**PRACTICAL-1**

## **AIM:-** To study and explain Cast XL tool for neural network.

**Step 1: Preparing Data**

The input data to neural network is not always hard physical measurements. Scaling the data sometimes is needed so that the neural network can learn better. 4Cast XL use a sigmoidal activation function as its output. Therefore, user need to scale the input data first so that the target activations to values can be comfortably learned. Please take note that 4Cast XL only process numeric data.

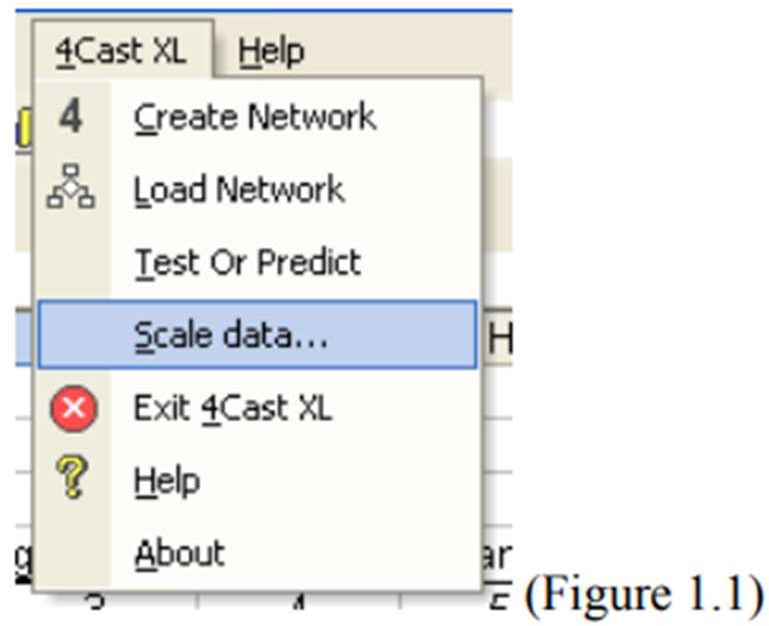
All the data need to be converted to the between 0 to 1. Neural network will learn better if there is uniformity in the data.

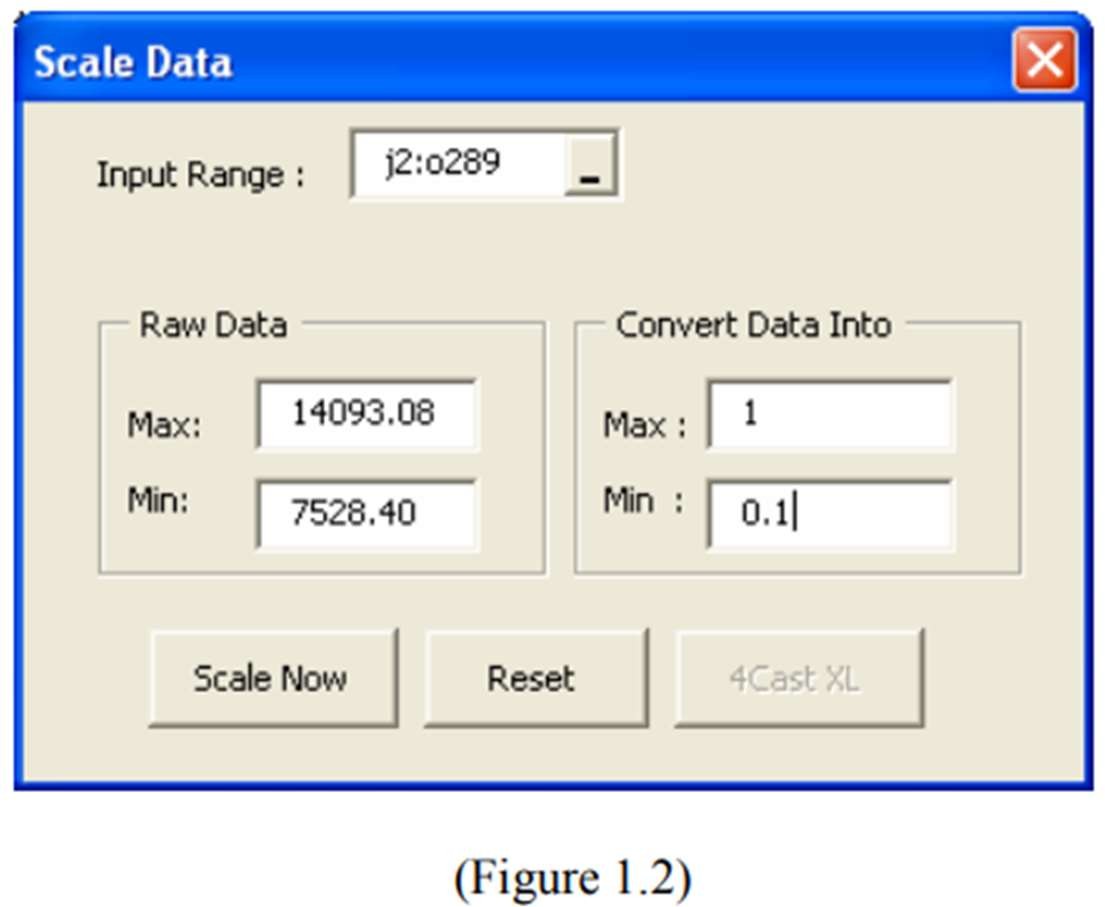
Here, a real-life example is use. This data is taken from the DJIA weekly prices, from the period from 22 April 2002 to 22 Oct 2007.

Let’s organize the data first. The basic assumption for time series forecasting is that the pattern found from historical data will also hold in the future. In practice, we partition the data into two parts.

The first part is used to train the neural network, while the second part of data is used to test the performance of the model. A model is considered good if the error of out-of-sample testing is the lowest compared with the other models. If the trained model is the best one for validation and also the best one for testing, one can assume that it is a good model for future forecasting.

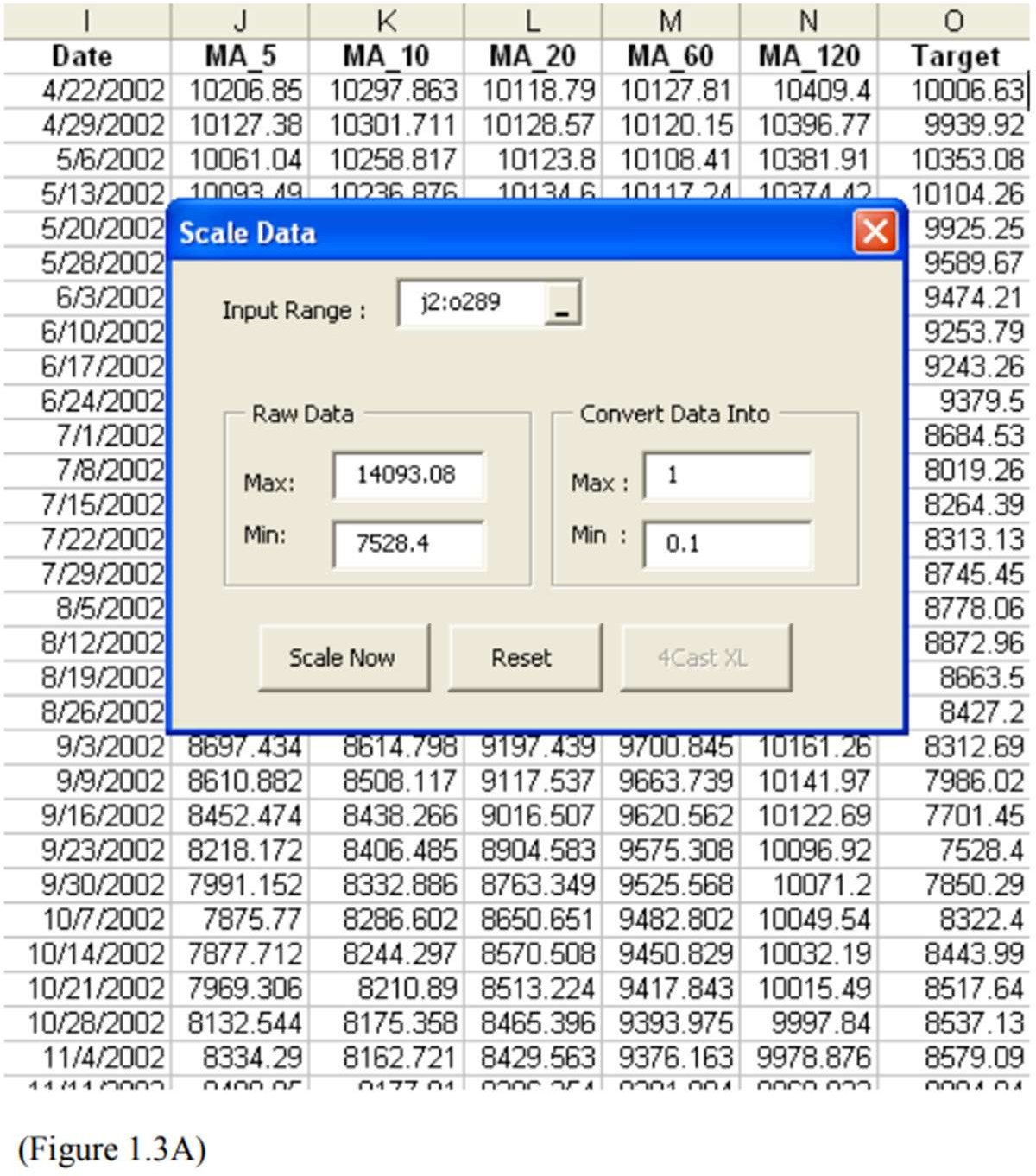
There are tradeoffs for testing and training. One should not say it is the best model unless he has tested it, but once he has tested it, he has not trained enough. The general partition rule for training and testing set is 70% and 30%, respectively.

1. Scaling of variables and data is automated. Just select Scale Data on the 4Cast XL menu (see Figure 1.1) and the input From below (Figure 1.2) will be shown:



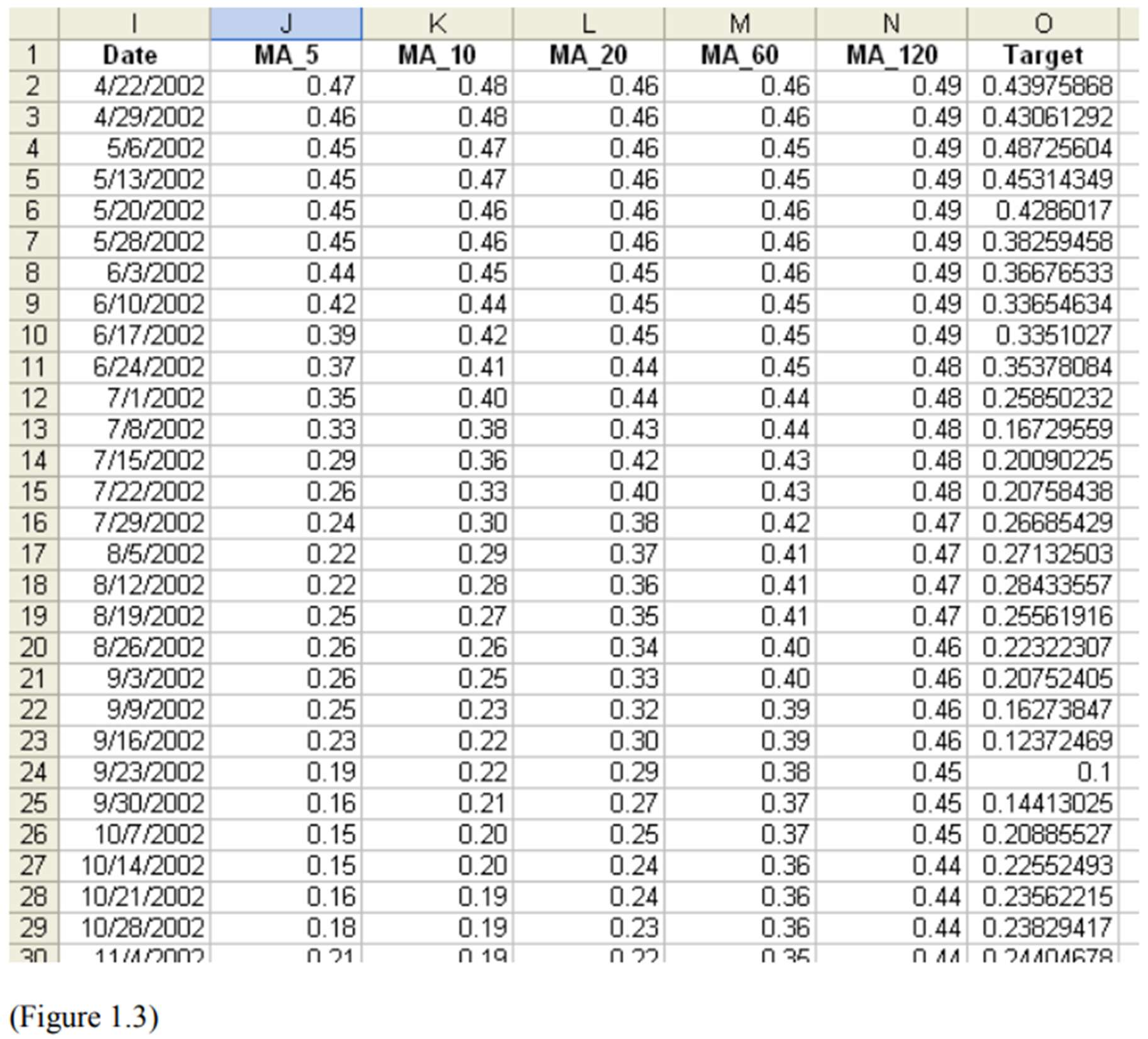
For column J, the input data consist of the 5 days moving averages of the DJIA weekly prices, column K consist of 10 days moving averages and so on. The highest price during the period from 22 April 2002 to 22 Oct 2007 is 14,093.08 and the lowest price is 7528.4.

288 weekly data is entered accordingly from column J2 to N289. Column O is the output/target data consist of actual weekly closing.

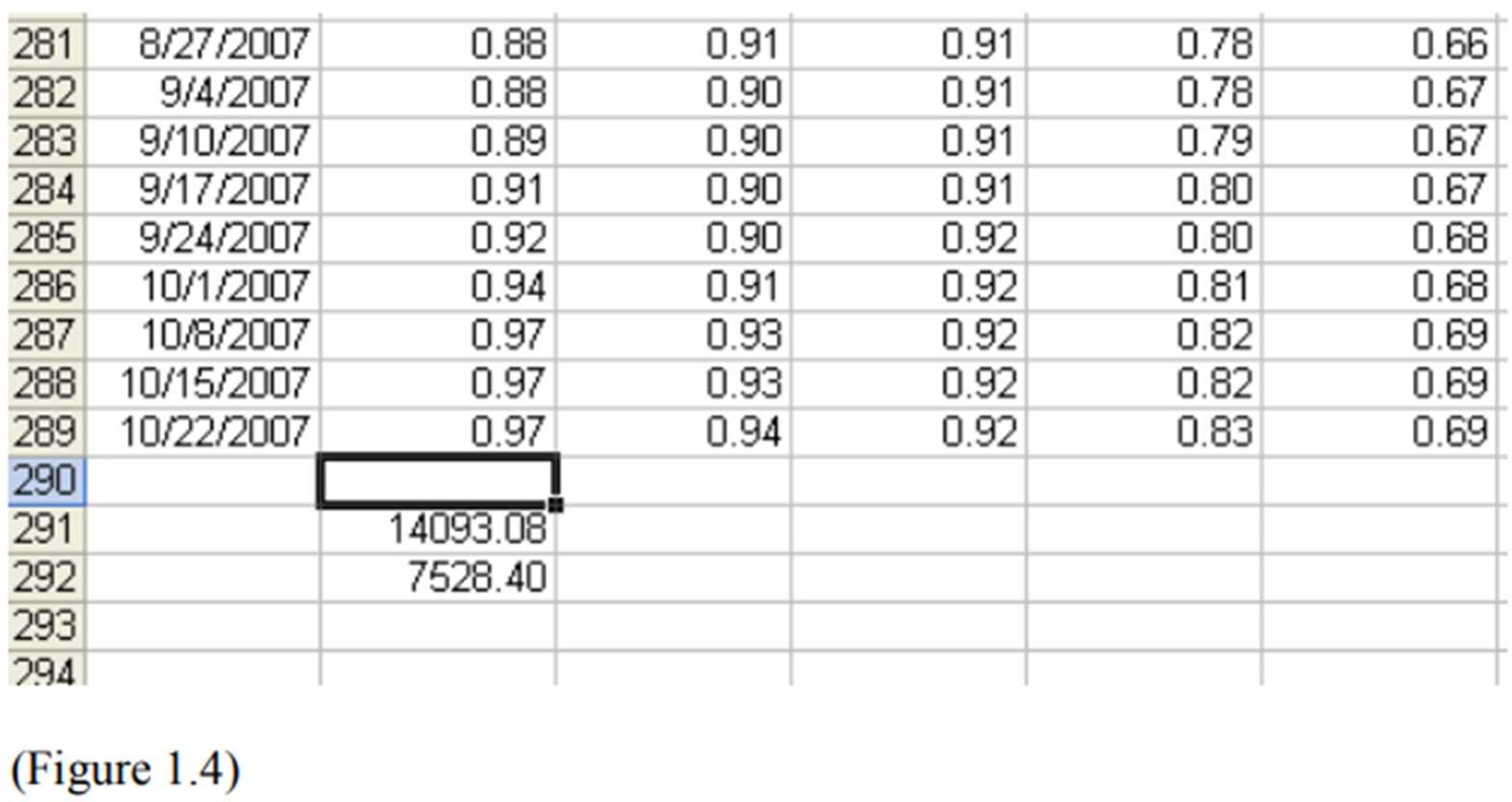


Thus we need to convert the data in the range J2:O289 (see Figure 1.3A ). That is 6 columns times 288 rows in the worksheet “Raw Data”. Enter J2:O289 into the Input Range edit box. Press the Tab key on your keyboard to exit. When you press Tab, 4Cast XL will automatically load the maximum (14,093.08) and the minimum (7528.4) in the Raw Data frame Min and Max textbox.

By default in the Convert Data Into frame, 4Cast XL use the value 1 for maximum and 0.1 for minimum (see Fig 1.2) above. Of course you can change this. It is advisable not to use the value 0 as the minimum as it represent nothing.

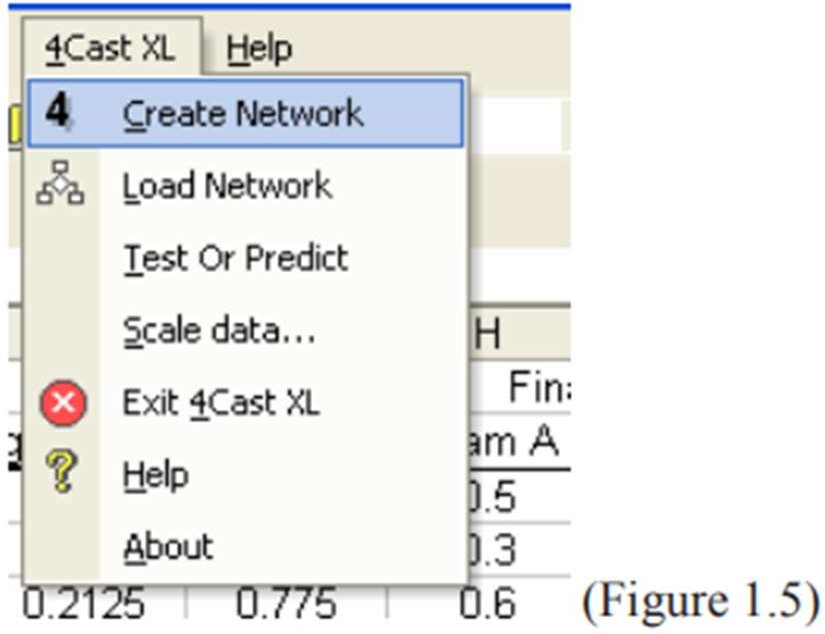
Click on the Scale Now button and all the data in the range J2:O289 will be converted to 0.01 to 1. I’ve format the converted values into 2 decimal points. (see Figure 1.3 below).

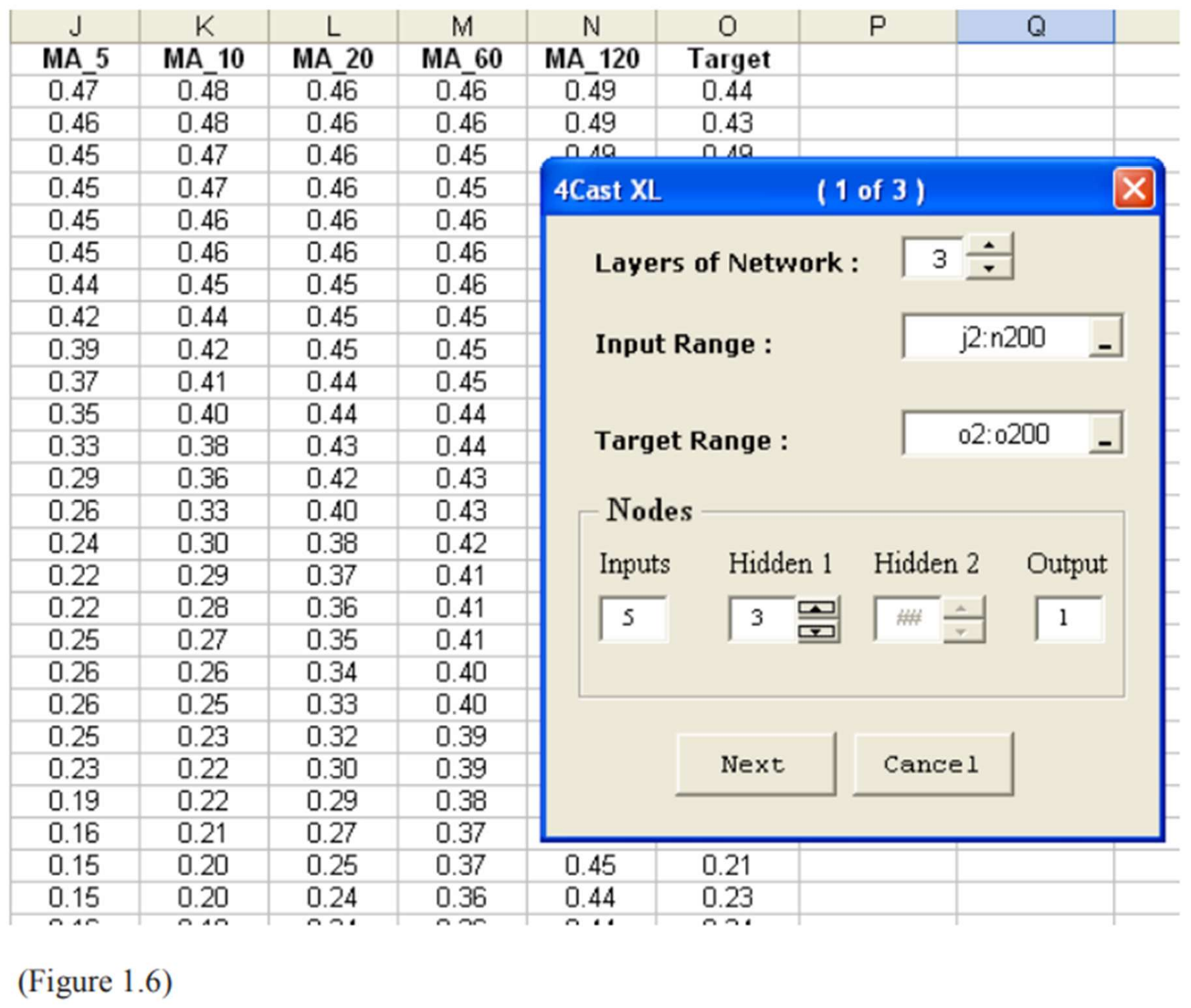
4cast XL will also automatically store the minimum (in cell J292) and the maximum (cell J291) value of the raw data in the last row and first column of the raw data. (see Figure 1.4 below).Delete the value in cell O289 because this is the value or weekly price that we want to predict.

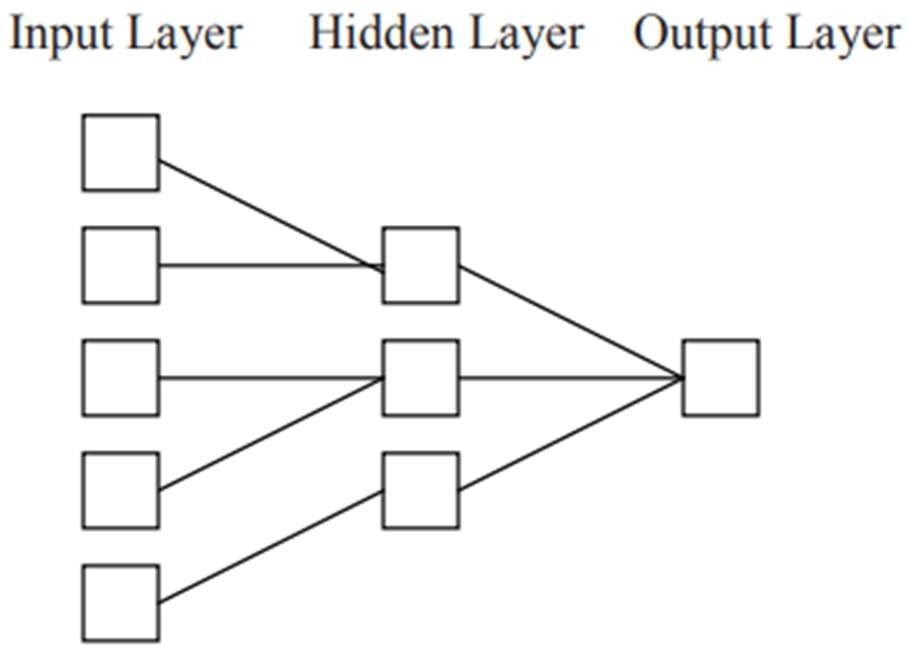


1. After converting the raw data, click on the X at the top right of dialog form to exit. Now we need to partition the data into 2 parts.
   1. Part 1, row 2 to row 200 as the training data
   2. Part 2, row 201 to row 288 as the testing data. User can start to build the neural network model after this.

### Step 2: Building the architecture of the neural network model.

1. Select Create Network on the 4Cast XL menu (see Figure 1.5) and the dialog box 4Cast XL (1 of 3) will be displayed (see Figure 1.6).

* Select 3 on the Layers of Network. Proven to be very effective on timeseries forecasting.
* On the Input Range edit box, enter the range J2:N200 (five columns of inputs and 199 rows of input pattern, i.e. the training data). Each time you exit, please use the Tab key on your keyboard.
* On the Target Range edit box, enter the range O2:O200 (1 column of output/target and 199 rows of pattern to be learned). Tab!!! When you tab to exit, 4Cast XL will automatically load the number of input column in the Inputs textbox on the Nodes frame. (Here we’ve 5 inputs)
* On the Hidden 1 textbox of the Nodes frame, enter 3. This mean that we have select 3 nodes for the hidden layer. The general rule to follow here is that we use the pyramidal scheme, 5-3-1. (you can also try 5-4-1, 5-2-1 etc)
* The Output textbox on the Nodes frame is already loaded with the value 1 as this is done automatically by 4Cast XL as well.
* Click the Next button. Graphically, it looks like this: (see below)



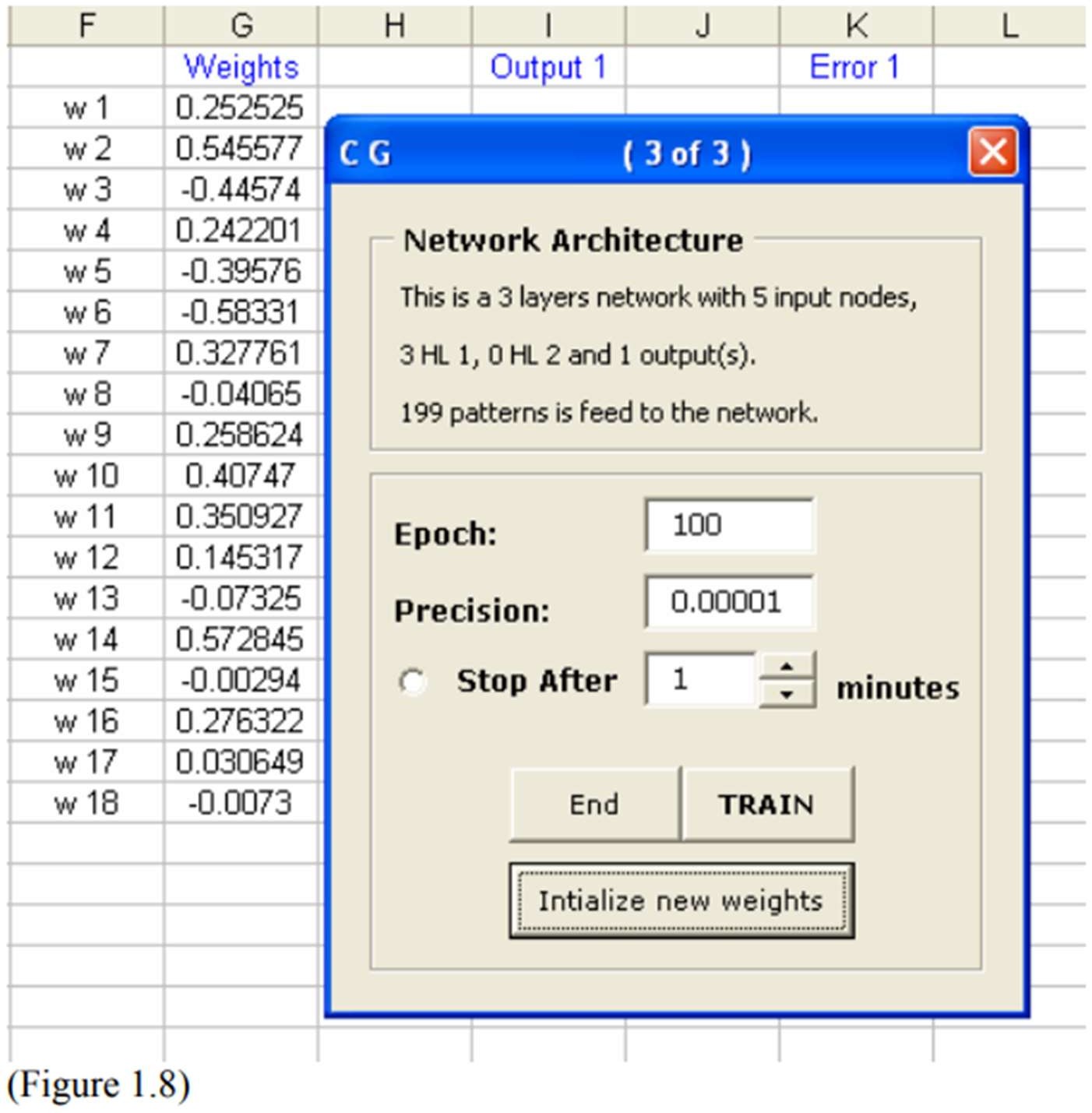
Click on the Next button and the dialog box 4Cast XL (2 of 3) below will be shown(see Fig 1.7). The worksheet Network Output will be created. This worksheet is very important as it is use to store the the model architecture(display on the Network Architecture frame), the weights(w1 to w18 ) and all other important parameters. (see Figure 1.7)



Currently, the worksheet is blank as we have not start training yet.

1. Select the training option that you want in the Training Methods frame. Training option includes Backpropagation with momentum, Conjugate Gradient and Genetic Optimization. Select the checkbox Show MSE Graph to show the training error in each epoch. (the Genetic Optimization training option will be discussed in Example 2)Choose the Conjugate Gradient training method. Click the Next button again and the Conjugate Gradient training method dialog form (see Fig 1.8 below) will be display.

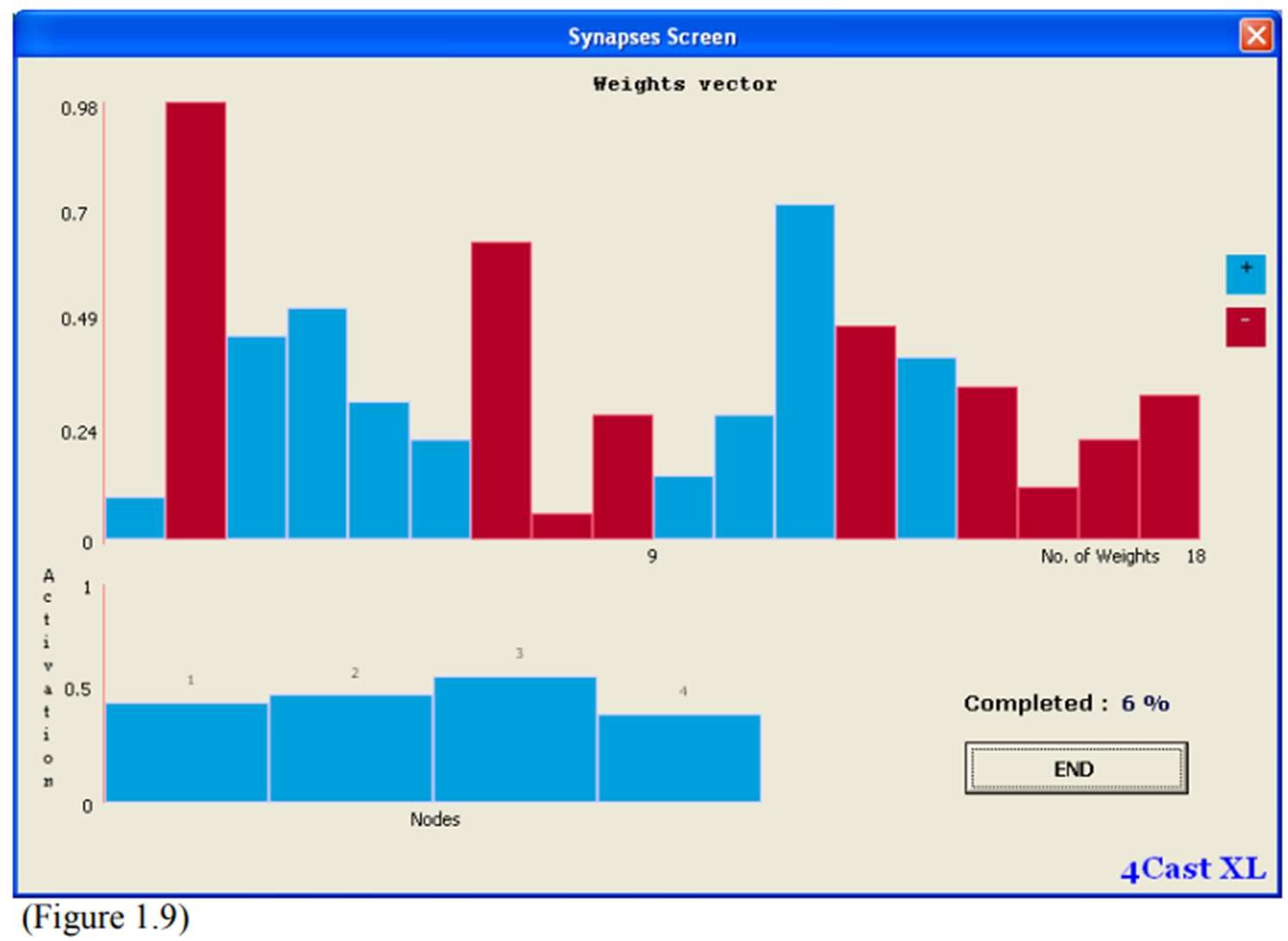
### **Step 3: Training the network model**



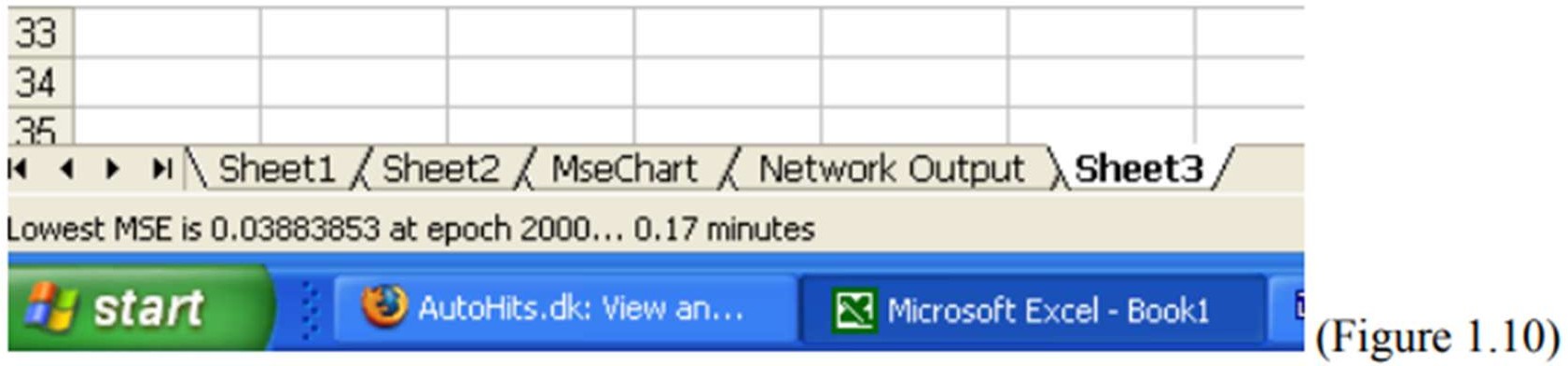
Select the Initialize new weights. 18 rows of value for the weights will be generated in G2:G19. (Note: This is optional as 4Cast XL will generate the weights for you automatically if you didn’t click on this button)

You also have the option to run the program for how many loops / Epoch (maximum for this program is 15000) or you can select the option to run the program for how many minutes. (maximum 300 minutes). User can stop training prematurely by clicking the End button.( Note: MSE graph will not be updated if user do this. This might take a few seconds.)

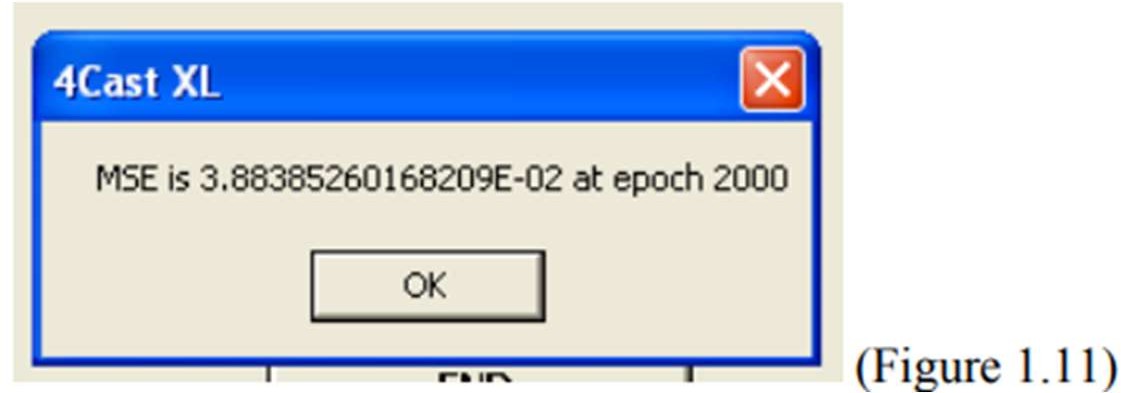
Click the Train button to start training the model. The Synapses Screen will be shown.(see Fig 1.9 below) and the program will start running.



You can see the activities of the neural network model training. For experience user, they will know whether a suitable model has been built, whether enough hidden layers is created, whether the network has good starting weights etc, just by looking at this screen. 4Cast XL is the only neural network program in the world that has this feature.



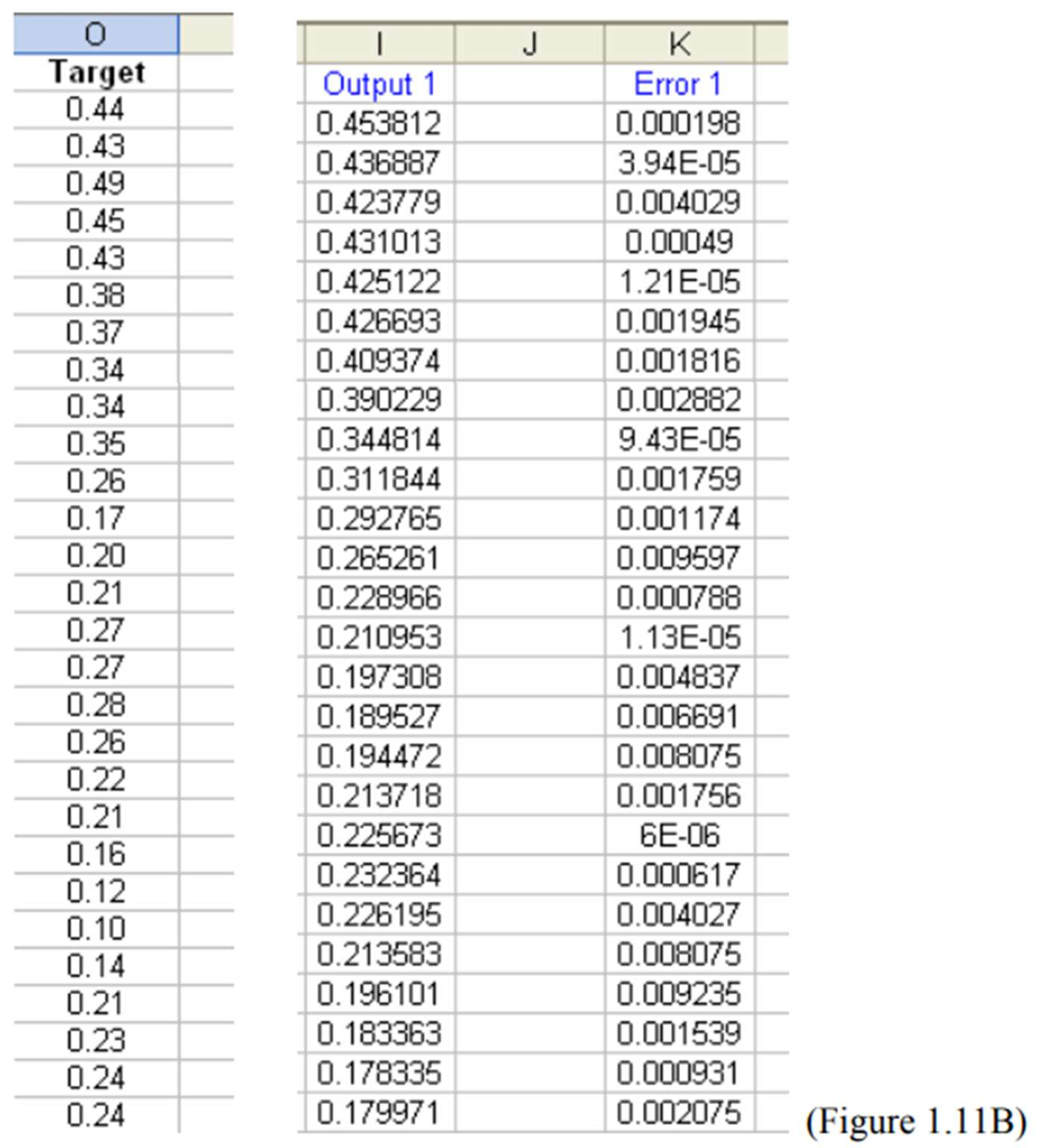
The number of Epoch and the Mean Square Error (MSE) will be shown in the Status Bar (see Figure 1.10 above)A message box will pops up after the model has finish training for the current session. (see Fig 1.11).



User can stop training prematurely by clicking the End button.( Note: MSE graph will not be updated if user do this. This might take a few seconds.)

Here it show 0.003883 which is not satisfactory. Usually we need the MSE value to be below or after training for 30000 epochs. As the result is not satisfactory we need to train the network model again. Before that, take a look at the Network Output worksheet. You’ll see all the parameters generated on this sheet. Don’t change anything here.

A complete analysis is presented in the Network Output worksheet. User can view and analyse the network model.(e.g. Each pattern output and error is presented thus, user can determine whether the network has learn well for a particular pattern.) In the hidden layer output, user will know which node is redundant or active. Weights of the network are also reported. User can also use the data here for graphing purposes, for example the Actual Vs Forecasted chart, Sensitivity of Inputs chart, error distribution chart, correlation graph etc.

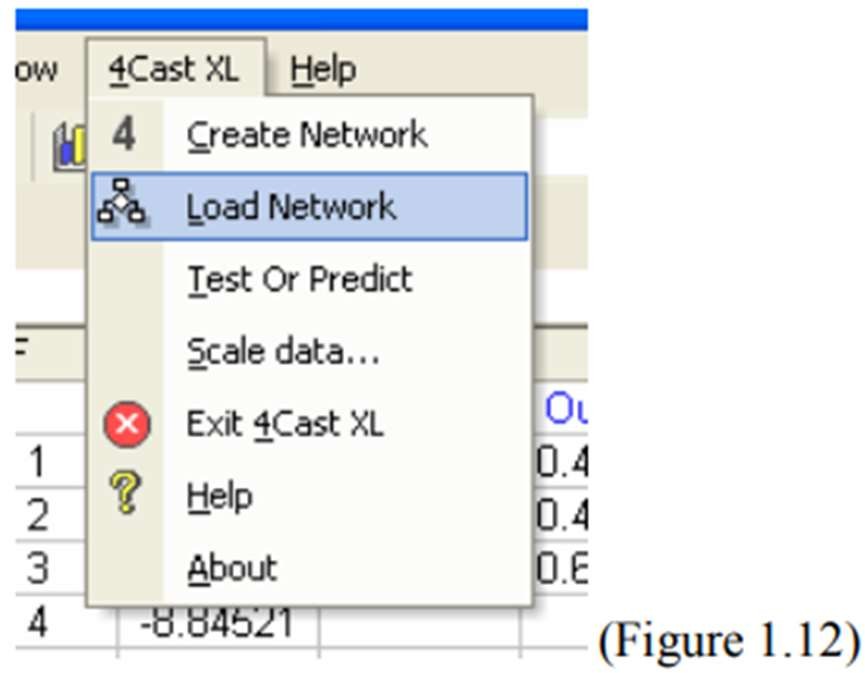
For example, if you want to built the Actual VS Forecasted chart, goto the Network Output worksheet and copy the output (Output 1 in column I ) and paste it beside the Target column in the data sheet. (see Figure 1.11B below).

As the result is not satisfactory, we need to continue training the network model.

### **Step 4: Retraining the Network Model.**

Here we do not need to rebuilt the whole network again. Make sure you have select the sheet that contain the training data as the active sheet before execute the step below.

* + - Select Load Network on the 4Cast XL menu (see Fig 1.12 below) and the dialog box 4Cast XL (2 of 3) will be displayed (see Figure 1.7 above)



* + - Choose the Conjugate Gradient training method and go through the whole process again until you’ve train the model for 20000 to 30000 epoch or when the MSE reach below 0.001.

Do not click on the Initialize New Weights unless you want to change a new set of weights. Here 4Cast XL will use the weights contain in the weight i.e.G2:G19 generated from the previous training session.

(For advance user, you can change the weights vector by selecting the Initialize New Weights to initialize new weights if you feel that the current model does not have good starting weights)

If you have no confidence on the network model and need to change or create a new model altogether with different architecture then goto Step 5 else goto Step 6.

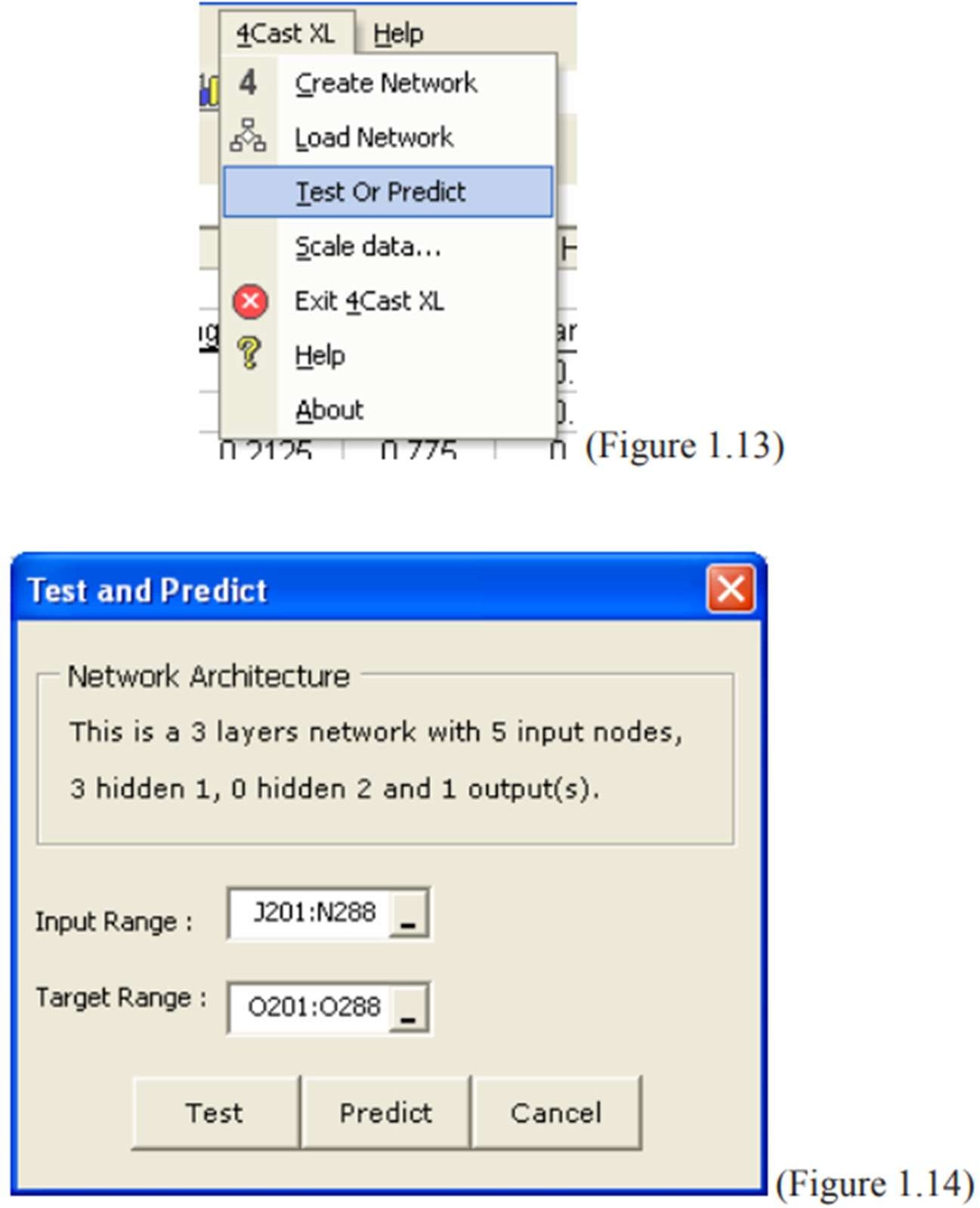
**Step 5 : Re-Build the Neural Network Model**

1. It is advisable for user to use different workbook to create different network architectures.
2. Or user can rename the Network Output worksheet if new network architecture is being created in the same workbook. 4Cast XL will also prompt user to rename the Network Output worksheet if the new model is being built on the same workbook.
3. Go through Step 2 , 3, 4 again until you are satisfied with the result.
4. If you are satisfied with the result then go to Step 6.

### **Step 6: Testing**

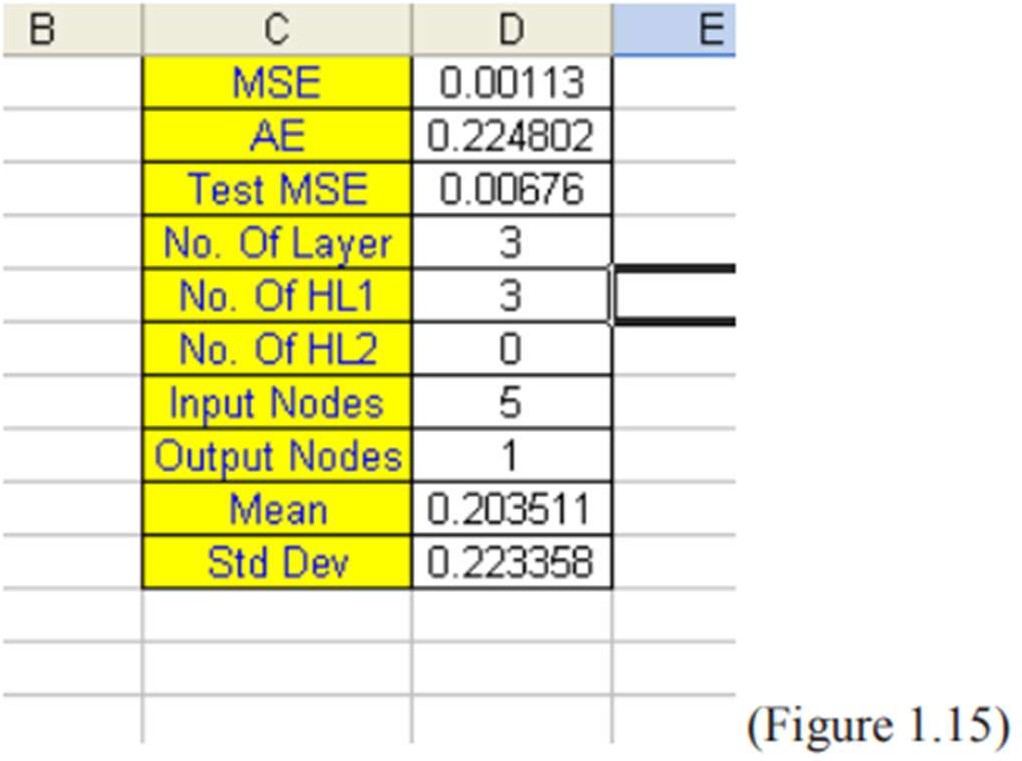
After you are satisfied with the training result, its time to test the neural network model. For our current example, the network model was trained for 10,000 epochs and the MSE has reach 0.00113.

1. Select Test or Predict on the 4Cast XL menu (see Fig 1.13) and the dialog box Test and Predict will be display. (see Figure 1.14).
2. Enter J201:N288 in the Input Range edit box
3. Enter O201:O288 in the Target Range edit box



Remember that we have allocated row 201 to 288 as the test set.

1. Select the Test button. 4Cast XL will run the test and pops up a message box displaying the Test Error and will also be stored on the Network Output worksheet (see Fig 1.15)



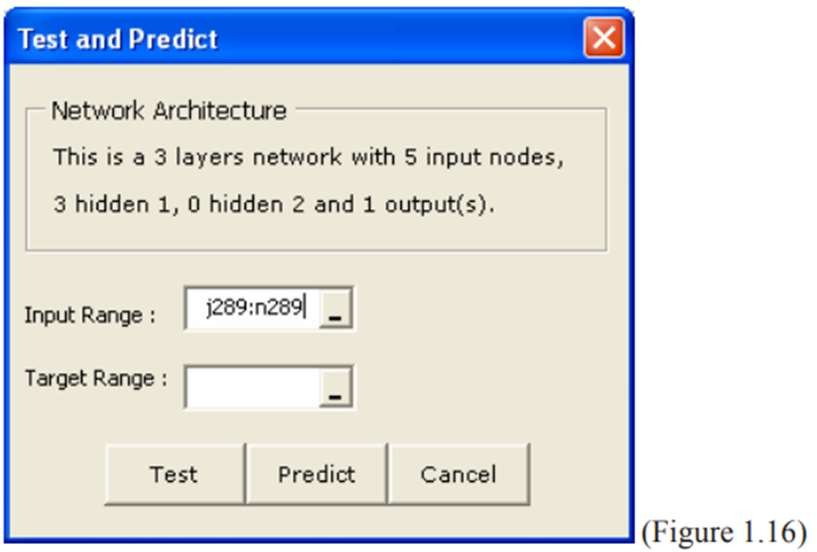
Here the test error is 0.00676 (see Figure 1.15) which is also display on the Network Output sheet. The MSE is 0.00113. The general rule here is that the test error and the actual MSE should not have a big difference. For example MSE = 0.00067 and Test error is 0.00089, this can be consider good modeling and suitable to be used as a forecasting model in real life scenario.

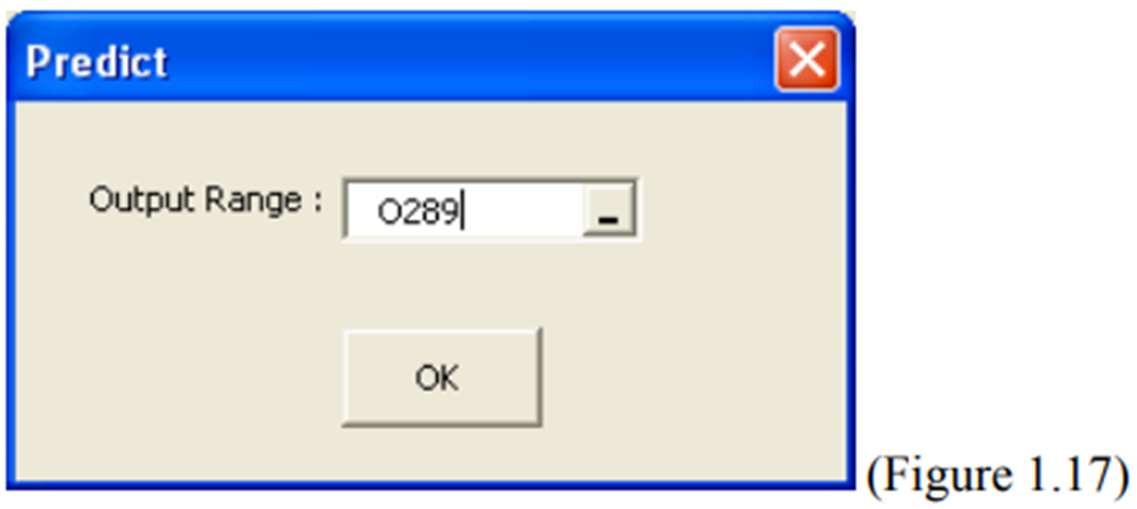
If you have no confidence on the result or after training for 20,000 to 30,000 epochs and the network model show no improvement, then you need to re-built the neural network and go through Steps 2,3,4 and 6. Neural network forecasting requires lots of training and testing before you can apply it to real life situations.

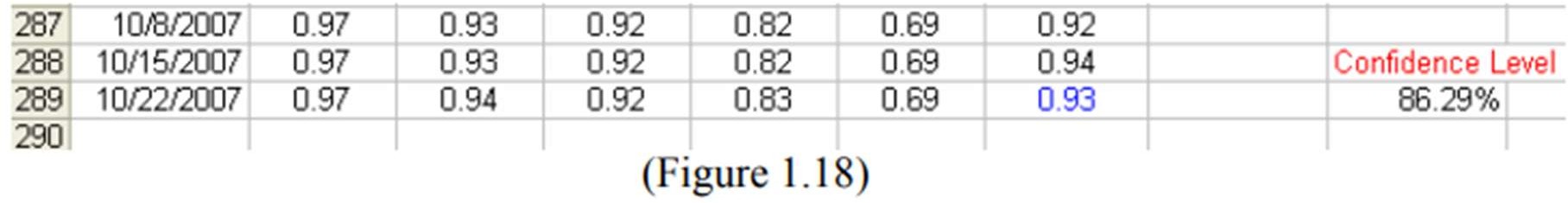
You will need some time to learn and to experience before you can gain the confidence to use neural network as a forecasting tool. Once you’ve became an expert user, you will be amaze when neural network shows its magic!!!

The result of the current model is quite satisfactory, after training and testing, its time now to do the prediction.

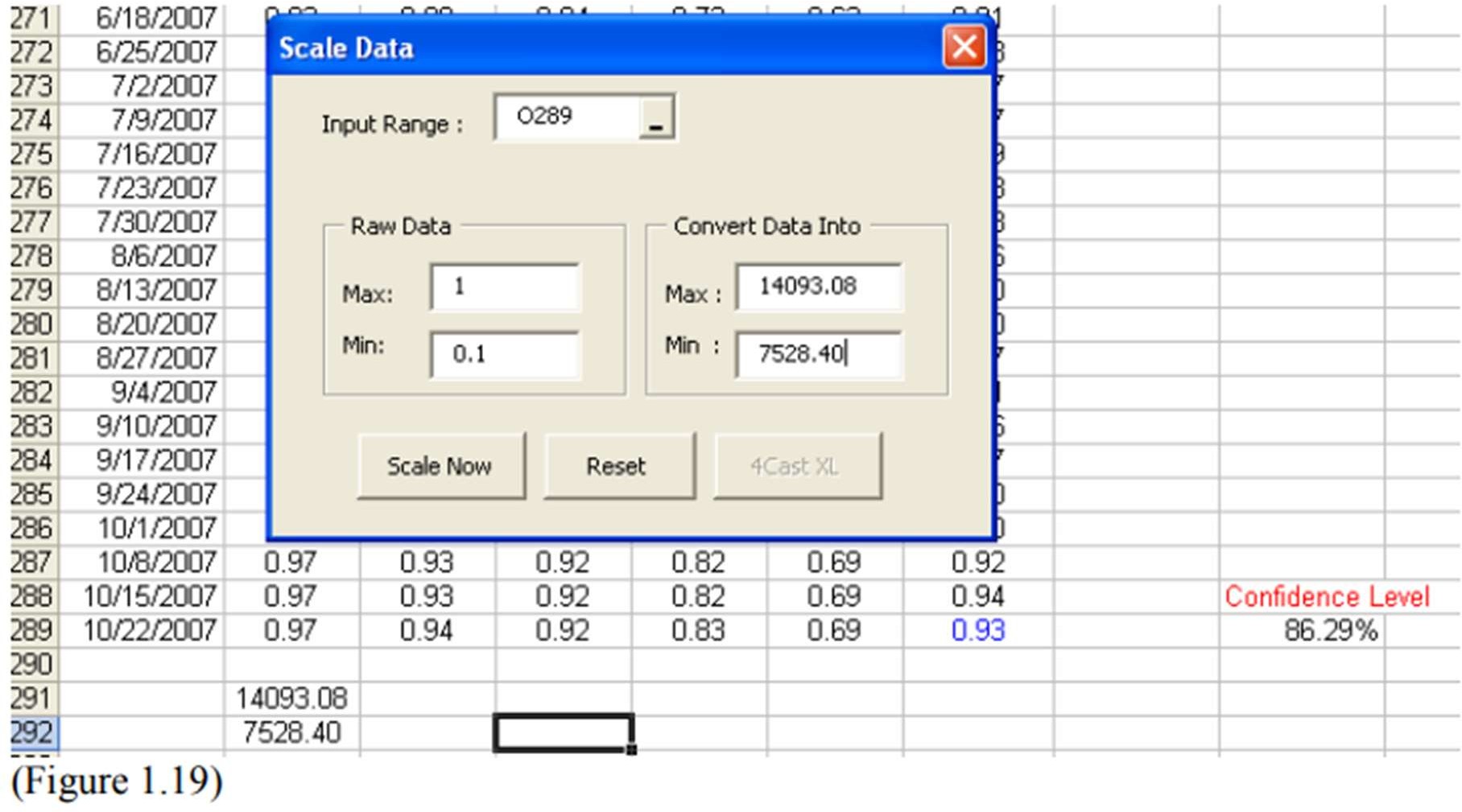
### **Step 7: Predicting**

1. Select Test or Predict on the 4Cast XL menu (see Fig 1.13 above) and the dialog box Test and Predict will be display. (see Figure 1.16 below). Enter J289:N289, i.e. the 5 input factors to predict the outcome/result ( i.e. 1 row of MA5, MA10, MA20, MA60 and MA120 ). Leave the Target Range blank
2. Click on the Predict button to bring up the Predict dialog box (Figure 1.17), enter O289 so that the result/output of the prediction will be contained on this cell.



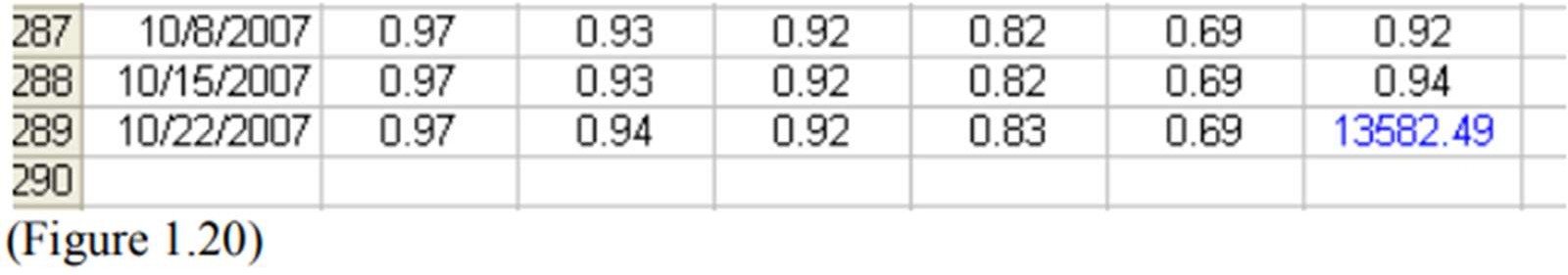
1. Click the OK button and the result will be displayed on the cell O289 in blue color font. The Confidence Level of 86.29% is also shown next to it. (Fig 1.18)
2. Next we need to convert this result/output back to the Dow Jones weekly price. (as 0.93 doesn’t mean anything to us)

Select Scale Data on the 4Cast XL menu (see Figure 1.1 above) and the dialog form below (Figure 1.19) will be shown: (we also do this step when we first convert the raw data)



Remember that 4Cast XL has stored the minimum in cell J292 and maximum in cell J291 of the raw data when we first convert the data in Step 1. Now we reverse the process. (see Figure 1.19)

1. Click the Scale Now button and we have the predicted result as 13582.49, (0.93 become 13582.49…see Figure 1.20 below)



To conclude, 4Cast XL predict that next week DJIA closing price is 13582.49.

**PRACTICAL-2**

**AIM:-** Create a simple ADALINE network with appropriate no. of input and output nodes. Train it using delta learning rule until no change in weights is required. Output the final weights.

#include <stdio.h>

#include <stdlib.h>

int main()

{

int inputs[4][2]={{-1,-1},{-1,1},{1,-1},{1,1}},outputs[4]={-1,1,1,1},x,y,epochs=10,actual;

float learning\_rate=0.2,weight[2]={1,1},bias=0,error=0,x1=0,x2=0,sum\_squared\_error= 0,unit;

printf("ketul Shah\n");

printf("Input array:\n");

for(x=0;x<4;x++){

for(y=0;y<2;y++)

{ printf("%d\t",inputs[x][y]);

}

printf("\n");

}

printf("Output array:\n");

for(x=0;x<4;x++){

printf("%d\t",outputs[x]);

}

for(x=0;x<10;x++){

sum\_squared\_error=0; for(y=0;y<4;y++){

actual=outputs[y];

x1=inputs[y][0];

x2=inputs[y][1];

unit=(x1\*weight[0])+(x2\*weight[1])+bias;

error=actual-unit;

sum\_squared\_error=sum\_squared\_error+(error\*error);

weight[0]=weight[0]+(learning\_rate\*error\*x1);

weight[1]=weight[1] + (learning\_rate \* error \* x2);

bias=bias+(learning\_rate\*error);

}

}

printf("\nWeights: ");

printf("[%f\t%f]",weight[0],weight[1]);

printf("\nBias: %f",bias);

printf("\nSum squared error: %f",sum\_squared\_error/4);

int output[4]={-1,-1,-1,1};

printf("\n\nInput array:\n");

for(x=0;x<4;x++){

for(y=0;y<2;y++)

{ printf("%d\t",inputs[x][y]);

}

printf("\n");

}

printf("Output array:\n");

for(x=0;x<4;x++){

printf("%d\t",output[x]);

}

learning\_rate=0.2;

float weights[2]={1,1};

bias=0;

for(x=0;x<10;x++){

sum\_squared\_error=0;

for(y=0;y<4;y++){

actual=output[y];

x1=inputs[y][0];

x2=inputs[y][1];

unit=(x1\*weights[0])+(x2\*weights[1])+bias;

error=actual-unit;

sum\_squared\_error=sum\_squared\_error+(error\*error);

weights[0]=weights[0]+(learning\_rate\*error\*x1);

weights[1]=weights[1] + (learning\_rate \* error \* x2);

bias=bias+(learning\_rate\*error);

}

}

printf("\nWeights: ");

printf("[%f\t%f]",weights[0],weights[1]);

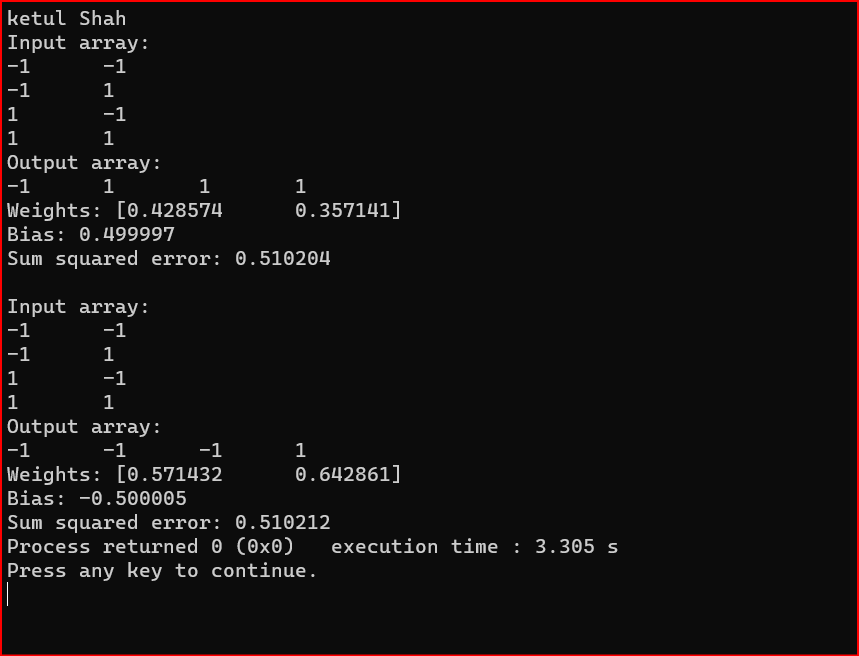
printf("\nBias: %f",bias);

printf("\nSum squared error: %f",sum\_squared\_error/4);

return 0;

}

**Output:-**

****

**PRACTICAL-3**

**AIM:-** Train the auto correlator by given patterns: A1=(‐1,1,‐1,1), A2=(1,1,1,-1), A3=(‐1, ‐1, ‐ 1, 1). Test it using patterns: Ax=(‐1,1,‐1,1), Ay=(1,1,1,1), Az=(‐1,‐1,‐1,‐1).

#include<stdio.h>

#include<conio.h>

int a1[1][4],a2[1][4],a3[1][4],p[1][4],pp[1][4],T[4][4]={0},s=0;

int i,j;

void printline(){ int p;

for(p=0;p<25;p++) printf("--");

printf("\n");

}

a1[1][4]={-1,1,-1,1};

a2[1][4]={1,1,1,-1};

a3[1][4]={-1,-1,-1,1};

void main()

{

printf("Ketul Shah \n");

printf("=============AUTOCORRELATORS============\n\n");

for(i=0;i<=3;i++){

for(j=0;j<=3;j++){

T[i][j]=T[i][j]+a1[0][i]\*a1[0][j];

}

}

for(i=0;i<=3;i++){

for(j=0;j<=3;j++){

T[i][j]=T[i][j]+a2[0][i]\*a2[0][j];

}

}

for(i=0;i<=3;i++){

for(j=0;j<=3;j++){

T[i][j]=T[i][j]+a3[0][i]\*a3[0][j];

}

}

printf("<----------Matrix >\n");

for(i=0;i<=3;i++){

for(j=0;j<=3;j++){

if(T[i][j]>0)

printf(" %d ",T[i][j]);

else

printf("%d ",T[i][j]);

}

printf("\n");

}

printline();

printf("Enter any pattern to match:\n");

for(j=0;j<4;j++){

scanf("%d",&p[0][j]);

}

for(i=0;i<4;i++){

for(j=0;j<4;j++){

s=s+T[i][j]\*p[0][j];

}

if(s>0)

pp[0][i]=1;

if(s==0)

pp[0][i]=p[0][i];

if(s<0)

pp[0][i]=-1; s=0;

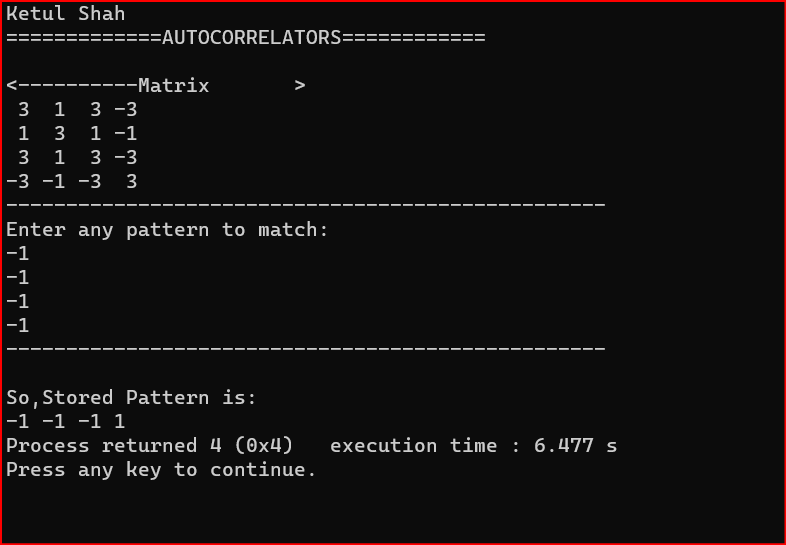
}

printline();

printf("\nSo,Stored Pattern is:\n"); for(j=0;j<4;j++)

printf("%d ",pp[0][j]);

}

**OutPut:-**

**PRACTICAL-4**

## **AIM:-** Train the heterocorrelator using multiple training encoding strategy for given patterns:A1=(000111001) B1=(010000111), A2=(111001110) B2=(100000001), A3=(110110101), B3(101001010).Test it using pattern A2.

#include<iostream>

#include<stdio.h>

#include<stdlib.h>

#include<process.h>

#include<conio.h>

#include<math.h>

#include<iomanip>

using namespace std;

int phi(int a){

if(a>0){

return(1);

}

else if(a==0){

return(a);

}

else{

return(-1);

}

}

int main(){

system("cls");

int a[3][9]={{0,0,0,1,1,1,0,0,1},{1,1,1,0,0,1,1,1,0},{1,1,0,1,1,0,1,0,1}},b[3][9]={{0,1,0,0,0,0,1,1,1},{1,0,0,0,0,0,0,0,1},{1,0,1,0,0,1,0,1,0}};

int x[3][9]={0}, y[3][9]={0}, M[9][9]={0}, j=0, k=0, i=0, sum=0, alpha[9]={0}, alphaM[9]={0}, betaDASH[9]={0}, betaDASHMT[9]={0}, alphaDASH[9]={0};

int alphaDASHM[9]={0}, betaDOUBLEDASH[9]={0};

cout<<"Ketul shah \n";

for(i=0;i<3;i++){

for(j=0;j<9;j++){

if(a[i][j]==0){

x[i][j]=-1;

}

else if(a[i][j]==1){

x[i][j]=1;

}

}

}

for(i=0;i<3;i++){

for(j=0;j<9;j++){

if(b[i][j]==0){

y[i][j]=-1;

}

else if(b[i][j]==1){

y[i][j]=1;

}

}

}

for(i=0;i<9;i++){

for(j=0;j<9;j++){ sum=0; for(k=0;k<3;k++){

sum+=(x[k][i]\*y[k][j]);

}

M[i][j]=sum;

}

}

cout<<"The Connection Matrix Is As Follows:\n";

for(i=0;i<9;i++){

for(j=0;j<9;j++){

cout<<M[i][j]<<"\t";

}

cout<<endl;

}

cout<<"\nTaking 'Alpha'=X2, We Will Try To Retrieve Y2 As 'Beta'\n";

cout<<"alpha Is As Follows:\n";

for(i=0;i<9;i++){

alpha[i]=x[1][i];

cout<<alpha[i]<<"\t";

}

cout<<endl; for(i=0;i<9;i++){

sum=0; for(j=0;j<9;j++){

sum+=(alpha[j]\*M[j][i]);

}

alphaM[i]=sum;

}

cout<<"\nalphaM Is As Follows:\n";

for(i=0;i<9;i++){

cout<<alphaM[i]<<"\t";

}

for(i=0;i<9;i++){

betaDASH[i]=phi(alphaM[i]);

}

cout<<"\nbetaDASH Is As Follows:\n";

for(i=0;i<9;i++){

cout<<betaDASH[i]<<"\t";

}

for(i=0;i<9;i++){

sum=0; for(j=0;j<9;j++){

sum+=(betaDASH[j]\*M[i][j]);

}

betaDASHMT[i]=sum;

}

cout<<"\nbetaDASHMT Is As Follows:\n";

for(i=0;i<9;i++){

cout<<betaDASHMT[i]<<"\t";

}

for(i=0;i<9;i++){

alphaDASH[i]=phi(betaDASHMT[i]);

}

cout<<"\nalphaDASH Is As Follows:\n";

for(i=0;i<9;i++){

cout<<alphaDASH[i]<<"\t";

}

for(i=0;i<9;i++){

sum=0;

for(j=0;j<9;j++){

sum+=(alphaDASH[j]\*M[j][i]);

}

alphaDASHM[i]=sum;

}

cout<<"\nalphaDASHM Is As Follows:\n";

for(i=0;i<9;i++){

cout<<alphaDASHM[i]<<"\t";

}

for(i=0;i<9;i++){

betaDOUBLEDASH[i]=phi(alphaDASHM[i]);

}

cout<<"\nbetaDOUBLEDASH Is As Follows:\n";

for(i=0;i<9;i++){

cout<<betaDOUBLEDASH[i]<<"\t";

}

cout<<"\nHere, betaDASH=betaDOUBLEDASH\nTHis, However Is An Incorrect Pattern Pair To Be Recalled\n";

return(0);

}

**Output:-**

**PRACTICAL-5**

## **AIM:-** Introductions to Python Basics with Numpy and Logistic Regression with a neural network mindset.

## **Part 1：Python Basics with Numpy**

## 1. Building basic functions with numpy

Numpy is the main package for scientific computing in Python. It is maintained by a large community ( www.numpy.org ). In this exercise we will learn several key numpy functions such as np.exp, np.log, and np.reshape. We will need to know how to use these functions for future assignments.

1.1 sigmoid function, np.exp()

Exercise: Build a function that returns the sigmoid of a real number xx. Use math.exp(x) for the exponential function.

## function.Reminder:  is sometimes also known as the logistic function. It is a non-linear

## function used not only in Machine Learning (Logistic Regression), but also in Deep Learning.

## To refer to a function belonging to a specific package you could call it using package\_name.function().

## Running the code below to see an example with math.exp().

## Code:-

# GRADED FUNCTION: basic\_sigmoid

import math

def basic\_sigmoid(x):

  s = 1 / (1 + math.exp(-x))

  return s

## Output:-

## 

Actually, we rarely use the “math” library in deep learning because the inputs of the functions are real numbers. In deep learning, we mostly use matrices and vectors. This is why numpy is more useful.

### One reason why we use "numpy" instead of "math" in Deep Learning ###

x = [1, 2, 3]

basic\_sigmoid(x) # you will see this give an error when you run it, because x is a vector.

In fact, if  x=(x1,x2,…,xn) is a row vector then **np.exp(x)**will apply the exponential function to every element of x. The output will thus be: *np.exp(x) =* (…. )

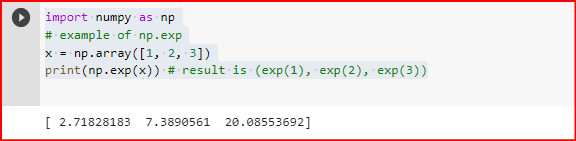
import numpy as np

# example of np.exp

x = np.array([1, 2, 3])

print(np.exp(x)) # result is (exp(1), exp(2), exp(3))

Output:-



Furthermore, if x is a vector, then a Python operation such as **s=x+3** or **s=1/x** will output **s** as a vector of the same size as **x**.

**Exercise**: Implement the sigmoid function using numpy.

**Instructions**: x could now be either a real number, a vector, or a matrix. The data structures we use in numpy to represent these shapes (vectors, matrices…) are called numpy arrays.



# GRADED FUNCTION: sigmoid

import numpy as np # this means you can access numpy functions by writing np.function() instead of numpy.function()

def sigmoid(x):

    """

    Compute the sigmoid of x

    Arguments:

    x -- A scalar or numpy array of any size

    Return:

    s -- sigmoid(x)

    """

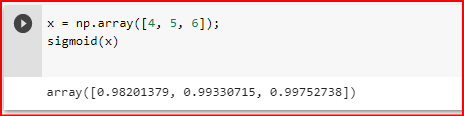
    ### START CODE HERE ### (≈ 1 line of code)

    s = 1 / (1 + np.exp(-x));

    ### END CODE HERE ###

    return s;

OutPut:-



### 1.2 Sigmoid gradient

**Exercise**: Implement the function **sigmoid\_grad()** to compute the gradient of the sigmoid function with respect to its input xx. The formula is:



You often code this function in two steps:

1. Set s to be the sigmoid of x. You might find your sigmoid(x) function useful.
2. Compute σ′(x) = s(1 − s)

  # GRADED FUNCTION: sigmoid\_derivative

import numpy as np; # this means you can access numpy functions by writing np.function() instead of numpy.function()

def sigmoid\_derivative(x):

    ### START CODE HERE ### (≈ 2 lines of code)

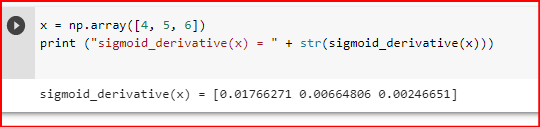
    s = 1 / (1 +  np.exp(-x));

    ds = s \* (1 - s);

    ### END CODE HERE ###

    return ds;

OutPut:-

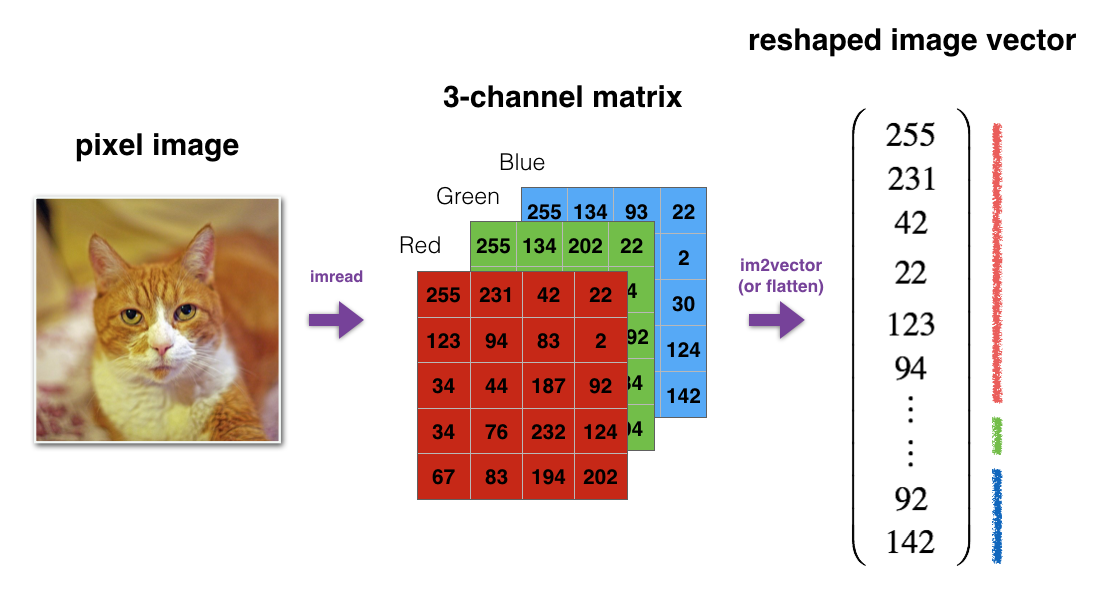


### **1.3 Reshaping arrays**

Two common numpy functions used in deep learning are **np.shape** and **np.reshape().**

* **X.shape** is used to get the shape (dimension) of a matrix/vector X.
* **X.reshape()** is used to reshape X into some other dimension.

For example, in computer science, an image is represented by a 3D array of shape (length, height, depth=3). However, when you read an image as the input of an algorithm you convert it to a vector of **shape (length∗height∗3,1)**. In other words, you “unroll”, or reshape, the 3D array into a 1D vector.



**Exercise**: Implement **image2vector()** that takes an input of shape(length, height, 3) and returns a vector of **shape(length \* height \* 3, 1)**. For example, if you would like to reshape an array v of shape (a, b, c) into a vector of shape (a\*b,c) you would do:

|  |  |
| --- | --- |
|  | # v.shape[0] = a ; v.shape[1] = b ; v.shape[2] = c;  v = v.reshape(v.shape[0] \* v.shape[1], v.shape[2]) |

**Please don’t hardcode the dimensions of image as a constant. Instead look up the quantities you need with image.shape[0], etc.**

  # GRADED FUNCTION: image2vector

def image2vector(image):

    """

    Argument:

    image -- a numpy array of shape (length, height, depth)

    Returns:

    v -- a vector of shape (length\*height\*depth, 1)

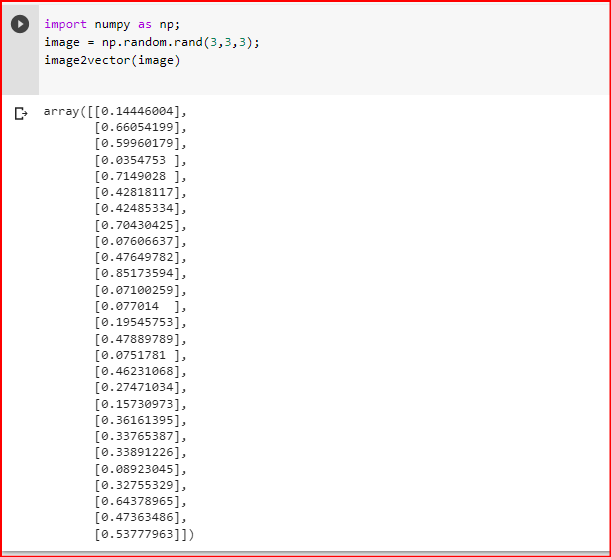
    """

    ### START CODE HERE ### (≈ 1 line of code)

    v = image.reshape(image.shape[0] \* image.shape[1] \* image.shape[2], 1);

    ### END CODE HERE ###

    return v;

**OutPut:-**

### **1.4 Normalizing rows**

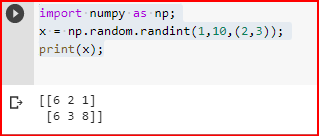
Another common technique we use in Machine Learning and Deep Learning is to normalize our data. It often leads to a better performance because gradient descent converges faster after normalization. Here, by normalization we mean changing x to x / ||x|| (dividing each row vector of x by its norm).

For example, if

import numpy as np;

x = np.random.randint(1,10,(2,3));

print(x);



then



and



**Code:**

import numpy as np;

x = np.random.randint(1,10,(2,3));

print(x);

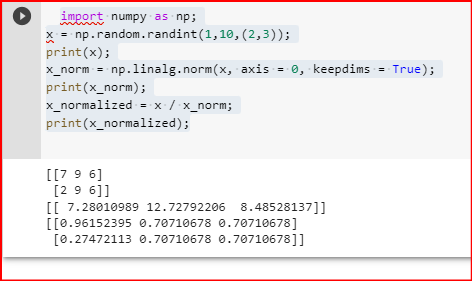
x\_norm = np.linalg.norm(x, axis = 0, keepdims = True);

print(x\_norm);

x\_normalized = x / x\_norm;

print(x\_normalized);

Output:-



Note that we can divide matrices of different sizes and it works fine: this is called **broadcasting**.

**Exercise**: Implement **normalizeRows()** to normalize the rows of a matrix. After applying this function to an input matrix x, each row of x should be a vector of unit length (meaning length 1).

  # GRADED FUNCTION: normalizeRows

def normalizeRows(x):

    """

    Implement a function that normalizes each row of the matrix x (to have unit length).

    Argument:

    x -- A numpy matrix of shape (n, m)

    Returns:

    x -- The normalized (by row) numpy matrix. You are allowed to modify x.

    """

    ### START CODE HERE ### (≈ 2 lines of code)

# Compute x\_norm as the norm 2 of x. Use np.linalg.norm(.., ord = 2, axis = .., keepdims = True)

    x\_norm = np.linalg.norm(x, axis=1, keepdims = True);

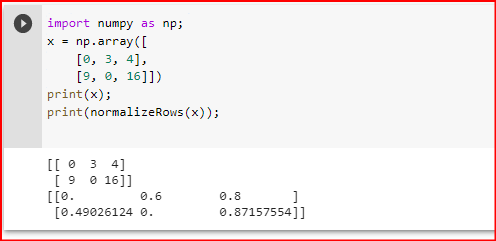
    # Divide x by its norm.

    x = x / x\_norm;

    ### END CODE HERE ###

    return x;

**Output:-**

****

**Note**:  
In **normalizeRows(),** you can try to print the shapes of **x\_norm** and x, and then rerun the assessment. You’ll find out that they have different shapes. This is normal given that **x\_norm** takes the norm of each row of **x**. So **x\_norm** has the same number of rows but only 1 column. So how did it work when you divided **x** by **x\_norm**? This is called **broadcasting**.

# **Part 2： Logistic Regression with a Neural Network mindset**

We will learn to:

* Build the general architecture of a learning algorithm, including:
* Initializing parameters
* Calculating the cost function and its gradient
* Using an optimization algorithm (gradient descent)
* Gather all three functions above into a main model function, in the right order.

## Packages Required:

First, let’s run the cell below to import all the packages that you will need during this assignment.

* [**numpy**](http://www.numpy.org/) is the fundamental package for scientific computing with Python.
* [**h5py**](http://www.h5py.org/) is a common package to interact with a dataset that is stored on an H5 file.
* [**matplotlib**](http://matplotlib.org/) is a famous library to plot graphs in Python.
* [**PIL**](http://www.pythonware.com/products/pil/) and [**scipy**](https://www.scipy.org/) are used here to test your model with your own picture at the end.

import numpy as np

import matplotlib.pyplot as plt

import h5py

import scipy

from PIL import Image

from scipy import ndimage

from lr\_utils import load\_dataset

% matplotlib inline

## Overview of the Problem Set:

Problem Statement: We will used a dataset (“data.h5”) containing:

* a training set of m\_train images labeled as cat (y=1) or non-cat (y=0)
* a test set of m\_test images labeled as cat or non-cat
* each image is of shape (num\_px, num\_px, 3) where 3 is for the 3 channels (RGB). Thus, each image is square (height = num\_px) and (width = num\_px).

We will build a simple **image-recognition algorithm** that can correctly classify pictures as cat or non-cat. Let’s get more familiar with the dataset. Load the data by running the following code.

# Loading the data (cat/non-cat)

train\_set\_x\_orig, train\_set\_y, test\_set\_x\_orig, test\_set\_y, classes = load\_dataset();

We added “\_orig” at the end of image datasets (train and test) because we are going to preprocess them. After pre-processing, we will end up with **train\_set\_x** and **test\_set\_x** (the labels **train\_set\_y** and **test\_set\_y** don’t need any pre-processing).

Each line of your **train\_set\_x\_orig** and **test\_set\_x\_orig** is an array representing an image. We can visualize an example by running the following code. If you want to change the index value and re-run to see other images.

**Code:**

# Example of a picture

index = 25

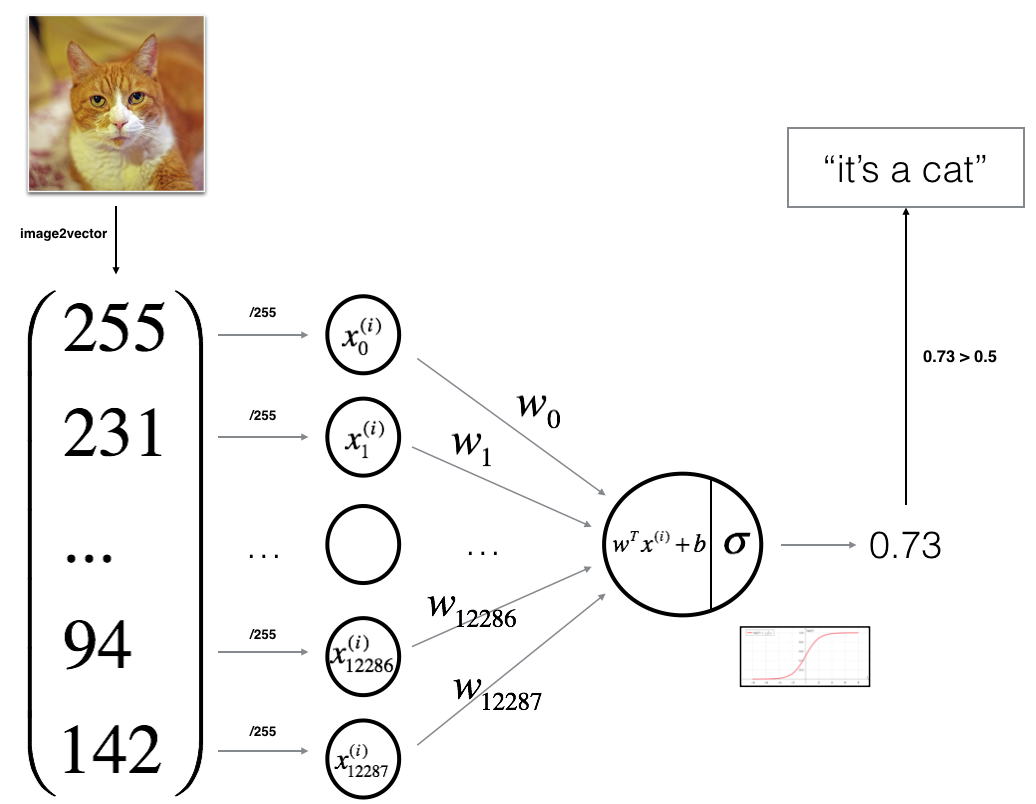
plt.imshow(train\_set\_x\_orig[index])

print("y = " + str(train\_set\_y[:, index]) + ", it's a '" + classes[np.squeeze(train\_set\_y[:, index])].decode("utf-8") +  "' picture.");

**Output:**

y = [1], it's a 'cat' picture.

**Logistic Regression with a Neural Network Mindset**



Logistic regression with a neural network mindset simply means that we will be doing a forward and backward propagation mode to code the algorithm as is usually the case with neural network algorithms. For logistic regression, the forward propagation is used to calculate the cost function and the output, y, while the backward propagation is used to calculate the gradient descent. This algorithm can be used to classify images as opposed to the ML form of logistic regression and that is what makes it stand out. The main steps for building the logistic regression neural network are:

1. Define the model structure (such as number of input features)
2. Initialize the model’s parameters
3. Loop:

* Calculate current loss (forward propagation)
* Calculate current gradient (backward propagation)
* Update parameters (gradient descent)

Let code, first we will begin by coding the sigmoid function by computing sigmoid (z) = 1/1+exp(-z), Where z = wx+b (Don’t worry about the formula if it does not make sense now, we will understand in the code below):

**Step 1: Implement the sigmoid function**

#Compute the sigmoid of z. z is a scalar or numpy array of any size

 s = 1/(1 + np.exp(-z))

 return s

Now, we will continue by initializing the model parameters. The model parameters are the weights (w) and bias (b) with x as the input feature.

**Step 2: Initialize the model parameters**

def initialize\_with\_zeros(m):

"""

This function creates a vector of zeros of shape (m, 1) for w and       initializes b to 0.

 Argument:

 dim — size of the w vector we want (or number of parameters in this case)

 Returns:

 w — initialized vector of shape (dim, 1)

 b — initialized scalar (corresponds to the bias)

 """

 w = np.zeros((m, 1))

 b = 0

 return w, b

**Step 3: Implement forward and backward propagation for learning the parameters**

The next step is to implement the function called propagate () that learn the parameters w, b, and y from x by computing the cost function (forward) and its gradient (backward).

def propagate(w, b, X, Y):

"""

Arguments:

 w — weights b — bias, a scalar

 X — input data

 Y — true “label” vectorReturn:

 cost — negative log-likelihood cost for logistic regression

 dw — gradient of the loss with respect to w, thus same shape as w

 db — gradient of the loss with respect to b, thus same shape as b

 """

 m = X.shape[1]

 # FORWARD PROPAGATION (FROM X TO COST)

 A = sigmoid(np.dot(w.T, X)+ b) # compute activation

 cost = -(1/m)\*(np.sum((Y\*np.log(A)) + (1-Y) \*np.log(1-A)))

 # BACKWARD PROPAGATION (TO FIND GRAD)

 dw = (1/m)\* np.dot(X, ((A-Y).T))

 db = (1/m) \* np.sum(A-Y) grads = {“dw”: dw,

 “db”: db}

 return grads, cost

**Step 4: Update parameters with gradient descent**

Having established the output and the cost function, we will need to optimize our algorithm by updating our parameters with gradient descent. This will reduce cost function and minimize loss.

def predict(w, b, X):

"""

 Predict whether the label is 0 or 1 using learned logistic regression parameters (w, b)

 Arguments:

 w — weights, a numpy array of size (num\_px \* num\_px \* 3, 1)

 b — bias, a scalar

 X — data of size (num\_px \* num\_px \* 3, number of examples)

 Returns:

 Y\_prediction — a numpy array (vector) containing all predictions (0/1) for the examples in X

 """

 m = X.shape[1]

 Y\_prediction = np.zeros((1,m))

 w = w.reshape(X.shape[0], 1)

 A = sigmoid(np.dot(w.T, X) + b)

 for i in range(A.shape[1]):

 # Convert probabilities A[0,i] to actual predictions p[0,i]

 Y\_prediction[0,i] = 1 if A[0, i] > 0.5 else 0

 pass

 return Y\_prediction

**Step 5: Putting it all together to form a model**

Now that we have our sigmoid function, cost function, and gradient descent, we will then combine everything into one single model and use this model to predict whether an image is a cat or non-cat.

def model(X\_train, Y\_train, X\_test, Y\_test, num\_iterations = 2000, learning\_rate = 0.5, print\_cost = False):

  """

  Builds the logistic regression model by calling the function you’ve implemented previously

  Arguments:

  X\_train — training set

  Y\_train — training labels

  X\_test — test set

  Y\_test — test labels

  num\_iterations — hyperparameter representing the number of iterations to optimize the parameters

  learning\_rate — hyperparameter representing the learning rate used in the update rule of optimize()

  print\_cost — Set to true to print the cost every 100 iterations

  Returns:

  d — dictionary containing information about the model.

  """

  w, b = initialize\_with\_zeros(X\_train.shape[0])# Gradient descent

  parameters, grads, costs = optimize(w, b, X\_train, Y\_train, num\_iterations, learning\_rate, print\_cost)

  # Retrieve parameters w and b from dictionary “parameters”

  w = parameters["w"]

  b = parameters["b"]

  # Predict test/train set examples

  Y\_prediction\_test = predict(w, b, X\_test)

  Y\_prediction\_train = predict(w, b, X\_train)# Print train/test Errors

  print("train accuracy: {} %".format(100 — np.mean(np.abs(Y\_prediction\_train — Y\_train)) \* 100))

  print("test accuracy: {} %".format(100 — np.mean(np.abs(Y\_prediction\_test — Y\_test)) \* 100))

  d = {"costs": costs,"Y\_prediction\_test": Y\_prediction\_test,  "Y\_prediction\_train" : Y\_prediction\_train,

  "w": w,  "b" : b,"learning\_rate" : learning\_rate,"num\_iterations": num\_iterations}

  return d

Output:-



Cost after iteration 0: 0.693147

Cost after iteration 100: 0.584508

Cost after iteration 200: 0.466949

Cost after iteration 300: 0.376007

Cost after iteration 400: 0.331463

Cost after iteration 500: 0.303273

Cost after iteration 600: 0.279880

Cost after iteration 700: 0.260042

Cost after iteration 800: 0.242941

Cost after iteration 900: 0.228004

Cost after iteration 1000: 0.214820

Cost after iteration 1100: 0.203078

Cost after iteration 1200: 0.192544

Cost after iteration 1300: 0.183033

Cost after iteration 1400: 0.174399

Cost after iteration 1500: 0.166521

Cost after iteration 1600: 0.159305

Cost after iteration 1700: 0.152667

Cost after iteration 1800: 0.146542

Cost after iteration 1900: 0.140872 train accuracy: 99.04306220095694 % test accuracy: 70.0 %

Conclusion: Training accuracy is close to 100%. This is a good sanity check: your model is working and has high enough capacity to fit the training data. Test error is 68%. It is actually not bad for this simple model, given the small dataset we used and that logistic regression is a linear classifier.