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()

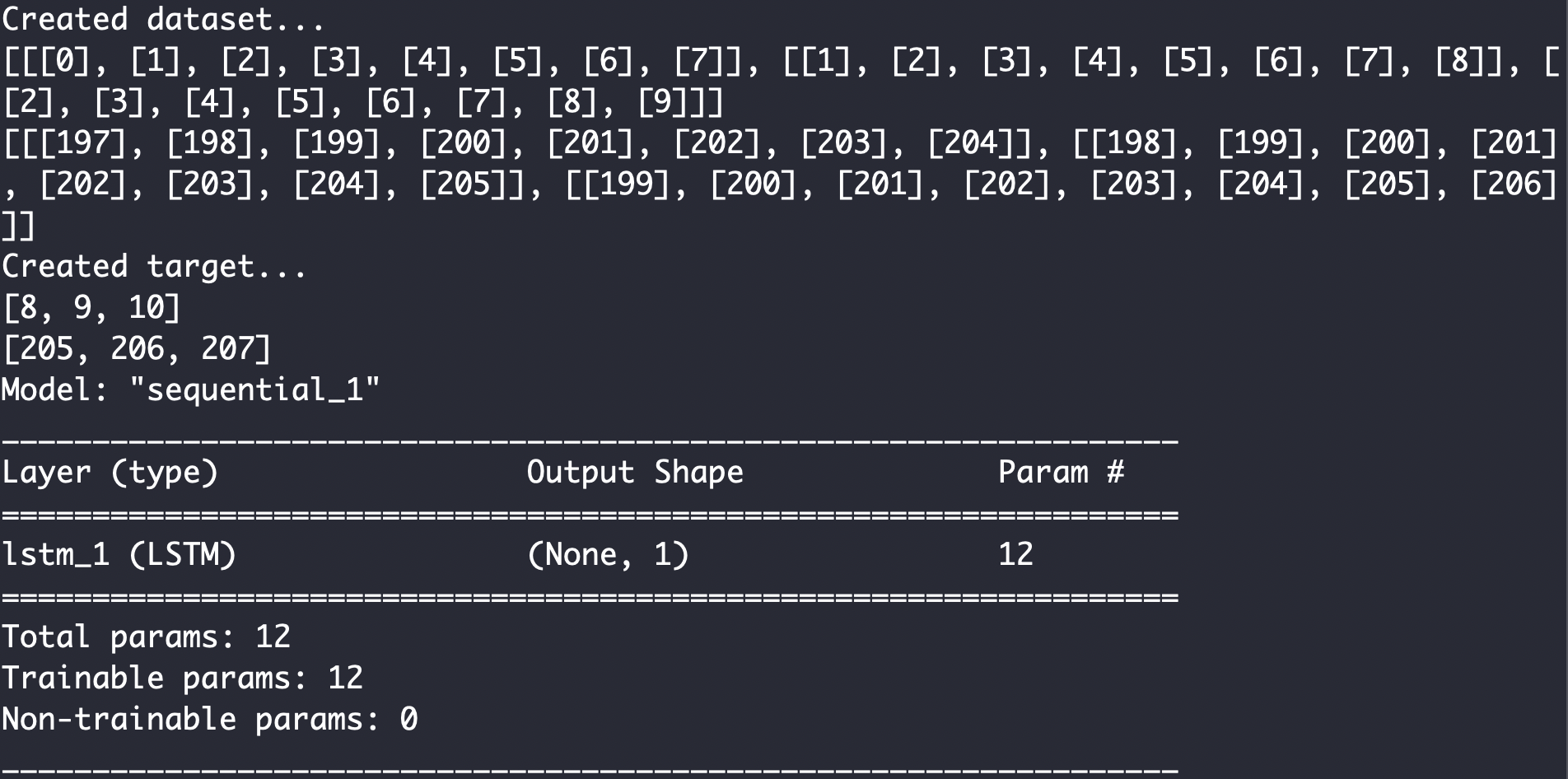
#Create the dataset

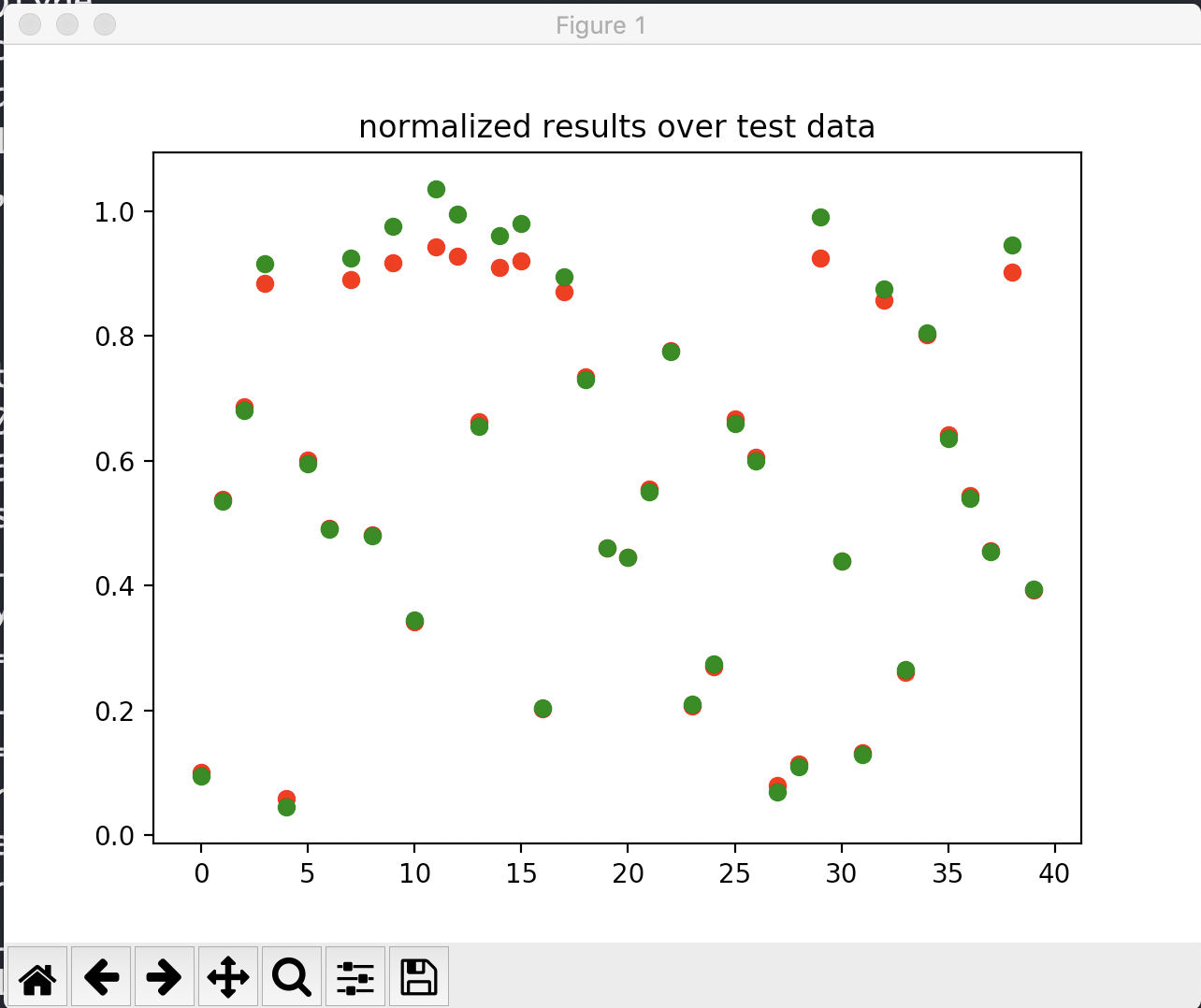
series, target = create\_dataset()

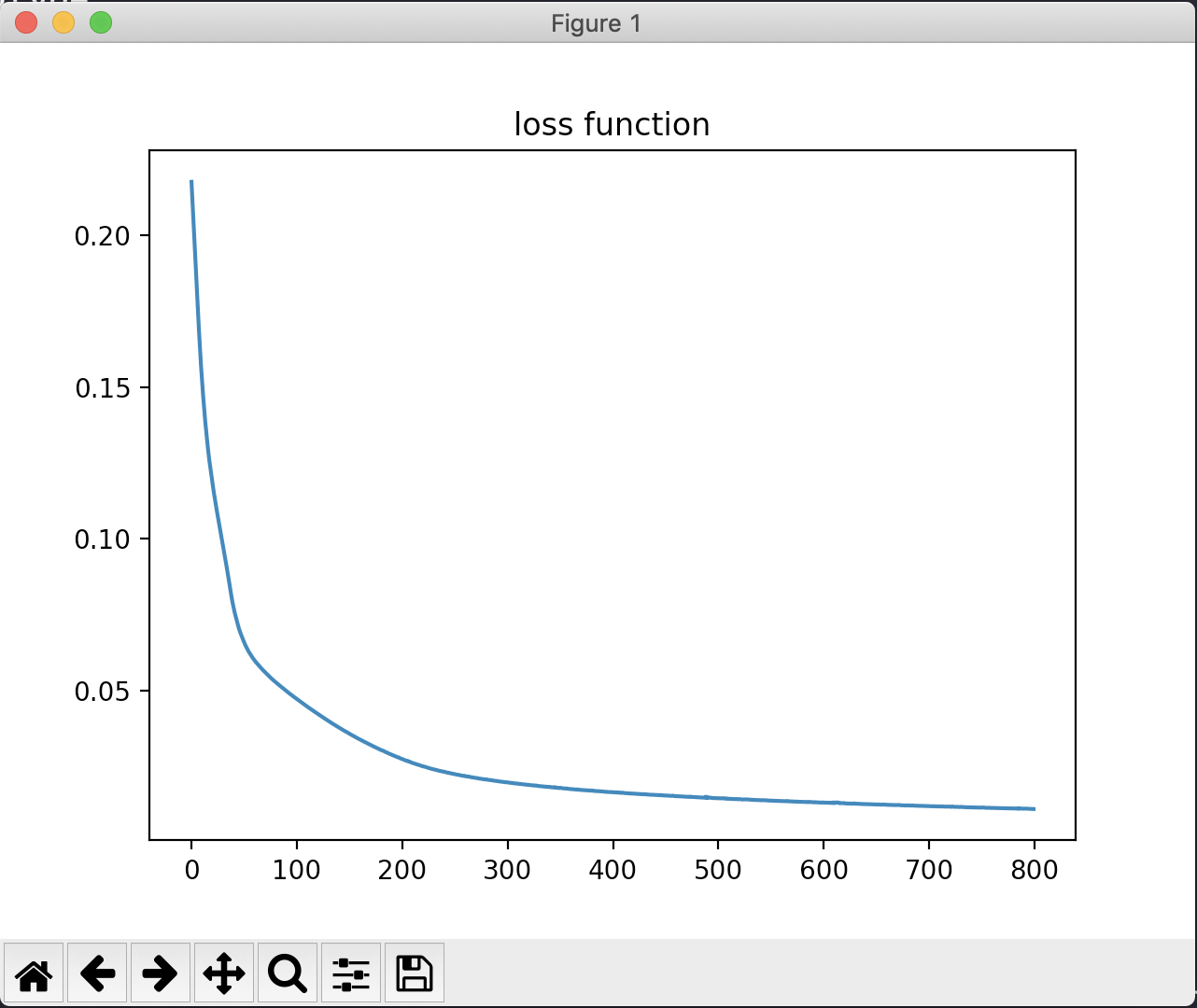
#Split into testing and training sets

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)







# 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()

