Reinforced Learning and Its Capabilities

Israel Nolazco
israelnolazco@lewisu.edu
CPSC-57200-02, Fall Two
Artificial Intelligence 2
Lewis University

I. Introduction

Reinforced Learning is a subset of Machine Learning and Artificial Intelligence and much like the name suggests it deals with the allowance of a system to learn from its "mistakes" and develop strategies to obtain a reward as efficiently as possible. The system operates in a way where the software agent takes sought after actions that will maximize its reward. Keep in mind that this subset deals with a lot of trial and error, thus a perfect system or perfect solution would ultimately take an infinite number of trials. However, this does not mean an infinite amount of time must be allocated in order to achieve an optimal solution. In essence, the solution from each trial should reduce the range of error until achieving the optimal solution. Another way to think of this method is to think of a basketball being drop. Each bounce, its bounce height decreases by half the distance than its initial drop. Each half will be significantly smaller than the other until the distance between bounce x to bounce y cannot be recognized. Albeit, instead of letting the system naturally get to an optimal solution; in this document, we are going to primarily focus on two techniques, utilize in Reinforce Learning. The first being, Adaptive Dynamic Programming, and the second Temporal-Difference Learning. The goal is to achieve an optimal solution as efficiently as possible and understand how those two techniques achieve this goal and in what scenarios would one triumph over the other.

II. Methods

In this document, we are going to use the programming language Python and the source code with the primary function of understanding and comprehending the work found in Russell And Norvig's "Artificial Intelligence - A Modern Approach," which were found on a Github Page.[1] This source code is going to be utilize for a simple environment of a 4X3 environment with an obstacle inside the environment. This is shown in Figure 1. The goal is weighting the utility of finding the optimal solution. Furthermore as explained in our introduction there will be two main types of reinforced learning. One being from Adaptive Dynamic Programming in which the agent takes advantage of the constraints among the utilities of states by learning the transition model that connects them and solves the corresponding Markov decision process utilizing a dynamic programming method. In essence, we are using Linear Algebra to weight the probabilities by measuring and taking into consideration the current and past state. Therefore, predicting the best action. As more interaction happens, the probabilities will match that of the optimal goal.[2] However, since we are talking of linear algebra and calculating probabilities it is heavy in computational resources. Thus, is best to utilize this system in a small environment rather than an extensive one. Albeit it only takes a small number of trials until the optimal goal is achieved. Secondly, we are going to make use of Temporal-Difference Learning. This method works by adjusting the utility estimate towards the ideal equilibrium. How does it do this? Well, this is done by looking at [2] expected closeness between the utilities of two consecutive states. In other words, it looks for the constrain between the neighboring state, if state is close to high utility then the utility of the state must also be high. This allows it to be lightweight in computational resources wise and a bit straightforward in applying this function. Although, it may take a larger number of trials in order for its results to get closer and closer to the optimal goal.

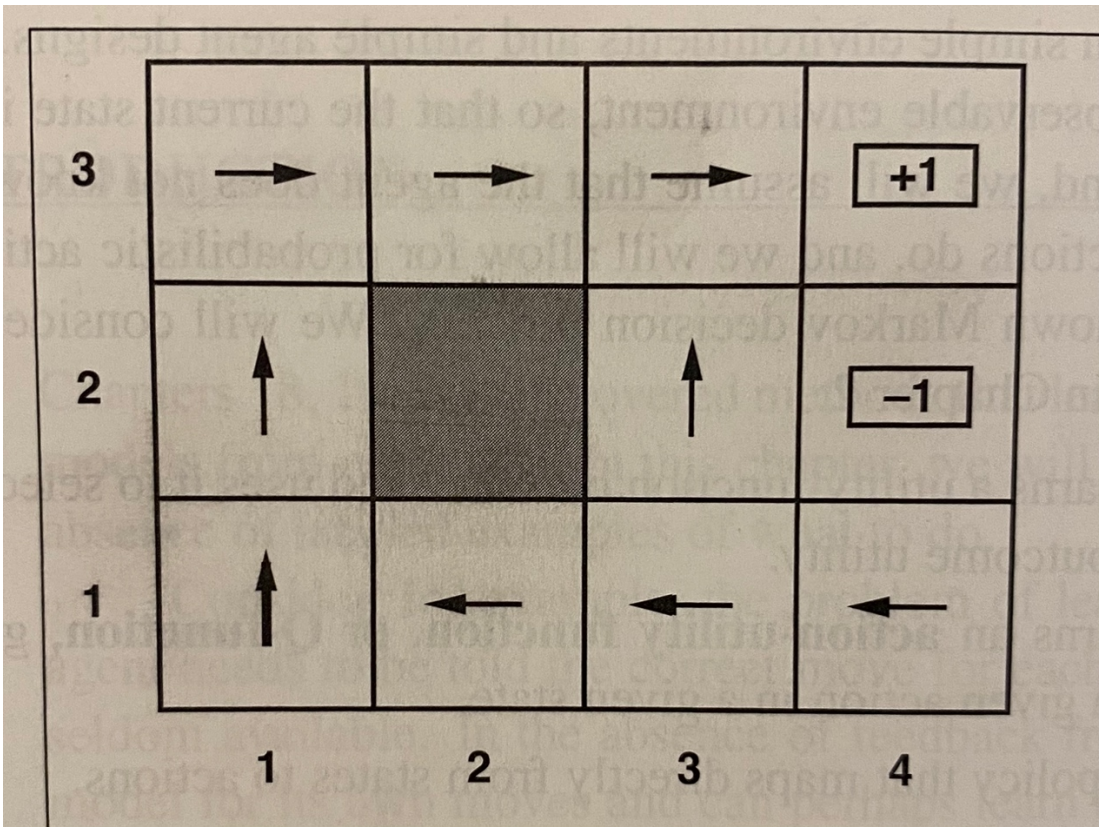


Figure 1. A 4X3 environment with an obstacle and with a policy that offers a reward of R(s) = -0.04

III. Analysis

In order to test out our hypothesis of which method of reinforce learning approaches an optimal goal we first ran both the Adaptive Dynamic Programming(ADP) and the Temporal-difference Learning (TD) with 500 number of trials. The results were then compared to the actual optimal solution. This is further reviewed in Figure 2.

```
ADPagent = PassiveADPAgent(policy, sequential_decision_environment)
for i in range(500):
    run_single_trial(ADPagent, sequential_decision_environment)
```

```
Warning: Transition table is empty.

The calculated utilities are:
```

```
print('\n'.join([str(k)+':'+str(v) for k, v in ADPagent.U.items()]))
```

```
(0, 0):0.29032269024462465
(0, 1):0.39635739371740647
(1, 2):0.6476921039860977
(3, 2):1.0
(3, 0):0.0
(3, 1):-1.0
(2, 1):0.46469880508113076
(2, 0):0.0
(2, 2):0.7994910488303061
(1, 0):0.1971931536157648
(0, 2):0.5085735332703466
```

A) Utility estimation based on ADP

```
for i in range(500):
    run_single_trial(TDagent,sequential_decision_environment)
```

The calculated utilities are:

```
print('\n'.join([str(k)+':'+str(v) for k, v in TDagent.U.items()]))
```

```
(0, 1):0.375982858719304
(1, 2):0.6732713932673657
(3, 2):1
(0, 0):0.26488484487133074
(3, 0):0.0
(3, 1):-1
(2, 1):0.3733054672127122
(2, 0):0.0
(2, 2):0.845128082895231
(1, 0):0.19596763653053392
(0, 2):0.5128839716848629
```

B) Utility estimation based on TD

```
from mdp import value_iteration
```

The values calculated by value iteration:

```
U_values = value_iteration(sequential_decision_environment)
print('\n'.join([str(k)+':'+str(v) for k, v in U_values.items()]))
```

```
(0, 1):0.3984432178350045
(1, 2):0.649585681261095
(3, 2):1.0
(0, 0):0.2962883154554812
(3, 0):0.12987274656746342
(3, 1):-1.0
(2, 1):0.48644001739269643
(2, 0):0.3447542300124158
(2, 2):0.7953620878466678
(1, 0):0.25386699846479516
(0, 2):0.5093943765842497
```

C) Actual utility calculated

Figure2. A side-by-side comparison of the calculation of utilities

As seen in the image above the measured utility for spot (2,2) can be measure with the following results:

ADP: 0.7994

TD: 0.8451

Actual: 0.7953

Therefore, as expected our ADP approach is the most accurate even though it is heavy in a computational sense, provides a result that is much closer than the actual. Although, the source code from the GitHub Page also provides a way to graphically review how the utility estimates vary according to the number of trials. Therefore, let us review why is it that our TD approach seems to be way off compared to an ADP Approach. This is shown in figure 3.

```
agent = PassiveTDAgent(policy, sequential_decision_environment, alpha=lambda n: 60./(59+n))
graph_utility_estimates(agent, sequential_decision_environment, 500, [(2,2)])
```

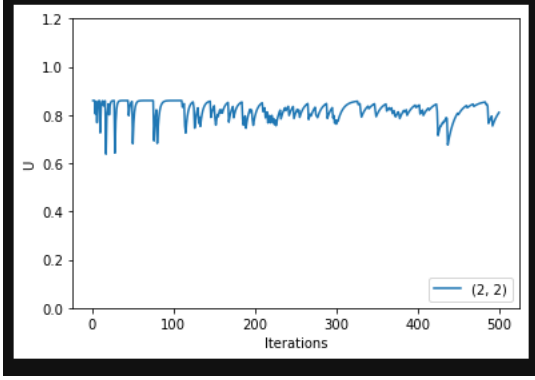


Figure 3. A graphical representation of the learning curve for a TD approach.

The image above therefore shows that the learning curve approaches the most accurate representation at 200th to 250th iteration. Any further iteration it seems the learning curve goes haywire. Why is this the case? This could be resulted in the aforementioned explanation in our methods section, "if the state is close to high utility then the utility of the state must also be high." Therefore, the more iterations happen the more the TD approach will allow for it to explore other utilities. In other words, the calculated approach will reach a peak of accuracy, then in its attempt to keep reaching further accuracy to the actual utility, it will go exploring leading to the great discrepancy from the actual utility. In essence thing of this a dissipating wave function. The amplitudes will start high, lower in a positive and negative value until reaching an equilibrium (actual utility) only to get disturbed again in order to get more knowledge in the environment, repeating that dissipating wave effect. So, what does this tell us? That the initial hypothesis of having reinforced learning running an infinite number of times in order to approach an optimal goal might not be necessary and in fact, a lesser number of iterations could lead to an acceptable utility measurement without compromising. This is furthered review in figure 4. In which the iterations are stopped at the 210th time. We find the utility for (2,2) to be valued at 0.7888

```
TDagent = PassiveTDAgent(policy, sequential_decision_environment, alpha = lambda n: 60./(59+n))
```

Now we run **200 trials** for the agent to estimate Utilities.

```
for i in range(210):
    run_single_trial(TDagent,sequential_decision_environment)
```

The calculated utilities are:

```
print('\n'.join([str(k)+':'+str(v) for k, v in TDagent.U.items()]))
```

```
(0, 1):0.4035178195108555
(1, 2):0.6533601619761997
(3, 2):1
(0, 0):0.29865083148944993
(3, 0):0.0
(3, 1):-1
(2, 1):0.649571796350913
(2, 0):0.0
(2, 2):0.7888187198360528
(1, 0):0.21299563525546175
(0, 2):0.508618615093094
```

Figure 4. An updated TD approach

Thus far, this analysis has been looking at the utilities of solving this 4X3 with the assumption of an obstacle. However, let us assume that there is no obstacle with an increase size of environment. It is then only logical to assume given the free space, the reinforce learning would need a lesser number of trials in order to find a utility that matches the actual value. The following test was conducted by lowering the trails to stop at the 50th run. Figure 5 shows the updated environment and the results of this hypothetical scenario.

```
# Action Directions
north = (0, 1)
south = (0,-1)
west = (-1, 0)
east = (1, 0)

policy = {

    (0, 3): east,  (1, 3): east,   (2, 3): east,   (3, 3): south,
    (0, 2): east,  (1, 2): east,   (2, 2): east,   (3, 2): None,
    (0, 1): north, (1, 1): north,  (2, 1): north,  (3, 1): None,
    (0, 0): north, (1, 0): north,  (2, 0): north,  (3, 0): west,

}
```

→	→	→	↓
→	→	→	+1
↑	↑	↑	-1
↑	↑	↑	←

A) A graphical and code interpretation of the environment without obstacles

```
ADPagent = PassiveADPAgent(policy, sequential_decision_environment)
for i in range(50):
    run_single_trial(ADPagent, sequential_decision_environment)
```

```
Warning: Transition table is empty.

The calculated utilities are:
```

```
print('\n'.join([str(k)+':'+str(v) for k, v in ADPagent.U.items()]))
```

```
(0, 0):0.28050905022485945
(0, 1):0.399894420942276
(1, 2):0.632390159749505
(3, 2):1.0
(3, 0):0.0
(3, 1):-1.0
(2, 1):0.5022513854679802
(2, 0):0.4120262469211822
(2, 2):0.7805572660098522
(1, 0):0.0009719338764606414
(0, 2):0.507941161897527
```

B) Calculated utilities utilizing ADP

```
TDagent = PassiveTDAgent(policy, sequential_decision_environment, alpha = lambda n: 60./(59+n))
```

Now we run **50 trials** for the agent to estimate Utilities.

```
for i in range(50):
    run_single_trial(TDagent,sequential_decision_environment)
```

The calculated utilities are:

```
print('\n'.join([str(k)+':'+str(v) for k, v in TDagent.U.items()]))
```

```
(0, 1):0.4258874487856481
(1, 2):0.6801970282474671
(3, 2):1
(0, 0):0.30539832670163
(3, 0):0.0
(3, 1):-1
(2, 1):0.5777026789962239
(2, 0):0.3214978214664107
(2, 2):0.8410206010585234
(1, 0):-0.51557915119087
(0, 2):0.5465224183116314
```

C) Calculated Utilities utilizing TD

```
from mdp import value_iteration
```

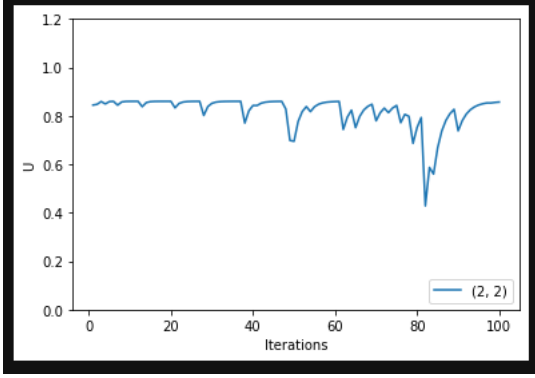
The values calculated by value iteration:

```
U_values = value_iteration(sequential_decision_environment)
print('\n'.join([str(k)+':'+str(v) for k, v in U_values.items()]))
```

```
(0, 1):0.3984432178350045
(1, 2):0.649585681261095
(3, 2):1.0
(0, 0):0.2962883154554812
(3, 0):0.12987274656746342
(3, 1):-1.0
(2, 1):0.48644001739269643
(2, 0):0.3447542300124158
(2, 2):0.7953620878466678
(1, 0):0.25386699846479516
(0, 2):0.5093943765842497
```

D) Actual value of the calculated utilities.

```
agent = PassiveTDAgent(policy, sequential_decision_environment, alpha=lambda n: 60./(59+n))
graph_utility_estimates(agent, sequential_decision_environment, 100, [(2,2)])
```



E) Graphical representation of the learning curve for a TD approach

Figure 5. Updated calculation without a barrier and increase environment.

As seen in the image above the measured utility for spot (2,2) can be measure with the following results:

ADP: 0.7805

TD: 0.8410

Actual: 0.7953

The results, therefore, tell us that an ADP approach actually benefits from an environment that offers a wide space and a plethora of options. Given that is guided by a linear algebraic and probabilistic approach. The lesser number of trials does not interfere with approximation. In comparison, the TD approach seems to still lack and requires an almost number of trials in order to be efficient and accurate.

IV. Conclusion

The analysis and experiment regarding reinforcing learning showed the final conclusions:

Adaptive dynamic programming requires a lesser number of trials in order to become efficient. It, however, requires a higher computational demand in order to successfully run. Furthermore, it is far more flexible and reliant in an open environment with numerous options.

Temporal-difference Learning on the other hand exceeds in an environment with several constraints. This could be explained by the way it compares the measure of utilities with its surroundings and benefits from past and current information. Therefore, in a complex environment whose possible solution is within single digits per se, temporal-difference learning could potentially find the most optimal solution without requiring the heavy amount of computational data required with ADP.

Reinforced learning, in general, offers a balance for exploration vs exploitation, and based on the problem a successful implementation creates the balance necessary to solve a problem.

References

[1] aimacode, "aimacode/aima-python", GitHub, 2020. [Online]. Available: https://github.com/aimacode/aima-python/blob/master/reinforcement_learning.ipynb. [Accessed: 14- Dec- 2020].

[2] R. Stuart, N. Peter, Artificial Intelligence, A Modern Approach Third Edition. New Jersey: Peason Education Inc., p. 832 – 839.