Department of Electrical & Computer Engineering
2022SPRING - Mathematical and Biological Foundation of RL

Project: find optimal policies of in a grid world

Date: May 26, 2022

Contents

1 Problem description

In a 4x4 grid world as illustrated in Fig 1.1. The red, blue, and green represent forbidden areas, the target, and the agent respectively. The agent's goal is to reach the target with the least steps within the constraints of forbidden areas.

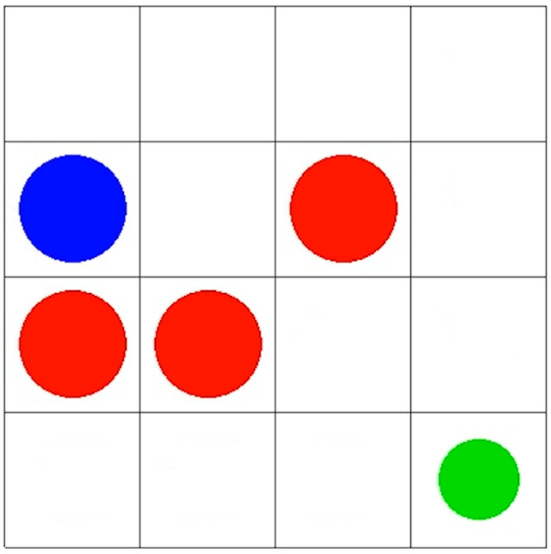


Figure 1.1: Grid world

For the agent, actions that lead to forbidden areas or boundaries are excluded. Therefore the forbidden areas do not correspond to the state.

2 Assignment 1

I build two files corresponding to the environment and agent, which will be introduced in the following subsections.

2.1 Environment setup

The environment can interact with the agent, and also visualize the grid world with the inheriting of the class gym.Env.

The states are set to be a list of 0-15 and remove 6, 8, 9 three states. The actions are set to be an array of [-4, 4, -1, 1], which represents the agent going up, down, left, and right. The reward is set to be -1 corresponding to all pairs of actions and states, excluded 100 for the pair leading to the target.

2.2 Agent

The agent's starting state is 15. In every step, the agent observes the state of itself and the environment and takes action according to a different policy. In assignment 1, I define additional two deterministic policies as in Fig 2.1. The agent will choose two policies in turn.

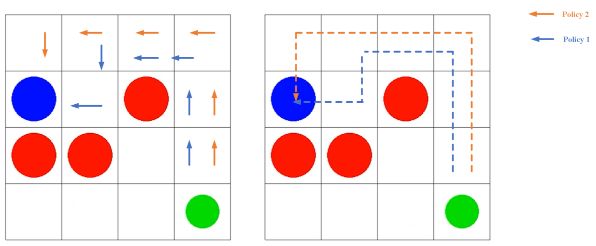
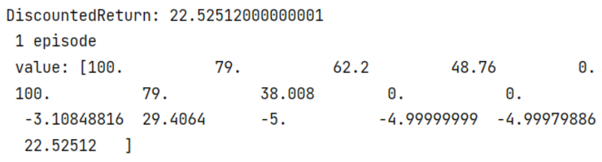


Figure 2.1: Trajectories of the two policies

2.3 Results

The discounted return of the two deterministic policies is illustrated in Fig. 2.2.



```
DiscountedReturn: 22.52512000000001
 1 episode
 value: [100.         79.         62.2        48.76        0.
 100.         79.         38.008       0.          0.
  -3.10848816  29.4064      -5.          -4.99999999  -4.99979886
  22.52512   ]
```

Figure 2.2: The discounted return

3 Assignment 2

In this assignment, I append the action ”Don’t move” and choose the two policies as in Fig. 3.1, one of which is not the optimal policy.

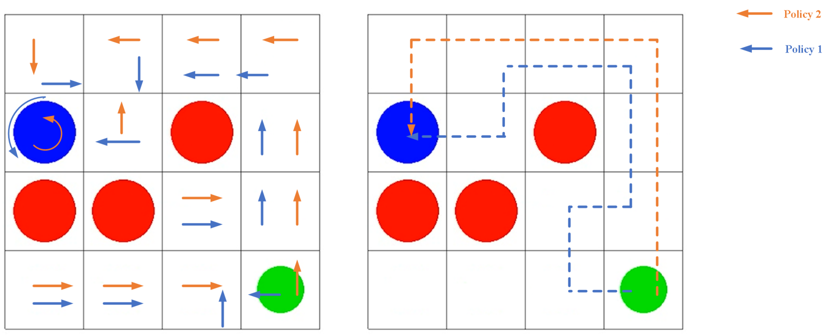


Figure 3.1: The new two policies

3.1 The parameters of the Bellman equation in matrix-vector form

The matrix-vector form of the Bellman equation can be written as :

$$v_\pi = r_\pi + \gamma P_\pi v_\pi \tag{1}$$

Where $v_\pi = [v_\pi(s_1), \ldots, v_\pi(s_n)]^T \in R^n$ denotes a vector contain all the state values.$r_\pi = [r_\pi(s_1), \ldots, r_\pi(s_n)]^T \in R^n$, is the mean reward can be obtain that starting from each state s corresponding to each element in v_π under policy π. $[P_\pi]_{ij} = p_\pi(s_j \mid s_i), P_\pi \in R^{n\times n}$ is the state-transform matrix.

In class agent, method deterministicPolicy() will return action based on the current state according to the two policies.

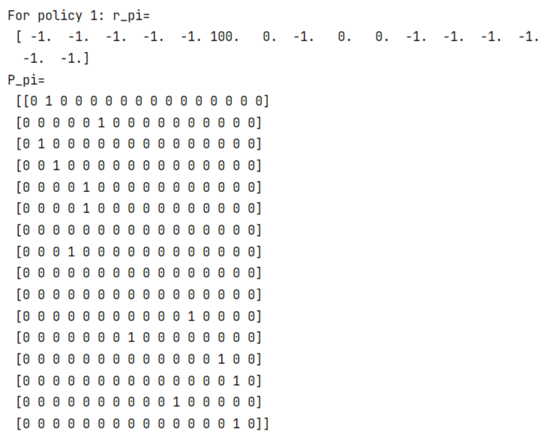
```
for i in agent.states:

    chosenAction=agent.deterministicPolicy(i,1)
    env.state=i
    nextstate, reward, Terminal = env.step(chosenAction)
    P_pi_1[i,nextstate]=1
    r_pi_1[i]=agent.rewards[chosenAction,i]
  #Value=0 means the forbiden state, Value=100 means the target
  ↪   state. others are -1

    chosenAction = agent.deterministicPolicy(i,2)
    nextstate, reward, Terminal = env.step(chosenAction)
    P_pi_2[i,nextstate]=1
    r_pi_2[i] =agent.rewards[chosenAction,i]
    print("For policy 1: r_pi=\n",r_pi_1,"\nP_pi=\n",P_pi_1,"\n For
     ↪  policy 2: r_pi=\n",r_pi_2,"\nP_pi:\n",P_pi_2)
```

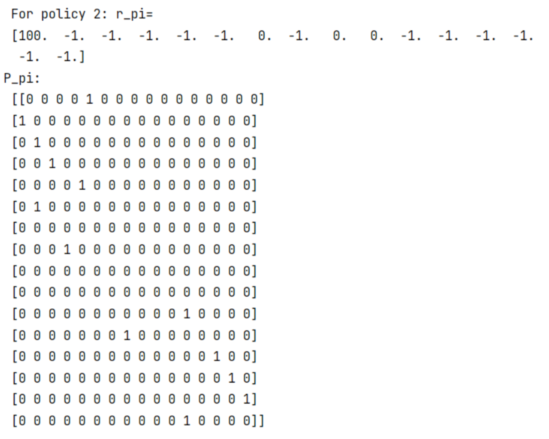
Listing 3.1: Code that calculates the parameters in the Bellman equation

Then I use a loop to walk through all the states to get r_π and P_π as shown in Fig 3.2.



```
For policy 1: r_pi=
 [ -1.  -1.  -1.  -1.  -1. 100.   0.  -1.   0.   0.  -1.  -1.  -1.  -1.
  -1.  -1.]
P_pi=
 [[0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0]
 [0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0]
 [0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0]
 [0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0]]
```

(a) r_π and P_π for the policy 1



```
For policy 2: r_pi=
[100.  -1.  -1.  -1.  -1.  -1.   0.  -1.   0.   0.  -1.  -1.  -1.  -1.
  -1.  -1.]
P_pi:
 [[0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0]
 [1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0]
 [0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0]
 [0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1]
 [0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0]]
```

(b) r_π and P_π for the policy 2

Figure 3.2: Matrix parameters in Bellman equation

3.2 Solve the state values by using the closed-form solution

The closed-form solution can be written as

$$v_\pi = (I - \gamma P_\pi)^{-1} r_\pi \tag{2}$$

For the two policies, we have:

$$v_{\pi 1} = \begin{bmatrix} 59.64 \\ 75.8 \\ 59.64 \\ 46.71 \\ -5. \\ 96. \\ 0. \\ 36.37 \\ 0. \\ 0. \\ 21.48 \\ 28.1 \\ 8.56 \\ 11.94 \\ 16.18 \\ 11.94 \end{bmatrix}, v_{\pi 2} = \begin{bmatrix} 96. \\ 75.8 \\ 59.64 \\ 46.71 \\ -5. \\ 59.64 \\ 0. \\ 36.37 \\ 0. \\ 0. \\ 21.48 \\ 28.1 \\ 8.56 \\ 11.94 \\ 16.18 \\ 21.48 \end{bmatrix} \tag{3}$$

3.3 Solve the state values by using the iterative solution.

The Bellman equation also can be written as iteration form:

$$v_s = r_\pi + \gamma P_\pi v_{s+1} \tag{4}$$

```
num_episodes = 100
max_number_of_steps=20

for p in [1, 2]:
    agent.vtab = np.zeros(len(agent.vtab))
    for i in range(num_episodes):
        env.reset()  #random choose the init state , or the agent
        ↪  will not go through all states

        for t in range(max_number_of_steps):
            env.render()

            #chosenAction
            chosenAction=agent.deterministicPolicy(env.state,p)
            nextstate, reward, Terminal=env.step(chosenAction)

            #Value ieration
            agent.vtab[env.state]=reward+agent.gamma*agent.vtab[next⌋
            ↪  state]  #Dynamic Programming -Value
            ↪  iterations
            env.state=nextstate

    print('Iteration_solution_Policy
    ↪  ',p,':\n',agent.vtab.reshape(len(agent.vtab),1))
```

Listing 3.2: Code that calculates the state value in iteration solution

Do the value iteration for every step and randomly choose the initial state to ensure the agent will go through all the states. After calculation, we have:

$$v_{\pi 1} = \begin{bmatrix} 59.64 \\ 75.8 \\ 59.64 \\ 46.71 \\ -5. \\ 96. \\ 0. \\ 36.37 \\ 0. \\ 0. \\ 21.48 \\ 28.1 \\ 8.56 \\ 11.95 \\ 16.18 \\ 11.94. \end{bmatrix} \quad v_{\pi 2} = \begin{bmatrix} 96. \\ 75.8 \\ 59.64 \\ 46.71 \\ -5. \\ 59.64 \\ 0. \\ 36.37 \\ 0. \\ 0. \\ 21.48 \\ 28.1 \\ 8.56 \\ 11.94 \\ 16.18 \\ 21.48 \end{bmatrix} \tag{5}$$

3.4 Discussion

In order to make sure that the iteration solution converges to the closed-form solution, three points need to consider:

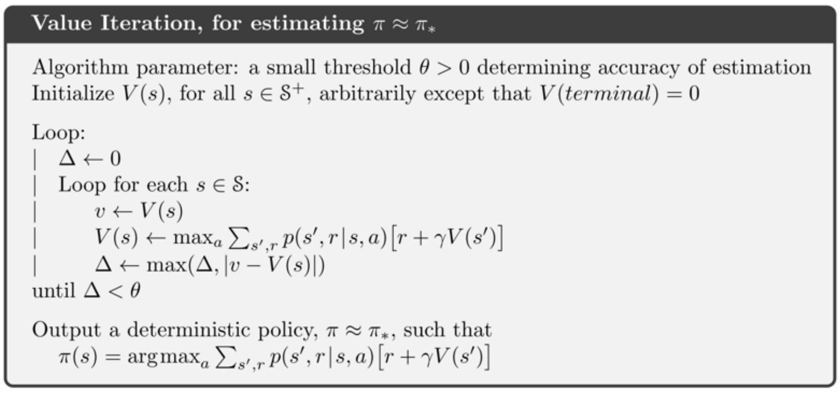
1. The number of steps should be large enough.
2. To ensure that the times of agent passing through each state is quite enough, the initial state of the agent should be random, and episodes should also be large enough.
3. Due to the bootstrapping algorithm, the state value will converge from the terminal state to the initial state, which means the previous state value will converge slower than the later ones

4 Assignment 3: model-based algorithms

First of all, to solve this problem I first consider model-based algorithms, which means the agent knows everything about the environment including calculating the next state based on the current state and action. Dynamic programming is a common class of model-based algorithms including value iteration and policy iteration.

4.1 Value iteration

In order to find the optimal policy, we can simply estimate the state value and define the policy as $\pi(s) = \arg\max_a \sum_{s',r} p(s', r \mid s, a)\left[r + \gamma V(s')\right]$. The off-policy algorithm can separate the value approximation and $\epsilon - greedy$ policy algorithm. Action policy Additionally, value iteration is much easier to execute.



Value Iteration, for estimating $\pi \approx \pi_*$

```
Algorithm parameter: a small threshold $\theta > 0$ determining accuracy of estimation
Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(terminal) = 0$

Loop:
|  $\Delta \leftarrow 0$
|  Loop for each $s \in \mathcal{S}$:
|      $v \leftarrow V(s)$
|      $V(s) \leftarrow \max_a \sum_{s',r} p(s', r|s, a)[r + \gamma V(s')]$
|      $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$

Output a deterministic policy, $\pi \approx \pi_*$, such that
   $\pi(s) = \arg\max_a \sum_{s',r} p(s', r|s, a)[r + \gamma V(s')]$
```

Figure 4.1: The pseudocode of value iteration

The initial state is shown in Fig 1.1. The training episode is set to be 200 times. For each episode, the game will end either the agent reaches the goal or the agent has run 100 steps. Denote the parameter $\gamma = 0.8, \epsilon = 0.8$, and $decrease = 0.01$, The training procedure is recorded in the attachment DyPro-v.mp4 and the return(without discount) is shown in Fig 4.2.

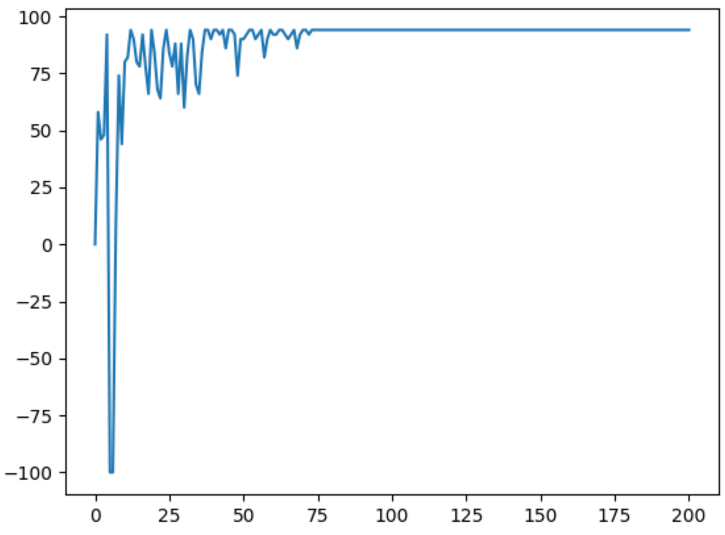
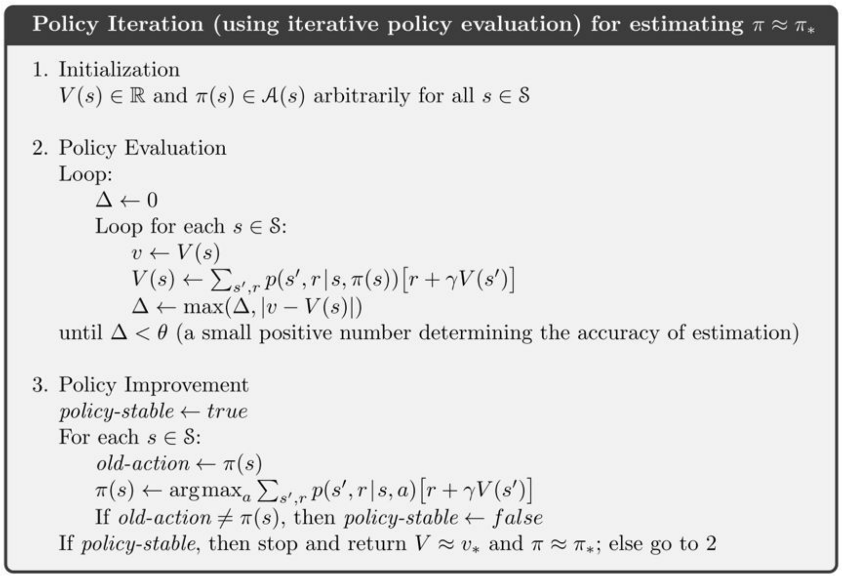


Figure 4.2: The return of each episode

After training, the state value vector is iterated and reshaped to a 4X4 matrix in order to corresponding to Fig 4.2 as follow:

$$v_s = \begin{bmatrix} 100. & 79. & 62.2 & 48.76 \\ 0. & 100. & 79. & 38.01 \\ 0. & 0. & -3.11 & 29.41 \\ -5. & -5.00 & -5.00 & 22.52 \end{bmatrix} \tag{6}$$

4.2 Policy iteration



Policy Iteration (using iterative policy evaluation) for estimating $\pi \approx \pi_*$

```
1. Initialization
   $V(s) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in \mathcal{S}$

2. Policy Evaluation
   Loop:
       $\Delta \leftarrow 0$
       Loop for each $s \in \mathcal{S}$:
           $v \leftarrow V(s)$
           $V(s) \leftarrow \sum_{s',r} p(s', r \mid s, \pi(s))\big[r + \gamma V(s')\big]$
           $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
   until $\Delta < \theta$ (a small positive number determining the accuracy of estimation)

3. Policy Improvement
   *policy-stable* $\leftarrow$ *true*
   For each $s \in \mathcal{S}$:
       *old-action* $\leftarrow \pi(s)$
       $\pi(s) \leftarrow \operatorname{argmax}_a \sum_{s',r} p(s', r \mid s, a)\big[r + \gamma V(s')\big]$
       If *old-action* $\neq \pi(s)$, then *policy-stable* $\leftarrow$ *false*
   If *policy-stable*, then stop and return $V \approx v_*$ and $\pi \approx \pi_*$; else go to 2
```

Figure 4.3: The pseudocode of policy iteration

For the policy evaluation, state value equals all the possible next states times the probability of the corresponding action. Besides, action value equals by the bellman equation as in List 3.2.

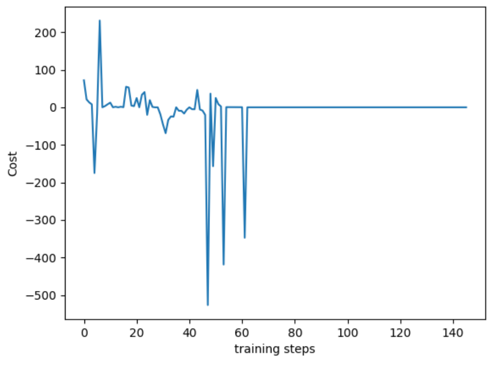
$$V(s) \leftarrow \sum_a \pi(a \mid s) \sum_{s',r} p\left(s', r \mid s, a\right)\left[r + \gamma V\left(s'\right)\right] Q(s,a) \leftarrow r + \gamma V\left(s'\right) \tag{7}$$

$$Q(s,a) \leftarrow r + \gamma V\left(s'\right) \tag{8}$$

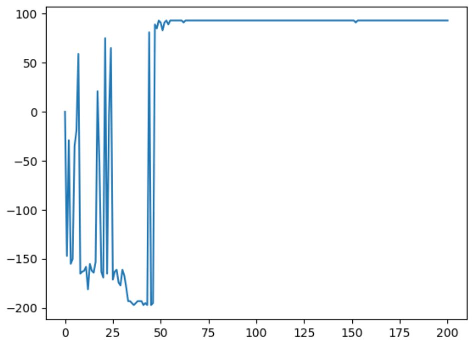
```
        self.vtab[observe_state] =
        ↪  torch.sum(actionpro*(assume_reward + self.gamma *
        ↪  assume_nextv))
        self.qtab[observe_state,int(chosenAction)]=assume_reward[int⌟
        ↪  (chosenAction)]+self.gamma*assume_nextv[int(chosenAction⌟
        ↪  )]
```

Listing 4.1: Code that calculates the state value and action value

For the policy improvement, stochastic policy based on the DNN as in Fig 5.2 is optimized by the advantage function $A^{\pi}(s,a) = Q^{\pi}(s,a) - V^{\pi}(s)$ with the loss $L = -log\pi(a|s)A^{\pi}(s,a)$.



(a) The training loss.



(b) The returns.

Figure 4.4: The training of policy iteration in 200 episodes

5 Assignment 4: MC-based learning

5.1 MC Exploring Starts: Policy gradient

In this section, we consider methods that learn a parameterized policy that can select actions without consulting a value function. A value function may still be used to learn the policy parameter but is not required for action selection. Policy-based method's advantage is that can output a continuous stochastic policy as in Fig 5.1.

```
function REINFORCE
    Initialise $\theta$ arbitrarily
    for each episode $\{s_1, a_1, r_2, ..., s_{T-1}, a_{T-1}, r_T\} \sim \pi_\theta$ do
        for $t = 1$ to $T - 1$ do
            $\theta \leftarrow \theta + \alpha \nabla_\theta \log \pi_\theta(s_t, a_t) v_t$
        end for
    end for
    return $\theta$
end function
```

Figure 5.1: The pseudocode of policy gradient

In this case, we use a DNN to output the action as in Fig 5.2. Two linear layers are connected in the middle using the Relu activation function and finally connected to Softmax to output the probability of each action. The policy will update at the end of each episode based on MC method.

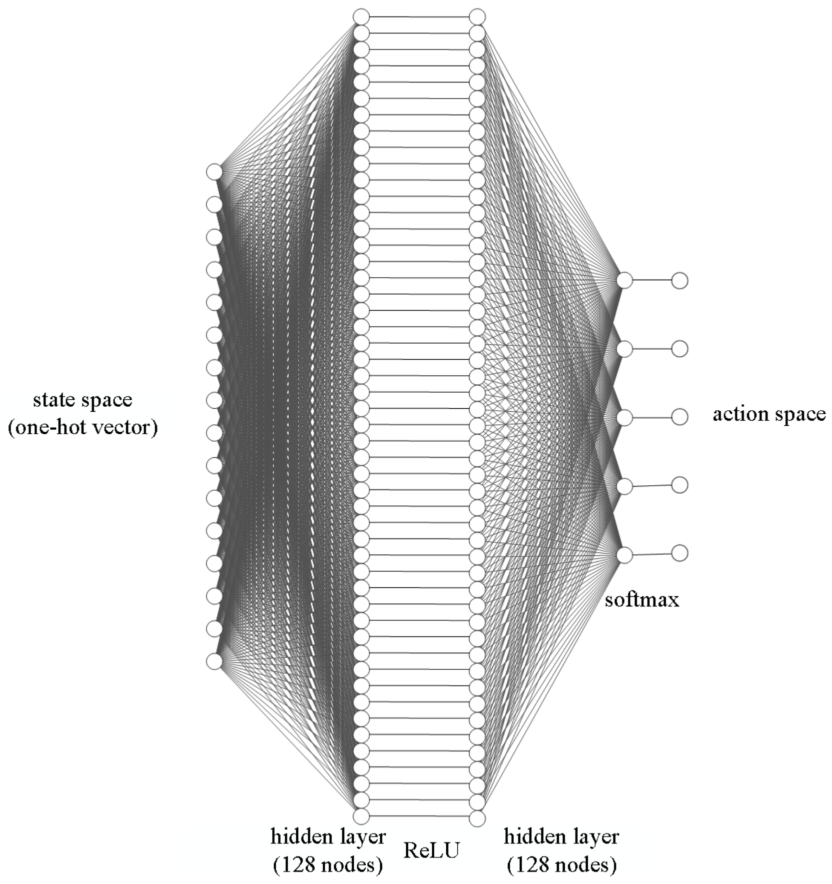
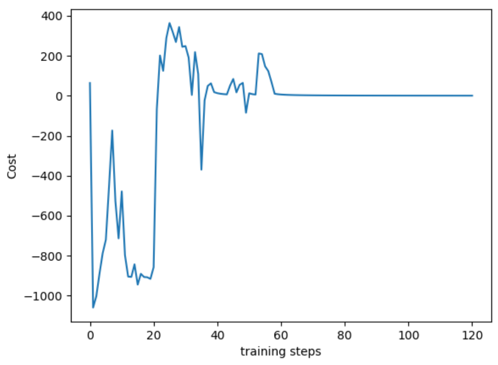


Figure 5.2: The structure of policy neural network

The training loss is $L = qlogP(a)$, which is similar to cross-entropy loss. It makes the distribution of action approximate the distribution of the action value under the current policy. The training process is shown in Fig 5.3. The PG algorithm converges in 50 episodes and behaves better than all the algorithms in the previous section.



(a) The training loss of policy gradient.

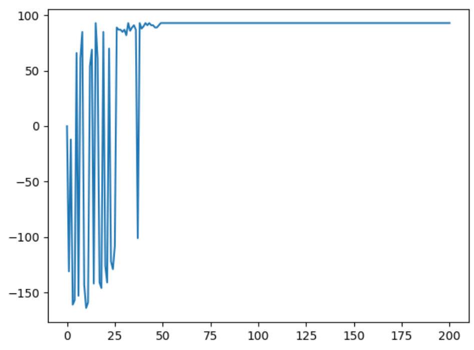

(b) The return of policy gradient

Figure 5.3: The training process of policy gradient in 200 episodes

5.2 MC ϵ-greedy: based on policy gradient

In this section, I rewrite the code with ϵ-greedy trick, which means the agent will choose random action with a decreasing hyperparameter possibility ϵ to ensure exploration. As the random action has no gradient I alternatively use gaussian noise to mask the probability of action with possibility ϵ as in list 5.1, the result is shown in Fig 5.4.

```
    def choose_action_e_greedy(self, state):
        # statetensor [16]->[1,16]  unsqueeze(0)
        s = torch.Tensor(state).unsqueeze(0)
        prob = self.pi(s)   # :

        global EPSILON

        if random.random() < EPSILON:
            sigma = Sigma
            prob1=torch.tensor(prob)
            for i in range(prob.shape[1]):
                prob1[0,i]=prob[0,i]+ random.gauss(0, sigma)
            prob2 = F.softmax(prob1,dim=1)
            m = torch.distributions.Categorical(prob2)
        # 1
        else:
            m = torch.distributions.Categorical(prob)   #
        action = m.sample()

        EPSILON -= ep_d

        return action.item(), m.log_prob(action)
```

Listing 5.1: ϵ-greedy with gaussian noise

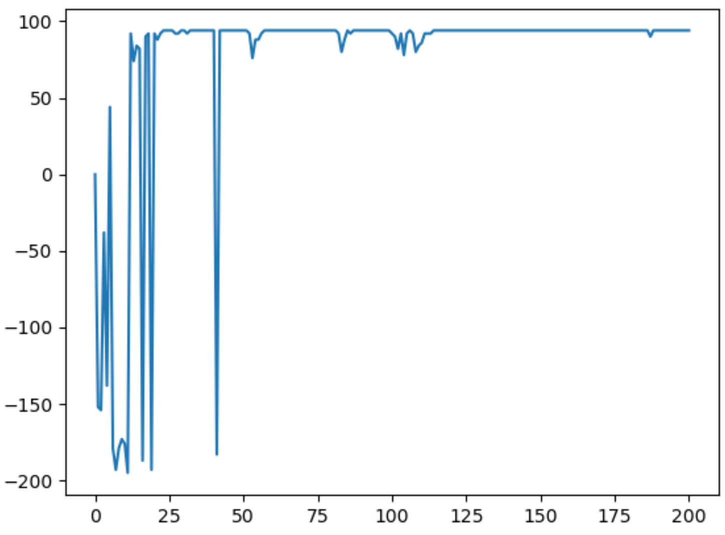


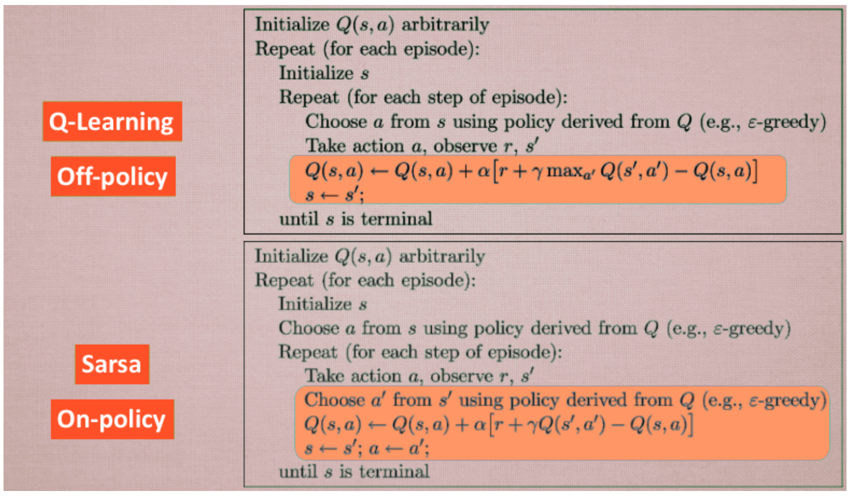
Figure 5.4: The return of each episode

6 model-free algorithms

Unlike the model-based algorithms, the file agent does not include any issues with the state transform matrix. To compare with the mode-based algorithm, the parameters are the same as in the previous section.

6.1 Value-based methods

Q-learning and SARSA are the most famous classical value-based methods. The difference between them is that Q-learning is the off-policy while SARSA is the on-policy as shown in Fig 6.1.

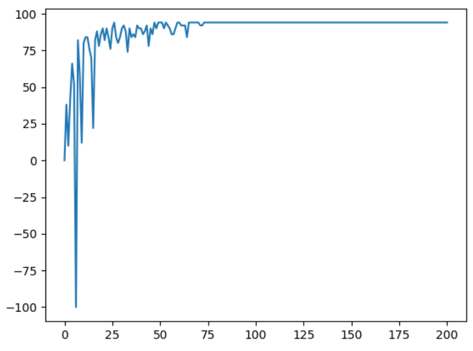

Q-Learning
Off-policy
Sarsa
On-policy

```
Initialize $Q(s,a)$ arbitrarily
Repeat (for each episode):
    Initialize $s$
    Repeat (for each step of episode):
        Choose $a$ from $s$ using policy derived from $Q$ (e.g., $\varepsilon$-greedy)
        Take action $a$, observe $r$, $s'$
        $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a') - Q(s,a)]$
        $s \leftarrow s'$;
    until $s$ is terminal
```

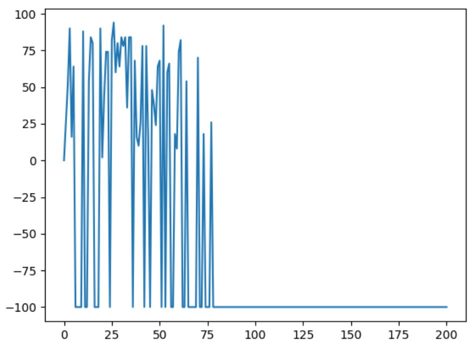
```
Initialize $Q(s,a)$ arbitrarily
Repeat (for each episode):
    Initialize $s$
    Choose $a$ from $s$ using policy derived from $Q$ (e.g., $\varepsilon$-greedy)
    Repeat (for each step of episode):
        Take action $a$, observe $r$, $s'$
        Choose $a'$ from $s'$ using policy derived from $Q$ (e.g., $\varepsilon$-greedy)
        $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma Q(s',a') - Q(s,a)]$
        $s \leftarrow s'$; $a \leftarrow a'$;
    until $s$ is terminal
```

Figure 6.1: The pseudocode of two policies

The return of each episode with respect to two policies is shown in Fig. 6.2. The SARSA algorithm hasn't converged in this case. On the other hand, the Q-learning algorithm behaves as good as well as the model-free method in the previous section. than the SARSA algorithm.



(a) The return of Q-learning



(b) The return of SARSA

Figure 6.2: Returns of two policies in 200 episodes

The action value of Q-learning is:

$$Q_{s,A} = \begin{bmatrix} 0. & 91.41006541 & 0 & 4.75603507 \\ 0. & 79. & 53.99 & 47.86 \\ 0. & 0. & 62.2 & 34.2 \\ 0. & 27.61 & 48.76 & 0. \\ 0. & 0. & 0. & 0. \\ 56.39 & 0. & 100. & 0. \\ 0. & 0. & 0. & 0. \\ 38.01 & 18.89 & 0. & 0. \\ 0. & 0. & 0. & 0. \\ 0. & 0. & 0. & 0. \\ 0. & 2.68 & 0. & 21.056 \\ 29.41 & 11.8 & 13.77 & 0. \\ 0. & 0. & 0. & -0.99993 \\ 0. & 0. & -0.99993 & 1.01 \\ 6.58 & 0. & -0.99997 & 14.05 \\ 22.53 & 0. & 6.358 & 0. \end{bmatrix} \tag{9}$$

6.2 Off policy for DRL

Off-policy algorithms such as DDPG don't work well. DDPG is suitable for continuous action. DDQN can converge if I use another state in the middle of the optimal path as target before training the agent to the real target state. I admit that this sparse reward setting is not so friendly

PPO is the most efficient algorithms among all the algorithms before, as it stably converges in 30 episodes.

7 Appendix

The Dynamic programming, Sarsa, Q-learning, and Policy gradient algorithms can also be found in the attachment, which will be introduced in the following assignments. The sourse code will keep up to date on my Github:
`https://github.com/GarrentDSTRC/RL_Project1_GridWorld`

List of Figures