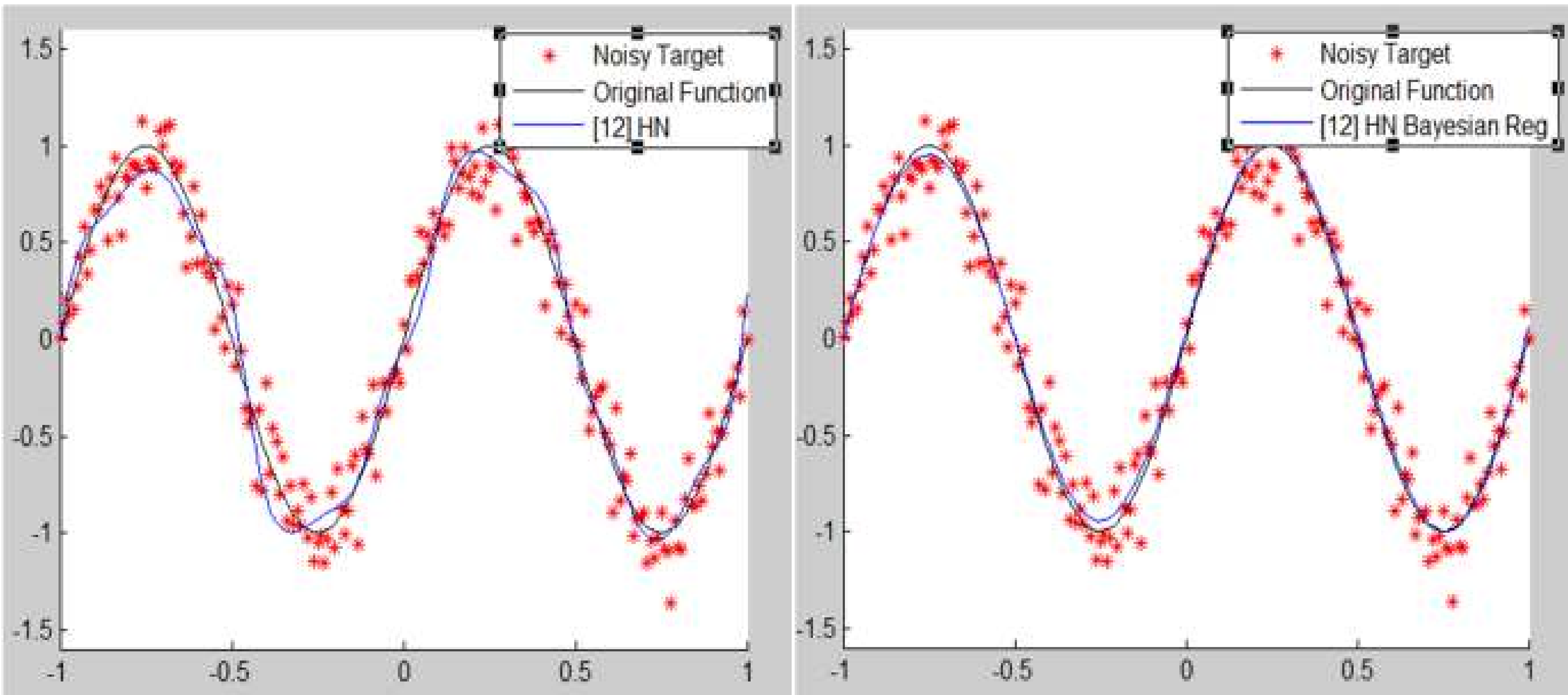




[12] HN, Bayesian-Regularized

Bayesian Methods

- Bayesian Regularization produces smoother fit:



- Num Effective Params went from 37 to 9.
- R^2s comparable (around the max) but a bit higher for BR.
- But fit is smoother with Bayesian Regularization (better Generalization).

Evaluating Classifiers

- Two-class classifiers (*e.g.* 0/1 or True/False) can make Two Types of Mistakes:
  - Type I Error: Assumes False when True;
  - Type II Error: Assumes True when False.
- Usually Type I Errors come at the expense of Type II Errors and vice-versa.
- Tools for evaluating Classifier Performance:
  - Confusion Matrix
  - ROC Curve

(Aside…)

- Biological Brains are far more biased towards Type II Error:
  - Far more likely to detect patterns out of randomness.
  - Far more likely to assume causal agency out of none.
- Cost/benefit analysis shows this logical blind spot likely had adaptive advantage in our evolutionary history; e.g.:
  - A noise in the bushes could be nothing, or a predator; Type II Error costs little, Type I can cost the animal’s life.
  - Recognizing faces can be paramount to human survival, while seeing a face in the clouds costs little …
- May explain superstitions, etc…

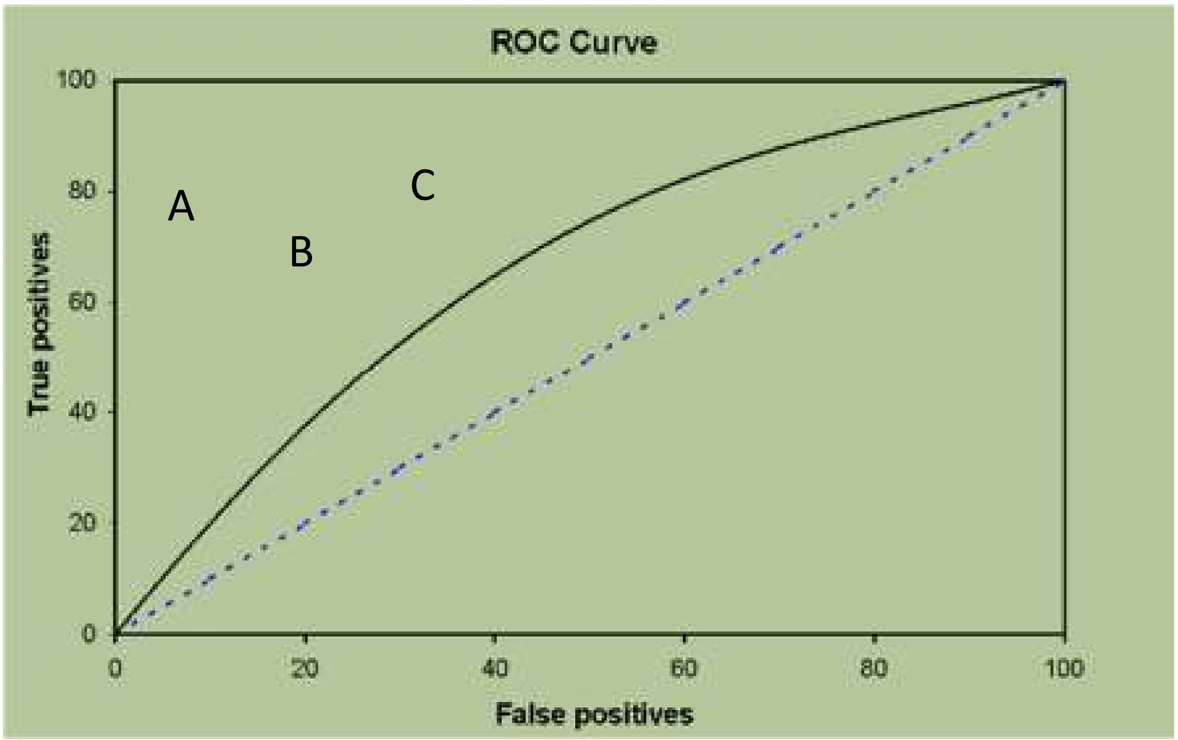
Confusion Matrix

- Quantifies predicted vs. actual classes.
- Quantifies error rates and correct classification rates.

		prediction outcome		
		p	*n*	total
actual value	*p'*	True Positive	False Negative	P'
	n'	False Positive	True Negative	N'
	total	P	N	

ROC Curves

- “Receiver Operating Characteristics” (name from use with radars historically).



- Dashed line is “Random” classifier.
- The more “NorthWest” a classifier is, the better.
- Classifier A is objectively better than B or C, but it’s not clear that B is better than C.
- Can incorporate costs of errors into ROC curves to make them more relevant.