Weekly Status Report (Jan 26 - Feb 2)

Yuetong Liu

02/02/2020

Abstract

Dynamic Programming is a collection of algorithms that can be used to compute optimal policies given a perfect model of the environment as a Markov decision process. It actually consists of two different versions of how it can be implemented which are policy iteration and value iteration. This report concludes the implementation of both algorithms and performance test results in the chess game environment.

1 Accomplishments (Summary)

Policy iteration and value iteration are applied to four agents: king, knight, bishop, and rook. For all agents, both two methods significantly shorten the paths between 2 squares, while policy iteration yields more stable results but value iteration takes less time to converge.

2 Introduction

Tackling chess is challenging because of its huge state space. Therefore I start with finding the shortest path between 2 squares on a chessboard. This problem has a small state space, therefore allows us to tackle this with simple RL algorithms. Moreover, moving chess effectively is an intermediate step to solve the chess problem. This time I used two Dynamic Programming technics - policy iteration and value iteration to solve the problem. As model-based reinforcement learning methods, policy iteration/value iteration tries to understand the chess game and create a model to represent it.

3 Mathematical Development

Below is the algorithm for policy iteration, where ϵ is a small positive number determining the accuracy of the estimation. Section 4 contains python code for this algorithm and its transform(value iteration).

- Initialize policy π_0 randomly
- Iterate until π_i converge to π^*

  1. Policy Evaluation:

     $\delta = 0$

     while $\delta >= \epsilon$:

     $v = V(s)$

     $V(s) = \sum P(s, \pi(s), s')[r(s, \pi(s), s') + \gamma V(s')]$ (*)

     $\delta = max(\delta, v - V(S)$

2. Policy Improvement:

   $\pi_{i+1}(s) = argmax \sum P(s, a, s')[r(s, a, s') + \gamma V^*(s')]$ (**)

   if $\pi_{i+1}(s) \neq \pi_i(s)$: go back to Policy Evaluation

4 Experimental Approach and Results

4.1 Experimental Approach

This part includes python code for algorithms mention in section 3 and the comments on codes are highlighted in green.

4.1.1 State Evaluation

A state (s) is as valuable (V) as the successor state (s') plus the reward (R) for going from s to s'. Since there can be mulitple actions (a) and multiple successor states they are summed and weighted by their probability (pi). In a non-deterministic environment, a given action could result in multiple successor states. We don't have to take this into account for this problem because move chess is a deterministic game. Successor state values are discounted with discount factor (gamma) that varies between 0 and 1.

```
def evaluate_state(self, state, gamma=0.9, synchronous=True):
        Calculates the value of a state based on the successor states and
           the immediate rewards.Args:state:  tuple of 2 integers 0-7
           representing the stategamma:  float, discount factorsynchronous:
           BooleanReturns:  The expected value of the state under the
           current policy.
        greedy_action_value = np.max(self.agent.policy[state[0],
           state[1], :]) # get max action value at give state
        greedy_indices = [i for i, a in enumerate(self.agent.policy
           [state[0], state[1], :]) if
                             a == greedy_action_value]  # list the
                                index of all actions with max action
```

```
                    value
        prob = 1 / len(greedy_indices)  # probability of an action occuring
        state_value = 0 # set V(S) = 0
        for i in greedy_indices: # for all actions with max action value
            self.env.state = state  # reset state to the one being evaluated
            reward, episode_end = self.env.step(self.agent.action_space[i]) # get reward
            if synchronous: # get successor state value
                successor_state_value = self.agent.value_function_prev[self.env.state]
                # if synchronous, successor state value is in the same iteration of policy evaluation
            else:
                # if not,  successor state value could be previous or the current value funtion,or combined
                successor_state_value = self.agent.value_function[self.env.state] # otherwise
            state_value += (prob * (
                    reward + gamma * successor_state_value))  # sum up rewards and discounted successor state value in equation(*)
        return state_value
```

4.1.2 Policy Evaluation

$python Apply state evaluation to all states. def evaluate_policy(self, gamma = 0.9, synchronous = True) :$ $self.agent.value_function_prev = self.agent.value_function.copy() For synchronous updates for row in range(self.age$ $in each row for col in range(self.agent.value_function.shape[1]) : in each column self.agent.value_function[row, col] =$ $self.evaluate_state((row, col), gamma = gamma, in each action, update value function synchronous =$ $synchronous)$

4.1.3 Policy Improvement

Policy Improvement is the act of making the policy greedy with respect to the value function.

```
def improve_policy(self):
        Finds the greedy policy w.r.t.  the current value function

        self.agent.policy_prev = self.agent.policy.copy() # get pi(i)
        for row in range(self.agent.action_function.shape[0]): # for each row
            for col in range(self.agent.action_function.shape[1]): # for each column
                for action in range(self.agent.action_function.shape[2]): # for each action
                    self.env.state = (row, col)  # reset state to the one being evaluated
                    reward, episode_end = self.env.step(self.agent.action_space[action]) # make step based on action
                    successor_state_value = 0 if episode_end else self.agent.value_function[self.env.state] #
```

```
                    update successor state value
                    self.agent.policy[row, col, action] = reward + successor_state_value # get pi(i+1) using (**)

                max_policy_value = np.max(self.agent.policy[row, col, :]) # get max policy value at given state
                max_indices = [i for i, a in enumerate(self.agent.policy[row, col, :]) if a == max_policy_value] # get the index of policy with max policy value
                for idx in max_indices:
                    self.agent.policy[row, col, idx] = 1 # 1 if the policy value is max, otherwise 0
```

4.1.4 Policy Iteration

Policy Iteration finds the optimal policy by doing policy evaluation and policy improvement until the policy is stable.

```
def policy_iteration(self, eps=0.1, gamma=0.9, iteration=1, k=32, synchronous=True):
        Finds the optimal policyArgs:eps:  float, exploration rategamma:
            float, discount factoriteration:  the iteration numberk:  (int)
            maximum amount of policy evaluation iterationssynchronous:
            (Boolean) whether to use synchronous are asynchronous
            back-upsReturns:
        policy_stable = True
        print(iteration:, iteration, )
        print( policy:)
        self.visualize_policy()
```

```python

        # Evaluate Policy
        print()
        value_delta_max = 0
        for _ in range(k): # max number of iteration for policy evaluation
            self.evaluate_policy(gamma=gamma, synchronous=synchronous) # update V(s)
            value_delta = np.max(np.abs(self.agent.value_function_prev - self.agent.value_function)) # get delta
            value_delta_max = value_delta
            if value_delta_max < eps: # terminate the loop if delta is small
                break
        print(Value function for this policy:)
        print(self.agent.value_function.round().astype(int))
        action_function_prev = self.agent.action_function.copy()

        # Improve Policy
        print( Improving policy:)
        self.improve_policy()

        # Check if Stable
        policy_stable = self.agent.compare_policies() < 1 #check if policy function is similar with previous iteration
        print(policy diff:, policy_stable)
```

```
        if not policy_stable and iteration < 1000: # if policy is not stable, restart the whole function
            iteration += 1
            self.policy_iteration(iteration=iteration)
        elif policy_stable:
            print(Optimal policy found in, iteration, steps of policy evaluation)
        else:
            print(failed to converge.)
```

4.1.5 Value Iteration

Theory Value iteration is nothing more than a simple parameter modification to policy iteration. Remember that policy iteration consists of policy evaluation and policy improvement. The policy evaluation step does not necessarily have to be repeated until convergence before we improve our policy. If we use only 1 iteration instead we call it value iteration.

4.2 Results

I selected four agents for training using both policy iteration and value iteration.

- King can move exactly one square horizontally, vertically, or diagonally.
- Rook can move any number of vacant squares vertically or horizontally.
- Bishop can move any number of vacant squares in any diagonal direction.
- Knight can move one square along any rank or file and then at an angle.

Each scenario is repeated 50 times to reduce bias. Here is an example using value iteration to train the agent “King” 50 times. The output is the number of steps the King take from (0,0) to (5, 7)

```python
eva3 = [] # a empty list to store number of step to get to desired position
for i in range(50): # repeat 50 times
  p = Piece(piece='king') # select a chess agent (knight, bishop or rook)
  env = Board() # 8*8 chess board
  r = Reinforce(p,env) # place king on the board at (0,0) by default
  r.policy_iteration(k=1) # when k=1, it's value iteration
  states, actions, rewards = r.play_episode(state = (0,0),
                    max_steps=1e3,
                    epsilon=0.1) # move chess until it gets to (5,7) by default

  eva3.append(len(actions)) # add number of steps to the list
eva3 # print list
```

Output: [11, 8, 9, 10, 8, 8, 8, 8, 9, 8, 8, 8, 8, 10, 12, 10, 9, 8, 8, 8, 8, 10, 11, 8, 8, 8, 11, 8, 12, 8, 12, 8, 8, 8, 10, 8, 8, 8, 8, 8, 10, 10, 10, 8, 8, 10, 9, 8, 12, 9]

The tables show the mean and variance of the total number of steps that an agent takes from (0,0) to (7,5).

Method	King	Knight	Bishop	Rook
None	213.2	169.38	157.36	141.82
Policy Iteration	8.96	5.58	3.28	3.22
Value Iteration	9.10	5.42	3.28	3.22

Table 1: Mean of NO. steps

None	24388.48	22948.03	30279.23	17110.63
Policy Iteration	1.52	1.47	0.36	0.29
Value Iteration	2.65	1.47	0.32	0.21

Table 2: Variance of NO. steps

According to the result table, the mean and variance for each agent reduce significantly after policy iteration/value iteration, while the difference between policy iteration and value iteration is not significant.

5 Next Steps

It is expensive to train them since they required iterations to converge. Moreover, policy evaluation calculates the state value by backing up the successor state values and the transition probabilities to those states. However, these probabilities are usually unknown. To work with unknown environments some model-free techniques such as Monte Carlo Control could be an alternative approach.

References

[1] Richard S. Sutton and Andrew G. Barto. 2018. Reinforcement Learning: An Introduction. A Bradford Book, Cambridge, MA, USA.

[2] Groen, A. 2019. Reinforcement Learning Chess. GitHub repository, https://github.com/arjangroen/RLC