ASSIGNMENT 1: TABULAR REINFORCEMENT LEARNING

Zhipei Qin
3977226

1 DYNAMIC PROGRAMMING

1.1 The algorithm describes the process of finding the optimal State-action value function $Q^*(s, a)$ and an associated optimal policy $\pi^*(s)$ using the Bellman equation for Q-value iteration. The pseudo code of the algorithm is given as **Algorithm 1**, which is introduced in the assignment task sheet.

Algorithm 1 Tabular Q-value iteration (Dynamic Programming)

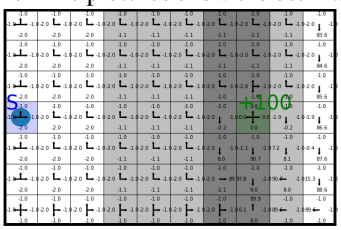
Require: Threshold $\eta \in R^+$
Ensure: The optimal value function $Q^*(s, a)$ and/or associated optimal policy $\pi^*(s)$.

Initialization: A State-action value table $\hat{Q}(s, a) = 0 \quad \forall s \in \mathcal{S}, a \in \mathcal{A}$
repeat
  $\Delta \leftarrow 0$
  for each $s \in \mathcal{S}$ **do**
    for each $a \in \mathcal{A}$ **do**
      $x \leftarrow Q(s, a)$
      $\hat{Q}(s, a) \leftarrow \sum_{s'} [p(s'|s, a) \cdot (r(s, a, s') + \gamma \max_{a'}(s', a'))]$
      $\Delta \leftarrow \max(\Delta, |x - \hat{Q}(s, a)|)$
    end for
  end for
until $\Delta < \eta$;
$Q^*(s, a) = \hat{Q}(s, a)$
$\pi^*(s) = \arg\max_a Q^*(s, a) \quad \forall s \in \mathcal{S}$
return $Q^*(s, a)$ and/or $\pi^*(s)$.

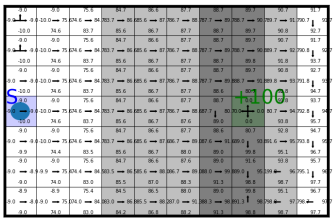
The core of this algorithm is to estimate the Q-value. For each pair of State and action, it returns the expected cumulative reward if that action is performed and then the optimal policy is followed. The algorithm iterates over the Q-values: the input is a threshold η, which determines when to stop iterating. During the iteration, we iterate through each State-action combination and store the current Q-value estimate of each iteration into a temporary variable x. After that, we update $\hat{Q}(s', a')$ according to the Bellman equation: $\hat{Q}(s, a) \leftarrow \sum_{s'} [p(s'|s, a) \cdot (r(s, a, s') + \gamma \max_{a'}(s', a'))]$. This update process involves:

- For each s', calculate the sum of $r(s, a, s')$ and the discounted maximum Q-value of the next State $\gamma \max_{a'} Q(s', a')$. $r(s, a, s')$ refers to the immediate reward obtained when taking action a in a State s and transitioning to State s'. γ is the discount factor, which determines the current value of future rewards and prevents infinite accumulation of rewards.
- Multiply the obtained $r(s, a, s') + \gamma \max_{a'} Q(s', a')$ by the probability of transition to this State $p(s'|s, a)$, and finally calculate these values and, get the total expected return of the current State-action pair. $p(s'|s, a)$ represents the probability of transitioning to State s' after taking action a in State s.
- After that, max error Δ is updated to the larger one between the current error and the previous max error. When $\Delta < \eta$, the Q-value table $\hat{Q}(s', a')$ is considered to have converged to $Q^*(s', a')$, and the iteration ends. The algorithm returns the optimal value function $Q^*(s', a')$ and/or the optimal policy $\pi^*(s)$, where $\pi^*(s)$ is determined by the action a that maximizes $Q*$ in each State s.

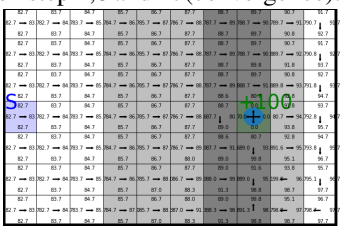
1.2 The pictures show the estimates at each State-action at the iteration step 1,8 and 17(convergence).



Q-value iteration step 1



Q-value iteration step 8



Q-value iteration step 17

The Q-values for all actions at the goal will be zero because it is a terminal State; No further rewards can be accumulated after reaching the goal. Working backward from the goal, we look for the State preceding it. The preceding State is the one corresponding to the action with the highest Q-value among all actions that can reach the goal. In this experiment, the preceding State is $(s = 57, location = (5, 8))$. Observing the Q-values of the four actions at this State [93.8 96.7 98.7 99.8], we find that the Q-value for moving left (the fourth element) is the highest. This indicates that in the State before reaching the goal, the optimal strategy is to moving left. When the agent moves left at $s = 57$, due to the influence of stochastic vertical wind, the agent has a 90% probability of reaching the goal. According to the algorithm, we analyze the entire Q-value table and always choose the action that leads to the highest Q-value. Similarly, we can determine that the State before $s = 57$ is $(s = 63, location = (6, 9))$, with the corresponding action being to move left at $s = 63$, with a Q-value of 98.7. From this, we find the optimal path from the starting point $s = 3$ to the goal as $s : 3 \rightarrow 10 \rightarrow 17 \rightarrow 25 \rightarrow 33 \rightarrow 41 \rightarrow 48 \rightarrow 55 \rightarrow 62 \rightarrow 69 \rightarrow 68 \rightarrow 67 \rightarrow 66 \rightarrow 65 \rightarrow 64 \rightarrow 63 \rightarrow 57 \rightarrow 52$ in this experiment. It can be observed that in this optimal path, the corresponding Q-value of each step's action is greater than that of the previous action.

1.3 The experiment was repeated 20 times and the converged optimal value at the start State $V^*(s = 3)$ = 83.678 was obtained. This value represents the expected cumulative reward that can be obtained from the starting State to the goal State when adopting the optimal policy. After the Q-value iteration converges, the maximum Q-value of each State corresponds to the value when the optimal action is taken in the State. Therefore, for the starting State $s = 3$, find the Q-values of all actions a corresponding to this State, and select the largest value from these Q-values, that is, $\max_a Q(s, a)$, then this value is the converged optimal value of the start State.

1.4 The optimal value at the start State $V(s = 3)$ can be represented as: $V(s = 3) = R_g * 1 + R_o * (x - 1)$ Where R_g = 100 is the magnitude of the final reward, R_o = -1 is the magnitude of the reward on every other step. x is the number of steps the agent needs to take to reach the goal. Thus, $x = 101 - V(s = 3)$. And the average reward per step is given by $\overline{R} = \frac{V(s=3)}{x} = \frac{V(s=3)}{101-V(s=3)}$. In 20 repetitions of experiments, the average reward per timestep under the optimal policy is 4.924.

1.5 The dynamic programming algorithm always converges because it takes into account all possible States and actions. If the goal State is terminal, the algorithm will still converge because for non-terminal States, it continuously updates their Q-values until convergence. The Q-value of terminal States is always set to 0 and remains unchanged during iteration, thus it does not affect the algorithm's ability to ultimately find the optimal Q-value for all States through the iterative process.

1.6 In 20 repetitions of experiments, when the goal State is at [7,3], the agent only takes 17 actions on average to reach the goal, however when the goal State is at [6,2], the agent takes significantly more actions (an average of 41.2), and the number of actions performed in each experiment is also very different (the least one went through 14 timesteps, and the most one went through 74 timesteps). This shows that when the goal State is at [6,2], the agent's movement has greater uncertainty. Further analysis reveals that in each experiment, the agent often experiences a few timesteps of "stalling" in the States $(s = 22, location = [1, 3])$, $(s = 29, location = [1, 4])$, and $(s = 36, location = [1, 5])$ under the optimal policy. This is because, in these three States, the optimal strategy is to move downward, which happens to counteract the effect of the stochastic vertical wind present in columns

3, 4, and 5. Choosing any other action in these three States would likely result in the agent being "blown upwards" by the wind, making it unable to reach the goal. Only when the wind does not blow can the agent move downward, and in the subsequent State, the optimal strategy for the agent shifts to moving right to approach the goal.

2 EXPLORATION

2.1: Methods. In this part, two types of policies used in reinforcement learning algorithms are used for the action selection: The ϵ-greedy policy and the Boltzmann policy (the softmax policy). Mathematically, the ϵ-greedy policy can be expressed as:

$$a = \begin{cases} \arg\max_a Q(s,a) & \text{with probability } 1-\epsilon, \\ \text{random action from } A(s) & \text{with probability } \epsilon. \end{cases}$$

Where $Q(s,a)$ is the estimated value of action a in State s, A is the set of all possible actions in State s. The parameter ϵ determines the likelihood of taking a random action. With probability ϵ, select a random action from the set of all possible actions. With probability $1-\epsilon$, select the greedy action, i.e., the action with the highest Q-value for the current State.
The formula for calculating $P(a|s)$ (i.e. the probability of selecting action a in State s) according to the Boltzmann policy is given by:$P(a|s) = e^{\frac{Q(s,a)}{\tau}} / \sum_{b \in A} e^{\frac{Q(s,b)}{\tau}}$.
Where τ is the temperature parameter. When the temperature parameter τ is high, the probabilities assigned to each action are more evenly distributed, which promotes a greater degree of exploration across different actions. When τ is low, the decision-making process becomes more deterministic, with a stronger preference for selecting the action that currently has the highest estimated value, thus favoring exploitation over exploration.

2.2: Experiments. In this experiment, we compare ϵ-greedy and softmax exploration with different parameter settings: ϵ-greedy with epsilon = [0.03,0.1,0.3] and softmax with temps = [0.01,0.1,1.0]. The results are averaged over 20 repetitions. The learning curves for each setting together with the optimum value obtained by Dynamic programming are shown in the figure below. The total timesteps are set to 50001 and we run independent evaluation episodes after every 1000 steps. The y-axis of the figure represents the mean return at these episodes, and x-axis represents the timestep.

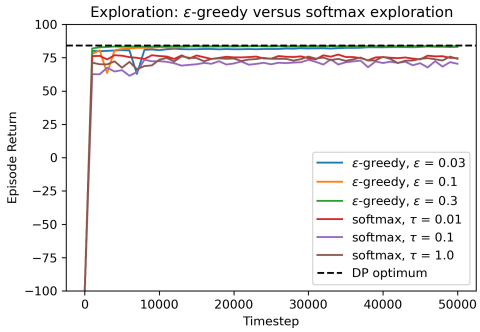


Figure 1: The learning curves for each setting together with the DP optimum

2.3: Discussion. It is clearly shown in the figure that under all the three settings of ϵ, the ϵ-greedy policy almost achieves the optimal performance from Dynamic Programming within 10,000 steps. In contrast, the softmax policy under the three τ settings performs worse, and the best-performing setting $\tau = 0.01$ only achieves about 75 episode return. Therefore, in this experiment, the ϵ-greedy policy should be adopted. The ϵ-greedy strategy provides a deterministic balance between exploration and exploitation, while the softmax policy may be more sensitive to the stochasticity in the environment, leading to less stable learning compared to the ϵ-greedy policy.

2.4: Multiple Goals. We add a second goal $(s = 38, location = (5,3))$ to the environment with a reward of +10. We continue to try parameter settings: ϵ-greedy with epsilon = [0.03,0.1,0.3] and

softmax with temps = [0.01, 0.1,1.0]. The total timesteps are set to 30001 and we run independent evaluation episodes after every 1000 steps. Similarly, take the average of 20 repeated experiments, and plot Their respective learning curves are shown in the figure below.

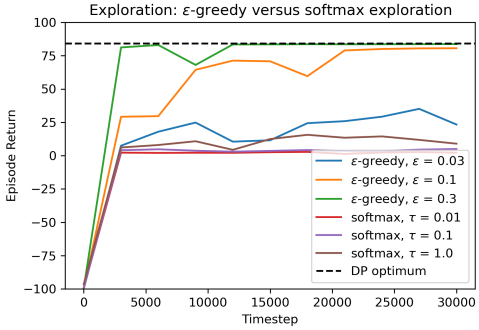


Figure 2: The learning curves for each setting when adding another goal $s = 38$

For the ϵ-greedy policy, setting ϵ to be 0.03 does not result in an episode return above 25. However, with ϵ values of 0.1 and 0.3, the episode return approaches the dynamic programming optimum after 30,000 timesteps, with $\epsilon = 0.3$ reaching it more quickly. This indicates that with a smaller ϵ, the agent tends to discover only the high-reward state at (5,3) and miss the highest reward at (7,3). A larger ϵ increases the likelihood of exploring the state (7,3). For the softmax policy, when the τ is set to smaller values (0.01, 0.1), the agent fails to explore the maximum reward state (7,3). Increasing τ to 1.0 improves performance, but the agent still struggles to avoid local optima.
Comparing the the single-goal situation, when the environment has two goals, even if an agent discovers a high-reward state early on, it should explore more rather than only exploit, to avoid missing the state with the highest reward. This is reflected in the change of the optimal exploration parameter: in this multiple-goals case, the largest ϵ and τ should be chosen for deeper exploration. To enhance the performance of the softmax policy, the τ value should be increased further.

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

3.1 Methods. In this part, we discuss and implement an on-policy reinforcement learning method called SARSA. The pseudo code of the algorithm is given in **Algorithm 2**, which is introduced in the assignment task sheet.

In the iteration loop of SARSA, the SARSA update rule, given observations s, a, r, s', a' is $\hat{Q}(s,a) \leftarrow \hat{Q}(s,a) + \alpha \left[r + \gamma \hat{Q}(s',a') - \hat{Q}(s,a) \right]$. Where $\hat{Q}(s,a)$ is the estimated Q-value for the current State-action pair, α is the learning rate, r is the reward, γ is the discount factor, $\hat{Q}(s',a')$ is the estimated Q-value for the next State-action pair. This rule is to adjust the Q-value for the current State-action pair (s,a).
The key aspect of SARSA is that it updates the Q-values using the action that was actually taken (a'), which aligns with the policy being followed, hence being referred to as an on-policy method. This contrasts with off-policy methods like Q-learning, which use the maximum Q-value for the next State regardless of the action taken.

3.2 Experiments. In this experiment, we compare on-policy (SARSA) and off-policy (Q-Learning) with different learning rates: [0.03,0.1,0.3]. The results are averaged over 20 repetitions. The total timesteps are set to 30001 and we run independent evaluation episodes after every 1000 steps. The comparison of the learning curves is shown in the Figure3 below:

3.3: Discussion. For Q-Learning, when the learning rate is set to 0.03, 0.1, and 0.3, the algorithm can all reach the DP optimum. In contrast, SARSA performs much worse. Even with the learning rate set to 0.3, the algorithm learns slower than Q-Learning and achieves a poorer episode return.
In this task, the Q-learning is more preferred. In environments where "risky" actions can have severe negative consequences, such as when setting a high penalty state near a goal state, SARSA is prob-

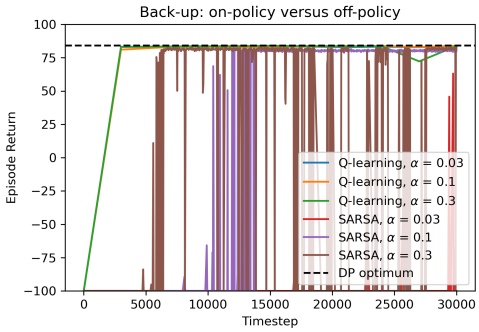


Figure 3: Q-Learning versus SARSA with learning rates [0.03,0.1,0.3]

ably preferred. This is due to SARSA's conservative nature considering the exploration tendencies of the current policy. It learns policies that avoid significant losses by anticipating the outcomes of actions within that policy.

Algorithm 2 Tabular SARSA

Input: Exploration parameter, learning rate $\alpha \in (0, 1]$, discount parameter $\gamma \in [0, 1]$, 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**
  $r, s' \sim p(r, s'|s, a)$
  $a' \sim \pi(a'|s')$
  $\hat{Q}(s, a) \leftarrow \hat{Q}(s, a) + \alpha * \Big[r + \gamma * \hat{Q}(s', a') - \hat{Q}(s, a)\Big]$
  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: $\hat{Q}(s, a)$

Algorithm 3 Tabular n-step Q-learning

Input: Exploration parameter $\epsilon \in (0, 1]$, learning rate $\alpha \in (0, 1]$, discount parameter $\gamma \in [0, 1]$, maximum episode length T, target depth n.
$\hat{Q}(s, a) \leftarrow 0, \forall s \in S, a \in A$.
while $budget$ **do**
  $s \sim p_0(s)$.
  for $t = 0 \dots (T - 1)$ **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 $t = 0$ to $(T_{ep} - 1)$ **do**
    $m \leftarrow \min(n, T_{ep} - t)$.
    if s_{t+m} is $terminal$ **then**
      $G_t \leftarrow \sum_{i=0}^{m-1} (\gamma)^i r_{t+i}$.
    else
      $G_t \leftarrow \sum_{i=0}^{m-1} (\gamma)^i r_{t+i} + (\gamma)^m \max_a \hat{Q}(s_{t+m}, a)$.
    end if
    $\hat{Q}(s_t, a_t) \leftarrow \hat{Q}(s_t, a_t) + \alpha \Big[G_t - \hat{Q}(s_t, a_t)\Big]$.
  end for
  Return: $\hat{Q}(s, a)$

4 Back-up: Depth of target

4.1 Methods. In this part, we discuss and implement tabular n-step Q-learning and tabular Monte Carlo. The pseudo code of the n-step Q-learning is given in **Algorithm 3**, which is introduced in the assignment task sheet. After the initial state is sampled, the n-step Q-learning algorithm does an episode collection, meaning it continuously samples actions (using strategies such as -

greedy or softmax), rewards, and states until the episode ends or reaches its maximum length. After the episode data is collected, the algorithm computes the n-step target for updating the Q-value: $G_t \leftarrow \sum_{i=0}^{m-1}(\gamma)^i r_{t+i} + (\gamma)^m \max_a \hat{Q}(s_{t+m}, a)$. This equation is understood as summing the discounted rewards up to n steps ahead (or until the episode ends) for each timestep within the episode. If the episode does not end within n steps, it also adds the discounted value of the Q-value at the nth step. Then, the Q-value for the current state-action pair is updated according to $\hat{Q}(s_t, a_t) \leftarrow \hat{Q}(s_t, a_t) + \alpha \left[G_t - \hat{Q}(s_t, a_t)\right]$.
The Monte Carlo method is another update method that does not use bootstrapping to update the value function. Instead, it updates the value of state-action pairs by directly estimating the expected return. This is done by accumulating all rewards in a complete episode and performing a Monte Carlo update at the end of an episode. For each state-action pair, it sums all the discounted rewards from the current timestep until the end of the episode. This total sum is $G_t = \sum_{i=0}^{\infty}(\gamma^i \cdot r_{t+i})$.

5 REFLECTION

5.1 DP versus RL: Given a complete Markov Decision Process model, Dynamic Programming (DP) can guarantee to find the optimal policy and value function the transition probabilities and reward functions are known accurately. A potential weakness of DP is that it can be computationally infeasible for large state spaces due to the "curse of dimensionality", because it requires an exhaustive evaluation of all possible state and action combinations. As the number of states and actions increases, the amount of computation grows exponentially.

5.2 Exploration: According to the experimental results of part 2, when there are single goal and two goals in the environment, ϵ-greedy policy outperforms softmax policy in both circumstances. In fact, I think that compared to ϵ-greedy policy, for the softmax policy, it is more difficult for us to set the temperature parameter τ reasonably. Another possible exploration method is the decaying ϵ-greedy policy. The intuition behind decaying ϵ is to start with a high exploration rate to encourage the agent to try various actions and learn from a wide range of experiences. As training progresses and the agent gathers knowledge, the emphasis shifts towards exploiting this learned information to make the best decisions, requiring a lower exploration rate. This approach ensures that the agent explores sufficiently early on but relies more on its accumulated knowledge later in training.

5.3 Exploration 2: Set the default reward per step to 0 and again implement the three settings of the ϵ-greedy policy and the softmax policy in 10001 timesteps. The learning curve is shown below:

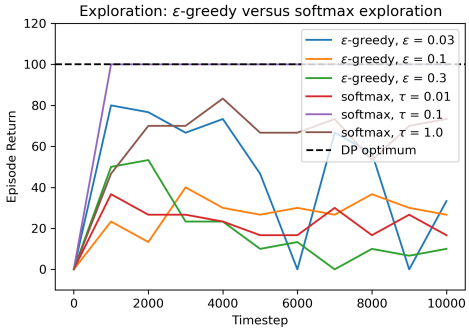


Figure 4: The learning curves for each setting when setting the default reward per step to 0

When the temperature parameter τ is set to 0.1, the softmax policy is capable of reaching the DP optimum (100). In contrast, none of the three settings of the -greedy policy produce desirable results within 10,001 timesteps.

5.4 Discount factor: When γ is set to 1, the rewards obtained in the future will not shrink over time, and the agent will evaluate each of its actions based on the sum total of all of its future rewards. Since the agent no longer receives penalties for each step it takes, , so the agent does not intentionally search for the shortest path. If γ is set to 0.99, the agent will be more inclined to choose the shorter path, because even if the same reward is obtained in the future, the return of the longer path

will be slightly reduced due to the discount effect.

5.5 Back-up, on-policy versus off-policy: The SARSA update formula is $Q(s,a) \leftarrow Q(s,a) + \alpha\left[r + \gamma Q(s',a') - Q(s,a)\right]$, where the update depends on a', indicating it uses the current policy π, hence it's called on-policy. In contrast, Q-learning updates with $Q(s,a) \leftarrow Q(s,a) + \alpha\left[r + \gamma \max_{a'} Q(s',a) - Q(s,a)\right]$, not using a' but rather the action that maximizes the Q-value, differing from the current policy π, making it off-policy.

For on-policy methods, the performance of the behavior policy during training is the same as the actual target policy. In contrast, for off-policy methods, the performance of the behavior policy during training might not be good, but the actual target policy performs better than the behavior policy. The advantage of on-policy is the real-time evaluation of the policy and the ability to train and use it in application scenarios simultaneously, though it can be slow and inefficient. Off-policy methods benefit from more extensive exploration and generating rich samples without affecting the target policy, showcasing stronger utilization of experience samples. However, they may be more prone to errors and instability due to potential overestimation or underestimation of some actions' values.

n-step Q-learning doesn't qualify as an off-policy method, because the first n rewards are obtained through actions selected by the current policy. Consequently, the target for updates predominantly aligns with the behavior policy, blending elements of both on-policy and off-policy methods.

5.6 Back-up, target depth:

5.7 Curse of dimensionality: Tabular RL algorithms are ideal for scenarios where the state-action space is sufficiently small to allow the storage of value functions or Q-values in tables. These algorithms are known for their strong convergence properties in tabular reinforcement learning problems. For example, One of the tabular RL algorithm, dynamic programming, is efficient in our task, and it is guaranteed to find optimal policy. Tabular RL algorithms run into trouble primarily due to the "curse of dimensionality," a term that describes the exponential growth in computational complexity and memory requirements as the number of dimensions (state or action spaces) increases. This makes it practically impossible to store and update the value for every possible state-action pair in high-dimensional or continuous spaces.

6 BONUS

Annealing exploration

In the study of section 2.4 (multiple goals), it was found that to encourage an agent to discover the state with the highest return, it is advisable to increase both the exploration parameter ϵ and the temperature parameter τ to encourage exploration. Therefore, in the experiment, three different ϵ values [0.1, 0.5, 0.9], and three different τ values [0.1, 1.0, 10.0] were tried. A second goal $(s = 31, location = (4,3))$ with a reward of +20 was added to the environment. The results show that only the softmax policy with $\tau = 10.0$ was successfully trained within 10,000 time steps, supporting the conclusions drawn about the setting of the τ value in section 2.4.

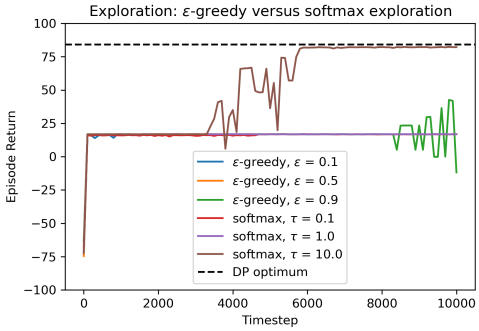


Figure 5: The learning curves for each setting when adding another goal $s = 31$