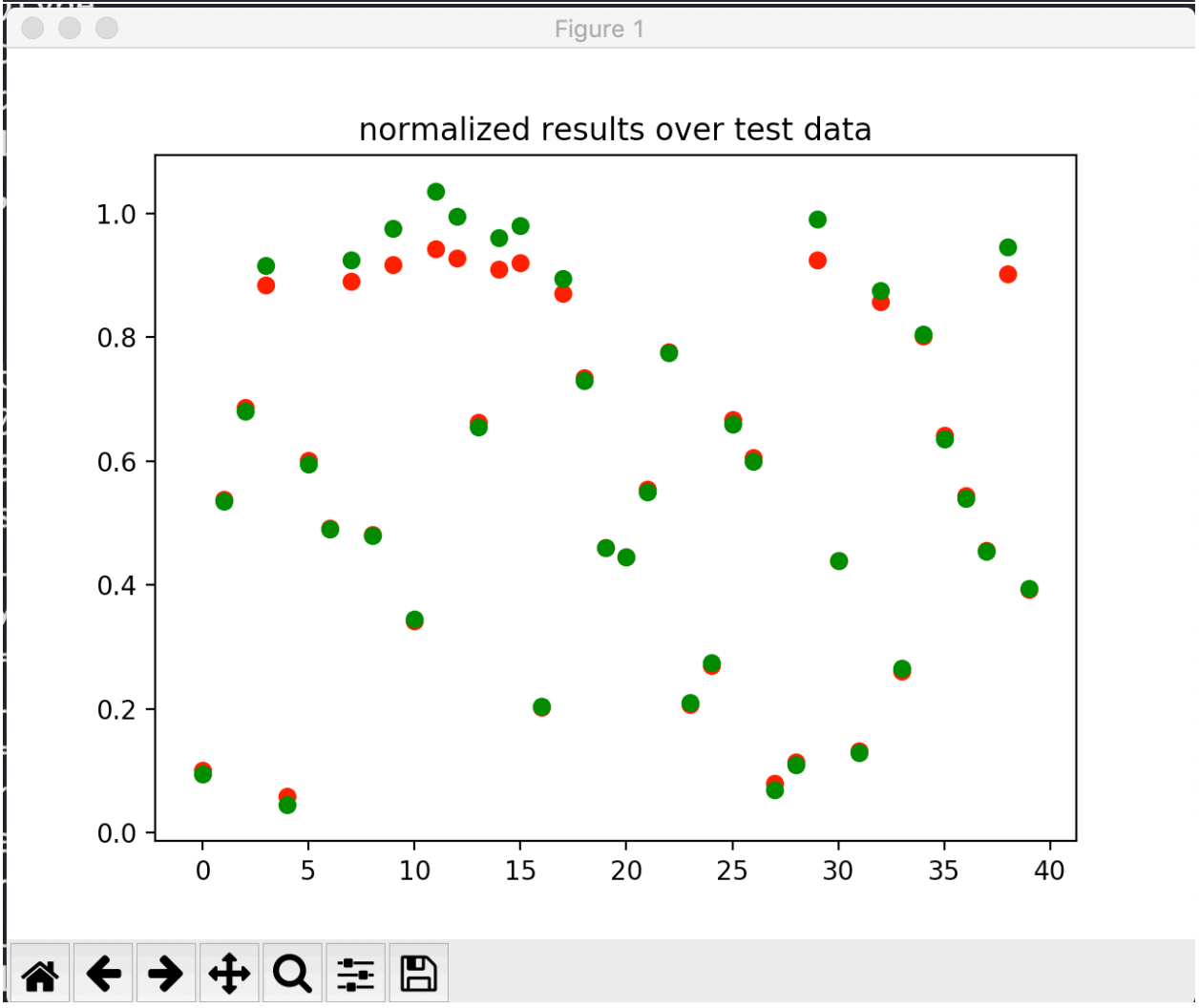
Week7 – Satish Ramachandran

Problem #1

```
'''
Week#7 - Problem #1
LSTM - next number in series predictor
'''

import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from keras.models import Sequential

#Using TensorFlow backend.
from keras.layers import LSTM

def create_dataset():
    #Create 200 series of numbers, 8 in each
    data_set = [[ [i+j] for i in range(8)] for j in range (200)]
    print('Created dataset...')
    print(data_set[0:3])
    print(data_set[-3:])

    #Create 200 target, one for each series created earlier
    target_set = [(i+8) for i in range(200)]
    print('Created target...')
    print(target_set[0:3])
    print(target_set[-3:])
    np_ds = np.array(data_set, dtype=float)
    np_target = np.array(target_set, dtype = float)

    #Scale it so that the model trains accurately.
    return np_ds/200, np_target/200

def create_train_test_set(data, target):
    x_train, x_test, y_train, y_test = train_test_split(data, target, test_size=0.2,
```

```
                                        random_state=4)
    return x_train, x_test, y_train, y_test

def create_train_model(series_train, series_test, target_train, target_test):
    model = Sequential()

    # Add the LSTM
    model.add(LSTM((1), batch_input_shape=(None,8,1), return_sequences=False))
    #model.add(LSTM((1), return_sequences=False))
    model.compile(loss='mean_absolute_error', optimizer='adam',
metrics=['accuracy'])

    # Dump model parameters
    model.summary()

    # Train the model
    history = model.fit(series_train, target_train, epochs=800,
validation_data=(series_test, target_test), verbose=0)

    results = model.predict(series_test)
    plt.title('normalized results over test data')
    plt.scatter(range(40), results, c='r')
    plt.scatter(range(40), target_test, c='g')
    plt.waitforbuttonpress()
    plt.close()

    # Plot the loss Function
    plt.title('loss function')
    plt.plot(history.history['loss'])
    plt.waitforbuttonpress()

#Create the dataset
series, target = create_dataset()
#Split into testing and training sets
series_train, series_test, target_train, target_test = create_train_test_set(series,
target)
create_train_model(series_train, series_test, target_train, target_test)
```

```
Created dataset...
[[[0], [1], [2], [3], [4], [5], [6], [7]], [[1], [2], [3], [4], [5], [6], [7], [8]], [
[2], [3], [4], [5], [6], [7], [8], [9]]]
[[[197], [198], [199], [200], [201], [202], [203], [204]], [[198], [199], [200], [201]
, [202], [203], [204], [205]], [[199], [200], [201], [202], [203], [204], [205], [206]
]]
Created target...
[8, 9, 10]
[205, 206, 207]
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
lstm_1 (LSTM)                (None, 1)                 12
=================================================================
Total params: 12
Trainable params: 12
Non-trainable params: 0
_________________________________________________________________
```



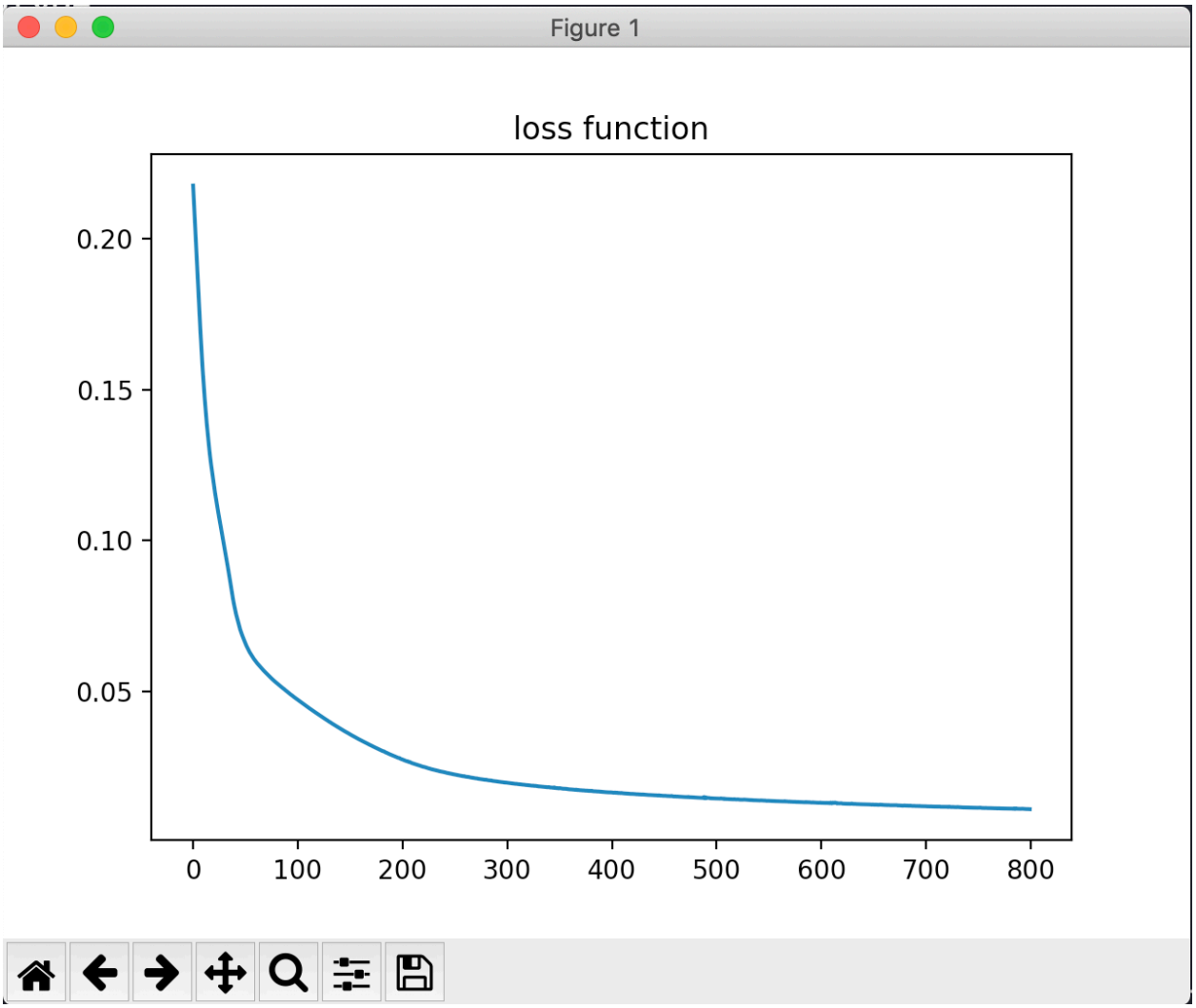


Figure 1
loss function
0.20
0.15
0.10
0.05
0
100
200
300
400
500
600
700
800

Problem #2

```
'''
Week #7 - Problem #2
Source code already provided.
Just making minor modifications
'''

import numpy as np
import pylab as plt

# map cell to cell, add circular cell to goal point
points_list = [(0,1), (1,5), (5,6), (5,4), (1,2), (2,3), (2,7), (7,8)]


goal = 8

import networkx as nx
G=nx.Graph()
G.add_edges_from(points_list)
pos = nx.spring_layout(G)
nx.draw_networkx_nodes(G,pos)
nx.draw_networkx_edges(G,pos)
nx.draw_networkx_labels(G,pos)
plt.title('realized graph')
plt.show()

# how many points in graph? x points
MATRIX_SIZE = 9

# create matrix x*y
R = np.matrix(np.ones(shape=(MATRIX_SIZE, MATRIX_SIZE)))
R *= -1

R

# assign zeros to paths and 100 to goal-reaching point
```

```
for point in points_list:
    print(point)
    if point[1] == goal:
        R[point] = 100
    else:
        R[point] = 0

    if point[0] == goal:
        R[point[::-1]] = 100
    else:
        # reverse of point
        R[point[::-1]]= 0
R
# add goal point round trip
R[goal,goal]= 100

R
###############################################


Q = np.matrix(np.zeros([MATRIX_SIZE,MATRIX_SIZE]))

Q

# learning parameter
gamma = 0.8

initial_state = 1

def available_actions(state):
    current_state_row = R[state,]
    av_act = np.where(current_state_row >= 0)[1]
    return av_act

available_act = available_actions(initial_state)

def sample_next_action(available_actions_range):
```

```
    next_action = int(np.random.choice(available_act,1))
    return next_action

action = sample_next_action(available_act)

def update(current_state, action, gamma):

  max_index = np.where(Q[action,] == np.max(Q[action,]))[1]

  if max_index.shape[0] > 1:
      max_index = int(np.random.choice(max_index, size = 1))
  else:
      max_index = int(max_index)
  max_value = Q[action, max_index]

  Q[current_state, action] = R[current_state, action] + gamma * max_value
  print('max_value', R[current_state, action] + gamma * max_value)

  if (np.max(Q) > 0):
    return(np.sum(Q/np.max(Q)*100))
  else:
    return (0)

update(initial_state, action, gamma)



# Training
scores = []
for i in range(700):
    current_state = np.random.randint(0, int(Q.shape[0]))
    available_act = available_actions(current_state)
    action = sample_next_action(available_act)
    score = update(current_state,action,gamma)
    scores.append(score)
    print ('Score:', str(score))

print("Trained Q matrix:")
```

```
print(Q/np.max(Q)*100)

# Testing
current_state = 0
steps = [current_state]

while current_state != goal:

   next_step_index = np.where(Q[current_state,]
      == np.max(Q[current_state,]))[1]

   if next_step_index.shape[0] > 1:
      next_step_index = int(np.random.choice(next_step_index, size = 1))
   else:
      next_step_index = int(next_step_index)

   steps.append(next_step_index)
   current_state = next_step_index

print("Most efficient path:")
print(steps)

plt.title('scores')
plt.plot(scores)
plt.show()
```

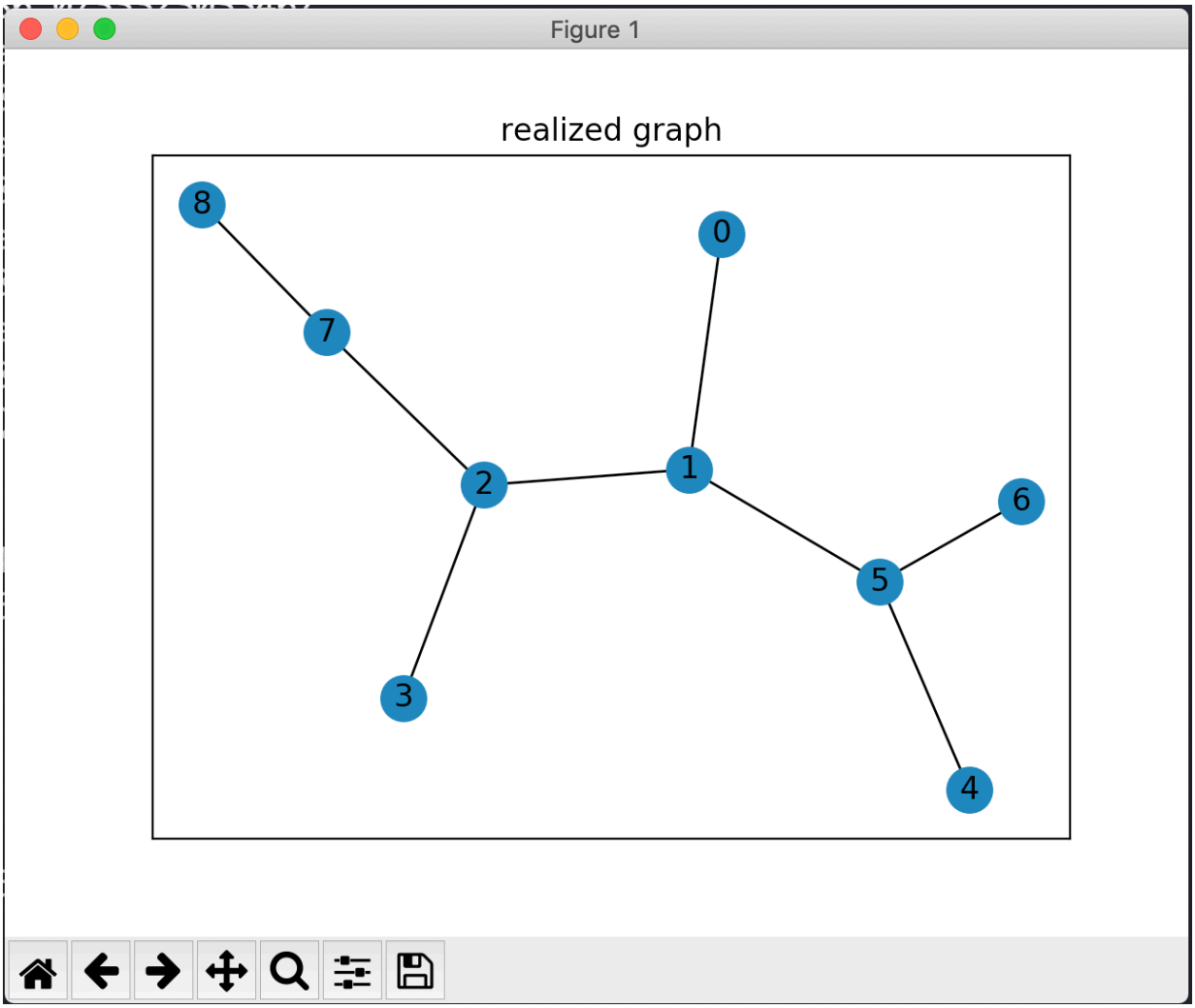


Figure 1
realized graph
8
0
7
2
1
6
5
3
4

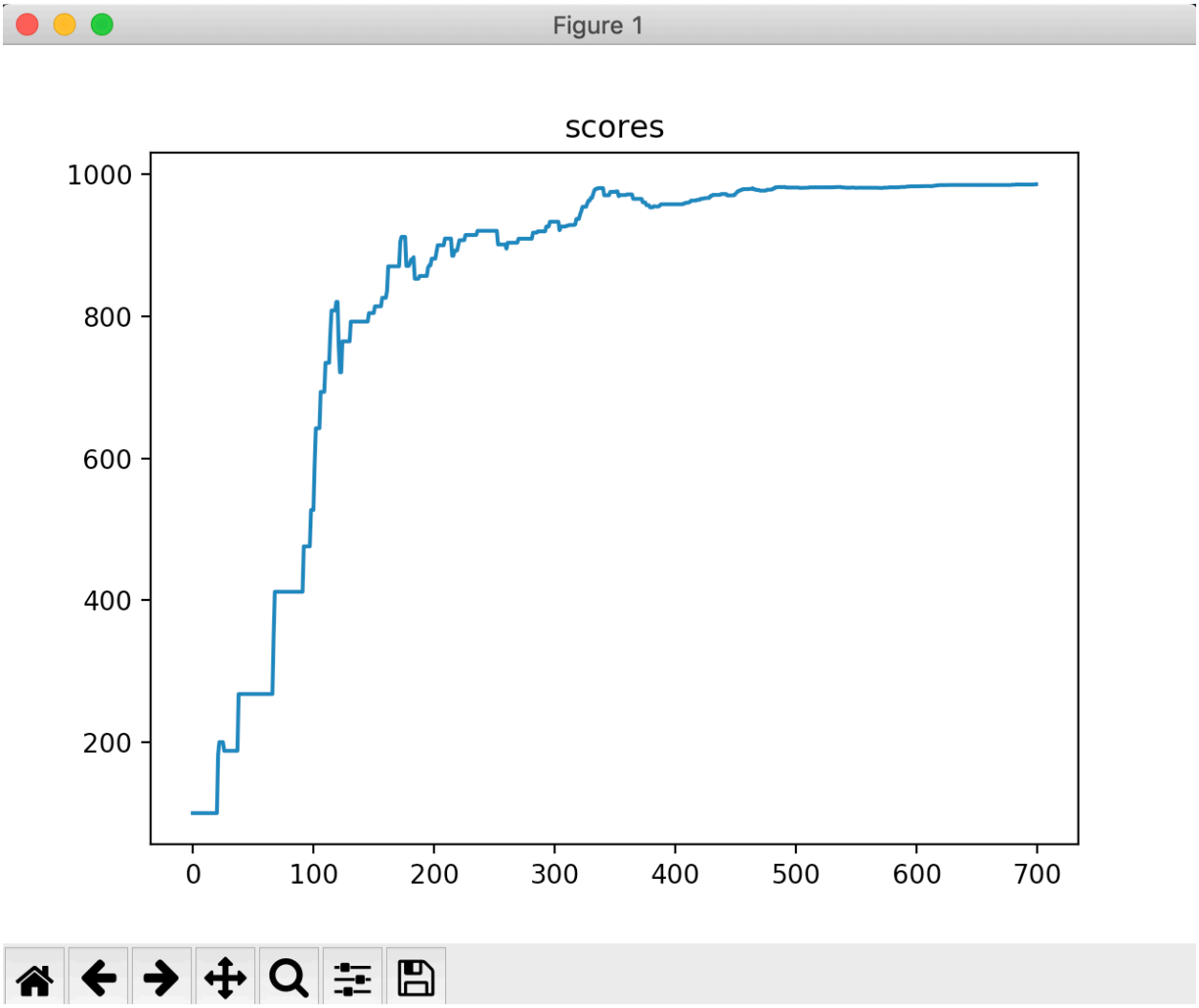


Figure 1
scores
1000
800
600
400
200
0
100
200
300
400
500
600
700

```
Trained Q matrix:
[[  0.          51.18019633   0.           0.           0.
    0.           0.           0.           0.        ]
 [ 40.94415707   0.          63.97524542   0.           0.
   40.94415707   0.           0.           0.        ]
 [  0.          51.18019633   0.          51.18831259   0.
    0.           0.          79.99188374   0.        ]
 [  0.           0.          63.993507     0.           0.
    0.           0.           0.           0.        ]
 [  0.           0.           0.           0.           0.
   40.94415707   0.           0.           0.        ]
 [  0.          51.18019633   0.           0.          32.75532565
    0.          32.75532565   0.           0.        ]
 [  0.           0.           0.           0.           0.
   40.94415707   0.           0.           0.        ]
 [  0.           0.          63.993507     0.           0.
    0.           0.           0.          99.98985468]
 [  0.           0.           0.           0.           0.
    0.           0.          79.99188374 100.        ]]
Most efficient path:
[0, 1, 2, 7, 8]
```