Tabular Reinforcement Learning
A Table-Turning Approach to Learning Optimal Actions

Ioannis Koutalios (s3365530)[1]

1. Introduction

In this assignment, we will be exploring various tabular reinforcement learning algorithms. In reinforcement learning, we have an agent that interacts with the environment and tries to learn and make decisions in order to maximize the cumulative reward. The way the agent learns is best described by the process of trial and error, where the agent tries different paths in order to find the optimal. The agent can also exploit an already-known path to the goal-state.

In tabular reinforcement learning the agent maintains a table to store the values of different state-action pairs. The agent tries to estimate the value function as best as possible by doing that, it essentially learns the environment and finds the optimal path.

The environment we will use in this assignment is called “Stochastic Windy Gridworld” and is an adaptation of an example used in Andrew (1999). The environment is a 10×7 grid. The agent can move through the grid in 4 different ways (up, down, left, right). The wind affects the movement of our agent, in columns $3, 4, 5, 8$ the agent is pushed one step up, while in columns $6, 7$ it is pushed two steps up. The stochasticity comes from the fact that the wind only affects our agent 80% of the time. The agent receives a reward of -1 in every cell except in the goal-state where it gets a $+40$ reward. The agent starts at the grid position $[0, 3]$ and the goal-state is at $[7, 3]$.

During this assignment, we will explore various tabular reinforcement learning algorithms. Firstly in Section 2 we will test the dynamic programming algorithm, in which we have access to the environment. As will be discussed this is an ideal case and is guaranteed to find the optimal policy. After that, we will experiment with model-free algorithms. In Section 3 we will implement and test various algorithms such as the Q-learning, SARSA, and Monte Carlo. All of them use tabular reinforcement learning but have important differences that should be compared for their efficiency and performance in our environment. This process is repeated in Appendix A where we more carefully tune some important parameters to achieve better results. Finally, in Section 4 we will summarize all the things we derived from the experiments and try to draw some conclusions.

[1]Leiden Observatory, Leiden University, P.O. Box 9513, 2300 RA Leiden, The Netherlands. Correspondence to: Ioannis Koutalios <koutalios@mail.strw.leidenuniv.nl>.

1st Assignment for the course of Reinforcement Learning,

2. Dynamic Programming

In tabular reinforcement learning the method of dynamic programming is used in order to find the optimal policy for an agent given the environment. To achieve that we have to recursively calculate for each state in our environment the expected value we will get for taking each action.

There are two main types of dynamic programming. The first one is **policy iteration** which involves two steps: policy evaluation and policy improvement. The second type is called **value iteration** which recursively updates the expected values for each state-action pair. In our implementation, we will use a form of value iteration and specifically the Q-value iteration which will be discussed extensively.

Dynamic programming techniques are guaranteed to converge in a finite amount of iterations to the optimal policy. There are although some conditions that the environment should satisfy. It should have a finite and discrete space of state-action pairs. The environment should also be Markovian, meaning that the future state of the environment only depends on the current state and not on the previous steps that were taken.

The limitations of dynamic programming are making its implementation rare in real-world problems. We can only use this method when we have perfect knowledge of the environment which is not something that usually happens in most problems. Dynamic programming also suffers from the curse of dimensionality and is computationally expensive, especially for complex problems. One additional problem is that it can not handle continuous state-action spaces, as is the case for all tabular methods.

2.1. Methods

As we already briefly discussed we will implement a Q-value iteration algorithm to solve the environment we were provided with. This method involves multiple iterations over the whole state-action space where we use the discrete Bellman equation to update the values.

From (Plaat, 2022) we have the Bellman equation:

$$V^{\pi}(s) \leftarrow \sum_{a \in A} \pi(a|s)[\sum_{s' \in S} p_a(s,s')[r_a(s,s') + \gamma V^{\pi}(s')]] \tag{1}$$

where V is the value, π is the probability of the action a given the state s, p is the stochastic transition function, r is the reward function, and γ is the discount rate.

For our implementation where instead of using the value we want to calculate the Q-value we derive the following equation:

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

where Q is the Q-value.

As it becomes clear from the above equations, in order to use the Bellman equation, we need to know the transition function and the reward function. This information is not usually known by the agent as they come directly from the environment in which we try to learn. If we, however, have access to the environment (as is the case for this task) we can implement the Q-value iteration algorithm which is guaranteed to converge to the optimal policy after a finite amount of iterations.

The algorithm is described in Algorithm 1 and is an implementation of the basic value iteration pseudocode that is described in Plaat (2022)

Algorithm 1 Q-value iteration

```
Input: threshold $\eta \in R^+$
Initialize $Q(s,a) = 0$.
repeat
  $\Delta = 0$
  for $s$ in $S$ do
    for $a$ in $a$ do
      $x = Q(s,a)$
      update the $Q(s,a)$ using Equation (2)
      $\Delta = \max(\Delta, |x - Q(s,a)|)$
    end for
  end for
until $\Delta < \eta$
Return: $Q(s,a)$
```

The goal state in our environment is terminal. This means that all the values of $Q(s = goal, a)$ should be 0. In our implementation, we treat the terminal state as any other state. The optimal policy will converge to the desired Q-values because the optimal policy in the terminal state is to take no action and receive the reward. One other way of solving the issue of the terminal state is to possibly exclude it from the training and modify the updating rule accordingly.

2.2. Results

After implementing the algorithm that we have already discussed we can see that our Q-values converge to the optimal policy after 18 iterations. In Figure 1 we can see the progression of this convergence at the beginning (1st iteration), midway (10th iteration), and at convergence (18th iteration).

After the first iteration, we can see that the Q-values are generally low with some small exceptions. The only values that are high are near the terminal state and on the path, that will eventually be followed by the agent in the optimal policy. At the midway point, the Q-values are becoming higher and are more stable, especially the ones on the path of the optimal policy. Finally, after the algorithm has converged we have the whole picture and the final optimal policy which the agent will follow.

During the execution of the algorithm, we are interested in calculating various other values to get a better understanding of our code and how it works. The first value we get is the optimal value at the start state which is

$$V^{\star}(s = \text{start}) = \max_a Q(s = \text{start}, a) = 23.3$$

. Two other values we are interested in are the average reward per time step and the average number of steps under optimal policy. To compute these we have the agent repeat the process under the optimal policy many times each time calculating these two values and finally averaging the results. The reason we do this multiple times is to account for the stochasticity of our experiments. The final results we get are 1.33 and 17.6 respectively.

We want to show how all these three values are connected. We know that the reward for each step of the agent is -1 except when it reaches the goal-state where the reward is $+40$. In the optimal policy, our agent will reach the goal-state at the final step, so we can compute the value $V^{\star}(s = \text{start}) = (n-1) \cdot (-1) + 40 = (17.6-1) \cdot (-1) + 40 = 23.4$ (where n is the average number of steps the agent takes in the optimal policy). We see that this is very close to what we get from directly measuring the expected value at the starting point. The average reward can be calculated very similarly as $\text{average}(r) = \frac{(n-1)\cdot(-1)+40}{n} = \frac{V^{\star}(s=\text{start})}{n} = 1.33$ which is the value we get when we directly calculate it during the execution of our code.

One last experiment we make during this task is to change the position of the terminal state. In our original environment, it is located at $[7, 3]$ and we change that to $[6, 2]$. The first thing we can observe is that we now need 68 iterations for the algorithm to converge (only 18 in the original problem). Under the optimal policy, the agent now takes more steps to reach the goal-state, 20.9 on average.

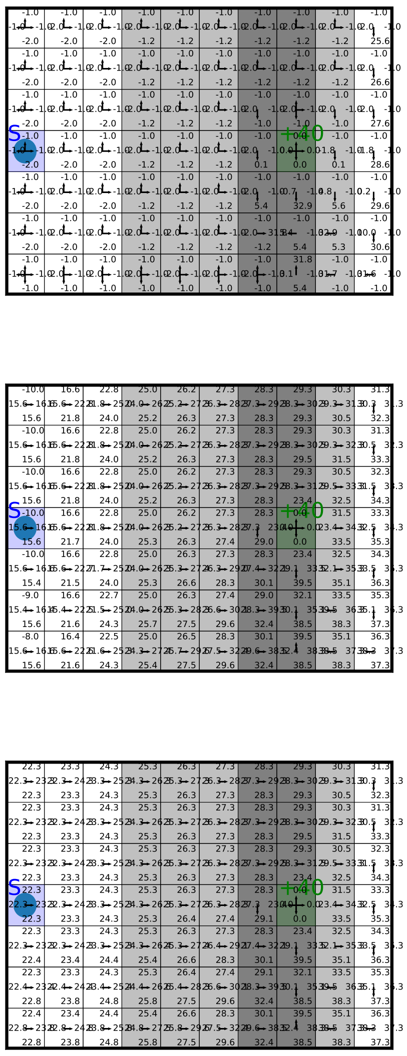

Figure 1. The Q-value table during different points of the execution of the Q-value iteration algorithm. From top to bottom, we see the 1st iteration, 10th iteration, and the 18th and final iteration which shows the optimal policy. We can see how the algorithm learns the optimal policy. The values near the goal state have already converged near the terminal state at the midway point while the values at states far away only converge at later iterations.

3. Model-free

In model-free reinforcement learning, we no longer have access to the model. Instead of using our model, the agent must learn by interacting with the environment in a trial-error style. The agent can learn by only observing the received reward after each transition.

In this assignment, we will discuss three different types of model-free learning. The first one is the **Temporal Difference**, where we use the concept of bootstrapping to learn the Q-values. **Bootstrapping** is the process in which we use the values of the previous state and/or the old values of the current state to update the value of the current state. The second type is the Monte Carlo method where we use the rewards we observed during a whole episode in order to update the Q-values after the end of each episode. The third type only uses n-depth to bootstrap and is therefore somewhere in between the Temporal Difference and Monte Carlo algorithms.

We will also experiment with the concept of **exploration vs exploitation**. During exploitation, the agent is selecting the optimal action based on its current knowledge of the environment. In exploration, the agent doesn't follow this optimal policy but randomly takes an action. Exploration is crucial in order to learn new policies that can lead to the optimal policy.

The next concept we will explore is **on-policy vs off-policy** learning. This will be achieved by comparing the performance of the Q-learning algorithm (off-policy) and the SARSA algorithm (on-policy). In off-policy learning, we bootstrap using the action that has the highest value, while in on-policy learning we bootstrap using the action that was actually taken during the episode. By doing this we include potential exploratory actions, which are not optimal based on the agent's knowledge of the environment.

Finally, we want to explore the concept of **depth**. The n-step Q-learning algorithm uses n future states to bootstrap. In Monte Carlo, we do not have any bootstrap and we use all the rewards observed in each episode. By doing this we essentially have infinite depth.

3.1. Methods

In order to cover all these we will implement four different algorithms. We also use two different policies for action selection. The first is the $\epsilon - \text{greedy}$ policy:

$$\pi(a|s) = \begin{cases} 1 - \epsilon \cdot \frac{|A|-1}{|A|} & \text{if } a = \underset{a' \in A}{\operatorname{argmax}}(Q(s,a')) \\ \frac{\epsilon}{|A|} & \text{otherwise} \end{cases} \tag{3}$$

where ϵ is a number smaller than 1 (usually much smaller values). In $\epsilon - \text{greedy}$ policy we select with probability $1 - \epsilon$ the optimal value, otherwise, we select a random action. The second policy is the Boltzmann softmax policy:

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

where τ is a positive number. In the Boltzmann softmax policy, we use a scaling parameter to scale the amount of exploration. When $\tau \to \infty$ all the actions get the same probability and we have full exploration while for $\tau \to 0$ we have full exploitation as the probability of the optimal action goes to 1.

The first algorithm to be implemented is the tabular Q-learning algorithm. To implement this we need to compute the back-up estimate:

$$G_t = r_t + \gamma \cdot \underset{a'}{max} Q(s_{t+1}, a) \tag{5}$$

where G_t is the target (back-up estimate)

As we can see we use the reward we observed and also take the Q-value of the optimal action in the next step. We then apply the general tabular learning algorithm:

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

where α is the learning rate (positive value smaller than 1).

The general algorithm is shown in Algorithm 2. After we initialize the environment the agent selects an action and takes a step. The action sampling is being done using either the $\epsilon - \text{greedy}$ or the Boltzmann softmax policy. We keep the rewards received by the agent and the next state before updating the Q-values. We repeat the process until we run out of the predetermined number of steps we want to allow our agent to take. If at any point we reach the terminal, we re-initialize the state.

Algorithm 2 Q-learning Tabular

```
Input: $\gamma$, $\alpha$, ($\epsilon$ or $\tau$)
Initialize $Q(s, a) = 0$
Initialize state $s_0$
repeat
  sample action
  take a step
  get reward and next state
  update Q-value using eqs. (5) and (6)
  budget $-= 1$
  if terminal then
    Initialize state $s_0$
  end if
until budget $= 0$
Return: $Q(s, a)$
```

The Q-learning algorithm is an off-policy learning technique. Next, we implement the SARSA algorithm which uses on-policy learning. The main difference is in the calculation of the back-up estimate:

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

As we can see this time we don't use the value of the best possible action but rather the action that we actually take. As discussed before, this might include potential exploratory actions.

The SARSA Tabular algorithm is otherwise almost identical to the Q-learning Tabular algorithm (Algorithm 2). The other difference is that we sample the action after we make the step as we need this value to calculate G_t and we also need to initialize the action every time we initialize the state (at the beginning and when we reach the terminal state).

After that, we want to experiment with the depth of tabular learning algorithms. In order to achieve that we implement two algorithms, the n-step Q-learning and the Monte Carlo.

For the n-step Q-learning algorithm, we want to calculate the back-up estimate using:

$$G_t = \sum_{i=0}^{n-1} \gamma^i \cdot r_{t+i} + \gamma^n \cdot \max_a Q(s_{t+n}, a) \tag{8}$$

where n is the number of steps used for bootstrapping.

As we can see this is a combination of on-policy and off-policy learning as we use the observed rewards inside the summation (on-policy), but also we maximize over the last action (off-policy).

The Monte Carlo algorithm does not use bootstrapping. It instead sums up all the rewards that were observed during each episode. By doing this it does not rely on any estimate of the value function to update the Q-value estimate. The target is calculated by:

$$G_t = \sum_{i=0}^{\infty} \gamma^i \cdot r_{t+i} \tag{9}$$

The general algorithm for the n-step Tabular learning is shown in Algorithm 3. The new concept here is the episode. Our algorithm learns in episodes which means that it performs a certain number of steps without updating the Q-values but keeps a record of all the states, actions, and rewards. Then the algorithm uses them to update the Q-values. The updating is different for the two implementations. In n-step Q-learning, we have the update described in Algorithm 4, while updating in Monte Carlo is described in Algorithm 5.

As we can see in n-step Q-learning we use the rewards we obtained over a certain depth, while in Monte Carlo we use all the rewards we obtained in each episode without bootstrapping. The Q-learning update rule also checks if the final step inside the depth is terminal. This is done because in Equation (8) we have the maximum Q-value of all the actions in the final state. If this is the terminal state we are not able to perform this calculation.

In all of our experiments, we average over 10 repetitions to account for the stochasticity of the environment. We

Algorithm 3 n-step Tabular

```
Input: $\gamma$, $\alpha$, $\epsilon$, episode length $T$, depth $n$
Initialize $Q(s, a) = 0$
repeat
    Initialize state $s_0$
    for $t = 0$ to $T - 1$ do
        sample action
        take a step
        get reward and next state
        budget $-= 1$
        if terminal then
            break
        end if
    end for
    $T_{ep} = t + 1$
    update Q-values
until budget $= 0$
Return: $Q(s, a)$
```

Algorithm 4 n-step Q-learning update

```
for $t = 0$ to $T_{ep} - 1$ do
    $m = \min(n, T_{ep} - t)$
    if $s_{t+m}$ is terminal then
        $G_t = \sum_{i=0}^{m-1} \gamma^i \cdot r_{t+i}$
    else
        calculate target using Equation (8)
    end if
    update Q-value using Equation (6)
end for
```

also smooth the plots for better interpretation of the results. The budget for all the algorithms is set to 50 000. The discount rate (γ) is set to 1.0 and the learning rate (α) is set to 0.25. In our first experiment, we test the different action selection policies, by using the Q-learning algorithm for the $\epsilon-$greedy and the Boltzmann softmax policy. In our second experiment, we test on-policy vs off-policy by comparing the Q-learning algorithm with the SARSA. For this experiment, the action selection policy is set to the $\epsilon-$greedy policy with $\epsilon = 0.1$. In our final experiment, we explore the performance of different values of depth. We do that by using the n-step Q-learning algorithm for different values of n and also the Monte Carlo algorithm. For this experiment, we use again the$\epsilon-$greedy policy with $\epsilon = 0.1$. The learning rate is $\alpha = 0.25$, the discount rate is $\gamma = 1.0$ and the episode length T is 150.

3.2. Results

For our first experiment, we are interested to find which action selection policy achieves better results for our problem. We do that by comparing the implementation of the $\epsilon-$greedy and the Boltzmann softmax policy in the Q-

Algorithm 5 Monte Carlo update

```
$G_t = 0$
for $t = T_{ep} - 1$ to $0$ do
    $G_t = r_i + \gamma G_{i+1}$
    update Q-value using Equation (6)
end for
```

learning tabular algorithm.

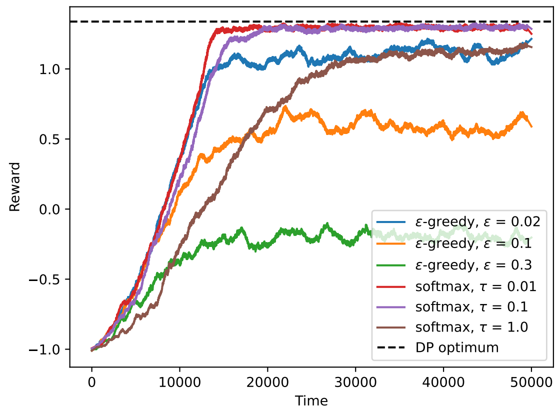

Figure 2. The average reward as a function of time (number of steps) for the $\epsilon-$greedy and the Boltzmann softmax policy. Each line represents a different value of the exploration parameter (ϵ or τ). The dashed line shows the mean reward achieved by the dynamic programming. We see that the softmax policy for low values of τ managed to achieve similar results to the optimal policy.

The results are shown in Figure 2. We explored the two different policies by using three different values for each exploration parameter (ϵ and τ). The values are ranging from what are typically low values ($\epsilon = 0.02$ an $\tau = 0.01$) to typically big values ($\epsilon = 0.3$ an $\tau = 1.0$). We also plot the average mean reward per time step that we calculate in our experimentation with the dynamic programming algorithm ($\text{average}(r) = 1.33$).

We can see that all of the different implementations achieved some level of learning, while some of them performed significantly better. The softmax policy achieved better results than the $\epsilon-$greedy in general. Low values of the τ parameter lead to very similar results when compared with the optimal policy that was learned during the dynamic programming experimentation. The $\epsilon-$greedy performed well only for the lowest value of the ϵ parameter.

From this experimentation, we can conclude that exploitation is favored more than exploration in our environment. Both policies performed significantly better for small values of the exploration parameter which made our agent more "greedy", meaning it was more likely to choose the best action from its current knowledge of the environment.

In our next experiment, we compare the two different ways to calculate the back-up target. As described in Equations (5) and (7) we can use either off-policy or on-policy to back-up information. The first was implemented using the Q-learning algorithm while for the latter we used the SARSA algorithm.

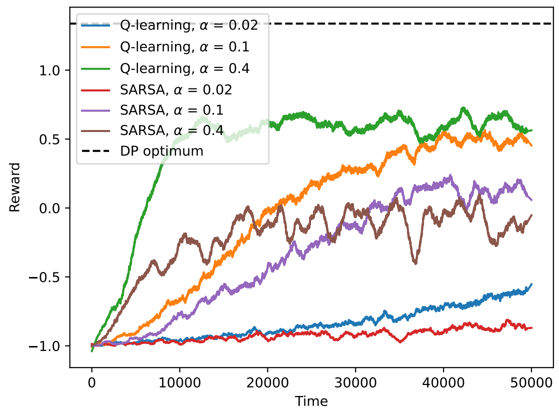

Figure 3. Comparing the rewards over time for the on-policy and off-policy. The Q-learning calculates the back-up using off-policy while the SARSA uses on-policy. For both implementations, we experimented with different values of the learning rate (α). We see that for the same values of the learning rate, Q-learning outperforms the SARSA, while neither of them managed to achieve the mean reward of the optimal policy of the dynamic programming (dashed line).

In Figure 3 we can compare the two different policies for different values of the learning rate. As we can notice for each different value of the learning rate the Q-learning algorithm outperforms SARSA. It also becomes clear that neither of these algorithms reached the optimum that was set during the implementation of the dynamic programming algorithm.

From that, we conclude that off-policy performs better in our environment. This is not a surprise since in our previous experiment we found that "greedy" actions also performed better. Off-policy also exploits those "greedy" actions as it uses the best possible action at the next state to update the Q-values while on-policy uses the actual action that was taken, which may include exploration, which seems to not be favored as much in our environment.

Off-policy can typically learn the optimal policy because it has a lower bias than the on-policy. On the other hand, on-policy is more stable since it has a lower variance. This is one typical example of the bias-variance tradeoff. One possible explanation as to why off-policy performs better can come out of this if we assume our environment is more stable and is therefore more rewarding to lower the bias and achieve a more optimal policy.

For our final experiment, we compare how the different values of depth perform in our n-step Q-learning algorithm. We also compare it to the Monte Carlo algorithm that uses no bootstrapping and updates the Q-values using all the rewards obtained during each episode.

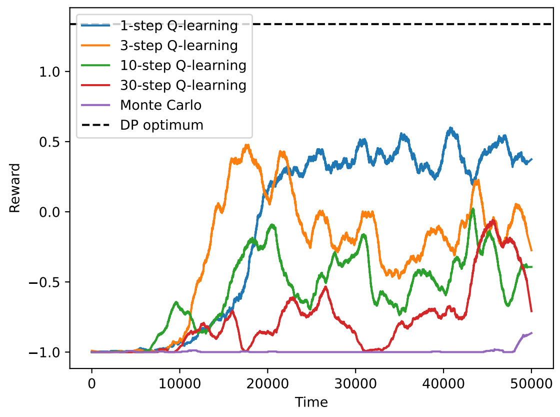

Figure 4. The average reward as a function of time (number of steps) for different values of depth for the n-step Q-learning algorithm, as well as the Monte Carlo algorithm. Monte Carlo uses no bootstrapping and is not able to learn the environment. The n-step Q-learning algorithm performs much better for low values of depth but is still not close to the optimum that was set from the dynamic programming implementation (dashed line).

In Figure 4 we see the final results of our experiment. The average reward for Monte Carlo indicates that it was not able to learn the environment, with a small possibility that it actually managed to achieve some results towards the end of the learning budget. Q-learning on the other hand achieved much better results, especially for small values of depth. One other clear observation is that the 3-step Q-learning algorithm achieved much better results early on but did not reach the same level of final performance as the 1-step algorithm.

To explain all that we can see in the bias-variance trade-off. Low depth can lead to higher bias with lower values of variance. It is clear that this is favored in this particular case as bigger values of depth lead to a worse final performance. The higher bias can also explain why the 3-step Q-learning managed to learn the environment quicker. The bigger values of variance made it more unstable and the final performance suffered as consequence.

In Appendix A we repeat the experiments we discussed in this chapter after changing the values of various parameters, in order to achieve higher performance.

4. Discussion

During this assignment, we experimented with many different algorithms and tested them for different values of the input parameters.

Dynamic programming managed to solve our problem by finding the optimal policy that will lead our agent from the start-state to the goal-state. It set the benchmark for all the other model-free algorithms. The final performance of the agent after dynamic programming was only matched by the Q-learning algorithm using the Boltzmann softmax policy for low values of the temperature parameter. All the other algorithms couldn't achieve the same level of performance. It is however worth mentioning one more time that dynamic programming only works in ideal situations where the agent has access to the model of the environment and can not be applied to many cases. This is why we have to rely on model-free algorithms for most real-world applications.

When experimenting with different exploration policies we discovered that for our environment exploitation was favored during training. Although exploration is essential for our agent to learn the optimal path, big values of the exploration parameter were not as successful. We also came to the conclusion that the Boltzmann softmax policy worked better in our world, as it is the only policy that managed to reach the optimum that was set during the dynamic programming.

After that, we tested on-policy and off-policy learning. The SARSA algorithm we implemented is an on-policy learning technique as it uses the value of the action that was taken. In contrast, Q-learning utilizes the value of the best possible action to bootstrap and update the values. Both have advantages and drawbacks. On-policy algorithms are considered more stable (Moerland, 2021) while off-policy ones have better performance but might be unstable. During our experiments, we found that for the given environment off-policy (Q-learning) is giving better results for the same value of the exploration parameter. N-step Q-learning is considered an off-policy method because it utilizes the maximum value of the last action. However, it also sums up the rewards of the path that was actually followed during an episode. This means that it combines some aspects of on-policy learning with off-policy learning.

For the next experiment, we decided to test how depth affects the performance of our algorithms. One-step learning methods (usually refer to as Temporal Difference) have low variance which is really beneficial as it means our agent can learn quickly. The signal however now becomes highly biased and might not converge to an optimal policy (Juliani, 2018). On the other extreme we have the Monte Carlo algorithm that omits bootstrapping and as a result has a completely unbiased signal. It suffers however from the high variance that comes from the stochasticity of the environment. This might lead to our agent not being able to find the optimal path. This is the concept of the bias-variance tradeoff. N-step algorithms are somewhere in between the two extremes and have some levels of variance and bias in their learning.

During our implementation, we found that low variance is extremely beneficial to our agent learning the environment. Monte Carlo was not able to learn anything with the same budget and parameters as the n-step Q-learning techniques. 1-step had the best performance, while 3-step Q-learning achieved better rewards early on but was not able to have the same final performance.

As we have already discussed tabular reinforcement learning techniques have many advantages. They can learn quickly and achieve the optimal policy. They suffer however from many limitations and drawbacks. They are not fit to deal with continuous problems unless we can successfully discretize them. They are also computationally expensive, especially for more complex problems that we usually find in real-world applications. As the number of dimensions of our problem increases the number of parameters in the table increases exponentially. Even simple problems in high-dimension space will become impossible to solve as tabular reinforcement learning will become impractical in terms of storage requirements and computation time. This concept is described as the "curse of dimensionality.

To solve these issues we resort to deep reinforcement learning techniques. Deep neural networks can approximate the value function, which can scale to very large state-action spaces. They can also work with continuous signals without the need to discretize the state-action space.

A. Extra experimentation

During our initial experiments in Section 3 we learned some valuable information for the optimal values of the different parameters. Parameter tuning is a very delicate process and requires lots of experimentation. In this appendix, we attempt to use what we learned in order to achieve the optimum results that were set during the dynamic programming (Section 2).

In our experiment for the best selection action policy, we found that the Boltzmann softmax policy was able to achieve a performance that was similar to the optimum for low values of the temperature parameter. The $\epsilon - \text{greedy}$ policy was also able to achieve good results when exploitation was

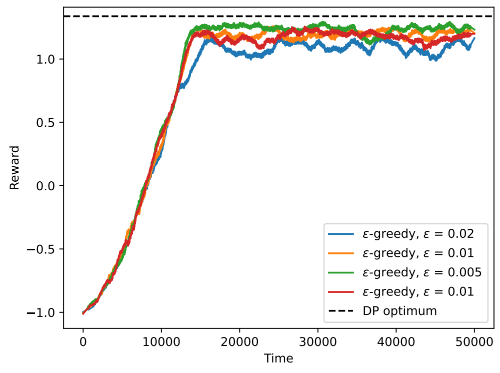

Figure 5. The average reward as a function of time (number of steps) for the $\epsilon -$ greedy policy. Each line represents a different value of the exploration parameter (ϵ). The dashed line shows the mean reward achieved by the dynamic programming. We see that the $\epsilon -$ greedy policy for low values of ϵ managed to achieve similar results to the optimal policy.

favored, so we decided to test whether further decreasing the ϵ parameter will lead to better performance.

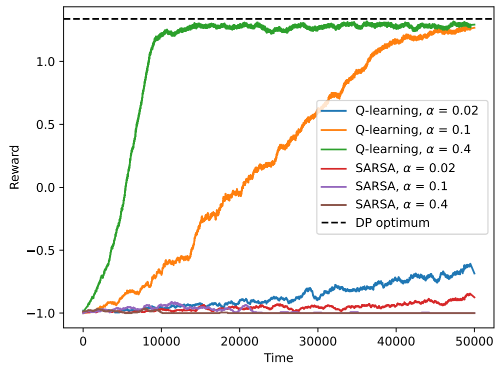

Figure 6. Comparing the rewards over time for the on-policy and off-policy. The Q-learning calculates the back-up using off-policy while the SARSA uses on-policy. For both implementations, we experimented with different values of the learning rate (α). Both implementations now use a softmax action selection policy with $\tau = 0.1$. We see that Q-learning outperforms the SARSA. Q-learning is capable of reaching the optimum (dashed line) within budget for big values of α.

As we can see in Figure 5 this is indeed the case. The best performance for $\epsilon = 0.05$ is very close to the optimal policy achieved during the dynamic programming. Further reducing the exploration parameter does not yield better results. It becomes even more evident that "greediness" is favored in our environment, as both $\epsilon -$ greedy and softmax policies were able to converge to the optimal policy for small values of their respective exploration parameters.

In our next two experiments, we tested the way we back-up information as well as the depth of the target. The testing was done while using a sub-optimal action selection policy ($\epsilon -$ greedy policy for $\epsilon = 0.1$). We decided to conduct the same experiments but this time we chose a policy that we know it can lead to the optimal policy, the Boltzmann softmax policy for $\tau = 0.1$.

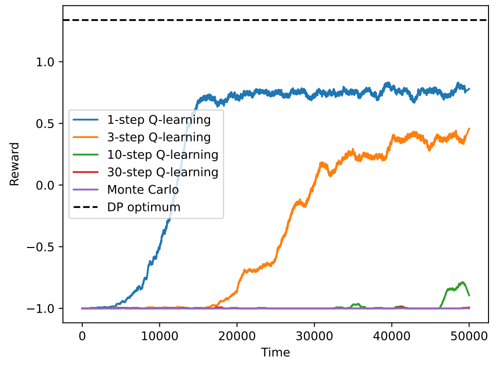

Figure 7. The average reward as a function of time (number of steps) for different values of depth for the n-step Q-learning algorithm, as well as the Monte Carlo algorithm. In this implementation, we use the Boltzmann softmax policy for $\tau = 0.1$. The optimum that was set from the dynamic programming implementation is shown with a dashed line. The 1-step algorithm achieved better results while high-depth algorithms were not able to learn.

The results of the new on-policy vs off-policy experiment are shown in Figure 6. As we can see contrary to the initial experiment we were able to reach the optimum within our budget. The off-policy Q-learning algorithm was once more the best-performing out of the two. It is also more clear that bigger values of learning rate can lead to faster convergence and they have no issue with stability in our environment.

The results for our new depth experiment are shown in Figure 7. The 1-step Q-learning algorithm managed to achieve higher performance when we compare it with the initial experiment. 3-step Q-learning also achieved higher final performance than before but took more time to converge. This is not something unexpected as the lower levels of exploration can lead to slower convergence. Higher-depth algorithms suffered from this change as they were completely unable to learn the environment and achieve higher rewards. The conclusion from all of these could be that exploration becomes more important when we go to higher-depth algorithms.

References

Andrew, A. M. Reinforcement learning: An introduction by richard s. sutton and andrew g. barto, adaptive computation and machine learning series, mit press (bradford book), cambridge, mass., 1998, xviii 322 pp, isbn 0-262-19398-1, (hardback, £31.95). *Robotica*, 17(2):229–235, 1999. doi: 10.1017/S0263574799211174.

Juliani, A. Making sense of the bias / variance trade-off in (deep) reinforcement learning, Feb 2018.

Moerland, T. Continuous markov decision process and policy search. *Lecture notes for the course reinforcement learning, Leiden University*, 28(71):106, 2021.

Plaat, A. Deep Reinforcement Learning, a textbook. *arXiv e-prints*, art. arXiv:2201.02135, January 2022. doi: 10.48550/arXiv.2201.02135.