Tabular Reinforcement Learning

Dimitrios Kourntidis s3841863

Introduction

This report investigates the effectiveness of tabular reinforcement learning methods in solving a problem small enough to store it's values in tabular form. It covers different algorithms like Q-value iteration, Q-learning, SARSA, N-Step and Monte Carlo.

1. Dynamic Programming

1.1. Methodology

In order to implement the Q-value algorithm some methods were required to be completed first. Specifically the implementation of the greedy policy (Equation 1) and the Q-iteration update (Equation 2).

$$\pi(s) = arg\,max_a\, Q(s,a) \tag{1}$$

$$Q(s,a) \leftarrow \sum_{s'}[p(s'|s,a)\cdot(r(s,a,s')+\gamma\cdot max_a\, Q(s',a'))] \tag{2}$$

After the implementation of those two methods the Algorithm 1 was implemented.

Algorithm 1 Q-value iteration (Dynamic Programming)

Input: Threshold $\eta \in \mathbb{R}^+$
Result: The optimal value function $Q^*(s,a)$
Initialization: $\hat{Q}(s,a) = 0\ \forall s \in S, a \in A$
repeat
  $\Delta \leftarrow 0$.
  for *each* $s \in$ S **do**
    for *each* $\alpha \in$A **do**
      $x \leftarrow Q(s,a)$
      $\hat{Q}(s,a) \leftarrow \sum_{s'}[p(s'|s,a)\cdot(r(s,a,s')+\gamma\cdot max_a\, Q(s',a'))]$
      $\Delta \leftarrow \max(\Delta, |x-\hat{Q}(s,a)|)$
    end for
  end for
until $\Delta < \eta$
$Q^*(s,a) = \hat{Q}(s,a)$
$\pi^*(s) = arg\,max Q^*(s,a)\ \forall s \in S$
Return $Q^*(s,a)$

In order to compute the optimal policy, we iterate through each state and action, and choose the action for the next step with a greedy policy. After each step, we update the Q-value for the corresponding state-action pair, and then compute the maximum error Δ. This procedure repeats until $\Delta < \eta$, where η is threshold value for convergence.

1.2. Results

With algorithm's end we have an action which the agent will choose in each state. With This is visible if we compare Figure 1 and Figure 2 where for each state we have only one desired action which will give the biggest cumulative reward for the beginning state. For this arrangement (starting from (0,3) and terminating at (7,3)) the algorithm needs 17 iterations to convergence.

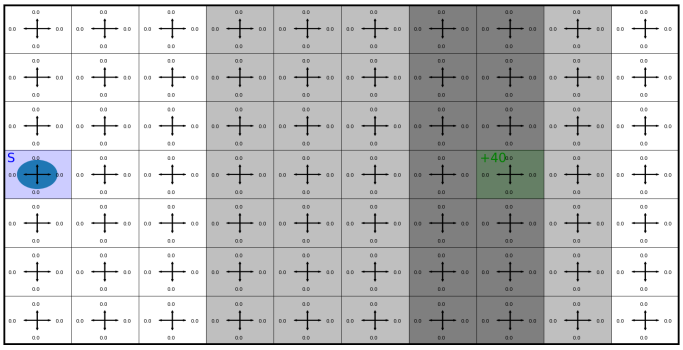

Figure 1. Q-values at the beginning of the algorithm, each Q(s,a) is initialized with 0

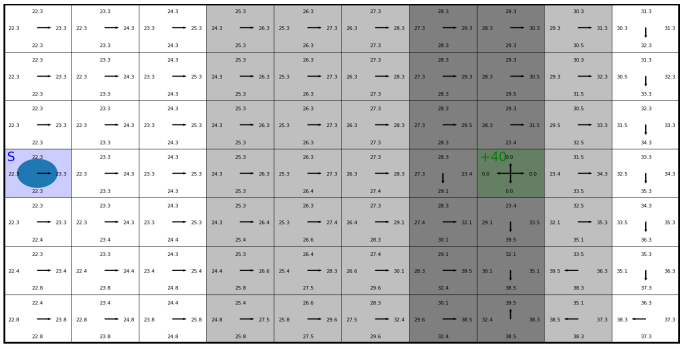

Figure 2. Q-values after the convergence of the Algorithm 1, each state has its optimal action to choose

We can also see from Figure 2 that $V^*(s = 3) = 23.3$. That means that the expected reward for our agent when he reaches the terminal state is 23.3

The average reward per timestep is the expected reward that the agent will receive on average in each timestep of the environment. For this problem the average reward per timestep is approximately 1.33. This value can be calculated as following:

$$reward\,per\,timestep = \frac{\sum_{i=1}^{n} \sum_{j=i}^{steps} r_j/steps}{n} \quad (3)$$

Where:

n = number of repetitions to get an average value
$steps$ = steps needed to reach the terminal state
r_j = reward for step j

The value of n was set to 100.

After changing the terminal location to (6,2) we need to compute again the optimal policy. For this terminal state this algorithm needs 67 iterations to convergence, almost four times more than it needed previously. Another noticable difference is the $V^*(s = 3)$ which now dropped from 23.3 to 19.7.

2. Exploration

2.1. Methodology

In Section 1, Dynamic Programming was explored. It is useful when the user has access to the environment. For application where there is no access to the model, some exploration is needed to validate that the steps taken so far are the optimal.

In this case for the action we have two policies.
The ϵ-greedy policy, where a random action is taken with small probability ϵ, otherwise the greedy action is selected. With the parameter ϵ the amount of exploration can be tuned, where $\epsilon \to 1$ gives random policy and $\epsilon \to 0$ gives greedy policy.
The second is the Boltzmann policy which can be represented with the Equation 4.

$$\pi(a|s) = \frac{e^{\hat{Q}(s,a)/\tau}}{\sum_{b \in A} e^{\hat{Q}(s,b)/\tau}} \quad (4)$$

With this equation the action with a higher current value estimate is given a higher probability. The amount of exploration using this policy can be tuned with the parameter τ. When $\tau \to 0$ the policy becomes greedy and when $\tau \to \infty$ the policy becomes uniform. That's because if $\tau \to \infty$ then the equation 4 becomes $\frac{1}{\sum_{b \in A} 1}$.

After selecting the action an update have to be performed. To do this we first compute a back-up estimate (Equation 5).

$$G_t = r_t + \gamma \cdot max_{a'}\, \hat{Q}(s_{t+1}, a') \quad (5)$$

Next we update the $\hat{Q}(s_t, a_t)$ values using the Equation 6

$$\hat{Q}(s_t, a_t) \leftarrow \hat{Q}(s_t, a_t) + \alpha \cdot [G_t - \hat{Q}(s_t, a_t)] \quad (6)$$

Where α denotes the learning rate.

Those two equations (Equation 5 and Equation 6) are used to construct the Q-learning algorithm (Algorithm 2).

Algorithm 2 Tabular Q-learning

Input: ϵ, $\alpha \in (0, 1]$, $\gamma \in (0, 1]$, total budget
$\hat{Q}(s, a) \leftarrow 0,\ \forall s \in S, a \in A$
$s \sim p_0(s)$
while $budget$ **do**
  $a \sim \pi(a|s)$
  $r, s' \sim p(r, s'|s, a)$
  $\hat{Q}(s, a) \leftarrow \hat{Q}(s, a) + \alpha \cdot [G_t - \hat{Q}(s, a)]$
  if s' is terminal **then**
    $s \sim p_0(s)$
  else
    $s \leftarrow s'$
  end if
end while
Return $\hat{Q}(s, a)$

After the implementation of the Q-learning algorithm an experiment run to examinate both ϵ-greedy policy and Boltzmann policy using different values for ϵ and τ.

2.2. Results

The experiment run for 50,000 timestep and was averaged over 50 repetitions in order to get smoother figure. The results can be observed in Figure 3

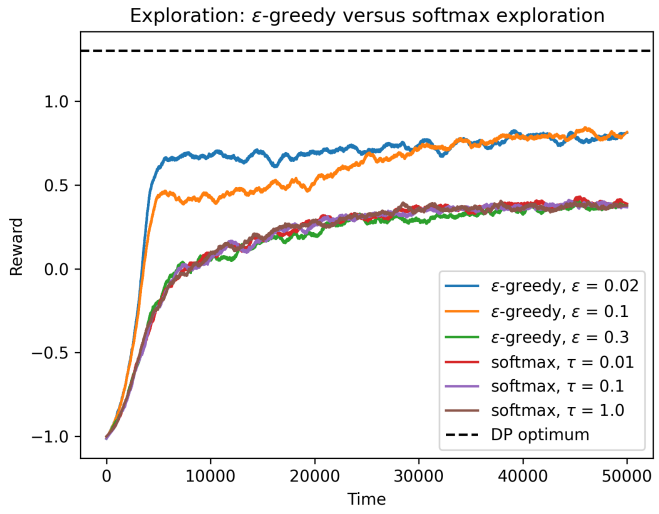

Figure 3. Reward per timestep using multiple values for ϵ and τ

From the plot it can be observed that neither policy (ϵ-greedy and Boltzmann) reaches the performance of the Dynamic Programming and that is because under Dynamic Programming each action is the optimal one, where now there is the possibility of exploration. It can also be seen that ϵ-greedy performs better than the Boltzmann policy, exepction to than is $\epsilon = 0.3$. In my opinion that's because the probability of a random choice is bigger than that of the other two choices. The method that I prefer is the ϵ-greedy with the $\epsilon = 0.02$ because it converges faster than the other methods.

3. Back-up: On-policy versus off-policy target

3.1. Methodology

In this section the way of the back-up will be studied. SARSA will be implemented to see how on-policy behave versus an off-policy (Q-learning).
For SARSA the action selection is the same with the Q-learning. The difference of between on-policy and off-policy is that the first plugs in the value of the best action at the next state and the second uses the value from the action it actually takes.

That leads to the following two equations:

$$G_t = r_t + \gamma \cdot \hat{Q}(s_{t+1}, a_{t+1}) \tag{7}$$

$$\hat{Q}(s_t, a_t) \leftarrow \hat{Q}(s_t, a_t) + a \cdot [G_t - \hat{Q}(s_t, a_t)] \tag{8}$$

With those equations, we can construct the Tabular SARSA algorithm.

Algorithm 3 Tabular SARSA

Input: ϵ, $\alpha \in (0, 1]$, $\gamma \in (0, 1]$, total budget
$\hat{Q}(s, a) \leftarrow 0, \ \forall s \in S, a \in A$
$s \sim p_0(s)$
$a \sim \pi(a|s)$
while *budget* **do**
  $r, s' \sim p(r, s'|s, a)$
  $a' \sim \pi(a'|s')$
  $\hat{Q}(s, a) \leftarrow \hat{Q}(s, a) + \alpha \cdot [G_t - \hat{Q}(s, a)]$
  if s' is terminal **then**
    $s \sim p_0(s)$
    $a \sim \pi(a|s')$
  else
    $s \leftarrow s'$
    $a \leftarrow a'$
  end if
end while
Return $\hat{Q}(s, a)$

After the implementation of the SARSA algorithm an experiment run to compare on-policy and off-policy using multiple values for α.

3.2. Results

After 50,000 timesteps we get the results that can be seen in Figure 4 which were averaged over 50 repetitions.

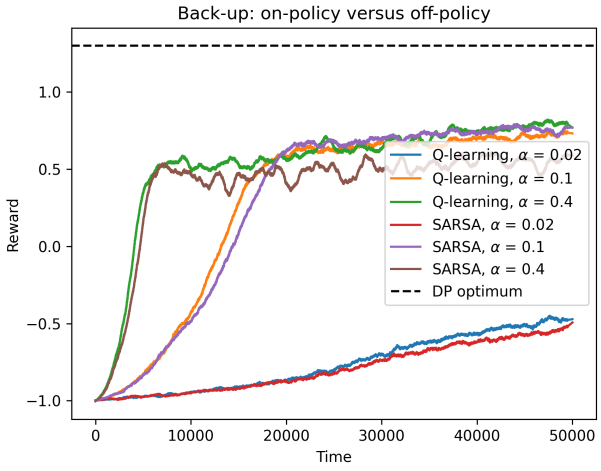

Figure 4. Reward per timestep using multiple values a for both Q-learning and SARSA

Both SARSA and Q-learning with $a = 0.4$ have the steepest plots, and they have the best performance for about 20,000 timesteps. After that the performance of the brown plot remains almost the same on average while green, orange and purple plots continue to improve. Finally, both SARSA and

Q-learning with $a = 0.02$ improve very slow but steadily.

4. Back-up: Depth of target

4.1. Method

In this section the depth of the back-up. So far the depth was 1 and now n-step algorithms will be examined. The n-step Q-learning computes the target following the Equation 9.

$$G_t = \sum_{i=0}^{n-1} (\gamma)^i \cdot r_{i+1} + (\gamma)^n * max_a Q(s_{t+n}, a) \quad (9)$$

Then we have the tabular update we had in Equation 6. This formulates the n-step Q-learning that can be seen in Algorithm 4

Algorithm 4 Tabular n-step Q-learning

Input: $\epsilon \in (0, 1]$, $\alpha \in (0, 1]$, $\gamma \in (0, 1]$, total budget, maximum episode length T , target depth n
$\hat{Q}(s, a) \leftarrow 0, \ \forall s \in S, a \in A$
while *budget* **do**
  $s_0 \sim p_0(s)$
  for $t = 0...(T-1)$ **do**
    $a_t \sim \pi(a|s_t)$
    $r_t, s_{t+1} \sim p(r, s'|s_t, a_t)$
    if s_{t+1} is terminal **then**
      break
    end if
  end for
  $T_{ep} \leftarrow t + 1$
  for $t = 0...(T_{ep} - 1)$ **do**
    $m = min(n, T_{ep} - t)$
    if s_{t+m} is terminal **then**
      $G_t \leftarrow \sum_{i=0}^{m-1} (\gamma)^i \cdot r_{t+i}$
    else
      $G_t \leftarrow \sum_{i=0}^{m-1} (\gamma)^i \cdot r_{t+i} + (\gamma)^m \cdot max_a \hat{Q}(s_{t+m}, a)$
    end if
    $\hat{Q}(s_t, a_t) \leftarrow \hat{Q}(s_t, a_t) + a \cdot [G_t - \hat{Q}(s_t, a_t)]$
  end for
end while
Return $\hat{Q}(s, a)$

We can also sum all the rewards that accumulated in the episode, instead of going until the n-step. That gives us the Monte Carlo approach. In this case the target becomes the following.

$$G_t = \sum_{i=0}^{\infty} (\gamma)^i \cdot r_{t+i} \quad (10)$$

The update remains the same as before, now we have slightly different algorithm

Algorithm 5 Tabular Monte Carlo reinforcement learning.

Input: $\epsilon \in (0, 1]$, $\alpha \in (0, 1]$, $\gamma \in (0, 1]$, total budget, maximum episode length T
$\hat{Q}(s, a) \leftarrow 0, \ \forall s \in S, a \in A$
while *budget* **do**
  $s_0 \sim p_0(s)$
  for $t = 0...(T-1)$ **do**
    $a_t \sim \pi(a|s_t)$
    $r_t, s_{t+1} \sim p(r, s'|s_t, a_t)$
    if s_{t+1} is terminal **then**
      break
    end if
  end for
  for $t = t...0$ **do**
    $G_i \leftarrow r_i + \gamma \cdot G_{i+1}$
    $\hat{Q}(s_i, a_i) \leftarrow \hat{Q}(s_i, a_i) + a \cdot [G_i - \hat{Q}(s_i, a_i)]$
  end for
end while
Return $\hat{Q}(s, a)$

4.2. Results

After implementing both Monte Carlo and n-step Q-learning algorithms, an experiment run for 50,000 timesteps as before. In this experiment different values for n were examined as well as the performance of Monte Carlo.

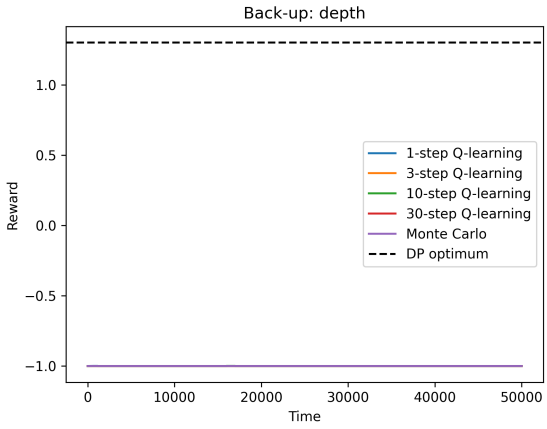

Figure 5. Average reward per timestep using 1,5,10 and 30 steps

Having the results in Figure 5 the average reward stays at -1 which mean the agent never reached the target.

5. Reflection

After many experiments some conclusions can be made. First it is visible that neither algorithm reached the performance of Dynamic Programming. That's because with Dynamic Programming there is the ability to traverse

all possible state-actions and choose the optimal action each time. Although Dynamic Programming creates the best policy, in can be computationally expensive as it was shown in Section 1 when the terminal stated changed.

In Section 2 two exploration methods were examined, ϵ-greedy and softmax. For the exploration method I prefer the softmax. ϵ-greedy is less likely to take a random action in order to explore. If no exploration happen then it is possible that the best policy will not be explored and the agent won't have a better option because he will be stuck in local optimum rather the global

After exploration methods in Section 3 two different back-up policies were studied. On-policy and off-policy. Off-policy algorithms such Q-learning create their policy with greedy selection independently the action the agent made. On-policy on the other hand create their policy based on the action they took (Sagar, 2020). One problem that on-policy has is The amount of steps needed to converge because the target policy is not updated each time with the optimal values (Plaat, 2022).

Studying those algorithms gave a great inside what are some solution when the problem is small enough to fit in a table. The algorithm studied will not work when the state-space is high-dimensional. This is also known as the curse of dimentionality. We can overcome this issue using deep learning, instead of trying to find the correct action, we try to approximate it.

References

Plaat, A. *Deep Reinforcement Learning*. Springer Nature Singapore, 2022. doi: 10.1007/978-981-19-0638-1. URL https://doi.org/10.1007%2F978-981-19-0638-1.

Sagar, R. On-policy vs off-policy reinforcement learning, Oct 2020. URL https://analyticsindiamag.com/reinforcement-learning-policy/.