

# Assignment 1

---

余北辰 519030910245

## 1 Experimental Purpose

---

- Build the Gridworld environment
- According to the idea of dynamic programming, respectively use policy iteration and value iteration to solve the shortest path problem in Gridworld
- Compare the performance of each of the two methods

## 2 Experimental Background

---

In this section, we present the methods for policy iteration and value iteration separately.

### 2.1 Policy Iteration

According to the Bellman Equation, policy iteration uses alternating steps of "policy evaluation" and "policy improvement" to find a sequence of policies that improve once at a time and eventually converge to the optimal policy.

#### Policy Evaluation

Based on the current policy, traverse the entire MDP, and at each state iteratively update the value function using Bellman Expectation Equation, until the judgment condition of convergence is satisfied.

#### Policy Improvement

Based on the current value function, traverse the entire MDP, and at each state iteratively update the policy greedily. At each iteration, determine whether the policy has been stabilized. If the policy is already stable, return the existing policy and value function, otherwise repeat the policy evaluation.

The pseudo-code for the policy iteration procedure is shown below:

### 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  $s \in \mathcal{S}$ :  
         $v \leftarrow V(s)$   
         $V(s) \leftarrow \sum_{s',r} p(s',r|s,\pi(s)) [r + \gamma V(s')]$   
         $\Delta \leftarrow \max(\Delta, |v - V(s)|)$   
    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*  $\leftarrow \pi(s)$   
         $\pi(s) \leftarrow \arg \max_a \sum_{s',r} p(s',r|s,a) [r + \gamma V(s')]$   
        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

## 2.2 Value Iteration

Comparing with policy iteration, the process of value iteration is relatively simpler. In value iteration, the value function at each state is updated towards the optimal direction iteratively until the judgment condition of convergence is satisfied. After that, according to the obtained value function, the policy is obtained using Bellman Optimality Equation .

The pseudo-code for the value iteration procedure is shown below:

### 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  $V(\text{terminal}) = 0$

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  
 $\pi(s) = \arg \max_a \sum_{s',r} p(s',r|s,a) [r + \gamma V(s')]$

## 3 Experimental Procedure

The source code for this experiment includes `gridworld.py`, `PolicyIteration.py`, `valueIteration.py` and `main.py`.

## 3.1 gridworld.py

`gridworld.py` builds the Gridworld environment. This module references [reinforcement-learning/gridworld.py at master · dennybritz/reinforcement-learning \(github.com\)](https://github.com/dennybritz/reinforcement-learning/blob/master/gridworld.py), and is modified according to the specific scenario of this course experiment.

The gridworld should be instantiated using the instruction as below:

```
1 | gridworld = GridworldEnv([6, 6])
```

The gridworld will be instantiated as below, in which state 1 and 35 are set as the terminal state.

0	1	2	3	4	5
6	7	8	9	10	11
12	13	14	15	16	17
18	19	20	21	22	23
24	25	26	27	28	29
30	31	32	33	34	35

The number of the states, the number of the actions and the state transfer matrix in gridworld can be obtained using the instruction as below:

```
1 | state_number = gridworld.ns # 36
2 | action_number = gridworld.na # 4
3 | state_transfer_matrix = gridworld.P # P[state][action] = (prob, next_state,
   |   reward, is_done)
```

## 3.2 PolicyIteration.py

`PolicyIteration.py` is implemented based on the above pseudo-code.

```
1 | def policy_evaluation(gridworld, policy, theta=0.001, gamma=0.99):
2 |     v = np.zeros(gridworld.ns)
3 |     iteration = 0
4 |     while True:
5 |         delta = 0.0
6 |         iteration += 1
7 |         for state in range(gridworld.ns):
8 |             v = 0.0
9 |             for action, pi in enumerate(policy[state]):
10 |                 for prob, next_state, reward, is_done in gridworld.P[state]
   | [action]:
11 |                     v += pi*prob*(reward+gamma*v[next_state])
12 |                 delta = max(delta, np.abs(v[state]-v))
13 |             v[state] = v
14 |             if delta < theta:
15 |                 break
```

```
16 |         return v, iteration
```

In the policy evaluation part, first initialize the value function  $V$  in each state to 0. Then using the Bellman Expectation Equation to update  $V$  once at one time, until the change of  $V$  at a time is less than the predetermined threshold  $\theta$ . Record the time of iterations that doing policy evaluation.

```
1  def policy_iteration(gridworld, theta=0.001, gamma=0.99):
2      policy = np.ones([gridworld.ns, gridworld.nA]) / gridworld.nA # Init
   the policy randomly
3      policy_stable = False
4      while not policy_stable:
5          policy_stable = True
6          v, iteration = policy_evaluation(gridworld, policy, theta, gamma)
7          for state in range(gridworld.ns):
8              old_action = np.argmax(policy[state])
9              action_values = np.zeros(gridworld.nA)
10             for action in range(gridworld.nA):
11                 for prob, next_state, reward, is_done in gridworld.P[state]
   [action]:
12                     action_values[action] += prob * \
13                         (reward+gamma*v[next_state])
14             best_action = np.argmax(action_values)
15             policy[state] = np.eye(gridworld.nA)[best_action]
16             if old_action != best_action:
17                 policy_stable = False
18     return v, policy, iteration
```

When doing policy iteration, first of all the policy is initialized as random. Then using a flag `policy_stable` to record the change of policy in an iteration. Update the policy greedily, and stop when the policy does not change.

### 3.3 ValueIteration.py

`valueIteration.py` is implemented based on the above pseudo-code.

```
1  def value_iteration(gridworld, theta=0.001, gamma=0.99):
2      v = np.zeros(gridworld.ns)
3      iteration = 0
4      while True:
5          delta = 0.0
6          iteration += 1
7          for state in range(gridworld.ns):
8              v = v[state]
9              action_values = np.zeros(gridworld.nA)
10             for action in range(gridworld.nA):
11                 for prob, next_state, reward, is_done in gridworld.P[state]
   [action]:
12                     action_values[action] += prob * \
13                         (reward+gamma*v[next_state])
14             v[state] = np.max(action_values)
15             delta = max(delta, np.abs(v[state]-v))
16             if delta < theta:
17                 break
18
19     policy = np.zeros([gridworld.ns, gridworld.nA])
```

```

20     for state in range(gridworld.nS):
21         action_values = np.zeros(gridworld.nA)
22         for action in range(gridworld.nA):
23             for prob, next_state, reward, is_done in gridworld.P[state]
[action]:
24                 action_values[action] += prob * \
25                     (reward+gamma*v[next_state])
26         best_action = np.argmax(action_values)
27         policy[state] = np.eye(gridworld.nA)[best_action]
28     return v, policy, iteration

```

When doing value iteration, first update the value function iteratively. Record the time of iterations that update the value function. When the change of  $V$  at a time is less than the predetermined threshold  $\theta$ , stop. After that, using Bellman Optimality Equation to get the optimal policy.

### 3.4 main.py

`main.py` instantiates the environment, and calls each module to find the shortest path.

```

1  def test_policy_iteration(gridworld):
2      v, policy, iteration = PolicyIteration.policy_iteration(gridworld)
3      print('-----Policy Iteration-----')
4      print('value Function:')
5      print(np.reshape(v, gridworld.shape))
6      print('Policy:(UP = 0, RIGHT = 1, DOWN = 2, LEFT = 3)')
7      print(np.reshape(np.argmax(policy, axis=1), gridworld.shape))
8      print('Iteration:')
9      print(iteration)
10     print('-----')
11
12
13  def test_value_iteration(gridworld):
14      v, policy, iteration = ValueIteration.value_iteration(gridworld)
15      print('-----Value Iteration-----')
16      print('value Function:')
17      print(np.reshape(v, gridworld.shape))
18      print('Policy:(UP = 0, RIGHT = 1, DOWN = 2, LEFT = 3)')
19      print(np.reshape(np.argmax(policy, axis=1), gridworld.shape))
20      print('Iteration:')
21      print(iteration)
22      print('-----')

```

`test_policy_iteration()` and `test_value_iteration()` respectively output the results of policy iteration and value iteration : the final value function, the final policy and the number of iterations.

```

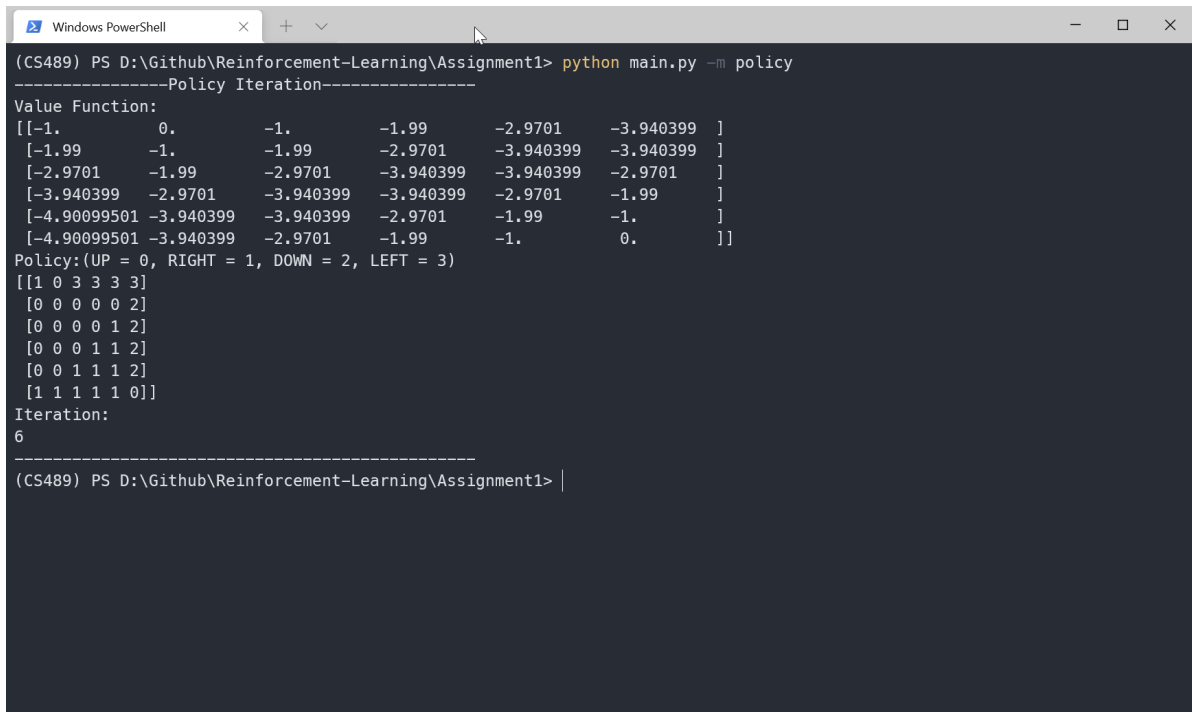
1  def main():
2      parser = argparse.ArgumentParser()
3      parser.add_argument('-m', '--method', default='policy')
4      args = parser.parse_args()
5      method = args.method
6
7      gridworld = GridworldEnv([6, 6])
8      if method == 'policy':
9          test_policy_iteration(gridworld)
10     elif method == 'value':
11         test_value_iteration(gridworld)

```

`main()` use parser to get the parameter entered on the command line, and depending on this parameter, policy or value iteration is decided.

## 4 Experimental Result

Run policy iteration:



```

(CS489) PS D:\Github\Reinforcement-Learning\Assignment1> python main.py -m policy
-----Policy Iteration-----
Value Function:
[[-1.      0.      -1.      -1.99     -2.9701   -3.940399 ]
 [-1.99    -1.      -1.99     -2.9701   -3.940399   -3.940399 ]
 [-2.9701   -1.99    -2.9701   -3.940399   -3.940399   -2.9701 ]
 [-3.940399 -2.9701   -3.940399   -3.940399   -2.9701    -1.99 ]
 [-4.90099501 -3.940399 -3.940399   -2.9701    -1.99     -1. ]
 [-4.90099501 -3.940399 -2.9701    -1.99     -1.      0. ]]
Policy: (UP = 0, RIGHT = 1, DOWN = 2, LEFT = 3)
[[1 0 3 3 3 3]
 [0 0 0 0 2]
 [0 0 0 0 1 2]
 [0 0 0 1 1 2]
 [0 0 1 1 1 2]
 [1 1 1 1 1 0]]
Iteration:
6
-----
(CS489) PS D:\Github\Reinforcement-Learning\Assignment1>

```

Run value iteration:

```
Windows PowerShell
(CS489) PS D:\Github\Reinforcement-Learning\Assignment1> python main.py -m value
-----Value Iteration-----
Value Function:
[[-1.          0.          -1.          -1.99         -2.9701        -3.940399   ]
 [-1.99        -1.          -1.99         -2.9701        -3.940399   -3.940399   ]
 [-2.9701       -1.99        -2.9701       -3.940399   -3.940399   -2.9701        ]
 [-3.940399     -2.9701       -3.940399     -3.940399   -2.9701        -1.99         ]
 [-4.90099501   -3.940399     -3.940399     -2.9701       -1.99         -1.          ]
 [-4.90099501   -3.940399     -2.9701       -1.99         -1.          0.          ]]
Policy:(UP = 0, RIGHT = 1, DOWN = 2, LEFT = 3)
[[1 0 3 3 3 3]
 [0 0 0 0 0 2]
 [0 0 0 0 1 2]
 [0 0 0 1 1 2]
 [0 0 1 1 1 2]
 [1 1 1 1 1 0]]
Iteration:
6
-----
(CS489) PS D:\Github\Reinforcement-Learning\Assignment1>
```

We can see that the final value function and final policy are the same after policy iteration and value iteration. After testing, the final policy is indeed the shortest path, which means the result is accurate.

It is worth noting that the time of iteration are both 6 in policy iteration and value iteration. In policy iteration we calculate the time of iteration in policy evaluation, while in value iteration we calculate the time of iteration in updating the value function. So we can conclude that overall the two methods are equally efficient. However, in policy iteration, the policy should be updated after each time when the policy is evaluated, which make it run slower. So value iteration is more efficient in this scenario than policy iteration.

## 5 Experimental Summary

- Systematically learn and understand policy iteration and value iteration
- Implement and successfully run the algorithm of policy iteration and value iteration based on pseudo-code
- Learn the use of `numpy` and `argparse`