Kyle Arechiga

INFO 550

April 27, 2022

Dyna-Q: Reinforcement Learning With Planning

Introduction

Reinforcement learning (RL) in artificial intelligence (AI) involves an agent learning from its experiences and developing its own policy of actions to perform in the environment to achieve the greatest reward possible. In a Markov Decision Process (MDP) the agent can choose from a set of actions that will result in a transition to the next state in the environment. However, the agent may have no prior knowledge of the rewards for each state transition, so it must learn how the environment works to receive the most rewards possible. **Q-Learning** is a model-free RL algorithm where for each state-action pair, an agent will update its previous estimated value for the state-action (this is called the Q-Value) based on newly perceived data from that state and action. The agent learns the value of each state-action through experience. The algorithm in pseudocode is shown below[1].

```
Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

Algorithm parameters: step size $\alpha \in (0,1]$, small $\varepsilon > 0$
Initialize $Q(s,a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

Loop for each episode:
    Initialize $S$
    Loop for each step of episode:
        Choose $A$ from $S$ using policy derived from $Q$ (e.g., $\varepsilon$-greedy)
        Take action $A$, observe $R$, $S'$
        $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$
        $S \leftarrow S'$
    until $S$ is terminal
```

Q-Learning includes a learning rate, α, and an exploration rate, ε, which can be tweaked between 0 and 1 to determine how much an agent will learn from a new perceived value and how much the agent may decide to deviate off the current policy (to perhaps find an even better policy). The update equation also includes the variable, γ, which represents the lookahead discount. If α, ε, and γ are set to reasonable values, Q-Learning will converge to the optimal policy eventually, in theory. The drawback with Q-Learning is it may take a large amount of episodes and time for the agent to learn the optimal policy depending on the complexity of the problem.

Dyna-Q can be an improvement on Q-Learning which aims to solve this problem by adding *planning* steps in addition to learning from experience. Dyna-Q+ can be even more of an

[1] Richard S. Sutton and Andrew G. Barto, *Reinforcement Learning: An Introduction*, Second edition, Adaptive Computation and Machine Learning Series (Cambridge, Massachusetts: The MIT Press, 2018).

improvement than Dyna-Q because it rewards more exploration to find the best policy. For this project, a Gridworld[2] environment was used to demonstrate the positives and negatives between these algorithms.

Dyna-Q and Dyna-Q+

Dyna-Q is similar to Q-Learning except it introduces a planning step in addition to the learning step. For each state-action pair that the agent visits, it will update the Q-Value based on the new state and reward that it perceives. This is the same update step that is in Q-Learning, as shown below:

(1) $Q(S,A) \leftarrow Q(S,A) + \alpha[R + \gamma max_a Q(S',a) - Q(S,A)]$

Additionally, the Dyna-Q agent will have a *model* that updates every time the agent performs an action. The model stores what the new reward (R) and new state (S') will be when the agent takes an action (A) from the current state (S). After the Q-Learning step, the model will be updated.

(2) $Model(S,A) \leftarrow \{R, S'\}$

After these steps, the agent will perform n iterations on Equations (3)-(6). For each planning iteration, the Dyna-Q agent will consider a *random* state-action that the agent has already experienced at least once. It will then use its *model* to **simulate** the perceived reward and new state it would have if the random state and random action were performed. Using the simulated reward and new state, we can use the Q-Value Update Equation (1) again to update the value for the random state and action for this specific planning iteration.

(3) $S_r \leftarrow Random(States_{visited})$
(4) $A_r \leftarrow Random(Actions_{S_r})$
(5) $\{R, S'\} \leftarrow Model(S_r, A_r)$
(6) $Q(S_r,A_r) \leftarrow Q(S_r,A_r) + \alpha[R + \gamma max_a Q(S',a) - Q(S_r,A_r)]$

Dyna-Q+ has all the steps that Dyna-Q has, except it keeps track of when the last time a state-action was visited. The purpose of Dyna-Q+ is to add an extra simulated reward to the agent for trying out state-actions that have not been attempted for a long time. This way, the agent may be able to find rewards that Q-Learning and regular Dyna-Q may never consider.

For Dyna-Q+ we can add an environment variable called $time$ which represents the number of time steps that have passed in a single episode. For each action that the agent takes, $time$ will

[2] Gridworld Environment source code for this project was built off of "Berkeley AI Materials," http://ai.berkeley.edu/home.html.

increment by 1. Additionally, the agent will have a counter[3] called t that will store the most recent $time$ that a state-action was visited by the agent (Equation (7)).

(7) $t(S, A) \leftarrow time$

The *model* update for Dyna-Q+ will also be slightly different than in Dyna-Q. For Dyna-Q+ the *model* for (S, A) will still update to $\{R, S'\}$ like in Dyna-Q Equation (2), but we will also intialize the model for each possible action a in the state S, if S had not previously been visited by the agent. The actions will initialize to have a reward of 0 and the next state will just be the current state, as shown in Equation (8).

(8) $Model(S, a) \leftarrow \{0, S\}$

During the planning iterations of Dyna-Q+, instead of using just the R and S' output by the model in Equation (5), we can add an additional simulated reward defined by the amount of $time$ that has passed since $t(S, A)$ and a user-defined small valued constant, κ, shown below in Equation (9).

(9) $R' \leftarrow R + \kappa\sqrt{time - t(S_r, A_r)}$

R' will then be plugged into Equation (6) instead of R during each planning iteration to update the Q-value for state and action chosen in Equations (3) and (4).

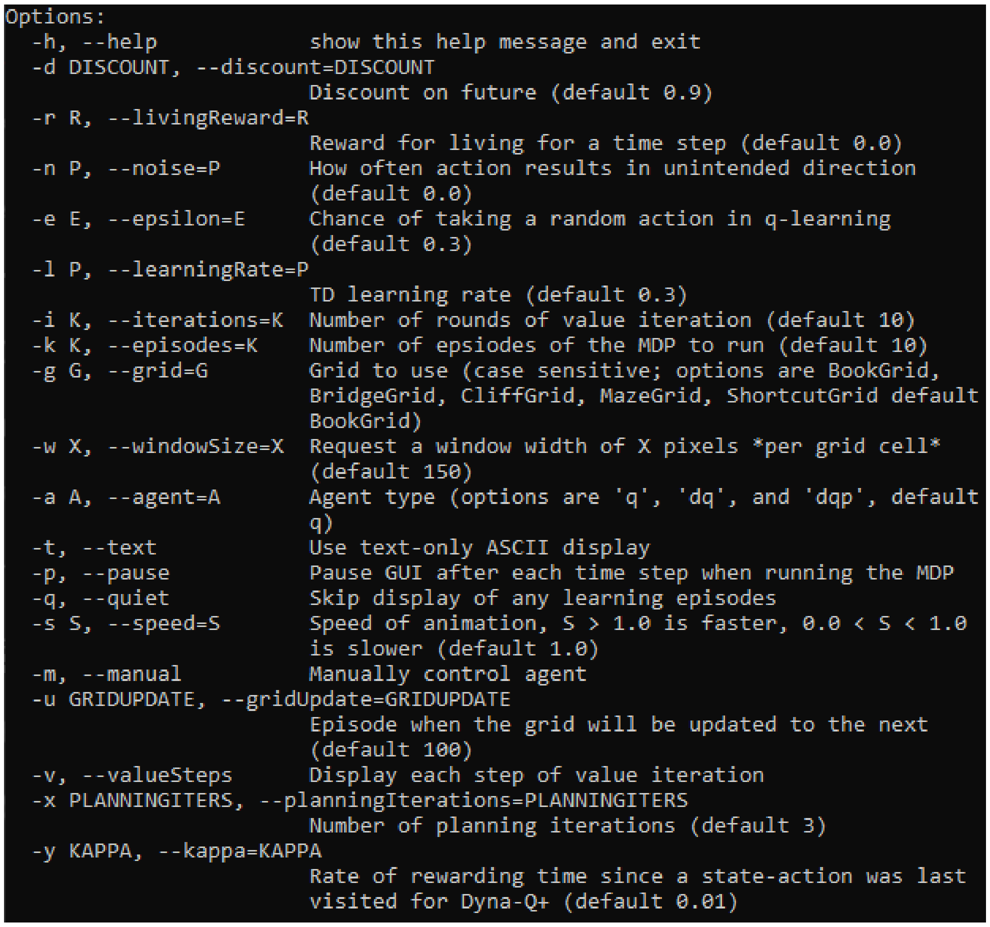
These algorithms were implemented using Python with the help of Sutton and Barto's *Reinforcement Learning: An Introduction* Chapters 8.1-8.3 which provide the useful pseudocode and descriptions for Dyna-Q and Dyna-Q+. Additionally, I referred to this Medium blog[4] for some help on implementing some of the Dyna-Q+ updates. For the Gridworld code, most of that was derived from "Berkeley AI Materials", with some adjustments and additions made to some of the classes to satisfy the Dyna-Q and Dyna-Q+ algorithms.

Usage

This project was ran using Windows 10 and should work on later versions of MAC OS as well. I used Python 3.9 to compile the project and run it.

`python gridworld.py -h` shows the following available options:

[3] "Util.Py," accessed April 30, 2022, https://www.cs.utexas.edu/~pstone/Courses/343spring12/assignments/classification/docs/util.html.
[4] Jeremy Zhang, "Reinforcement Learning — Model Based Planning Methods Extension," Medium, July 18, 2019, https://towardsdatascience.com/reinforcement-learning-model-based-planning-methods-extension-572dfee4cceb.



```
Options:
  -h, --help            show this help message and exit
  -d DISCOUNT, --discount=DISCOUNT
                        Discount on future (default 0.9)
  -r R, --livingReward=R
                        Reward for living for a time step (default 0.0)
  -n P, --noise=P       How often action results in unintended direction
                        (default 0.0)
  -e E, --epsilon=E     Chance of taking a random action in q-learning
                        (default 0.3)
  -l P, --learningRate=P
                        TD learning rate (default 0.3)
  -i K, --iterations=K  Number of rounds of value iteration (default 10)
  -k K, --episodes=K    Number of epsiodes of the MDP to run (default 10)
  -g G, --grid=G        Grid to use (case sensitive; options are BookGrid,
                        BridgeGrid, CliffGrid, MazeGrid, ShortcutGrid default
                        BookGrid)
  -w X, --windowSize=X  Request a window width of X pixels *per grid cell*
                        (default 150)
  -a A, --agent=A       Agent type (options are 'q', 'dq', and 'dqp', default
                        q)
  -t, --text            Use text-only ASCII display
  -p, --pause           Pause GUI after each time step when running the MDP
  -q, --quiet           Skip display of any learning episodes
  -s S, --speed=S       Speed of animation, S > 1.0 is faster, 0.0 < S < 1.0
                        is slower (default 1.0)
  -m, --manual          Manually control agent
  -u GRIDUPDATE, --gridUpdate=GRIDUPDATE
                        Episode when the grid will be updated to the next
                        (default 100)
  -v, --valueSteps      Display each step of value iteration
  -x PLANNINGITERS, --planningIterations=PLANNINGITERS
                        Number of planning iterations (default 3)
  -y KAPPA, --kappa=KAPPA
                        Rate of rewarding time since a state-action was last
                        visited for Dyna-Q+ (default 0.01)
```

Running **`python gridworld.py`** will run with the default Q-Learning options and result in something that looks like the left output in Figure 1. **`python gridworld.py -a dq`** will run Dyna-Q with the default options and should result in something like on the right. The "Q-Values" are shown in each quadrant of each state. The policy (preferred actions) for the agent is shown as the quadrant with the highest Q-Value for each state. Dyna-Q is quickly able to converge to the optimal policy, whereas Q-Learning cannot get the optimal policy in only 10 episodes.

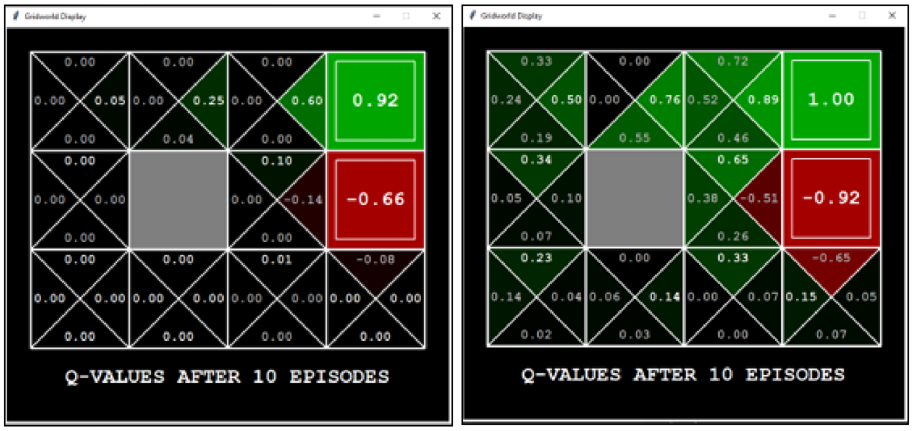


Figure 1 Q-Learning vs Dyna-Q with the default options

Dyna-Q can vary the amount of planning iterations the agent does with each action. The example above uses the default value of 3 but we can change this by adding **`-x`** and the specified number of iterations in the command. For example, changing the command to **`python gridworld.py -a dq -k 3 -g MazeGrid -x 5`** will have the agent do 3 episodes on the MazeGrid layout and do 5 planning iterations for each time step. Compare this with **`python gridworld.py -a dq -k 3 -g MazeGrid -x 25`**. You should notice that once the agent completes the first episode, the more planning iterations we have, the faster the optimal policy will converge. When we set it to 25 planning iterations per timestep, it will converge very quickly at the beginning of the second episode (however this will be more computationally expensive).

Running **`python gridworld.py -a dqp -k 2000 -y .07 -g BridgeGrid -q`** will show the end result of run Dyna-Q+ through 2000 episodes on the "BridgeGrid" environment. We can contrast this with running Dyna-Q through 2000 episodes with the command **`python gridworld.py -a dq -k 2000 -g BridgeGrid -q`**. Remove the **`-q`** at the end to see how the two agents differ in their processes. You will notice that Dyna-Q+ wants to explore incrementally to find out what is on the other side of the "bridge". Dyna-Q and Q are content with sticking with the safe reward (they may get lucky and get to the other side on some runs). The *most likely* end results for the two commands are shown in Figure 2. The higher you set **`-y`** (kappa) to be, the more the Dyna-Q+ agent will want to explore. If set too high though, the agent will disregard the *real* rewards in the environment and will just want to visit states it has not visited in a while. That will not converge to the optimal policy in that scenario.

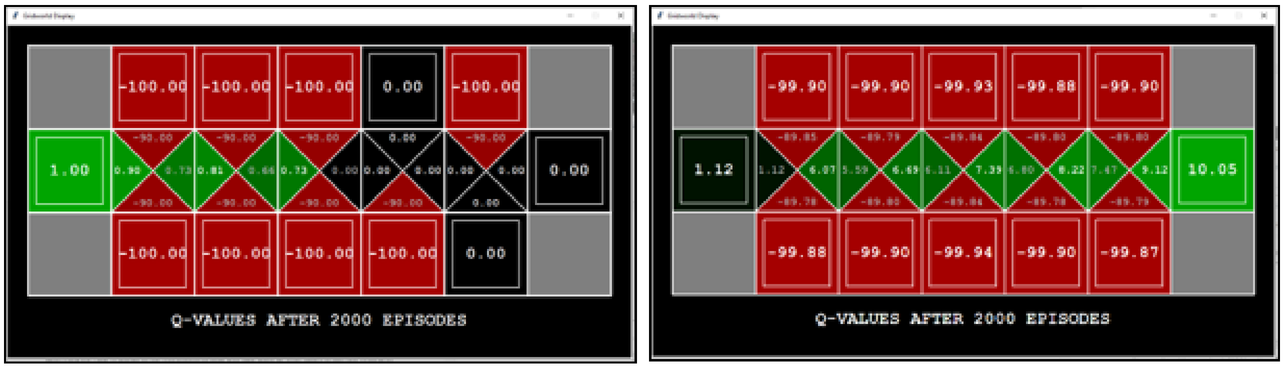


***Figure 2 Dyna-Q vs Dyna-Q+ on the bridge grid**. More often than not, Dyna-Q will not find the reward at the other side whereas Dyna-Q+ will and will converge to the optimal policy.*

Finally, we can test the agents on a "Shortcut Maze" where the environment will change after a specified number of episodes. Each agent will find an optimal policy to go around a large wall to get to the reward. Eventually though, an opening will appear in the wall, leading to a shortcut to the reward. If we run **`python gridworld.py -a dq -k 30 -u 11 -x 5 -g ShortcutGrid`**, the Dyna-Q agent will run 10 episodes with the entire wall present then at Episode 11, the shortcut will open up. The Dyna-Q will have no idea, unless it gets very lucky, that a shortcut opened up. In fact, you can run 1000 more episodes like the following command does: **`python gridworld.py -a dq -k 1030 -u 11 -x 5 -g ShortcutGrid -q`**.

Dyna-Q will still be pretty unlikely to find the shortcut. On the other hand if we run **`python gridworld.py -a dqp -k 30 -u 11 -x 5 -g ShortcutGrid -y 0.01`**, the Dyna-Q+ agent will be more likely to find the shortcut due to it checking states that it hasn't visited in a while. If you run this command a few times, it will find the shortcut much of the time although sometimes it won't in only 30 episodes. If we up it to 130 episodes, it will find the shortcut and the optimal policy a high percent of the time: **`python gridworld.py -a dqp -k 130 -u 11 -x 5 -g ShortcutGrid -y 0.01 -q.`**

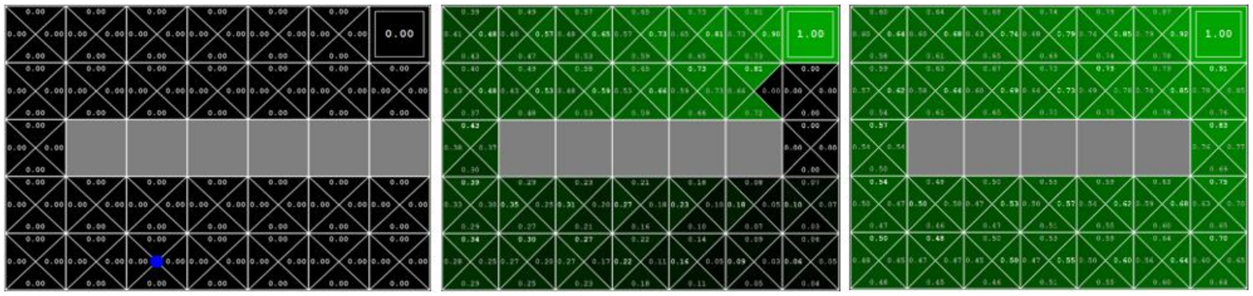


***Figure 3 Shortcut Grid**: Start state shown to the left, the final policies when the agent doesn't find the shortcut vs when the agent does find the shortcut are shown in the middle and to the right.*

Evaluation

To further highlight the differences between these Q-Learning algorithms, I created "dynaQTests.py" which generates the plots shown below in Figure 4. I used **pyplot** from the **matplotlib** Python package as well as **numpy** to generate these plots.

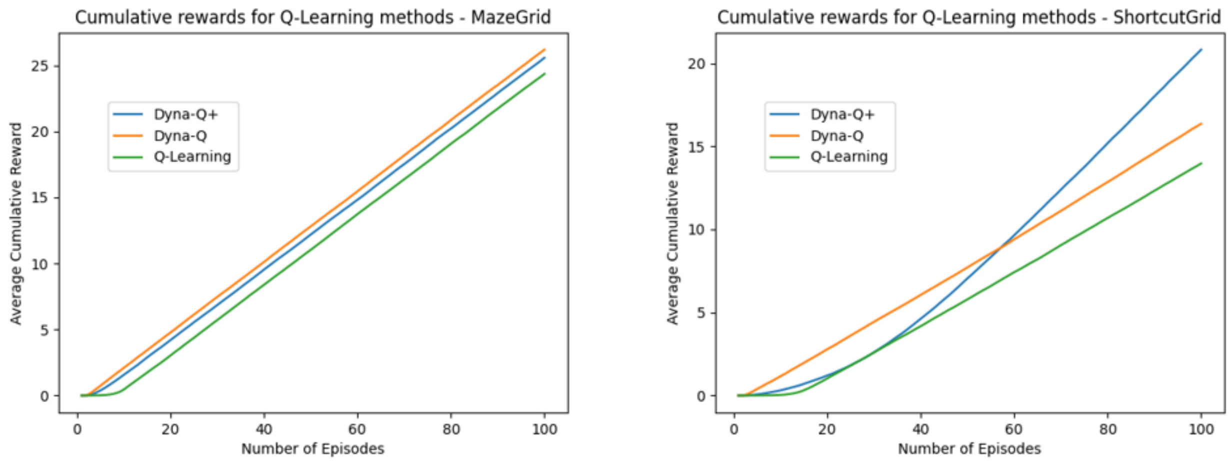


Figure 4 Cumulative Rewards for Different Q Algorithms.

In "dynaQTests.py", it simulates 100 runs of these three algorithms on both the "MazeGrid" and the "ShortcutGrid" with 100 episodes per run, and it averages the cumulative rewards between the runs. All three algorithms had the same exact parameters (α, ε, γ, etc.) and they were set to the same random seed value for each agent.

These plots both show us that regular Q-Learning takes a longer time to converge than Dyna-Q and Dyna-Q+. On the MazeGrid, Dyna-Q+ takes *slightly* longer to converge than Dyna-Q due to its stronger desire to explore other possibilities. However, in the ShortcutGrid, Dyna-Q+ also takes longer to converge in the initial policy, but it catches up quick to the Dyna-Q algorithm once the Shortcut was introduced in the 15th episode. This further illustrates how Dyna-Q+ is more adaptive to change than Dyna-Q and Q-Learning.

While Dyna-Q and Dyna-Q+ are better performing than Q-Learning in deterministic environments, it is important to note that regular Q-Learning can often handle uncertainty better than the formers. If there is uncertainty (noise in the gridworld scenario) in the agent's actions, Dyna-Q's planning model will not be as effective. When we introduce 30% noise in the gridworld environment, the effects are shown in **Figure 5**.

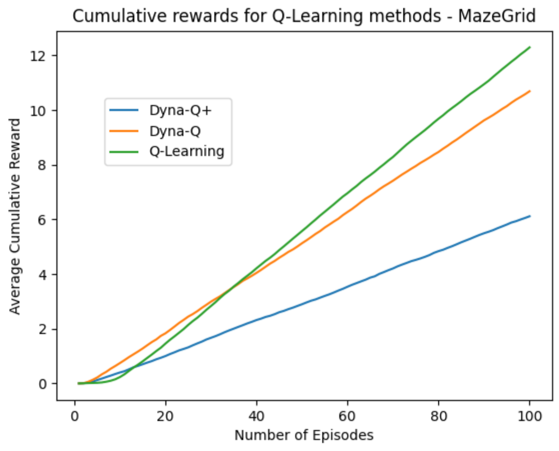


Figure 5 Cumulative Rewards with 30% Noise introduced

In this scenario we see that Q-Learning is a more effective approach than using either Dyna-Q algorithm. Q-Learning is also less computationally expensive, which in this scenario may not be as important, but in certain applications it could be crucial. It is important to recognize these since, in many cases, there will be uncertainty and more complexity in an environment.

References

"Berkeley AI Materials." Accessed April 27, 2022. http://ai.berkeley.edu/home.html.

Sutton, Richard S., and Andrew G. Barto. *Reinforcement Learning: An Introduction*. Second edition. Adaptive Computation and Machine Learning Series. Cambridge, Massachusetts: The MIT Press, 2018.

"Util.Py." Accessed April 30, 2022. https://www.cs.utexas.edu/~pstone/Courses/343spring12/assignments/classification/docs/util.html.

Zhang, Jeremy. "Reinforcement Learning — Model Based Planning Methods Extension." Medium, July 18, 2019. https://towardsdatascience.com/reinforcement-learning-model-based-planning-methods-extension-572dfee4cceb.