I. Problem Statement

The model "Acrobot-v1" in Open AI [1], hereinafter referred as the "environment", is a dynamic system with an external input, which constitutes two links with two hinges of single degree of freedom. The arrangement is shown in Fig. 1, where the first link is hinged at a point on a fixed frame, while its other end is hinged with the second link. The state vector (say $s := [s^1, s^2, s^3, s^4, s^5, s^6] \in \mathbb{R}^6$) of the environment is $[\cos(\theta_1), \sin(\theta_1), \cos(\theta_2), \sin(\theta_2), \dot{\theta}_1, \dot{\theta}_2]$, where θ_1 is the angle of inclination of the upper link wrt the vertical axis, and θ_2 is the angle of the lower link, relative to the upper one. The external input is a torque that is applied at the hinge between the two links, and can take three possible values: 0 (no torque applied), +1 (clockwise unit torque), or -1 (anticlockwise unit torque). In other words the action space ($\mathcal{A} := \{0, +1, -1\}$) of the model has three possible actions, while the state space ($\mathcal{S} := \{s \mid |s^1|, |s^2|, |s^3|, |s^4| \le 1,\ s^5, s^6 \in \mathbb{R}\}$) is uncountably infinite. In an episode, the environment evolves over discrete time steps, starting from the initial state (where both the links are vertically pointing downwards, can also be seen as the stable equilibrium of the dynamic assembly), where in every time step t one action ($a_t \in \mathcal{A}$) is taken (possibly perturbing the environment from its equilibrium), until the free tip of the lower link touches the horizontal line. Also in every time step t the environment generates a reward of -1, and returns an observation of the states ($s_t \in \mathcal{S}$). Hence if it takes n time steps for an episode to complete, the total score for the episode turns out to be $-n$.

Fig. 1. Schematic representation of Acrobot-v1

The goal is to design a control agent, such that it automatically computes an optimal policy to maximize the total score in the sequential episodes (i.e. to minimize the time steps taken to complete the episodes), while the agent doesn't know the dynamics of the environment explicitly. However, the agent can probe the environment by taking actions, and learn about the environment from the gained experiences (in terms of the observed states and the rewards).

II. Approach

The reinforcement learning (RL) algorithms [2] are designed to determine an optimal control policy for driving a environment from its initial state for a given goal, typically in presence of uncertainty about the environment, where the control policy is defined as a function of the state vector. While the algorithm starts with a random control policy that is possibly sub-optimal, it eventually adapts towards the optimal policy, by run-time iterative optimization of a well-defined return, which is the total discounted reward

accumulated along the trajectory of an episode (for episodic tasks). Such return computed at a given state for a given action (and thereafter following a certain policy), is also known as the state-action value or the Q-value (denoted by $Q(s,a)$) corresponding to the state, action pair (for the policy). Typically, the agent is allowed to explore the environment at a predefined exporation rate ϵ, by taking random actions, in order to effectively learn the uncertain environment by experiencing the rewards and the state transitions, and therefrom learn the optimal policy for the environment. The problem stated above, can indeed be posed as one of the RL problems.

Since the agent doesn't know the dynamics of the environment, finding the optimal policy requires estimation of the optimal state-action values (i.e. Q-values, in which case the optimal action in any time-step is simply: $a_t^* = \arg\max_{a \in \mathcal{A}} Q(s_t, a)$). The continuous state space requires discretization, for a tabular representation of the Q-values corresponding to the states in $\mathcal{S}$. Since the last two states (s^5, s^6) of the state vector can take any real value over the state-space, discretization of the state-space with an appropriate trade-off between the accuracy and complexity, is not straightforward. Instead, we can get rid of the computational complexity of using tabular representation, yet maintain the accuracy to an acceptable extent, by using a function approximator that approximates the function $f : \mathcal{S} \to \mathbb{R}^3$ that maps the state space to the set of Q-values corresponding to the three possible actions in $\mathcal{A}$.

The function approximator can be designed either using a single layer neural network, or using a deep neural network (DNN) that is allowed to contain multiple hidden layers. In the former approach, one needs to map the state vector into a linearly independent and *appropriate* (for approximating the function f) feature space, before using it to train the network, while a DNN can be designed to be capable of extracting the features directly from the state-vector in course of training. The trade-off is in terms of the computational complexity which is higher in the latter, as compared to the former. However, the latter is more flexible and general in terms of application, owing to its ability of self-feature-extraction. Here, Q-learning algorithm has been used to train the agent, where in every time step (technically, iterative step), the estimate of Q-values are updated based on a policy different from that, under which the current action has taken place. This approach is termed as Deep Q Network (DQN) [3] in literature. While [3] try to solve similar control problem (but with a different environment) by an indirect approach, in which the rendered image of the environment is extracted in each time step, and a running convolution neural network (CNN) is trained to learn the Q values, with a target network that is periodically cloned from the running network. The algorithm used here in different [3] in that it directly utilize the observation and reward returned by the environment for training a fully connected neural network. The algorithm used here is provided in Section IV.

III. Architecture of DNN

The DNN (Q) that is used here to approximate f, has two hidden layers, where the first layer has 32 neurons, and the number of neurons in the second layer is 64. The input layer has six nodes, which correspond to the respective elements of the state vector. The output layer has three nodes, each of which indicate the Q-value corresponding to a particular action. The neurons in the hidden layers have ReLu activations, and all the layers in the DNN are fully connected.

IV. Pseudo-code for Algorithm

The algorithm used for training of the agent is as follows:

a. Set parameters ϵ_{max} (starting rate of exploration), ϵ_{step} (the steps of decay of ϵ), ϵ_{min} (minimum value of epsilon), B (batch size) and γ (discount factor). Also set $\epsilon = \epsilon_{max}$.
b. Initialize replay memory D to sufficiently large capacity N
c. Initialize the weights of the function approximator Q with random weights θ.
d. For episode = 1 to M :
    i. Reset environment
    ii. For every time step t starting from 1, until $t = T$ or end of the episode, whichever is earlier:
        1. Randomly choose an ϵ-greedy action a_t (i.e. choose a random action with probability ϵ, or the action $\arg\max_{a \in \mathcal{A}} Q(s_t, \theta, a)$ with probability $1-\epsilon$)
        2. Take action a_t and observe the reward r_t and the next state s_{t+1}
        3. Store the tuple $[s_t, a_t, r_t, s_{t+1}, done]$ in D, where $done$ is a binary flag that is set if and only if the current episode terminates.
        4. Randomly sample B tuples (possibly repetitive) from D
        5. Set for every sample i :

$$\hat{Q}_i = \begin{cases} r_i & \text{if episode terminates at } (i+1)^{th} \text{ step} \\ r_i + \gamma Q(r_{i+1}, a_i, \theta) & \text{otherwise} \end{cases}$$

        6. Perform a gradient descent step on $(\hat{Q}_i - Q(s_i, a_i, \theta))^2$ with respect to θ
    iii. If $\epsilon > \epsilon_{\min}$, then $\epsilon = \epsilon - \epsilon_{step}$

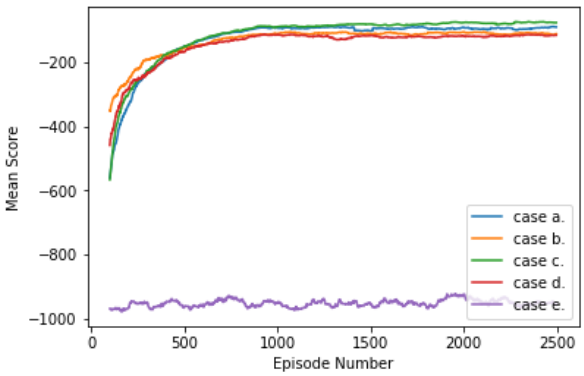
V. Results

The above algorithm is implemented in Python, using the package Keras [4] for training the DNN. The implementation code is provided in the Appendix. With the general algorithm parameters set at the following values: $M = 2500$, $N = 100,000$, $T = 1000$, $\epsilon_{\min} = 0.1$, $\epsilon_{max} = 0.9$, $\epsilon_{step} = 0.001$, and $\gamma = 0.99$, the performance for following five different cases are investigated:

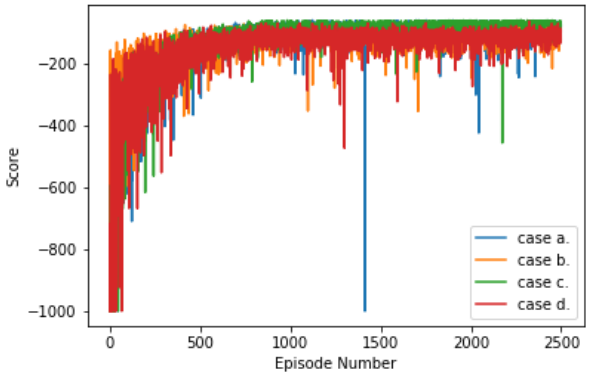
a. Setting the batchsize to $B = 32$
b. Same as a. except that the step d.ii.1 of the pseudo-code is performed every fourth time step, and in the other time steps, the action taken in the last time step is repeated. Also a third fully connected hidden layer with 64 ReLu activated is added in the Q-network.
c. Same as a. except that a third fully connected hidden layer with 64 ReLu activated is added
d. Same as a. except that the step d.ii.1 of the pseudo-code is performed every fourth time step, and in the other time steps, the action taken in the last time step is repeated
e. Taking random action without a control agent

The implementation code of case b. is provided in the Appendix, while the code for the other cases are trivial modification of that of b. The executable codes (.py) are however, submitted for all of them

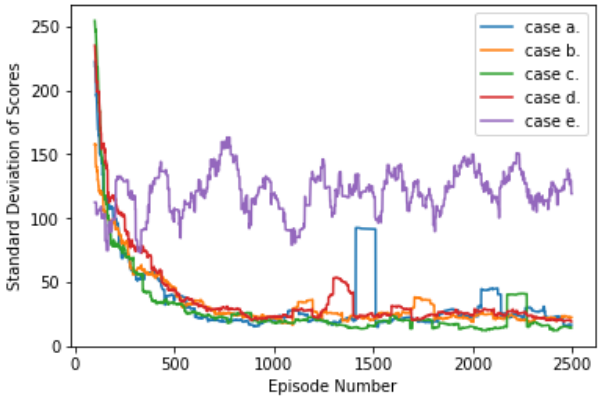
individually. The comparison of the performance of the individual cases are shown in Fig.2. While the Q-network architecture of both case a. and case d. are the same, the former achieves a superior mean score over the other, at the cost of having inferior standard deviation. When the architecture is enhanced by adding the third hidden layer (i.e. realizing case c. case and b.), their individual performance improves significantly, while the case c. turns out to be the best. The training times of case c. and b. is much higher (almost 1.5 times) from those of case a. and d., due to additional hidden layer in the DNN. However, the performance of all four agents (a. through d.) are by far superior than the without controller random action case, namely case e. Note that the curve of episode-wise score for case e. is extremely random, and hence is excluded from the Fig. 2 b) for the sake of clarity. Also the curve of DNN loss over time steps is shown only for case c. The final mean and standard deviation at the end of 2500 episodes are compared for the five cases in Table 1.



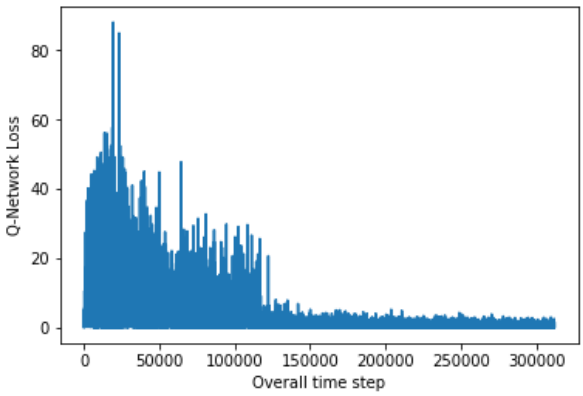
a) Comparison of mean (over last 100 episodes) score



b) Comparison of score (over 2500 episodes)



c) Comparison of standard deviation (over last 100 episodes)



d) Comparison of Q-network loss for case c.

Fig. 2. Curves depicting performance comparison of the five cases (a., b., c., d., and e.) explored

Table 1

Case	Mean over last 100 episodes	Standard Deviation over last 100 episodes
Case a.	-90.71	16.70
Case b.	-112.32	22.39
Case c.	-77.00	14.45
Case d.	-120.84	35.70
Case e.	-958.07	119.17

VI. Lessons Learned and Possible Improvement

The key lesson learned in this project is that the training of a DQN is extremely sensitive to the setting of the hyper-parameters. Especially, the decay rate of ϵ plays a key role in convergence of the algorithm. The capacity of the replay memory is also important, and shall be large enough to accommodate sufficient samples from which effective learning can continually take place.

Uniform sampling from the replay memory renders equal importance to all the historical experiences. An improvement over this algorithm could be to devise a strategy such that the experiences from which we can learn the most, may be given more priority. This may be achieved using the method called "priority sweep" introduced in [5]. Also the function approximator based Q-learning is largely affected by the error in approximation, since the ϵ-greedy action is obtained directly by a greedy search of the optimum Q-value over $\mathcal{A}$. This problem can be partially overcome by using policy gradient based methods e.g. actor-critic based deep reinforcement learning, or by using policy network in place of a value network.

VII. Acknowledgement

The code for DQN control of CartPole environment of OpenAI, provided in the following website is referred for developing the code in this project:

http://www.puzzlr.org/lets-build-a-dqn-simple-implementation/

VIII. Reference

[1] OpenAI, "Gym: A toolkit for developing and comparing reinforcement learning algorithms." [Online]. Available: https://gym.openai.com. [Accessed: 15-Dec-2018].
[2] R. S. Sutton and A. G. Barto, *Reinforcement Learning | The MIT Press*. .
[3] "Human-level control through deep reinforcement learning | Nature." [Online]. Available: https://www.nature.com/articles/nature14236. [Accessed: 17-Dec-2018].
[4] "Home - Keras Documentation." [Online]. Available: https://keras.io/. [Accessed: 17-Dec-2018].
[5] A. W. Moore and C. G. Atkeson, "Prioritized sweeping: Reinforcement learning with less data and less time," *Mach. Learn.*, vol. 13, no. 1, pp. 103–130, Oct. 1993.

```
"""
                                Appendix

Created on Fri Dec 14 14:45:08 2018

@author: talukder
"""

import gym
import numpy as np
import matplotlib.pyplot as plt
from keras import models as nn
from keras import layers as nlayers
from keras import optimizers as optim

# Set the training paremeters
M = 2500    #   Number of episodes
gamma = 0.99    #   Discount factor
epsilon_max = 0.9   #   Max value of exploration rate
epsilon_min = 0.1   #   Min value of exploration rate
epsilon_decay = 0.9 #   Decay rate of exploration after minimum reached
epsilon_step = 0.001    # Reduction step of exploration rate
B = 32  #   Batch size from training DNN
N = 100000  #   Capacity of replay memory
T = 1000    # Max time steps in an episode
epsilon = epsilon_max

# Initialize the result storing registers by empty array
loss = []   # stores the loss of the DNN training in each time step
meanscore = []  # stores the mean of scores of last 100 episodes
stdscore = []   # stores the standard deviation of scores of last 100 episodes
scores = []  # stores rewards of each epsiode
replay_memory = [] # replay memory holds s, a, r, s'

#   Create an instance of the environment
env = gym.make("Acrobot-v1")
env._max_episode_steps = T
action_dim = env.action_space.n
input_dim = env.observation_space.shape[0]

#   Create a DNN model
model = nn.Sequential()
model.add(nlayers.Dense(32, input_dim = input_dim , activation = 'relu'))
model.add(nlayers.Dense(64, activation = 'relu'))
model.add(nlayers.Dense(64, activation = 'relu'))
model.add(nlayers.Dense(action_dim, activation = 'linear'))
model.compile(optimizer = optim.Adam(), loss = 'mse')

# Define replay memory to store the experience from the environment
def replay(replay_memory, minibatch_size):
    global loss
    minibatch = np.random.choice(replay_memory, minibatch_size, replace=True)
    # Extract the state, action, reward, next state, and done from the batch
    state_l =      np.array(list(map(lambda x: x['state'], minibatch)))
    action_l =      np.array(list(map(lambda x: x['action'], minibatch)))
    reward_l =      np.array(list(map(lambda x: x['reward'], minibatch)))
```

```
    next_state_l = np.array(list(map(lambda x: x['next_state'], minibatch)))
    done_l   = np.array(list(map(lambda x: x['done'], minibatch)))
    # Estimate Q-values of 'next states' using current estimate of weights
    qvals_next_state_l = model.predict(next_state_l)
    # Estimate Q-values of the 'states' using current estimate of weights
    target_f = model.predict(state_l) # includes the other actions, states

    for i,(state,action,reward,qvals_next_state, done) in \
          enumerate(zip(state_l,action_l,reward_l,qvals_next_state_l, done_l)):
        # Get the estimate of current states from the next states
        if not done:  target = reward + gamma * np.max(qvals_next_state)
        else:         target = reward
        target_f[i][action] = target
    # Update DNN weights by gradient descent
    mod_ret = model.fit(state_l,target_f, epochs=1, verbose=0)
    # Store the current loss
    loss.append(mod_ret.history)
    return model

# Training loop
for n in range(M):
    state = env.reset() # reset the environment
    done=False
    score = 0   # initialize score to zero
    time = 0    # initialize episode time to zero
    while not done:
        # Uncomment this to see the agent learning
        env.render()
        # Feedforward pass for current state to get predicted q-values for all actions
        qvals_state = model.predict(state.reshape(1,6))
        # Choose action to be epsilon-greedy
        if time%4 == 0:
            if np.random.random() < epsilon: action = env.action_space.sample()
            else:                            action = np.argmax(qvals_state);
        # Take action, get observation from the environment

        next_state, reward, done, info = env.step(action)
        score += reward # update episode score

        # add to memory, respecting memory buffer limit
        if len(replay_memory) > N:
            replay_memory.pop(0)
        replay_memory.append({"state":state,"action":action,"reward":reward\
                              ,"next_state":next_state,"done":done})
        # Update state
        state = next_state
        # Train the DNN using the samples from replay memory
        model=replay(replay_memory, B)
        time += 1   # update episode time

    # Decrease epsilon until we hit a target threshold
    if epsilon > epsilon_min:
        epsilon -= epsilon_step
    else:
        epsilon = epsilon*epsilon_decay # Decay at slower rate after threshold
```

```
        print('Episode number:',n)
        print("Total reward:", score)

        scores.append(score)    # store episode score
        if n>100:
            meanscore.append(np.mean(scores[-101:-1])) # store mean
            stdscore.append(np.std(scores[-101:-1]))    # store standard deviation

        # plot result for every 100 episodes
        if n % 100 == 0 and n>0:
            print(n)
            plt.figure(1)
            plt.plot(scores)
            plt.xlabel("Episode Number")
            plt.ylabel("Score")
            plt.figure(2)
            plt.plot(list(map(lambda x: x['loss'], loss)))
            plt.xlabel("Overall time step")
            plt.ylabel("Q-Network Loss")
            plt.figure(3)
            xaxis = np.arange(0, len(meanscore)) + 100
            plt.plot(xaxis, meanscore)
            plt.xlabel("Episode Number")
            plt.ylabel("Mean Score (over last 100 episodes)")
            plt.figure(4)
            plt.plot(xaxis, stdscore)
            plt.xlabel("Episode Number")
            plt.ylabel("Standard Deviation of Scores (over last 100 episodes)")
            plt.show()

env.close() # close the environment

# Save the results for comparison
np.savez('complex_act_fix.npz', scores, loss, meanscore, stdscore)
```