第 3 章

高级强化学习算法的实现

本章提供了简明教程使用 **TensorFlow 2.x** 从头开始实现高级**强化学习**算法和智能体，包括构建 **Deep-Q-Networks（DQN）**、**Double Deep Q-Networks（DDQN）**、**Double Dueling Deep Q-Networks（DDDQN）**、**Deep Recurrent Q-Neworks（DRQN）**、**Asynchronous Advantage Actor-Critic（A3C）**、**Proximal Policy Optimization（PPO）**以及 **Deep Deterministic Policy Gradients（DDPG）**的方法。

在本章中将会讨论以下内容：

- 实现 Deep Q 学习算法、DQN 和 Double-DQN 智能体；
- 实现 Dueling DQN 智能体；
- 实现 Dueling Double DQN 算法和 DDDQN 智能体；
- 实现深度递归 Q 学习算法和 DRQN 智能体；
- 实现异步优势行动者-评论家算法和 A3C 智能体；
- 实现近端策略优化算法和 PPO 智能体；
- 实现深度确定性策略梯度算法和 DDPG 智能体。

3.1 技术要求

本书的代码已经在 Ubuntu 18.04 和 Ubuntu 20.04 上进行了广泛的测试，而且可以在安装了 Python 3.6+ 的 Ubuntu 后续版本中正常工作。在安装 Python 3.6 的情况下，搭配每项内容开始时列出的必要 Python 工具包，本书的代码也同样可以在 Windows 和 macOS X 上运行。建议读者创建和使用一个命名为 tf2rl-cookbook 的 Python 虚拟环境来安装工具包以及运行本书的代码。推荐读者安装 Miniconda 或 Anaconda 来管理 Python 虚拟环境。

3.2 实现 Deep Q 学习算法、DQN 和 Double-DQN 智能体

DQN 智能体采用深度神经网络来学习 Q 值函数。DQN 是一种针对离散动作空间环境和问题的强有力的算法，并当在 Atari 游戏中取得了成功时，DQN 成为了深度强化学习

历史上的一个重要的里程碑。

Double-DQN 智能体使用了两种相同的深度神经网络，它们的更新方式不同，因此也具有不同的权重。第二个神经网络是之前某个时间（通常从上一个回合开始）的主神经网络的副本。

通过本节，可以使用 TensorFlow 2.x 从头开始实现一个完整的 DQN 和 Double-DQN 智能体，该智能体能够在任何离散动作空间的强化学习环境中进行训练。

3.2.1 前期准备

为了完成本节内容，需要激活命名为 tf2rl-cookbook 的 Python/Conda 虚拟环境并在命令行运行 pip install -r requirements.txt。如果运行下面的导入语句时没有出现问题，就可以准备开始了：

```
import argparse
from datetime import datetime
import os
import random
from collections import deque

import gym
import numpy as np
import tensorflow as tf
from tensorflow.keras.layers import Dense, Input
```

3.2.2 实现步骤

DQN 智能体包含以下部分，即**回放缓冲区**、DQN 类、Agent 类和 train() 函数。使用 TensorFlow 2.x 执行以下步骤从头开始实现上述每个部分，从而构建完整的 DQN 智能体。

（1）创建一个参数解析器处理脚本的配置输入：

```
parser = argparse.ArgumentParser(prog="TFRL-Cookbook-Ch3-DQN")
parser.add_argument("--env , default="CartPole-v0")
parser.add_argument("--lr", type=float, default=0.005)
parser.add_argument("--batch_size", type=int, default=256)
parser.add_argument("--gamma", type=float, default=0.95)
parser.add_argument("--eps", type=float, default=1.0)
parser.add_argument("--eps_decay", type=float, default=0.995)
parser.add_argument("--eps_min", type=float, default=0.01)
parser.add_argument("--logdir", default="logs")
args = parser.parse_args()
```

（2）创建一个 Tensorboard 日志，记录智能体在训练时的有用统计信息：

```
logdir = os.path.join(
  args.logdir, parser.prog, args.env,
  datetime.now().strftime("%Y%m%d-%H%M%S")
)
print(f"Saving training logs to:{logdir}")
writer = tf.summary.create_file_writer(logdir)
```

（3）实现一个 ReplayBuffer 类：

```
class ReplayBuffer:
  def __init__(self, capacity=10000):
    self.buffer = deque(maxlen=capacity)

  def store(self, state, action, reward, next_state,
  done):
    self.buffer.append([state, action, reward,
    next_state, done])

  def sample(self):
    sample = random.sample(self.buffer,
    args.batch_size)
    states, actions, rewards, next_states, done = \
    map(np.asarray, zip(*sample))
    states = np.array(states).reshape(
    args.batch_size, -1)
    next_states = np.array(next_states).\
    reshape(args.batch_size, -1)
    return states, actions, rewards, next_states,
    done

  def size(self):
    return len(self.buffer)
```

（4）使用 TensorFlow 2.x 定义深度神经网络的 DQN 类：

```
class DQN:
  def __init__(self, state_dim, aciton_dim):
    self.state_dim = state_dim
    self.action_dim = aciton_dim
    self.epsilon = args.eps

    self.model = self.nn_model()
```

```
  def nn_model(self):
    model = tf.keras.Sequential(
      [
        Input((self.state_dim,)),
        Dense(32, activation="relu"),
        Dense(16, activation="relu"),
        Dense(self.action_dim),
      ]
    )
    model.compile(loss="mse",
            optimizer=Adam(args.lr))
    return model
```

（5）为了从 DQN 中获得预测和动作，实现 predict() 函数和 get_ action() 函数：

```
def predict(self, state):
  return self.model.predict(state)

def get_action(self, state):
  state = np.reshape(state, [1, self.state_dim])
  self.epsilon *= args.eps_decay
  self.epsilon = max(self.epsilon, args.eps_min)
  q_value = self.predict(state)[0]
  if np.random.random() < self.epsilon:
    return random.randint(0, self.action_dim - 1)
  return np.argmax(q_value)
def train(self, states, targets):
  self.model.fit(states, targets, epochs=1)
```

（6）当其他部分实现后，就开始实现 Agent 类：

```
class Agent:
  def __init__(self, env):
    self.env = env
    self.state_dim = \
    self.env.observation_space.shape[0]
    self.action_dim = self.env.action_space.n

    self.model = DQN(self.state_dim, self.action_dim)
    self.target_model = DQN(self.state_dim,
              self.action_dim)
    self.update_target()

    self.buffer = ReplayBuffer()
```

```
def update_target(self):
  weights = self.model.model.get_weights()
  self.target_model.model.set_weights(weights)
```

（7）Deep Q-学习算法的关键是 Q 学习的更新和经验回放（experience replay）：

```
def replay_experience(self):
  for _ in range(10):
    states, actions, rewards, next_states, done=\
      self.buffer.sample()
    targets = self.target_model.predict(states)
    next_q_values = self.target_model.\
            predict(next_states).max(axis=1)
    targets[range(args.batch_size), actions] = (
      rewards + (1 - done) * next_q_values * \
      args.gamma
    )
    self.model.train(states, targets)
```

（8）这是一个关键步骤，即实现 train() 函数训练智能体：

```
def train(self, max_episodes=1000):
  with writer.as_default(): # Tensorboard logging
    for ep in range(max_episodes):
      done, episode_reward = False, 0
      observation = self.env.reset()
      while not done:
        action = \
          self.model.get_action(observation)
        next_observation, reward, done, _ = \
          self.env.step(action)
        self.buffer.store(
          observation, action, reward * \
          0.01, next_observation, done
        )
        episode_reward += reward
        observation = next_observation
      if self.buffer.size() >= args.batch_size:
        self.replay_experience()
      self.update_target()
      print(f"Episode#{ep} Reward:{
                  episode_reward}")
      tf.summary.scalar("episode_reward",
```

```
                episode_reward, step=ep)
        writer.flush()
```

（9）创建主函数并开始训练智能体：

```
if __name__ == "__main__":
  env = gym.make("CartPole-v0")
  agent = Agent(env)
  agent.train(max_episodes=20000)
```

（10）可以执行以下命令在默认环境（CartPole-v0）中训练 DQN 智能体：

```
python ch3-deep-rl-agents/1_dqn.py
```

（11）也可以使用以下命令行参数在任何 OpenAI Gym 兼容的离散动作空间的环境中训练 DQN 智能体：

```
python ch3-deep-rl-agents/1_dqn.py --env "MountainCar-v0"
```

（12）实现 Double DQN 智能体，修改 replay_ experience() 函数以使用 Double Q 学习的更新步骤，如下所示：

```
  def replay_experience(self):
    for _ in range(10):
      states, actions, rewards, next_states, done=\
        self.buffer.sample()
      targets = self.target_model.predict(states)
      next_q_values = \
        self.target_model.predict(next_states)[
        range(args.batch_size),
        np.argmax(self.model.predict(
                    next_states), axis=1),
      ]
      targets[range(args.batch_size), actions] = (
        rewards + (1 - done) * next_q_values * \
          args.gamma
      )
      self.model.train(states, targets)
```

（13）训练 Double DQN 智能体，可以保存并运行使用了更新后的 replay_experience() 函数的脚本，也可以使用本书提供的源码脚本：

```
python ch3-deep-rl-agents/1_double_dqn.py
```

3.2.3　工作原理

根据下式对 DQN 中的权重进行更新：

$$\Delta w = \alpha[R + r\underbrace{\max_a \hat{Q}(s', a; w)}_{s' \text{ 的最大 } Q \text{ 值}} - \underbrace{\hat{Q}(s, a; w)}_{\text{预测 } Q \text{ 值}}]\underbrace{\nabla w \hat{Q}(s, a; w)}_{Q \text{ 值的梯度}}$$

其中，Δw 是 DQN 的参数（权重）的变化量，s 是当前状态，a 是当前动作，s' 是下一状态，w 表示 DQN 的权重，γ 是折扣因子，α 是学习率。$\hat{Q}(s, a; w)$ 表示权重为 w 的 DQN 网络预测的给定状态 (s) 和动作 (a) 的 Q 值。

为了理解 DQN 智能体和 Double-DQN 智能体之间的区别，比较第（8）步（DQN）和第（12）步（Double DQN）中的 replay_ experience() 方法，可以发现关键的不同在于计算 next_ q_ values。DQN 智能体使用了所预测 Q 值的最大值（可能会高估），而 Double DQN 智能体使用两个不同的神经网络预测的 Q 值，以避免出现 DQN 智能体高估 Q 值的问题。

3.3　实现 Dueling DQN 智能体

Dueling DQN 智能体通过修改后的网络结构显式估计两个量：

（1）状态值 $V(s)$；

（2）优势值 $A(s, a)$。

状态值估计状态 s 的价值，而优势值表示在状态 s 中采取动作 a 的优势。这种将两个量进行显式和单独估计的关键思想使 Dueling DQN 的性能优于 DQN。本节将引导读者使用 TensorFlow 2.x 从头开始实现一个 Dueling DQN 智能体。

3.3.1　前期准备

为了完成本节内容，需要激活命名为 tf2rl-cookbook 的 Python/Conda 虚拟环境并在命令行运行 pip install -r requirements.txt。如果运行下面的导入语句时没有出现问题，就可以准备开始了：

```
import argparse
import os
import random
from collections import deque
from datetime import datetime

import gym
import numpy as np
import tensorflow as tf
from tensorflow.keras.layers import Add, Dense, Input
```

```
from tensorflow.keras.optimizers import Adam
```

3.3.2 实现步骤

Dueling DQN 智能体包含以下内容，即**回放缓冲区**、DuelingDQN() 类、Agent() 类和 train() 函数。使用 TensorFlow 2.x 执行以下步骤从头开始实现上述每个部分，从而构建完整的 Dueling DQN 智能体。

（1）创建一个参数解析器处理对脚本的命令行配置输入：

```
parser = argparse.ArgumentParser(prog="TFRL-Cookbook-Ch3-DuelingDQN")
parser.add_argument("--env", default="CartPole-v0")
parser.add_argument("--lr", type=float, default=0.005)
parser.add_argument("--batch_size", type=int, default=64)
parser.add_argument("--gamma", type=float, default=0.95)
parser.add_argument("--eps", type=float, default=1.0)
parser.add_argument("--eps_decay", type=float, default=0.995)
parser.add_argument("--eps_min", type=float, default=0.01)
parser.add_argument("--logdir", default="logs")

args = parser.parse_args()
```

（2）创建一个 Tensorboard 日志，记录智能体在训练时的有用统计信息：

```
logdir = os.path.join(
  args.logdir, parser.prog, args.env,
  datetime.now().strftime("%Y%m%d-%H%M%S")
)
print(f"Saving training logs to:{logdir}")
writer = tf.summary.create_file_writer(logdir)
```

（3）实现一个 ReplayBuffer 类：

```
class ReplayBuffer:
  def __init__(self, capacity=10000):
    self.buffer = deque(maxlen=capacity)

  def store(self, state, action, reward, next_state,
  done):
    self.buffer.append([state, action, reward,
              next_state, done])

  def sample(self):
    sample = random.sample(self.buffer,
                args.batch_size)
```

```
    states, actions, rewards, next_states, done = \
              map(np.asarray, zip(*sample))
    states = np.array(states).reshape(
                  args.batch_size, -1)
    next_states = np.array(next_states).reshape(
                  args.batch_size, -1)
    return states, actions, rewards, next_states,
    done
  def size(self):
    return len(self.buffer)
```

（4）使用 TensorFlow 2.x 定义深度神经网络的 DuelingDQN 类：

```
class DuelingDQN:
  def __init__(self, state_dim, aciton_dim):
    self.state_dim = state_dim
    self.action_dim = aciton_dim
    self.epsilon = args.eps

    self.model = self.nn_model()

  def nn_model(self):
    backbone = tf.keras.Sequential(
      [
        Input((self.state_dim,)),
        Dense(32, activation="relu"),
        Dense(16, activation="relu"),
      ]
    )
    state_input = Input((self.state_dim,))
    backbone_1 = Dense(32, activation="relu")\
              (state_input)
    backbone_2 = Dense(16, activation="relu")\
              (backbone_1)
    value_output = Dense(1)(backbone_2)
    advantage_output = Dense(self.action_dim)\
                (backbone_2)
    output = Add()([value_output, advantage_output])
    model = tf.keras.Model(state_input, output)
    model.compile(loss="mse",
            optimizer=Adam(args.lr))
    return model
```

（5）为了从 Dueling DQN 中获得预测和动作，实现 predict() 函数和 get_action() 函数以及 train() 函数：

```
  def predict(self, state):
    return self.model.predict(state)

  def get_action(self, state):
    state = np.reshape(state, [1, self.state_dim])
    self.epsilon *= args.eps_decay
    self.epsilon = max(self.epsilon, args.eps_min)
    q_value = self.predict(state)[0]
    if np.random.random() < self.epsilon:
      return random.randint(0, self.action_dim - 1)
    return np.argmax(q_value)

  def train(self, states, targets):
    self.model.fit(states, targets, epochs=1)
```

（6）实现 Agent 类：

```
class Agent:
  def __init__(self, env):
    self.env = env
    self.state_dim = \
      self.env.observation_space.shape[0]
    self.action_dim = self.env.action_space.n

    self.model = DuelingDQN(self.state_dim,
                self.action_dim)
    self.target_model = DuelingDQN(self.state_dim,
                    self.action_dim)
    self.update_target()

    self.buffer = ReplayBuffer()

  def update_target(self):
    weights = self.model.model.get_weights()
    self.target_model.model.set_weights(weights)
```

（7）Dueling Deep Q 学习算法的关键是 Q 学习的更新和经验回放：

```
  def replay_experience(self):
    for _ in range(10):
      states, actions, rewards, next_states, done=\
```

```
        self.buffer.sample()
      targets = self.target_model.predict(states)
      next_q_values = self.target_model.\
            predict(next_states).max(axis=1)
      targets[range(args.batch_size), actions] = (
        rewards + (1 - done) * next_q_values * \
        args.gamma
      )
      self.model.train(states, targets)
```

（8）实现 train() 函数训练智能体：

```
  def train(self, max_episodes=1000):
    with writer.as_default():
      for ep in range(max_episodes):
        done, episode_reward = False, 0
        state = self.env.reset()
        while not done:
          action = self.model.get_action(state)
          next_state, reward, done, _ = \
                  self.env.step(action)
          self.buffer.put(state, action, \
                  reward * 0.01, \
                  next_state, done)
          episode_reward += reward
          state = next_state

        if self.buffer.size() >= args.batch_size:
          self.replay_experience()
        self.update_target()
        print(f"Episode#{ep} \
            Reward:{episode_reward}")
        tf.summary.scalar("episode_reward",\
                episode_reward, step=ep)
```

（9）创建主函数并训练智能体：

```
if __name__ == "__main__":
  env = gym.make("CartPole-v0")
  agent = Agent(env)
  agent.train(max_episodes=20000)
```

（10）可以执行以下命令在默认环境（CartPole-v0）中训练 Dueling DQN 智能体：

```
python ch3-deep-rl-agents/2_dueling_dqn.py
```

（11）也可以使用以下命令行参数在任何 OpenAI Gym 兼容的离散动作空间环境中训练 DQN 智能体：

```
python ch3-deep-rl-agents/2_dueling_dqn.py --env "MountainCar-v0"
```

3.3.3 工作原理

Dueling-DQN 智能体和 DQN 智能体的区别在于神经网络的结构。图 3.1 总结了这些区别。

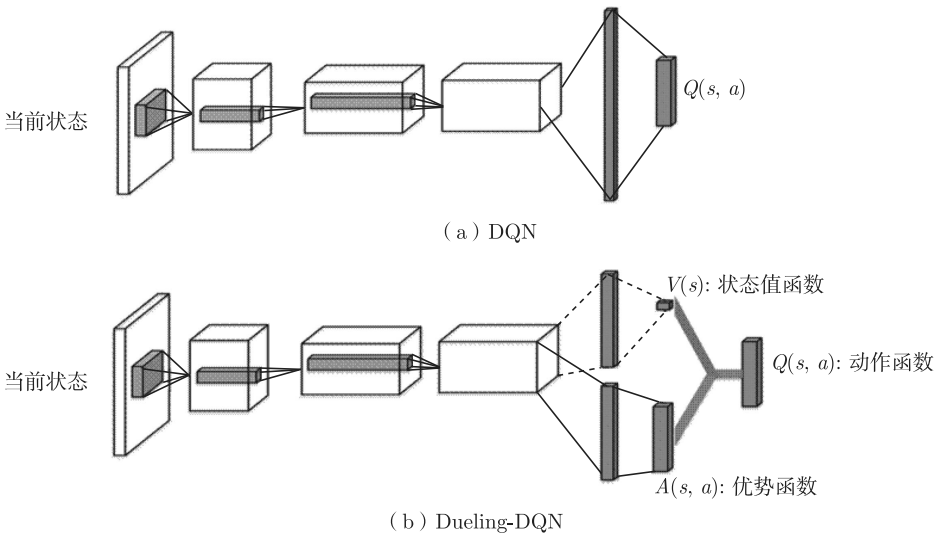

图 3.1 DQN 和 Dueling-DQN 的对比

图 3.1（a）中所示的 DQN 具有线性体系结构，并预测单个数量 $Q(s,a)$，而 Dueling-DQN 在最后一层具有分叉结构，可以预测多个量。

3.4 实现 Dueling Double DQN 算法和 DDDQN 智能体

DDDQN 结合了 Double Q 学习和 Dueling 结构的优点。Double Q 学习可以通过纠正高估动作值来改进 DQN。Dueling 结构使用修改后的神经网络结构分别学习状态值函数（V）和优势函数（A）。这样一种明确的分离使算法可以学习得更快，特别是当有很多动作可供选择以及这些动作彼此非常相似时。与 DQN 智能体不同，Dueling 结构使智能体即使在某个状态中仅执行过一个动作时也可以学习，因为它可以更新和估计状态值函数，而 DQN 智能体无法从尚未采取的动作中学习。在本书的最后，读者将实现一个完整的 DDDQN 智能体。

3.4.1　前期准备

为了完成本节内容，需要激活命名为 tf2rl-cookbook 的 Python/Conda 虚拟环境并在命令行运行 pip install -r requirements.txt。如果运行下面的导入语句时没有出现问题，就可以准备开始了：

```
import argparse
from datetime import datetime
import os
import random
from collections import deque

import gym
import numpy as np
import tensorflow as tf
from tensorflow.keras.layers import Add, Dense, Input
from tensorflow.keras.optimizers import Adam
```

3.4.2　实现步骤

DDDQN 智能体结合了 DQN、Double DQN 和 Dueling DQN 三者的思想。使用 TensorFlow 2.x 执行以下步骤从头开始实现上述每个部分，从而构建完整的 Dueling Double DQN 智能体。

（1）创建一个参数解析器处理对脚本的命令行配置输入：

```
parser = argparse.ArgumentParser(prog="TFRL-Cookbook-Ch3-DuelingDoubleDQN"
    )
parser.add_argument("--env", default="CartPole-v0")
parser.add_argument("--lr", type=float, default=0.005)
parser.add_argument("--batch_size", type=int, default=256)
parser.add_argument("--gamma", type=float, default=0.95)
parser.add_argument("--eps", type=float, default=1.0)
parser.add_argument("--eps_decay", type=float, default=0.995)
parser.add_argument("--eps_min", type=float, default=0.01)
parser.add_argument("--logdir", default="logs")

args = parser.parse_args()
```

（2）创建 Tensorboard 日志，记录智能体在训练时的有用统计信息：

```
logdir = os.path.join(
  args.logdir, parser.prog, args.env, \
  datetime.now().strftime("%Y%m%d-%H%M%S")
)
```

```
print(f"Saving training logs to:{logdir}")
writer = tf.summary.create_file_writer(logdir)
```

（3）实现一个 ReplayBuffer 类：

```
class ReplayBuffer:
  def __init__(self, capacity=10000):
    self.buffer = deque(maxlen=capacity)

  def store(self, state, action, reward, next_state, done):
    self.buffer.append([state, action, reward, \
    next_state, done])

  def sample(self):
    sample = random.sample(self.buffer, \
                args.batch_size)
    states, actions, rewards, next_states, done = \
              map(np.asarray, zip(*sample))
    states = np.array(states).reshape(
                    args.batch_size, -1)
    next_states = np.array(next_states).\
                reshape(args.batch_size, -1)
    return states, actions, rewards, next_states, \
    done

  def size(self):
    return len(self.buffer)
```

（4）实现 Dueling DQN 类，该类根据 Dueling 结构定义神经网络，在以后的步骤中向其中添加 Double DQN 的更新：

```
class DuelingDQN:
  def __init__(self, state_dim, aciton_dim):
    self.state_dim = state_dim
    self.action_dim = aciton_dim
    self.epsilon = args.eps

    self.model = self.nn_model()

  def nn_model(self):
    state_input = Input((self.state_dim,))
    fc1 = Dense(32, activation="relu")(state_input)
    fc2 = Dense(16, activation="relu")(fc1)
    value_output = Dense(1)(fc2)
```

```
    advantage_output = Dense(self.action_dim)(fc2)
    output = Add()([value_output, advantage_output])
    model = tf.keras.Model(state_input, output)
    model.compile(loss="mse", \
          optimizer=Adam(args.lr))
    return model
```

（5）为了从 Dueling DQN 中获得预测和动作，实现 predict() 函数和 get_action() 函数：

```
  def predict(self, state):
    return self.model.predict(state)

  def get_action(self, state):
    state = np.reshape(state, [1, self.state_dim])
    self.epsilon *= args.eps_decay
    self.epsilon = max(self.epsilon, args.eps_min)
    q_value = self.predict(state)[0]
    if np.random.random() < self.epsilon:
      return random.randint(0, self.action_dim - 1)
    return np.argmax(q_value)
  def train(self, states, targets):
    self.model.fit(states, targets, epochs=1)
`~\\`
```

（6）实现 Agent 类：

```
class Agent:
  def __init__(self, env):
    self.env = env
    self.state_dim = \
      self.env.observation_space.shape[0]
    self.action_dim = self.env.action_space.n

    self.model = DuelingDQN(self.state_dim,
                self.action_dim)
    self.target_model = DuelingDQN(self.state_dim,
                  self.action_dim)
    self.update_target()

    self.buffer = ReplayBuffer()

  def update_target(self):
    weights = self.model.model.get_weights()
```

```
    self.target_model.model.set_weights(weights)
```

（7）Dueling Double Deep Q学习算法最主要的部分是 Q 学习的更新和经验回放：

```
def replay_experience(self):
  for _ in range(10):
    states, actions, rewards, next_states, done=\
              self.buffer.sample()
    targets = self.target_model.predict(states)
    next_q_values = \
      self.target_model.predict(next_states)[
      range(args.batch_size),
      np.argmax(self.model.predict(
                  next_states), axis=1),
    ]
    targets[range(args.batch_size), actions] = (
      rewards + (1 - done) * next_q_values * \
      args.gamma
    )
    self.model.train(states, targets)
```

（8）实现 train() 函数训练智能体：

```
def train(self, max_episodes=1000):
  with writer.as_default():
    for ep in range(max_episodes):
      done, episode_reward = False, 0
      observation = self.env.reset()
      while not done:
        action = \
          self.model.get_action(observation)
        next_observation, reward, done, _ = \
          self.env.step(action)
        self.buffer.store(
          observation, action, reward * \
            0.01, next_observation, done
        )
        episode_reward += reward
        observation = next_observation

      if self.buffer.size() >= args.batch_size:
        self.replay_experience()
      self.update_target()
      print(f"Episode#{ep} \
```

```
            Reward:{episode_reward}")
        tf.summary.scalar("episode_reward",
                          episode_reward,
                          step=ep)
```

（9）创建主函数并训练智能体：

```
if __name__ == "__main__":
  env = gym.make("CartPole-v0")
  agent = Agent(env)
  agent.train(max_episodes=20000)
```

（10）可以执行以下命令在默认环境（CartPole-v0）中训练 DQN 智能体：

```
python ch3-deep-rl-agents/3_dueling_double_dqn.py
```

（11）也可以使用以下命令行参数在任何 OpenAI Gym 兼容的离散动作空间环境中训练 Dueling Double DQN 智能体：

```
python ch3-deep-rl-agents/3_dueling_double_dqn.py  - env "MountainCar-v0"
```

3.4.3　工作原理

Dueling Double DQN 结构将 Double DQN 和 Dueling 结构的优势结合在一起。

3.5　实现深度递归 Q 学习算法和 DRQN 智能体

DRQN 使用递归神经网络学习 Q 值函数。DRQN 更适合在具有部分可观测性的环境中进行强化学习。DRQN 中的循环网络层允许智能体通过整合来自观测的时间序列的信息来学习。例如，DRQN 智能体可以推断环境中移动对象的速度，而无须对其输入进行任何更改（例如，不需要帧堆叠）。通过本节，可以实现一个完整的 DRQN 智能体，该智能体随时可以在选择的强化学习环境中进行训练。

3.5.1　前期准备

为了完成本节内容，需要激活命名为 tf2rl-cookbook 的 Python/Conda 虚拟环境并在命令行运行 pip install -r requirements.txt。如果运行下面的导入语句时没有出现问题，就可以准备开始了：

```
import tensorflow as tf
from datetime import datetime
import os
from tensorflow.keras.layers import Input, Dense, LSTM
from tensorflow.keras.optimizers import Adam
```

```
import gym
import argparse
import numpy as np
from collections import deque
import random
```

3.5.2 实现步骤

使用 TensorFlow 2.x 执行以下步骤从头开始实现上述每个部分，从而构建完整的 DRQN 智能体。

（1）创建一个参数解析器处理脚本的命令行配置输入：

```
parser = argparse.ArgumentParser(prog="TFRL-Cookbook-Ch3-DRQN")
parser.add_argument("--env", default="CartPole-v0")
parser.add_argument("--lr", type=float, default=0.005)
parser.add_argument("--batch_size", type=int, default=64)
parser.add_argument("--time_steps", type=int, default=4)
parser.add_argument("--gamma", type=float, default=0.95)
parser.add_argument("--eps", type=float, default=1.0)
parser.add_argument("--eps_decay", type=float, default=0.995)
parser.add_argument("--eps_min", type=float, default=0.01)
parser.add_argument("--logdir", default="logs")
args = parser.parse_args()
```

（2）创建 TensorBoard 日志，记录智能体在训练时的有用统计信息：

```
logdir = os.path.join(
  args.logdir, parser.prog, args.env, \
  datetime.now().strftime("%Y%m%d-%H%M%S")
)
print(f"Saving training logs to:{logdir}")
writer = tf.summary.create_file_writer(logdir)
```

（3）实现 ReplayBuffer 类：

```
class ReplayBuffer:
  def __init__(self, capacity=10000):
    self.buffer = deque(maxlen=capacity)

  def store(self, state, action, reward, next_state,\
  done):
    self.buffer.append([state, action, reward, \
              next_state, done])
```

```
  def sample(self):
    sample = random.sample(self.buffer,
              args.batch_size)
    states, actions, rewards, next_states, done = \
      map(np.asarray, zip(*sample))
    states = np.array(states).reshape(
                  args.batch_size, -1)
    next_states = np.array(next_states).reshape(
                  args.batch_size, -1)
    return states, actions, rewards, next_states, \
    done

  def size(self):
    return len(self.buffer)
```

（4）使用 TensorFlow 2.x 定义深度神经网络的 DRQN 类：

```
class DRQN:
  def __init__(self, state_dim, action_dim):
    self.state_dim = state_dim
    self.action_dim = action_dim
    self.epsilon = args.eps

    self.opt = Adam(args.lr)
    self.compute_loss = \
      tf.keras.losses.MeanSquaredError()
    self.model = self.nn_model()

  def nn_model(self):
    return tf.keras.Sequential(
      [
        Input((args.time_steps, self.state_dim)),
        LSTM(32, activation="tanh"),
        Dense(16, activation="relu"),
        Dense(self.action_dim),
      ]
    )
```

（5）为了从 DRQN 获得预测和动作，实现 predict() 函数和 get_action() 函数：

```
  def predict(self, state):
    return self.model.predict(state)

  def get_action(self, state):
```

```
    state = np.reshape(state, [1, args.time_steps,
                    self.state_dim])
    self.epsilon *= args.eps_decay
    self.epsilon = max(self.epsilon, args.eps_min)
    q_value = self.predict(state)[0]
    if np.random.random() < self.epsilon:
      return random.randint(0, self.action_dim - 1)
    return np.argmax(q_value)

  def train(self, states, targets):
    targets = tf.stop_gradient(targets)
    with tf.GradientTape() as tape:
      logits = self.model(states, training=True)
      assert targets.shape == logits.shape
      loss = self.compute_loss(targets, logits)
    grads = tape.gradient(loss,
              self.model.trainable_variables)
    self.opt.apply_gradients(zip(grads,
             self.model.trainable_variables))
```

（6）实现 Agent 类：

```
class Agent:
  def __init__(self, env):
    self.env = env
    self.state_dim = \
      self.env.observation_space.shape[0]
    self.action_dim = self.env.action_space.n

    self.states = np.zeros([args.time_steps,
                self.state_dim])

    self.model = DRQN(self.state_dim,
            self.action_dim)
    self.target_model = DRQN(self.state_dim,
                self.action_dim)
    self.update_target()

    self.buffer = ReplayBuffer()

  def update_target(self):
    weights = self.model.model.get_weights()
    self.target_model.model.set_weights(weights)
```