assign3

April 20, 2020

1 Problem 1 : Deep Q Learning : Preliminaries

1.0.1 a)

In the Walkins Q Learning algorithm, the Q(s,a) pairs are different and updating one of them doesn't affect the other which is not the case in Online Q Learning algorithm.

1.0.2 b)

Updating the policy plays a key role here. After an iteration, the samples must be taken from the updated policy so that it will learn correctly.

2 Problem 2 : Deep Q Learning : Programming

DQN and DDQN implementations of Mountain Car and Pong -> Couldn't train Pong completely because of the computational constraints. -> Learning curves for the performance of each model are shown. -> For the part (c) of this question, i have chosen the hyper-parameter as the replay buffer size -> For the part (d) of this question, I have done three seeds for the Mountain Car and couldn't do the same for pong because of the computational constraints. -> I am attaching the final checkpoints for all the models trained.

```
In [4]: import numpy as np
        import os,random
        import gym
        from collections import deque
        from tensorflow.keras.models import Model,Sequential
        from tensorflow.keras.layers import Dense,Input,Conv2D,Flatten
        from tensorflow.keras.optimizers import Adam
        import tensorflow.keras.backend as ks
        import matplotlib.pyplot as plt
        %matplotlib inline
```

```
In [18]: def plot(scores,n):
             x = []
             y = []
             y_best = []
             n_scores = []
```

```
            for i in range(len(scores)):
                if (i+1)%n == 0:
                    x.append((i//n)+1)
                    y.append(np.mean(n_scores))
                    y_best.append(np.max(n_scores))
                    n_scores = []
                n_scores.append(scores[i])

            return x,y,y_best
```

3 DQN For MountainCar

In [4]:
```
env = gym.make('MountainCar-v0')
state_size = env.observation_space.shape
action_size = env.action_space.n
print(state_size)
print(action_size)
```

```
(2,)
3
```

In [5]:
```
env.reset()
done = False
while not done:
    action = np.random.randint(action_size)
    _,reward,done,_ = env.step(action)
    print(reward)
```

```
-1.0
-1.0
-1.0
-1.0
-1.0
-1.0
-1.0
-1.0
-1.0
-1.0
-1.0
-1.0
-1.0
-1.0
-1.0
-1.0
-1.0
-1.0
-1.0
-1.0
```

```
-1.0
-1.0
-1.0
-1.0
-1.0
-1.0
-1.0
-1.0
-1.0
-1.0
-1.0
-1.0
-1.0
-1.0
-1.0
-1.0
-1.0
-1.0
-1.0
-1.0
-1.0
-1.0
-1.0
-1.0
-1.0
-1.0
-1.0
-1.0
-1.0
-1.0
-1.0
-1.0
-1.0
-1.0
-1.0
-1.0
-1.0
```

The reward is -1 for every step the car takes and a maximum of 200 steps allowed. So a score of -200 in the worst case is possible.

```
In [7]: class DQNAgent:
            def __init__(self,buffer_size,gamma,alpha,batch_size):
                self.state_size = state_size
                self.action_size = action_size
                self.gamma = gamma
                self.alpha = alpha
```

```
        self.batch_size = batch_size
        self.epsilon = 1
        self.epsilon_min = 0.05
        self.epsilon_decay = 0.95
        self.replay_buffer = deque(maxlen = buffer_size)
        self.model = self.build_model()
        self.target_model = self.build_model()
        self.target_model.set_weights(self.model.get_weights())

    def build_model(self):
        model = Sequential()
        model.add(Dense(40,activation='relu',input_shape=self.state_size))
        model.add(Dense(40,activation='relu'))
        model.add(Dense(self.action_size,activation='linear'))
        model.compile(loss='mse',optimizer=Adam(lr=self.alpha))
        return model

    def chose_action(self,state):
        self.epsilon = max(self.epsilon_min,self.epsilon)
        if np.random.rand(1) < self.epsilon:
            action = np.random.randint(self.action_size)
        else:
            action = np.argmax(self.model.predict(state)[0])
        return action

    def replay(self):
        if len(self.replay_buffer)<self.batch_size:
            return
        batch = random.sample(self.replay_buffer,self.batch_size)

        states = []
        next_states = []
        for item in batch:
            state,action,reward,next_state,done = item
            states.append(state)
            next_states.append(next_state)

        states = np.array(states).reshape(self.batch_size,2)
        next_states = np.array(next_states).reshape(self.batch_size,2)

        targets = self.model.predict(states)
        next_state_targets = self.target_model.predict(next_states)

        for i,item in enumerate(batch):
            state,action,reward,next_state,done = item
            if done:
                targets[i][action] = reward
            else:
```

```
                next_Q_max = max(next_state_targets[i])
                targets[i][action] = reward + self.gamma*next_Q_max

        self.model.fit(states,targets,epochs=1,verbose=0)

    def train(self,num_episodes,save_flag):
        print('------- Training -------')
        scores = []
        for episode in range(num_episodes):
            print('Episode',episode)
            state = env.reset().reshape(1,2)
            score = 0
            done = False
            while not done:
                action = self.chose_action(state)
                next_state,reward,done,_ = env.step(action)
                next_state = next_state.reshape(1,2)
                if next_state[0][0] >= 0.5:
                    reward = 10
                self.replay_buffer.append([state,action,reward,next_state,done])
                self.replay()
                state = next_state
                score += reward

            self.target_model.set_weights(self.model.get_weights())
            env.close()
            self.epsilon *= self.epsilon_decay
            scores.append(score)

            if save_flag and (episode+1)%10 == 0:
                self.save(episode+1)


        print('------- Finished Training -------')
        return scores

    def load(self,episode):
        self.model.load_weights('Checkpoints-MountainCar-DQN/'+str(episode)+'-dqn.h5')
        self.target_model.set_weights(self.model.get_weights())

    def save(self,episode):
        if not os.path.exists('Checkpoints-MountainCar-DQN'):
            os.mkdir('Checkpoints-MountainCar-DQN')
        self.model.save_weights('Checkpoints-MountainCar-DQN/'+str(episode)+'-dqn.h5')

    def test(self,num_episodes,render_flag):
        print('------- Testing -------')
        wins = 0
```

```
        scores = []
        for episode in range(num_episodes):
            state = env.reset().reshape(1,2)
            score = 0
            done = False
            while not done:
                action = np.argmax(self.model.predict(state)[0])
                next_state,reward,done,_ = env.step(action)
                if render_flag:
                    env.render()
                next_state = next_state.reshape(1,2)
                if next_state[0][0] >= 0.5:
                    reward = 10
                    wins += 1
                score += reward
                state = next_state
            scores.append(score)
            env.close()

        print('\tWon {} out of {} games'.format(wins,num_episodes))
        print('\tAverage Score : {}'.format(np.mean(scores)))
```

```
In [69]: agent_dqn = DQNAgent(20000,0.99,0.001,32)
```

```
In [5]: scores_dqn = agent_dqn.train(500,True)
```

```
WARNING:tensorflow:From /home/ak0808/.local/lib/python3.6/site-packages/tensorflow/python/ops/
Instructions for updating:
Call initializer instance with the dtype argument instead of passing it to the constructor
------- Training -------
Episode 0
Episode 1
Episode 2
Episode 3
Episode 4
Episode 5
Episode 6
Episode 7
Episode 8
Episode 9
Episode 10
Episode 11
Episode 12
Episode 13
Episode 14
Episode 15
Episode 16
```

```
Episode 497
Episode 498
Episode 499
------- Finished Training -------
```

```
In [6]: # Loading the final model from the folder. Just pass the number as parameter
        # Testing for 50 games.
        agent_dqn.load(500)
        agent_dqn.test(50,False)
```

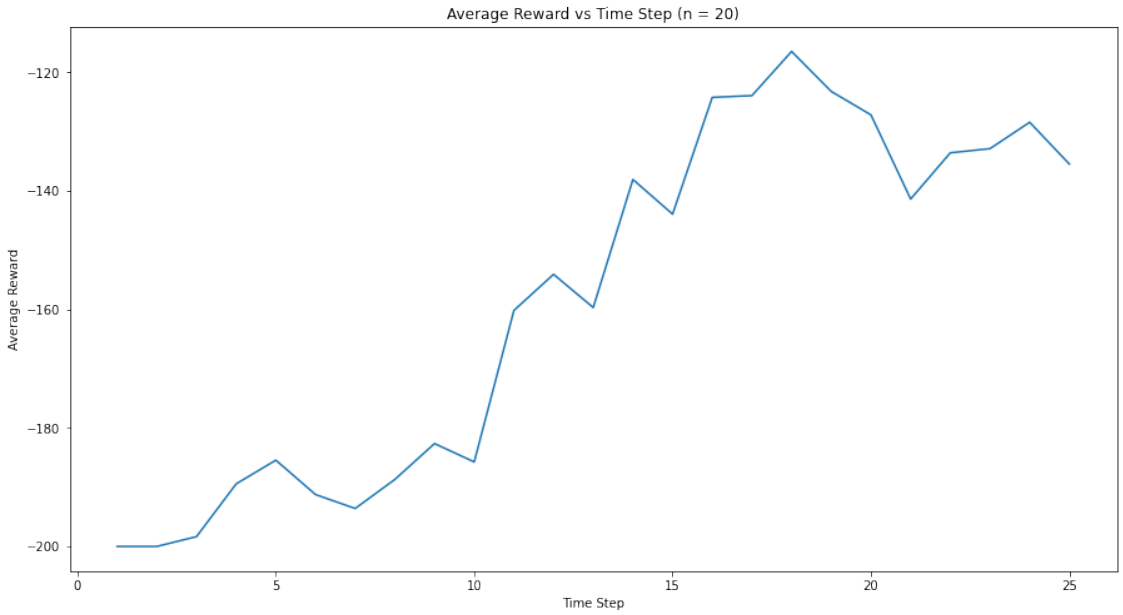
```
------- Testing -------
        Won 50 out of 50 games
        Average Score : -118.64
```

```
In [71]: # Loading the final model from the folder. Just pass the number as parameter
         # For seeing the agent in action.
         agent_dqn.load(500)
         agent_dqn.test(1,True)
```

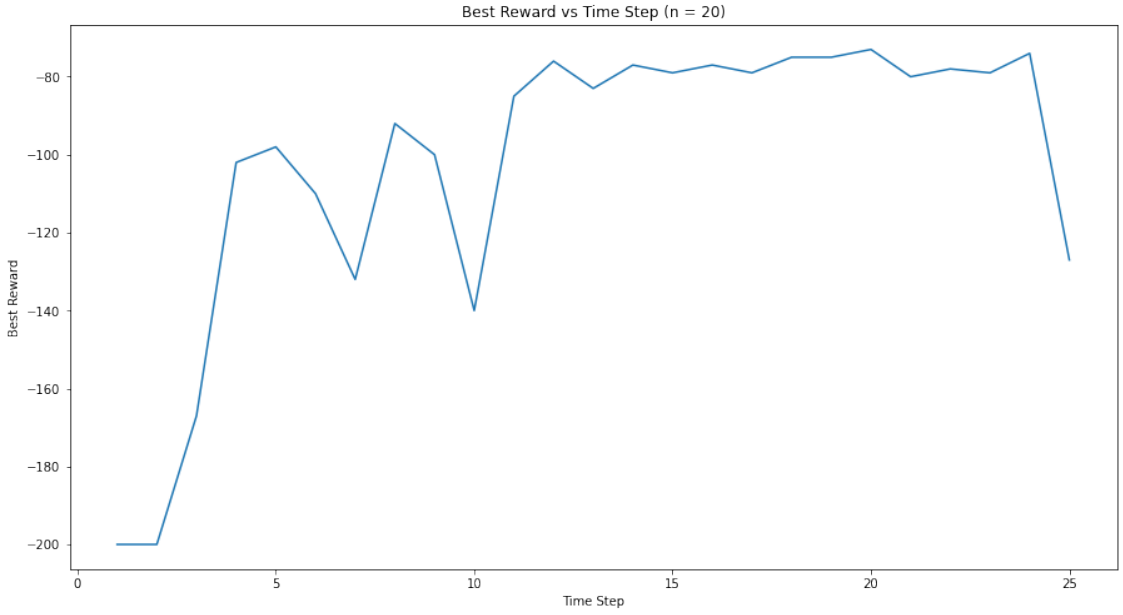
```
------- Testing -------
        Won 1 out of 1 games
        Average Score : -75.0
```

```
In [7]: x_dqn,y_dqn,y_best_dqn = plot(scores_dqn,20)
```

```
In [8]: plt.figure(figsize=(15,8))
        plt.plot(x_dqn,y_dqn)
        plt.title('Average Reward vs Time Step (n = 20)')
        plt.xlabel('Time Step')
        plt.ylabel('Average Reward')
        plt.show()
```




```
In [9]: plt.figure(figsize=(15,8))
        plt.plot(x_dqn,y_best_dqn)
        plt.title('Best Reward vs Time Step (n = 20)')
        plt.xlabel('Time Step')
        plt.ylabel('Best Reward')
        plt.show()
```



```
In [12]: agent1 = DQNAgent(40,0.99,0.001,32)
         scores_40 = agent1.train(500,False)

------- Training -------
Episode 0
Episode 1
Episode 2
Episode 3
Episode 4
Episode 5
Episode 6
Episode 7
Episode 8
Episode 9
Episode 10
Episode 11
Episode 12
Episode 13
Episode 14
Episode 15
Episode 16
Episode 17
Episode 18
Episode 19
Episode 20
Episode 21
Episode 22
Episode 23
Episode 24
Episode 25
Episode 26
Episode 27
Episode 28
Episode 29
Episode 30
Episode 31
Episode 32
Episode 33
Episode 34
Episode 35
Episode 36
Episode 37
Episode 38
Episode 39
Episode 40
Episode 41
Episode 42
Episode 43
```

```
Episode 476
Episode 477
Episode 478
Episode 479
Episode 480
Episode 481
Episode 482
Episode 483
Episode 484
Episode 485
Episode 486
Episode 487
Episode 488
Episode 489
Episode 490
Episode 491
Episode 492
Episode 493
Episode 494
Episode 495
Episode 496
Episode 497
Episode 498
Episode 499
------- Finished Training -------
```

```
In [13]: agent2 = DQNAgent(200,0.99,0.001,32)
         scores_200 = agent2.train(500,False)
```

```
------- Training -------
Episode 0
Episode 1
Episode 2
Episode 3
Episode 4
Episode 5
Episode 6
Episode 7
Episode 8
Episode 9
Episode 10
Episode 11
Episode 12
Episode 13
Episode 14
Episode 15
Episode 16
```

```
Episode 497
Episode 498
Episode 499
------- Finished Training -------
```

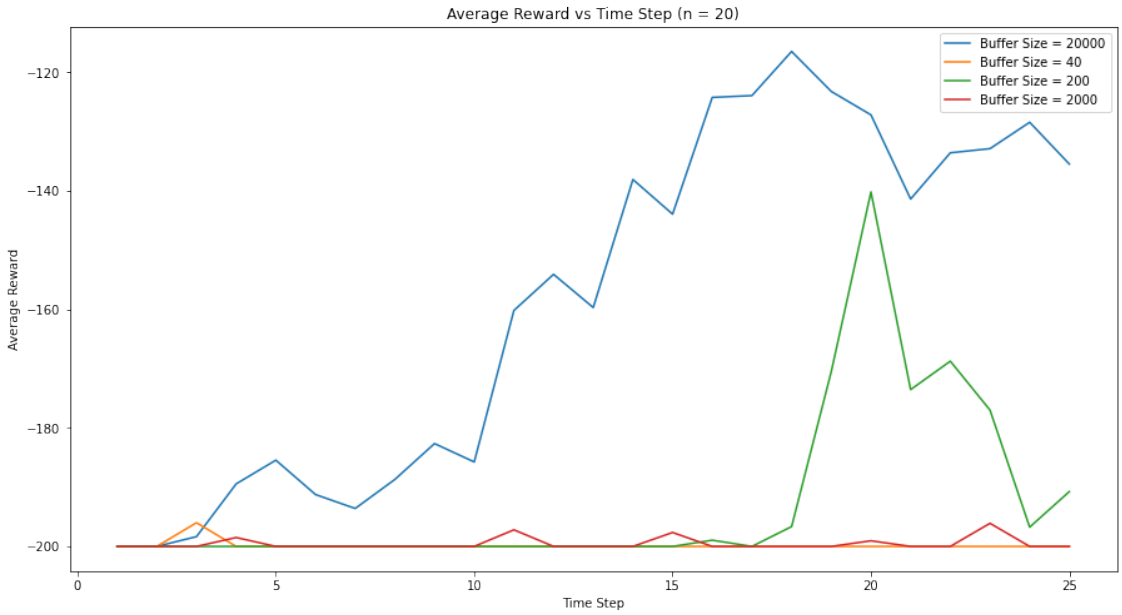
```
In [14]: agent3 = DQNAgent(2000,0.99,0.001,32)
         scores_2000 = agent3.train(500,False)
```

```
------- Training -------
Episode 0
Episode 1
Episode 2
Episode 3
Episode 4
Episode 5
Episode 6
Episode 7
Episode 8
Episode 9
Episode 10
Episode 11
Episode 12
Episode 13
Episode 14
Episode 15
Episode 16
Episode 17
Episode 18
Episode 19
Episode 20
Episode 21
Episode 22
Episode 23
Episode 24
Episode 25
Episode 26
Episode 27
Episode 28
Episode 29
Episode 30
Episode 31
Episode 32
Episode 33
Episode 34
Episode 35
Episode 36
Episode 37
```

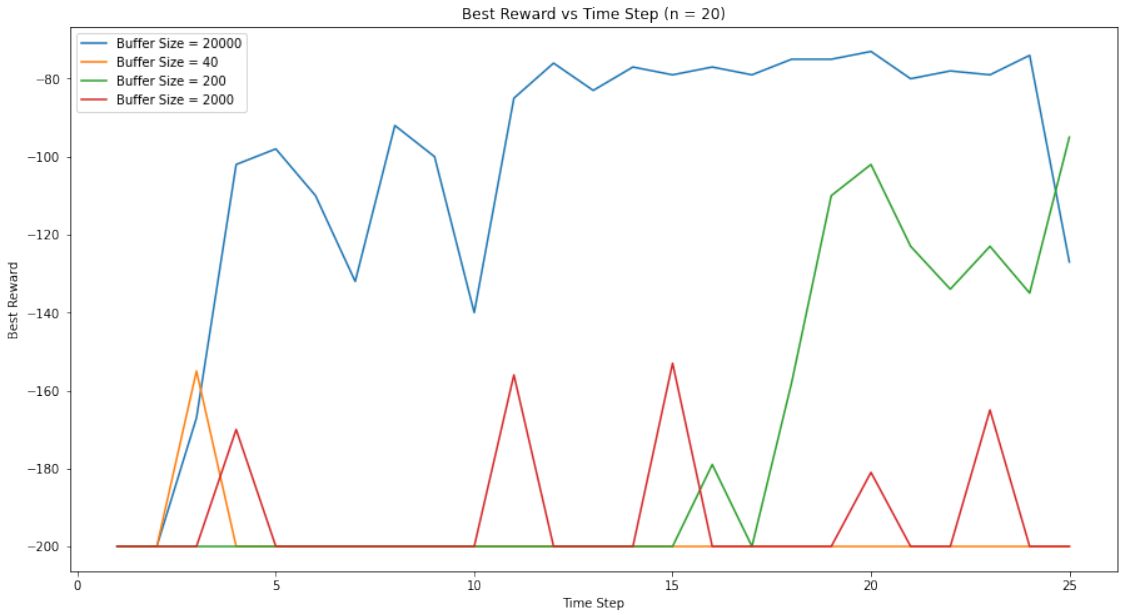
```
Episode 470
Episode 471
Episode 472
Episode 473
Episode 474
Episode 475
Episode 476
Episode 477
Episode 478
Episode 479
Episode 480
Episode 481
Episode 482
Episode 483
Episode 484
Episode 485
Episode 486
Episode 487
Episode 488
Episode 489
Episode 490
Episode 491
Episode 492
Episode 493
Episode 494
Episode 495
Episode 496
Episode 497
Episode 498
Episode 499
------- Finished Training -------
```

```
In [15]: x_40,y_40,y_best_40 = plot(scores_40,20)
         x_200,y_200,y_best_200 = plot(scores_200,20)
         x_2000,y_2000,y_best_2000 = plot(scores_2000,20)
```

```
In [16]: plt.figure(figsize=(15,8))
         plt.plot(x_dqn,y_dqn,label = 'Buffer Size = 20000')
         plt.plot(x_40,y_40,label = 'Buffer Size = 40')
         plt.plot(x_200,y_200,label = 'Buffer Size = 200')
         plt.plot(x_2000,y_2000,label = 'Buffer Size = 2000')
         plt.title('Average Reward vs Time Step (n = 20)')
         plt.xlabel('Time Step')
         plt.ylabel('Average Reward')
         plt.legend()
         plt.show()
```




```
In [17]: plt.figure(figsize=(15,8))
         plt.plot(x_dqn,y_best_dqn,label = 'Buffer Size = 20000')
         plt.plot(x_40,y_best_40,label = 'Buffer Size = 40')
         plt.plot(x_200,y_best_200,label = 'Buffer Size = 200')
         plt.plot(x_2000,y_best_2000,label = 'Buffer Size = 2000')
         plt.title('Best Reward vs Time Step (n = 20)')
         plt.xlabel('Time Step')
         plt.ylabel('Best Reward')
         plt.legend()
         plt.show()
```



4 DDQN For MountainCar

```
In [8]: class DDQNAgent:
            def __init__(self,buffer_size,gamma,alpha,batch_size):
                self.state_size = env.observation_space.shape
                self.action_size = env.action_space.n
                self.gamma = gamma
                self.alpha = alpha
                self.batch_size = batch_size
                self.epsilon = 1
                self.epsilon_min = 0.05
                self.epsilon_decay = 0.95
                self.replay_buffer = deque(maxlen = buffer_size)
                self.model = self.build_model()
                self.target_model = self.build_model()
                self.target_model.set_weights(self.model.get_weights())

            def build_model(self):
                model = Sequential()
                model.add(Dense(40,activation='relu',input_shape=self.state_size))
                model.add(Dense(40,activation='relu'))
                model.add(Dense(self.action_size,activation='linear'))
                model.compile(loss='mse',optimizer=Adam(lr=self.alpha))
                return model

            def chose_action(self,state):
```

```
        self.epsilon = max(self.epsilon_min,self.epsilon)
        if np.random.rand(1) < self.epsilon:
            action = np.random.randint(3)
        else:
            action = np.argmax(self.model.predict(state)[0])
        return action

    def replay(self):
        if len(self.replay_buffer)<self.batch_size:
            return
        batch = random.sample(self.replay_buffer,self.batch_size)

        states = []
        next_states = []
        for item in batch:
            state,action,reward,next_state,done = item
            states.append(state)
            next_states.append(next_state)

        states = np.array(states).reshape(self.batch_size,2)
        next_states = np.array(next_states).reshape(self.batch_size,2)

        targets = self.model.predict(states)
        targets_next = self.model.predict(next_states)
        next_state_targets = self.target_model.predict(next_states)

        for i,item in enumerate(batch):
            state,action,reward,next_state,done = item
            if done:
                targets[i][action] = reward
            else:
                a = np.argmax(targets_next[i])
                targets[i][action] = reward + self.gamma*next_state_targets[i][a]

        self.model.fit(states,targets,epochs=1,verbose=0)

    def train(self,num_episodes,save_flag):
        print('------- Training -------')
        scores = []
        for episode in range(num_episodes):
            print('Episode',episode)
            state = env.reset().reshape(1,2)
            score = 0
            done = False
            while not done:
                action = self.chose_action(state)
                next_state,reward,done,_ = env.step(action)
                next_state = next_state.reshape(1,2)
```

```
                if next_state[0][0] >= 0.5:
                    reward = 10
                self.replay_buffer.append([state,action,reward,next_state,done])
                self.replay()
                state = next_state
                score += reward

            self.target_model.set_weights(self.model.get_weights())
            env.close()
            self.epsilon *= self.epsilon_decay
            scores.append(score)

            if save_flag and (episode+1)%10 == 0:
                self.save(episode+1)


        print('------- Finished Training -------')
        return scores

    def load(self,episode):
        self.model.load_weights('Checkpoints-MountainCar-DDQN/'+str(episode)+'-ddqn.h5
        self.target_model.set_weights(self.model.get_weights())

    def save(self,episode):
        if not os.path.exists('Checkpoints-MountainCar-DDQN'):
            os.mkdir('Checkpoints-MountainCar-DDQN')
        self.model.save_weights('Checkpoints-MountainCar-DDQN/'+str(episode)+'-ddqn.h5

    def test(self,num_episodes,render_flag):
        print('------- Testing -------')
        wins = 0
        scores = []
        for episode in range(num_episodes):
            state = env.reset().reshape(1,2)
            score = 0
            done = False
            while not done:
                action = np.argmax(self.model.predict(state)[0])
                next_state,reward,done,_ = env.step(action)
                if render_flag:
                    env.render()
                next_state = next_state.reshape(1,2)
                if next_state[0][0] >= 0.5:
                    reward = 10
                    wins += 1
                score += reward
                state = next_state
            scores.append(score)
```

```
                    env.close()

                print('\tWon {} out of {} games'.format(wins,num_episodes))
                print('\tAverage Score : {}'.format(np.mean(scores)))
```

```
In [73]: agent_ddqn = DDQNAgent(20000,0.99,0.001,32)
```

```
In [19]: scores_ddqn = agent_ddqn.train(500,True)
```

```
------- Training -------
Episode 0
Episode 1
Episode 2
Episode 3
Episode 4
Episode 5
Episode 6
Episode 7
Episode 8
Episode 9
Episode 10
Episode 11
Episode 12
Episode 13
Episode 14
Episode 15
Episode 16
Episode 17
Episode 18
Episode 19
Episode 20
Episode 21
Episode 22
Episode 23
Episode 24
Episode 25
Episode 26
Episode 27
Episode 28
Episode 29
Episode 30
Episode 31
Episode 32
Episode 33
Episode 34
Episode 35
Episode 36
```

```
Episode 469
Episode 470
Episode 471
Episode 472
Episode 473
Episode 474
Episode 475
Episode 476
Episode 477
Episode 478
Episode 479
Episode 480
Episode 481
Episode 482
Episode 483
Episode 484
Episode 485
Episode 486
Episode 487
Episode 488
Episode 489
Episode 490
Episode 491
Episode 492
Episode 493
Episode 494
Episode 495
Episode 496
Episode 497
Episode 498
Episode 499
------- Finished Training -------
```

```
In [24]: # Loading the final model from the folder. Just pass the number as parameter
         # Testing for 50 games.
         agent_ddqn.load(500)
         agent_ddqn.test(50,False)
```

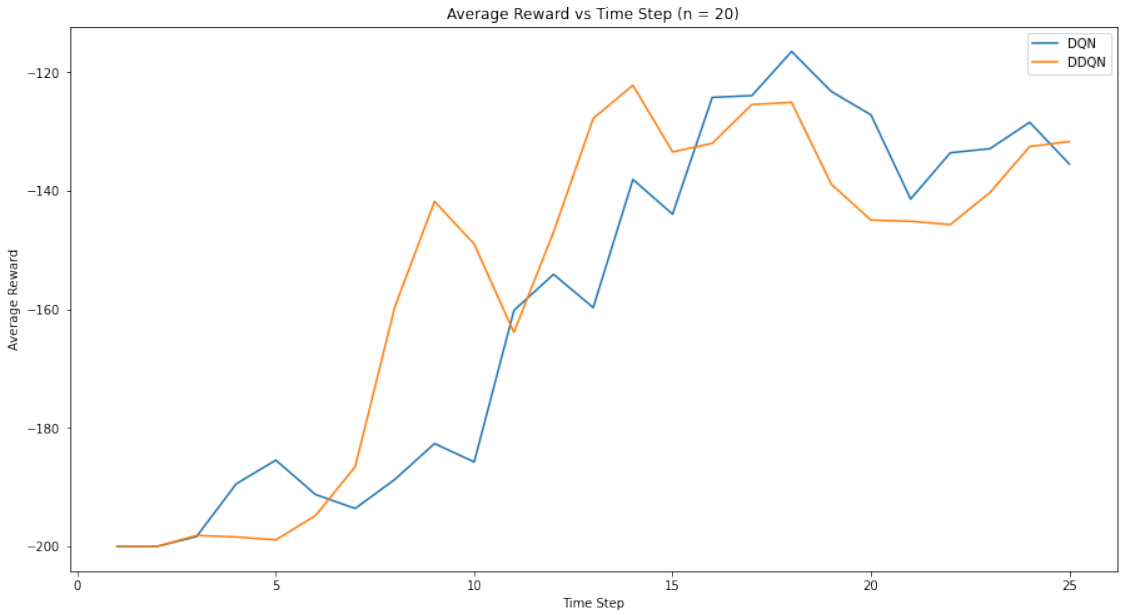
```
------- Testing -------
        Won 50 out of 50 games
        Average Score : -133.88
```

```
In [74]: # Loading the final model from the folder. Just pass the number as parameter
         # For seeing the agent in action.
         agent_ddqn.load(500)
         agent_ddqn.test(1,True)
```

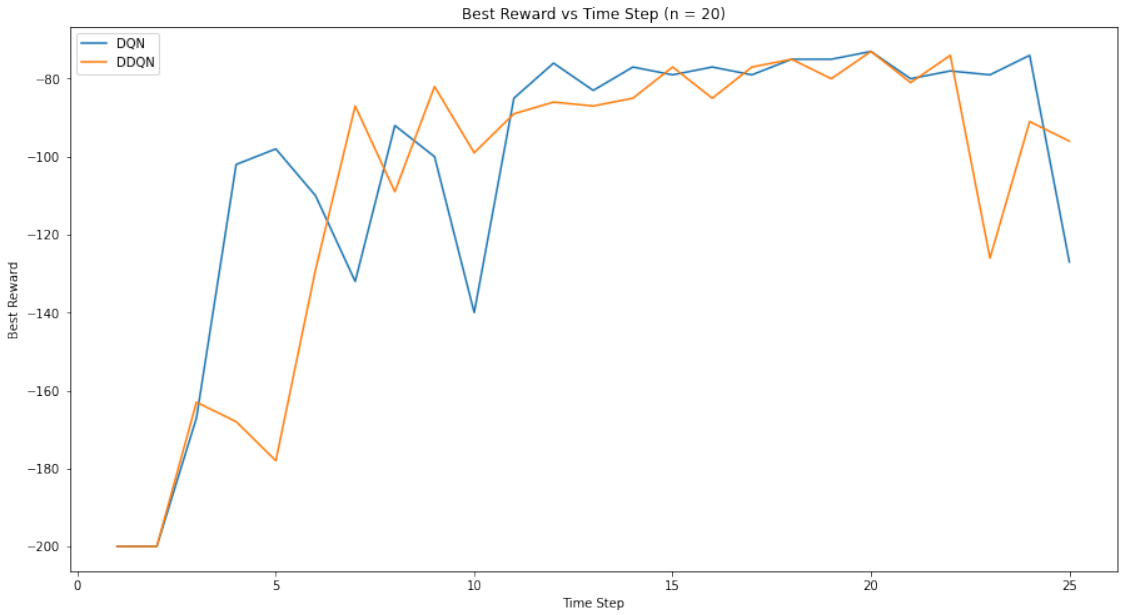
```
------- Testing -------
        Won 1 out of 1 games
        Average Score : -125.0
```

```
In [21]: x_ddqn,y_ddqn,y_best_ddqn = plot(scores_ddqn,20)
```

```
In [22]: plt.figure(figsize=(15,8))
         plt.plot(x_dqn,y_dqn,label = 'DQN')
         plt.plot(x_ddqn,y_ddqn,label = 'DDQN')
         plt.title('Average Reward vs Time Step (n = 20)')
         plt.xlabel('Time Step')
         plt.ylabel('Average Reward')
         plt.legend()
         plt.show()
```




```
In [23]: plt.figure(figsize=(15,8))
         plt.plot(x_dqn,y_best_dqn,label = 'DQN')
         plt.plot(x_ddqn,y_best_ddqn,label = 'DDQN')
         plt.title('Best Reward vs Time Step (n = 20)')
         plt.xlabel('Time Step')
         plt.ylabel('Best Reward')
         plt.legend()
         plt.show()
```

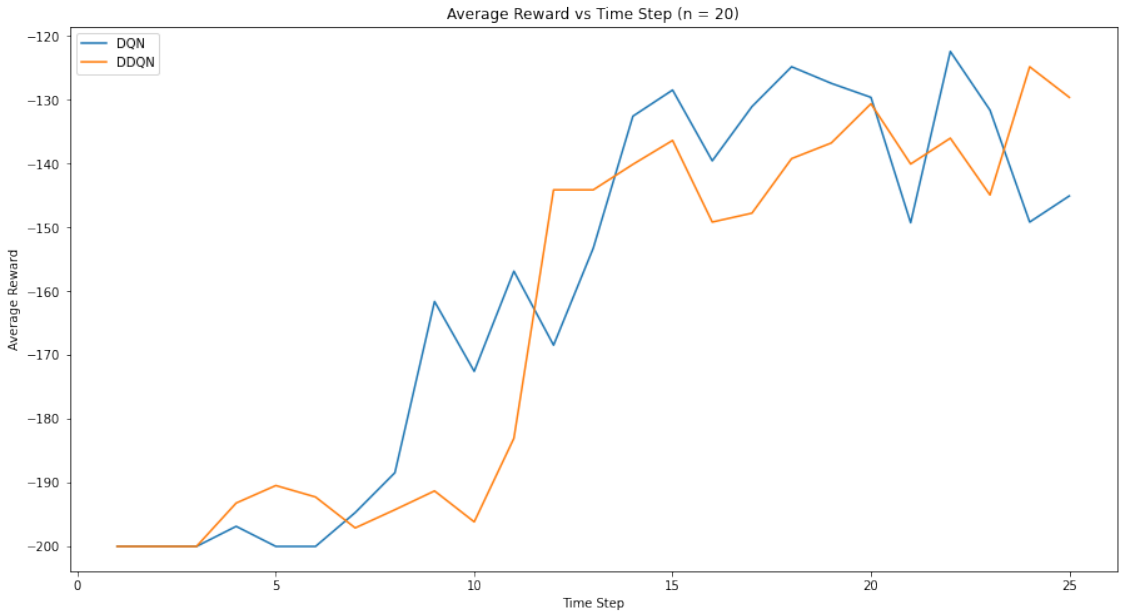



```
In [5]: env.seed(100)

Out[5]: [100]

In [10]: agent_dqn = DQNAgent(20000,0.99,0.001,32)
         agent_ddqn = DDQNAgent(20000,0.99,0.001,32)

In [11]: scores_dqn = agent_dqn.train(500,False)
         scores_ddqn = agent_ddqn.train(500,False)

------- Training -------
Episode 0
Episode 1
Episode 2
Episode 3
Episode 4
Episode 5
Episode 6
Episode 7
Episode 8
Episode 9
Episode 10
Episode 11
Episode 12
Episode 13
Episode 14
Episode 15
Episode 16
```

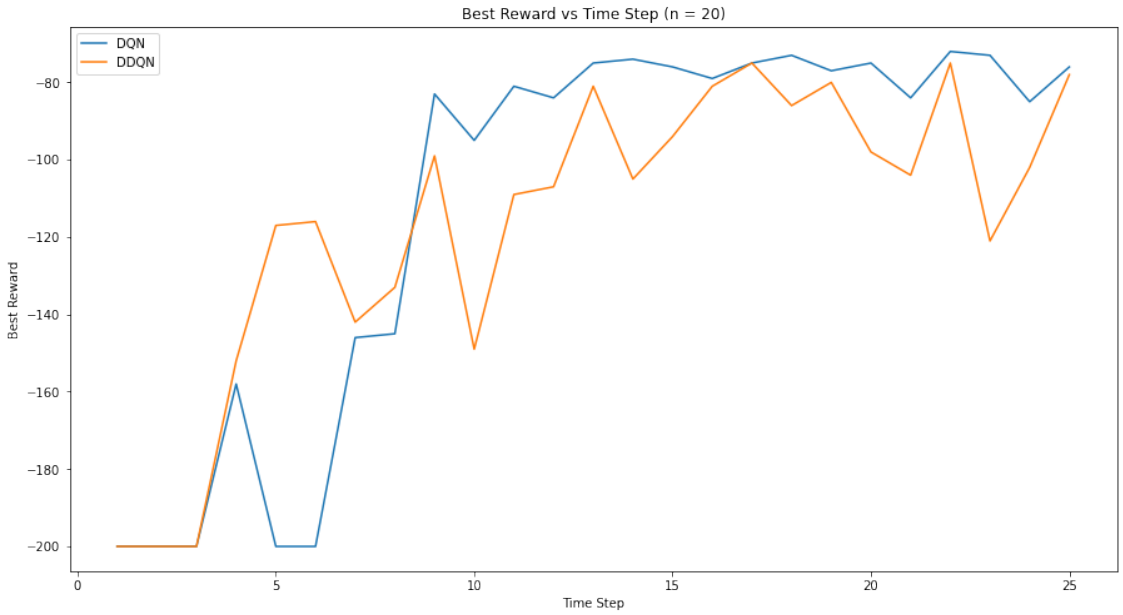
```
Episode 475
Episode 476
Episode 477
Episode 478
Episode 479
Episode 480
Episode 481
Episode 482
Episode 483
Episode 484
Episode 485
Episode 486
Episode 487
Episode 488
Episode 489
Episode 490
Episode 491
Episode 492
Episode 493
Episode 494
Episode 495
Episode 496
Episode 497
Episode 498
Episode 499
------- Finished Training -------
```

```
In [12]: x_dqn,y_dqn,y_best_dqn = plot(scores_dqn,20)
         x_ddqn,y_ddqn,y_best_ddqn = plot(scores_ddqn,20)
```

```
In [13]: plt.figure(figsize=(15,8))
         plt.plot(x_dqn,y_dqn,label = 'DQN')
         plt.plot(x_ddqn,y_ddqn,label = 'DDQN')
         plt.title('Average Reward vs Time Step (n = 20)')
         plt.xlabel('Time Step')
         plt.ylabel('Average Reward')
         plt.legend()
         plt.show()
```




```
In [14]: plt.figure(figsize=(15,8))
         plt.plot(x_dqn,y_best_dqn,label = 'DQN')
         plt.plot(x_ddqn,y_best_ddqn,label = 'DDQN')
         plt.title('Best Reward vs Time Step (n = 20)')
         plt.xlabel('Time Step')
         plt.ylabel('Best Reward')
         plt.legend()
         plt.show()
```

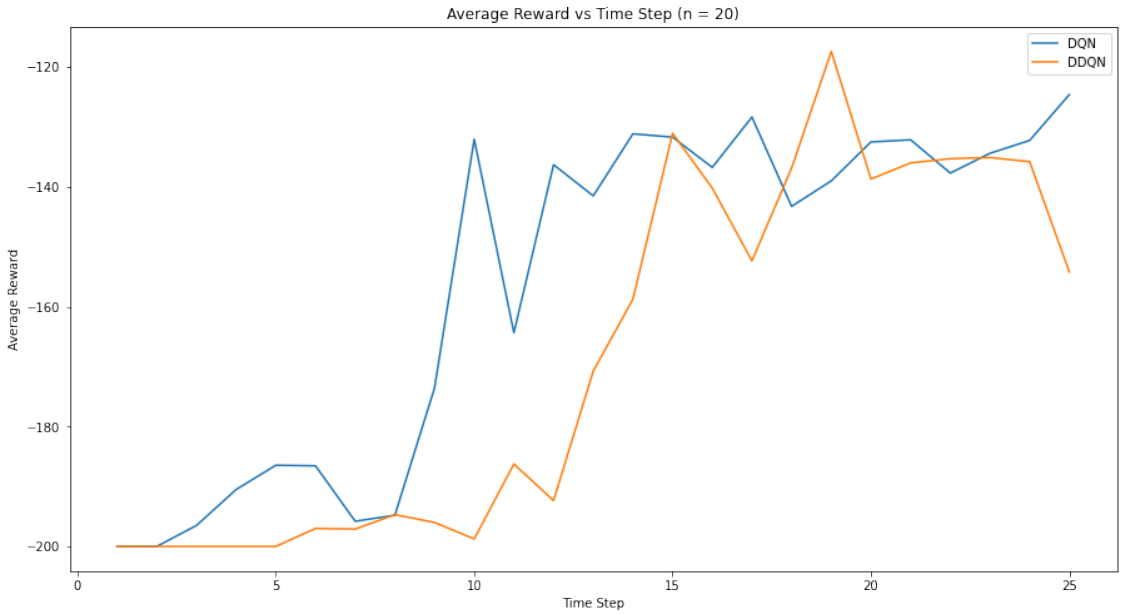


```
In [15]: env.seed(200)

Out[15]: [200]

In [16]: agent_dqn = DQNAgent(20000,0.99,0.001,32)
         agent_ddqn = DDQNAgent(20000,0.99,0.001,32)

In [17]: scores_dqn = agent_dqn.train(500,False)
         scores_ddqn = agent_ddqn.train(500,False)

------- Training -------
Episode 0
Episode 1
Episode 2
Episode 3
Episode 4
Episode 5
Episode 6
Episode 7
Episode 8
Episode 9
Episode 10
Episode 11
Episode 12
Episode 13
Episode 14
Episode 15
Episode 16
Episode 17
Episode 18
Episode 19
Episode 20
Episode 21
Episode 22
Episode 23
Episode 24
Episode 25
Episode 26
Episode 27
Episode 28
Episode 29
Episode 30
Episode 31
Episode 32
Episode 33
Episode 34
```

```
Episode 493
Episode 494
Episode 495
Episode 496
Episode 497
Episode 498
Episode 499
------- Finished Training -------
```

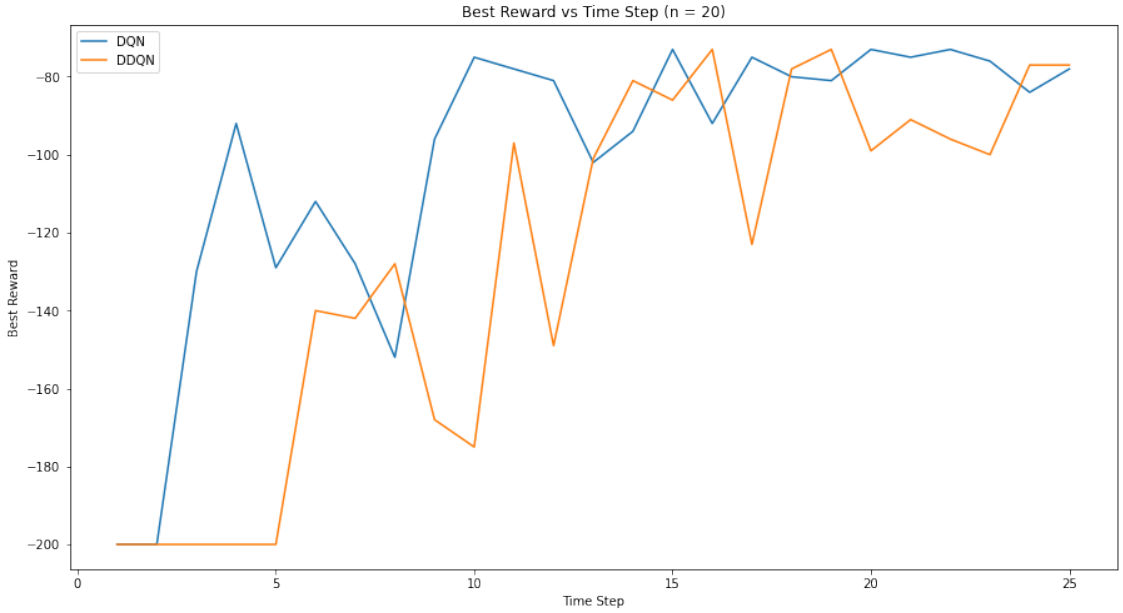
```
In [18]: x_dqn,y_dqn,y_best_dqn = plot(scores_dqn,20)
         x_ddqn,y_ddqn,y_best_ddqn = plot(scores_ddqn,20)
```

```
In [19]: plt.figure(figsize=(15,8))
         plt.plot(x_dqn,y_dqn,label = 'DQN')
         plt.plot(x_ddqn,y_ddqn,label = 'DDQN')
         plt.title('Average Reward vs Time Step (n = 20)')
         plt.xlabel('Time Step')
         plt.ylabel('Average Reward')
         plt.legend()
         plt.show()
```




```
In [20]: plt.figure(figsize=(15,8))
         plt.plot(x_dqn,y_best_dqn,label = 'DQN')
         plt.plot(x_ddqn,y_best_ddqn,label = 'DDQN')
         plt.title('Best Reward vs Time Step (n = 20)')
         plt.xlabel('Time Step')
```

```
plt.ylabel('Best Reward')
plt.legend()
plt.show()
```



5 DQN for Pong

```
In [5]: env = gym.make('Pong-v0')
        state_size = env.observation_space.shape
        action_size = env.action_space.n
        print(state_size)
        print(action_size)
```

```
(210, 160, 3)
6
```

```
In [6]: env.reset()
        done = False
        while not done:
            action = np.random.randint(action_size)
            _,reward,done,_ = env.step(action)
            print(reward)
```

```
0.0
0.0
0.0
0.0
```

```
0.0
0.0
0.0
0.0
0.0
0.0
0.0
0.0
0.0
0.0
0.0
0.0
0.0
0.0
0.0
0.0
0.0
0.0
0.0
0.0
0.0
0.0
0.0
0.0
0.0
0.0
0.0
-1.0
```

The reward is $+1$ for every point our player scores and -1 for every point the opponent scores. The game is for 21 points. So the maximum possible score is 21 and the least possible score is -21.

```
In [12]: class DQNAgent:
             def __init__(self,buffer_size,gamma,alpha,batch_size):
                 self.state_size = (80,80,4)
                 self.action_size = env.action_space.n

                 self.replay_buffer = deque(maxlen = buffer_size)
                 self.gamma = gamma
                 self.alpha = alpha
                 self.batch_size = batch_size
                 self.epsilon = 1
                 self.epsilon_min = 0.05
                 self.epsilon_decay = 0.95
                 self.model = self.build_model()
```

```
        self.target_model = self.build_model()
        self.target_model.set_weights(self.model.get_weights())

    def build_model(self):
        model = Sequential()
        model.add(Conv2D(32,8,padding='same',strides=4,activation='relu',input_shape=s
        model.add(Conv2D(64,4,padding='same',strides=2,activation='relu'))
        model.add(Conv2D(128,4,padding='same',strides=2,activation='relu'))
        model.add(Flatten())
        model.add(Dense(512,activation='relu'))
        model.add(Dense(256,activation='relu'))
        model.add(Dense(64,activation='relu'))
        model.add(Dense(self.action_size,activation='linear'))
        model.compile(loss='mse',optimizer=Adam(lr=self.alpha))
        return model

    def chose_action(self,state):
        self.epsilon = max(self.epsilon_min,self.epsilon)
        if np.random.rand(1) < self.epsilon:
            action = env.action_space.sample()
        else:
            state = np.expand_dims(state,axis=0)
            action = np.argmax(self.model.predict(state)[0])
        return action

    def replay(self):
        if len(self.replay_buffer)<self.batch_size:
            return
        batch = random.sample(self.replay_buffer,self.batch_size)

        states = []
        next_states = []
        for item in batch:
            state,action,reward,next_state,done = item
            states.append(state)
            next_states.append(next_state)

        states = np.array(states)
        next_states = np.array(next_states)
        targets = self.model.predict(states)
        next_state_targets = self.target_model.predict(next_states)

        for i,item in enumerate(batch):
            state,action,reward,next_state,done = item
            if done:
                targets[i][action] = reward
            else:
                next_Q_max = max(next_state_targets[i])
```

```
                targets[i][action] = reward + self.gamma*next_Q_max

        self.model.fit(states,targets,epochs=1,verbose=0)

    def stack_states(self,stack,state):
        state = cv2.cvtColor(state,cv2.COLOR_BGR2GRAY)
        state = cv2.resize(state,(80,80),interpolation = cv2.INTER_AREA)
        state = state/255
        stack.append(state)
        states = np.stack(stack,axis=2)
        return states


    def train(self,num_episodes,save_flag):
        print('------- Training -------')
        if save_flag:
            self.save('start')
        scores = []
        for episode in range(num_episodes):
            print('Episode',episode)
            state = env.reset()
            done = False
            states_stack = deque([np.zeros((80,80)) for _ in range(4)],maxlen = 4)
            states = self.stack_states(states_stack,state)
            prev_states_stack = deque([np.zeros((80,80)) for _ in range(4)],maxlen =
            score = 0
            step = 0
            while not done:
                action = self.chose_action(states)
                next_state,reward,done,_ = env.step(action)
                states = self.stack_states(states_stack,next_state)
                prev_states = self.stack_states(prev_states_stack,state)
                self.replay_buffer.append([prev_states,action,reward,states,done])
                self.replay()
                state = next_state
                score += reward
                step += 1

            self.target_model.set_weights(self.model.get_weights())
            env.close()
            self.epsilon *= self.epsilon_decay
            scores.append(score)
            if episode > 0 and save_flag and (episode+1)%10==0:
                self.save(episode)

        print('------- Finished Training -------')
        if save_flag:
            self.save('final')
```

```
                return scores


            def load(self,episode):
                self.model.load_weights('Checkpoints-Pong/'+str(episode)+'-dqn.h5')
                self.target_model.set_weights(self.model.get_weights())

            def save(self,episode):
                if not os.path.exists('Checkpoints-Pong'):
                    os.mkdir('Checkpoints-Pong')
                self.model.save_weights('Checkpoints-Pong/'+str(episode)+'-dqn.h5')

            def test(self,num_episodes,render_flag):
                print('------- Testing -------')
                scores = []
                for episode in range(num_episodes):
                    state = env.reset()
                    done = False
                    states_stack = deque([np.zeros((80,80)) for _ in range(4)],maxlen = 4)
                    states = self.stack_states(states_stack,state)
                    score = 0
                    while not done:
                        temp = np.expand_dims(states,axis=0)
                        action = np.argmax(self.model.predict(temp)[0])
                        next_state,reward,done,_ = env.step(action)
                        states = self.stack_states(states_stack,next_state)
                        if render_flag:
                            env.render()
                        score += reward

                    scores.append(score)
                    env.close()

                print('\tAverage Score : {} over {} games.'.format(np.mean(scores),num_episode
                return np.mean(scores)
```

In [14]:
```
agent = DQNAgent(50000,0.99,1e-5,64)
```

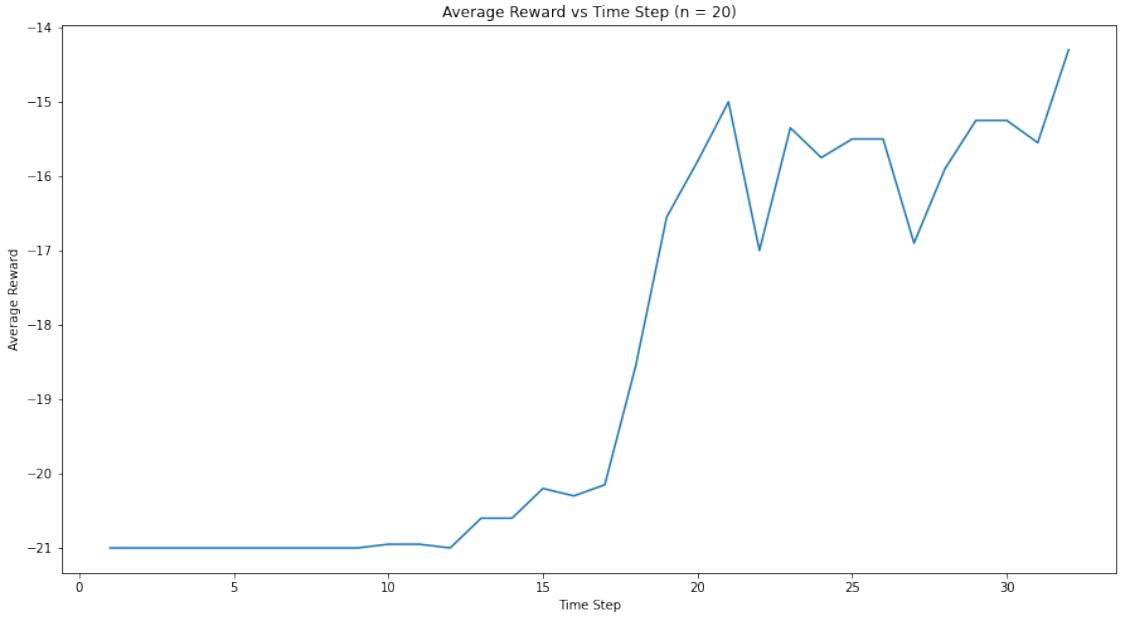
In [ ]:
```
scores = agent.train(2000,True)
```

In [ ]:
```
# Loading the final model from the folder. Just pass the number as parameter
# Testing for 50 games.
agent.load(700)
agent.test(50,False)
```

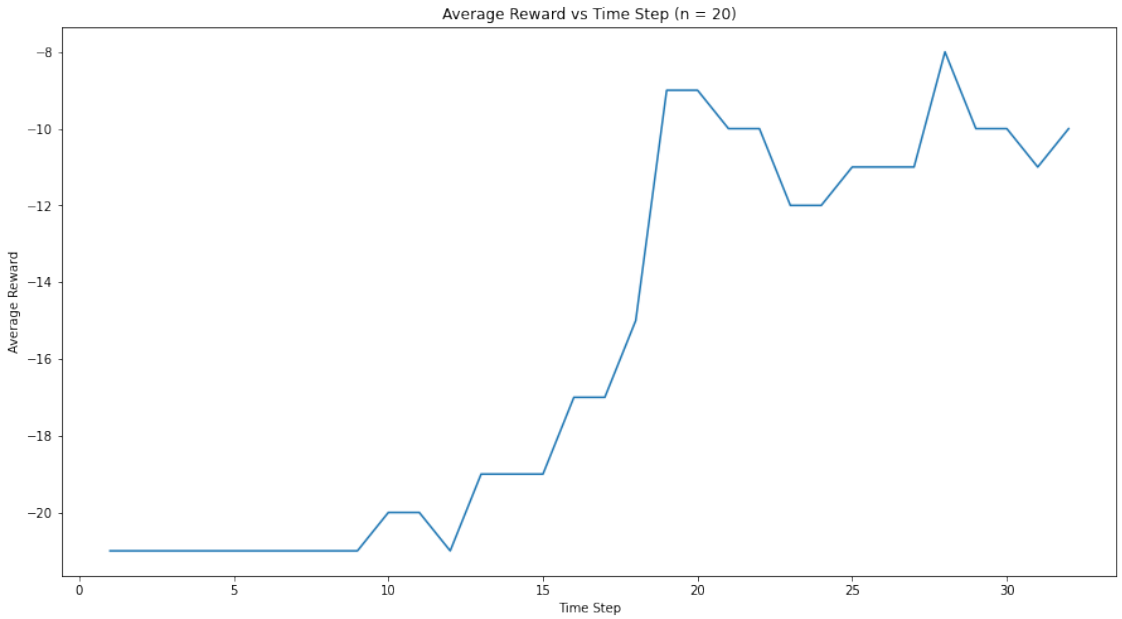
In [ ]:
```
# Loading the last model from the folder. Just pass the number as parameter
# For seeing the agent in action.
agent.load(700)
agent.test(1,True)
```

In [22]:
```
x_dqn,y_dqn,y_best_dqn = plot(scores,20)
```

In [23]:
```
plt.figure(figsize=(15,8))
plt.plot(x_dqn,y_dqn)
plt.title('Average Reward vs Time Step (n = 20)')
plt.xlabel('Time Step')
plt.ylabel('Average Reward')
plt.show()
```



In [24]:
```
plt.figure(figsize=(15,8))
plt.plot(x_dqn,y_best_dqn)
plt.title('Best Reward vs Time Step (n = 20)')
plt.xlabel('Time Step')
plt.ylabel('Best Reward')
plt.show()
```



6 Problem 3 : Policy Gradient

PG implementations of Cartpole and Lunar Lander -> PGAgent1 is an agent with reward to go and advantage normalization functionality. -> PGAgent2 is an agent without reward to go and advantage normalization functionality. -> Learning curves for the performance of each model are shown. -> I am attaching the final checkpoints for all the models trained.

```
In [6]: class PGAgent1:
            def __init__(self,state_size,action_size,gamma,alpha,n_hidden_1,n_hidden_2,name):
                self.state_size = state_size
                self.action_size = action_size
                self.gamma = gamma
                self.alpha = alpha
                self.name = name
                self.state_memory = []
                self.action_memory = []
                self.reward_memory = []
                self.baseline = []

                self.model,self.predict = self.build_model(n_hidden_1,n_hidden_2)

            def build_model(self,n_hidden_1,n_hidden_2):
                input_ = Input(shape=self.state_size)
                advantages = Input(shape=[1])
                l1 = Dense(n_hidden_1,activation='relu')(input_)
                l2 = Dense(n_hidden_2,activation='relu')(l1)
```

```
        out = Dense(self.action_size,activation='softmax')(l2)

        def custom_loss(y_true,y_pred):
            prob = ks.clip(y_pred,1e-8,1-1e-8)
            log_lik = y_true*ks.log(prob)
            return ks.sum(-log_lik*advantages)

        model = Model(inputs=[input_,advantages],outputs=[out])
        model.compile(loss=custom_loss,optimizer=Adam(lr=self.alpha))

        predict = Model(inputs=[input_],outputs=[out])
        return model,predict

    def chose_action(self,state):
        state = np.expand_dims(state,axis=0)
        prob = self.predict.predict(state)[0]
        action = np.random.choice(np.arange(self.action_size),p=prob)
        return action

    def replay(self):
        state_memory = np.array(self.state_memory)
        action_memory = np.array(self.action_memory)
        reward_memory = np.array(self.reward_memory)

        actions = np.zeros((len(action_memory),self.action_size))
        actions[np.arange(len(action_memory)),action_memory] = 1

        reward_to_go = np.zeros_like(reward_memory)
        for t in range(len(reward_memory)):
            G_sum = 0
            discount = 1
            for tt in range(t,len(reward_memory)):
                G_sum += reward_memory[tt]*discount
                discount *= self.gamma
            reward_to_go[t] = G_sum

        self.baseline.append(reward_to_go[0])
        advantage = reward_to_go - np.mean(self.baseline)
        mean = np.mean(advantage)
        std = np.std(advantage) if np.std(advantage)>0 else 1
        advantage = (advantage-mean)/std
        self.model.train_on_batch([state_memory,advantage],actions)

        self.state_memory = []
        self.action_memory = []
        self.reward_memory = []

    def train(self,num_episodes,save_flag):
```

```
        print('------- Training -------')
        scores = []
        for episode in range(num_episodes):
            print('Episode',episode)
            state = env.reset()
            done = False
            score = 0
            while not done:
                action = self.chose_action(state)
                next_state,reward,done,_ = env.step(action)
                self.state_memory.append(state)
                self.action_memory.append(action)
                self.reward_memory.append(reward)
                state = next_state
                score += reward

            env.close()
            scores.append(score)
            self.replay()
            if save_flag and (episode+1)%50 == 0:
                self.save(episode+1)

        print('------- Finished Training -------')
        return scores

    def load(self,episode):
        self.model.load_weights('Checkpoints-'+self.name+'-PG-1/'+str(episode)+'-pg.h5

    def save(self,episode):
        if not os.path.exists('Checkpoints-'+self.name+'-PG-1'):
            os.mkdir('Checkpoints-'+self.name+'-PG-1')
        self.model.save_weights('Checkpoints-'+self.name+'-PG-1/'+str(episode)+'-pg.h5

    def test(self,num_episodes,render_flag):
        print('------- Testing -------')
        scores = []
        for episode in range(num_episodes):
            state = env.reset()
            score = 0
            done = False
            while not done:
                action = self.chose_action(state)
                next_state,reward,done,_ = env.step(action)
                if render_flag:
                    env.render()
                score += reward
                state = next_state
            scores.append(score)
```

```
                env.close()

            print('\tAverage Score : {}'.format(np.mean(scores)))
```

```
In [7]: class PGAgent2:
            def __init__(self,state_size,action_size,gamma,alpha,n_hidden_1,n_hidden_2,name):
                self.state_size = state_size
                self.action_size = action_size
                self.gamma = gamma
                self.alpha = alpha
                self.name = name
                self.state_memory = []
                self.action_memory = []
                self.reward_memory = []

                self.model,self.predict = self.build_model(n_hidden_1,n_hidden_2)

            def build_model(self,n_hidden_1,n_hidden_2):
                input_ = Input(shape=self.state_size)
                advantages = Input(shape=[1])
                l1 = Dense(n_hidden_1,activation='relu')(input_)
                l2 = Dense(n_hidden_2,activation='relu')(l1)
                out = Dense(self.action_size,activation='softmax')(l2)

                def custom_loss(y_true,y_pred):
                    prob = ks.clip(y_pred,1e-8,1-1e-8)
                    log_lik = y_true*ks.log(prob)
                    return ks.sum(-log_lik*advantages)

                model = Model(inputs=[input_,advantages],outputs=[out])
                model.compile(loss=custom_loss,optimizer=Adam(lr=self.alpha))

                predict = Model(inputs=[input_],outputs=[out])
                return model,predict

            def chose_action(self,state):
                state = np.expand_dims(state,axis=0)
                prob = self.predict.predict(state)[0]
                action = np.random.choice(np.arange(self.action_size),p=prob)
                return action

            def replay(self):
                state_memory = np.array(self.state_memory)
                action_memory = np.array(self.action_memory)
                reward_memory = np.array(self.reward_memory)

                actions = np.zeros((len(action_memory),self.action_size))
```

```
        actions[np.arange(len(action_memory)),action_memory] = 1

        advantage = np.zeros_like(reward_memory)
        G_sum = 0
        discount = 1
        for t in range(len(reward_memory)):
            G_sum += reward_memory[t]*discount
            discount *= self.gamma

        advantage.fill(G_sum)
        self.model.train_on_batch([state_memory,advantage],actions)

        self.state_memory = []
        self.action_memory = []
        self.reward_memory = []

    def train(self,num_episodes,save_flag):
        print('------- Training -------')
        scores = []
        for episode in range(num_episodes):
            print('Episode',episode)
            state = env.reset()
            done = False
            score = 0
            while not done:
                action = self.chose_action(state)
                next_state,reward,done,_ = env.step(action)
                self.state_memory.append(state)
                self.action_memory.append(action)
                self.reward_memory.append(reward)
                state = next_state
                score += reward

            env.close()
            scores.append(score)
            self.replay()
            if save_flag and (episode+1)%50 == 0:
                self.save(episode+1)

        print('------- Finished Training -------')
        return scores

    def load(self,episode):
        self.model.load_weights('Checkpoints-'+self.name+'-PG-2/'+str(episode)+'-pg.h5

    def save(self,episode):
        if not os.path.exists('Checkpoints-'+self.name+'-PG-2'):
            os.mkdir('Checkpoints-'+self.name+'-PG-2')
```

```
            self.model.save_weights('Checkpoints-'+self.name+'-PG-2/'+str(episode)+'-pg.h5

        def test(self,num_episodes,render_flag):
            print('------- Testing -------')
            scores = []
            for episode in range(num_episodes):
                state = env.reset()
                score = 0
                done = False
                while not done:
                    action = self.chose_action(state)
                    next_state,reward,done,_ = env.step(action)
                    if render_flag:
                        env.render()
                    score += reward
                    state = next_state
                scores.append(score)
                env.close()

            print('\tAverage Score : {}'.format(np.mean(scores)))
```

7 Policy Gradients : Cartpole

```
In [6]: env = gym.make('CartPole-v1')
        state_size = env.observation_space.shape
        action_size = env.action_space.n
        print(state_size)
        print(action_size)
```

```
(4,)
2
```

```
In [7]: env.reset()
        done = False
        while not done:
            action = np.random.randint(action_size)
            _,reward,done,_ = env.step(action)
            print(reward)
```

```
1.0
1.0
1.0
1.0
1.0
1.0
1.0
```

```
1.0
1.0
1.0
1.0
```

The reward is +1 for every step it survive and a maximum of 500 steps allowed. So a score of 500 in the best case is possible.

```
In [115]: agent_cartpole_1 = PGAgent1(state_size,action_size,0.99,0.001,20,20,'Cartpole')

In [116]: scores_cartpole_1 = agent_cartpole_1.train(2000,True)

------- Training -------
Episode 0
Episode 1
Episode 2
Episode 3
Episode 4
Episode 5
Episode 6
Episode 7
Episode 8
Episode 9
Episode 10
Episode 11
Episode 12
Episode 13
Episode 14
Episode 15
Episode 16
Episode 17
Episode 18
Episode 19
Episode 20
Episode 21
Episode 22
Episode 23
Episode 24
Episode 25
Episode 26
Episode 27
Episode 28
Episode 29
Episode 30
Episode 31
Episode 32
Episode 33
Episode 34
```

```
In [119]: # Loading the final model from the folder. Just pass the number as parameter
          # Testing for 50 games.
          agent_cartpole_1.load(2000)
          agent_cartpole_1.test(50,False)

------- Testing -------
        Average Score : 496.6


In [120]: # Loading the final model from the folder. Just pass the number as parameter
          # For seeing the agent in action.
          agent_cartpole_1.load(2000)
          agent_cartpole_1.test(1,True)

------- Testing -------
        Average Score : 500.0


In [117]: agent_cartpole_2 = PGAgent2(state_size,action_size,0.99,0.001,20,20)

In [118]: scores_cartpole_2 = agent_cartpole_2.train(2000,True)

------- Training -------
Episode 0
Episode 1
Episode 2
Episode 3
Episode 4
Episode 5
Episode 6
Episode 7
Episode 8
Episode 9
Episode 10
Episode 11
Episode 12
Episode 13
Episode 14
Episode 15
Episode 16
Episode 17
Episode 18
Episode 19
Episode 20
Episode 21
Episode 22
Episode 23
Episode 24
Episode 25
```

```
Episode 1994
Episode 1995
Episode 1996
Episode 1997
Episode 1998
Episode 1999
------- Finished Training -------
```

```
In [121]: # Loading the final model from the folder. Just pass the number as parameter
          # Testing for 50 games.
          agent_cartpole_2.load(2000)
          agent_cartpole_2.test(50,False)
```

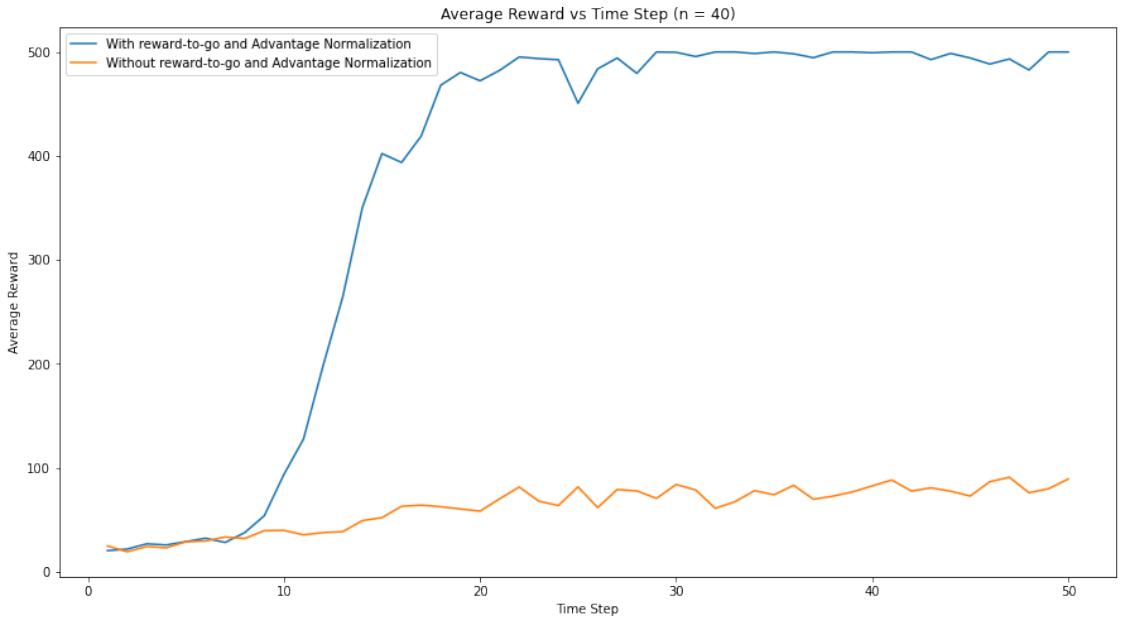
```
------- Testing -------
        Average Score : 92.74
```

```
In [122]: # Loading the final model from the folder. Just pass the number as parameter
          # For seeing the agent in action.
          agent_cartpole_2.load(2000)
          agent_cartpole_2.test(1,True)
```

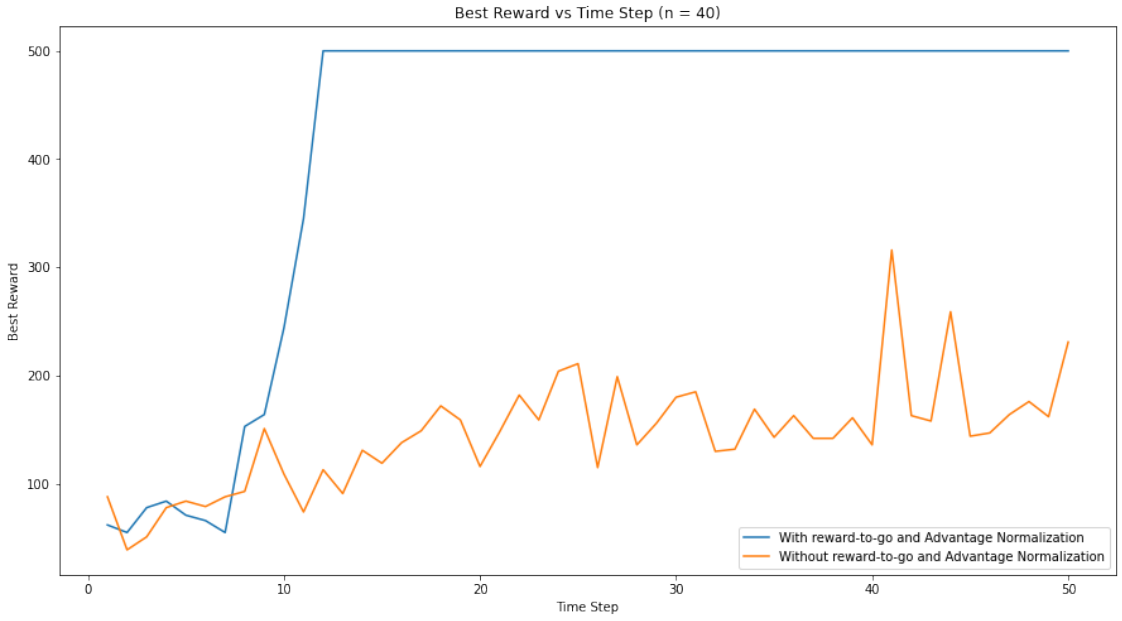
```
------- Testing -------
        Average Score : 48.0
```

```
In [125]: x_1,y_1,y_best_1 = plot(scores_cartpole_1,40)
          x_2,y_2,y_best_2 = plot(scores_cartpole_2,40)
```

```
In [128]: plt.figure(figsize=(15,8))
          plt.plot(x_1,y_1,label = 'With reward-to-go and Advantage Normalization')
          plt.plot(x_2,y_2,label = 'Without reward-to-go and Advantage Normalization')
          plt.title('Average Reward vs Time Step (n = 40)')
          plt.xlabel('Time Step')
          plt.ylabel('Average Reward')
          plt.legend()
          plt.show()
```




```
In [129]: plt.figure(figsize=(15,8))
          plt.plot(x_1,y_best_1,label = 'With reward-to-go and Advantage Normalization')
          plt.plot(x_2,y_best_2,label = 'Without reward-to-go and Advantage Normalization')
          plt.title('Best Reward vs Time Step (n = 40)')
          plt.xlabel('Time Step')
          plt.ylabel('Best Reward')
          plt.legend()
          plt.show()
```



8 Policy Gradients : Lunar Lander

```
In [10]: env = gym.make('LunarLander-v2')
         state_size = env.observation_space.shape
         action_size = env.action_space.n
         print(state_size)
         print(action_size)
```

```
(8,)
4
```

```
/home/ak0808/.local/lib/python3.6/site-packages/gym/logger.py:30: UserWarning: WARN: Box bound
  warnings.warn(colorize('%s: %s'%('WARN', msg % args), 'yellow'))
```

```
In [9]: env.reset()
        done = False
        while not done:
            action = np.random.randint(action_size)
            _,reward,done,_ = env.step(action)
            print(reward)
```

```
-1.3158848304880735
-0.8776420000802136
-0.7219433525906414
-1.025772608351133
-0.41756387412880147
-1.4164355061576817
-0.9243878245015935
4.358627135470226
-1.4778102122598102
3.4778008838608914
-1.0402053891412493
-0.9966526161857132
-1.689184755050319
2.8010050054813123
1.1716091003363658
4.848020466855201
2.7158789888046444
-0.48841253410534935
-0.9959264615978611
-0.2122627352530617
-0.6878245163350414
0.001738509949661876
```

```
0.11798076731656806
2.5798902801020462
-0.30370451234264806
0.4072344658651741
3.4881929367722764
1.8994044359671818
-1.548762262854466
-1.3526330217652276
-1.1516255935250672
-0.6329273792323409
-0.723267234629559
-0.8434801586487584
5.068743549302053
-1.1371909731680887
4.514922977142118
-1.144754071437659
-1.239243165191482
-1.194188257895404
4.413280016602113
-1.0859655106507375
-1.0546948032558408
-1.1385721346809408
-0.8600861128767292
-0.9403558029806891
-1.0686974077269429
-1.1746473078914335
4.767323117230927
-1.16852204279283
4.6472246807704325
3.0190737806123424
-1.1933095733323842
-1.0900902810909088
-0.9461492027955092
-1.0826809586441744
-1.0412823770128614
-1.2669816774775302
-1.3266869209831782
-1.4487605883404342
-1.8757962209901666
-1.8538902518319549
-1.9666675720722242
-2.3818531396893547
6.831307369588388
8.408620959576933
-100
```

Reward for moving from the top of the screen to landing pad and zero speed is about 100..140

points. If lander moves away from landing pad it loses reward back. Episode finishes if the lander crashes or comes to rest, receiving additional -100 or $+100$ points. Each leg ground contact is $+10$. Firing main engine is -0.3 points each frame. Solved is 200 points.

```
In [9]: agent_lunarlander_1 = PGAgent1(state_size,action_size,0.99,0.001,50,50,'LunarLander')

In [10]: scores_lunarlander_1 = agent_lunarlander_1.train(5000,True)

------- Training -------
Episode 0
WARNING:tensorflow:From /home/ak0808/.local/lib/python3.6/site-packages/tensorflow/python/ops/
Instructions for updating:
Use tf.where in 2.0, which has the same broadcast rule as np.where
Episode 1
Episode 2
Episode 3
Episode 4
Episode 5
Episode 6
Episode 7
Episode 8
Episode 9
Episode 10
Episode 11
Episode 12
Episode 13
Episode 14
Episode 15
Episode 16
Episode 17
Episode 18
Episode 19
Episode 20
Episode 21
Episode 22
Episode 23
Episode 24
Episode 25
Episode 26
Episode 27
Episode 28
Episode 29
Episode 30
Episode 31
Episode 32
Episode 33
Episode 34
Episode 35
```

```
Episode 4980
Episode 4981
Episode 4982
Episode 4983
Episode 4984
Episode 4985
Episode 4986
Episode 4987
Episode 4988
Episode 4989
Episode 4990
Episode 4991
Episode 4992
Episode 4993
Episode 4994
Episode 4995
Episode 4996
Episode 4997
Episode 4998
Episode 4999
------- Finished Training -------
```

```
In [11]: # Loading the final model from the folder. Just pass the number as parameter
         # Testing for 50 games.
         agent_lunarlander_1.load(5000)
         agent_lunarlander_1.test(50,False)
```

```
------- Testing -------
        Average Score : 106.11524924179581
```

```
In [12]: # Loading the final model from the folder. Just pass the number as parameter
         # For seeing the agent in action.
         agent_lunarlander_1.load(5000)
         agent_lunarlander_1.test(1,True)
```

```
------- Testing -------
        Average Score : 111.36902972119223
```

```
In [13]: agent_lunarlander_2 = PGAgent2(state_size,action_size,0.99,0.001,50,50,'LunarLander')
```

```
In [14]: scores_lunarlander_2 = agent_lunarlander_2.train(5000,True)
```

```
------- Training -------
Episode 0
Episode 1
Episode 2
```

```
Episode 4995
Episode 4996
Episode 4997
Episode 4998
Episode 4999
------- Finished Training -------
```

```
In [15]: # Loading the final model from the folder. Just pass the number as parameter
         # Testing for 50 games.
         agent_lunarlander_2.load(5000)
         agent_lunarlander_2.test(50,False)
```

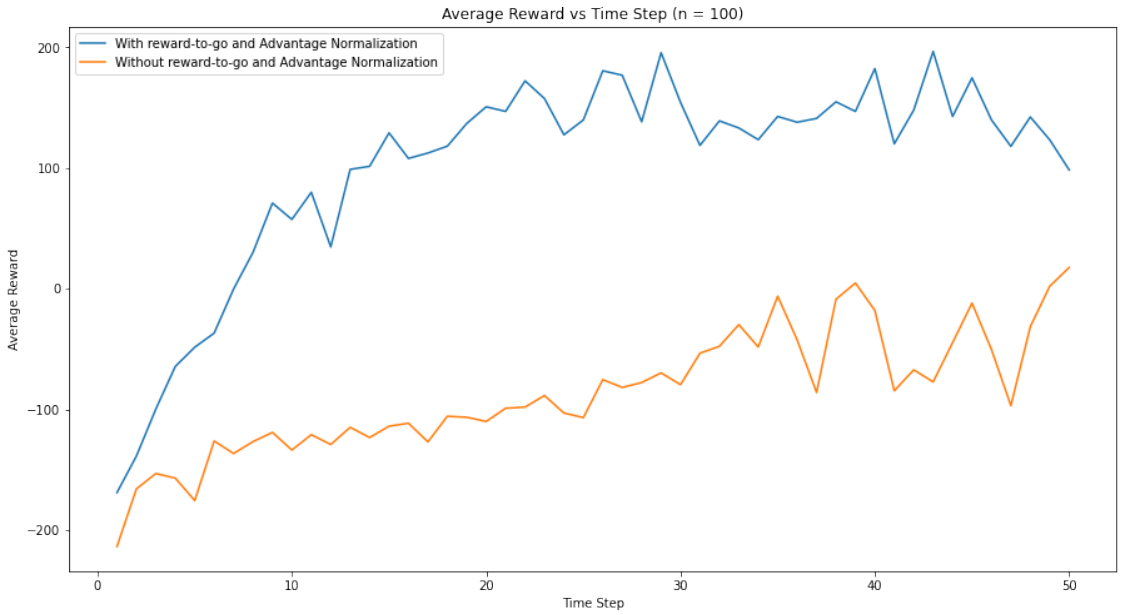
```
------- Testing -------
        Average Score : 30.10525589744494
```

```
In [16]: # Loading the final model from the folder. Just pass the number as parameter
         # For seeing the agent in action.
         agent_lunarlander_2.load(5000)
         agent_lunarlander_2.test(1,True)
```

```
------- Testing -------
        Average Score : -46.26395437162222
```

```
In [19]: x_1,y_1,y_best_1 = plot(scores_lunarlander_1,100)
         x_2,y_2,y_best_2 = plot(scores_lunarlander_2,100)
```

```
In [20]: plt.figure(figsize=(15,8))
         plt.plot(x_1,y_1,label = 'With reward-to-go and Advantage Normalization')
         plt.plot(x_2,y_2,label = 'Without reward-to-go and Advantage Normalization')
         plt.title('Average Reward vs Time Step (n = 100)')
         plt.xlabel('Time Step')
         plt.ylabel('Average Reward')
         plt.legend()
         plt.show()
```




```
In [21]: plt.figure(figsize=(15,8))
         plt.plot(x_1,y_best_1,label = 'With reward-to-go and Advantage Normalization')
         plt.plot(x_2,y_best_2,label = 'Without reward-to-go and Advantage Normalization')
         plt.title('Best Reward vs Time Step (n = 100)')
         plt.xlabel('Time Step')
         plt.ylabel('Best Reward')
         plt.legend()
         plt.show()
```