13

RECENT ADVANCEMENTS AND NEXT STEPS

OVERVIEW

This chapter will introduce you to the state-of-the-art techniques for model-based reinforcement learning. By the end of this chapter, you will be able to find possible solutions to difficult areas such as complex robotics behavior, automatic vehicle control, and one-shot learning. You will also explore learning from human preference, hindsight experience relay, hierarchical reinforcement learning, and inverse reinforcement learning. This chapter will prove to be an effective tool for developing your programming skills so that you can work on reinforcement learning projects independently.

INTRODUCTION

Up until now, you have delved into the development of **Reinforcement Learning** (**RL**) algorithms, from learning about concepts such as dynamic programming and Markov decision processes to implementing deep learning algorithms using TensorFlow 2 and OpenAI. You have also familiarized yourself with deep learning concepts in RL such as deep Q-learning, deep recurrent Q-networks, and policy-based methods.

In the previous chapter, you examined the combination of neuroevolutionary strategies and RL. You started by looking at the differences between gradient-based and gradient-free methods, the principles of genetic algorithms, how to develop and implement genetic algorithms, and combining them with neural networks for parameter selection and topology evolution. Finally, you applied all this knowledge to train an RL-based system using a neural network that was optimized using a genetic algorithm.

In this chapter, you will discover the novel methods of implementing RL within the field with an emphasis on areas of further exploration such as one-shot learning and transferable domain priors. One of the significant challenges for RL is mimicking the decision-making process of a human when controlling a vehicle. By using inverse RL based on expert demonstrations, the control agent can automatically control a helicopter, for example. Model-based RL can also learn occupant preferences within a cabin environment and ensure a comfortable environment 89% of the travel time, as demonstrated in *An Examination of Comfort and Sensation for Manual and Automatic Controls of the Vehicle HVAC System by Petre et.al. 2019*.

> **NOTE**
>
> Refer to the following link for more information on the thesis, *An Examination of Comfort and Sensation for Manual and Automatic Controls of the Vehicle HVAC System by Petre et.al. 2019*:
>
> https://pureportal.coventry.ac.uk/en/publications/an-examination-of-comfort-and-sensation-for-manual-and-automatic-

Compared to the previous chapter, where you concentrated on model-free approaches, in this final chapter, you will explore model-based approaches and discover the differences between these two branches. So, the next step in the evolution of RL-based algorithms is exploring the relationship between expert human instruction and RL agent learning for it. Therefore, you will explore RL approaches that use human feedback and the types of learning that are available. Among the various types of learning, you will explore Q-learning and complete a demonstration exercise. The benefits of hindsight experience replay will be discussed as well, followed by an examination of hierarchical RL.

Finally, inverse RL will be outlined, and a discussion of the pitfalls of the methods and advancements of the RL domain. You will conclude this chapter by implementing a model-based RL method within the context of OpenAI Gym and examine the performance of the RL agent. In the next section, we'll begin by looking at the problems that occur with one-shot learning and the advantages of transferring domain priors.

NEXT-GENERATION RL – ONE-SHOT LEARNING AND TRANSFERABLE DOMAIN PRIORS

In the previous chapters, you have explored how the basis of RL is to represent negative and positive rewards (reinforcement). The rewards are associated with the environmental state and the actions that can alter the state. Due to the use of rewards for training the agent, this framework is classified as active learning.

One of the most challenging problems is conveying information to an agent in a single instance (also known as one-shot learning), which gives the agent the ability to infer and make correct decisions based on that information; for example, identifying a specific person based on a single photo, irrespective of the changes undergone by the person or the environment. As the person is identified in multiple situations, the RL agent can build a database of instances (or experiences). Learning from such experiences is one of the most sought-after skills of machine learning systems, especially within the robotics field.

One-shot learning is an advantage as it avoids the continuous interaction of the agent with the environment, which can cause high energy consumption, memory shortages, and relies on the availability of a human to supervise. However, single-instance training often proves insufficient for conveying enough information to the agent for it to perform well under any situation, especially for model-based agents that rely on accurate models of the environment. Additionally, a problem with RL systems is that the step-by-step learning of tasks is slow. Moreover, it is task-dependent as its learning process is affected by the complexity of the problem at hand.

In order to improve the learning speed of RL agents, domain-related experience is conveyed via human information in a generalized manner: either by using actions that are abstract and at a higher level, thus creating an abstract form of the state space using function approximation, or by dividing the task into a set of subtasks. This type of generalization for a task or between different tasks is known as **transfer learning** (**TL**).

TL relies on the RL agent being able to do any of the following:

- Identify a set of original tasks or a single original task from which the agent can infer actions that solve the current task.
- Infer the relationship between the original and current tasks.
- Transfer knowledge related to the original task to the current task.

In order to achieve these steps, the objective of the transfer needs to be identified. It can rely on reducing the time it takes for the agent to accomplish a task, or on using part of the process that was developed in solving a past task for a new task.

There are several ways to measure how well the agent makes use of the transferred information, as follows:

- Through a performance boost by using prior knowledge from the past task
- By improving the agent's final performance through the information provided
- Through the total reward accumulated by the agent when using the transferred information
- Through the ratio between the rewards accumulated by the agent trained with transferred information and the one without any transferred information
- The time it takes for the agent to reach a predefined threshold performance.

An emerging problem is that RL agents can take actions that can be considered dangerous or unsafe. To avoid this, we can use domain knowledge, which is extracted and refined from learned policies. This knowledge is referred to as domain priors and relies on incorporating prior policy-related information that incorporates undesirable behaviors within the state-action function of the agent in order to direct the agent toward exploring alternative actions that are safe for learning the current task (off-policy).

Alternatively, starting from a rough dynamic model that the behaviors are derived from, the domain knowledge is then transferred onto an online platform, where the model is then adapted based on its current experiences.

> **NOTE**
>
> For further reading on task-driven deep transfer learning, refer to the following paper by *Fu et al., 2016*:
>
> https://ieeexplore.ieee.org/document/7472110

Transfer learning can be exemplified by the reconfiguration of a neural network model that was pre-trained with a dataset by adding a layer that is fully connected as well as a classifier. This enables the neural network to use the parameters of the pre-trained model and further refine the classification output, as you can see in the following code snippet:

```
new_model = Sequential()
model.add(pre_trained_model)
model.add(Dense(230, activation= 'relu', input_dim = input_shape))
model.add(Dropout(0.2))
model.add(Dense(1, activation = 'sigmoid'))
```

The preceding code snippet changes the architecture of the deep neural network model to include two additional layers. The pre-trained model contains the weights and biases that are optimized through training and evaluation. The new model uses transfer learning as part of the architecture is already trained, but the new layers still need to be optimized.

In the following exercise, you will evaluate the performance of a model trained using the information and structure of a pre-trained model. You will re-evaluate the elements that you learned about in previous chapters related to TensorFlow and Jupyter Notebooks by using a MobileNetV2 pre-trained model. This model is instantiated with the ImageNet database.

EXERCISE 13.01: IMPLEMENTING TRANSFER LEARNING FOR IMAGE RECOGNITION

In this exercise, you will train and validate a deep neural network model using a pre-trained **`MobileNetV2`** model (available in Keras TensorFlow) on the ImageNet database, which constitutes a simple implementation of transfer learning. The goal of this exercise is for you to implement a model using the weights of a subsequent model in order to remember past commands in TensorFlow for model training and validation. This essentially means that the model is trained with parameter priors. By doing this, you will be able to explore the pitfalls of implementing a model that's been trained with a different dataset in a new dataset and examine its performance.

> **NOTE**
>
> We will be using the horse-or-human dataset in this exercise. The full dataset is sourced from http://www.laurencemoroney.com/horses-or-humans-dataset/.
>
> The dataset is also available on our GitHub repository at https://packt.live/37lodcu.
>
> The pretrained model, **`MobileNetV2`**, is available from the official documentation site at https://keras.io/api/applications/mobilenet/#mobilenetv2-function.

The following steps will help you to complete this exercise:

1. Create a new Jupyter notebook.
2. Next, import the necessary libraries, as shown in the following code snippet:

```
import tensorflow as tf
from tensorflow.keras import layers
import tensorflow_datasets as tfds
import matplotlib.pylab as plt
import os
import zipfile
from tensorflow.keras.preprocessing.image import ImageDataGenerator
```

3. Unzip the horse-or-human training dataset and validation-horse-or-human dataset, as shown in the following code snippet:

```
local_zip = '../Dataset/horse-or-human.zip'
zip_ref = zipfile.ZipFile(local_zip, 'r')
zip_ref.extractall('../Dataset')
local_zip = '../Dataset/validation-horse-or-human.zip'
zip_ref = zipfile.ZipFile(local_zip, 'r')
zip_ref.extractall('../Dataset/validation-horse-or-human')
zip_ref.close()
```

4. Next, create directories of each class label for training and validation:

```
# Directory with our training horse pictures
train_horse_dir = os.path.join('../Dataset/horse-or-human/horses')

# Directory with our training human pictures
train_human_dir = os.path.join('../Dataset/horse-or-human/humans')

# Directory with our training horse pictures
validation_horse_dir = os.path.join('../Dataset/validation-horse-or-
human/horses')

# Directory with our training human pictures
validation_human_dir = os.path.join('../Dataset/validation-horse-or-
human/humans')
```

5. Now, create a label list and print it:

```
train_horse_names = os.listdir(train_horse_dir)
print(train_horse_names[:10])

train_human_names = os.listdir(train_human_dir)
print(train_human_names[:10])

validation_horse_names = os.listdir(validation_horse_dir)
print(validation_horse_names[:10])

validation_human_names = os.listdir(validation_human_dir)
print(validation_human_names[:10])
```

The output will be as follows:

```
['horse01-0.png', 'horse01-1.png', 'horse01-2.png', 'horse01-3.png',
'horse01-4.png', 'horse01-5.png', 'horse01-6.png', 'horse01-7.png',
'horse01-8.png', 'horse01-9.png']

['human01-00.png', 'human01-01.png', 'human01-02.png', 'human01-03.
png', 'human01-04.png', 'human01-05.png', 'human01-06.png', 'human01-
07.png', 'human01-08.png', 'human01-09.png']

['horse1-000.png', 'horse1-105.png', 'horse1-122.png', 'horse1-127.
png', 'horse1-170.png', 'horse1-204.png', 'horse1-224.png', 'horse1-
241.png', 'horse1-264.png', 'horse1-276.png']

['valhuman01-00.png', 'valhuman01-01.png', 'valhuman01-02.png',
'valhuman01-03.png', 'valhuman01-04.png', 'valhuman01-05.png',
'valhuman01-06.png', 'valhuman01-07.png', 'valhuman01-08.png',
'valhuman01-09.png']
```

6. Print the lengths of each of the directories:

```
print('total training horse images:', len(os.listdir(train_horse_
dir)))
print('total training human images:', len(os.listdir(train_human_
dir)))
print('total validation horse images:', len(os.listdir(validation_
horse_dir)))
print('total validation human images:', len(os.listdir(validation_
human_dir)))
```

You will get the following output:

```
total training horse images: 350
total training human images: 310
total validation horse images: 128
total validation human images: 128
```

7. Create generations and lower the images to **300x300** based on the binary levels for both the training and validation datasets:

```
# All images will be rescaled by 1./255
train_datagen = ImageDataGenerator(rescale=1/255)
validation_datagen = ImageDataGenerator(rescale=1/255)


# Flow training images in batches of 128 using train_datagen
generator
train_generator = train_datagen.flow_from_directory(
        '../Dataset/horse-or-human/',  # This is the source directory
for training images
        target_size=(224, 224),  # All images will be resized to
150x150
        batch_size=128,
```

```
        # Since we use binary_crossentropy loss, we need binary
labels
        class_mode='binary')


# Flow training images in batches of 128 using train_datagen
generator
validation_generator = validation_datagen.flow_from_directory(
        '../Dataset/validation-horse-or-human/',  # This is the
source directory for training images
        target_size=(224, 224),  # All images will be resized to
150x150
        batch_size=32,
        # Since we use binary_crossentropy loss, we need binary
labels
        class_mode='binary')
```

In the preceding code snippet, we are loading the training and validation datasets.

The output will be as follows:

```
Found 660 images belonging to 1 classes.
Found 256 images belonging to 2 classes.
```

8. Instantiate the image shape and create a base model using **`MobileNetV2`**:

```
IMG_SHAPE = (224, 224, 3) # Create the base model from
the pre-trained model MobileNet V2 base_model = tf.keras.
applications.MobileNetV2(input_shape=IMG_SHAPE, include_top=False,
weights='imagenet')
base_model.compile(optimizer=tf.keras.optimizers.Adam(),
             loss='binary_crossentropy',
              metrics=['accuracy'])
```

9. Next, create a second neural network model:

```
average_pool = tf.keras.Sequential()
average_pool.add(layers.AveragePooling2D())
average_pool.add(layers.Flatten())
average_pool.add(layers.Dense(1, activation='sigmoid'))
```

10. Join the two models together using the **`Sequential`** function. At this stage, you are starting the transfer learning process by joining the pre-trained model, **`MobileNetV2`**, that learned from the ImageNet dataset and combining it with a second deep neural network model that has no prior training:

```
standard_model = tf.keras.Sequential([base_model,
    average_pool])
```

11. Compile the new model using **`binary_crossentropy`** as a **`loss`** function and **`accuracy`** for measuring performance:

```
standard_model.compile(optimizer=tf.keras.optimizers.Adam(),
              loss='binary_crossentropy',
               metrics=['accuracy'])
```

12. Train the model and fit it against the validation data for **5** epochs:

```
# import PIL
# print(standard_model)

history = standard_model.fit(
      train_generator,
      steps_per_epoch=5,
      epochs=5,
      verbose=1,
      validation_data = validation_generator, validation_steps = 3)
```

You will get the following output:

```
Train for 5 steps, validate for 3 steps
Epoch 1/5
5/5 [==============================] - 179s 36s/step - loss: 0.1007 -
accuracy: 0.9530 - val_loss: 19.4820 - val_accuracy: 0.4062
Epoch 2/5
5/5 [==============================] - 159s 32s/step - loss: 7.0636e-
08 - accuracy: 1.0000 - val_loss: 24.6309 - val_accuracy: 0.4062
Epoch 3/5
5/5 [==============================] - 186s 37s/step - loss: 4.1694e-
08 - accuracy: 1.0000 - val_loss: 27.7029 - val_accuracy: 0.4062
Epoch 4/5
5/5 [==============================] - 153s 31s/step - loss: 1.9931e-
08 - accuracy: 1.0000 - val_loss: 29.6762 - val_accuracy: 0.4062
Epoch 5/5
5/5 [==============================] - 150s 30s/step - loss: 1.5185e-
08 - accuracy: 1.0000 - val_loss: 30.7541 - val_accuracy: 0.4062
```

13. Create plots for examining the accuracy and cross-entropy for the trained and validated data:

```
# print(history.history)
a = history.history['accuracy']
v_a = history.history['val_accuracy']


l = history.history['loss']
```

```
v_l = history.history['val_loss']



plt.figure(figsize=(8, 8))
plt.subplot(2, 1, 1)
plt.plot(a, label='Accuracy of Training Data ')
plt.plot(v_a, label='Accuracy of Validation Data')
plt.legend(loc='lower left')
plt.ylabel('Accuracy')
plt.ylim([min(plt.ylim()),1.2])
plt.title('Accuracy of Transfer Model')

plt.subplot(2, 1, 2)
plt.plot(l, label='Loss for Training Data')
plt.plot(v_l, label='Loss for Validation Data')
plt.legend(loc='upper right')
plt.ylabel('Cross-Entropy')
plt.ylim([-0.5,1.2])
plt.title('Loss for Transfer Model')
plt.xlabel('Epochs')
plt.show()
```

In the preceding code snippet, we are measuring the performance of the transfer learning-based model (using accuracy and cross-entropy) against the training and validation data. It is expected that the performance of the training data will be higher than the validation one as the classification labels are known, whereas the validation data presents unknown classifications for the horse and human images. The model uses prior parameters from the pre-trained deep neural network model.

The final output will showcase two graphs:

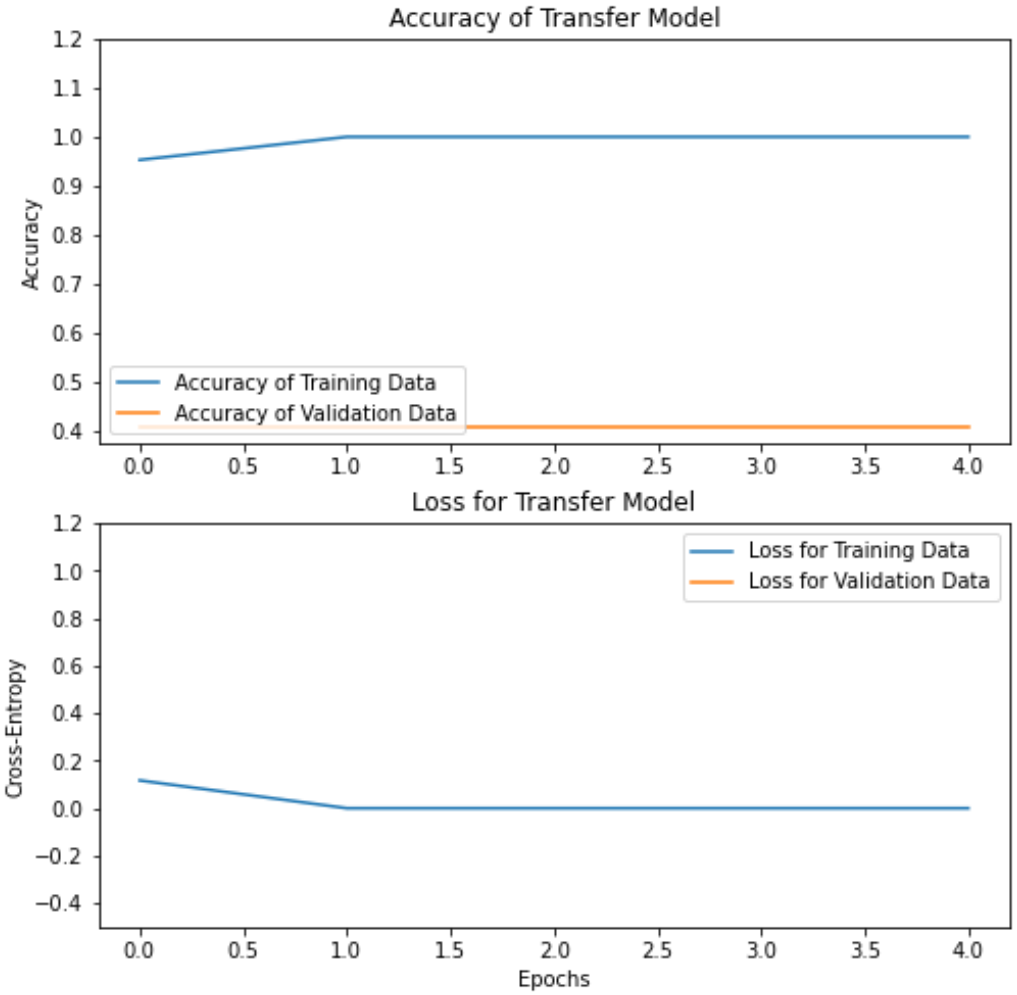

Figure 13.1: Accuracy and cross-entropy of the transfer learning model

Here, you can observe that while the accuracy for the training data is high, it decreases for the validation match as the weights of the initial model were obtained from a separate dataset (using the transfer learning approach). We can observe that the cross-entropy scores are identical for both the training and validation data (as the lines are overlapping). Potentially, if the model had been trained with a similar dataset, the accuracy would have achieved higher scores for the validation data, whereas the cross-entropy would have achieved a similar score (good performance is indicative of close to 0 log-loss).

> **NOTE**
>
> To access the source code for this specific section, please refer to https://packt.live/37oOF4Q.
>
> This section does not currently have an online interactive example, and will need to be run locally.

This exercise enabled you to understand and practically implement transfer learning using parameter priors from a model that was trained with a separate dataset. Now, you are able to identify the caveats of transfer learning since the type of prior knowledge is essential for improving model performance, and this knowledge is domain- (or dataset-) specific. The problem with most RL algorithms is that the agents cannot apply their pre-trained knowledge to new or altered environments. As a solution to this, in the next section, we will look at model-based RL.

MODEL-BASED REINFORCEMENT LEARNING

Model-free RL examines environments that do not possess an exact model. These kinds of RL systems were explored in *Chapter 11, Policy-Based Methods for RL* and *Chapter 12, Evolutionary Strategies for RL*. Conversely, model-based RL relies on the fact that the dynamics of the environment are already known. The latter method has several advantages, such as that it is sample-efficient and achieves faster learning times.

The environment is Markovian as the transitions between the current state, s_t, to the current action, a_t, and the following state, s_{t+1} are independent. The environment is represented by a set of parameters defined by a (S, A, P, R, γ) tuple. The next state, s_{t+1}, pertaining to the state set, S, can happen at time $t+1$ with probability P, as a result at time t of current action a_t of the action set A and state s_t belonging to S. The evaluation of the state-action pair (s_t, a_t) is represented by the reward function $R(R : S \times A \times S \to \mathbb{R})$.

The aim of the agent is to achieve the return, also known as the maximum total reward (G_t), that is discounted, as shown in the following equation:

$$G_t = \sum_{k=0}^{T} \gamma^k R(s_t, a_t)$$

Figure 13.2: Expression for maximum total reward

The discount factor $(\gamma \in [0\ 1])$ is a weight that's assigned to the future rewards that exponentially decrease their value. This, in turn, affects the impact of future rewards on current probabilities, thus favoring current rewards.

In order to reach a specific goal (for example, a car passing a mountain from a valley) the agent using RL explores the environment by attempting the actions that alter its state. For example, the car needs to move forward and backward to gain momentum in order to reach the peak of the mountain. Mapping the states of the environment (for example, a combination of car-related parameters) and actions (movement) is defined by policy $\pi(\pi: S \rightarrow A)$.

The agent selects the action that will lead to the next state with the highest reward associated by means of the value function. The value function determines the return based on the total reward accumulated by the agent, strictly concerning the current and future states by following policy π. E_π represents the mathematical term for the expected value:

$$V(S_t) = E_\pi \{ G_t \mid s_0 = s_t \}$$

Figure 13.3: Value function for total reward

The state and actions in the framework can be visualized as follows:

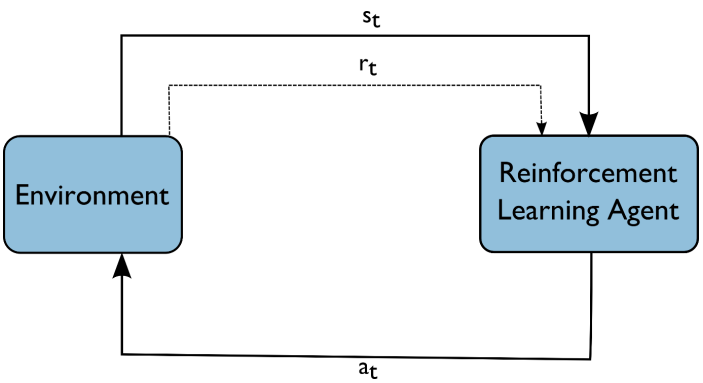

Figure 13.4: Mapping the states and action based on the RL framework

In some cases, the next state is a result of the actions of the RL agent, not just the states. Therefore, pairing the state and its actions is essential using the value function associated with them, based on policy π:

$$Q(s_t, a_t) = E_\pi\{G_t \mid s_0 = s_t = a_t\}$$

Figure 13.5: Value function for pairing the state and action

A policy is optimal (π^*) if it achieves the maximal long-term total reward determined by the state-action value function. Therefore, the optimal policy, compared to alternative policies, would be higher:

$$Q^*(s_t, a_t) = max_\pi\left[Q_\pi(s_t, a_t)\right]$$

Figure 13.6: State value function with optimal policy

In order to identify the optimal policy used by a reinforcement learning agent, two main approaches can be identified:

- Optimizing the value function
- Policy search

The former approach is based on finding an action course that leads to the desired states of the environment by means of modeling the return. This not only impacts the immediate reward but also the future rewards associated with the states (and the actions, respectively), resulting in policy optimization, which will be explored in the following sections. The latter deals with state and action spaces that have high dimensionality (policy-based methods were explored in *Chapter 11, Policy-Based Methods for RL*). It concentrates on parameter optimization for policy improvement.

The model-free and model-based approaches are most notably applied and explored within the robotics field for developing robotic behavior. For recurring or habitual behaviors, model-free RL is used. A caveat of this is that the learning is lengthy as it relies on updating the Q-table based on the states that the agent explores, so the agent is only concentrating on a part of a subspace where the task is executed. For model-based RL, the behavior of the agent is target-oriented as environmental transitions and rewards are available to guide it toward task completion. These two parameters are updated incrementally, and the Q-table is updated when a change is identified.

We studied Q-learning in depth in *Chapter 9, What is Deep Q Learning?* In the following exercise, you will implement a Q-learning-based agent to solve an OpenAI problem that will enable you to understand how to implement RL algorithms when the model of the environment is known.

EXERCISE 13.02: IMPLEMENTING Q-LEARNING FOR THE FROZENLAKE-V0 ENVIRONMENT

In this exercise, you will implement a Q-learning-based agent to solve the FrozenLake-v0 problem in OpenAI gym, where the agent needs to cross a lake that is frozen. The risk is that in some parts of the lake, the ice has melted, which means the agent can fall in. In this version of the experiment, the agent can also slip, which means that the agent will move into a problem area. In this exercise, you will learn how to develop a Q-learning-based model that captures the state-action transitions and acts based on the actions that trigger the maximum return.

The goal of this exercise is to allow you to implement a Q-learning agent in a model-based discrete environment (similar to a maze) and understand the differences between model-based and model-free algorithms.

The following steps will help you to complete this exercise:

1. Create a new Jupyter notebook.
2. Import all the necessary libraries, as shown in the following code snippet:

```
import gym
import random
import numpy as np
import time
```

3. Instantiate the environment as **`FrozenLake-v0`** (this step creates a model of the environment):

```
#invoke the environment
env_name = "FrozenLake-v0"
# instantiate environment
env = gym.make(env_name)
```

4. Print out the parameters for the states and actions (this step allows you to check the type of environment, that is, whether it's discrete or continuous):

```
# output variables for state and action
print("Observation space:", env.observation_space)
print("Action space:", env.action_space)
```

You will get the following output:

```
Observation space: Discrete(16)
Action space: Discrete(4)
```

In the preceding output, you can check the type of the state and action space (which is discrete) and the number of available states (or observations – 16) and the number of actions (4 – corresponding to the agent's movement: up, down, left, and right).

5. Create a class for the agent, along with an initialization function:

```
#create agent class
class Agent():
    def __init__(self, env):
        #initialize variable
        self.is_discrete = \
            type(env.action_space) == gym.spaces.discrete.Discrete
```

6. Within the initialization class, check if the space is discrete and identify the bounds of the action space, as shown in the following code snippet:

```
        if self.is_discrete:
        # save action size -discrete space
            self.action_numbers = env.action_space.n
            print ("Action size:", self.action_numbers)
        else:
        # identify parameters for the action space
            self.action_min = env.action_space.min
            self.action_max = env.action_space.max
            self.action_dist = env.action_space.dist
            print("Action range:" , self.action_min,self.action_max)
```

In the preceding code snippet, we are identifying the boundaries of the environment so that we can update the actions.

7. Next, create a function that will take an action that's been either randomly or uniformly sampled from the action space:

```
    def  get_action(self, current_state):
        if self.is_discrete:
            action  = random.choice(range(self.action_numbers))
        else:
            #get an action that is uniformly selected from the range
```

```
            action = np.random.uniform(self.action_min,self.action_
max, self.action_dist)
        return action
```

In the preceding code snippet, we created a function for action selection. If the value is already available and the environment is discrete, then an action is randomly selected from the available ones. If it is continuous, the action is selected based on the action space bounds.

8. Create a Q-learning agent, along with an initialization function for both the state space and the parameters, and build the model:

```
# create Q-learning agent class
class QLAgent(Agent):
    def __init__(self, env, gamma = 0.97, alpha = 0.01):
        super().__init__(env)
        self.state_numbers = env.observation_space.n
        print('State size:', self.state_numbers)

        self.greedy = 1.0
        self.gamma = gamma
        self.alpha = alpha
        self.build_model()
```

In the preceding code snippet, we are constructing the Q-learning agent by setting the parameters and calling the **`build_model()`** function.

9. Create a function for building the model:

```
    def build_model(self):
            self.q_table = 1e-4*np.random.random([self.state_
numbers,self.action_numbers])
```

10. Create a function for selecting the action invoking the **`super()`** class of the agent and selecting the action greedily:

```
    def get_action(self, current_state):
            current_q = self.q_table[current_state]
            action_greedy = np.argmax(current_q)
            action_random  = super().get_action(current_state)
            return action_random if random.random() < self.greedy
else action_greedy
```

In the preceding code snippet, we are implementing greedy selection of the action based on the highest value of state-action pairs in the Q-table.

11. Create a function for training the agent by updating the Q-table:

```
    def train (self,current_state, action, new_state, reward,
experience):
            done = experience

            new_q = self.q_table[new_state]
            new_q= np.zeros([self.action_numbers]) if done else new_q
            target_q = reward + self.gamma*np.max(new_q)

            update_q = target_q - self.q_table[current_state,action]
            self.q_table[current_state,action] += self.alpha*update_q

            if done:
                self.greedy = self.greedy * 0.99
```

In the preceding code snippet, we are creating a function for updating the Q-table with the changed transitions based on a Bellman equation.

12. Instantiate the RL agent and the total reward and run a for loop for 200 episodes. Reset the environment at every episode:

```
agent = QLAgent(env)
total_reward = 0
for ep in range (200):
    current_state = env.reset()
    done = False
```

13. Run a while loop until the goal is reached or the total steps are complete. Then, select the action and the tuples based on the action. Train the agent with the tuples (update the Q-table) and update the state and total reward:

```
    while not done:
        action = agent.get_action(current_state)
        new_state, reward, done, info  = env.step(action)
        agent.train(current_state,action,new_state,reward,done)
        current_state = new_state
        total_reward += reward

        print("state:", current_state, "action:", action, "next_
state:", new_state)
```

```
        print("episode: {}, total reward: {}, epsilon: {}"
.format(ep,total_reward, agent.greedy))
        env.render()
        time.sleep(0.05)
```

In the preceding code snippet, we are running the model step by step and choosing the actions greedily, based on the actions the additional environmental and agent-based parameters (the next state, the reward, and checking if the goal has been achieved) have updated. Once the Q-table has been updated, the state and the return are updated, leading to a new step.

14. For every episode, print the total reward and the greedy selection parameter, and at every state, print the action and new state:

```
        print("state:", current_state, "action:", action, "next_
state:", new_state)
        print("episode: {}, total reward: {}, epsilon: {}"
.format(ep,total_reward, agent.greedy))
        env.render()
        time.sleep(0.05)
```

In the preceding code snippet, you are outputting the model-based parameters, such as the total reward for each episode and the action selected. In order to create an episode update for the model, you will need to use **`render()`**, which is a function that's available in the gym package. We are also introducing a 0.05 second delay so that all the information can be displayed.

You will get an output (few lines displayed) similar to the one shown in the following screenshot since the actions are chosen randomly. Depending on the number of episodes you will be running, your output may vary slightly from this one:

```
  (Up)
SFFF
FHFH
FFFH
HFFG
state: 3 action: 2 next_state: 3
episode: 99, total reward: 2.0, epsilon: 0.36972963764972655
  (Right)
SFFF
FHFH
FFFH
HFFG
state: 3 action: 0 next_state: 3
episode: 99, total reward: 2.0, epsilon: 0.36972963764972655
  (Left)
SFFF
FHFH
FFFH
HFFG
state: 3 action: 0 next_state: 3
episode: 99, total reward: 2.0, epsilon: 0.36972963764972655
  (Left)
SFFF
FHFH
FFFH
HFFG
state: 7 action: 0 next_state: 7
episode: 99, total reward: 2.0, epsilon: 0.36603234127322926
```

Figure 13.7: Output for FrozenLake

Here, you will notice that each of the states that the agent is in are highlighted in red and that the action is also displayed in text form. We have only shown a part of the output here. In the complete output, you will observe that the reward is 0 at the beginning since the actions are randomly selected and the agent is not succeeding in achieving their goal (traversing the frozen lake). However, as can be seen from the preceding truncated output, the reward increases eventually by the time we reach episode 99.

> **NOTE**
>
> To access the source code for this specific section, please refer to https://packt.live/3fiG8De.
>
> You can also run this example online at https://packt.live/37vOGEk.

In the next section, we will learn how human preferences influence the sphere of RL.

LEARNING FROM HUMAN PREFERENCE

In the previous chapters, the implementations of RL algorithms have lacked one major component: human input or feedback. In various fields, predominantly robotics and gaming, the potential of human preference and feedback is widely explored. Feedback from the user can benefit the RL algorithms in two ways, as follows:

- Improves the learning performance for control optimization
- Allows research into human behavior by enabling both the human and RL agent to learn from one another

These benefits depend on the expertise of the user, the quality of the information that is provided, and the manner in which the information is provided.

Integrating user preferences or commands at different levels of the algorithm not only improves the learning performance, but also leads to a personalized system and enables the agent to preserve its autonomy when little feedback or erroneous information is provided. The feedback from the user can act as a bypass for specific actions (for example, changing their order of execution). This enables the agent to focus on a specific region of the environment (or specific characteristics of it) while learning more aspects about the preferences of the user that enable it to select alternative strategies without altering the original policy, thus avoiding any loops that are triggered by being stuck at locally optimal rewards.

The phenomenon of the agent going in a loop when choosing a set of suboptimal actions with a high current reward continuously is also known as a positive cycle. One solution to combating such a pitfall is to integrate human feedback in a manner suitable for the people that are sharing their knowledge and combining human preferences with data from sensors. There are multiple ways in which humans can convey their preferences, as follows:

- Directly changing the policy of the agent
- Evaluating the agent by giving negative rewards
- Guiding the agent toward specific actions or areas of the environment via positive reinforcement

These types of training promote real-time corrections of unwanted agent behavior and positively impact their learning performance. Moreover, it can be conducted by normal users – not just expert programmers – through the use of regular communication behaviors. There are three communication strategies that humans can use to convey their knowledge to an RL agent, as follows:

- By demonstration
- By voice
- By shaping (for the latter, a subcategory is represented by active shaping, also known as **Training an Agent Manually via Evaluative Reinforcement** – **TAMER**)

The most frequently used technique is demonstration (which will be covered in *Exercise 13.03, Demonstration Capture*). This method relies on capturing human behavior by means of sensors and having the RL agent reproduce it (either via simulation or in the real world). Voice commands and evaluations constitute the method known as advice, where humans communicate directly with the agents. However, a challenge for these methods is speech recognition. The third method is shaping, which relies on human-defined rewards (either negative or positive) for training the agent so that it achieves an optimal behavior. TAMER is an interactive method of shaping associated with supervised learning that combines the reward from the human with the reward from the environment to improve the learning speed of the RL agent.

> **NOTE**
>
> Refer to the following dissertation by Knox for further reading on TAMER:
>
> https://www.bradknox.net/wp-content/uploads/2013/06/thesis-knox.pdf

Demonstration is a technique that's implemented predominantly within the robotics and gaming fields, especially for automating kinematic tasks that ease the work of humans. It relies on conveying training data either by directly operating the robot, where the movement is recorded using sensors; by the movements of the human being recorded through the robot's sensors, where the actions are then reproduced by the robot; or by placing sensors on the human body or the environment, where the information of the movements is transmitted to the robot. This technique is comprised of examples derived from the behavior that's exhibited by the human and learned by the RL agent. This behavior is represented as state-action pairs that are recorded during the demonstration.

In the following exercise, you will recognize the potential of expert demonstrations by creating your own batch of experiences for a MountainCar-v0 exercise. We, as humans, already have an understanding of how to guide the car so that it reaches the top of the hill. This means we can record demonstrations of us playing the MountainCar-v0 game using the arrow keys of our keyboard to guide the position and the velocity of the vehicle, then use it to train an RL-based agent so that it can use the transitions and rewards as a basis for its own learning process.

EXERCISE 13.03: DEMONSTRATION CAPTURE

In this exercise, you will be capturing how you play the MountainCar-v0 experiment as an expert game player. The aim of this exercise is for you to explore the potential of demonstrations in training an RL agent by using human gameplay.

Your task is to observe and understand the way in which you can connect the use of the keyboard to controlling an OpenAI gym environment and capturing a set of game playthroughs that will be used to train an RL agent. In this exercise, you will be the expert game player that controls the car via the left, right, and down arrow keys. Your demonstrations will be captured and saved in an **`Expert demonstration.npy`** file.

The following steps will help you to complete this exercise:

1. Create a new Jupyter notebook.
2. Import all the necessary libraries, as shown in the following code snippet:

```
import gym
import numpy as np
from pynput import keyboard
```

3. Create a function for outputting actions based on the arrow keys:

```
#create function for outputting actions based on the arrow keys
def key_call():
    with keyboard.Events() as events:
        for event in events:
            if event.key == keyboard.Key.left:
                return 0

            elif event.key == keyboard.Key.down:
                return 1
```

```
            elif event.key == keyboard.Key.right:
                return 2
```

In the preceding code, you are creating a function that associates the arrow keys with values that are equivalent to the three actions that are available to the agent controlling the mountain car.

4. Initialize the MountainCar-v0 environment and create an array for capturing the transitions:

```
#initialize mountain car environment
env = gym.make('MountainCar-v0')

transitions = []
episodes = 0
```

5. Run a **`for`** loop for **`episodes`** and display the simulation:

```
#run for loop for episodes
for episode in range(1):
    transition = []
    step = 0

    env.reset()
```

6. Run the simulation for 200 steps and call **`.call()`** to capture the value for the action demonstrated by the expert:

```
#run for loop for steps
    for step in range(200):

        env.render()

        action = key_call()
        current_state, reward, done, _ = env.step(action)
```

7. Set a condition for **`episode end`** if the car manages to mount the hill:

```
#set condition for episode end if goal is achieved
        if current_state[0] >= env.env.goal_position and step > 129:
            break
```

In the preceding code, we set a condition for ending the simulation once the goal of reaching the peak of the hill has been met.

8. Capture the transitions per step:

```
# capture transitions per step
        transition.append((current_state[0],current_state[1],
action))
        step +=1
```

9. Reshape the transitions as a float and print out the transitions for each step. Then, capture the transitions per episode:

```
#capture the transitions per episode
    trans = np.array(transition, float)
    print("Shape of transition:", trans.shape)
    episodes += 1
    transitions.append(transition)
```

10. Reshape the transitions for each episode and print them out:

```
#reshape transitions for each episode and print them
transitions_array = np.array(transitions,float)

print("Shape of transitions array", transitions_array.shape)
```

The output will be as follows:

```
Shape of transition: (200, 3)
Shape of transitions array (1, 200, 3)
```

11. Save the array in a file:

```
# save the array of demonstration in a file
np.save("Expert demonstration", arr= transitions_array)
```

Once the screen has been activated, you will need to start playing the game. You will go through a cycle of 20 instances of the game. The objective is for you to control the car using the left, right, and down keys on your keyboard to get the car past the top of the hill (the green flag). Once you've done this, the instance will reset. You can control the cart using the keys and as output, you will get an **`Expert demonstration.npy`** file that captures the transitions of the car.

> **NOTE**
>
> To access the source code for this specific section, please refer to https://packt.live/3flN5Uj.
>
> This section does not currently have an online interactive example, and will need to be run locally.

You should have an output similar to the one shown here.

Note that you will need to use the left key to help the car gain momentum as the hill with the flag is higher than the left one. The keys are directing the agent to get past the slope as soon as possible (with the least actions).

Here is a screenshot of where we first go left to gain momentum and then press the down key in order to go past the valley:

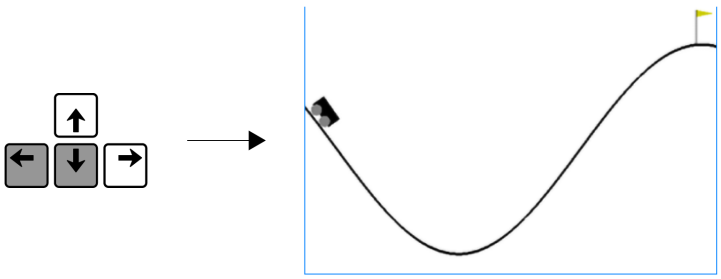

Figure 13.8: Using the left and down keys to gain momentum

Then, we use the right arrow key to climb up the hill:

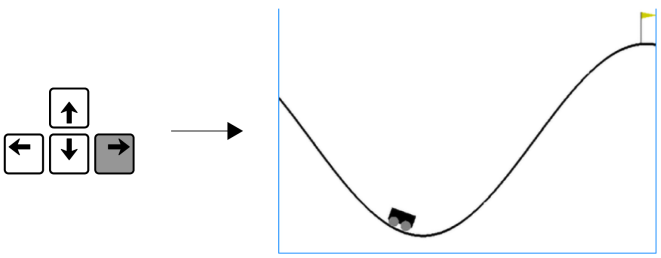

Figure 13.9: Using the right key to make the car climb the hill

We keep going right until we reach the flagpole, as shown here:

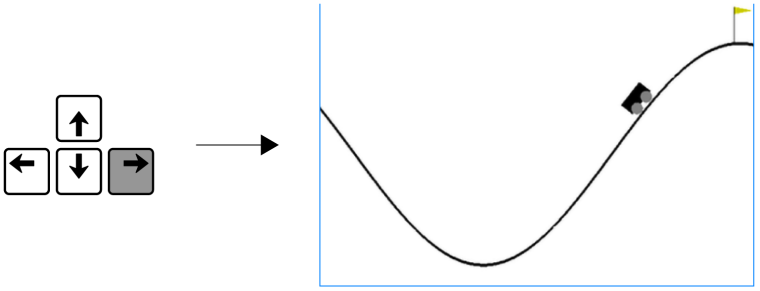

Figure 13.10: Using the right key to make the car reach the flagpole

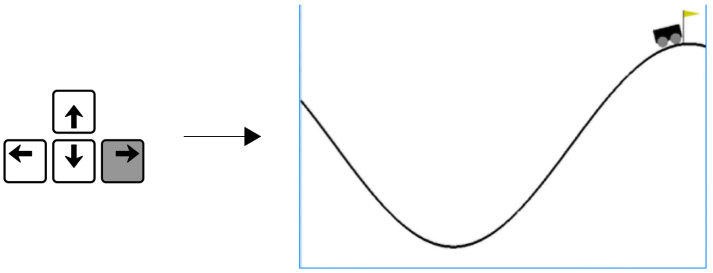

Figure 13.11: Using the right key until the car reaches the flagpole

As mentioned earlier, your demonstrations will be captured in the **`Expert demonstration.npy`** file. Thus, in this exercise, you have learned how to capture the demonstration of a human expert by using the OpenAI environment and keyboard control.

HINDSIGHT EXPERIENCE REPLAY

DQN relies on the use of passive memories of replaying past plays. These memories are batched together and are sampled to train the agent so that they interact efficiently and for training stabilization. This is a form of TL as the agent uses its own experiences to enhance its learning. This is unlike the demonstrations of an expert human, which rely on another agent's or the demonstrator's trajectories.

Experience replay, which you explored and implemented in *Chapter 9, What is Deep Q-Learning?*, is a method of training RL agents by generating unlinked data for deep neural networks. This is based on a replay buffer that captures the state-action pairs, the next state, and the reward associated with them, that is, $e=(s_t, a_t, r_t, s_{t+1})$. The buffer is a bridge between the training data that's used for each batch when updating the deep neural network. The RL agent interacting with the environment to find the optimal policy generates the data. The following code snippet represents how experience replay is appended:

```
experience.append((state, action, reward, new_state, done))
```

> **NOTE**
>
> Please refer to the following link for further reading from *Lipton et al., 2016* on replay buffers:
>
> https://www.researchgate.net/publication/306284840_Efficient_Exploration_for_Dialogue_Policy_Learning_with_BBQ_Networks_Replay_Buffer_Spiking

Each timestep experience over a set of episodes constitutes the dataset that is saved within a replay memory. Samples of the data are randomly selected and updates are applied to them for the Q-learning steps, leaving the RL agent to select the action greedily. The data is used efficiently as its samples are used to update the neural network's weights. The variance of consecutive samples is reduced due to randomized selection. The distribution of the behavior is averaged over the past states, leading to smoothening of the learning process and avoiding the agent being stuck at the local minima.

The caveat with this type of experience replay is the limited number of experience tuples (s_t, a_t, r_t, s_{t+1}) and the buffer having a fixed size of 10^6. Therefore, the buffer replaces the older tuples with newly generated experiences since it's not able to differentiate between important tuples and erroneous ones. The agent does not replace the demonstration data, thus allowing the agent to have a proportional sample of the information provided through demonstration and its own experience.

In order to solve this problem, prioritized replay was proposed, which determines the choice in tuple storage and replay toward efficient memory storage. This can be done by creating an oracle that selects the transition for the respective state greedily. To avoid function approximation errors and the greedy selection focusing only on a subset of the experiences, the method relies on stochastic sampling and uses the probability of the respective experience, $P(e)$, as shown in the following expression:

> **NOTE**
>
> For further reading on prioritized replay, refer to the following paper by *Schaul et al., 2016*:
>
> https://www.researchgate.net/publication/306284840_Efficient_Exploration_for_Dialogue_Policy_Learning_with_BBQ_Networks_Replay_Buffer_Spiking

$$P(e) = \frac{p_e^a}{\Sigma_k p_k^a}$$

Figure 13.12: Probability of the respective experience

The expression can be realized in code as follows:

```
x = random()
priority_tot = 0
for i in range(len(experiences)):
priority_tot += priority[i]**alpha
for i in range(len(experiences)):

probability =  (priority[i]**alpha)/ priority_tot

if x < probability:
            return experience[i]
```

The preceding code snippet accumulates the sum of all potential priorities of the subset k, then divides each subset priority by the sum to obtain the probability, $P(e)$. The experience is sampled based on the probability being higher than a random number, **x**.

The priority of the respective experience tuple, p_e, depends on the sum of the absolute value of the temporal difference error recently calculated, $|TDe|$, and the positive constant, ϵ. The priority is determined by the exponent, α (if it is 0, then it results in uniform sampling):

$$p_e = |TD_e| + \epsilon$$

Figure 13.13: Expression for priority of the respective experience

This expression can be realized in code as follows:

```
priority = alpha*(reward + gamma* next_Q - Q_val)
```

Because of the change in distribution, the priority in sampling introduces a bias for selecting the experience tuples and changes the convergence to a solution. This is averted by using weights that sample by importance, thus averting non-uniform probabilities by setting the parameter β to 1 and considering the number of samples, (N), as shown in the following expression:

$$w_e = \left(\frac{1}{N} \times \frac{1}{P(e)}\right)^{\beta}$$

Figure 13.14: Expression for weights that sample by importance

This expression can be realized in code as follows:

```
weight_exp = ((1/no_samples) + (1/probability))**beta
```

Alternatively, interactive replay relies on building a memory of the first exploration of the environment and using this model to train the agent in order to minimize the interaction with the real environment in the training phase.

Hindsight Experience Replay (**HER**) is a method that instead of using the original scope of the task, introduces a set of subgoals.

After the agent undergoes a set of episodes, when using the replay buffer, the agent is trained with the set of experiences (in the format $e_g = (s_t, g, a_t, r_t, s_{t+1})$), that it has undergone to achieve a subgoal (not reaching the actual goal of the task).

The challenge is selecting the subgoals that are to be used within the training process. This goal can be achieved when the agent reaches the final state of the respective episode (with each tuple becoming $e_g = (s_t, s_f, a_t, r_t, s_{t+1})$).

The advantage of this method is that there is no need to control the initial states of the environment. An additional advantage is that this method, combined with an off-policy algorithm such as DQN, can learn from sparse rewards, thus outperforming reward shaping methods. As the agent goes through the training process, it can slowly reach states similar to the ones for the optimal policy, with HER ensuring that the agent retains and uses these states to reach the state space area in which the actual goal (optimal solution) can be reached. This method is combined with **Deep Deterministic Policy Gradients** (**DDPG**), which was developed by *Lillicrap et al*. in 2015. This is an actor-critic based algorithm that uses two deep neural networks – one for the target policy (actor) and another that acts as a function approximator (critic) for the Q-function. The actor is trained with a gradient descent method on the loss function, whereas the critic is trained with a DQN Q-function, with the target being computed using the actions selected by the actor.

> **NOTE**
>
> For further reading on DDPG, refer to the following paper by *Lillicrap et al. (2015)*:
>
> https://arxiv.org/pdf/1509.02971.pdf

The combination of HER with DDPG succeeds in training robots in various complex tasks such as pushing, sliding, picking, and placing objects.

In the next exercise, you will understand and experiment with designing a buffer for HER.

EXERCISE 13.04: HINDSIGHT EXPERIENCE REPLAY CLASS

In this exercise, you will develop a replay buffer that relies on hindsight experience. This exercise will aid your implementation capabilities by getting you to insert sub-goals or targets for the experience tuples in order to enable the agent to learn from its experience by getting rewarded at different stages of its progress.

The following steps will help you to complete this exercise:

1. Create a new Jupyter notebook.
2. Import the **`NumPy`** and **`gym`** packages and install the **`collections`** package. You will need to import **`deque`** from the **`collections`** package:

```
#import packages
from collections import deque
```

```
import numpy as np
import gym
```

3. Create a class for the replay buffer, along with functions for initializing and resetting, by using the **`deque`** function:

```
class HindsightExpRep:
    #initialize buffer
    def __init__(self, m):
        self.buffer = deque()
        self.no = m

    #reset buffer
    def renew(self):
        self.buffer = deque()
```

4. Create a function for appending the samples:

```
    #append sample to buffer
    def append_sample(self, sample):
        self.buffer.append(sample)
```

5. Create a function for identifying previous samples as goals. Instantiate the target and length of the buffer:

```
    # function for setting the previous sets as goals
    def hindsight(self):
        length = len(self.buffer)
        target = self.buffer[-1][-2][0:self.no]
```

6. Run a **`for`** loop through the buffer and set instances as subgoals and compare the current instances to the target. If they match, then the session is complete. Return the replay buffer:

```
        for le in range(length):
            self.buffer[-1-le][2] = -1.0
            self.buffer[-1-le][-2][self.no:] = [target]
            self.buffer[-1-le][0][self.no:] = [target]
            self.buffer[-1-le][4] = False
            if (np.sum(np.abs(self.buffer[-1-le][-2][self.no:] -
target[0]))==0):
                self.buffer[-1-le][2] = 0.0
                self.buffer[-1-le][4] = True
            print(self.buffer)
        return self.buffer
```

7. Run the **MountainCar-v0** simulation for **5** steps and use the HER buffer:

```
m = 5
env = gym.make("MountainCar-v0")
no_acts = env.action_space.n
current_state= env.reset()
her = HindsightExpRep(m)

for j in range(m):
    her.renew()
    action = env.action_space.sample()
    new_state, reward, done, _ = env.step(action)
    sample = [current_state, action, reward,new_state, done]
#     print(current_state[0].squeeze(0).numpy())
#     print(new_state.squeeze())
    if (done == True):
        her.append_sample(sample)
    her.append_sample(sample)

mem_list = her.hindsight()
```

The output will be as follows:

```
deque([[array([-0.58623087,  0.        ]), 1, 0.0, array([-
0.56759566,  0.00512722]), True]])
```

Here, you can observe that the recorded instance is the last one since the agent has gone through the required steps. The goal of this exercise was to design a HER buffer that could be implemented within the RL environment. This can be utilized in combination with a simple RL agent for capturing the reward after a set of steps, but it can also be further expanded for a DQN agent.

> **NOTE**
>
> To access the source code for this specific section, please refer to https://packt.live/2UCDkcu.
>
> You can also run this example online at https://packt.live/3dXeYBI.

This exercise has helped you use the captured state-action tuples based on past experiences while using subgoals to reach the final solution. You now have the ability to implement a memory buffer that sets subgoals (targets) based on the accumulated experiences. These captured experiences can be passed onto a standard replay buffer that can be used to store the average reward as a threshold for current experiences.

Now, let's explore deep Q-learning from demonstrations.

DEEP Q-LEARNING FROM DEMONSTRATIONS

Deep RL based on Q-learning has garnered recognition as it offers a good approximation of the state-action value function (Q value) for high-dimensional state and action spaces. Q-learning models have good performance on environments with low dimensions where information about the state, action, reward, and the next state can be stored in a Q-table. However, it has poor performance for models that are larger in terms of scale as there is too much data to store.

In order to solve this problem, a deep neural network is used to map the transitions from states to actions. It does this with the objective of maximizing the optimal value function, $Q^*(s_t, a_t)$, in order to determine the optimal policy, $\pi^*(s_t)$:

$$\pi^*(s_t) = argmax_{a_t \in A} Q^*(s_t, a_t)$$

Figure 13.15: Expression for the optimal policy

This expression can be realized in code as follows:

```
Policy = amax(Q_optimal)
```

The optimal policy depends on the optimal value function fulfilling the Bellman equation, as shown in the following expression:

$$Q^*(s_t, a_t) = E_\pi\left[R(s_t, a_t) + \gamma \max_{a_t+1} Q^*(s_{t+1}, a_{t+1})\right]$$

Figure 13.16: Optimal value function fulfilling the Bellman equation

This expression can be realized in code as follows:

```
Q_optimal= reward + gamma* next_Q
```

The preceding code states that the optimal solution can be determined by selecting a_{t+1}, which is done by maximizing the expected value of the reward, $R(s_t, a_t)$, and the discounted optimal value function for the following state and action pairs ($\gamma \max_{a_{t+1}} Q^*(s_{t+1}, a_{t+1})$).

For approximating the state-action value function, instead of a function approximator, a deep neural network is used. The network acts as a replacement for the Q-table, attributing weights, (θ), to the Q-network,$Q(s,a,\theta_i)$. The **Deep Q-network** (**DQN**) has the objective of minimizing the sequenced loss functions that iteratively change, $L_i(\theta_i)$:

$$L_i(\theta_i) = E_{s,a \in p(s,a)}\left[(Q_i - Q(s,a,\theta_i))^2\right]$$

Figure 13.17: Minimization of the sequenced loss functions

This expression can be realized in code as follows:

```
loss_iterative = tf.reduce_mean(tf.squared_difference(Q-input,Q_val))
```

The state-action pair depends on the distribution of behaviors, $p(s,a)$, which is essentially a probability distribution over the sequences of states and actions. The iteration-dependent target (Q_i) is defined in the following equation:

$$Q_i = E_\pi\left[R(s_t, a_t) + \gamma \max_{a_t+1} Q(s_{t+1}, a_{t+1})\right]$$

Figure 13.18: Expression for the iteration-dependent target

The previous equation's weights, (θ_{i-1}), are fixed for optimizing the loss function. There are two advantages of DQN: it offers stability in terms of the Q values as it uses a separate network for each τ steps that are copied from the original network, and it can use experience replay (which will be discussed in the following section) in order to update the network. However, the DQN method produces overestimations for the value functions.

As a solution, double Q-learning expands on DQN, where instead of the maximum value of Q, a neural network obtains the maximum argument for the next action, depending on the next state and the target network $argmax_{a_t \in A} Q(s_{t+1}, a_{t+1}, \theta_i)$:

$$Q_i = E_\pi \left[R(s_t, a_t) + \gamma Q(s_{t+1}, argmax_{a_{t+1} \in A} Q(s_{t+1}, a_{t+1}, \theta_i), \theta_{i-1} \right]$$

Figure 13.19: Expression for double Q-learning, which expands on DQN

In order to further accelerate the learning speed of the DQN, **Deep Q-learning from Demonstration** (**DQfD**) was proposed by Hester et al. in 2018. The agent uses demonstration data to identify the initial parameters of the network before its environmental implementation. This technique not only uses the DQN loss function (which applies to a single step), but it also uses an n-step loss function, a large margin classification loss function that is supervised, and a **regularization loss** (**L2**). This combination is presented in the equation, where λ is the weighting parameter for the loss functions.

The Q-learning losses ensure that the Bellman equation is satisfied, whereas the supervised loss classifies the actions from the demonstration data. This enables the agent to prioritize the actions executed in the demonstrations. The n-step loss conveys the trajectory values of the demonstration to all the previous states, which determines a reward favoring future states. The L2 regularization loss is related to the network parameters' adjustment for the bias and weights in order to prevent overfitting in the case of a very limited dataset of demonstrations.

The loss function for demonstration data can be expressed as follows:

$$L_D(\theta) = \max_{a_t \in A} \left[Q(s_t, a_t, \theta) + l(a_{Dt}, a_t) \right] - Q(s_t, a_{\mathrm{Dt}})$$

Figure 13.20: Expression for supervised loss classifying actions from demo data

This expression can be realized in code as follows:

```
loss_demo = max(Q_val + self.loss_lin(action_demo, action))-Q_demo
```

The Q value learning function can be expressed as follows:

$$Q_N = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots + \gamma^{N-1} r_{t+N-1} + \max_a \gamma^N Q(s_{t+N}, a)$$

Figure 13.21: Q-learning loss function

This expression can be realized in code as follows:

```
Q_network = reward[i] + [gamma**i]* next_Q
```

The n-step loss function can be expressed as follows:

$$L_N(\theta) = E_{s,a \in p(s,a)}\left[\left(Q_N - Q(s,a,\theta)\right)^2\right]$$

Figure 13.22: n-step loss function

This expression can be realized in code as follows:

```
loss_iterative = tf.reduce_mean(tf.math.squared_difference(Q_network,Q_
val)))
```

The L2 regularization loss function can be expressed as follows:

$$L_2(\theta) = \theta_1^2 + \theta_2^a + \cdots + \theta_N^2$$

Figure 13.23: L2 regularization loss

This expression can be realized in code as follows:

```
loss_l2 = tf.math.reduce_sum([tf.math.reduce_mean(reg_l) for reg_l in
tf.compat.v1.get_collection(tf.GraphKeys.REGULARIZATION_LOSSES)])
```

The DQN loss function can be expressed as follows:

$$L(\theta) = L_i(\theta_i) + \lambda_1 L_N(\theta) + \lambda_2 L_D(\theta) + \lambda_3 L_2(\theta)$$

Figure 13.24: Sum of losses

This expression can be realized in code as follows:

```
loss = loss_iterative+ lambda1*loss_network+lambda2*loss_
demo+lambda3*loss_l2
```

Once training using the demonstrations is complete, the agent starts collecting more information by interacting with the environment.

Active Deep Q-learning with Demonstration (**ARLD**) improves the efficiency of demonstrations and introduces a query to the RL agent when there is a high degree of uncertainty regarding the action to be taken at the respective state. This query is addressed by the demonstrator (the expert human that provided the demonstration) and the algorithm decides between either the action recommended by the expert and the action chosen by the agent. The aim of this algorithm is to minimize the steps taken for solving the task (essentially combining the demonstration with the advice technique).

EXERCISE 13.05: CLASS DEVELOPMENT OF A DEEP Q-LEARNING AGENT FROM DEMONSTRATIONS

In this exercise, you will design a DQfD class in order to implement training from demonstrations. We will be designing this class by implementing the equations from *Figure 13.20* to *Figure 13.24* that we saw in the previous section. We will be creating a deep Q-learning agent that learns from demonstrations. The aim of this exercise is to transition from mathematical modeling to Python programming.

The following steps will help you to complete this exercise:

1. Create a new Jupyter notebook.
2. Import all the necessary libraries, as shown in the following code snippet:

```
#import packages
import tensorflow as tf
import tensorflow.compat.v1 as tfc
import numpy as np
import random
import functools
from keras.models import Sequential
from keras.layers import Dense
from keras.optimizers import Adam
```

3. Instantiate the agent class and configure the agent and the session start:

```
class DQfDAgent:
    def __init__(self, env, configuration,trajectories =None):
        self.session = tfc.InteractiveSession()
        self.conf = configuration
```

4. Instantiate the replay and demonstration memories:

```
        self.replay = Memory(capacity=self.conf.replay_buffer_size,
permanent_data=len(trajectories))
        self.demos = Memory(capacity=self.conf.demo_buffer_size,
permanent_data=self.conf.demo_buffer_size)
        self.include_trajectories(trajectories= trajectories)  # add
demo data to both demo_memory & replay_memory
```

   In the preceding code snippet, we are allocating memory buffers for the agent replaying its own simulated actions and for the demonstrations that have been captured from a human expert.

5. Instantiate the number of steps and state-action space:

```
        self.step = 0
        self.eps = self.conf.eps_init
        self.state_no = env.observation_space.shape[0]
        self.action_no = env.action_space.n
```

6. Create a function that will include transitions to demonstration memory:

```
    #function to include transitions to demonstration memory
    def include_trajectories(self, trajectories):
        for tra in trajectories :
            self.demos.store(np.array(tra, dtype=object))
            self.replay.store(np.array(tra, dtype=object))
```

   In the preceding code snippet, the demonstration and experienced trajectories are stored in an array with a conditional length for the length of the preset transitions.

7. Create a function for the greedy policy:

```
#function for greedy policy
    def greedy_act(self, current_state, model):
        if np.random.random() < self.eps:
            return np.random.randint(0, self.action_no - 1)
        return np.argmax(model.predict(current_state)[0])
```

   In this function, we are setting the greedy selection of the actions based on the maximum Q value (or random selection) using the greedy selection parameter known as epsilon.

8. Create a function for the model of deep neural network layers:

```
#function for deep neural network layers
    def neural_net_layers(self, no_units1, no_units2, reg=None):
        model = Sequential()
        model.add(Dense(no_units1, input_dim = self.state_no, kernel_
regularizer= reg))
        model.add(Dense(no_units2, kernel_regularizer= reg))
        model.add(Dense(self.action_no))
        model.compile(loss = "mean_squared_error", optimizer =
Adam(lr = self.conf.alpha))

        return model
```

In the preceding code snippet, we are creating the deep neural network step by step, building the layer connections and compiling the resulting model using the Adam optimizer.

9. Create a function for selecting the Q-table network:

```
    #function for selecting network
    def select_q(self):
        reg = tf.keras.regularizers.l2(l=0.2)
        return self.neural_net_layers(24, 48, reg)
```

In the preceding code snippet, we are building the input layer for the neural network that acts as a Q-table by including the input weight and bias.

10. Create a function for evaluating the network:

```
    #function for evaluating the network
    def eval_q(self):
        return self.neural_net_layers(24, 48)
```

In the preceding code snippet, we are building the input layer of the evaluation Q-table (or the n-step target, Q).

11. Create a function for training the network by using either the replayed function or the demonstrations:

```
    def train_network(self, train =False, update=True):

        self.step= self.step + 1

        actual_mem = self.demos if train else self.replay
        minibatch = actual_mem.sample(self.conf.minibatch)
```

```
        np.random.shuffle(minibatch)
        current_state_batch = [data[0] for data in minibatch]
        action_batch = [data[1] for data in minibatch]
        reward_batch = [data[2] for data in minibatch]
        new_state_batch = [data[3] for data in minibatch]
        done_batch = [data[4] for data in minibatch]
        demos_data = [data[5] for data in minibatch]
        nth_step_reward_batch = [data[6] for data in minibatch]
        nth_step_state_batch = [data[7] for data in minibatch]
        nth_step_done_batch = [data[8] for data in minibatch]
        actual_no = [data[9] for data in minibatch]
```

12. Call the neural network models for both individual and batched learning and initialize the batched learning arrays:

```
        # provide for placeholder, compute first
        select_q = self.select_q()
        eval_q = self.eval_q()
        n_step_select_q = self.select_q()
        n_step_eval_q = self.eval_q()

        y_batch = np.zeros((self.conf.minibatch, self.action_no))
        n_step_y_batch = np.zeros((self.conf.minibatch, self.action_
no))
```

13. Create a **`for`** loop for selecting the actions and updating the Q-tables based on the actions that have been selected:

```
        for i in range(self.conf.minibatch):
            temp = select_q.predict(current_state_batch[i].reshape((-
1, self.state_no)))[0]
            temp_0 = temp
            # add 1-step reward
            action = self.greedy_act(current_state_batch[i].
reshape((-1, self.state_no)), select_q)
            new_state_batch = new_state_batch[i].reshape(1,4)
            new_q = max(eval_q.predict(new_state_batch)[0])
            temp[action_batch[i]] = reward_batch[i] + (1 - int(done_
batch[i])) * self.conf.gamma * new_q
            y_batch[i] = temp
            # add n-step reward
            action = self.greedy_act(nth_step_state_
batch[i],select_q)
            n_step_new_q = max(n_step_eval_q.predict(new_state_
```

```
batch[i].reshape(1,4))[0])
            q_nth_step = (1 - int(nth_step_done_batch[i])) * self.
conf.gamma**actual_no[i] *n_step_new_q
            temp_0[action_batch[i]] = nth_step_reward_batch[i] + q_
nth_step
            n_step_y_batch[i] = temp_0


        return y_batch,n_step_y_batch,current_state_batch, action_
batch,demos_data
```

In the preceding code snippet, we are training and updating the network table and updating the memory bank either with replayed experiences or demonstration experiences.

14. Create functions for the losses:

```
    def loss_lin(self, ae, a):
        return 0.0 if ae == a else 0.75


    def loss_selec(self, select_q, action_batch, demo_data):
        inp = 0.0
        for i in range(self.conf.minibatch):
            ae = action_batch[i]
            max_val = float("-inf")
            for act in range(self.action_no):
                max_val = max(select_q[i][act] + self.loss_lin(ae,
act), max_val)
            inp += demo_data[i] * (max_val - select_q[i][ae])
        return inp


    def loss(self, select_q, y_batch, n_step_y_batch,action_
batch,demo_data, weights):
        loss_dq = tf.math.reduce_mean(tf.math.squared_
difference(select_q, y_batch))
        lloss_dq = tf.math.reduce_mean(tf.math.squared_
difference(select_q, n_step_y_batch))
        loss_inp = self.loss_selec(y_batch, action_batch, demo_data)
        loss_l2 = tf.math.reduce_sum([tf.math.reduce_mean(reg_l) for
reg_l in tfc.get_collection(tfc.GraphKeys.REGULARIZATION_LOSSES)])
        return weights * tf.math.reduce_sum([l * lam for l, lam in
zip([loss_dq, lloss_dq, loss_inp, loss_l2], self.conf.lamb)])
```

In the preceding code snippet, we are defining the loss functions that are represented by the equations from *Figure 13.20* to *Figure 13.23* in the *Deep Q-Learning from Demonstrations* section. The function returns the multiplication of the weights with the sum of the losses represented by the equation in *Figure 13.24*, which can be found in the same section.

15. Create functions for updating the trajectories and replaying them:

```
        def perceive(self, trajectory):
            self.replay.store(np.array(trajectory))

            if self.replay.full==True:
                self.eps = max(self.conf.eps_fin, self.eps * self.conf.
    decay)
```

In the preceding code snippet, we are passing the parameters for the neural network. The function stores the replayed trajectories and updates the epsilon parameter if the replay buffer is full.

16. Create a function for training from demonstrations:

```
        #function for training
        def train_ahead(self):
            for tr in range(self.conf.pretraining):
                self.train_network(train=True)
                if tr % 200 == 0 and tr > 0:
                    print('Training step with expert demonstrations: {}'.
    format(tr))
            self.step= 0
            print('Finished demo training')
```

In the preceding code snippet, we have written a function for the agent to run the simulations and update the Q-table using the demonstration transitions.

17. Create a **`Memory`** class that updates the instances and randomly selects samples from the batch of demonstrations or experiences:

```
    class Memory(object):
        def __init__(self, capacity, permanent_data):
            self.capacity = capacity
            self.permanent_data = permanent_data
```

```
            assert 0<=self.permanent_data <= self.capacity
            self.mem = []
            self.pos = 0
            self.full = False

        def store(self, transition):
            if len(self.mem) < self.capacity:
                self.mem.append(None)
            self.mem[self.pos] = transition
            self.pos = (self.pos + 1) % self.capacity
            if self.permanent_data >= self.capacity:
                self.full = True

        def sample(self, batch_size):
            return random.sample(self.mem, batch_size)

        def __len__(self):
            return len(self.mem)
```

In the preceding code snippet, you developed a **`Memory`** class that checks that the capacity is not exceeded and stores the updated tuples. It also randomly selects samples based on the batch size and can output the length of the allocated memory.

> **NOTE**
>
> To access the source code for this specific section, please refer to https://packt.live/37qeGRp.
>
> You can also run this example online at https://packt.live/3dXd9oq.

By completing this exercise, you were able to recreate the DQfD algorithms in an agent class by creating functions that combine all of the mathematical aspects of the algorithm. Due to this, you can implement this agent as a model to be trained using previous actions and demonstration trajectories for different OpenAI environments such as CartPole or MountainCar-v0.

In the next section, we will be examining a new type of RL that's based on hierarchies.

HIERARCHICAL REINFORCEMENT LEARNING

One of the main problems faced by RL agents is that as the state-action space increases, the number of parameters that are learned increases exponentially. A method that's used to solve this issue is to avoid making decisions at every step by using activities based on the respective policies until the end of the episode.

It also avoids the RL agent relearning a specific set of tasks every time a change is made to the environment or the environment is changed completely. **Hierarchical reinforcement learning** (**HRL**) is based on framing tasks, similar to programming a computer, by using subroutines and alternative programs where the high-level program calls the sub-levels for task execution. This essentially relies on hierarchical structures and representations that are based on semi-Markov decisions for formalization and are called at a high level.

Semi-Markovian decision processes (**SMDPs**) are a generalization of the MDP principle in that the duration between one decision and the next is defined by a random number (a real or integer number). The duration is a real value for systems of discrete events that last a continuous time, whereas integer-based values (positive) are used for discrete-time systems. This means that the RL agent will maintain the current state for a randomly assigned time, after which it instantaneously moves into the next state. Most hierarchical RL developments use discrete-time SMDP frameworks. This concept relies on the feudal hierarchies that have inspired an algorithm called feudal Q-learning.

The waiting time, τ, therefore becomes an additional variable to be accounted into the transition probability, $T\left(s_{t+1},\tau|s_t,a_t\right)$, from the current state, s_t, to the next state, s_{t+1}, where there are bounded immediate rewards, $R\left(s_t,a_t\right)$. This changes the optimal state value function, $V*\left(s_t\right)$, and optimal state-action value function equations, $Q*\left(s_t,a_t\right)$:

$$V*\left(s_t\right)=\max_{a\in A}\left[R\left(s_t,a_t\right)+\sum_{s_{t+1},\tau}\gamma^{\tau}T\left(s_{t+1},\tau|s_t,a_t\right)V*\left(s_{t+1}\right)\right]$$

Figure 13.25: Optimal state value function

The expression for the optimal action value function is as follows:

$$Q*\left(s_t, a_t\right) = R\left(s_t, a_t\right) + \sum_{s_{t+1},\tau} \gamma^{\tau} T\left(s_{t+1}, \tau | s_t, a_t\right) \max_{a_{t+1} \in A} Q*\left(s_{t+1}, a_{t+1}\right)$$

Figure 13.26: Optimal state action value function

Hierarchical RL relies on introducing subroutines that can execute basic commands or actions. These subroutines focus on policies that cover a subset of the state space. This leads to timestep actions to be defined as activities, which can trigger alternative activities, depending on the hierarchical distribution of the policy. Grouping these activities constitutes the high-level action, a_t, that is executed during the waiting time, τ.

This relationship can be described as having a high-level SMDP (the parent) that calls a child SMDP policy. Each child SMDP can call alternative actions. This design is a specifically defined form that states which actions and states are covered at which level. Some of the policies can be hand-coded to provide the information of a human expert. This approach has several advantages such as faster learning and high generalization capacity, the ability to explore the action through sub-policies and activities, and it also facilitates better knowledge transfer from an expert to the agent, or from one agent to another.

There are several hierarchical algorithms, the most notable of which are **Hierarchical Abstract Machines** (**HAMs**), which use finite state machines that call on lower-level machines with the policy represented by a finite state automaton and four types of state machines (actions, calls, choices, and stops).

Another algorithm, MAXQ, decomposes the Q-function as a sum of the total reward for the state-action pair that's executed by the child, $V^{\pi}\left(a_t, s_t\right)$, and the total reward, $C^{\pi}(j, a_t)$, for the performance of the parent based on taking the respective task (referred to as a completion function). The MAXQ framework manages to find the optimal policy of the parent, j, based on the policies learned by the children, thereby developing a temporal and spatial abstraction of the problem. The following is the expression for the Q function in terms of the MAXQ framework:

$$Q^{\pi}\left(j, s_t, a_t\right) = V^{\pi}\left(a_t, s_t\right) + C^{\pi}\left(j, s_t, a_t\right)$$

Figure 13.27: Q function in terms of MAXQ

The sample code for the Q function is as follows:

```
Q_policy= V_child + C_parent
```

Hierarchical deep reinforcement learning (**h-DQN**) is a framework that organizes a set of deep RL modules at different time scales. The two-level decision hierarchy is comprised at a low level, where the actions are taken based on the states and goals selected at a high level. The low-level action selection continues until the end of the episode or the goal has been achieved. The optimization method uses gradient descent, which uses varied time scales in order to achieve the highest rewards at both low and high levels.

In the following exercise, you will implement the classes for feudal Q-learning, which is a hierarchical RL algorithm that relies on reproducing the feudal hierarchies of medieval times.

EXERCISE 13.06: Q-TABLE UPDATE USING FEUDAL Q-LEARNING

In this exercise, you will apply the knowledge you've gained on how to construct a hierarchical RL to update the Q-learning table and establish the relationship between the parent and child agents (higher level to lower levels) by executing a one-step update for a grid world exercise. In the grid world, the agent can move in four directions: up, down, right, and left. The goal is to mimic the transition from an initial step to the agent moving right and printing out the Q-level update.

The following steps will help you to complete this exercise:

1. Create a new Jupyter notebook.
2. Import the necessary libraries:

```
import numpy as np
import pandas as pd
```

3. Create a class for updating the Q-table and write an initialization function for the table by dividing it into layers (hierarchical levels) based on the actions:

```
class FQlearnTable:
    #initialize the table by dividing it into layers based on the
actions
    def __init__(self,no_actions,no_layers=3):
        self.no_acts = no_actions
        self.no_layers = no_layers
        self.levels = {0: FLevel(action = [no_actions])}
        if(no_layers > 1):
            for i in range(1, no_layers-1):
                self.levels[i] = FLevel(action = list(range(no_
actions+1)))
            self.levels[no_layers - 1] = FLevel(action =
list(range(no_actions)))
```

In the preceding code snippet, we initialized the number of actions available to use in the environment and the number of hierarchical layers used for the sub-distribution. The Q-table is then sub-organized into multiple levels, creating sub-tables that capture parts of the state-action transitions.

4. Create a function for selecting an action depending on the level of the state:

```
# function for selecting an action depending on the level of the
state
    def select_action(self, current_state):
        lev_state = self.get_lev_st(current_state)
        action_array = []
        for act in range(self.no_layers):

action_array.append(self.levels[act].select_action(lev_state[act]))
        return action_array
```

In the preceding code, we have set an array of actions based on the current state at the respective level. Once an action is selected, it will be appended to the array.

5. Next, create a function that captures the model tuples based on the state update by allocating the lower-level reward:

```
#function capturing the model tuples based on the state update by
allocating the lower level reward
    def memory(self, current_state, action, reward, new_state, done):
        lev_state = self.get_lev_st(current_state)
        lev_primes = self.get_lev_st(new_state)

        good_r = 0
        bad_r = -1

        for i in range(self.no_layers):
            if i == 0:
                rwd = reward
            else:
                if action[i-1] == 4:
                    rwd = reward
                else:
                    if action[i-1] == action[i]:
                        rwd = good_r
                    else:
                        rwd = bad_r

            self.levels[i].memory(lev_state[i], action[i],
                                  rwd, lev_primes[i], done)
```

In the preceding code, the current state and the new state that were defined at the primary levels are passed through. At each level (layer), the reward is updated and the tuples are updated for the Q-function.

6. Create a function for updating the state based on the hierarchical levels:

```
#function for updating the state based on the hierarchical levels
    def get_lev_st(self,current_state):
        state_array = []
        state_array.append(current_state)
        for i in range(self.no_layers -2, -1, -1):
            state_array.append((int(state_array[-1][0]/2),int(state_
array[-1][1]/2)))
#           state_array.remove()
        return state_array
```

7. Create a base-level class and a function for initializing the parameters for Q-learning:

```
class FLevel:
#initializing the parameters for q-learning
    def __init__(self, action, lr = 0.01, r_decay=0.9, epsilon_g
=0.9):
        self.action = action
        self.lr = lr
        self.gamma = r_decay
        self.epsilon = epsilon_g
        self.q_table = pd.DataFrame(columns=self.action, dtype=np.
float_)
```

In the preceding code snippet, we passed the parameters at each level by creating a class that passes the action selected within the sub-level to the higher level.

8. Create a function for action selection:

```
#function for action selection
    def select_action(self, obs):
        obs = str(obs)
        self.verify_state(obs)
        #select action
        if np.random.uniform() < self.epsilon:
            #select the best action
            best = self.q_table.loc[obs, :]
            best = best.reindex(np.random.permutation(best.index))
            action = best.idxmax()
        else:
            action = np.random.choice(self.action)
        return action
```

In the preceding code, we are updating the Q-table by selecting the activity with the highest score and returning the activity to be passed to higher levels.

9. Create a function for updating the Q-table:

```
#a function for updating the q-table
    def memory(self, current_state, act, reward, new_state, done):
        current_state = str(current_state)
        new_state = str(new_state)
        self.verify_state(new_state)
        new_q = self.q_table.loc[current_state, act]
        if not done:
            q_threshold = reward + self.gamma * self.q_table.loc[new_
state, :].max()
        else:
            q_threshold = reward
        self.q_table.loc[current_state,act] += self.lr * (q_threshold
- new_q)
```

10. Create a function that checks for the index of the state corresponding to the specific level:

```
#function that checks for the index of the state corresponding to the
specific level
    def verify_state(self, current_state):
        if current_state not in self.q_table.index:
            self.q_table = self.q_table.append(pd.Series(
                 [0]*len(self.action), index= self.q_table.columns,
                name = str(current_state),))
```

11. Now, we will implement the gridworld environment and see how the Q-table gets updated. Run the code and test the comparison between lower-level and higher-level updates for a simple grid world update:

```
flevel = FLevel(['Left','Right','Top','Bottom'])  #grid world with
four actions
print("Q_Table at start\n", flevel.q_table)
start_state=(0,0) # the grid world (x,y) coordinates
flevel.verify_state(start_state)
print("\n\nQ Table after verifying initial state\n", flevel.q_table)


# now update memory for taking an action 'Right' and moving from
(0,0) to (0,1) with done=False(i.e. episode has not ended)
flevel.memory((0,0), 'Right', 4, (0,1), False)
print("\n\nQ Table after making a transition and getting a reward\n",
flevel.q_table)
```

You will get the following output:

```
Q_Table at start
 Empty DataFrame
Columns: [Left, Right, Top, Bottom]
Index: []


Q Table after verifying initial state
         Left  Right  Top  Bottom
(0, 0)   0.0    0.0  0.0     0.0


Q Table after making a transition and getting a reward
         Left  Right  Top  Bottom
(0, 0)   0.0   0.04  0.0     0.0
(0, 1)   0.0   0.00  0.0     0.0
```

Here, you can observe that after executing the transition to the right, the Q-table is updated with the new state of (0,1) and that the value of the right transition is updated with 0.04 instead of 0.0. This means that the lower level is updating the task selection and that the upper level updates the reward. The goal of this exercise was to enable you to apply the knowledge you've gained about constructing hierarchical RL by programming classes that update the Q-learning table and establish the level between (in this case) the master and servant for actions and policies. In this exercise, you built two different classes – a primary level and a sub-level one – and defined at which level the Q-update is made (lower level), which actually restructures the table for the entire structure (high level).

> **NOTE**
>
> To access the source code for this specific section, please refer to https://packt.live/2YHIQvF.
>
> You can also run this example online at https://packt.live/2MSFJeA.

In the next section, we will explore another type of transfer learning based on demonstration trajectories, which is known as inverse RL.

INVERSE REINFORCEMENT LEARNING

In RL, the goal is to maximize the expected reward. Here, its function is predefined, which, in turn, helps the agent learn the optimal behavior for solving a specific problem. **Inverse reinforcement learning** (**IRL**) changes the strategy by extracting the reward function from the behavior of a human expert. The main principle is to use captured data from the demonstrations of the human executing a specific task, such as driving or piloting a helicopter. From this data, an approximate reward function based on the human's behavior is extracted.

While the previous sections concentrated on finding the appropriate policy based on human demonstrations, IRL concentrates on finding the most suitable reward function for the agent, irrespective of the policy that the human acts upon.

Once the reward function has been identified, the problem can be solved by using any appropriate RL-based algorithm (for example, model-based or policy-based approaches or deep learning ones). This method eliminates the need for a predefined reward function and its limits, thus expanding the areas in which RL can be applied since it can be implemented as a form of inverse optimal control method. A subsequent benefit of this method is that it has high generalization potential since the identified reward function can be transferred to alternative agents, subject to the agents performing tasks within the same or a similar environment. Its goal is to be extended to cover the state-action space for these agents.

IRL stems from the development of human and animal behavior models and is associated with apprenticeship learning, of which two situations can occur, as follows:

- Predicting an agent's behavior by learning from it
- Learning a human expert's behavior (or actions) in order to execute the preferences and techniques used in the expert's behavior

The basis of IRL is that the reward function is represented by a sum of features that are weighted. The features, $\phi_k(s,a)$, are functions that capture information from the state space, which can be high-dimensional, $\phi: S \rightarrow \mathbb{R}$, with the weights, ω, being the fitting parameters that are not yet known. The objective is to identify the best values for the weights:

$$R(s,a) = \sum_{1}^{k} \omega_k \phi_k(s,a)$$

Figure 13.28: Reward function

The reward function is improved, step by step, by comparing the policy of the expert with a set of policies. The weights are initialized randomly and the applied policy is also randomly selected from the policy set. The algorithm relies on optimizing the policy by estimating the value function of the expert's policy, as well as the value functions for the alternative policies, and comparing them. These steps apply for both linear function approximation and for trajectory sampling:

$$max\sum_{s\in s}{}_{a}\min_{\in a_2,\ldots,a_m}\left\{p\left(E_{s_t+1}\sim p_{sa_1}\left[V^{\pi_1}\left(s_t+1\right)\right]-E_{s_t+1}\sim p_{sa}\left[V^{\pi_l}\left(s_t+1\right)\right]\right)\right\}\ for|\omega_i|\le 1, i=1,\ldots,k$$

Figure 13.29: Linear function approximation

$$max\sum_{i=1}^{k} p\left(\widehat{V}^{\pi*}\left(s_0\right)-\widehat{V}^{\pi_i}\left(s_0\right)\right) for|\omega_i|\le 1, i=1,\ldots,k$$

Figure 13.30: Trajectory sampling

This occurs when there are multiple reward functions that can apply to the dataset of demonstrations. Due to having a distribution of reward functions, a bias appears. To solve this problem, the maximum entropy principle is applied to develop the distribution and eliminate the bias. This relies on calculating the transition probability, $p(T)$, based on the likelihood of the demonstrated trajectory, T, having the maximum reward, divided by the likelihood of all possible trajectories:

$$p(T)=\frac{exp\left(R_{\omega}(T)\right)}{\int exp\left(R_{\omega}(T)\right)dT}$$

Figure 13.31: Expression for transition probability

Alternatively, the maximum entropy IRL relies on is the distribution of trajectories, $(P(T))$, with the maximum entropy among all distributions. The condition is that the features of the chosen policy are in line with the features of the demonstration, $(T\in D)$:

$$max\widehat{P(T)}\left(-\sum_{T\in D}P(T)logP(T), subject\ to\ \sum_{T\in D}P(T)=1\ and\ \sum_{T\in D}P(T)\sum_{t=1}^{\infty}\gamma^t\phi_k\left(s_t,a_t\right),\forall k\right)$$

Figure 13.32: Maximum entropy for all distributions

By introducing this method, the risk is that the state space increases, but given that the trajectories have similar distributions, this means that the states that are close to the current state can result in actions close to the optimal one. This means we can introduce a set of constraints for the features that favor these states.

The maximum entropy IRL was expended by *Wulfmeier et al.* in 2015 to using deep neural networks as a function approximator (for the reward function – a feedforward neural network) and for calculating the maximum entropy using gradient descent methods. Here, the backpropagated neural networks are separating the loss function into the rewards and the gradient of the rewards that are connected to the weights of the network, as shown in *Figure 13.33*. The goal is to maximize the joint posterior distribution, $P(D|r_t)$, of the demonstration, D, features for the current reward, r_t, and the neural network's weights (this can be extended to other network parameters), $P(\omega)$, as demonstrated in *Figure 13.34*. **Fully Convolutional Neural Networks** (**FCNNs**) are used to fulfill both types of networks:

$$L(\omega) = log\ P(D|r_t) + log\ P(\omega) = L_D + L_\omega$$

Figure 13.33: Expression for the loss function

$$\frac{\partial L(\omega)}{\partial \omega} = \frac{\partial L_D}{\partial \omega} + \frac{\partial L\omega}{\partial \omega} = \frac{\partial L_D}{\partial r_t} \times \frac{\partial r_t}{\partial \omega} + \frac{\partial L\omega}{\partial \omega}$$

Figure 13.34: Separating the loss function for the rewards and gradient of rewards

> **NOTE**
>
> For further reading on maximum entropy IRL, refer to the paper by *Ziebart (2008)* at the following link:
>
> https://www.aaai.org/Papers/AAAI/2008/AAAI08-227.pdf.

A set of problems emerged due to this method: the structure of this architecture belongs to a model-based RL branch, which means that the transition probabilities for the environment are already known. This process is time-consuming as it estimates the reward function at every step. Additionally, the process is affected by the variations in trajectory lengths.

Deep IRL by logistic regression is a model-free-based method that samples from two datasets: a demonstration one and a baseline one. From these, two policies are derived and their ratio is calculated. Logistic regression is used to estimate the densities for the logarithmic values of the ratio. The deep neural network uses the current and following state of the environment and outputs a label that identifies whether the two states pertain to the demonstration or the baseline dataset. The deep network is composed of a network estimating the logarithm of the density ratio, another calculating the reward, and a final one calculating the value function, which also takes into account the following state. Using the outputs of these three networks, the Bellman equation is computed and logistic regression is used for classifying the probabilities.

> **NOTE**
>
> For more reading on deep IRL, refer to the paper by *Uchibe (2017)* at the following link:
>
> https://link.springer.com/article/10.1007/s11063-017-9702-7

In the following exercise, you will create a program that trains an agent using a set of demonstrations from the MountainCar-v0 exercise and trains the agent via IRL. It will do this by using the captured trajectories from the demonstrations and analyzing their performance by plotting the rewards that have been learned.

EXERCISE 13.07: IMPLEMENTING INVERSE REINFORCEMENT LEARNING FOR MOUNTAINCAR

IRL relies on training an agent with the trajectories of expert demonstrations. In order to do so, the demonstrations need to be decoded by means of the tuples that are learned by the agent. This is done by developing a set of functions that capture the indexes of the states of the expert demonstration and using these to train a simple agent at every step, comparing its trajectories with the ones of the expert.

In this exercise, you will train an agent with the MountainCar-v0 exercise and analyze its performance by plotting the reward versus the training episodes. Follow these steps to complete this exercise:

1. Create a new Jupyter notebook.
2. Import the **`gym`**, **`pylab`**, and **`numpy`** packages:

```
#import packages
import gym
```

```
import pylab
import numpy as np
import os
```

3. Instantiate the parameters based on the number of simulations. The file contains 20 instances, so the state will have 20*20 states. There will be three actions. Set the parameter values for gamma to **0.99**, alpha to **0.03**, and theta to **0.05**:

```
#instantiate parameters
no_states = 400
no_actions = 3
no_st_feature = 20
table = np.zeros((no_states,no_actions))
matrix_features = np.eye((no_states))
gamma_p= 0.99
alpha_p = 0.03
theta_p = 0.05
```

4. Use a random seed; select a random seed number such as **223**:

```
#random seed
np.random.seed(223)
```

5. Create a function for mapping the environmental bounds of the demonstration. This will enable you to identify the position and velocity parameter indexes for the state:

```
#mapping the demo environment
def env_params(env, no_st_feature):
    env_min = env.observation_space.low
    env_max = env.observation_space.high
    dist = (env_max - env_min) / no_st_feature
    return env_min, env_max, dist
```

6. Create a function for identifying the indexes for the position and velocity parameters and return the index of the state:

```
#mapping the index of the state

def params_state(matrix, env_min, dist, no_st_feature):
    pos_indx = int((matrix[0] - env_min[0]) / dist[0])
    vel_indx = int((matrix[1] - env_min[1]) / dist[1])
    state_indx = pos_indx + vel_indx * no_st_feature
    return state_indx
```

7. Create a function for compiling the demonstrations in order to capture the information of the demonstrations in array form. Use the **`Expert demonstration.npy`** file that you created in the *Learning Human Preferences* section:

```
#mapping demonstrations
def demonstration_indx(env, no_st_feature):
    env_min, env_max, dist = env_params(env,no_st_feature)
    demo_load = np.load(file="../Dataset/Expert demonstration.npy")
    demos = np.zeros((len(demo_load),len(demo_load[0]),3))
    for x in range(len(demo_load)):
        for y in range(len(demo_load[0])):
            state_indx = params_state(demo_load[x][y],env_min, dist,
no_st_feature)
            demos[x][y][0] = state_indx
            demos[x][y][1] = demo_load[x][y][2]
    return demos
```

In the preceding code snippet, you are decoding the trajectories from the demonstrations and including the indexes in a matrix of features.

8. Create a function for capturing the index of the state during the simulations:

```
#mapping the index of the simulated states
def attach_state(env, state):
    env_min, env_max, dist = env_params(env, no_st_feature)
    state_indx = params_state(state, env_min, dist,no_st_feature)
    return state_indx
```

9. Create a function for updating the Q-table:

```
def qtable_update(current_state, action, reward,new_state):
    current_q = table[current_state][action]
    new_q = reward + gamma_p * max(table[new_state])
    table[current_state][action] += alpha_p * (new_q - current_q)
```

10. Create a function for getting the trajectories of the demonstrations:

```
def expert_config(matrix_features, demonstrations):
    feat_exp = np.zeros(matrix_features.shape[0])

    for demonstration in demonstrations:
        for state_indx, _, _ in demonstration:
            feat_exp += matrix_features[int(state_indx)]
```

```
    feat_exp /= demonstrations.shape[0]
    return feat_exp
```

In the preceding code snippet, you are using the matrix of features to output the demonstration information in a format that the RL agent can use as a target based on the expert's transitions.

11. Create a function for inverse RL application by calculating the difference between the learned and expert trajectories:

```
def inverse_rl(expert, learner, theta, alpha):
    difference = expert - learner
    theta += alpha * difference
    for j in range(len(theta)):
        if theta[j] > 0:
            theta[j] = 0
```

In the preceding code, you are calculating the differences in features between the RL agent-executed information and the target features of the expert demonstrations, which are used to update the theta parameter.

12. Create a function that will update the reward that was received from the expert demonstrations:

```
def use_reward(matrix_features, theta, no_states, state_indx):
    rewards_demon = matrix_features.dot(theta).reshape((no_states,))
    return rewards_demon[state_indx]
```

In the preceding code snippet, you have to update the array of rewards based on the theta parameter for a specific state.

13. Create a main function for running the **`MountainCar-v0`** environment with IRL. We are doing this to train the agent:

```
def main():
    env = gym.make('MountainCar-v0')
    demonstrations = demonstration_indx(env,no_st_feature)
    expert = expert_config(matrix_features, demonstrations)
    learner_features = np.zeros(no_states)
    theta = -(np.random.uniform(size =(no_states,)))
    episodes, rewards = [], []
```

14. Run the simulation for **3000** episodes:

```
    for episode in range (3000):
        current_state = env.reset()
        score = 0
        if(episode != 0 and episode == 1000) or (episode > 1000 and
episode % 500 == 0):
            learner = learner_features / episode
            inverse_rl(expert, learner, theta, theta_p)
```

In the preceding snippet, you have initialized the state and set a condition to check the score at every **1000** episodes. You updated the learner by dividing the features by the episode and applied inverse RL to compare how close the learned features are to those executed by the expert.

15. Create a **while** loop for updating the state, selecting the action, obtaining the next state and reward, and checking if the simulation is done:

```
        while True:
            state_indx = attach_state(env, current_state)
            action = np.argmax(table[state_indx])
            new_state, reward, done, _ = env.step(action)

            reward_demo = use_reward(matrix_features, theta, no_
states,state_indx)
            new_state_indx = attach_state(env, new_state)
```

16. Update the Q-table with the learned tuples and add the learned features to the learning matrix:

```
qtable_update(state_indx,action,reward_demo,new_state_indx)
            learner_features += matrix_features[int(state_indx)]

            score += reward
            current_state = new_state

            if done:
                rewards.append(score)
                episodes.append(episode)
                break
```

17. At every **500** episodes, output the average reward and a graph displaying the reward that's been accumulated:

```
        if episode % 500 == 0:
            mean = np.mean(rewards)
            print("Average reward for episode {} is {}".
format(episode,mean))
            pylab.plot(episodes, rewards, 'b')
            pylab.show()
            # pylab.savefig("values.png")
            # np.save("q_table", arr = table)
```

In the preceding code snippet, you are aiming to output the average reward per 500 episodes and a plot showcasing the development of the reward as you are running through the episodes.

18. Run the main function:

```
if __name__ == '__main__':
    main()
```

You will get an output similar to the following for every 500 episodes. Note that since you are using your own demonstrations, the results may differ for you:

```
Average reward for episode 2500 is -199.9328268692523
```

The plot can be visualized as follows:

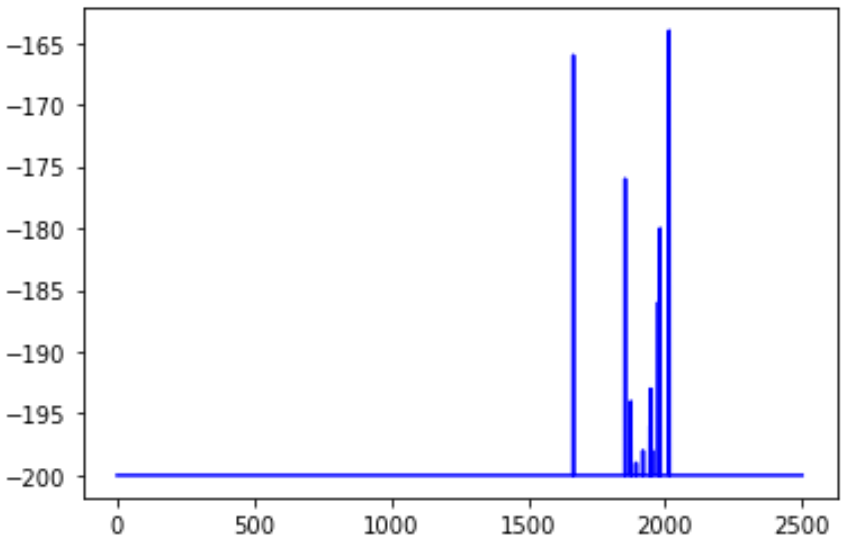

Figure 13.35: Reward per 500 episodes

At every 500 episodes, you will also have a graph displayed that shows the average rewards achieved. With this exercise, you have used trajectories to train an agent using IRL. The performance of the agent is increasing steadily based on the number of running episodes. Since the reward is negative, the performance of the agent is increasing, with a maximum reward being achieved around episode **2000**.

> **NOTE**
>
> To access the source code for this specific section, please refer to https://packt.live/2YxcDXy.
>
> This section does not currently have an online interactive example, and will need to be run locally.

In this exercise, you have been able to analyze and implement the previously created demonstration file in order to train the IRL agent so that it learns from your own demonstrations.

CAUTIONARY NOTES – AI WINTER AND SUPERINTELLIGENCE

There are many advantages to implementing and developing new machine learning techniques, especially in the RL field. One of the most interesting and exciting proposals was the combination of deep neural networks as function approximators for RL agents. This combination and various architectures such as Q-learning implementations, inverse and hierarchical RL are just emerging. Not only are the research field exploring the potential of deep RL strategies, but companies are starting to implement these techniques in various fields. However, as interest and development increases (attracting funding and investments), it can also be marred by failure and a lack of enthusiasm, as was experienced in the 1970s with the first AI winter. This phenomenon was met with criticism and pessimism in the AI field as it was considered obsolete. Even today, most advancements in this technology are just regarded as engineering wonders, despite their contribution to the machine learning field.

The major problem with RL-based architectures is the way in which the agents learn from the environment and humans. Two dangers arise from these learning strategies, as follows:

- The policies that are learned are not in accordance with human expectations since dangerous actions are taken that can harm the environment or other agents.
- The optimization strategies lead to dangerous situations.

Since agents are targeting the highest rewards, more often than not, the actions that are taken can prove risky and unrealistic since humans do not have control over them. These problems can be mitigated by reducing the training of RL agents in real environments and favoring simulation ones instead. The problem with such a choice and using abstract or simplified models for real environments is that the agent learns in an unrealistic environment, therefore using a reduced set of parameters, which means they cannot account for all the states and actions that would be available in the real world, and thus its learning is reduced. Additionally, specifically for demonstration strategies, there is a high reliance on expert programmers or instructors that control and showcase their behavior. There is a risk when mimicking such behavior as it can be erroneous or incomplete, or occur with ill intent.

The risk of implementing such systems in the real world and not bounding the information or their learning is more in the line of science fiction novels, where AI gains consciousness and humanity has to face dire consequences due to the capacity and actions of the AI. Alternatively, as society becomes more and more focused on technology, our reliance on RL agents becomes a comfort, thereby eliminating the capacity of humans to make their own decisions.

Nevertheless, while there are many dangers to be considered, the most important aspect of the RL framework is its capability of learning from both positive and negative reinforcement, the principle of which is based on behavior theory. Since RL systems are extending to other machine learning areas such as neural networks and genetic algorithms that are inspired by nature, the interaction between machines and humans has proved to be a source of knowledge for both the AI research community as well as the behavioral sciences, thus strengthening the bond between multiple academic fields and industries. Instead of fearing discovery, we should explore these advancements and try to better them so that we can better ourselves in the process.

ACTIVITY 13.01: SOLVING MOUNTAINCAR WITH EXPERIENCE REPLAY DQN

In this activity, you will develop a model-based agent to solve the MountainCar-v0 problem, which relies on replayed experience to optimize its trajectories, thus capturing the information within a memory class.

The agent runs for 200 steps and 500 episodes and should use the DQN model. The agent has been trained with negative rewards of -1 for every step that is different than the goal. You will introduce positive rewards for the agent to achieve intermediate states.

The aim of this activity is to implement a deep learning algorithm that uses experience replay and positive rewards to control the movement of a car that crosses a mountain from a valley. By completing this activity, you will be able to create your own classes and evaluate your application and understanding of the potential of model-based RL.

The following steps will help you complete this activity:

1. Create a new Jupyter notebook and import the **`numpy`**, **`matplotlib.pylab`**, **`random`**, **`math`**, and **`gym`** classes (include Keras packages for deep neural networks, that is, **`Sequential`**, **`Dense`**, and **`Adam`**).
2. Create a memory class that includes functions to initialize, append, and generate samples.
3. Create the agent class for initializing the environment memory and the parameters that will be used.
4. Create a function for the deep neural network model.
5. Create a function for action selection.
6. Create a function for updating the Q values based on the batch size.
7. Create a function for updating the weights of the model.
8. Create a main function that runs for 100 episodes and displays the attempts of the agent.

9. Update the reward within the step loop so that the agent receives positive rewards for each sub-goal it achieves.
10. Update the agent, the state, and the reward and collect information about the state and rewards.
11. Plot the state and rewards that were obtained for each episode.

    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:

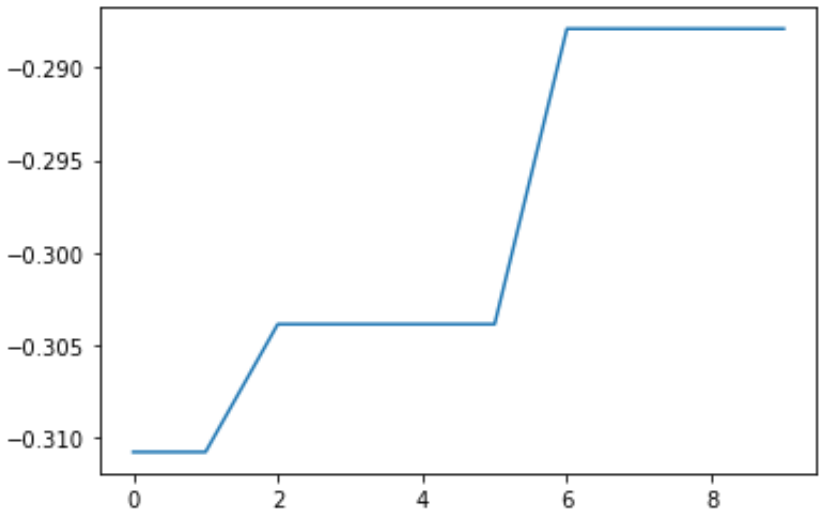

Figure 13.36: Maximum state selection over the number of episodes

In the preceding plot, the *x* axis denotes the number of episodes, whereas the *y* axis denotes the maximum rewards for the respective states, focusing on the selection of states.

The following plot denotes the rewards obtained over the episodes. The *x* axis denotes the range of episodes, whereas the *y* axis denotes the rewards obtained over the states:

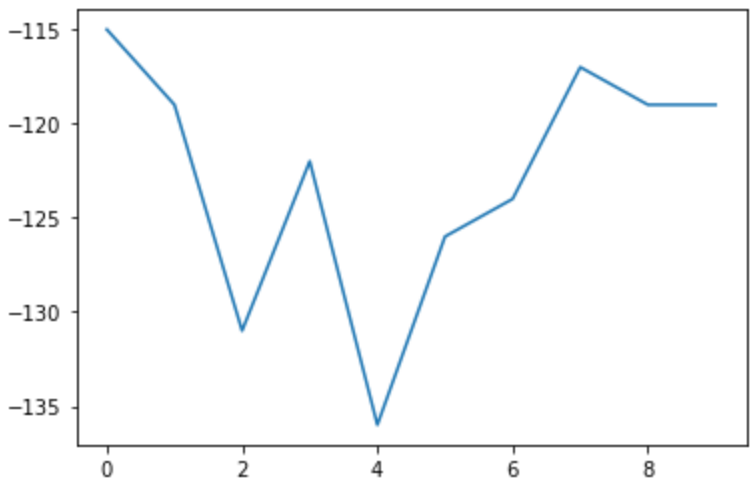

Figure 13.37: Total reward per episode

By completing this activity, you will be able to test your understanding of implementing transfer learning by using experience replay buffers and developing a deep Q-learning-based RL agent. Using this combination, you will observe a direct improvement of the global return of the agent, which will help you solve the MountainCar-v0 problem and succeed in climbing the hill within a short time frame.

> **NOTE**
>
> The solution to this activity can be found here - http://packt.live/30XAFwp.

SUMMARY

In this chapter, you have identified the caveats and problems of transfer learning, especially when using models trained with different data. Though performance may rise, it does not necessarily learn the appropriate information. A solution for when the model transitions are known is to use model-based RL methods for learning an optimal policy based on the reward function. This was implemented on a FrozenLake-v0 exercise using Q-learning. Moreover, you have explored the potential of human preferences with an emphasis on demonstrations. This extended to understanding and creating a class for deep Q-learning from demonstrations. Agents do not rely solely on demonstrations, but also on the buffers that store the past information, thus allowing them to replay their transitions. One such buffer is the HER buffer. The next section examined building hierarchies that have been inspired by feudalism and societal situations in combination with RL algorithms. Finally, a variation of learning from demonstrations by reproducing the transitions directly from a human expert was examined. We concluded this chapter by providing cautionary notes on the continuous development of RL and, most notably, deep learning. Finally, we ended this chapter with an activity on implementing the MountainCar-v0 exercise with a DQN agent that learns from its past experience and has a tailored reward.

With this activity, you have concluded this book on designing, developing, and implementing RL using Python-based algorithms for a variety of environments and applications. You are now capable of applying the concepts that you learned about in this book using various coding techniques and various models that can help further enhance your field of expertise and potentially bring new changes and advancements. Your journey has just begun – you have taken the first steps to deciphering the world of RL, and you now have the tools to enhance your Python programming skills for RL, all of which you can independently apply.