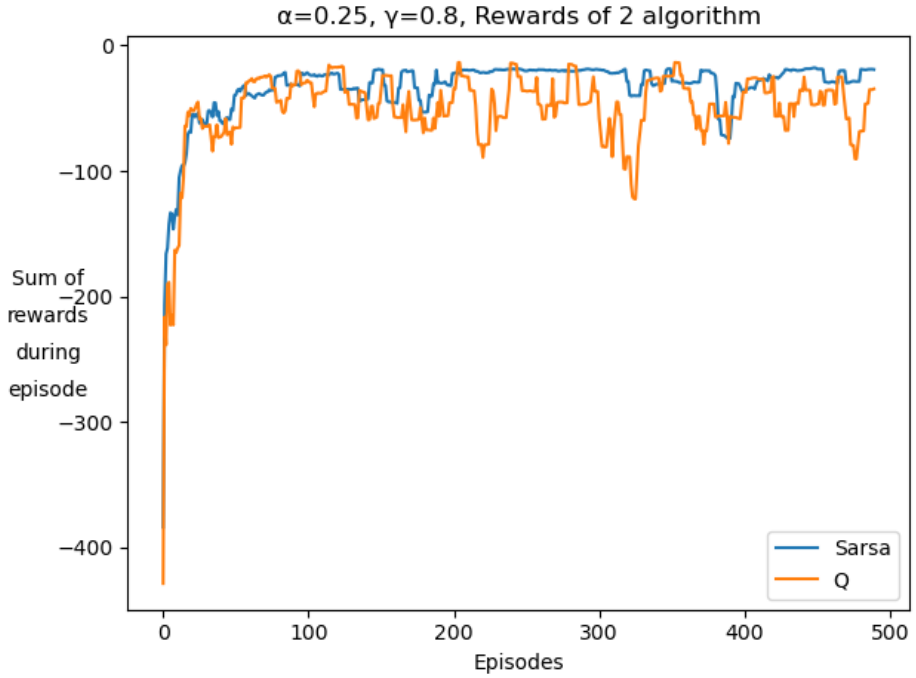
编程作业 3 实验报告

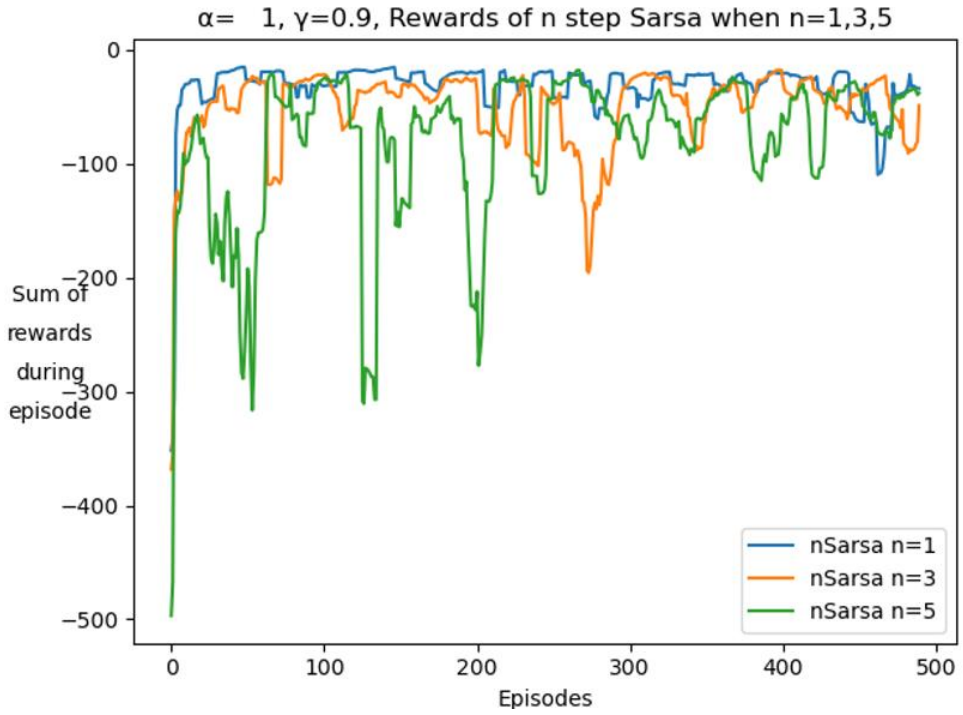
191300073 杨斯凡 191300073@smail.nju.edu.cn

一、实验结果

因为除了ε之外，超参数γ，α也可能对算法的结果产生影响，我将 Sarsa 和 Q-Learning 算法的α设为 0.25，γ设为 0.8，ε设为 0.1，得到的图像如下:

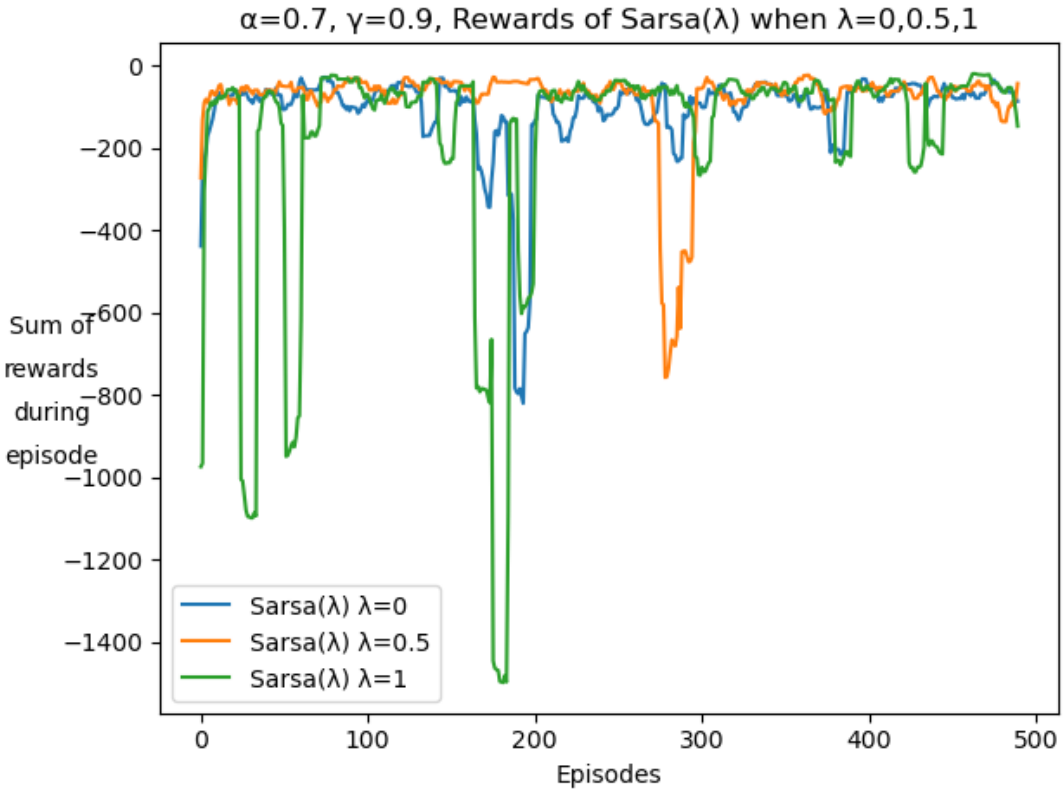


对于 n 步 Sarsa，超参数同样也会造成影响，在这里我统一将α设为 0.1，γ设为 0.9，ε设为了 0.1，实验结果如下:



从图中可以看出，步数较大的时候，一开始的回报较小，因为随机性过强，而到后面的时候，步数较大的时候回报会变高。

对于 Sarsa(λ)，超参数同样也会造成影响，在这里我统一将α设为 0.7，γ设为 0.9，ε设为了 0.1，实验结果如下所示



由图可知，λ越大的时候，学习效果反而不好，这是因为累积迹衰减的过少，导致之前的决策影响较大。

二、代码框架

这次作业的游戏环境我经过查阅资料后，使用了 gym 库中的 CliffWalking-v0 悬崖行走环境。其中接口 API 有：gym.make(env)：生成游戏环境；env.action_space：可以做出的所有行动；env.reset()：重新初始化游戏；env.step(action)：做出 action 的行动，并返回状态，奖赏，是否完成和信息；.env.close()：关闭游戏

我定义了四个类：Sarsa（Sarsa 算法），Q_Learing（Q 学习），nSarsa（n 步 Sarsa），Sarsalamda（Sarsa(λ)算法），其中 Play 是用来进行训练的函数。

下面分别对四个算法进行描述：

Sarsa:

```python
class Sarsa():
    def __init__(self, alpha, epsilon, gamma, env='CliffWalking-v0'): # 0 up, 1 right, 2 down, 3 left
        self.alpha = alpha
        self.epsilon = epsilon
        self.gamma = gamma
        self.env = gym.make(env)
        self.action_num = 4
        self.state_num = 48
        self.Q = np.zeros((48,4))
    def choose(self, state):
        if np.random.uniform(0, 1) <= self.epsilon:
            return self.env.action_space.sample()
        else:
            return np.argmax(self.Q[state])
```

首先进行初始化，Sarsa 接受 3 个参数，分别为α，ε，γ，然后对类进行初始化，action_num 为可以做出的行动总数，因为在 gym 的环境中，0-3 分别代表上右下左，故可以使用数字进行表示，state_num 为总的状态数，Q 为 48*4 的矩阵。

Choose 为ε-greedy 策略，有 0.1 的几率随机选择，0.9 的几率选使得当前 Q 值最大的行动

```python
    def Play(self):
        lst = []
        for i in range(47):
            for j in range(4):
                self.Q[i,j]=random.random()
        for episode in range(MAX):
            state = self.env.reset()
            action = self.choose(state)
            done = False
            sreward = 0
            while not done:
                obs, reward, done, info = self.env.step(action)
                sreward += reward
                next_action = self.choose(obs)
                self.Q[state,action]+=self.alpha * (reward + self.gamma * self.Q[obs, next_action] - self.Q[state, action])
                state, action = obs, next_action
            lst.append(sreward)
        self.env.close()
        return lst
```

然后是训练过程，首先对非终止态的 Q 进行随机初始化，然后进行迭代，迭代次数 MAX 为 500，首先重置游戏，然后随机选择行动，然后进行行动，计入奖赏，然后根据下面的公式更新 Q:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\Big[R_t + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)\Big]$$

然后不断迭代即可。

Q-learning:

```
def Play(self):
    for i in range(47):
        for j in range(4):
            self.Q[i,j]=random.random()
    lst = []
    for episode in range(MAX):
        state = self.env.reset()
        action = self.choose(state)
        done = False
        sreward = 0
        while not done:
            obs, reward, done, info = self.env.step(action)
            sreward += reward
            self.Q[state, action] += self.alpha * (reward + self.gamma * np.max(self.Q[obs]) - self.Q[state, action])
            state = obs
            action = self.choose(state)
        lst.append(sreward)
    self.env.close()
    return lst
```

初始化过程与 Sarsa 相同，这里不多做赘述；Q 学习和 Sarsa 唯一不同的地方是更新 Q 的的时候使用的公式不同，Q 学习使用的是使下一个状态最大的行动的 Q 值进行更新

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_t + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t) \right]$$

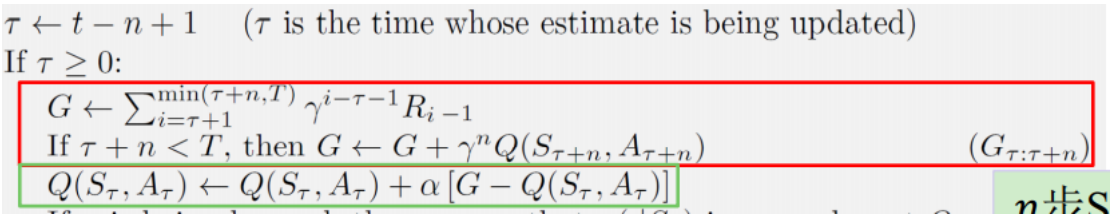
n 步 Sarsa：

n 步 Sarsa 相比于上面两个算法会多一个 n 的参数，初始化的时候需要进行接收

```
def Play(self):
    lst=[]
    for i in range(MAX):
        state = self.env.reset()
        action = self.choose(state)
        done = False
        state_list, action_list, reward_list = [state], [action], [0]
        T=infinity
        t=0
        while True:
            if t<T:
                obs, reward, done, info = self.env.step(action_list[-1])
                state_list.append(obs)
                reward_list.append(reward)
                if done:
                    T=t+1
                else:
                    action_list.append(self.choose(state_list[-1]))
            temp=t-self.n+1
            if temp>0:
                G = 0
                for i in range(temp + 1, min(temp + self.n, T) + 1):
                    G += self.gamma ** (i - temp - 1) * reward_list[i-1]
                if temp + self.n < T:
                    G += self.gamma ** self.n * self.Q[state_list[temp + self.n], action_list[temp + self.n]]
                    s, a = state_list[temp], action_list[temp]
                    self.Q[s, a] += self.alpha * (G - self.Q[s, a])
            if temp == T - 1:
                break
            t+=1
        lst.append(sum(reward_list))
    self.env.close()
    return lst
```

在 n 步 Sarsa 更新的过程中，会使用到之后的回报，因此需要使用一个列表进行存储，为了一一对应，我将选择的前后的状态和行动也使用了列表进行存储。

然后就是复现伪代码，当 t<T 的时候，做出行动，然后将行动之后的状态和奖赏加入列表中，然后进行 G 的计算，模拟了下面的公式：

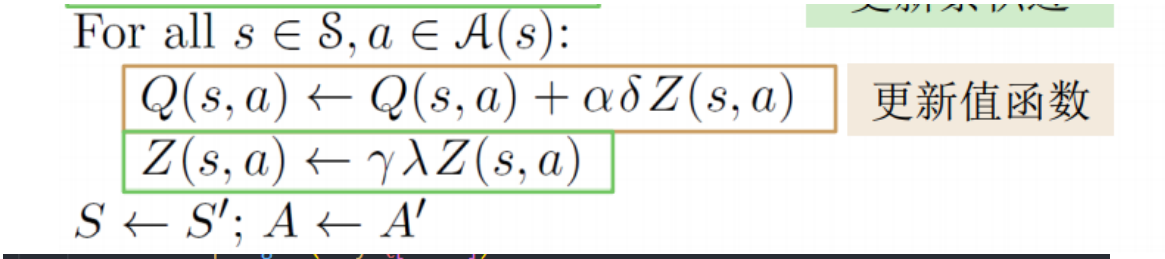


最后将奖赏列表之和加入总奖赏列表中即可。

Sarsa(λ):

Sarsa(λ)算法有一个λ的参数，初始化的时候需要进行接收。

我实现的 Sarsa(λ)使用了累积迹，在每次迭代的过程中，需要对迹进行初始化，然后随机采取一个行动，当每做出一个行动之后，需要计算 TD 误差，然后将累积迹加一，然后通过下面的公式进行值函数的更新:




```python
def Play(self):
    lst=[]
    for i in range(MAX):
        Z=np.zeros((48, 4))
        state = self.env.reset()
        action = self.env.action_space.sample()
        done = False
        sreward=0
        while not done:
            obs, reward, done, info = self.env.step(action)
            next_action=self.choose(obs)
            delta=reward+self.gamma*self.Q[obs,next_action] - self.Q[state,action]
            Z[state, action] += 1
            sreward+=reward
            for i in range(48):
                for j in range(4):
                    self.Q[i,j]+=self.alpha*delta*Z[i,j]
                    Z[i,j]*=(self.gamma*self.lamda)
            state, action = obs, next_action
        lst.append(sreward)
    self.env.close()
    return lst
```