Nonlinear Regression with Neural Networks

Swetha Varadarajan

October 24, 2014

Contents

Abstract

Linear models are being well sorted for its ease of view and many other special properties. But,in reality, we come across many irregularities with the actual norm. These are the non-linear qualities that disrupts or add qualities to the normality. In this report, non linear regression with neural network has been presented. Its implementation in Python language is explained. Demonstration with a simple function and also using the dataset selected from StatLib Datasets Archive is also presented.

1 Introduction

Artificial Neural Network can be described as a computational model that depicts behaviour of the biological nervous system to solve the problems such as pattern recognition etc. Linear models cannot be used for these kind of networks because of the curse of dimensionality. These models has a fixed basis function. But,it is necessary to adapt basis functions to the data to be practical. Neural Networks solve this problem by

using parametric basis functions in which parameters are adapted during the training of the model. These parameters depend on the kind of algorithm we chose to reduce the error. Like in linear models, this cannot be done by simply updating because of the complexity involve. So, we resort to a method called SCG(Scaled Conjugate Gradient) wherein, an approximation of the second derivative is used to find the error convergence.

1.1 Objective

Objective is to develop a Neural Network model using the SCG method of error convergence. So, the algorithm or steps is as follows:

1. Assume some random initial weights for both the hidden and the output layer
2. Do the forward pass by calculating the target values from the selected weight
3. Calculate the Gradient.
4. Update the weights and the hidden values.
5. Repeat till convergence.

2 Python Implementation

This section gives a detailed description of **python code** for reading the data, constructing input and target matrices, visualizations of the data, random partitioning of data into training, validation, and testing, standardization, neural network with **scg** code, variation of hidden units and finally the comparison of linear and neural network models.

2.1 Reading,constructing and visualization

The data set NO2 from the numeric folder of StatLib Datasets Archive was taken. The code for reading the data is shown in the following listing. The code is adapted from Piazza discussion dated October 16, 2014. The scipy package was used to read data.

```
# reading data
from scipy.io import arff
import numpy as np
data,meta = arff.loadarff('no2.arff')
```

The first column is taken as the target and the remaining 7 columns were taken as the input data. So, the input matrix is of dimension $500X7$ and the target matrix is a column vector with 500 rows(samples).

```
# constructing
X = data[meta.names()[1:]].view(np.float).reshape((data.shape[0],-1))
T = data[meta.names()[0]].view(np.float).reshape((data.shape[0], -1))
```

To visualize the target variables as a function of the 7 input variables, 7 plots were drawn and the code corresponding to it is shown below. The names of the attributes are adopted from [2].

```
# visualization
Xnames=['log of #cars per hour', 'temp. $2$ meter above ground (degree C)',
                'wind speed (meters/second)', 'temp. diff between $25$ and $2$
                meters' , 'wind direction (degrees between 0 and 360)',
                'hour of day', 'day number from October 1. 2001.']
plt.figure(figsize=(10,10))
for c in range(X.shape[1]):
    plt.subplot(3,3, c+1)
    plt.plot(X[:,c], T[:], 'o', alpha=0.5)
    plt.ylabel('log conc. of NO2')
    plt.xlabel(Xnames[c])
```

```
12      plt.subplots_adjust( hspace=0.85 , wspace =0.5 )
13  plt.savefig('visualization-hw4.png')
```

2.2 Random partitioning of data and standardization

The data set is divided into 60 percentage of training, 20 percentage of validation and 20 percentage of testing data as shown in the following listing. The code is modified such that it can either partition the data for validation or not. So, there is a **if condition** to check this attribute. Fractions is a list of length 3 which contains the partitioning ratio. The data is partitioned in a random manner by shuffling the row indices and grouping. So, at the end we get 6 sets which is shown in table 1.

Data set name	size
XTrain(Input training data)	(300, 7)
TTrain(Target training data)	(300, 1)
XTest(Input testing data)	(100, 1)
TTest(Target testing data)	(100, 1)
XValidate(Input validating data)	(100, 1)
TValidate(Target validating data)	(100, 1)

Table 1: Partitioned data sets and size

```
1   #random data partitioning
2   trainFraction = fractions[0]
3       if len(fractions) == 2:
4           # Skip the validation step
5           validateFraction = 0
6           testFraction = fractions[1]
7       else:
8           validateFraction = fractions[1]
9           testFraction = fractions[2]
10
11      n = X.shape[0]
12      nTrain = round(trainFraction * n)
13      nValidate = round(validateFraction * n)
14      nTest = n - nTrain - nValidate
15
16      rowIndices = np.arange(n)
17
18      # Random arrangement of row indices
19      np.random.shuffle(rowIndices)
20      # Assign Xtrain and Train. Remove columns from Xtrain
21      # that contain constant values.
22      Xtrain = X[rowIndices[:nTrain],:]
23      Ttrain = T[rowIndices[:nTrain],:]
24
25      if nValidate > 0:
26           # Assign Xvalidate and Tvalidate
27           Xvalidate = X[rowIndices[nTrain:nTrain+nValidate],:]
28           Tvalidate = T[rowIndices[nTrain:nTrain+nValidate],:]
29
30      # Assign Xtest and Ttest
31      Xtest = X[rowIndices[nTrain+nValidate:],:]
32      Ttest = T[rowIndices[nTrain+nValidate:],:]
```

The standardization of data function is defined in the Neural Networks class and called upon in both the train and use methods. The code is shown in function defined in the following listing. Lines 7-14 defines

the function for standardization and un-standardization of input and target data. It takes the object and the variable as the input and returns the standardized result. Lines 2-5 is nothing but an example where the function is called. The formula for standardization is given in equation-3 in which z is the standardized data, x is the original data, μ is the mean and σ is the standard deviation. Mean and standard deviation are found using numpy package functions numpy.mean and numpy.std respectively.

$$z = \frac{x - \mu}{\sigma} \tag{1}$$

```
1  # standardization of data
2  if self.Xmeans is None:
3          self.Xmeans = X.mean(axis=0)
4      self.Xstds = X.std(axis=0)
5  X = self._standardizeX(X)
6
7  def _standardizeX(self,X):
8      return (X - self.Xmeans) / self.Xstds
9  def _unstandardizeX(self,Xs):
10     return self.Xstds * Xs + self.Xmeans
11 def _standardizeT(self,T):
12     return (T - self.Tmeans) / self.Tstds
13 def _unstandardizeT(self,Ts):
14     return self.Tstds * Ts + self.Tmeans
```

2.3 Neural network with scg code

This is the part of the code that performs the neural network. In the following listing, the line-1 assigns an object to the class **NeuralNetwork**.The object **nnet** calls the method **train** passing the arguments of training data and the number of iteration. Later, the method **use** which is being used 3 times to find out the predicted training, validation and testing values. In the next two sub-sub-sections, the code for these 2 methods are presented. The listing also shows the constructor associated with the object creation. The number of input, output and hidden values are given by ni, no and nhs respectively. The hidden layer and weights are assigned some random uniform initial values with dimension as shown. Mean and standard deviation are assigned None.

```
1  # Neural network
2  nnet = nn.NeuralNetwork(Xtrain.shape[1],nHiddens,1)
3  nnet.train(Xtrain,Ttrain,nIterations=500)
4  Ptrain = nnet.use(Xtrain)
5  if nValidate > 0:
6      Pvalidate = nnet.use(Xvalidate)
7  Ptest = nnet.use(Xtest)
8
9  #Constructor
10   def __init__(self,ni,nhs,no):
11         try:
12             nihs = [ni] + list(nhs)
13         except:
14             nihs = [ni] + [nhs]
15             nhs = [nhs]
16         self.Vs = [np.random.uniform(-0.1,0.1,size=(1+nihs[i],nihs[i+1]))
17                    for i in range(len(nihs)-1)]
18         self.W = np.random.uniform(-0.1,0.1,size=(1+nhs[-1],no))
19         # print [v.shape for v in self.Vs], self.W.shape
20         self.ni,self.nhs,self.no = ni,nhs,no
21         self.Xmeans = None
22         self.Xstds = None
23         self.Tmeans = None
```

```
24          self.Tstds = None
25          self.iteration = mp.Value('i',0)
26          self.trained = mp.Value('b',False)
27          self.reason = None
28          self.errorTrace = None
```

2.3.1 Method 1: train(Training)

The train method takes the input data, target data, number of iterations, weight precision, error precision and verbose as inputs. It determines the mean and standard deviation if not defined before in lines 3-14. It has many other methods listed in it.

1. **def objectiveF(w)**: It calls the forward pass function and with the returned values, finds the RMSE and returns it.
2. **def gradF(w)** : It also calls the forward pass function, calculates the constant value and then passes it to the backward pass function. Finally, returns the updated weights.
3. **scg.scg()**: this is a external method which is called. The result of this SCG is stored in a single array, extracted, assigned to the object variables and returned.

The pack and unpack are nothing but stacking 2 arrays and releasing them into separate arrays respectively.

```
1  def train(self,X,T,
2                nIterations=100,weightPrecision=0,errorPrecision=0,verbose=False):
3          if self.Xmeans is None:
4              self.Xmeans = X.mean(axis=0)
5              self.Xstds = X.std(axis=0)
6          X = self._standardizeX(X)
7
8          if T.ndim == 1:
9              T = T.reshape((-1,1))
10
11         if self.Tmeans is None:
12             self.Tmeans = T.mean(axis=0)
13             self.Tstds = T.std(axis=0)
14         T = self._standardizeT(T)
15
16         # Local functions used by gradientDescent.scg()
17
18         def objectiveF(w):
19             self._unpack(w)
20             Y,_ = self._forward_pass(X)
21             return 0.5 * np.mean((Y - T)**2)
22
23         def gradF(w):
24             self._unpack(w)
25             Y,Z = self._forward_pass(X)
26             delta = (Y - T) / (X.shape[0] * T.shape[1])
27             dVs,dW = self._backward_pass(delta,Z)
28             return self._pack(dVs,dW)
29
30         scgresult = scg.scg(self._pack(self.Vs,self.W), objectiveF, gradF,
31                             xPrecision = weightPrecision,
32                             fPrecision = errorPrecision,
33                             nIterations = nIterations,
34                             iterationVariable = self.iteration,
35                             ftracep=True,
36                             verbose=verbose)
```

```
37
38          self._unpack(scgresult['x'])
39          self.reason = scgresult['reason']
40          self.errorTrace = scgresult['ftrace']
41          self.numberOfIterations = len(self.errorTrace) - 1
42          self.trained.value = True
43          return self
```

2.3.2 Method 2: use(Predicting)

It does the forward pass of the algorithm. this is mainly used after selecting the best weight values from analysing the training and testing data.

```
1   def use(self,X,allOutputs=False):
2           Xst = self._standardizeX(X)
3           Y,Z = self._forward_pass(Xst)
4           Y = self._unstandardizeT(Y)
5           return (Y,Z[1:]) if allOutputs else Y
```

2.3.3 Method 3: Forward and backward pass

The forward pass is nothing but the implementation of the algorithm as said in objective with the support of associated mathematical equations. The corresponding python code is shown in the listing. Z is the output of the first layer which is the dot product of input and the hidden elements. Y is the final output which is the dot product of Z and the weights. The backward pass consists of updating the weights and the hidden value according to the derived formula.

```
1   def _forward_pass(self,X):
2           Zprev = X
3           Zs = [Zprev]
4           for i in range(len(self.nhs)):
5               V = self.Vs[i]
6               Zprev = np.tanh(np.dot(Zprev,V[1:,:]) + V[0:1,:])
7               Zs.append(Zprev)
8           Y = np.dot(Zprev, self.W[1:,:]) + self.W[0:1,:]
9           return Y, Zs
10
11      def _backward_pass(self,delta,Z):
12          dW = np.vstack((np.dot(np.ones((1,delta.shape[0])),delta),
13                                                          np.dot( Z[-1].T, delta)))
14          dVs = []
15          delta = (1-Z[-1]**2) * np.dot( delta, self.W[1:,:].T)
16          for Zi in range(len(self.nhs),0,-1):
17              Vi = Zi - 1 # because X is first element of Z
18              dV = np.vstack(( np.dot(np.ones((1,delta.shape[0])), delta),
19                               np.dot( Z[Zi-1].T, delta)))
20              dVs.insert(0,dV)
21              delta = np.dot( delta, self.Vs[Vi][1:,:].T) * (1-Z[Zi-1]**2)
22          return dVs,dW
```

2.4 Variation of hidden units

A separate function called $partition_train_validate_test$ has been defined for to understand the variation of hidden units on the result. Here, the network shapes is the number of hidden variables passed to the function. Lines 2-16 is the partitioning. There are 2 iterations in this.

1. Iteration over the hidden value range

2. Repetition for a given hidden value to get the averaged RMSE.

Lines 26-41 is the randomization. Lines 43-49 is the NeuralNetwork model building. Lines 51-60 calculates RMSE for the various partitioned data sets. The line 67-68 is how we call the function. The textbfrmse is another method which is defined outside and being called in lines52,54,55. The final result is a 4-Dimensional array with nHiddens, RMSE Train, RMSE Test and RMSE Validate as its dimensions. Number of rows will be number of iteration times the number of unique hidden values selected for the experiment.

```
1  def partition_train_validate_test(X,T, fractions, network_shapes, nRepetitions):
2      trainFraction = fractions[0]
3      if len(fractions) == 2:
4          # Skip the validation step
5          validateFraction = 0
6          testFraction = fractions[1]
7      else:
8          validateFraction = fractions[1]
9          testFraction = fractions[2]
10
11     n = X.shape[0]
12     nTrain = round(trainFraction * n)
13     nValidate = round(validateFraction * n)
14     nTest = n - nTrain - nValidate
15     print(nTrain,nValidate,nTest)
16     rowIndices = np.arange(n)
17
18     results = []  # will contain nHiddenUnits, RMSEtrain, RMSEvalidate, RMSEtest
19
20     for nHiddens in network_shapes:
21
22         print('Working with hidden layers =', nHiddens)
23
24         for rep in range(nRepetitions):
25
26             # Random arrangement of row indices
27             np.random.shuffle(rowIndices)
28
29             # Assign Xtrain and Train. Remove columns from
30                 # Xtrain that contain constant values.
31             Xtrain = X[rowIndices[:nTrain],:]
32             Ttrain = T[rowIndices[:nTrain],:]
33
34             if nValidate > 0:
35                 # Assign Xvalidate and Tvalidate
36                 Xvalidate = X[rowIndices[nTrain:nTrain+nValidate],:]
37                 Tvalidate = T[rowIndices[nTrain:nTrain+nValidate],:]
38
39             # Assign Xtest and Ttest
40             Xtest = X[rowIndices[nTrain+nValidate:],:]
41             Ttest = T[rowIndices[nTrain+nValidate:],:]
42
43             # build the model and test it
44             nnet = nn.NeuralNetwork(Xtrain.shape[1],nHiddens,1)
45             nnet.train(Xtrain,Ttrain,nIterations=500)
46             Ptrain = nnet.use(Xtrain)
47             if nValidate > 0:
48                 Pvalidate = nnet.use(Xvalidate)
49             Ptest = nnet.use(Xtest)
50
51             # Use the model to predict and calculate errors.
52             RMSEtrain = rmse(Ptrain,Ttrain)
```

```
            if nValidate > 0:
                RMSEvalidate = rmse(Pvalidate,Tvalidate)
            RMSEtest = rmse(Ptest,Ttest)

            if nValidate > 0:
                results.append([nHiddens, RMSEtrain, RMSEvalidate, RMSEtest])
            else:
                results.append([nHiddens, RMSEtrain,RMSEtest])

    return np.array(results), Ptrain, Ttrain, nnet
def rmse(P,T):
    return np.sqrt(np.mean(P-T)**2)

# Invoking
results,_,_,_ = partition_train_validate_test
(X,T, (0.6,0.2,0.2), (1,2,5,10,20,40,50), 10)
```

2.5 Comparing linear and neural network models

The linear least square model is implemented. The lines is the standardization definition. Lines are the partitioning followed by randomization. the weights are been calculated and the predicted results are obtained. This is followed by the Neural Network code in which the best hidden values are taken for calculation. Both the results are plotted with target versus predicted data and their corresponding regression lines. The discussion on the result is given in next to next section.

```
# Linear least square model :
def makeStandardize(X):
    means = X.mean(axis=0)
    stds = X.std(axis=0)

    def standardize(origX):
        return (origX - means) / stds

    def unStandardize(stdX):
        return stds * stdX + means

    return (standardize, unStandardize)
X1 = np.hstack((np.ones((X.shape[0],1)), X))
nrows = X1.shape[0]
nTrain = int(round(nrows*0.8))
nTest = nrows - nTrain
#nTrain,nTest,nTrain+nTest
rows = np.arange(nrows)
np.random.shuffle(rows)
#rows
trainIndices = rows[:nTrain]
testIndices = rows[nTrain:]
#trainIndices,testIndices
Xtrain = X[trainIndices,:]
Ttrain = T[trainIndices,:]
Xtest = X[testIndices,:]
Ttest = T[testIndices,:]

(standardize, unStandardize) = makeStandardize(Xtrain)

XtrainS = standardize(Xtrain)
XtestS = standardize(Xtest)
np.mean(XtrainS,axis=0), np.std(XtrainS,axis=0),
```

```
      np.mean(XtestS,axis=0), np.std(XtestS,axis=0)
XtrainS1 = np.hstack((np.ones((XtrainS.shape[0],1)), XtrainS))
XtestS1 = np.hstack((np.ones((XtestS.shape[0],1)), XtestS))
w = np.linalg.lstsq( np.dot(XtrainS1.T,XtrainS1),
    np.dot(XtrainS1.T, Ttrain))[0]  # see this [0]?

prediction = np.dot(XtestS1,w)

# plotting LLS
plt.clf()
plt.figure(figsize=(10,10))
plt.plot(prediction[:],Ttest[:],'o')
aa = min(min(prediction[:]),min(Ttest[:]))
ba = min(max(prediction[:]),max(Ttest[:]))
plt.plot([aa,ba],[aa,ba],'yellow',linewidth=3)

# NeuralNetwork
newresults,Ptest,Ttest,nnet = partition_train_validate_test
(X,T, (0.8,0.2), (bestHiddens,), 1)
RMSEtrain = newresults[0,1]
RMSEtest = newresults[0,2]

print('Best nHidden of',bestHiddens,' results in RMSE
train of',RMSEtrain,' and RMSE test of',RMSEtest)

# Plotting NN
plt.plot(Ttest,Ptest,'^')
a = max(min(Ptest),min(Ttest))
b = min(max(Ptest),max(Ttest))
plt.plot([a,b],[a,b],'red',linewidth=7,ls='dotted')
plt.xlabel('Actual Intensity')
plt.ylabel('Predicted Intensity')
plt.title('Using best nHiddens ='+str(bestHiddens));
plt.legend(('LLS data','LLS model','NN data','NN model'),loc='best');
plt.savefig('comparison.png')

print(aa,ba,a,b)
```

2.6 Plotting code

The list of plots discussed in this report are,

1. **Visualization:** Already discussed in Section 2.2
2. **Iterations versus RMSE:** The code is shown in the listing. It calculates the RMSE of each iteration for 2 of the hidden value(1 and 200) against iterations. It is done separately on the 3 partitioned data. The plot has 2 subplots. One is with training data other with the other 2 data.

```
                results=np.array(results)
uniqueNHiddens = np.unique(results[:,0])
plt.figure(figsize=(10,10))
plt.subplot(2,2,1)
plt.plot(results[:100,1])
plt.legend(('Train'),loc='best')
plt.ylabel('RMSE')
plt.xlabel('Iterations')
plt.subplot(2,2,2)
plt.plot(results[:100,2:])
plt.legend(('Validate','Test'),loc='best')
```

```
plt.ylabel('RMSE')
plt.xlabel('Iterations')
plt.subplot(2,2,3)
plt.plot(results[101:,1])
plt.legend(('Train'),loc='best')
plt.ylabel('RMSE')
plt.xlabel('Iterations')
plt.subplot(2,2,4)
plt.plot(results[101:,2:])
plt.legend(('Validate','Test'),loc='best')
plt.ylabel('RMSE')
plt.xlabel('Iterations')
plt.subplots_adjust(hspace=0.5)
plt.savefig('iterations-hw4.png')
```

3. **Linear versus Neural**: Discussed in the previous subsection

4. **Hidden units versus RMSE**: The results obtained from the model has number of hidden values * number of iteration results. The mean RMSE is taken over the iteration corresponding to each hidden value and the plot is drawn for all the 3 partitioned data.

```
        results = np.array(results)
uniqueNHiddens = np.unique(results[:,0]) # should be equal to lambdas
means = []
for nHidden in uniqueNHiddens:
    mask = nHidden == results[:,0]
    means.append([nHidden] + np.mean(results[mask,1:],axis=0).tolist())
means = np.array(means)
plt.figure(1)
plt.subplot(2,1,1)
plt.plot(means[:,0],means[:,1:2])
plt.legend(('Train',))
plt.subplot(2,1,2)
plt.plot(means[:,0], means[:,2:])
plt.legend(('Validate','Test'),loc='best');
plt.savefig('hiddenvar.png')
```

2.7 Additional question:

I was curious about the "temp. 2 meter above ground (degree C)" attribute from the visualization graph. It strongly had some non-linearity. So, I just used the single input single output with this attribute as the input and the NO2 concentration as output. There is a change only in reading the input value. Remaining code is the same.

```
X = data[meta.names()[4]].view(np.float).reshape((data.shape[0],-1))
```

A change in the activation function and the result's observation. Code is as follows.

```
def sigmoid(X):
                    return 1/(1+np.exp(-X))
```

The calculation of the prediction variable is done as follows

```
Y = np.dot(self.addOnes(sigmoid(np.dot(X1, self.V))), self.W)
```

3 Toy data demonstration

A simple input is taken(completely adapted from class notebook). the target is some non-linear function of the input. It has exponential functions.

```
nSamples = 200
X = np.linspace(0,50,nSamples).reshape((-1,1))
T = 1.5 - 0.6 * X + 30 * np.exp(-0.1*(X-20)**2)  -
 40 * np.exp(-0.1*(X-35)**2)  + np.random.uniform(-5,5,size=(nSamples,1))
```

The model is run on it for the list of hidden values shown and for 10 iterations each.

```
results,_,_,_ = partition_train_validate_test(X,T,
 (0.6,0.2,0.2), (1,2,5,10,20,40,50), 10)
```

Best hidden values and the newresults obtained with this parameter is obtained. Here, validation is not necessary since it is used only to find the best hidden value.

```
bestHiddens = uniqueNHiddens[np.argmin(means[:,2])]
# third column contains validation data RMSE
newresults,Ptest,Ttest,nnet = partition_train_
validate_test(X,T, (0.8,0.2), (bestHiddens,), 1)
```

From this, the RMSE is found and the plots are drawn. The best value of hidden layer is found to be 20.

```
RMSEtrain = newresults[0,1]
RMSEtest = newresults[0,2]
print('Best nHidden of',bestHiddens,' results in RMSE train
of',RMSEtrain,' and RMSE test of',RMSEtest)
plt.figure(2)
plt.clf()
plt.plot(Ttest,Ptest,'o')
a = max(min(Ptest),min(Ttest))
b = min(max(Ptest),max(Ttest))
plt.plot([a,b],[a,b],'r',linewidth=3)
plt.xlabel('Actual Intensity')
plt.ylabel('Predicted Intensity')
plt.title('Using best nHiddens ='+str(bestHiddens));
```

Hidden versus RMSE graph and also the actual vs predicted result graph is shown in figure 6 and 7 respectively. It can be seen that the model fits the data very accurately for best hidden value of 20. The RMSE plot also shows a decrease in the RMSE of the test data when hidden value is 20.

4 Discussion

This section discusses the initial plots of data, iterations and hidden units versus results and also about the efficiency of model prediction.

4.1 Initial data set

The data set selected is NO2. There is no concrete reason to select. It was the first suggested data-set. So, took it. The first column is the NO2 concentration which is dependent on the remaining 7 attributes. The figure-1 shows the input data versus target data. From this, the following observations can be drawn.

1. **log of number of cars per hour:** Increases as it is obvious.
2. **temp. 2 meter above ground (degree C):**Seems like a parabolic behaviour
3. **wind speed (meters/second):**Seems to decrease gradually with increase in speed.
4. **temp. diff between 25 and 2 meters:**This too seems like a parabola

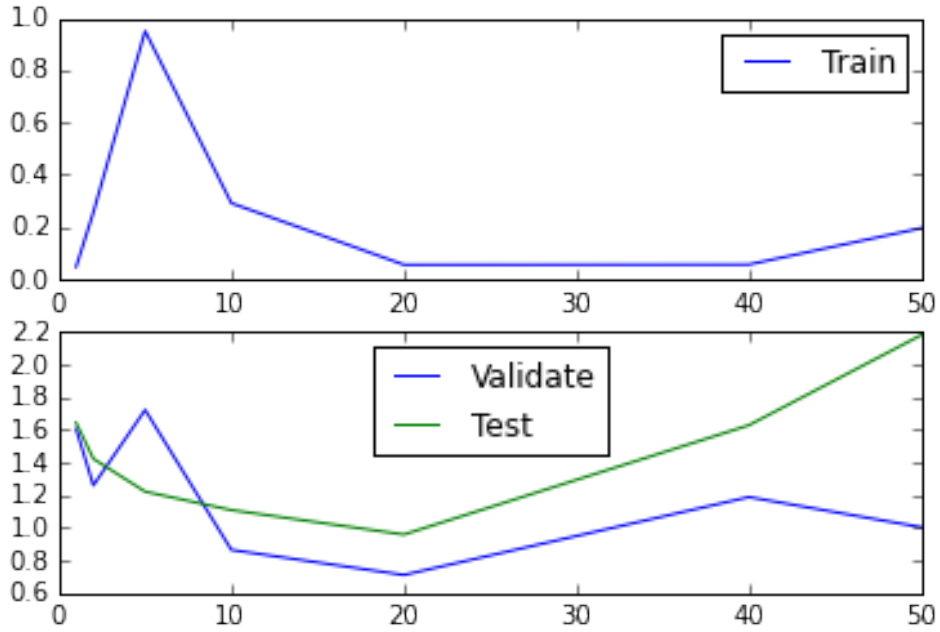

Figure 1: Hidden units versus RMSE

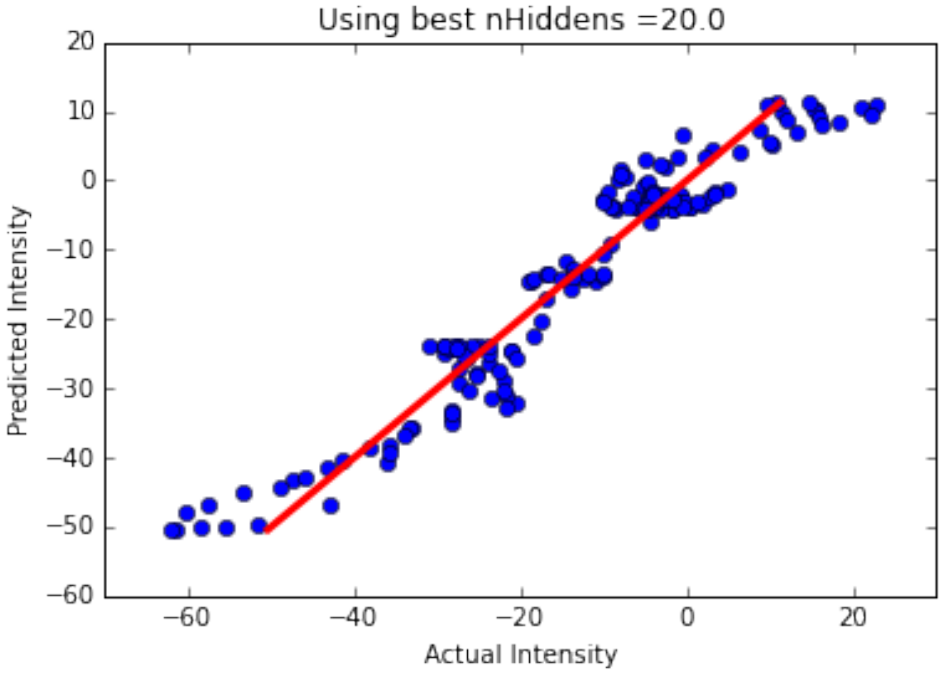

Figure 2: Actual vs Predicted

5. **wind direction (degrees between 0 and 360):**More concentrated at 50 and 250 degree angle
6. **hour of day:**constant throughout. Sometimes less during the start and sometimes high during the noon.
7. **day number from October 1. 2001:**concentrated from 0-200 and 400-600. empty every where else.May be asymptotic.

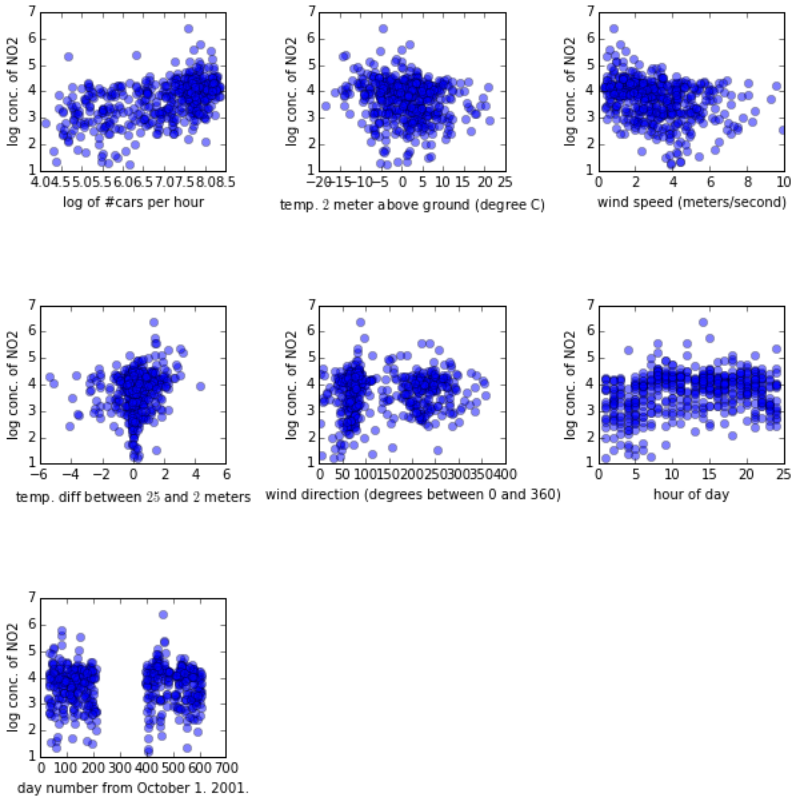

Figure 3: Input data versus target data

4.2 Iterations versus RMSE

The test was performed for hidden layers 1 and 200. 1 was selected because it was the best hidden layer. 200 was selected so as to select some extreme value condition. The calling function is shown in the listing.

```
1 results,_,_,_ = partition_train_validate_test(X,T, (0.6,0.2,0.2), (1,200), 100)
```

The graph obtained on this result is also shown. Sub plots-(2,2,1) and (2,2,2) corresponds to hidden value 1 and the other two correspond to 200 in both the plots. The figure-2 was not clear enough. So, took only the first 25 iterations as shown in figure-3. In figure-2, it can be seen that the RMSE for hidden value 1 is less than the other but the variation in RMSE is filled with crusts and troughs for the validation and test data. But, for hidden value 200, the peaks reduce with iteration. In figure-3, we can conclude that RMSE decreases with iteration (18 for hidden value 1 and 10 for hidden value 200) and then shoots up.

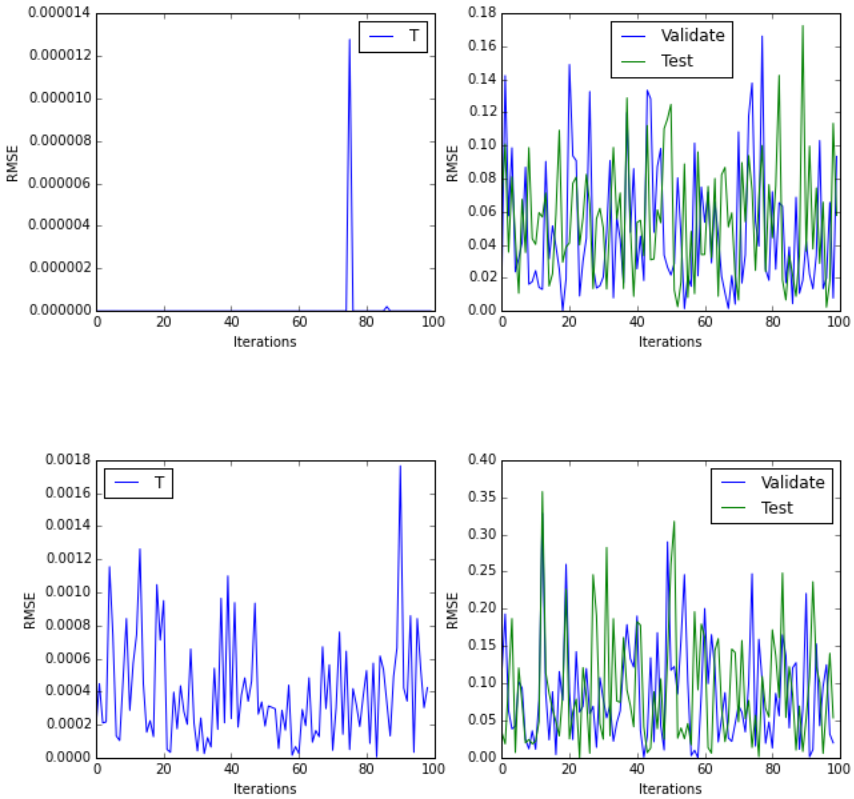

Figure 4: Iterations versus RMSE: iterations= 100

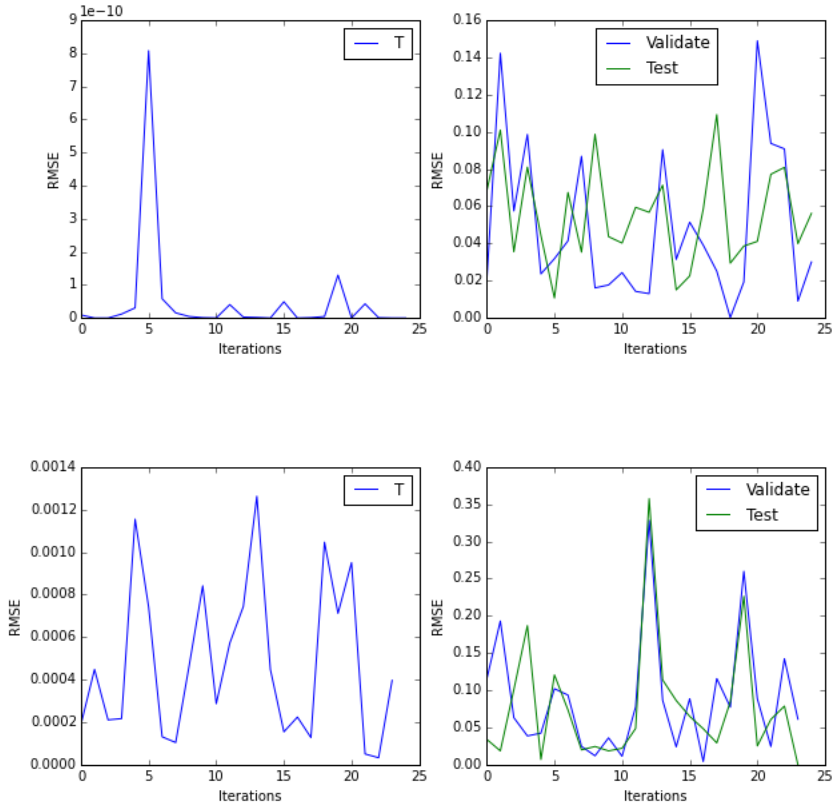

Figure 5: Iterations versus RMSE: iterations = 25

4.3 Linear and Neural model

Although the best hidden value that I obtained was 1, i excluded that and repeated the iterations for remaining values and got the best value as 5. I used this to compare with the linear model since if we use a hidden value of 1, it means that it is almost linear. to confirm this, the figure 4 shows the predicted versus actual data as well as the regression line. It can be seen that the regression lines coincide with each other. But, the data in linear model seems to be more close to the line that that of the neural network model. From this, we can conclude that the value of 1 is the best value of hidden layer and that our selected example behaves almost linear.

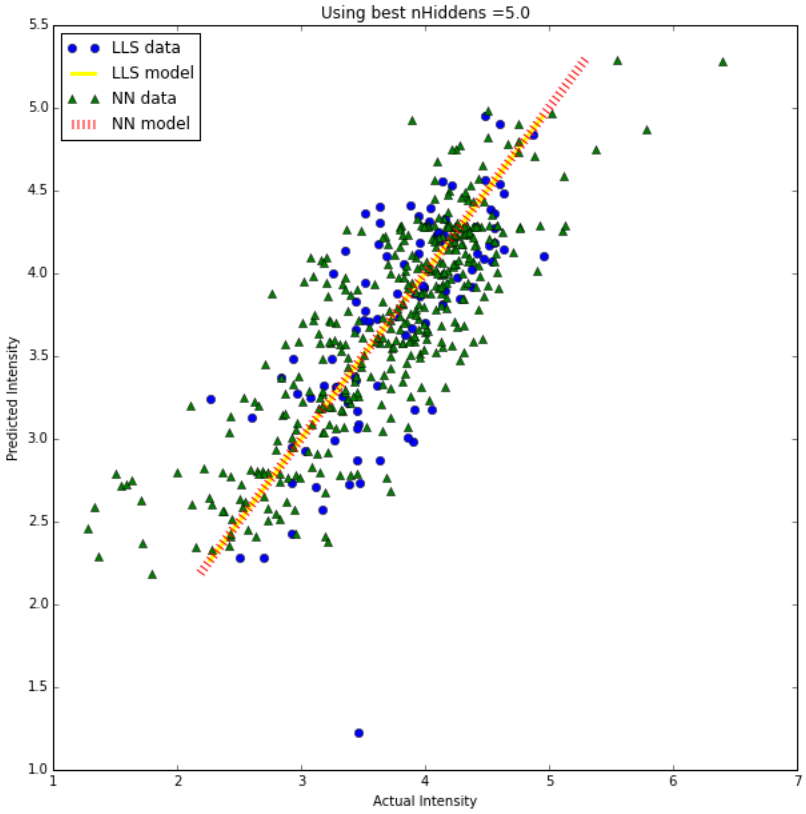

Figure 6: Linear versus Neural model

4.4 Hidden units versus RMSE

it can be seen from figure-5 that the test error shoots up for hidden value 20, but it behaves well for other values and also we obtain the best hidden value as 1 as seen in all the 3 traces. Using the validate result's observation, when taken 1 as the best value, the error decreased well in the testing data.

4.5 Additional

The figure-8 shows the RMSE versus hidden. Here too, we obtain 1 as the best value. But, the test value behaves very well in this case. With 1 as the hidden value, a comparison with the linear model yielded the figure-9. It is interesting that the regression line coincides. But, the alignment of the data are perpendicular

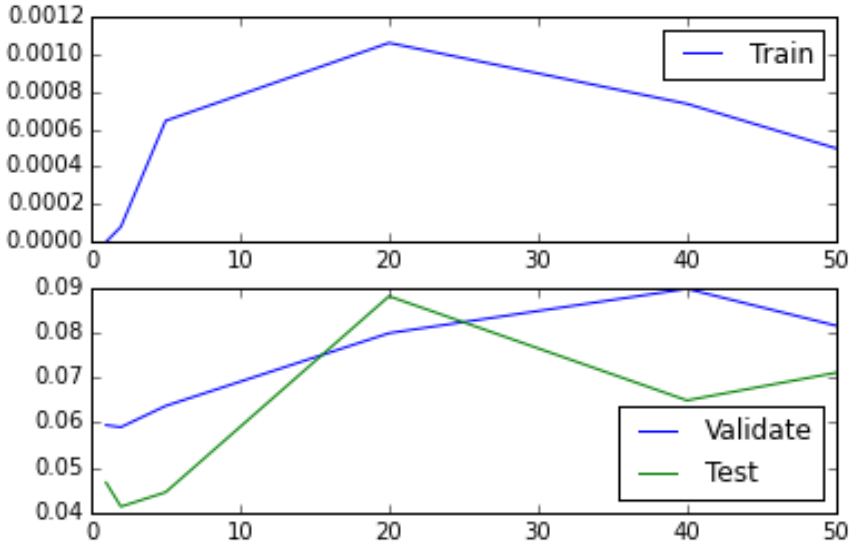

Figure 7: Hidden units versus RMSE

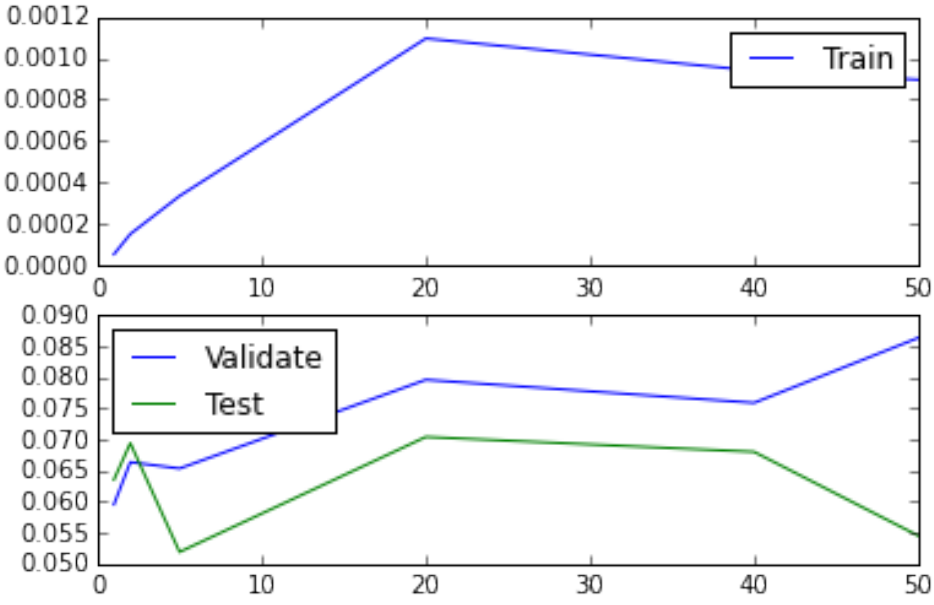

Figure 8: RMSE vs Hidden units

to each with respect to the linear and Neural model. It is interesting to observe the pattern but I don't have any intuition to explain this. The new graph is shown in the Figure 10. RMSE is less than the previous.

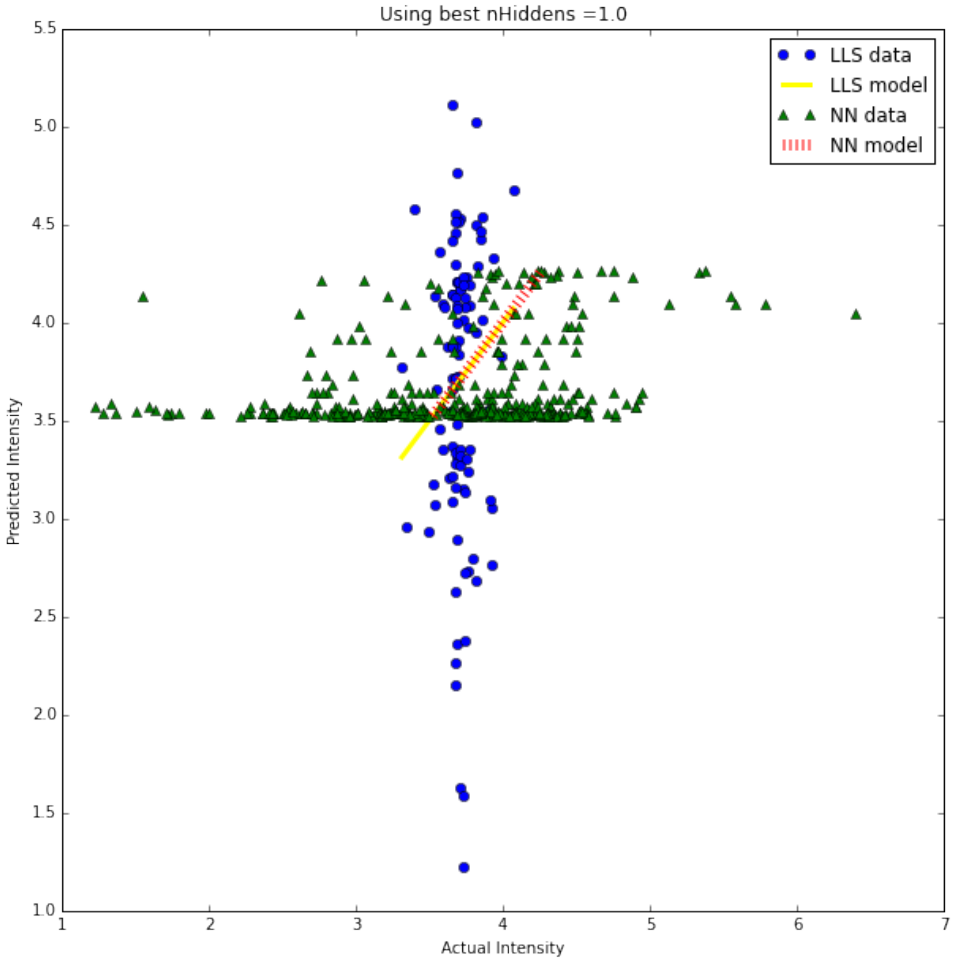

Figure 9: Linear versus Neural model

5 Conclusion

Thus, the non-linear regression with Neural Networks have been studied and results reported. It is surprising that the selected example ha=s behaviour more similar to the linear model. But, analysing the model with respect to individual attributes might bring out the clear cut non-linearity hidden.

References

[1] Christopher M.Bishop., "Pattern Recognition and Machine Learning", Springer, 2006.

[2] `http://lib.stat.cmu.edu/datasets/`

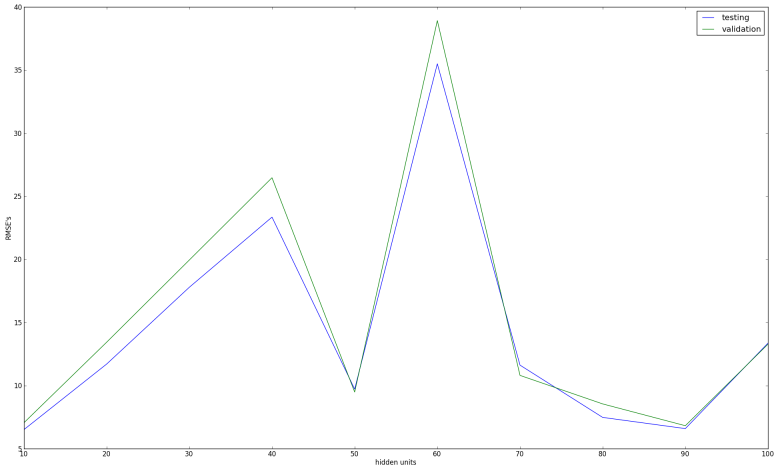

Figure 10: RMSE vs Hidden units