Dynamic Programming

Carlos Matheus Barros da Silva, *Computer Engineering Bachelor Student of ITA*

Prof. Marcos Ricardo Omena de Albuquerque Máximo

Abstract

This paper evaluates the core concepts of the behind the Reinforcement Learning theory on the environment of the Markov Decision Process (MDP) and Dynamic programming.

It was observed how policy iteration and value itaration works and how they are affected by the probability of correctly executing the chosen action factor (α) and the discount factor(γ).

It was observed that for a deterministic world ($\gamma = 1$ and $\alpha = 1$), the learning is sensibly slower, which means that was required much more iterations in order to the value converge when compared to a little decrease on γ and α ($\gamma = 0.98$ and $\alpha = 0.8$).

Index Terms

Reinforcement Learning, Markov Decision Process, MDP, Dynamic Programming, policy, states

I. Introduction

Reinforcement learning (RL) is an area of machine learning concerned with how software agents ought to take actions in an environment so as to maximize some notion of cumulative reward. Reinforcement learning is one of three basic machine learning paradigms, alongside supervised learning and unsupervised learning.

It differs from supervised learning in that labelled input/output pairs need not be presented, and sub-optimal actions need not be explicitly corrected. Instead the focus is finding a balance between exploration (of uncharted territory) and exploitation (of current knowledge).

The environment is typically formulated as a Markov decision process (MDP), as many reinforcement learning algorithms for this context utilize dynamic programming techniques. The main difference between the classical dynamic programming methods and reinforcement learning algorithms is that the latter do not assume knowledge of an exact mathematical model of the MDP and they target large MDPs where exact methods become infeasible.

II. Dynamic Programming Implementation

The implementation was based on the file *dynamic programming*. The essence of the implementation is shown from Code 1 to Code 5.

```
value = np.copy(initial_value)
policy = np.copy(initial_policy)

for i in range(num_iterations):
    value = policy_evaluation(grid_world, value, policy, evaluations_per_policy, epsilon)
    new_policy = greedy_policy(grid_world, value)
```

```
        if not is_policy_equal(policy, new_policy):
            policy = new_policy
        else:
            break

    return value, policy
```

Code 1. Code of function *policy iteration*, executes policy iteration for a grid world.

```
    dimensions = grid_world.dimensions
    possible_actions = [STOP, UP, RIGHT, DOWN, LEFT]
    old_value = initial_value
    value = np.copy(initial_value)

    for _ in range(num_iterations):
        for i in range(dimensions[0]):
            for j in range(dimensions[1]):
                current_state = (i, j)
                max_val = -float('inf')

                for action in possible_actions:
                    rew = grid_world.reward(current_state, action)
                    successors_states = grid_world.get_valid_sucessors(current_state)
                    val_sum = 0
                    for successor in successors_states:
                        prob = grid_world.transition_probability(current_state, action, successor)
                        val = old_value[successor[0]][successor[1]]
                        val_sum += prob * val
                    max_val = max(max_val, rew + grid_world.gamma * val_sum)

                value[current_state[0]][current_state[1]] = max_val

        if changed_val(old_value, value, dimensions, epsilon):
            old_value = value
            value = np.copy(old_value)
        else:
            break

    return value
```

Code 2. Code of function *value iteration*, executes value iteration for a grid world.

```
    dimensions = grid_world.dimensions
    value = evaluate(grid_world, initial_value, policy, dimensions, num_iterations, epsilon)
    return value
```

Code 3. Code of function *policy evaluation*, executes policy evaluation fora a policy executed on a grid world.

```
    old_value = initial_value
    value = np.copy(initial_value)

    for _ in range(num_iterations):
        for i in range(dimensions[0]):
            for j in range(dimensions[1]):
                current_state = (i, j)
```

```
                possible_actions = policy[current_state[0]][current_state[1]]
                sum_val = 0
                for action in range(len(possible_actions)):
                    rew = grid_world.reward(current_state, action)
                    successors_states = grid_world.get_valid_sucessors(current_state)
                    val_sum = 0
                    for successor in successors_states:
                        prob = grid_world.transition_probability(current_state, action, successor)
                        val = old_value[successor[0]][successor[1]]
                        val_sum += prob * val
                    sum_val += (rew + grid_world.gamma * val_sum) * possible_actions[action]

                value[current_state[0]][current_state[1]] = sum_val

        if changed_val(old_value, value, dimensions, epsilon):
            old_value = value
            value = np.copy(old_value)
        else:
            break

    return value
```

Code 4. Code of function *evaluate*, will evaluate for the *policy evaluation* function.

```
    for j in range(dimensions[0]):
        for i in range(dimensions[1]):
            if abs(new_value[i][j] - old_value[i][j]) > epsilon:
                return True
    return False
```

Code 5. Code of function *changed val*, will check whether the value changed some of its elements values or not.

III. Dynamic Programming Analysis

A. Dynamic Programming Analysis With $\gamma = 1$ and $\alpha = 1$

For this case, the output can be seen on the Code 6.

```
Evaluating random policy, except for the goal state, where policy always executes stop:
Value function:
[  -384.09,  -382.73,  -381.19,     *   ,  -339.93,  -339.93]
[  -380.45,  -377.91,  -374.65,     *   ,  -334.92,  -334.93]
[  -374.34,  -368.82,  -359.85,  -344.88,  -324.92,  -324.93]
[  -368.76,  -358.18,  -346.03,     *   ,  -289.95,  -309.94]
[     *   ,  -344.12,  -315.05,  -250.02,  -229.99,     *   ]
[  -359.12,  -354.12,     *   ,  -200.01,  -145.00,     0.00]
Policy:
[ SURDL , SURDL , SURDL ,    *   , SURDL , SURDL ]
[ SURDL , SURDL , SURDL ,    *   , SURDL , SURDL ]
[ SURDL , SURDL , SURDL , SURDL , SURDL , SURDL ]
[ SURDL , SURDL , SURDL ,    *   , SURDL , SURDL ]
[    *  , SURDL , SURDL , SURDL , SURDL ,    *  ]
[ SURDL , SURDL ,    *  , SURDL , SURDL ,    S  ]
```

```
Value iteration:
Value function:
[ -10.00,  -9.00,  -8.00,   *  ,  -6.00,  -7.00]
[  -9.00,  -8.00,  -7.00,   *  ,  -5.00,  -6.00]
[  -8.00,  -7.00,  -6.00,  -5.00,  -4.00,  -5.00]
[  -7.00,  -6.00,  -5.00,   *  ,  -3.00,  -4.00]
[   *  ,  -5.00,  -4.00,  -3.00,  -2.00,   *  ]
[  -7.00,  -6.00,   *  ,  -2.00,  -1.00,   0.00]
Policy:
[  RD  ,  RD  ,  D  ,  *  ,  D  ,  DL  ]
[  RD  ,  RD  ,  D  ,  *  ,  D  ,  DL  ]
[  RD  ,  RD  ,  RD  ,  R  ,  D  ,  DL  ]
[  R  ,  RD  ,  D  ,  *  ,  D  ,  L  ]
[  *  ,  R  ,  R  ,  RD  ,  D  ,  *  ]
[  R  ,  U  ,  *  ,  R  ,  R  ,  SURD  ]
```

```
Policy iteration:
Value function:
[ -10.00,  -9.00,  -8.00,   *  ,  -6.00,  -7.00]
[  -9.00,  -8.00,  -7.00,   *  ,  -5.00,  -6.00]
[  -8.00,  -7.00,  -6.00,  -5.00,  -4.00,  -5.00]
[  -7.00,  -6.00,  -5.00,   *  ,  -3.00,  -4.00]
[   *  ,  -5.00,  -4.00,  -3.00,  -2.00,   *  ]
[  -7.00,  -6.00,   *  ,  -2.00,  -1.00,   0.00]
Policy:
[  RD  ,  RD  ,  D  ,  *  ,  D  ,  DL  ]
[  RD  ,  RD  ,  D  ,  *  ,  D  ,  DL  ]
[  RD  ,  RD  ,  RD  ,  R  ,  D  ,  DL  ]
[  R  ,  RD  ,  D  ,  *  ,  D  ,  L  ]
[  *  ,  R  ,  R  ,  RD  ,  D  ,  *  ]
[  R  ,  U  ,  *  ,  R  ,  R  ,  SURD  ]
```

Code 6. Output for *test dynamic programming* for $\gamma = 1$ and $\alpha = 1$ case.

B. Dynamic Programming Analysis With $\gamma = 0.98$ *and* $\alpha = 0.8$

For this case, the output can be seen on the Code 7.

```
Evaluating random policy, except for the goal state, where policy always executes stop:
Value function:
[ -47.19,  -47.11,  -47.01,   *  ,  -45.13,  -45.15]
[ -46.97,  -46.81,  -46.60,   *  ,  -44.58,  -44.65]
[ -46.58,  -46.21,  -45.62,  -44.79,  -43.40,  -43.63]
[ -46.20,  -45.41,  -44.42,   *  ,  -39.87,  -42.17]
[   *  ,  -44.31,  -41.64,  -35.28,  -32.96,   *  ]
[ -45.73,  -45.28,   *  ,  -29.68,  -21.88,   0.00]
Policy:
[ SURDL  ,  SURDL  ,  SURDL  ,  *  ,  SURDL  ,  SURDL  ]
[ SURDL  ,  SURDL  ,  SURDL  ,  *  ,  SURDL  ,  SURDL  ]
[ SURDL  ,  SURDL  ,  SURDL  ,  SURDL  ,  SURDL  ,  SURDL  ]
[ SURDL  ,  SURDL  ,  SURDL  ,  *  ,  SURDL  ,  SURDL  ]
[  *  ,  SURDL  ,  SURDL  ,  SURDL  ,  SURDL  ,  *  ]
[ SURDL  ,  SURDL  ,  *  ,  SURDL  ,  SURDL  ,  S  ]
```

```
Value iteration:
Value function:
[  -11.65,  -10.78,   -9.86,    *  ,   -7.79,   -8.53]
[  -10.72,   -9.78,   -8.78,    *  ,   -6.67,   -7.52]
[   -9.72,   -8.70,   -7.59,   -6.61,   -5.44,   -6.42]
[   -8.70,   -7.58,   -6.43,    *  ,   -4.09,   -5.30]
[    *  ,   -6.43,   -5.17,   -3.87,   -2.76,    *  ]
[   -8.63,   -7.58,    *  ,   -2.69,   -1.40,    0.00]
Policy:
[   D  ,   D  ,   D  ,   *  ,   D  ,   D  ]
[   D  ,   D  ,   D  ,   *  ,   D  ,   D  ]
[   RD ,   D  ,   D  ,   R  ,   D  ,   D  ]
[   R  ,   RD ,   D  ,   *  ,   D  ,   L  ]
[   *  ,   R  ,   R  ,   D  ,   D  ,   *  ]
[   R  ,   U  ,   *  ,   R  ,   R  ,   S  ]
```

```
Policy iteration:
Value function:
[  -11.66,  -10.80,   -9.89,    *  ,   -7.81,   -8.55]
[  -10.75,   -9.82,   -8.80,    *  ,   -6.68,   -7.53]
[   -9.77,   -8.72,   -7.61,   -6.62,   -5.45,   -6.43]
[   -8.73,   -7.60,   -6.44,    *  ,   -4.09,   -5.31]
[    *  ,   -6.44,   -5.18,   -3.88,   -2.76,    *  ]
[   -8.67,   -7.60,    *  ,   -2.69,   -1.40,    0.00]
Policy:
[   D  ,   D  ,   D  ,   *  ,   D  ,   D  ]
[   D  ,   D  ,   D  ,   *  ,   D  ,   D  ]
[   R  ,   D  ,   D  ,   R  ,   D  ,   D  ]
[   R  ,   RD ,   D  ,   *  ,   D  ,   L  ]
[   *  ,   R  ,   R  ,   D  ,   D  ,   *  ]
[   R  ,   U  ,   *  ,   R  ,   R  ,   S  ]
```

Code 7. Output for *test dynamic programming* for $\gamma = 0.98$ and $\alpha = 0.8$ case.

IV. Conclusion

It is clear, therefore, that this experiment exemplifies that *value iteration* algorithm allows in a given world define a value grid that leads to an optimal policy by getting a greedy solution. It also shows that this can be achieved by iterate in *policy itaration*, by vary between evaluating a policy and getting a greedy approach, which leads to an optimal policy much quicker (with fewer iterations) when compared to *value iteration*.

This paper also shown how policy iteration and value itaration works and how they are affected by the probability of correctly executing the chosen action factor (α) and the discount factor(γ).

It was observed that for a deterministic world ($\gamma = 1$ and $\alpha = 1$), the learning is sensibly slower, which means that was required much more iterations in order to the value converge when compared to a little decrease on γ and α ($\gamma = 0.98$ and $\alpha = 0.8$), which is as expected for this problem, since this increases the exploration and allows the algorithm find a better solution quicker.