

# 1 MDPs: Warm-Up

1. What is the Markov Property?

The **Markov Property** states that action outcomes depend on the current state only (action outcomes do not depend on the past states and actions).

2. What are the Bellman equations, and when are they used?

The **Bellman Equations** give a definition of "optimal utility" via expectimax recurrence. They give a one-step lookahead relationship between utilities at a current time step and the next time step.

3. What is a policy? What is an optimal policy?

A **policy** is a function that maps states to actions;  $\pi(s)$  gives an action for state  $s$ .  
An **optimal policy** is a policy that maximizes the expected utility if an agent follows it.

4. How does the discount factor  $\gamma$  affect how the agent finds the optimal policy? Why do we restrict gamma  $\gamma < 1$ ?

$\gamma$  determines how much the value of a state should take into account the value of future states that the agent could wind up in.

A gamma  $0 < \gamma < 1$  helps our algorithms converge and to prevents against infinite rewards if our game lasts forever.

5. Fill in the following table explaining the effects of having different gamma values:

$\gamma$	Effect on policy search:
$\gamma = 0$	
$\gamma = 1$	

$\gamma$	Effect on policy search:
$\gamma = 0$	A smaller $\gamma$ indicates smaller "horizon," or a shorter term focus. When $\gamma$ is 0, we only consider immediate rewards. We do not consider rewards in the future as having any value.
$\gamma = 1$	The higher the discount factor, the more the state would value distant future states. When $\gamma$ is 1, we begin to act as though rewards at any given point in time are equally valuable.

6. What are the two steps to Policy Iteration?

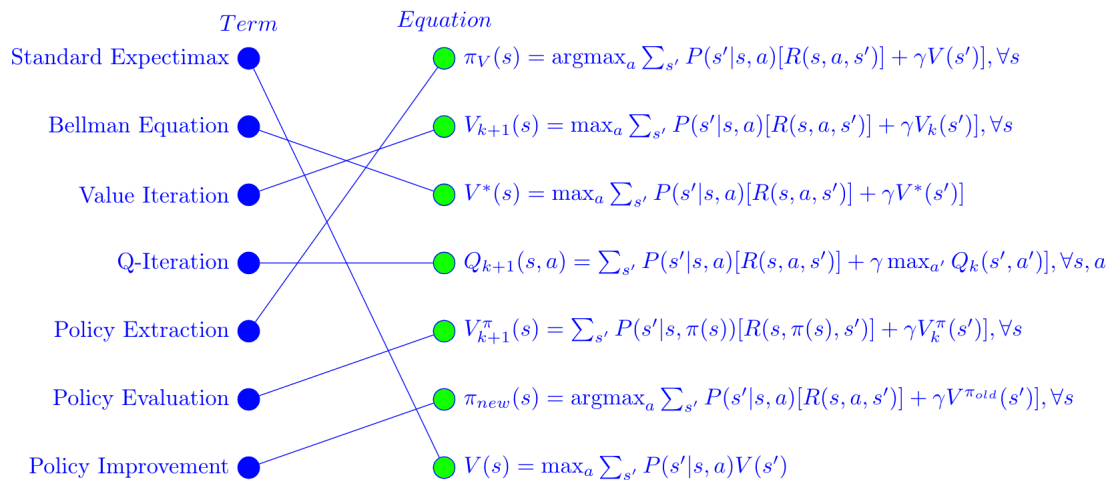
## Policy evaluation and policy improvement.

7. What is the relationship between  $V^*(s)$  and  $Q(s, a)$ ?

$$V^*(s) = \max_a Q(s, a)$$

8. (MDP Notation Review) Draw a line connecting each term in the left column with its corresponding equation in the right column.

<i>Term</i>	<i>Equation</i>
Standard Expectimax ●	● $\pi_V(s) = \operatorname{argmax}_a \sum_{s'} P(s' s, a)[R(s, a, s')] + \gamma V(s')], \forall s$
Bellman Equation ●	● $V_{k+1}(s) = \max_a \sum_{s'} P(s' s, a)[R(s, a, s')] + \gamma V_k(s')], \forall s$
Value Iteration ●	● $V^*(s) = \max_a \sum_{s'} P(s' s, a)[R(s, a, s')] + \gamma V^*(s')]$
Q-Iteration ●	● $Q_{k+1}(s, a) = \sum_{s'} P(s' s, a)[R(s, a, s')] + \gamma \max_{a'} Q_k(s', a')], \forall s, a$
Policy Extraction ●	● $V_{k+1}^\pi(s) = \sum_{s'} P(s' s, \pi(s))[R(s, \pi(s), s')] + \gamma V_k^\pi(s')], \forall s$
Policy Evaluation ●	● $\pi_{new}(s) = \operatorname{argmax}_a \sum_{s'} P(s' s, a)[R(s, a, s')] + \gamma V^{\pi_{old}}(s')], \forall s$
Policy Improvement ●	● $V(s) = \max_a \sum_{s'} P(s' s, a)V(s')$

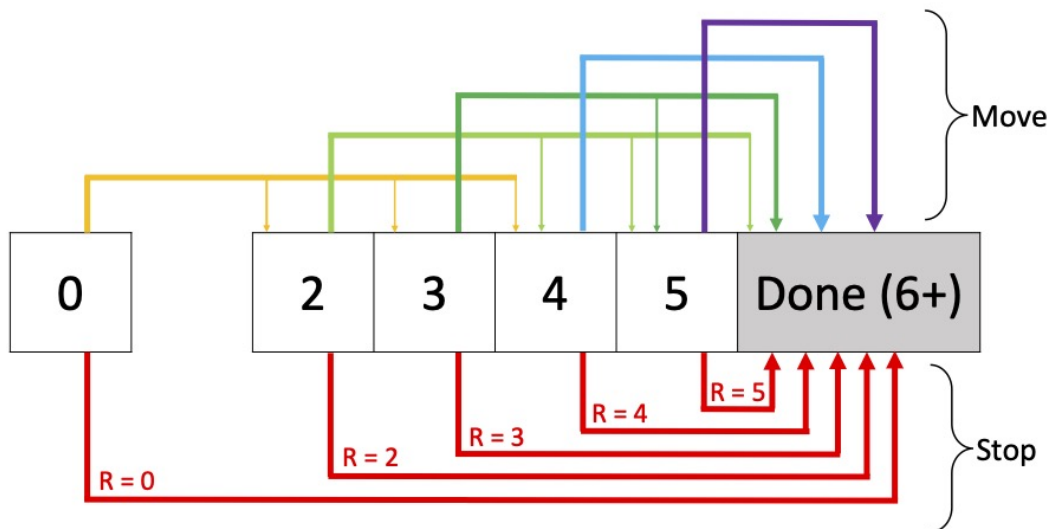


## 2 MDPs: Racing

Consider a modification of the racing robot car example seen in lecture. In this game, the car repeatedly moves a random number of spaces that is equally likely to be 2, 3, or 4. The car can either Move or Stop if the total number of spaces moved is less than 6.

If the total spaces moved is 6 or higher, the game automatically ends, and the car receives a reward of 0. When the car Stops, the reward is equal to the total spaces moved (up to 5), and the game ends. There is no reward for the Move action.

Let's formulate this problem as an MDP with the states  $\{0, 2, 3, 4, 5, Done\}$ .



1. What is the transition function for this MDP? (You should specify discrete values for specific state/action inputs.)

$$\begin{aligned}
 T(s, Stop, Done) &= 1, \text{ for } s \neq Done \\
 T(0, Move, s') &= \frac{1}{3} \text{ for } s' \in \{2, 3, 4\} \\
 T(2, Move, s') &= \frac{1}{3} \text{ for } s' \in \{4, 5, Done\} \\
 T(3, Move, 5) &= \frac{1}{3} \\
 T(3, Move, Done) &= \frac{2}{3} \\
 T(4, Move, Done) &= 1 \\
 T(5, Move, Done) &= 1 \\
 T(s, a, s') &= 0 \text{ otherwise.}
 \end{aligned}$$

2. What is the reward function for this MDP?

$$\begin{aligned}
 R(s, Stop, Done) &= s, s \leq 5 \\
 R(s, a, s') &= 0 \text{ otherwise}
 \end{aligned}$$

3. Recall the value iteration update equation:

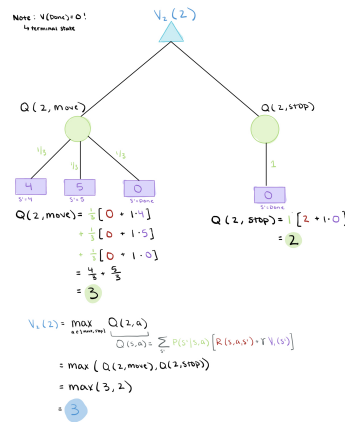
$$V_{k+1}(s) = \max_a \sum_{s'} T(s, a, s') [R(s, a, s') + \gamma V_k(s')]$$

Perform value iteration for 4 iterations with  $\gamma = 1$ .

States	0	2	3	4	5	Done
$V_0$						0
$V_1$						0
$V_2$						0
$V_3$						0
$V_4$						0

States	0	2	3	4	5	Done
$V_0$	0	0	0	0	0	0
$V_1$	0	2	3	4	5	0
$V_2$	3	3	3	4	5	0
$V_3$	$\frac{10}{3}$	3	3	4	5	0
$V_4$	$\frac{10}{3}$	3	3	4	5	0

Value iteration tree for  $V_2(2)$  :



4. You should have noticed that value iteration converged above. What is the optimal policy?

States	0	2	3	4	5
$\pi^*$					

States	0	2	3	4	5
$\pi^*$	Move	Move	Stop	Stop	Stop

5. How would our results change with  $\gamma = 0.1$ ?

By decreasing  $\gamma$ , we focus more and more on immediate rewards. This effectively makes our algorithm more greedy, valuing short-term rewards more than long-term ones.

For this game, with a discount factor of 0.1, value iteration converges in fewer iterations, but upon a different policy (Move, Stop, Stop, Stop, Stop). For state 2, the algorithm preferred the short-term reward of Stopping over the long-term reward of Moving.

In the most extreme case with  $\gamma = 0$ , the values converge immediately and yield an optimal policy of Stop for all states other than state 0, and either Move or Stop at state 0 (because all actions at state 0 lead to an expected utility of 0).

6. Now imagine we changed the rules so that moving to spaces after 5 does not immediately end the game, but rather wraps back around to the beginning states (this is the case where the states exist side by side in a loop).

For example if you start in state 5, select the action Move, and move 2 spaces, you would end up in state 1 (this is a new state we would need introduce).

To be clear, the states would now be states  $\{0,1,2,3,4,5, \text{Done}\}$ .

How would this modification change our results? (Assume we again use  $\gamma=1$ )

This modification essentially adds cycles to our state-space graph. With a discount factor  $\gamma=1$ , cycles in the state-space graph will allow for values to feedback into other states. This feedback would occur for every cycle introduced and each would prolong convergence of value iteration.

The new feedback for each state would also change the converged value of each state and the optimal policy.

### 3 MDPs: Policy Iteration

Recall the racing MDP from the prior problem. In this game, the car repeatedly moves a random number of spaces that is equally likely to be 2, 3, or 4. The car can either Move or Stop if the total number of spaces moved is less than 6.

If the total spaces moved is 6 or higher, the game automatically ends, and the car receives a reward of 0. When the car Stops, the reward is equal to the total spaces moved (up to 5), and the game ends. There is no reward for the Move action.

States:  $\{0, 2, 3, 4, 5, Done\}$

Transition	Reward
$T(s, Stop, Done) = 1, \forall s \neq Done$	$R(s, Stop, Done) = s, \forall s \leq 5$
$T(0, Move, s') = \frac{1}{3}, \forall s' \in \{2, 3, 4\}$	$R(s, a, s') = 0$ otherwise
$T(2, Move, s') = \frac{1}{3}, \forall s' \in \{4, 5, Done\}$	
$T(3, Move, 5) = \frac{1}{3}$	
$T(3, Move, Done) = \frac{2}{3}$	
$T(4, Move, Done) = 1$	
$T(5, Move, Done) = 1$	
$T(s, a, s') = 0$ otherwise	

Now recall the policy evaluation and policy improvement equations, which together make up policy iteration.

$$\text{Policy Evaluation: } V_{k+1}^{\pi}(s) = \sum_{s'} T(s, \pi(s), s') [R(s, \pi(s), s') + \gamma V_k^{\pi}(s')]$$

$$\text{Policy Improvement: } \pi_{new}(s) \leftarrow \operatorname{argmax}_a \sum_{s'} T(s, a, s') [R(s, a, s') + \gamma V^{\pi_{old}}(s')]$$

Perform two iterations of policy iteration for one step of this MDP, starting from the fixed policy below. Use

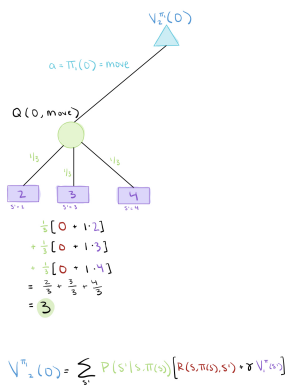
States	0	2	3	4	5
$\pi_0$	Move	Stop	Move	Stop	Move
$V_0^{\pi_0}$					
$V_1^{\pi_0}$					
$V_2^{\pi_0}$					
$V_3^{\pi_0}$					
$\pi_1$					
$V_0^{\pi_1}$					
$V_1^{\pi_1}$					
$V_2^{\pi_1}$					
$V_3^{\pi_1}$					
$\pi_2$					

the initial  $\gamma = 1$ .

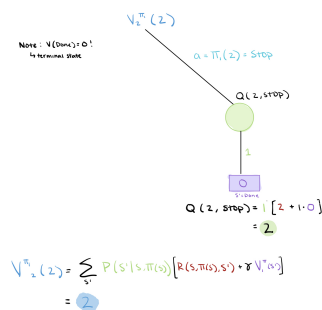
States	0	2	3	4	5
$\pi_0$	Move	Stop	Move	Stop	Move
$V_0^{\pi_0}$	0	0	0	0	0
$V_1^{\pi_0}$	0	2	0	4	0
$V_2^{\pi_0}$	2	2	0	4	0
$V_3^{\pi_0}$	2	2	0	4	0
$\pi_1$	Move	Stop	Stop	Stop	Stop
$V_0^{\pi_1}$	0	0	0	0	0
$V_1^{\pi_1}$	0	2	3	4	5
$V_2^{\pi_1}$	3	2	3	4	5
$V_3^{\pi_1}$	3	2	3	4	5
$\pi_2$	Move	Move	Stop	Stop	Stop

**Side Note:** Above when evaluating  $\pi_1$ , we started policy evaluation off with all 0s again - this is called a cold start (terminology not super important). We also could have started off instead with the optimal values we'd converged upon last round of policy evaluation ( $V_3^{\pi_0}$ ), which often converges upon the optimal values for  $\pi_1$  faster.

Policy iteration tree for  $V_2^{\pi_1}(0)$  :



Policy iteration tree for  $V_2^{\pi_1}(2)$  :



## 4 MDPs: Conceptual Questions

1. What are the key distinctions between the value iteration and policy iteration algorithms, and when might you prefer one to the other?

Key distinctions (possible answers): policy iteration is focused on evaluating the policies themselves, while value iteration evaluates states or state-action pairs and implicitly derives a policy from there.

2. What are some limitations of value iteration? What are some limitations of policy iteration?

Limitations of value iteration:

- (a) each iteration takes  $O(|S|^2|A|)$  time, which can be costly depending on the size of  $S$  and  $A$ ;
- (b) values of many states do not change in one iteration, but the process has to continue as long as there is change in some states;
- (c) sometimes the corresponding policy (extract the policy as if the current  $V_k$  is  $V^*$ ) has already converged to optimal, but the values have not converged and therefore we have to continue the value iteration process, which is a waste of time.

Limitation of policy iteration:

- (a) Each iteration of policy iteration involves policy evaluation.  
Recall that the equation for policy evaluation is  $V_{k+1}^\pi(s) = \sum_{s'} P(s'|s, \pi(s))[R(s, \pi(s), s')] + \gamma V_k^\pi(s')]$ ,  $\forall s$   
Since policy evaluation is an iterative process and is only guaranteed to converge as  $k \rightarrow \infty$ , we do not know the time of convergence a priori.

Depending on the conditions, one algorithm may converge faster than the other.

3. When does policy iteration end? Immediately after policy iteration ends (without performing additional computation), do we have the values of the optimal policy?

Policy iteration ends when the policy converges, i.e., when  $\pi_{new} = \pi_{old}$  after running policy improvement. We do have the values for the optimal policy. Since we necessarily ran policy evaluation on the most recent iteration of policy iteration, we have the value function  $V^{\pi_{old}}$  corresponding to  $\pi_{old}$ . We know that  $\pi_{new} = \pi_{old}$ , implying that  $V^{\pi_{old}} = V^{\pi_{new}}$ . Therefore, we have  $V^{\pi_{new}}$ , which is the value function corresponding to the optimal policy  $\pi_{new}$ .

4. What changes if during policy iteration, you only run one iteration of Bellman update instead of running it until convergence? Do you still get an optimal policy?

You will get an optimal policy if the termination condition is updated.

*Note:* policy iteration can converge too early because the policy might not change between two iterations of changing values. For an example, refer to the value iteration demo in Lecture 15, where we run policy extraction after every iteration of value iteration. There are a couple of early iterations where the non-optimal policy doesn't change.

The key observation here is that policy iteration is effectively the same as value iteration, since value iteration involves one step of evaluation as well. In this case, we update values based on our current best policy, and keep doing that until convergence of a policy!

Recall that the equation for value iteration is:

$$V_{k+1}(s) = \max_a \sum_{s'} T(s, a, s')[R(s, a, s') + \gamma V_k(s')] \quad (1)$$

And the equation for policy evaluation is:

$$V_{k+1}^{\pi_i}(s) = \sum_{s'} T(s, \pi(s), s') [R(s, \pi(s), s') + \gamma V_k^{\pi_i}(s')] \quad (2)$$

given some fixed policy  $\pi$  that we update in the policy improvement step, given by:

$$\pi_{i+1}(s) \leftarrow \arg \max_a \sum_{s'} T(s, a, s') [R(s, a, s') + \gamma V_k^{\pi_i}(s')] \quad (3)$$

which looks very similar to policy extraction done in value iteration.

## 5 RL: What's Changed Since MDPs?

1. Recall the Bellman Equation we used in MDPs to determine the value of a given state:

$$V^*(s) = \max_a \sum_{s'} T(s, a, s') [R(s, a, s') + \gamma V(s')]$$

What information do we no longer have direct access to in the transition to RL?

In the switch from MDP to RL, we don't have access to some information on the environment model, namely the entire transition function:  $T(s, a, s')$  and the reward function:  $R(s, a, s')$ . Instead, in RL, we receive episodes of information, or one sequence of states, actions, and rewards (one point in our  $T$  and  $R$  functions). RL still has an MDP as its backbone, but we don't have access to the complete transition and reward functions.

2. What is the difference between online and offline learning? Which type of learning does MDP use? How about RL?

Online learning agents learn values and policies by actually taking actions in the environment, offline learners learn solely by analyzing the dynamics (transition and reward functions) of the environment model, without actually needing to take actions.

MDPs, having access to the dynamics and environment model, use offline learning methods like Q-value iteration or policy iteration which never need to take actions in the environment to learn an optimal value/policy. RL agents, on the other hand, must take actions in the environment in order to gain information on these dynamics to which MDPs have a priori access.

## 6 RL: Conceptual Questions

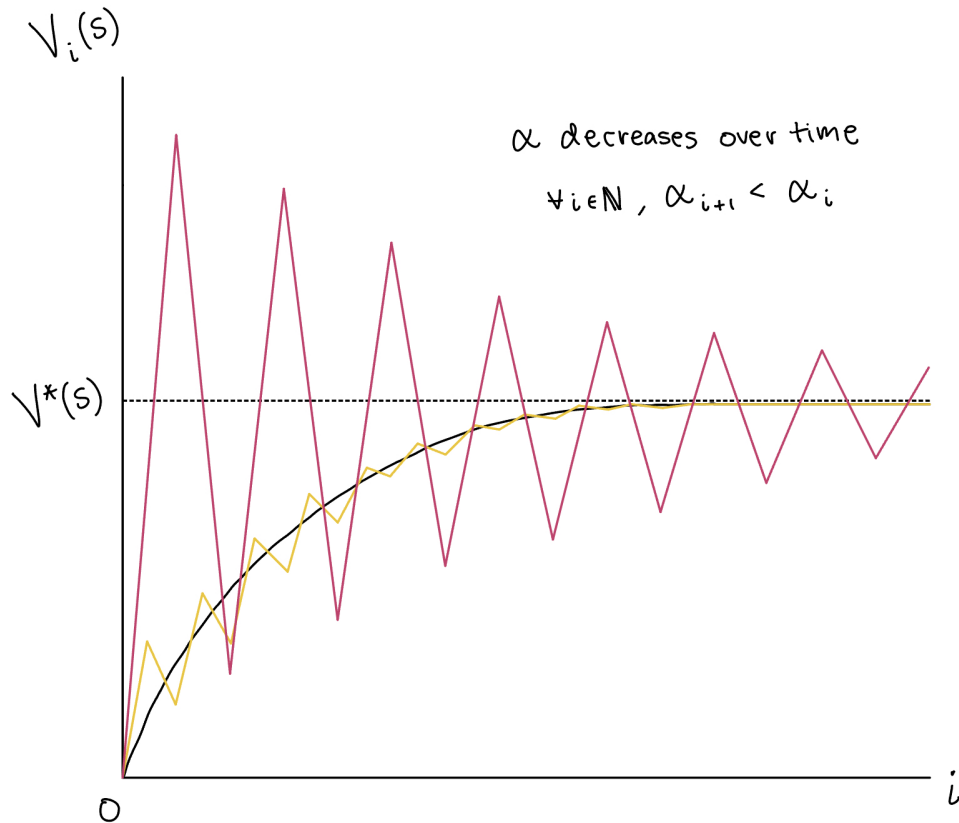
Recall that in Q-learning, we continually update the values of each Q-state by learning through a series of episodes, ultimately converging upon the optimal policy.

1. What's the main shortcoming of TD learning that Q-learning resolves?

TD value learning provides a value for each state for a given policy  $\pi$ . It is impossible to get the optimal policy directly from the learned values because the state values are learned for the given policy  $\pi$ . And if we want to follow policy iteration to extract an improved policy from these values, we would need to use the  $R$  and  $T$  functions (which we don't have). With Q-learning, we can get values of Q-states (i.e., (state, action) pairs) of the optimal policy, from which we can extract an optimal policy simply by taking the action corresponding to the maximum Q-value from each state.



2. We are given two runs of TD-learning using the same sequence of samples but different  $\alpha$  values depicted in the plot below. Assume the dashed horizontal line represents the optimal value for a specific state  $s$  and the black curve represents the smoothest transition to the optimal value given this sequence of samples. In both runs  $\alpha$  decreases over time (or iterations), but one run has  $\alpha$  values larger than the other run at any point in time. Which run (red or yellow) corresponds to the smaller values of  $\alpha$ ? How do the relative sizes of  $\alpha$  affect the rate of convergence to the optimal value?



The yellow run has smaller  $\alpha$  values over time as the changes in  $V^*(s)$  are smaller, showcasing the smaller weighting to new samples. The larger the  $\alpha$  (or the longer  $\alpha$  stays large compared to 0), generally the longer it takes for the run to reach convergence.

3. We are given a pre-existing table of current estimate of Q-values (and its corresponding policy), and asked to perform  $\epsilon$ -greedy Q-learning. Individually, what effect does setting each of the following constants to 0 have on this process?

Remember that in  $\epsilon$ -greedy Q-learning, we follow the following formulation for choosing our action:

$$\text{action at time } t = \begin{cases} \arg \max_{Q(s,a)} & \text{with probability } 1 - \epsilon \\ \text{any action } a & \text{with probability } \epsilon \end{cases}$$

TD value learning provides a value for each state for a given policy  $\pi$ . It is impossible to get the optimal policy directly from the learned values because the state values are learned for the given policy  $\pi$ . And if we want to follow policy iteration to extract an improved policy from these values, we would need

to use the  $R$  and  $T$  functions (which we don't have). With Q-learning, we can get values of Q-states (i.e., (state, action) pairs) of the optimal policy, from which we can extract an optimal policy simply by taking the action corresponding to the maximum Q-value from each state.

(a)  $\alpha$

$Q(s, a) = Q(s, a) + \alpha[r + \gamma \max_{a'} Q(s', a') - Q(s, a)]$  becomes  $Q(s, a)$ .

We put 0 weight on newly observed samples, never updating the Q-values we already have.

Additional remarks about the value of  $\alpha$ :  $\alpha$  is the learning rate or step size determining to what extent newly acquired information overrides old information. When the environment is stochastic, the algorithm converges under some technical conditions on the learning rate that require it to decrease to zero. In practice, sometimes a constant learning rate is used, such as  $\alpha_t = 0.1$  for all  $t$ . If you want to learn more about learning rate in Q-learning, you can search for research papers, e.g., Even-Dar and Mansour, JMLR 2005 (<http://www.jmlr.org/papers/volume5/evendar03a/evendar03a.pdf>).

(b)  $\gamma$

$Q(s, a) = Q(s, a) + \alpha[r + \gamma \max_{a'} Q(s', a') - Q(s, a)]$  becomes  $(1 - \alpha)Q(s, a) + \alpha r$ .

Our valuation of reward becomes short-sighted, as we weight Q-values of successor states with 0. Continue the Q-learning process with  $\gamma = 0$  and gradually decreasing  $\alpha$  will eventually lead to Q-values of  $Q(s, a) = \sum_{s'} T(s, a, s') R(s, a, s')$  because we only care about immediate reward.

(c)  $\epsilon$

By definition of an  $\epsilon$ -greedy policy, we randomly select actions with probability  $\epsilon$  and select our policy's recommended action with probability  $1 - \epsilon$ ; we exclusively exploit the policy we already have.

4. Consider a variant of the  $\epsilon$ -greedy Q-learning algorithm that is changed such that instead of using the policy extracted from our current Q-values, we use a fixed policy instead. We still perform exploration with probability  $\epsilon$ . If this fixed policy happens to be optimal, how does the performance of this algorithm compare to normal  $\epsilon$ -greedy Q-learning?

Both algorithms will result in finding the optimal Q-values eventually. However, normal  $\epsilon$ -greedy Q-learning makes more mistakes along the way, racking up more *regret* (the difference between actual yielded rewards and the optimal expected rewards).

In practice, normal  $\epsilon$ -greedy Q-learning with a small  $\epsilon$  may lead to a policy that is "pretty good" but not necessarily optimal, thus making it very unlikely for it to change unless given an extremely high number of iterations to allow for random chance to find a better policy. This result is known as a local optimum.  $\epsilon$ -greedy Q-learning is in spirit similar to the simulated annealing algorithm in local search.

5. Recall the count exploration function used in the modified Q-update:

$$f(u, n) = u + \frac{k}{n + 1}$$

$$Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma \max_{a'} f(Q(s', a'), N(s', a')) - Q(s, a)]$$

Remember that  $k$  is a hyperparameter that the designer chooses, and  $N(s', a')$  is the number of times we've visited the  $(s', a')$  pair. What is the effect of increasing or decreasing  $k$ ?

Count exploration Q-learning incentives agents to explore states it's seen less or hasn't seen by "weighting" Q-values of state-action pairs we don't visit often with  $\frac{k}{n+1}$ . This is because if we haven't seen a state-action pair often, our  $n$  will be small, so the amount we're adding to our updated Q-value will be greater. Thus if we increase  $k$ , we give more weight to lesser-seen states in our final policy, and if we decrease  $k$ , we give less weight to lesser-seen states.

6. Let's revisit the [Nim code](#). What RL strategies does **AgentRL** employ? Does it evaluate states or Q-states?

**AgentRL** plays out each game (either randomly playing each round with probability  $\epsilon$  if `explore_mode` is on, or by exploiting its learned policy) and records the lose/win rate for each state, action pair seen along the way.

**AgentRL** uses direct evaluation, with an option to execute  $\epsilon$ -greedy exploration. It evaluates Q-states.

7. Contrast the following pairs of reinforcement learning terms:

- (i) Off-policy vs. on-policy learning

An off-policy learning algorithm learns the value of the optimal policy independently of the policy based on which the agent chooses actions. Q-learning is an off-policy learning algorithm. An on-policy learning algorithm learns the value of the policy being carried out by the agent.

- (ii) Model-based vs. model-free

In model-based learning, we estimate the transition and reward functions by taking some actions, then solve the MDP using them. In model-free learning, we don't attempt to model the MDP, and instead just try to learn the values directly.

- (iii) Passive vs. active

Passive learning involves using a fixed policy as we try to learn the values of our states, while active learning involves improving the policy as we learn.