BIRLA INSTITUTE OF TECHNOLOGY AND SCIENECE
NLP LAB -I
[LOGISTIC REGRESSION & RNN]

DATE: 09/04/2023 TIME: 02 Hours

This lab-sheet is made for the purpose of refreshing or learning the concept of derivation/partial derivation and use these concepts to apply at Regression model, and demonstrate simple approach to understand how exactly the Regression learns to predict the actual output. The work contains two steps as follows:

1. Concepts of Derivation
2. Walking through step by step the Regression w.r.t its parameters update with Gradient Update

1. Derivation:

To understand how derivatives work, we have provided one example of addition of a function that has two variables (x, y). Below is solved example of addition:

$$f(x,y) = x + y \qquad \rightarrow \qquad \frac{\partial f}{\partial x} = 1 \qquad \frac{\partial f}{\partial y} = 1$$

When we are dealing with two variables, then we take derivatives in flow like we consider one variable as constant and take derivative w.r.t to other variable (Known as partial derivative)

Computational Graph and Chain Rule:
what if we do have equation of more than two vaiables?

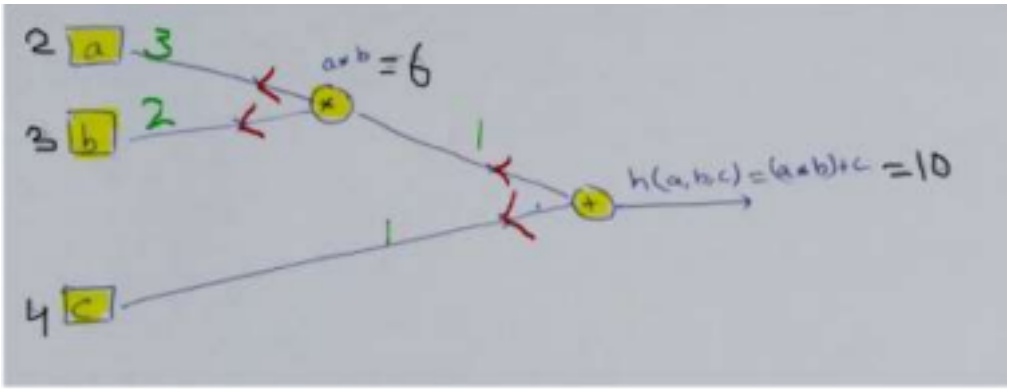
$$h(a,b,c) = (a*b) + c$$

We use computation graph for equations. The structure of graph is simple, we place variables at input positions and operators in between the graph. And the function output is achieved at the end of graph. mathematical operations are performed in sequence steps; like *a* and *b* are first multiplied and their results are added to *c*. We may give any name to intermediate state, that might be helpful in derivation steps. For determining partial derivative of *a* and *b* we will be using chain rule. And it states that multiply all previous derivates to new. Thus, the partial derivatives of *a* and *b* will be computed as follows:

$h(a,b,c)=(a*b)+c$, $(a*b)=tmp$, $h(a,b,c)=tmp+c$
$\partial h/\partial c=\partial h/\partial c(tmp+c)$, $\partial h/\partial c=1$, $\partial h/\partial tmp=\partial h/\partial tmp(tmp+c)$, $\partial h/\partial tmp=1$
$\partial h/\partial a=(\partial h/\partial tmp)*(\partial tmp/\partial a)$, $\partial h/\partial b=(\partial h/\partial tmp)*(\partial tmp/\partial b)$

$\partial h/\partial a=\partial h/\partial tmp*\partial tmp/\partial a$, $\partial tmp/\partial a=\partial tmp/\partial a(a*b)$, $\partial tmp/\partial a=b$,
$\partial h/\partial a=3*1=3:[\partial a=3;\partial tmp=1]$
$\partial h/\partial b=\partial h/\partial tmp*\partial tmp/\partial b$, $\partial tmp/\partial b=\partial tmp/\partial a(a*b)$, $\partial tmp/\partial b=a$,
$\partial h/\partial b=2*1=2:[\partial a=2;\partial tmp=1]$

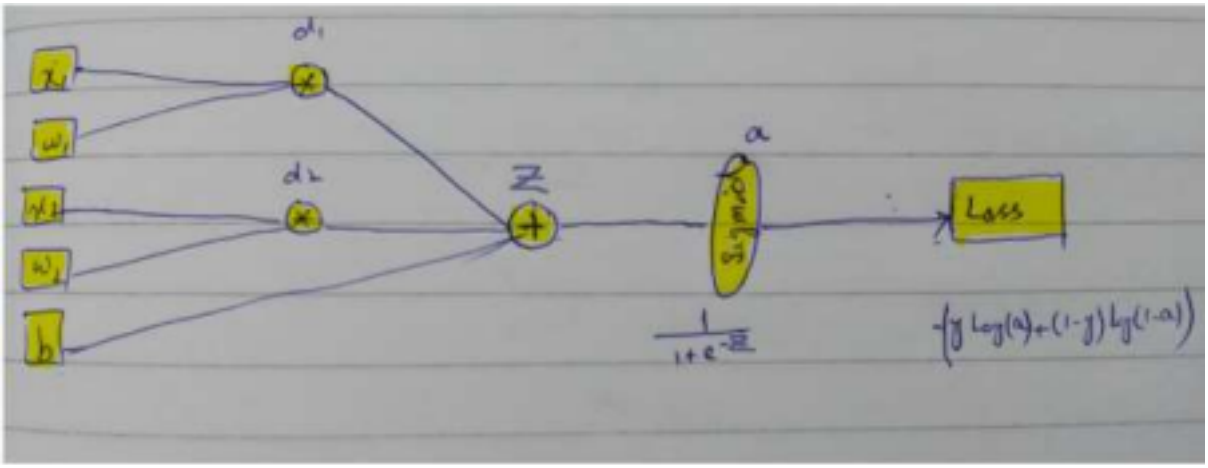
The diagram of function output and derivatives below can validate the above concepts.



2. Regression

In this section, the forward pass is shown in Computational Graph to describe how to compute loss and using that loss calculate gradient and update its weight. For the sake of simplicity we consider only one training example with two feature sets i.e. two inputs *x*1, *x*2. These input values are then used in XOR function to generate output (target 'y'). The generated output is used to calculate the loss by comparing this value to the activation generated through the process.

In the following computational graph, the two feature set i.e. *x*1 and *x*2 are multiplied by weights *w*1 and *w*2 and their product is shown in *d*1 and *d*2 states. The bias *b* is added to both *d*1 and *d*2 state yielding the *z* state. To get the activation of *z*; the sigmoid function is applied and finally the loss is computed with actual value of XOR function (using same values of *x*1 and *x*2) *y* and the activation value *a*. We may say this whole process as **Forward Pass**.



let's implement the above graph with using random generated weights and bias and see how activation is different from the actual value.

First set the python3 environment and import libraries for onward use

```
import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g.
pd.read_csv) import matplotlib.pyplot as plt # ploting graph
```

Define sigmoid function

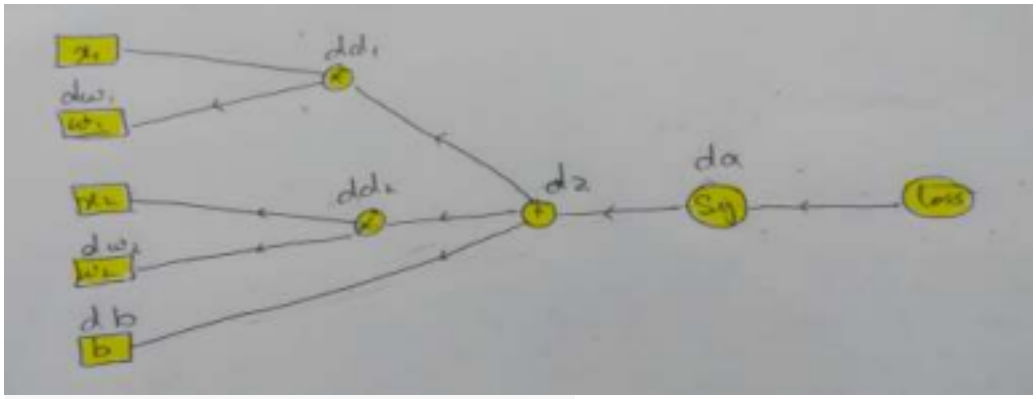
```
def sigmoid(z):
 # Takes input as z and return sogmoid of value
 s = //Write formula
 return s
```

To generate data with two feature set and get label of XOR function

```
x1, x2 = 1, 1
y = int(np.logical_xor(x1,x2))
```

```
print('actual value(y): ', y)
#define paramters i.e. weights and bias to random
w1, w2, b = 0.1, 0.5, 0.005

# print('Parameters Before update')
# print('w1: ', w1, 'w2: ', w2, 'b: ', b)
Yield a Z state

z = w1*x1 + w2*x2 + b

activation of values
a = sigmoid(z)
print('activation value: ', a)

compute the loss of the function(since we have training exmple equal to 1, so cost==loss )
cost = -1 * (y * np.log(a) + (1 - y) * (np.log(1 - a))) # compute cost
print('loss of function: ', cost)


Store cost, activation, weights and bias
my_dic = {}
my_dic['w1'] = w1
my_dic['w2'] = w2
my_dic['b'] = b
my_dic['activation'] = a
my_dic['cost'] = cost

You will get ouput as follows:
actual value(y): 0
activation value: 0.6467993925579857
loss of function: 1.0407190904293084
```

We can see that the actual value is *0* is far away from the activation value *0.65* and cost of this function is much higher. Now let's see how gradient update the weight and affects the loss (reduce) to get closer value of activation to the actual value. Before implementing code, let's do some maths stuff to see how gradients of weights & bias according to loss can be calculated. In this step, we will go from right to left opposite to Forward pass and we may say it **Backward Pass.**




```
# BACKWARD PROPAGATION (TO FIND GRAD)
dw1 = x1*(a-y)
dw2 = x2*(a-y)
db = a - y
```

The gradients of weights and bias are used to update the initial weights and bias. In this step we multiply gradient with a learning rate (this controls the gradient update contribution in the parameters weights and bias).

```
update parameter w1, w2 and b by the equation
lr =0.07 # learning rate

w1 = w1 - (lr*dw1)
w2 = w2 - (lr*dw2)
b = b - (lr*db)

z = w1*x1 + w2*x2 + b
activation of values
a = sigmoid(z)
print('activation value before update: ', my_dic['activation'])
print('activation value after update: ', a)
print('')

compute the loss of the function(since we have training exmple equal to 1, so cost==loss )
cost = -1 * (y * np.log(a) + (1 - y) * (np.log(1 - a))) # compute cost
print('loss of function before update: ', my_dic['cost']) print('loss of function after update: ', cost)
activation value before update: 0.6467993925579857
activation value after update: 0.6151877913801179

loss of function before update: 1.0407190904293084
loss of function after update: 0.95499983
```

From the above result we can easily recognize the change in activation and loss in one step **Backward Pass**. Before performing update in weights, we are getting higher loss while in one step update we yield in smaller loss than before. Let's do update multiple time to see how the weight and bias update effect the loss.

Update Parameters w and b multiple times:

We will update parameters weights and bias multiple times by looping over the process we have followed above and see the results, following are the steps:

1. Define the function that will compute activation and loss
2. Define the function to update the parameters
3. Define the function plot the results
4. Iterate over the above function multiple times and plot the result

```
def get_activation_loss(x1, x2, w1, w2, b):
        # this function compute activations, cost and z
        # x : input features
        # w : weight
        # b : bias

 z = w1*x1 + w2*x2 + b

#activation of values
```

```
a = sigmoid(z)

compute the loss of the function(since we have training exmple equal to 1, so cost==loss )
 cost =
-1 * (y * np.log(a) + (1 - y) * (np.log(1 - a))) # compute cost
 return(a,cost, z)

def update_paramters(x1, x2, w1, w2, b, a, y, lr):

This function computes gradient of parameters and then update
them  # returns updated parameters weights and bias
 # x: input features
 # w: weights
 # b: bias
 # a: activation
 # y: actual label
 # lr: learning rate

BACKWARD PROPAGATION (TO FIND GRAD)
dw1 = x1*(a-y)
 dw2 = x2*(a-y)
  db = a - y

update parameter w1, w2 and b by the equation
w1 = w1 - (lr*dw1)
 w2 = w2 - (lr*dw2)
 b = b - (lr*db)

  return(w1, w2, b)

def plt_res(lst, ylab, lr):
```

Plot the results:

This will plot the list of values at y axis while x axis will contain number of iteration

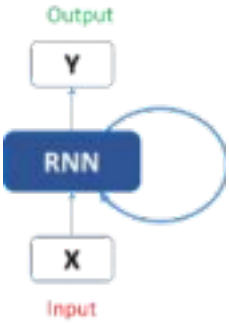
```
 #lst: list of action/cost
 #ylab: y-axis label
 #lr: learning rate
plt.plot(lst)
 plt.ylabel(ylab)
 plt.xlabel('iterations')
 plt.title("Learning rate =" + str(lr))
 plt.show()

define parameters i.e. weights and bias to random to new random values w1, w2, b = 0.04, 0.75, 0.0015
lst_cost = []
lst_activation = []

In code below, update parameters about 1000 or 2000 times and see the differen ce
num_iter = 1000
lr = 0.007
```

```
#generate data with two feature set and get label
x1, x2 = 1, 0
y = int(np.logical_xor(0,1))

print ('x1: ', x1, '; x2: ',x2)
print('xor value(y): ', y)

for i in range(num_iter):
 a,cost,z = get_activation_loss(x1, x2, w1, w2, b)

# print('cost at iteration', i,': ', cost)
# print('activation at iteration', i,': ',a)
w1, w2, b = update_paramters(x1, x2, w1, w2, b, a, y, lr)
lst_cost.append(cost)
 lst_activation.append(a)

plt_res(lst_cost, 'loss', lr)
plt_res(lst_activation,'activation', lr)
x1: 1 ; x2: 0
xor value(y): 1
```

RNN

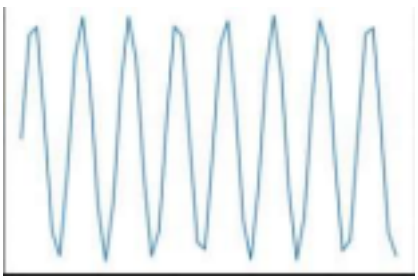
There is another concept– Recurrent Neural Networks (RNN)!

A typical RNN looks like this:



Sequence Prediction using RNN

1. We will do a sequence prediction problem using RNN. One of the simplest tasks is sine wave prediction. This is what a sine wave looks like:



2. We will first devise a recurrent neural network from scratch to solve this problem. 3. We will formulate our problem like this – given a sequence of 50 numbers belonging to a sine wave, predict the 51st number in the series.

Data Preparation

#Prepare the data
What does our model expect the data to be like?
It would accept a single sequence of length 50 as input. So the shape of the input data will be:

```
(number_of_records x length_of_sequence x types_of_sequences)
```

Here, types_of_sequences is 1, because we have only one type of sequence – the sine wave.On the other hand, the output would have only one value for each record. This will of course be the 51st value in the input sequence. So it's shape would be:

```
(number_of_records x types_of_sequences)
```

Import libraries

```
%pylab inline
import numpy as np
import matplotlib.pyplot as plt
import math
```

To create a sine wave like data, we will use the sine function from Python's *math* library:

```
sin_wave = np.array([math.sin(x) for x in np.arange(200)])
#Visualizing the sine wave we've just generated:

plt.plot(sin_wave[:50])#plot the sin wave
```

We will create the data in the below code block:

```
X = []
Y = []
seq_len = 50
num_records = len(sin_wave) - seq_len

for i in range(num_records - 50):
 X.append(sin_wave[i:i+seq_len])
 Y.append(sin_wave[i+seq_len])

X = np.array(X)
X = np.expand_dims(X, axis=2)

Y = np.array(Y)
Y = np.expand_dims(Y, axis=1)
```

Print the shape of the data:

```
X.shape, Y.shape
```

((100, 50, 1), (100, 1))

```
#Note that we looped for (num_records - 50) because we want to set
```

```
aside 50 records as our validation data. We can create this
validation data now:
X_val = []
Y_val = []

#num_records = 100
#seq_len = 3
for i in range(num_records - 50, num_records):
 X_val.append(sin_wave[i:i+seq_len])
 Y_val.append(sin_wave[i+seq_len])

X_val = np.array(X_val)
X_val = np.expand_dims(X_val, axis=2)

Y_val = np.array(Y_val)
Y_val = np.expand_dims(Y_val, axis=1)
```

Create the Architecture for our RNN model

Our next task is defining all the necessary variables and functions we'll use in the RNN model. Our model will take in the input sequence, process it through a hidden layer of 100 units, and produce a single valued output:

```
learning_rate = 0.0001
nepoch = 2
T = 50 # length of sequence
hidden_dim = 100
output_dim = 1

bptt_truncate = 5
min_clip_value = -10
max_clip_value = 10


#We will then define the weights of the network:
U = np.random.uniform(0, 1, (hidden_dim, T))
W = np.random.uniform(0, 1, (hidden_dim, hidden_dim))
V = np.random.uniform(0, 1, (output_dim, hidden_dim))
```

Here,

U is the weight matrix for weights between input and hidden layers

V is the weight matrix for weights between hidden and output layers

W is the weight matrix for shared weights in the RNN layer (hidden layer)

Finally, we will define the activation function, sigmoid, to be used in the hidden layer:

Train the Model

Now that we have defined our model, we can finally move on with training it on our sequence data. We can subdivide the training process into smaller steps, namely:

Step 2.1 : Check the loss on training data
Step 2.1.1 : Forward Pass
Step 2.1.2 : Calculate Error
Step 2.2 : Check the loss on validation data
Step 2.2.1 : Forward Pass
Step 2.2.2 : Calculate Error
Step 2.3 : Start actual training
Step 2.3.1 : Forward Pass
Step 2.3.2 : Backpropagate Error
Step 2.3.3 : Update weights

We need to repeat these steps until convergence.

Step 2.1: Check the loss on training data

We will do a forward pass through our RNN model and calculate the squared error for the predictions for all records in order to get the loss value.

```
for epoch in range(nepoch):
 # check loss on train
 loss = 0.0

 # do a forward pass to get prediction
 for i in range(Y.shape[0]):
 x, y = X[i], Y[i] # get input, output values of each record
 prev_s = np.zeros((hidden_dim, 1)) # here, prev s is the value of the
previous activation of hidden layer; which is ini tialized as all
zeroes
 for t in range(T):
 new_input = np.zeros(x.shape) # we then do a forwa rd pass for every
timestep in the sequence
 new_input[t] = x[t] # for this, we defin e a single input for that
timestep
 mulu = np.dot(U, new_input)
  mulw = np.dot(W, prev_s)
 add = mulw + mulu
 s = sigmoid(add)
 mulv = np.dot(V, s)
 prev_s = s

 # calculate error
 loss_per_record = (y - mulv)**2 / 2
 loss += loss_per_record
 loss = loss / float(y.shape[0])
```

Step 2.2: Check the loss on validation data

We will do the same thing for calculating the loss on validation data (in the same loop):

```
# check loss on val
val_loss = 0.0
for i in range(Y_val.shape[0]):
 x, y = X_val[i], Y_val[i]
```

```
 prev_s = np.zeros((hidden_dim, 1))
 for t in range(T):
 new_input = np.zeros(x.shape)
 new_input[t] = x[t]
 mulu = np.dot(U, new_input)
 mulw = np.dot(W, prev_s)
 add = mulw + mulu
 s = sigmoid(add)
 mulv = np.dot(V, s)
 prev_s = s
 loss_per_record = (y - mulv)**2 / 2
 val_loss += loss_per_record
val_loss = val_loss / float(y.shape[0])

print('Epoch: ', epoch + 1, ', Loss: ', loss, ', Val Loss: ', val_loss)
```

Step 2.3: Start actual training

We will now start with the actual training of the network. In this, we will first do a forward pass to calculate the errors and a backward pass to calculate the gradients and pdate them.

```
# train model
for i in range(Y.shape[0]):
 x, y = X[i], Y[i]

 layers = []
 prev_s = np.zeros((hidden_dim, 1))
 dU = np.zeros(U.shape)
 dV = np.zeros(V.shape)
 dW = np.zeros(W.shape)

 dU_t = np.zeros(U.shape)
 dV_t = np.zeros(V.shape)
 dW_t = np.zeros(W.shape)

 dU_i = np.zeros(U.shape)
 dW_i = np.zeros(W.shape)
```

Step 2.3.1: Forward Pass

In the forward pass:

We first multiply the input with the weights between input and hidden layers

Add this with the multiplication of weights in the RNN layer. This is because we want to capture the knowledge of the previous timestep

Pass it through a sigmoid activation function
Multiply this with the weights between hidden and output layers
At the output layer, we have a linear activation of the values so we do not explicitly pass the value through an activation layer

Save the state at the current layer and also the state at the previous time step in a dictionary

Here is the code for doing a forward pass (note that it is in continuation of the above loop):

```
# forward pass
 for t in range(T):
 new_input = np.zeros(x.shape)
 new_input[t] = x[t]
 mulu = np.dot(U, new_input)
 mulw = np.dot(W, prev_s)
 add = mulw + mulu
 s = sigmoid(add)
 mulv = np.dot(V, s)
 layers.append({'s':s, 'prev_s':prev_s}) prev_s = s
```

Step 2.3.2 : Backpropagate Error

After the forward propagation step, we calculate the gradients at each layer, and backpropagate the errors. We will use truncated back propagation through time (TBPTT), instead of vanilla backprop. It may sound complex but it's actually pretty straight forward.

The core difference in BPTT versus backprop is that the backpropagation step is done for all the time steps in the RNN layer. So if our sequence length is 50, we will backpropagate for all the timesteps previous to the current timestep.

If you have guessed correctly, BPTT seems very computationally expensive. So instead of backpropagating through all previous timestep , we backpropagate till x timesteps to save computational power. Consider this ideologically similar to stochastic gradient descent, where we include a batch of data points instead of all the data points.

Here is the code for backpropagating the errors:

```
# derivative of pred
dmulv = (mulv - y)

# backward pass
for t in range(T):
 dV_t = np.dot(dmulv, np.transpose(layers[t]['s'])) dsv =
np.dot(np.transpose(V), dmulv)

 ds = dsv
 dadd = add * (1 - add) * ds

 dmulw = dadd * np.ones_like(mulw)

 dprev_s = np.dot(np.transpose(W), dmulw)


 for i in range(t-1, max(-1, t-bptt_truncate-1), -1): ds = dsv
+ dprev_s
 dadd = add * (1 - add) * ds
```

```
dmulw = dadd * np.ones_like(mulw) dmulu = dadd *
np.ones_like(mulu)

dW_i = np.dot(W, layers[t]['prev_s']) dprev_s =
np.dot(np.transpose(W), dmulw)
new_input = np.zeros(x.shape) new_input[t] = x[t]
dU_i = np.dot(U, new_input) dx = np.dot(np.transpose(U), dmulu)

dU_t += dU_i
dW_t += dW_i

dV += dV_t
dU += dU_t
dW += dW_t
```

Step 2.3.3 : Update weights

Lastly, we update the weights with the gradients of weights calculated. One thing we have to keep in mind that the gradients tend to explode if you don't keep them in check. This is a fundamental issue in training neural networks, called the exploding gradient problem. So we have to clamp them in a range so that they don't explode. We can do it like this

```
if dU.max() > max_clip_value:
dU[dU > max_clip_value] = max_clip_value if dV.max() >
max_clip_value:
dV[dV > max_clip_value] = max_clip_value if dW.max() >
max_clip_value:
dW[dW > max_clip_value] = max_clip_value

if dU.min() < min_clip_value:
dU[dU < min_clip_value] = min_clip_value if dV.min() <
min_clip_value:
dV[dV < min_clip_value] = min_clip_value if dW.min() <
min_clip_value:
dW[dW < min_clip_value] = min_clip_value
# update
U -= learning_rate * dU
V -= learning_rate * dV
W -= learning_rate * dW
```

Step 3: Get predictions

We will do a forward pass through the trained weights to get our predictions:

```
preds = []
for i in range(Y.shape[0]):
x, y = X[i], Y[i]
prev_s = np.zeros((hidden_dim, 1))
# Forward pass
for t in range(T):
mulu = np.dot(U, x)
mulw = np.dot(W, prev_s)
add = mulw + mulu
```

```
 s = sigmoid(add)
 mulv = np.dot(V, s)
 prev_s = s

 preds.append(mulv)

preds = np.array(preds)
#Plotting these predictions alongside the actual values:

plt.plot(preds[:, 0, 0], 'g')
plt.plot(Y[:, 0], 'r')
plt.show()
```

This was on the training data. How do we know if our model didn't overfit? This is where the validation set, which we created earlier, comes into play:

```
preds = []
for i in range(Y_val.shape[0]):
 x, y = X_val[i], Y_val[i]
 prev_s = np.zeros((hidden_dim, 1))
 # For each time step...
 for t in range(T):
 mulu = np.dot(U, x)
 mulw = np.dot(W, prev_s)
 add = mulw + mulu
 s = sigmoid(add)
 mulv = np.dot(V, s)
 prev_s = s

 preds.append(mulv)

preds = np.array(preds)

plt.plot(preds[:, 0, 0], 'g')
plt.plot(Y_val[:, 0], 'r')
plt.show()
```

Exercise:-

Expand the before mentioned logistic regression with XOR function and generalized it to *m* number of rows and *n_x* feature set (columns) and implement the following:

1. A function which generates data set of desired length and width (row, columns)
2. The labels are created using *XOR*; the data generated in the above step then passed to this step to get the value of *XOR* operation
3. Implement Regression model
4. Discuss the performance of Regression on different datasets and different parameters.