Reinforcement Learning

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, 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

In this analysis was used two test functions: *sum greater than* and *xor*. This initial analysis is not using regularization.

The Keras' Neural Network performed regular on *sum greater than* function. On the graphs represented by the images from Image **??** to Image **??**. It is possible to verify that the result is overfitted on the Image **??** and the convergence is not so fast on the Image **??**.

The Neural Network performed regular on *xor* function. On the graphs represented by the images from Image **??** to Image **??**. It is also possible to see that in this case, it also heaped some overfit causing some distortion and leading to some mistakes on the data set on the graph represented by the Image **??**.

B. Keras' Neural Network Analysis without regularization

In this analysis was used the same two test functions: *sum greater than* and *xor*. But now using regularization $\lambda_{l_2} = 0.002$.

The Keras' Neural Network performed well on *sum greater than* function. On the graphs represented by the images from Image **??** to Image **??**. It is possible to verify that the result now is much less overfitted on the Image **??**, it is much softer, and the convergence is now faster on the Image **??**.

The Neural Network performed well on *xor* function. On the graphs represented by the images from Image **??** to Image **??**. It is also possible to see that in this case, it also heaped much less overfit leading to a much softer image on the graph represented by the Image **??**.

C. Keras' Neural Network Analysis in Imitation Learning

In order to do the Imitation Learning, it was used the Code **??** implementation.

It was made using Keras, not using regularization, and using mean squared error.

The result of this Neural Network on the robot movement can be seen on the graphs from Image **??** to Image **??**.

IV. Conclusion

It was clear, therefore, that the Keras' Neural Network worked as expected. Both test cases (greater than function and xor function) the Neural Network worked as well, with a much better result with regularization, because without regularization the results were overfitted.

For the Imitation Learning with Keras, the results were good in some cases and very precise in most some cases.