Forecasting using RNN

Alok Jadhav

In [1]:
```
#What are we working with?
import sys
sys.version
```

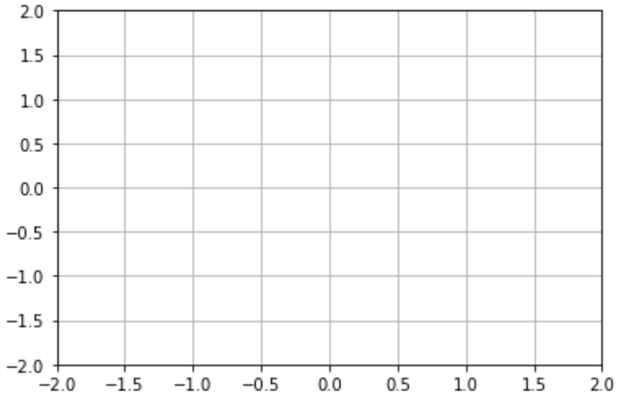
Out[1]:
```
'3.6.3 |Anaconda custom (64-bit)| (default, Oct 13 2017, 12:02:49) \n[GCC
7.2.0]'
```

In [2]:
```
#Import Libraries
import tensorflow as tf
import pandas as pd
import numpy as np
import os
import matplotlib
import matplotlib.pyplot as plt
import random
%matplotlib inline
import tensorflow as tf
import shutil
import tensorflow.contrib.learn as tflearn
import tensorflow.contrib.layers as tflayers
from tensorflow.contrib.learn.python.learn import learn_runner
import tensorflow.contrib.metrics as metrics
import tensorflow.contrib.rnn as rnn

from numpy import sin, cos
import numpy as np
import matplotlib.pyplot as plt
import scipy.integrate as integrate
import matplotlib.animation as animation
import random
```

In [3]:
```
#TF Version
tf.__version__
```

Out[3]:
```
'1.3.0'
```

In [4]:
```
# for each experiment value of l1,l2,m1,m2 and th1,th2,w1,w2 are same so
explicitely add these features after training.

G = 9.8  # acceleration due to gravity, in m/s^2
L1 = 1.0  # length of pendulum 1 in m
L2 = 1.0  # length of pendulum 2 in m
M1 = 1.0  # mass of pendulum 1 in kg
M2 = 1.0  # mass of pendulum 2 in kg


def derivs(state, t):

    dydx = np.zeros_like(state)
    dydx[0] = state[1]

    del_ = state[2] - state[0]
    den1 = (M1 + M2)*L1 - M2*L1*cos(del_)*cos(del_)
    dydx[1] = (M2*L1*state[1]*state[1]*sin(del_)*cos(del_) +
               M2*G*sin(state[2])*cos(del_) +
               M2*L2*state[3]*state[3]*sin(del_) -
               (M1 + M2)*G*sin(state[0]))/den1

    dydx[2] = state[3]

    den2 = (L2/L1)*den1
    dydx[3] = (-M2*L2*state[3]*state[3]*sin(del_)*cos(del_) +
               (M1 + M2)*G*sin(state[0])*cos(del_) -
               (M1 + M2)*L1*state[1]*state[1]*sin(del_) -
               (M1 + M2)*G*sin(state[2]))/den2

    return dydx

# create a time array from 0..100 sampled at 0.05 second steps
dt = 0.1
t = np.arange(0.0, 100, dt)

# th1 and th2 are the initial angles (degrees)
# w10 and w20 are the initial angular velocities (degrees per second)

th1 = 120.0
w1 = 0.0
th2 = 0.0
w2 = 0.0

# initial state
state = np.radians([th1, w1, th2, w2])

# integrate your ODE using scipy.integrate.
y = integrate.odeint(derivs, state, t)

x1 = L1*sin(y[:, 0])
y1 = -L1*cos(y[:, 0])

#print("x1 : ",x1)
#print("y1 : ",y1)

x2 = L2*sin(y[:, 2]) + x1
y2 = -L2*cos(y[:, 2]) + y1

#print("x2 : ",x2)
#print("y2 : ",y2)

fig = plt.figure()
ax = fig.add_subplot(111, autoscale_on=False, xlim=(-2, 2), ylim=(-2, 2)
)
ax.grid()

line, = ax.plot([], [], 'o-', lw=2)
```

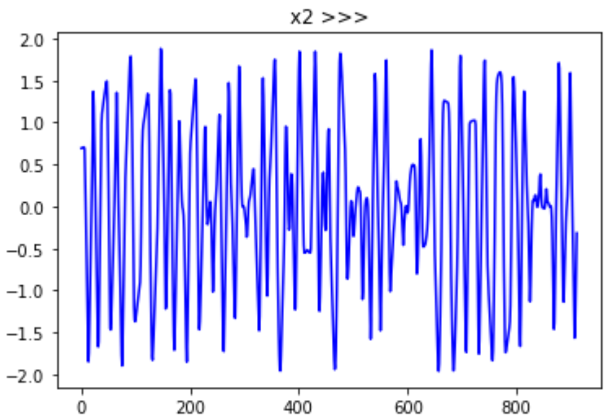


Generate some data

In [5]:
```
random.seed(111)
#ts.head(10)

f = open("data/x2.txt" , "r")
t = open("data/time_slots.txt" , "r")
array1 = []
array2 = []

for line in f.read().split('\n') :
  array1.append(line )
for line in t.read().split('\n') :
  array2.append(line )
f.close()
t.close()
array1.pop()
array2.pop()
data = []

for i in range(len(array1)):
    data.append([array1[i]])
test = np.array(data)
mylist = test.astype(np.float)

print("length of test data : ",len(mylist))
print(mylist.shape)
mylist.reshape(1,-1)
#print(mylist)

ts = np.delete(mylist,[i for i in range(912,1001)],0)

print("length of test data : ",len(ts))
print(ts.shape)
ts.reshape(1,-1)
#print(ts)

plt.plot(ts,c='b')
plt.title('x2 >>>')
plt.show()
```

```
length of test data :  1000
(1000, 1)
length of test data :  912
(912, 1)
```

```
/home/omkarthawakar/anaconda3/lib/python3.6/site-packages/ipykernel_launc
her.py:29: DeprecationWarning: in the future out of bounds indices will r
aise an error instead of being ignored by `numpy.delete`.
```



Convert data into array that can be broken up into training "batches" that we will feed into our RNN model. Note the shape of the arrays.

In [6]:
```
TS = np.array(ts)
num_periods = 100
f_horizon = 1  #forecast horizon, one period into the future

x_data = TS[:(len(TS)-(len(TS) % num_periods))]
#print(len(x_data))
#print("x_data : ",x_data)
x_batches = x_data.reshape(-1, 100, 1)

#print (len(x_batches))
print ("x_batches shape : ",x_batches.shape)
#print (x_batches[0:1])

y_data = TS[1:(len(TS)-(len(TS) % num_periods))+f_horizon]
#print(y_data.shape)
y_batches = y_data.reshape(-1, 100, 1)


#print ("y_batches : ",y_batches[0:1])
print ("y_batches shape : ",y_batches.shape)
```

```
x_batches shape :  (9, 100, 1)
y_batches shape :  (9, 100, 1)
```

Pull out our test data

In [7]:
```
def test_data(series,forecast,num_periods):
    test_x_setup = TS[-(num_periods + forecast):]
    testX = test_x_setup[:num_periods].reshape(-1, 100, 1)
    testY = TS[-(num_periods):].reshape(-1, 100, 1)
    return testX,testY

X_test, Y_test = test_data(TS,f_horizon,num_periods )
print (X_test.shape)
print(len(X_test))
#print (X_test)

print (Y_test.shape)
print(len(Y_test))
#print (Y_test)
```

```
(1, 100, 1)
1
(1, 100, 1)
1
```

In [8]:
```
tf.reset_default_graph()   #We didn't have any previous graph objects ru
nning, but this would reset the graphs

num_periods = 100      #number of periods per vector we are using to pre
dict one period ahead
inputs = 1           #number of vectors submitted
hidden = 100          #number of neurons we will recursively work throug
h, can be changed to improve accuracy
output = 1            #number of output vectors

X = tf.placeholder(tf.float32, [None, num_periods, inputs])   #create va
riable objects
y = tf.placeholder(tf.float32, [None, num_periods, output])


basic_cell = tf.contrib.rnn.BasicRNNCell(num_units=hidden, activation=tf
.nn.relu)   #create our RNN object
multi_rnn_cell = tf.contrib.rnn.MultiRNNCell([basic_cell] * 2)
rnn_output, states = tf.nn.dynamic_rnn(basic_cell, X, dtype=tf.float32)
#choose dynamic over static

learning_rate = 0.001   #small learning rate so we don't overshoot the m
inimum

stacked_rnn_output = tf.reshape(rnn_output, [-1, hidden])          #cha
nge the form into a tensor
stacked_outputs = tf.layers.dense(stacked_rnn_output, output)        #sp
ecify the type of layer (dense)
outputs = tf.reshape(stacked_outputs, [-1, num_periods, output])
#shape of results

loss = tf.reduce_sum(tf.square(outputs - y))    #define the cost functio
n which evaluates the quality of our model
optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate)
#gradient descent method
training_op = optimizer.minimize(loss)          #train the result of the
application of the cost_function

init = tf.global_variables_initializer()           #initialize all the v
ariables
```

In [9]:
```
with tf.Session() as sess:
        writer = tf.summary.FileWriter("output", sess.graph)
        print(sess.run(init))
        writer.close()
```

```
None
```

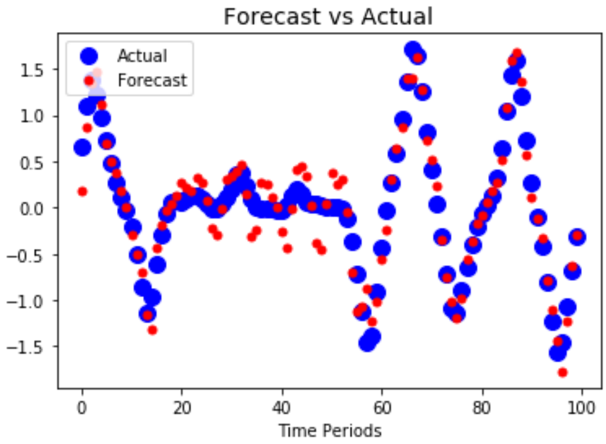
In [10]:
```
print(states)
```

```
Tensor("rnn/while/Exit_2:0", shape=(?, 100), dtype=float32)
```

In [11]:
```
epochs = 2000     #number of iterations or training cycles, includes bot
h the FeedFoward and Backpropogation
errors = []
iterations = []
with tf.Session() as sess:
    init.run()
    for ep in range(epochs):
        sess.run(training_op, feed_dict={X: x_batches, y: y_batches})
        errors.append(loss.eval(feed_dict={X: x_batches, y: y_batches}))
        iterations.append(ep)
        if ep % 100 == 0:
            mse = loss.eval(feed_dict={X: x_batches, y: y_batches})
            print(ep, "\tMSE:", mse)
    y_pred = sess.run(outputs, feed_dict={X: X_test})
    print(y_pred)
```

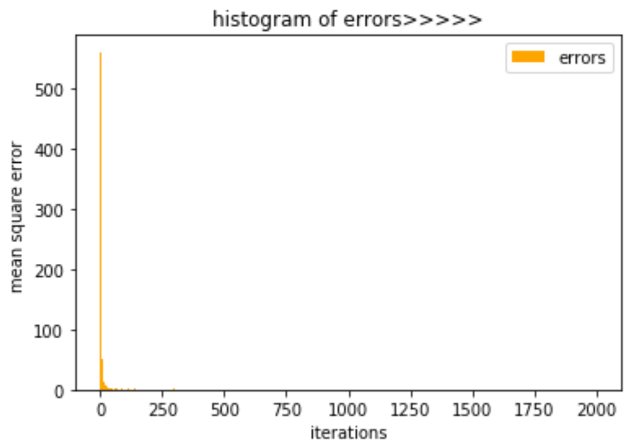
```
0       MSE: 747.177
100     MSE: 27.5026
200     MSE: 15.717
300     MSE: 10.3511
400     MSE: 7.62401
500     MSE: 5.55129
600     MSE: 5.53953
700     MSE: 5.43905
800     MSE: 3.67262
900     MSE: 3.69929
1000    MSE: 3.04312
1100    MSE: 2.62233
1200    MSE: 2.15797
1300    MSE: 4.10035
1400    MSE: 1.8404
1500    MSE: 2.06344
1600    MSE: 1.64905
1700    MSE: 1.72112
1800    MSE: 1.59529
1900    MSE: 1.38258
[[[ 0.17329071]
  [ 0.86608672]
  [ 1.33736837]
  [ 1.47347653]
  [ 1.10909045]
  [ 0.69186318]
  [ 0.50513339]
  [ 0.38218972]
  [ 0.18720019]
  [ 0.0047162 ]
  [-0.29770005]
  [-0.49732751]
  [-0.69665539]
  [-1.15240753]
  [-1.30788076]
  [-0.42793113]
  [-0.19250099]
  [-0.02645888]
  [ 0.0447229 ]
  [ 0.13230005]
  [ 0.27332643]
  [ 0.22449352]
  [ 0.17366348]
  [ 0.31709319]
  [ 0.27249038]
  [ 0.07533644]
  [-0.21920963]
  [-0.29585707]
  [-0.01399584]
  [ 0.31173533]
  [ 0.34362358]
  [ 0.38991249]
  [ 0.45729887]
  [ 0.14348882]
  [-0.31445092]
  [-0.2452506 ]
  [ 0.26954317]
  [ 0.24890573]
  [ 0.11890997]
  [ 0.0047248 ]
  [-0.26254639]
  [-0.43562245]
  [-0.00609477]
  [ 0.41147438]
  [ 0.44951177]
  [ 0.33982807]
  [ 0.01781033]
  [-0.37846527]
```

In [12]:
```
plt.title("Forecast vs Actual", fontsize=14)
plt.plot(pd.Series(np.ravel(Y_test)), "bo", markersize=10, label="Actual
")
#plt.plot(pd.Series(np.ravel(Y_test)), "w*", markersize=10)
plt.plot(pd.Series(np.ravel(y_pred)), "r.", markersize=10, label="Foreca
st")
plt.legend(loc="upper left")
plt.xlabel("Time Periods")

plt.show()
```



In [13]:
```
#!/usr/bin/env python
import numpy as np
import matplotlib.mlab as mlab
import matplotlib.pyplot as plt

errors=np.array(errors)
iterations=np.array(iterations)
print(errors.shape)
#print(errors)

plt.hist(errors,iterations,label='errors', facecolor='orange')

plt.xlabel('iterations')
plt.ylabel('mean square error ')
plt.title('histogram of errors>>>>>')
plt.legend()
plt.show()
```

```
(2000,)
```



In [14]:
```
#!/usr/bin/env python
import numpy as np
import matplotlib.mlab as mlab
import matplotlib.pyplot as plt

errors=np.array(errors)
iterations=np.array(iterations)
print(errors.shape)
#print(errors)

plt.plot(errors,iterations,label='errors',color='green')

plt.xlabel('iterations')
plt.ylabel('mean square error ')
plt.title('plot of errors>>>>>')
plt.legend()
plt.show()
```

```
(2000,)
```