Final DQN

December 1, 2020

[ ]:
```
import numpy as np
import random
import time

# from IPython.display import clear_output
from collections import deque

import matplotlib.pyplot as plt
import gym

import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import Model, Sequential
from tensorflow.keras.layers import Dense, Embedding, Reshape
from tensorflow.keras.optimizers import Adam
```

[ ]:
```
'''
ReplayBuffer is used to store and sample experiences. It is initialized with a␣
 ↪maximum number
of experiences that it can store.

Methods:
add: adds a new experience to the end of the ReplayBuffer queue

sample: randomly samples batch_size number of episodes to provide for training␣
 ↪the q_network
'''

class ReplayBuffer:
    def __init__(self, maxlen):
        self.buffer = deque(maxlen=maxlen)  #number of experiences to store

    def add(self, experience):
        self.buffer.append(experience)

    def sample(self, batch_size):
        sample_size = min(len(self.buffer), batch_size)
        samples = random.choices(self.buffer, k=sample_size)
        return map(list, zip(*samples))
```

[ ]:
```
'''
DQAgent
Builds an AI agent using a DQN. Initialized with a gamma value, epsilon value,␣
 ↪a minimum value for epsilon,
an environment, an optimizer, input dimensions, and the sizes for the 2 fully␣
 ↪connected layers of the model.

Methods:
store: If an episode is terminated, the epsilon value is reduced by the decay␣
 ↪rate.
This pushes the agent from exploration to exploitation as the agent learns.
Also stores the episodes in the replay buffer using the add method of the␣
 ↪ReplayBuffer class.

build_dqn: Builds a Sequental model with 3 Dense layers. The input layersize is
the shape of the observation space of the environment. The output layer is the␣
 ↪number
of available actions. The fc1 and fc2 input are the dimensions of the input and␣
 ↪hidden layer.

align_target_model: Initializes the target network with the same weights as the
initial q_network.

act: Gets the action the agent will take based on the state, the max value of
the available actions, and the epsilon greedy method. The action returns a␣
 ↪random action
with the probability of epsilon.

retrain: Updates the q_network using random samples from the stored episodes in
the ReplayBuffer, and based on the Bellman equation, using the target
network to stabilize the learning.

'''
class DQAgent:
    def __init__(self, gamma, epsilon, eps_min, environment, optimizer,␣
 ↪input_dims, fc1, fc2):
        #initialize state and action space based on environment object
        self.state_size = env.observation_space.shape[0]
        self.action_size = environment.action_space.n
        self.optimizer = optimizer
        self.gamma = gamma
        self.epsilon = epsilon
        self.eps_min = eps_min
        #initilize the replay memory
        self.replay_buffer = ReplayBuffer(maxlen=10000)

        #build the Q Network and Target network
        self.q_network = self.build_dqn(self.action_size, input_dims, fc1, fc2)
        self.target_network = self.build_dqn(self.action_size, input_dims, fc1,␣
↪fc2)

        #align the weights
        self.align_target_model()

    #append experience to experience memory
    def store(self, state, action, reward, next_state, terminated):
        if terminated:
            self.epsilon = max(eps_min, 0.99 * self
                                .epsilon)
        self.replay_buffer.add((state, action, reward, next_state,␣
↪(1-int(terminated))))


    #Build deep q network
    def build_dqn(self, n_actions, input_dims, fc1, fc2): #fc = fully connected␣
↪dimensions
        model = keras.Sequential([
            keras.layers.Dense(fc1, input_shape = input_dims,␣
↪activation='relu'),
            keras.layers.Dense(fc2, activation='relu'),
            keras.layers.Dense(n_actions, activation=None), #output layer,␣
↪n_actions is number of available actions
        ])

        model.compile(loss='mse', optimizer = self.optimizer)
        return model

    def align_target_model(self):
        self.target_network.set_weights(
            self.q_network.get_weights()) #passes q_network weights to target␣
↪model

    #exploration vs. exploitation with probability of epsilon
    def act(self, state):
        state = np.array([state])
        q_values = self.q_network.predict(state)
        action_greedy = np.argmax(q_values[0])
        action_random = np.random.randint(self.action_size)
        action = action_random if random.random() < self.epsilon else␣
↪action_greedy

        return action


    #take random samples from experience replay memory and train the q_network
    def retrain(self, batch_size):

        states, actions,rewards, next_states,terms = self.replay_buffer.
↪sample(batch_size)

        states = np.array(states)
        next_states = np.array(next_states)

        q_network = self.q_network.predict(states)
        q_next = self.q_network.predict(next_states)

        q_target = np.copy(q_network)
        batch_index = np.arange(len(q_target), dtype=np.int32)


        q_target[batch_index, actions] = rewards + self.gamma * np.ndarray.
↪max(q_next, axis=1) * terms

        self.q_network.train_on_batch(states,q_target)
```

[ ]:
```
env_name = 'LunarLander-v2'
env = gym.make(env_name)
env = gym.wrappers.Monitor(env, "./vid3", video_callable=lambda episode_id:␣
↪episode_id%10==0)
'''
Hyperparameters are all here so I can tune them all in the same spot

Learning rate
gamma
epsilon
optimizer
Dense layer shape
'''
lr = 0.0005
gam = 0.99
eps = 1.0
eps_min = 0.01
opt = Adam(learning_rate=lr)
```

```
dims = env.observation_space.shape
layer_1 = 64
layer_2 = 64

agent = DQAgent(gamma = gam, epsilon = eps, eps_min = eps_min, environment =␣
↪env, optimizer = opt,
                input_dims = dims, fc1 = layer_1, fc2 = layer_2)
```

```
[ ]: batch_size = 64
num_episodes = 500
total_rewards = []
start_time = time.time()
for e in range(num_episodes):
    #reset the environment to get random initial state
    state = env.reset()

    total_reward = 0
    terminated = False

    while not terminated:
        env.render()
        #gets the action for the agent to take at the current state
        action = agent.act(state)

        # take action; returns the next state, the reward for taking the␣
↪action, and whether the episode terminated
        next_state, reward, terminated, info = env.step(action)

        total_reward += reward

        agent.store(state, action, reward, next_state, terminated) #add␣
↪experience to buffer
        state = next_state
        agent.retrain(batch_size) # retrain = store experience in buffer, get␣
↪samples
    total_rewards.append(total_reward)
    print("Episode: {}, total reward: {}, epsilon: {}".format(e,total_reward,␣
↪agent.epsilon))

print("Total time: %s hours" % ((time.time() - start_time)/3600))



env.close()
```

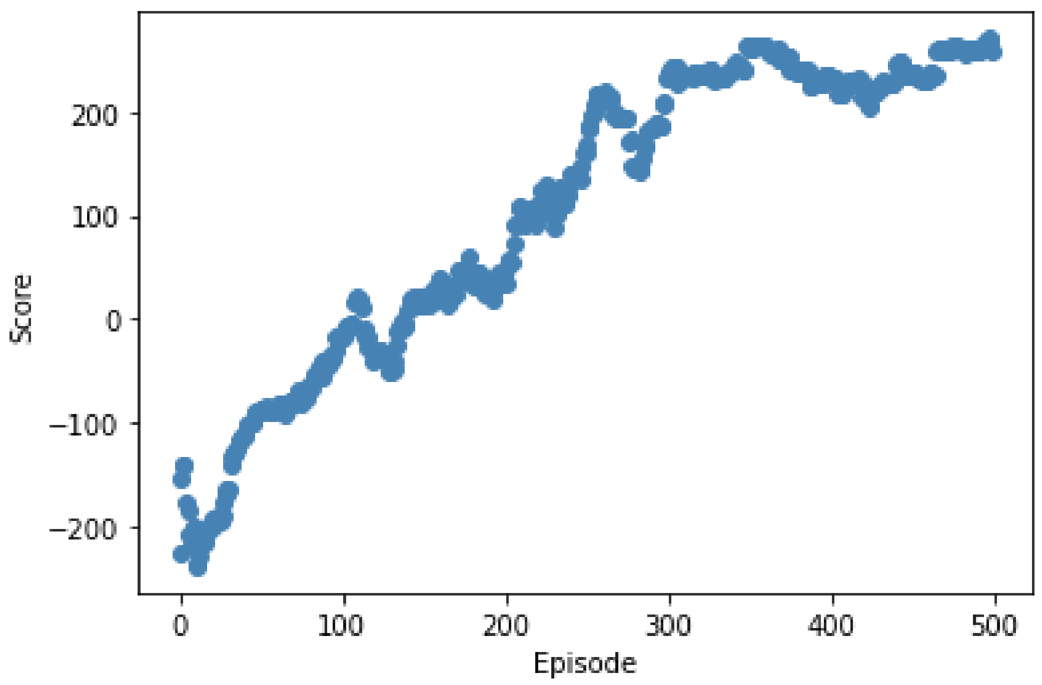
```
Episode: 0, total reward: -226.36609264855755, epsilon: 0.99
Episode: 1, total reward: -81.98728626732843, epsilon: 0.9801
```

```
Episode: 2, total reward: -117.4186861323085, epsilon: 0.9702989999999999
Episode: 3, total reward: -133.33822860907006, epsilon: 0.96059601
Episode: 4, total reward: -327.9103678243599, epsilon: 0.9509900498999999
Episode: 5, total reward: -222.56930181799123, epsilon: 0.9414801494009999
Episode: 6, total reward: -343.95747213187025, epsilon: 0.9320653479069899
Episode: 7, total reward: -286.9153056897922, epsilon: 0.92274469442792
Episode: 8, total reward: -49.20342806855217, epsilon: 0.9135172474836407
Episode: 9, total reward: -248.01033789253708, epsilon: 0.9043820750088043
Episode: 10, total reward: -569.171130442405, epsilon: 0.8953382542587163
Episode: 11, total reward: -254.03479465729066, epsilon: 0.8863848717161291
Episode: 12, total reward: -93.23258253563475, epsilon: 0.8775210229989678
Episode: 13, total reward: -68.28920633598548, epsilon: 0.8687458127689781
Episode: 14, total reward: -151.74482415291595, epsilon: 0.8600583546412883
Episode: 15, total reward: -254.786703657628, epsilon: 0.8514577710948754
Episode: 16, total reward: -102.20061396538556, epsilon: 0.8429431933839266
Episode: 17, total reward: -174.51998236643368, epsilon: 0.8345137614500874
Episode: 18, total reward: -99.73442698347371, epsilon: 0.8261686238355865
Episode: 19, total reward: -130.15962513962756, epsilon: 0.8179069375972307
Episode: 20, total reward: -249.9733263448847, epsilon: 0.8097278682212583
Episode: 21, total reward: -95.62231176081617, epsilon: 0.8016305895390458
Episode: 22, total reward: -100.75747783398128, epsilon: 0.7936142836436553
Episode: 23, total reward: -60.923232845670995, epsilon: 0.7856781408072188
Episode: 24, total reward: -88.45148056183888, epsilon: 0.7778213593991465
Episode: 25, total reward: -467.81226811707444, epsilon: 0.7700431458051551
Episode: 26, total reward: -118.34492753330767, epsilon: 0.7623427143471035
Episode: 27, total reward: -51.38744027151198, epsilon: 0.7547192872036325
Episode: 28, total reward: -33.19319687786752, epsilon: 0.7471720943315961
Episode: 29, total reward: -127.31453906176938, epsilon: 0.7397003733882802
Episode: 30, total reward: -125.75745475003427, epsilon: 0.7323033696543974
Episode: 31, total reward: -86.51701091779725, epsilon: 0.7249803359578534
Episode: 32, total reward: -101.13557742279114, epsilon: 0.7177305325982748
Episode: 33, total reward: -3.9027197564845295, epsilon: 0.7105532272722921
Episode: 34, total reward: -98.13734253375846, epsilon: 0.7034476949995692
Episode: 35, total reward: -54.641305162261816, epsilon: 0.6964132180495735
Episode: 36, total reward: -93.25611017122029, epsilon: 0.6894490858690777
Episode: 37, total reward: -107.71910931681475, epsilon: 0.682554595010387
Episode: 38, total reward: -98.34610838544181, epsilon: 0.6757290490602831
Episode: 39, total reward: -87.38284752998506, epsilon: 0.6689717585696803
Episode: 40, total reward: -107.0273733178029, epsilon: 0.6622820409839835
Episode: 41, total reward: -53.70480246709469, epsilon: 0.6556592205741436
Episode: 42, total reward: -67.79430918936475, epsilon: 0.6491026283684022
Episode: 43, total reward: -66.745031882329, epsilon: 0.6426116020847181
Episode: 44, total reward: -30.85460877922975, epsilon: 0.6361854860638709
Episode: 45, total reward: -108.75615380498557, epsilon: 0.6298236312032323
Episode: 46, total reward: -305.3790542004173, epsilon: 0.6235253948912
Episode: 47, total reward: -41.42904970893622, epsilon: 0.617290140942288
Episode: 48, total reward: -51.68571603462939, epsilon: 0.6111172395328651
Episode: 49, total reward: -39.25019403213855, epsilon: 0.6050060671375365
```

```
Episode: 482, total reward: 276.6508272807346, epsilon: 0.01
Episode: 483, total reward: 281.87500171566927, epsilon: 0.01
Episode: 484, total reward: 273.21894179172455, epsilon: 0.01
Episode: 485, total reward: 278.16095859890544, epsilon: 0.01
Episode: 486, total reward: 266.7587338951323, epsilon: 0.01
Episode: 487, total reward: 253.9807375541103, epsilon: 0.01
Episode: 488, total reward: 275.5350852823379, epsilon: 0.01
Episode: 489, total reward: 265.6217805030024, epsilon: 0.01
Episode: 490, total reward: 256.80287936678485, epsilon: 0.01
Episode: 491, total reward: 255.381201789181, epsilon: 0.01
Episode: 492, total reward: 296.6895356055095, epsilon: 0.01
Episode: 493, total reward: 306.0171677649415, epsilon: 0.01
Episode: 494, total reward: 302.12692456273766, epsilon: 0.01
Episode: 495, total reward: 275.4475270169578, epsilon: 0.01
Episode: 496, total reward: 295.9571419430454, epsilon: 0.01
Episode: 497, total reward: 257.74485904243346, epsilon: 0.01
Episode: 498, total reward: 305.1925557371201, epsilon: 0.01
Episode: 499, total reward: -13.372360381014218, epsilon: 0.01
Total time: 19.424482261472278 hours
```

[ ]:
```python
# Create graph for running average of scores to check if agent is learning or␣
↪overtraining

x = [i for i in range(num_episodes)]
N = len(total_rewards)
running_avg = np.empty(N)
for t in range(N):
    running_avg[t] = np.mean(total_rewards[max(0, t-20):(t+1)])
plt.xlabel('Episode')
plt.ylabel('Score')
plt.scatter(x, running_avg, color="steelblue");
```



200
100
0
−100
−200
Score
0
100
200
300
400
500
Episode

```
[ ]:
```