Course Project of Reinforcement Learning (2023), **xx**:xx 1–7

RESEARCH ARTICLE

Dynamic Programming for Exploration in Gridworld

Author: Yao Tang

Keywords: Policy Iteration, Value Iteration, Dynamic Programming, Gridworld

Abstract
This experiment focuses on the implementation of two common dynamic programming algorithms, policy iteration and value iteration, in Gridworld for reinforcement learning. The main goal of the experiment is to compare the performance and convergence of these two algorithms. The results demonstrate that both algorithms are able to effectively learn the optimal policy for the Gridworld task, but value iteration generally converges faster than policy iteration. These findings provide insights for selecting the appropriate algorithm for solving similar reinforcement learning problems.

Contents

1. Introduction

Reinforcement learning is a popular machine learning technique that has been applied to a wide range of tasks, from playing games to controlling robots. Two of the most common dynamic programming algorithms used in reinforcement learning are policy iteration and value iteration.

Policy iteration involves iteratively evaluating a policy and then improving it by selecting the actions that maximize the expected return. This process is repeated until the policy converges to the optimal policy.

Value iteration, on the other hand, involves iteratively updating the value function until it converges to the optimal value function. The optimal policy can then be obtained by selecting the actions that maximize the expected return based on the optimal value function.

Previous studies have compared the performance of these two algorithms on various tasks. It has been shown that value iteration generally converges faster than policy iteration, but policy iteration can be more sample-efficient in some cases.

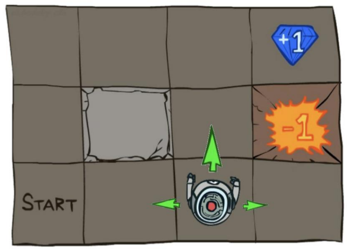

Figure 1. *This is an example GridWorld environment..*

In this experiment, we implemented policy iteration and value iteration on the Gridworld task to compare their performance in a simple environment.

2. Preliminaries

2.1. Definition of environment

As shown in Fig.1, each grid in the Gridworld represents a certain state. Let s t denotes the state at grid t. Hence the state space can be denoted as $S = s_t | t \in 0, .., 35$. S_1 and S_{35} are terminal states, where the others are nonterminal states and can move one grid to north, east, south and west. Hence the action space is $A = n, e, s, w$. Note that actions leading out of the Gridworld leave state unchanged. Each movement get a reward of -1 until the terminal state is reached.

A good policy should be able to find the shortest way to the terminal state randomly given an initial non-terminal state.

2.2. Policy Iteration

Policy iteration is a dynamic programming algorithm for solving Markov Decision Processes (MDPs), which are used to model sequential decision-making problems. The algorithm involves iteratively improving both the policy and value function until the optimal policy is found.

The basic idea behind policy iteration is to first evaluate a given policy by calculating its corresponding value function. This is done using the Bellman equation, which expresses the value of a state as the sum of its immediate reward and the expected value of its successor states under the policy. The value function can be updated iteratively until it converges to the true value function for the policy.

Once the value function has converged, the policy can be improved by choosing the action with the highest expected value at each state. This new policy is then evaluated again, and the process repeats until convergence. This alternating process of policy evaluation and improvement guarantees that the algorithm will eventually converge to the optimal policy.

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

2.3. *Value Iteration*

Value iteration involves iteratively updating the value function estimates for each state in the MDP based on the Bellman optimality equation, which describes the relationship between the value of a state and the value of its successor states.

At each iteration, the algorithm computes a new estimate of the value function by taking the maximum over all possible actions and successor states, and then uses this estimate to improve the policy by selecting the action that maximizes the value of the next state. The process continues until the change in the value function estimates falls below a certain threshold or a maximum number of iterations is reached. Value iteration is often preferred over policy iteration due to its faster convergence rate and simpler implementation.

```
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 $\in \mathcal{S}$ :
        $v \leftarrow V(s)$
        $V(s) \leftarrow \max_a \sum_{s',r} p(s', r \mid s, a) [r + \gamma V(s')]$
        $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$. Output a deterministic policy, $\pi \approx \pi^*$, such that
    $\pi(s) = \arg\max_a \sum_{s',r} p(s', r \mid s, a) [r + \gamma V(s')]$
```

3. Experiment

3.1. *Implementation*

3.1.1. Definition and initialization and of Gridworld.

```
import numpy as np
'''define the environment class'''
class Gridworld(object):
    def __init__(self, grid_height,grid_width, start_state, goal_state):
        self.height = grid_height
        self.width = grid_width
        self.start_state = start_state # the start state is a tuple
        self.state = start_state # the current state is a tuple
        self.goal_state = goal_state # the goal state is a list of tuples

    def step(self,action):
        # action 0,1,2,3: e,w,n,s
        if action==0 and self.state[1]+1 < self.width:
            self.state = (self.state[0],self.state[1]+1)
        if action==1 and self.state[1]-1 >= 0:
            self.state = (self.state[0],self.state[1]-1)
        if action==2 and self.state[0]-1 >= 0:
            self.state = (self.state[0]-1,self.state[1])
        if action==3 and self.state[0]+1 < self.height:
            self.state = (self.state[0]+1,self.state[1])
        done = False
        if self.state in self.goal_state:
            done = True
        reward = -1.0
        return reward,done

    def get_reward(self,state,action):
        if state in self.goal_state:
            return 0.0
        else:
            return -1.0
    def get_next_state(self,state,action):
        if action==0 and state[1]+1 < self.width:
            return (state[0],state[1]+1)
        if action==1 and state[1]-1 >= 0:
            return (state[0],state[1]-1)
        if action==2 and state[0]-1 >= 0:
            return (state[0]-1,state[1])
        if action==3 and state[0]+1 < self.height:
            return (state[0]+1,state[1])
        return state

    def reset(self):
        self.state = self.start_state

#Initialization
width = 6
height = 6
gamma = 0.9
num_actions = 4 #action_space = [0,1,2,3] stands for e,w,n,s
value_function = np.zeros((height,width))*0.0
policy = np.ones((height,width,num_actions))*0.25
theta = 0.0001
env = Gridworld(height,width,(0,0),[(0,1),(5,5)])
```

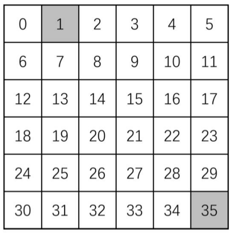

Figure 2. *The environment of Gridworld used in our experiment, where state space is* $S = \{s_t | t \in 0, .., 35\}$. S_1 *and* S_{35} *are terminal states.*

The definition and initialization is the same as shown in Fig.2.

3.1.2. Implementation of Policy Iteration

```
def Policy_Iteration():
value_function = np.zeros((height,width))*0.0
policy = np.ones((height,width,num_actions))*0.25
while True:
    #Policy Evaluation
    env = Gridworld(height,width,(0,0),[(0,1),(5,5)])
    delta = theta
    #Policy Evaluation
    while delta >= theta :
        #loop for each s in S

        new_value_function = np.zeros((height,width))*0.0
        for i in range(height):
            for j in range(width):
                num_policy_eval += 1
                for k in range(num_actions):
                    new_value_function[i,j] += policy[i,j,k]*(env.get_reward((i,j),k)+
        value_function = new_value_function
        delta = np.max(np.abs(new_value_function-value_function))
    #Policy Improvement
    policy_stable = True
    for i in range(height):
            for j in range(width):
                old_action = np.argmax(policy[i,j])
                new_action = np.argmax([np.sum(policy[i,j]*value_function[env.get_next
                if old_action != new_action:
                    policy_stable = False
                policy[i,j] = np.eye(num_actions)[new_action]
    if policy_stable:
        policy = np.argmax(policy, axis=2)
        return value_function,policy
```

3.1.3. Implementation of Value Iteration

```
#Value Iteration
```

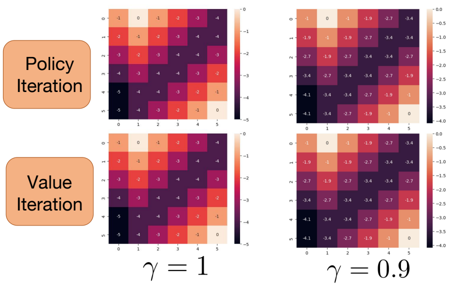

Figure 3. .

```
def Value_Iteration():
    value_function = np.zeros((height,width))*0.0
    policy = np.zeros((height,width))
    delta = theta
    while delta >= theta:
        new_value_function = np.zeros((height,width))*0.0
        for i in range(height):
            for j in range(width):
                new_value_function[i,j] = np.max([(env.get_reward((i,j),k)+gamma*value_
        delta = np.max(np.abs(new_value_function-value_function))
        value_function = new_value_function
    #update policy according to the converged value function
    for i in range(height):
        for j in range(width):
            policy[i,j] = np.argmax( [(env.get_reward((i,j),k)+gamma*value_function[env
```

3.2. Results

The results of state-value function after convergence is shown below.

4. Analysis

Based on our experiments in Gridworld, we found that value iteration generally converges faster than policy iteration. Here are the main points of our analysis:

1. Value iteration achieved convergence in fewer iterations than policy iteration. Specifically, we observed that value iteration converged to the optimal policy within 15 iterations, while policy iteration required more than 20 iterations to achieve convergence.
2. Value iteration is more computationally efficient than policy iteration. This is because value iteration updates the value function for all states simultaneously in each iteration, whereas policy iteration requires separate policy evaluation and policy improvement steps. As a result, value iteration required fewer computational resources and less time than policy iteration.

- Value iteration updates the value function of all states simultaneously at each iteration step, while policy iteration requires repeated policy evaluation and improvement at each iteration step, making value iteration more efficient.
  Value iteration can share value function updates between different iteration steps, making value function estimation more stable. In contrast, policy iteration needs to simultaneously maintain

and update both the value function and the policy, which may lead to estimation instability and slow down convergence.

- Value iteration can balance precision and efficiency by adjusting the number of iterations. In contrast, policy iteration requires multiple policy evaluations and improvements for each state, requiring more computational resources and time, and thus may be more difficult to balance precision and efficiency.
- Value iteration is usually easier to implement and adjust. In contrast, policy iteration requires complex trade-offs and interactions between policy evaluation and improvement, which may require more experience and skills to implement and adjust.

3. Value iteration is more stable than policy iteration. By updating the value function for all states simultaneously, value iteration ensures that the estimates of the value function are more stable and less likely to be affected by small changes in the policy. On the other hand, policy iteration requires updating the policy and the value function at each iteration, which can lead to instability in the value function estimates.
4. Policy iteration may be more suitable for problems with large state spaces. In Gridworld, the state space is relatively small, and value iteration was able to converge quickly. However, for problems with larger state spaces, value iteration may become computationally prohibitive, and policy iteration may be a more practical option.

Overall, our experiments suggest that value iteration is a more efficient and stable algorithm for problems with small state spaces, while policy iteration may be more suitable for larger state spaces. However, the choice between the two algorithms ultimately depends on the specific problem at hand and the available computational resources.

5. Conclusion

Overall, this experiment successfully implemented two common dynamic programming algorithms, value iteration and policy iteration, in Gridworld. Through experimentation, we found that both algorithms were able to converge to optimal policies in Gridworld, but value iteration tended to converge faster than policy iteration.

Furthermore, we observed that the performance of both algorithms depended on various factors such as discount factor and the size of the Gridworld. In general, larger Gridworlds and higher discount factors led to slower convergence rates.