Project B 第三阶段实验报告

利用Q-learning 训练CartPole模型

前言：

在本阶段，打算利用强化学习的方法对CartPole进行训练，目前学习的强化学习模型有policy gradient，Q-learning和Deep-Q-learning（DQN），由于第一个模型书上已给出示例，因此选用后两个模型进行实验。

一、Q-learning

1.1模型介绍

Q-learning基于马尔可夫决策链产生，其核心思想为如下公式：

Equation 16-5. Q-Learning algorithm

$$Q_{k+1}(s,a) \leftarrow (1-\alpha)Q_k(s,a) + \alpha\left(r + \gamma \cdot \max_{a'} Q_k(s',a')\right)$$

对于CartPole模型来说，如果有如下一个表，在agent的任意一个状态（state）下，对于每一个动作（action 为0或1）都有一个价值（value）给出，那么就可以通过查询表格选择产生最大价值的动作。

State	Action 0 （left）	Action 1（right）
[pos, v, angle, ang_v]		

游戏的最终目标是小车移动200次后立杆不倒，游戏结束前的每一次移动都为最终的目标的实现贡献了价值，因此每一个动作的价值计算，和最终的结果有关。

本实验中使用Q-learning的目的即是创建Q-Table，有了表格后，自然可以知道在任一状态下如何选择下一步的行动。

1.2模型实现

初始化表格

```
from collections import defaultdict
import pickle # 保存模型用
import gym
import numpy as np

# 默认将Action 0,1的价值初始化为0
Q = defaultdict(lambda: [0, 0])
```

连续值离散化

本实验中运用Q-learning的最大问题在于，Q-learning的状态需要是离散值，而当前模型中状态的四个参数（位置，速度，角度，角速度）均为连续值，因此需要用对连续值进行归一化处理，将state的值映射到某有限空间中。

```
: print(env.observation_space.high)
```

```
[4.8000002e+00 3.4028235e+38 4.1887903e-01 3.4028235e+38]
```

```
: print(env.observation_space.low)
```

```
[-4.8000002e+00 -3.4028235e+38 -4.1887903e-01 -3.4028235e
+38]
```

模型中状态的四个参数的上下界如上图所示，可以发现各参数的绝对值值均小于5，因此考虑使用如下函数对state进行转换，将其限制在0-50的区间内：

```
env = gym.make('CartPole-v0')

def transform_state(state):
    """将 position, velocity, angle, angular velovity 通过线性转换映射到 [0, 40]
范围内"""
    pos, v, angle, ang_v= state
    pos_low, v_low, angle_low, ang_v_low= env.observation_space.low
    pos_high, v_high,angle_high, ang_v_high = env.observation_space.high

    ad_pos= 50 * (pos-pos_low)/(pos_high-pos_low)
    ad_v =  50* (v-v_low)/(v_high-v_low)
    ad_angle = 50 * (angle-angle_low)/(angle_high-angle_low)
    ad_ang_v = 50*(ang_v-ang_v_low)/(ang_v_high-ang_v_low)

    return int(ad_pos), int(ad_v),int(ad_angle),int(ad_ang_v)
```

填充Q-table

在对state进行转换后，需要利用Q-learning的核心公式对表格进行填充

Equation 16-5. Q-Learning algorithm

$$Q_{k+1}(s,a) \leftarrow (1-\alpha)Q_k(s,a) + \alpha\left(r + \gamma \cdot \max_{a'} Q_k(s',a')\right)$$

即如下公式：

```
Q[s][a] = (1 - lr) * Q[s][a] + lr * (reward + discount_factor *
max(Q[next_s]))
#s, a, next_s:为当前状态，当前动作（0或者1），下一个状态
#reward:执行a动作可以获得的奖励
#Q[s][a]：状态s下a动作可以产生的价值
#lr：learning_rate
#discount_factor：衰减因子
```

开始训练

训练思路为，设置learning_rate和discount_factor，分别为0.7，0.95

先训练100000次，每一次训练中先用transform_state函数对状态s进行离散化，清空分数，使用ε-greedy policy的方式选择一个动作，执行动作并将状态传给next_s。根据公式更新Q-Table，并开启下一轮循环。

当游戏结束，则输出第n次训练中的得分，以及当前训练中的最高得分。

```
lr, discount_factor = 0.7, 0.95
episodes = 10000  # 训练10000次
score_list = []  # 记录所有分数
for i in range(episodes):
    s = transform_state(env.reset())
    score = 0
    while True:
        a = np.argmax(Q[s])
        # 实现ε-greedy policy
        if np.random.random() > i * 3 / episodes:
            a = np.random.choice([0, 1])
        # 执行动作
        next_s, reward, done, _ = env.step(a)
        next_s = transform_state(next_s)
        # 根据上面的公式更新Q-Table
        Q[s][a] = (1 - lr) * Q[s][a] + lr * (reward + factor * max(Q[next_s]))
        score += reward
        s = next_s
        if done:
            score_list.append(score)
            print('episode:', i, 'score:', score, 'max:', max(score_list))
            break
env.close()
```

运行

运行后发现训练结果较不理想，十万次训练结束后，尽管max_score达到200，但score无法稳定在200处，基本处在100上下，遂分析算法，查找原因。

```
episode: 99985 score: 94.0 max: 200.0
episode: 99986 score: 102.0 max: 200.0
episode: 99987 score: 117.0 max: 200.0
episode: 99988 score: 200.0 max: 200.0
episode: 99989 score: 103.0 max: 200.0
episode: 99990 score: 106.0 max: 200.0
episode: 99991 score: 99.0 max: 200.0
episode: 99992 score: 92.0 max: 200.0
episode: 99993 score: 105.0 max: 200.0
episode: 99994 score: 102.0 max: 200.0
episode: 99995 score: 187.0 max: 200.0
episode: 99996 score: 97.0 max: 200.0
episode: 99997 score: 200.0 max: 200.0
episode: 99998 score: 99.0 max: 200.0
episode: 99999 score: 126.0 max: 200.0
```

1.3模型修正

首先打印Q表，发现已经存在的状态中，velocity, angular velocity值均为0，检查后发现他们的最大值为3.4028235*10^38（为什么会这样并不清楚），因此需要对transform_state函数进行修正。

```
def transform_state(state):
    """将 position, velocity, angle, angular velovity 通过线性转换映射到 [0, 40] 范围内"""
    pos, v, angle, ang_v= state
    pos_low, v_low, angle_low, ang_v_low= env.observation_space.low
    pos_high, v_high,angle_high, ang_v_high = env.observation_space.high

    ad_pos= 50 * (pos-pos_low)/(pos_high-pos_low)
    ad_v =  50* (v-3.4)/6.8
    ad_angle = 50 * (angle-angle_low)/(angle_high-angle_low)
    ad_ang_v = 50*(ang_v-3.4)/6.8

    return int(ad_pos), int(ad_v),int(ad_angle),int(ad_ang_v)
```

利用修正后的函数再次进行训练，效果仍然不好，分析发现原因为当前状态下共有四个参数，每个参数有50个可能的取值，因此全部的参数状态为50^4=6250000个，仅进行一万次训练远远不够。

查看此时Q表发现，当前四个参数的分布较为集中，并不需要对每个参数给予50值空间，再次修改transform_state函数如下：

```
def transform_state(state):
    """将 position, velocity, angle, angular velovity 通过线性转换映射到 [0, 40]
范围内"""
    pos, v, angle, ang_v= state
    pos_low, v_low, angle_low, ang_v_low= env.observation_space.low
    pos_high, v_high,angle_high, ang_v_high = env.observation_space.high

    ad_pos= 20 * (pos-pos_low)/(pos_high)
    ad_v =  20* (v+3.4)/(3.4)
    ad_angle = 20 * (angle-angle_low)/(angle_high)
    ad_ang_v = 20*(ang_v+3.4)/3.4

    return int(ad_pos), int(ad_v),int(ad_angle),int(ad_ang_v)
```

用如上方式将参数状态调整为共20^4=160000个，并将训练次数调整为1000000次。

由下图可见，在运行到30万次时该模型已可以较好地完成学习任务，仅在个别情况下出现score小于200的情况。在运行到32万次时（即参数状态个数的两倍时），几乎可以完美的完成训练任务。

```
episode: 299982 score: 200.0 max: 200.0
episode: 299983 score: 200.0 max: 200.0
episode: 299984 score: 200.0 max: 200.0
episode: 299985 score: 200.0 max: 200.0
episode: 299986 score: 200.0 max: 200.0
episode: 299987 score: 200.0 max: 200.0
episode: 299988 score: 200.0 max: 200.0
episode: 299989 score: 200.0 max: 200.0
episode: 299990 score: 200.0 max: 200.0
episode: 299991 score: 200.0 max: 200.0
episode: 299992 score: 200.0 max: 200.0
episode: 299993 score: 200.0 max: 200.0
episode: 299994 score: 200.0 max: 200.0
episode: 299995 score: 200.0 max: 200.0
episode: 299996 score: 200.0 max: 200.0
episode: 299997 score: 200.0 max: 200.0
episode: 299998 score: 200.0 max: 200.0
episode: 299999 score: 200.0 max: 200.0
episode: 300000 score: 200.0 max: 200.0
```

计时发现，运行完100万次大概花费3小时，尽管最终结果可以较为成功地完成游戏，但耗时过大，主要由于state有4个参数且每个参数的状态较多导致。

综上，可以认为Q-learing模型不是CartPole游戏的最佳方法

二、DQN（Deep-Q-learning）

2.1 模型介绍

DQN是对Q-learning的改进，核心思想为利用深度神经网络对Q值函数化，以解决state为连续值时的情况。由此猜测，DQN模型应当对当前游戏较为适用。

由第一部分可知，Q-table如下：

```
Q = defaultdict(lambda: [0, 0])
```

输入一个一维向量（state），输出一个一维向量（action0_value, action1_value）

按照这个思路，可以搭建一个简单的神经网络,其中最关键的部分为构造DQN类，其需要的函数功能如下：初始化，实现神经网络的构造，预测动作，记录状态，训练模型，保存模型

初始化

```
class DQN(object):
    def __init__(self):
        self.step = 0
        self.update_freq = 100  # 模型更新频率
        self.replay_size = 10000  # 训练集大小
        self.replay_queue = deque(maxlen=self.replay_size)
        self.model = self.create_model()
        self.target_model = self.create_model()
```

构造神经网络

```
 def create_model(self):
        """创建一个隐藏层为100的神经网络"""
        STATE_DIM, ACTION_DIM = 4, 2
        model = models.Sequential([
            layers.Dense(100, input_dim=STATE_DIM, activation='relu'),
            layers.Dense(ACTION_DIM, activation="linear")
        ])
        model.compile(loss='mean_squared_error',
                      optimizer=optimizers.Adam(0.001))
        return model
```

使用ε-greedy policy进行动作预测

```
def act(self, s, epsilon=0.1):
        """预测动作"""
        # ε-greedy policy
        if np.random.uniform() < epsilon - self.step * 0.0002:
            return np.random.choice([0, 1])
        return np.argmax(self.model.predict(np.array([s]))[0])
```

保存历史记录

```
def remember(self, s, a, next_s, reward):
        self.replay_queue.append((s, a, next_s, reward))
```

训练模型

```
def train(self, batch_size=64, lr=1, factor=0.95):
        if len(self.replay_queue) < self.replay_size:
            return
        self.step += 1
        # 每 update_freq 步，将 model 的权重赋值给 target_model
        if self.step % self.update_freq == 0:
                    self.target_model.set_weights(self.model.get_weights())

        replay_batch = random.sample(self.replay_queue, batch_size)
        s_batch = np.array([replay[0] for replay in replay_batch])
        next_s_batch = np.array([replay[2] for replay in replay_batch])

        Q = self.model.predict(s_batch)
        Q_next = self.target_model.predict(next_s_batch)

        # 使用公式更新训练集中的Q值
        for i, replay in enumerate(replay_batch):
            _, a, _, reward = replay
            Q[i][a] = (1 - lr) * Q[i][a] + lr * (reward + factor *
np.amax(Q_next[i]))

        # 传入网络进行训练
        self.model.fit(s_batch, Q, verbose=0)
```

保存模型

```
def save_model(self, file_path='CartPole.h5'):
        print('model saved')
        self.model.save(file_path)
```

训练模型

以上函数使得神经网络准备就绪，接下来可以开始训练

```
env = gym.make('MountainCar-v0')
episodes = 1000  # 训练1000次
score_list = []  # 记录所有分数
agent = DQN()
for i in range(episodes):
    s = env.reset()
    score = 0
    while True:
        a = agent.act(s)
```

```
        next_s, reward, done, info = env.step(a)
        agent.remember(s, a, next_s, reward)
        agent.train()
        score += reward
        s = next_s
        if done:
            score_list.append(score)
             print('episode:', i, 'score:', score, 'max:', max(score_list))
            break
    # 最后20次的平均分等于200时，停止并保存模型
    if np.mean(score_list[-20:])==200:
        agent.save_model()
        break
env.close()
```

使用如上代码运行测试，训练用时较长，可以看到模型在训练了15个周期后逐渐达到稳定，可以成功完成游戏