APPENDIX

Activity 13.01: Solving MountainCar with Experience Replay DQN

1. Create a new Jupyter Notebook and import the necessary packages (include the **`keras`** packages for deep neural networks):

```
#import packages
import gym
import numpy as np
import matplotlib.pylab as plt
import random as r
import math as m

# import neural network characteristics from keras
from keras.models import Sequential
from keras.layers import Dense
from keras.optimizers import Adam
```

2. Create a memory class that includes functions to initialize, append, and generate samples:

```
#initialize memory class
class MemoryRL:
    # initialization function
    def __init__(self, mem_max):
        self.mem_max = mem_max
        self.samps = []

    # appending function
    def append_sample(self, sample):
        self.samps.append(sample)
        if len(self.samps) > self.mem_max:
            self.samps.pop(0)

    # sample generation function
    def generate(self, samp_nos):
        if samp_nos > len(self.samps):
            return r.sample(self.samps, len(self.samps))
        else:
            return r.sample(self.samps, samp_nos)
```

In the preceding code snippet, you created a memory class for storing the samples that are generated when the RL agent interacts with the environment.

3. Create an agent class for initializing the environment memory and parameters that will be used:

```
class AgentRL:
    # agent parameters initialization
    def __init__(self, env):
        self.env = env
        self.memory = MemoryRL(5000)
        self.model = self.create_model()
        self.target = self.create_model()

        # initialize parameters
        self.epsilon_max = 1
        self.epsilon_min = 0.01
        self.gamma_p = 0.99
        self.batch_size = 50
        self.lamda = 0.0001
        self.tau = 0.099
```

In the preceding code, you instantiated the memory with a capacity of 5,000 samples and instantiated the parameters for Q-learning.

4. Create a function for the deep neural network model:

```
    #define deep neural network model
    def create_model(self):
        agent = Sequential()
        s_shape = self.env.observation_space.shape
        agent.add(Dense(50, input_dim=s_shape[0], activation="relu"))
        agent.add(Dense(50, activation="relu"))
        agent.add(Dense(self.env.action_space.n))
        agent.compile(loss="mean_squared_error",
optimizer=Adam(lr=0.02))
        return agent
```

In the preceding code snippet, you are creating the deep neural network by adding three layers and using **`mean_squared_error`** as the loss function and **`Adam`** as the optimizer.

5. Create a function for action selection:

```
    #create function for action selection
    def select_action(self, state, step):
        epsilon = self.epsilon_min + (self.epsilon_max - self.
epsilon_min)* m.exp(-self.lamda *step)
        if np.random.random() < epsilon:
            return r.randint(0, self.env.action_space.n - 1)
        else:
            return np.argmax(self.model.predict(state)[0])
```

   The preceding code updates the epsilon parameter at each step and enables greedy action selection.

6. Create functions for capturing the tuples and updating the Q values based on the batch size:

```
    #create function for updating the agent information
    def capture_memory(self, state, action, reward, new_state, done):
        self.memory.append_sample((state, action, reward, new_state,
done))

    #create function for q-updates
    def replay(self):
        batch_size = self.batch_size
        samples = self.memory.generate(batch_size)
        for sample in samples:
            state, action, reward, new_state, done = sample
            Q_val = self.target.predict(state)
            next_Q = self.target.predict(new_state)
            if done:
                Q_val[0][action] = reward
            else:
                next_Q = np.amax(next_Q)
                Q_val[0][action] = reward + next_Q * self.gamma_p
            self.model.fit(state, Q_val, epochs=1, verbose=0)
```

   In the preceding code snippet, you are appending the environmental tuples through the **`capture_memory`** function and activating the replay buffer by passing the batch size. You are then going through the memory array to determine the current and following Q values and updating the value-function.

7. Create a function for updating the weights of the model:

```
    #create function for updating the weights of the network
    def train(self):
        weights = self.model.get_weights()
        target_weights = self.target.get_weights()
        for i in range(len(target_weights)):
            target_weights[i] = weights[i] * self.tau + target_
weights[i] * (1 - self.tau)
        self.target.set_weights(target_weights)
```

The preceding snippet showcases the training function for updating the weights of the neural network model using the **`tau`** parameter.

8. Create a main function that runs for 100 episodes and displays the attempts of the agent:

```
#main function
if __name__ == "__main__":
    #generate the mountain car model
    env = gym.make("MountainCar-v0")
    trials = 10
    trial_len = 200
    dqn_agent = AgentRL(env=env)
    count_trials = 0
    stored_reward = []
    max_state = -100
    stored_f_state = []

    #for loop for episodes initialize state and reward
    for trial in range(trials):
        if trials % 10 == 0:
            print("Episode {} of {}".format(trial+1, trials))
        current_state = env.reset()
        current_state= np.reshape(current_state,[1, 2])
        total_reward = 0
        step_count = 0
```

By running the main function using 500 trials with 200 steps, you are also setting the maximum state and keeping track of the reward and the feature state. Print each episode start and initialize the current state and reshape it. You are reshaping the state so that you have the state format required for the DQN model.

9. Update the reward within the step loop so that the agent receives positive

rewards for each sub-goal it achieves:

```
        # for loop for steps, update action and pass information to 
env.
        for step in range(trial_len):
            env.render()
            action = dqn_agent.select_action(current_state, step + 1)
            new_state, reward, done, info = env.step(action)

            #reset reward values as positive rewards for states 
closer to the optimum
            if new_state[0] >= -0.5:
                reward += 1
            elif new_state[0] >= -0.1:
                reward += 5
            elif new_state[0] >= 0.1:
                reward += 10
            elif new_state[0] >= 0.25:
                reward += 20
            elif new_state[0] >= 0.5:
                reward += 100

            if new_state[0] > max_state:
                max_state = new_state[0]
            new_state = np.reshape(new_state,[1, 2])
```

The preceding code represents the iterative step loop. For the agent to learn faster, you assign higher rewards for states that are getting closer to the target. You want the new state position to be as close as possible to 0.5. You also perform a maximum state update, thus replacing any new state with the maximum threshold.

10. Update the agent, the state, and the reward, and collect information about the state and rewards:

```
            #update memory with newly generated values, replay and 
train
            dqn_agent.capture_memory(current_state, action, reward, 
new_state, done)
            dqn_agent.replay()
            dqn_agent.train()

            #update move to the new state unless the end of the 
episode
```

```
            current_state = new_state
            total_reward += reward
            step_count += 1
            if done:
                break
        #update total reward and final state printout steps and reward
        stored_reward.append(total_reward)
        stored_f_state.append(max_state)
        print("Steps {}, total reward {}".format(step_count, total_
reward))
```

In the preceding code snippet, you are updating the memory of the DQN model updating the current state with the new state and the total reward, and incrementing the steps. You are also using the **`print`** command to show the number of steps and the total reward.

11. Plot the state and rewards that were obtained for each episode:

```
    #plot the final states achieved and rewards for the episodes
    plt.plot(stored_f_state)
    plt.show()
    plt.close("all")
    plt.plot(stored_reward)
    plt.show()
```

You will obtain two plots: one for the states achieved compared to the number of episodes and one of the total rewards per episode compared to the number of episodes.

You should obtain an output similar to the following:

```
Episode 1 of 10
Steps 200, total reward -115.0
Episode 2 of 10
Steps 200, total reward -119.0
Episode 3 of 10
Steps 200, total reward -131.0
Episode 4 of 10
Steps 200, total reward -122.0
Episode 5 of 10
Steps 200, total reward -136.0
Episode 6 of 10
Steps 200, total reward -126.0
```

```
Episode 7 of 10
Steps 200, total reward -124.0
Episode 8 of 10
Steps 200, total reward -117.0
Episode 9 of 10
Steps 200, total reward -119.0
Episode 10 of 10
Steps 200, total reward -119.0
```

The plot for state selection over episodes can be visualized as follows:

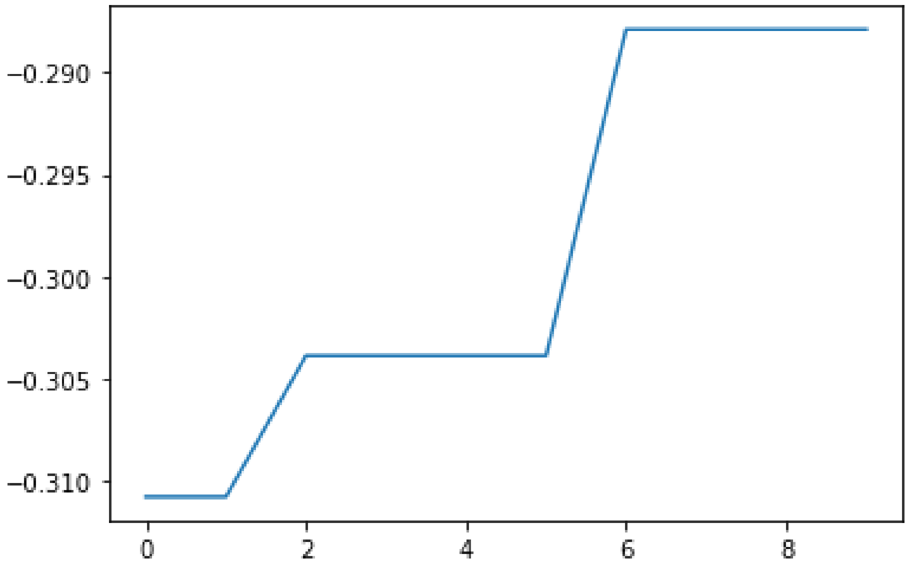

Figure 13.38: Maximum state selection over the number of episodes

In the preceding plot, the *x* axis denotes the number of episodes, whereas the *y* axis denotes the rewards for the states.

As you can see, the state rises from **`-0.31`** to **`-0.29`**, which means that the agent has found a local optimum for the first set of episodes for the state parameters: a combination of the car's velocity and position. As the number of episodes increases, the state begins to increase, getting closer to the optimum swing, which can enable the car to move past the hilltop. You can also observe that the values are maintained for several episodes (as is the case for the value between episodes **2** to **5**), which means that the agent manages to achieve the same final state, irrespective of its original position. This is indicative of its learning.

The following plot denotes the rewards that were obtained over the episodes:

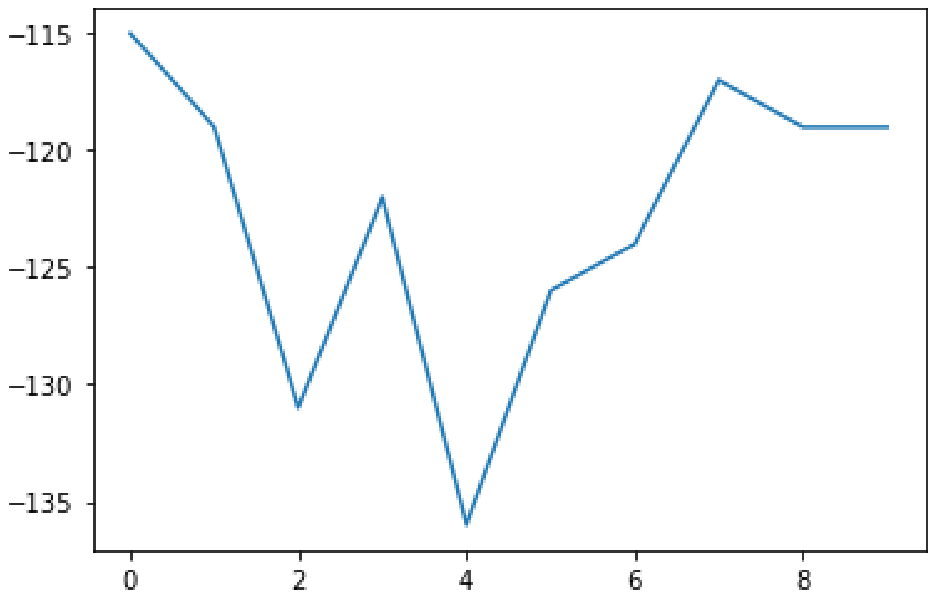

Figure 13.39: Total reward per episode

In the preceding plot, the *x* axis denotes the range of episodes, whereas the *y* axis denotes the rewards obtained over the states.

From this, you can observe that the reward registers a decline up to episode **4**, when it slowly rises and is maintained from episode **8** onward. Depending on the initial state (the position and velocity of the agent), the agent will be penalized if it spends too much time in specific positions that do not enable it to reach the goal. Once the optimal swing is found (close to -0.25), then the reward increases (around episode **6**).

NOTE

To access the source code for this specific section, please refer to https://packt.live/2AjoVuN.

This section does not currently have an online interactive example, and will need to be run locally.

By completing this activity, you have mastered model-based RL with experience replay and are now familiar and able to implement state-of-the-art RL algorithms using Python and TensorFlow to solve different OpenAI Gym problems.