# Time Series Forecasting Using Neural Networks

# Introduction:

A time series [3] is a sequence of [data points](http://en.wikipedia.org/wiki/Data_point), typically consisting of successive measurements made over a time interval. Examples of time series are ocean [tides](http://en.wikipedia.org/wiki/Tides), counts of [sunspots](http://en.wikipedia.org/wiki/Sunspots), and the daily closing value of the [Dow Jones Industrial Average](http://en.wikipedia.org/wiki/Dow_Jones_Industrial_Average). Time series are very frequently plotted via [line charts](http://en.wikipedia.org/wiki/Line_chart). Time series are used in [statistics](http://en.wikipedia.org/wiki/Statistics), [signal processing](http://en.wikipedia.org/wiki/Signal_processing), [pattern recognition](http://en.wikipedia.org/wiki/Pattern_recognition), [econometrics](http://en.wikipedia.org/wiki/Econometrics), [mathematical finance](http://en.wikipedia.org/wiki/Mathematical_finance), [weather forecasting](http://en.wikipedia.org/wiki/Weather_forecasting), [earthquake prediction](http://en.wikipedia.org/wiki/Earthquake_prediction), [electroencephalography](http://en.wikipedia.org/wiki/Electroencephalography), [control engineering](http://en.wikipedia.org/wiki/Control_engineering), [astronomy](http://en.wikipedia.org/wiki/Astronomy), [communications engineering](http://en.wikipedia.org/wiki/Communications_engineering), and largely in any domain of applied [science](http://en.wikipedia.org/wiki/Applied_science) and [engineering](http://en.wikipedia.org/wiki/Engineering) which involves [temporal](http://en.wikipedia.org/wiki/Time) measurements.

Time series analysis comprises methods for analysing time series data in order to extract meaningful statistics and other characteristics of the data. Time series forecasting is the use of a [model](http://en.wikipedia.org/wiki/Model_%28abstract%29) to predict future values based on previously observed values. While [regression analysis](http://en.wikipedia.org/wiki/Regression_analysis) is often employed in such a way as to test theories that the current values of one or more independent time series affect the current value of another time series, this type of analysis of time series is not called "time series analysis", which focuses on comparing values of a single time series or multiple dependent time series at different points in time.

I will be concentrating here on the Neural Networks for the time series forecasting. I will use 3 different methods.

1. First Order Neural Networks with no regularization.
2. First Order Neural Networks with regularization.
3. Second Order Neural Networks using LM.

# Process:

As mentioned above I have used 3 different neural network algorithms to train the time series. I have not used any early stopping as I was looking to see how it would behave. Also, I played with number of hidden layers, learning rate and the weight decay parameters (for regularization). This is a single hidden layer (with multiple neurons) fully connected network. I have used python as the programming language. Also, while predicting future value for time series I have used actual values till that day and then predicted on next day. I could have done it other way as well e.g. I could have predicted say for Jan 1 2015 based on data till Dec 31 2014 and then for predicting Jan 2 2015, I could have been used the predicted value of Jan 1 2015 rather than actual value of the Jan 1 2015. It also looks like that there is no noise in the data and that is why test and train data seems to fit very well.

I have not used any biases either for the hidden layer or for output layers. I will be presenting only selected graphs here but you can look all the graphs in github or in attached directory. These documents are. “Output\_figures.pdf” This has all the first order graphs with and without regularization. “Output\_figures\_second\_order.pdf” This has all the second order graphs. The code is quite flexible and you can change the different parameters like number of hidden layers, training examples, number of previous values to determine future value of time series, learning eta and smoothing parameters.

## Network:

# My network is fully connected network without any bias. You can add bias to hidden layers of outer layers by just passing the bs and out\_bs values as 1.0 instead of 0.0. Number of inputs is 10 but it can be varied. Number of hidden layers is varied and I tried values between 2 and 10. Output layer is just single node and it is fixed and it cannot be changed.

## First Order Neural Networks with no regularization:

Background:

See 8.

MLP became applicable on practical tasks after the discovery of a supervised training algorithm for learning their weights, this is the back propagation learning algorithm. The back propagation algorithm for training multilayer neural networks is a generalization of the LMS training procedure for nonlinear logistic outputs. As with the LMS procedure, training is iterative with the weights adjusted after the presentation of each example.

The error back propagation algorithm includes two passes through the network:   
- forward pass and   
- backward pass.

During the backward pass the weights are adjusted in accordance with the error correction rule. It suggests that the actual network output is subtracted from the given output in the example. The weights are adjusted so as to make the network output more closer to the desired one.

The backpropagation algorithm does gradient descent as it moves in direction opposite to the gradient of the error that is in direction of the steepest decrease of the error. This is the direction of most rapid error decrease by varying all the weights simultaneously in proportion of how much good is done by the individual changes:

∇ E( w ) = [ ∂E/∂w0, ∂E/∂w1, …, ∂ E/∂ wd ]

This is the gradient descent learning strategy that requires a continuous activation function at the nodes of the MLP network.

The backpropagation training algorithm uses the gradient search technique to minimize a cost function equal to the mean square difference between the desired and actual net outputs. The network is trained initially selecting small random weights and then presenting all training data incrementally. Weights are adjusted after every trial using side information specifying the correct class until weights converge and the cost function is reduced to an acceptable value.

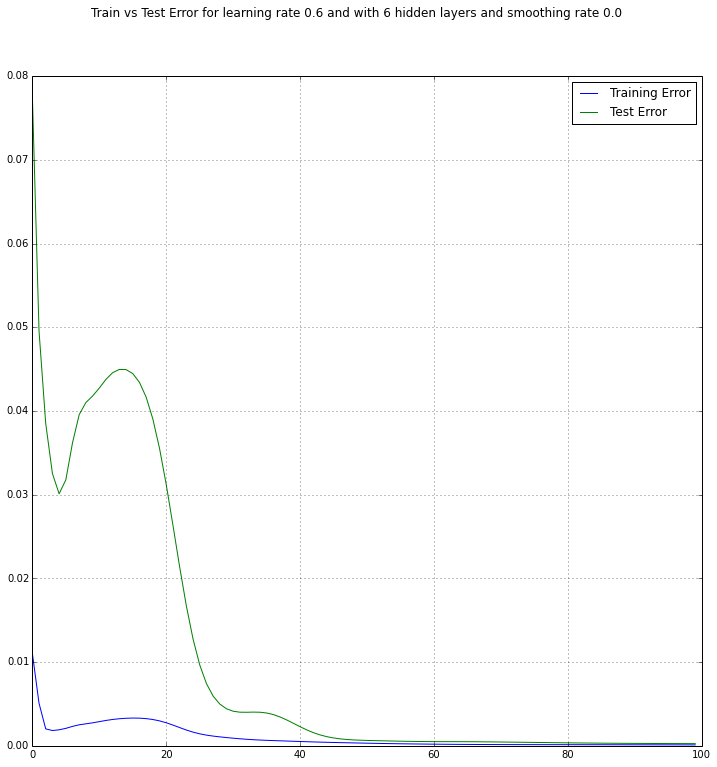
See [8] for more details on backpropagation and derivation of the weight update formulas and algorithm.

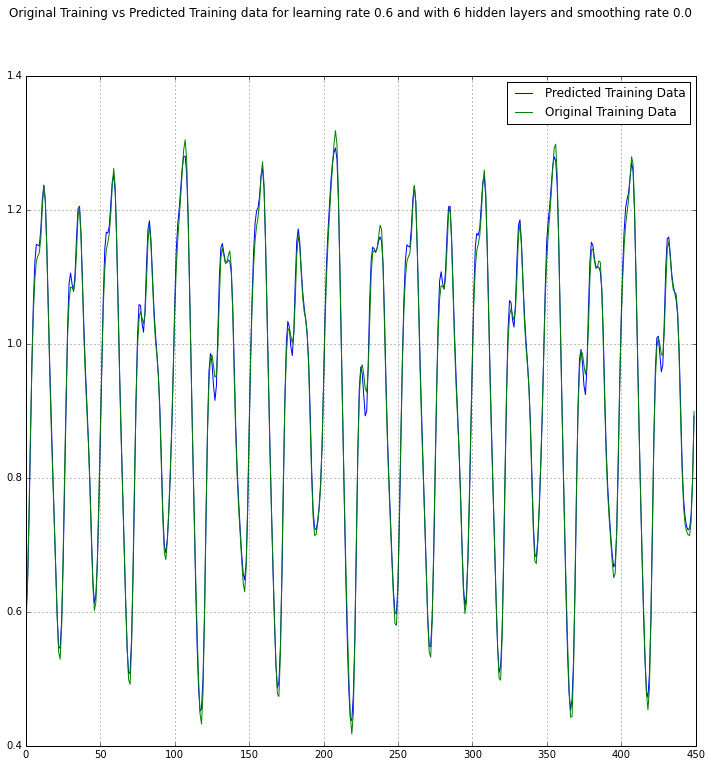
Code:

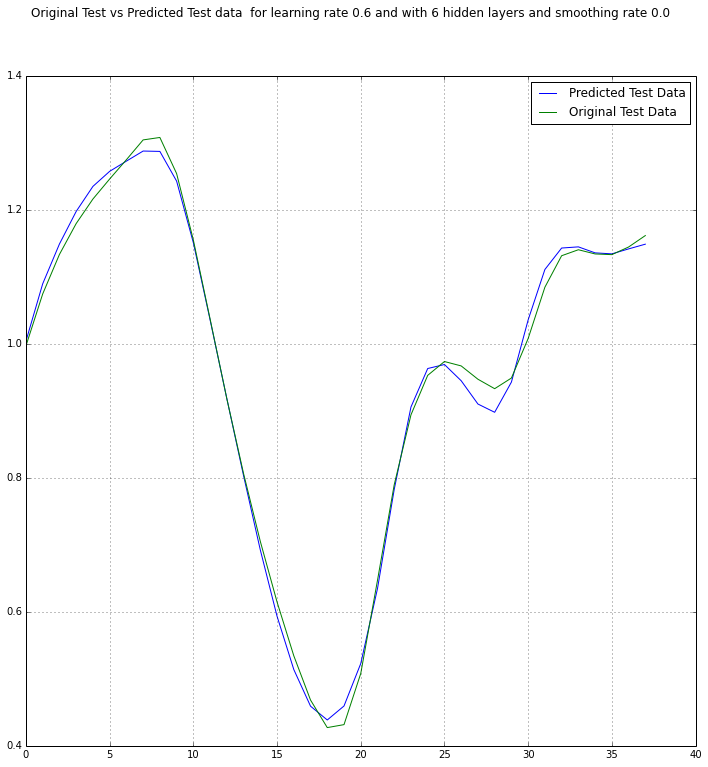
The code for this is in the file NN.py and the TS\_first\_order.py. This one is generic. Passing wgt\_decay as 0.0 will make sure that no regularisation is being used. I have run it for different number of hidden layers and different values of learning eta. Below shown are some graphs.

Graphs:

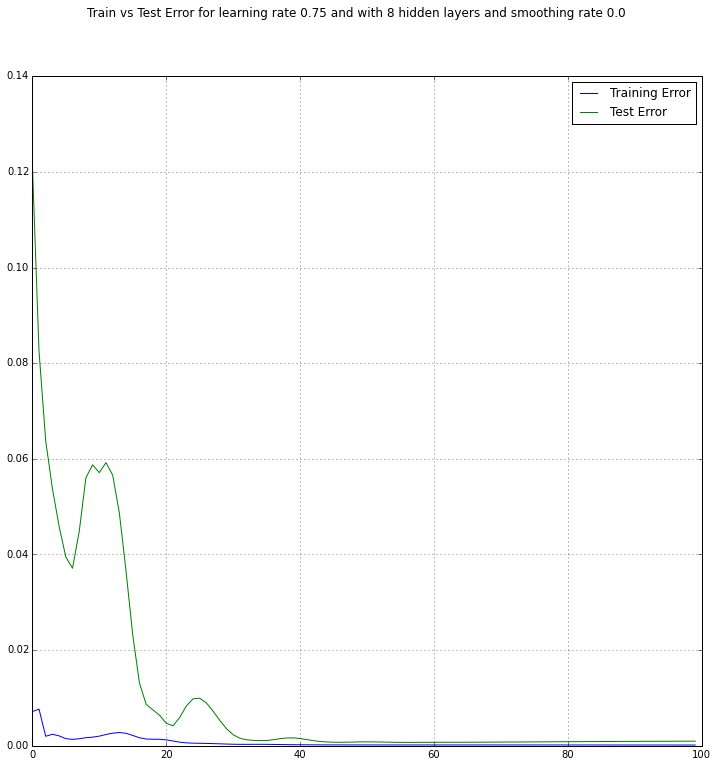
Below is a graph showing 3 hidden neurons and smoothing rate of 0.0 (no regularization) and a learning rate of 0.6. Around epoch 50 the error came to its minimal and the fit is not so great but it is quite good fit even on test data.

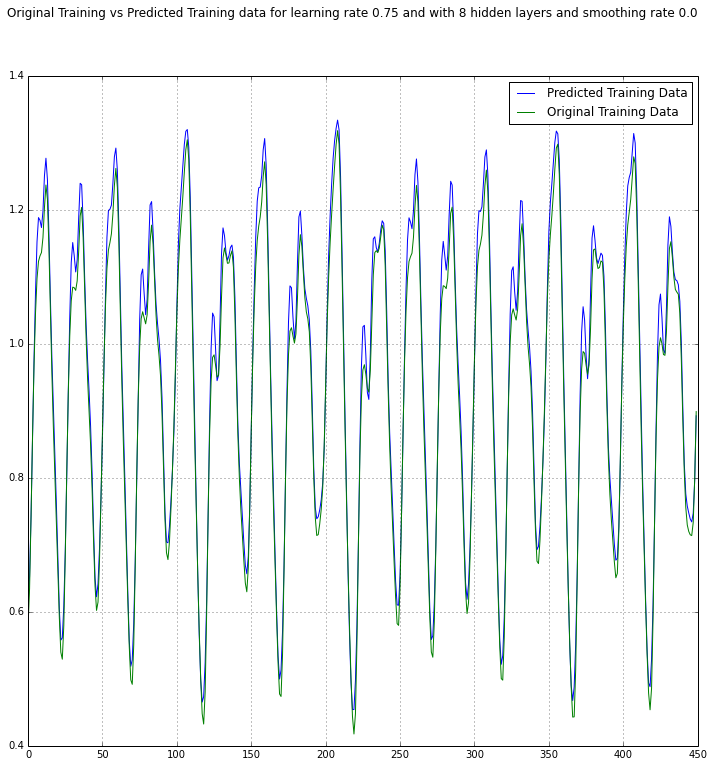


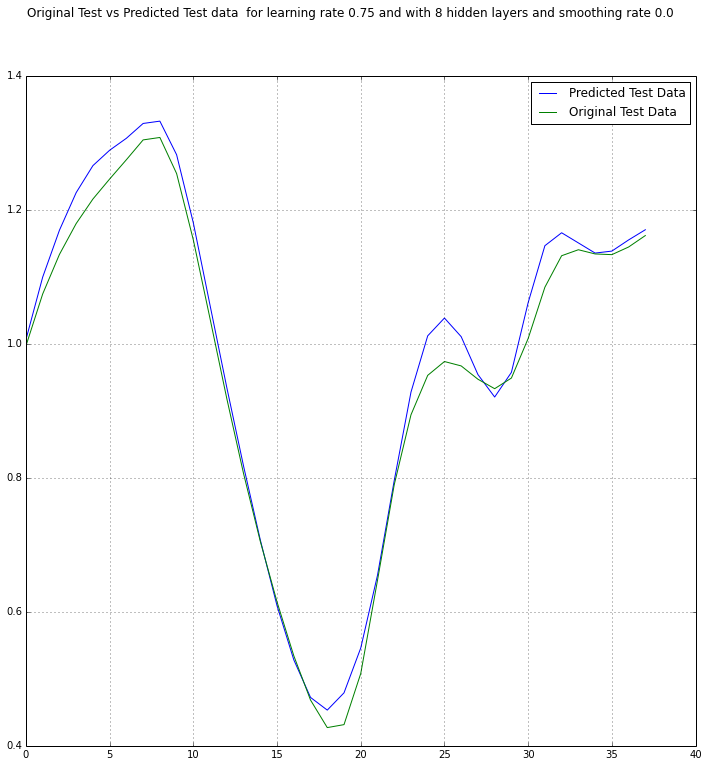




Now I will show with 8 hidden layers and learning rate of 0.75. Here again the fit is quite good and training data seems to over fit but the test data fit is quite good as well. This might be because there is no noise in the data. The error was at its minimal around 50th epoch.







## First Order Neural Networks with regularization:

Background:

The risk of over fitting the examples could be minimized if the variance factor is used in the error function to penalize neural network models with high curvatures. For this reason a weight decay factor that stimulates learning of low-magnitude weights is accommodated in the error making it a regularized average error RAE:

RAE = ( 1 / N ) ( ∑i=1N ( yi - f( xi ))2 + k ∑j=1M wj2 )

where k is a regularization parameter, wj are the network weights, and M is the number of all weights in the network.

The regularization is a roughness penalty since small magnitude weights imply a more ``regular'' approximation. A choice of k=0 favours network function surfaces interpolating the example points tightly, while a large k favours flat function surfaces.

The choice of proper values for the regularization parameter k is subtle since it determines the degree of fitting the provided training examples and governs the amount of smoothing.

Selecting a Proper Regularization Parameter

Proper values for the regularization parameter k may be selected from a certain interval relying on a proof that as long as k lies within this interval:

0 < k < 2s2 / wTw

mean squared error of the RAE approximator is smaller than that of the best least squares approximator without regularization.

As an unbiased estimator of s2 the residual error over the training examples can be used.

Network Learning with Weight Decay

In case of gradient descent learning the update of weights will be used based on below formula.

w( t ) = ( 1 - ηk )w( t-1 ) - η∂Ee/∂w( t-1 ).

Here K is a tuning parameter and proper value for this parameter needs to be validated on test data.

Thus I will be varying K to smoothen the surface. However, I will try ranges from 0.0001 to 0.0005.

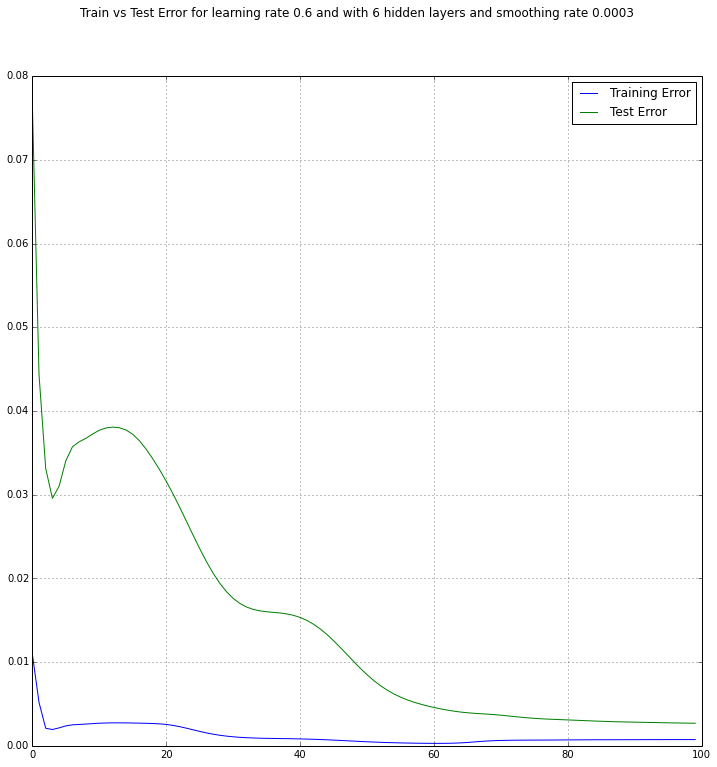
See [7] for more details on other type of reguarizations.

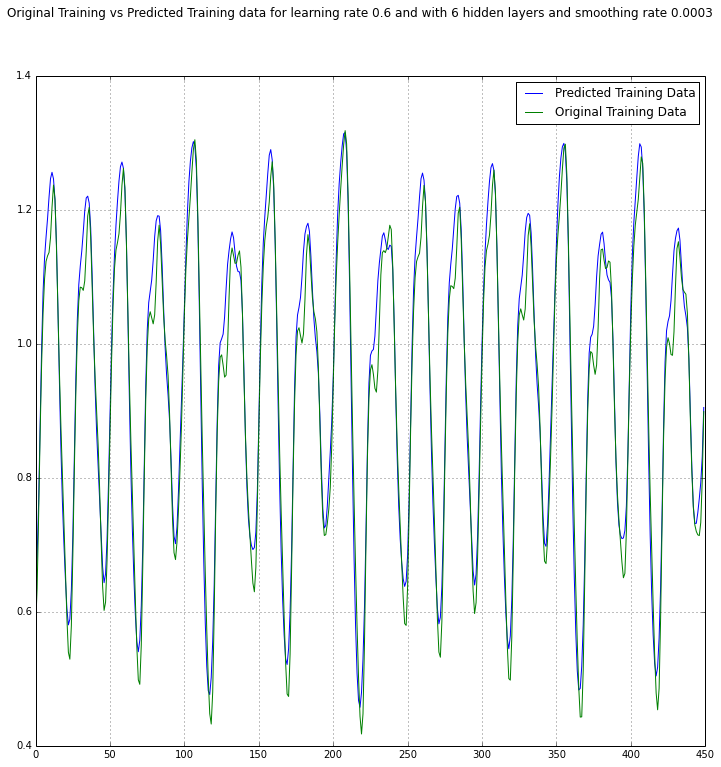
Code:

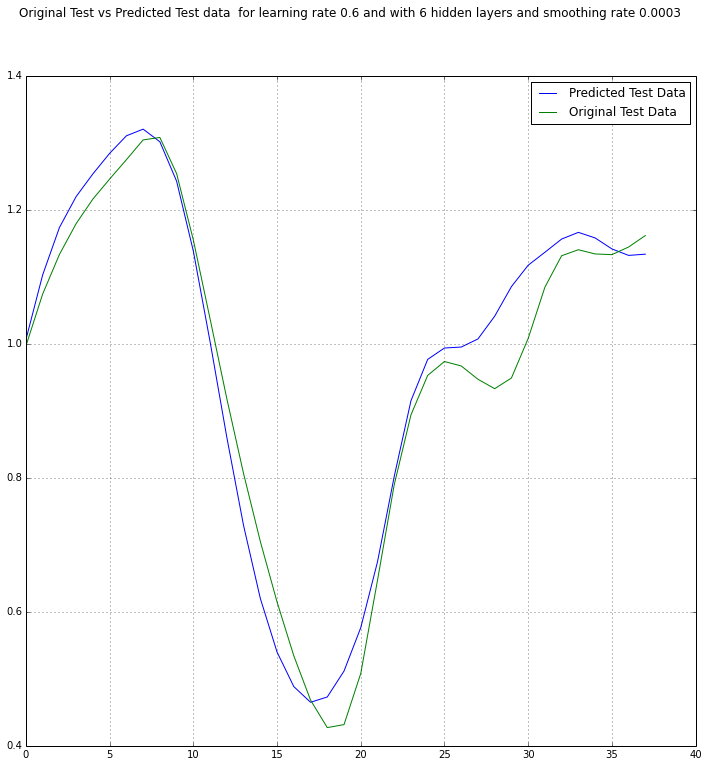
The code for this is in the file **NN.py** and the **TS\_first\_order.py**. This one is generic. Passing wgt\_decay greater than 0.0 will make sure that regularisation is being used. I have run it for different number of hidden layers and different values of learning eta an different values of weight decays.. Below shown are some graphs. Usually regularization is quite good when there is some noise in the data. The data is quite fine and there doesn’t seems to be any noise and thus any regularization will make curves more smoother and thus will make the data fit less well as compared to unregularized in this data set.

*Graphs:*

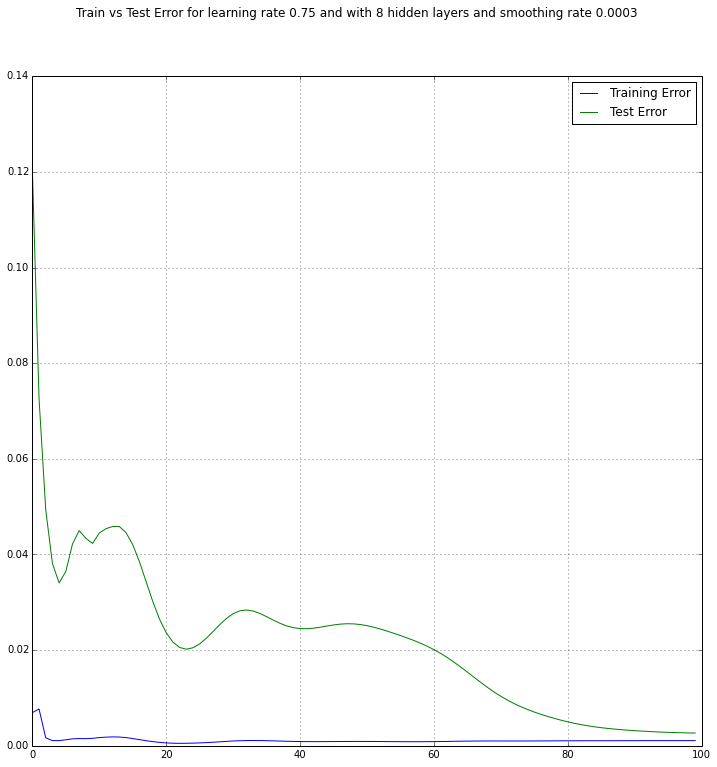
Below is a graph showing 3 hidden layers and smoothing rate of 0.0003 and a learning rate of 0.6.Error was minimal around 50th Epoch and then it increased and finally went down again at epoch 80. The fit is good but it is smoother and thus it is less fit than without regularization.

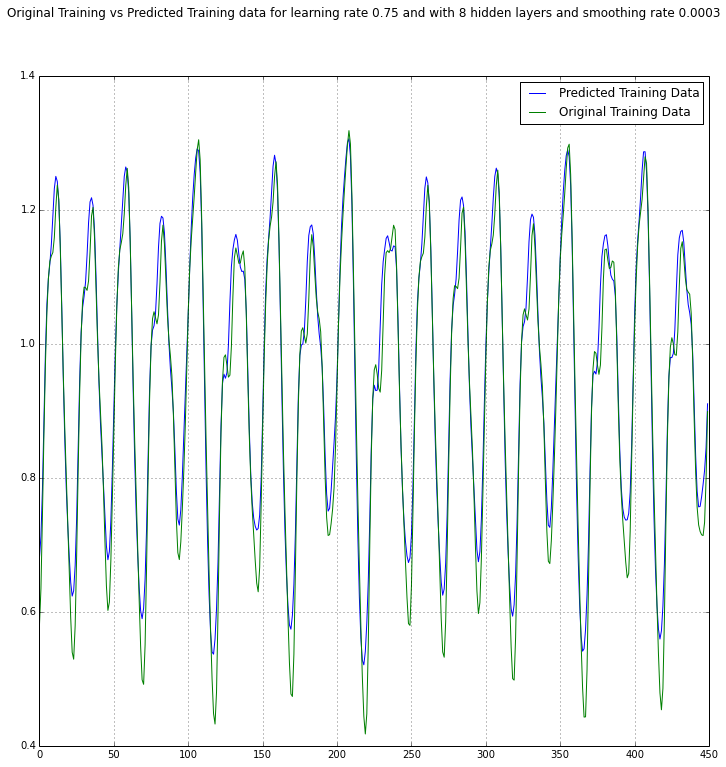


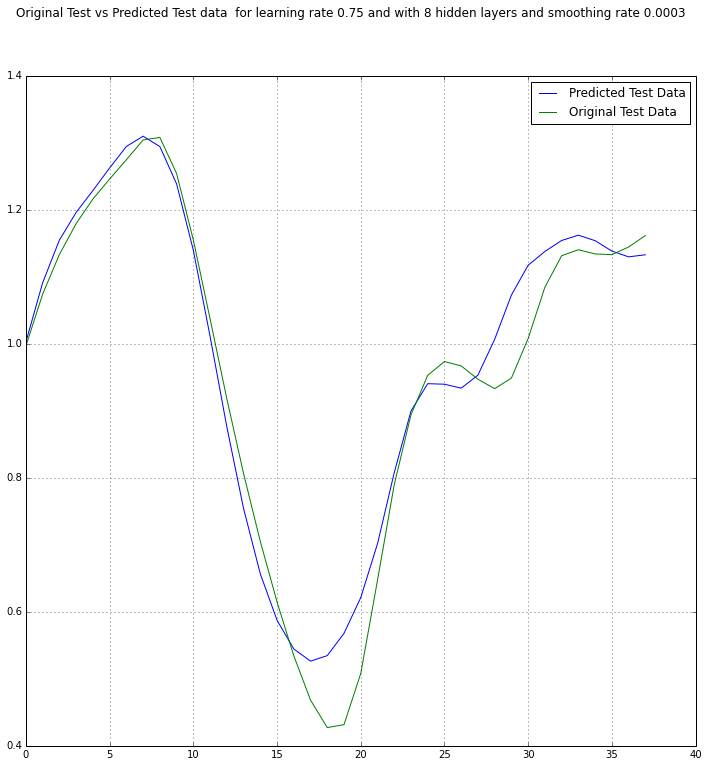




Now I will show with 8 hidden layers and learning rate of 0.75. and smoothing rate of 0.0003 The error was minimal at epoch 20 and then it increased and finally went stabilized after epoch 40. The fit is quite good but as there is smoothing in the curve this fit is not better than the unregularized one.







## Second Order Neural Networks using LM:

Background:

First order training is considered slow and it takes lots of time to train if you have large network and large number of training examples. Second order Neural Network training using Newton’s method provides a much faster way to train neural network but these are not used in practice for following reasons.

Second order training needs Hessian calculation and this is very resource consuming and as the number of weights are increasing the hessian will be increased as well.

It also needs the Inverse of Hessian and there are chances that Hessian matrix is not invertible.

To fix the issues with the Hessian calculations you can approximate the Hessian. There are different ways but I will consider here the Gauss Newton Method.

Here the Hessian is calculated from Jacobian Matrix

**H** = **J***T***J**

Gradient can be calculated as

**g** = **J***T***e**

where **e** is error vector.

However, it doesn’t still solve the problem 2, which is inverse of the Hessian, and thus you can add a small value to each element of the Hessian to make it invertible and this is formula for Levenberg-Marquardt

**H** = **J***T***J +** *μ***I**

The weights will be updated using following formula

**Wk+1 = Wk +H-1 + g**

When the scalar µ is zero, this is just Newton's method, using the approximate Hessian matrix. When µ is large, this becomes gradient descent with a small step size. Newton's method is faster and more accurate near an error minimum, so the aim is to shift toward Newton's method as quickly as possible. Thus, µ is decreased after each successful step (reduction in performance function) and is increased only when a tentative step would increase the performance function. In this way, the performance function is always reduced at each iteration of the algorithm.

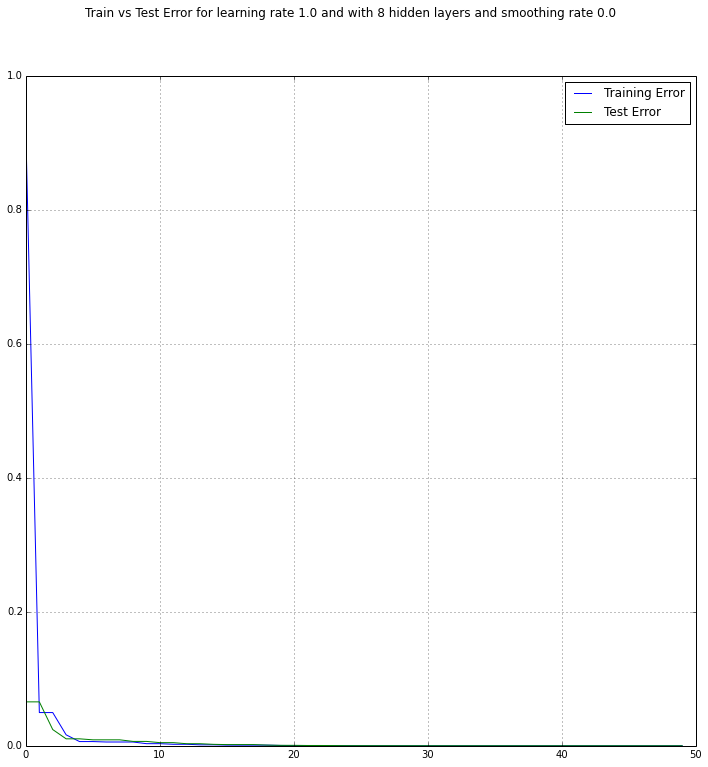
See [2] for more details on the formulas and algortithm.

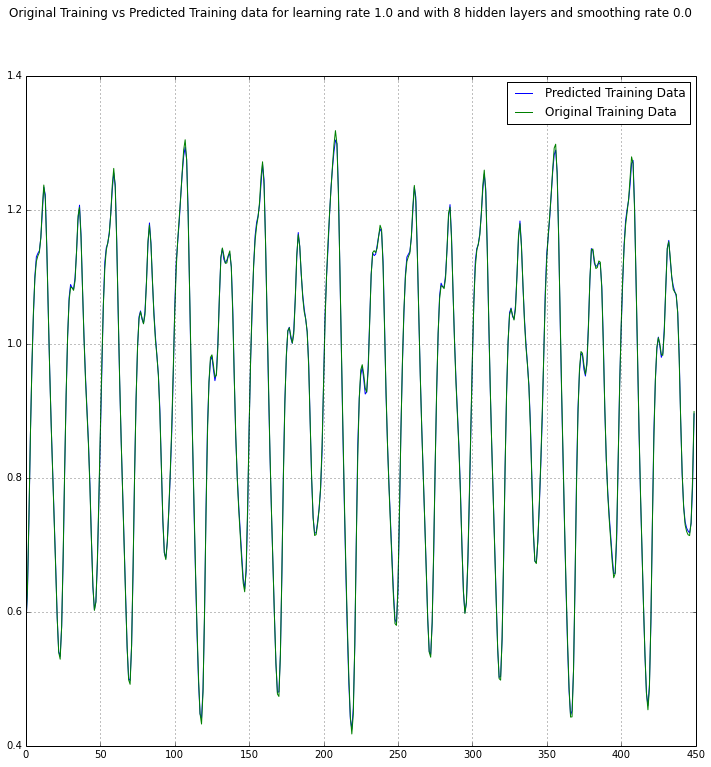
Code:

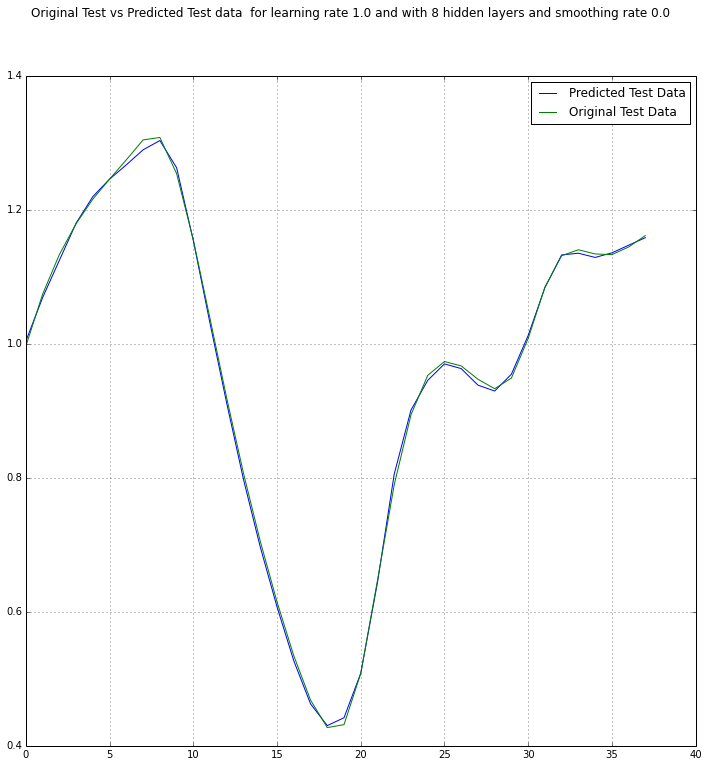
The code for this is in the file **NN\_2\_1.py** and **TS\_second\_order.py**. Here I have used a single value of the learning\_eta(This is mu and this value will be multiplied with identity matrix and will be added to Hessian and then it will be reversed. Here I experimented with different values of the hidden layers. Graphs are shown below. I have not used any early stopping as I was looking to see how it would behave. I have followed my approach as well as approaches discussed in [1] and [2].

Graphs:

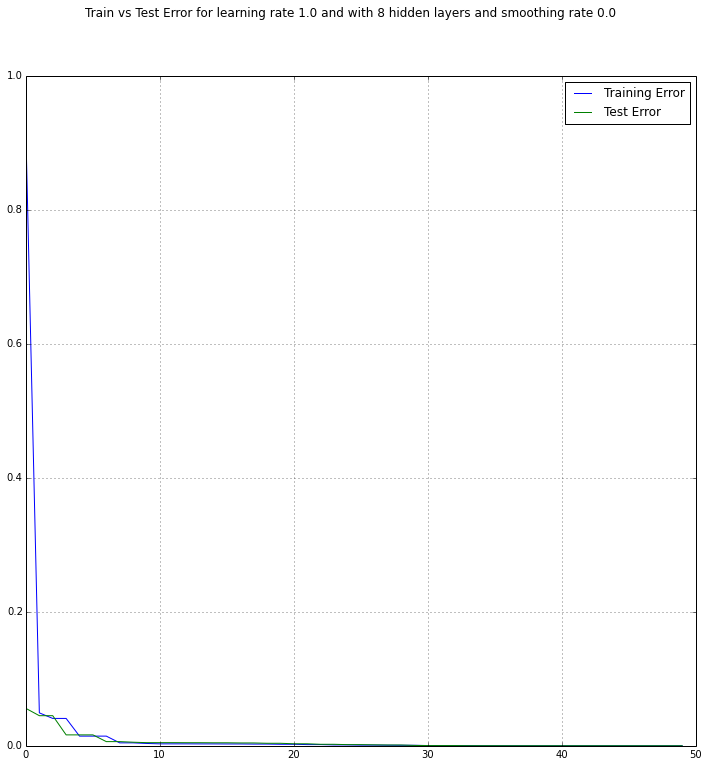
I have chosen the number of hidden layers as 8 to show the graphs. As you could see, from the below graph that the error dropped to very minimal levels in 3rd-4th Epoch. After 9th Epoch it reached its minimal.

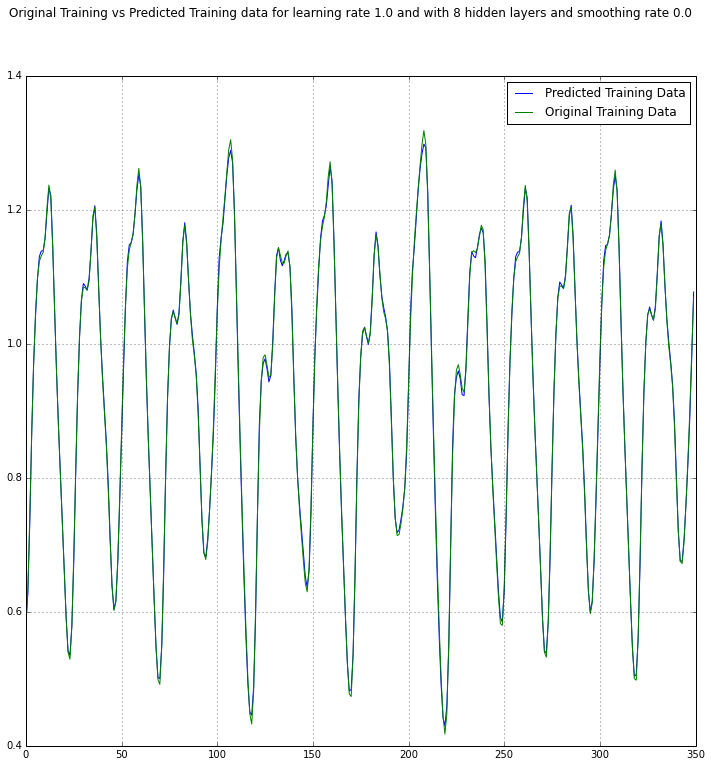


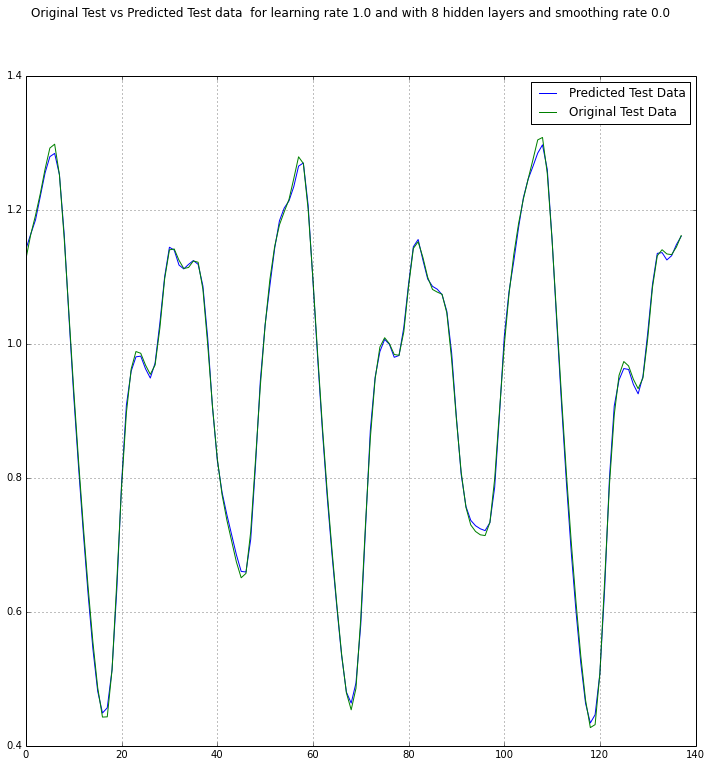




Below are graphs when I used 350 examples for training and rest for the test. I wanted to see how would it look like. The fit is similar when I used 450 examples. Thus, the neural network is working fine.



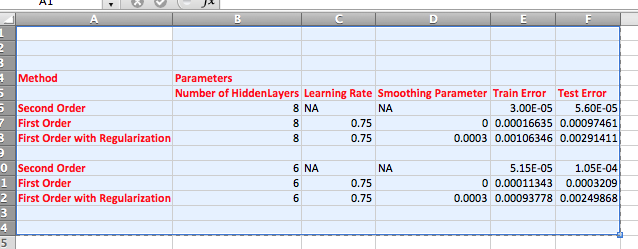




# Conclusion:

I have tried 3 different methods for time series forecasting. The Second Order LM training gave really good fit and error was minimized between epochs 3 and 8, which is quite fast as compared to other 2 methods where it was 40 or more epochs. Also, regularization provided less fitted data and smoother curve and was not that beneficial in this case as the data did not have any noise. But, using regularization is much better way to handle the data which over fits.

Below is a table, which shows the data for all these 3 methods and their test and train errors.



As you could see that for First Order methods with 8 hidden layers there is slight over fitting as compared to 6 hidden layers for both Regularized as well as non-regularized methods. Thus, I should have tried with 6 hidden layers. You can find all these graphs in the file **[output\_figures.pdf and output\_figures\_second\_order.pdf]** which are in the same folder as this document, for other settings of hidden layers and learning rates and smoothing parameters and their graphs. The code can be run by changing these settings as well as you can change the number of period for lagging which is 10 here.

I will use some of this code especially order 1 with regularization in my final project when I will predicate the price of CPI or some other items which are being used to find CPI.

Full code is in github [4].

# References:

1. <http://www.eng.auburn.edu/~wilambm/pap/2011/Neural%20Network%20Training%20with%20Second%20Order%20Algorithms.PDF>
2. <http://www.eng.auburn.edu/~wilambm/pap/2011/K10149_C012.pdf>
3. <http://en.wikipedia.org/wiki/Time_series>
4. <https://github.com/darshanmeel/NN2>
5. <http://research.microsoft.com/en-us/um/people/cmbishop/prml/>
6. <http://homepages.gold.ac.uk/nikolaev/cis311.htm>
7. <http://homepages.gold.ac.uk/nikolaev/311tune.htm>
8. <http://homepages.gold.ac.uk/nikolaev/311bpr.htm>
9. <http://uk.mathworks.com/help/nnet/ref/trainlm.html>