Recitation 8 October 27

1 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 in 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.

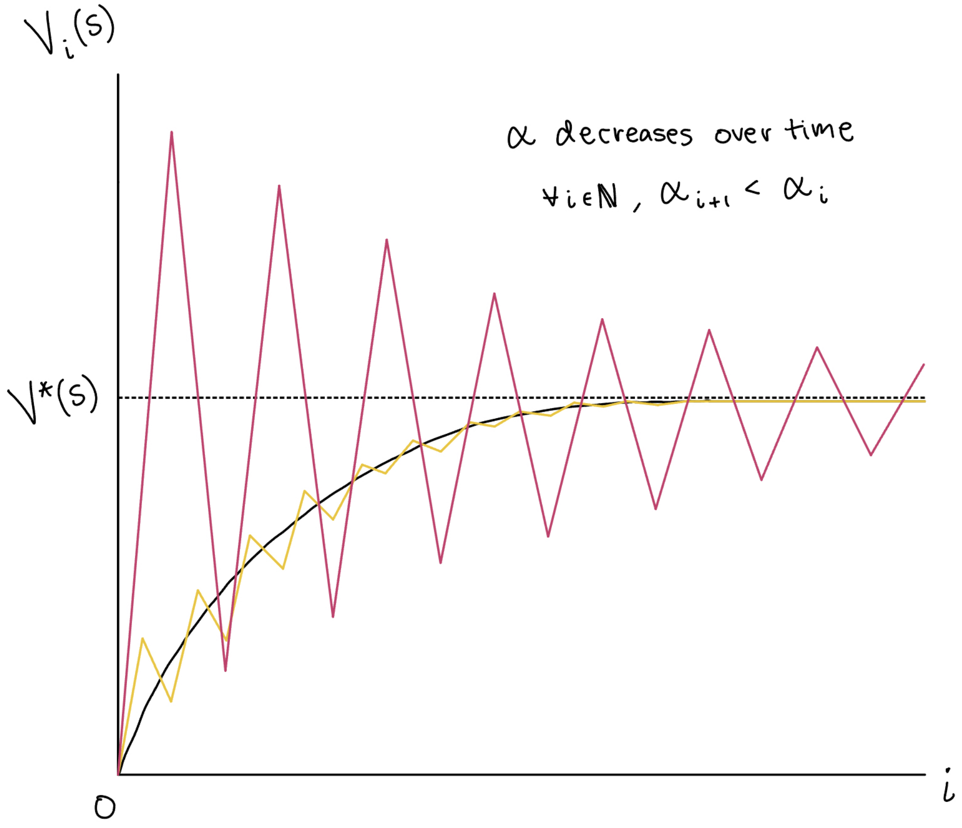
2 RL: Conceptual Questions

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 π. It is impossible to get the optimal policy directly from the learned values because the state values are learned for the given policy π. 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 α 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 α decreases over time (or iterations), but one run has α values larger than the other run at any point in time. Which run (red or yellow) corresponds to the smaller values of α? How do the relative sizes of α affect the rate of convergence to the optimal value?



The yellow run has smaller α values over time as the changes in $V^*(s)$ are smaller, showcasing the smaller weighting to new samples. The larger the α (or the longer α 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 ϵ-greedy Q-learning. Individually, what effect does setting each of the following constants to 0 have on this process?
Remember that in ϵ-greedy Q-learning, we follow the following formulation for choosing our action:

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

TD value learning provides a value for each state for a given policy π. It is impossible to get the optimal policy directly from the learned values because the state values are learned for the given policy π. 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) α

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

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

By definition of an ϵ-greedy policy, we randomly select actions with probability 0 and select our policy's recommended action with probability 1; we exclusively exploit the policy we already have.

4. Consider a variant of the ϵ-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 ϵ. If this fixed policy happens to be optimal, how does the performance of this algorithm compare to normal ϵ-greedy Q-learning?

Both algorithms will result in finding the optimal Q-values eventually. However, normal ϵ-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 ϵ-greedy Q-learning with a small ϵ 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. ϵ-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 ϵ 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 ϵ-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

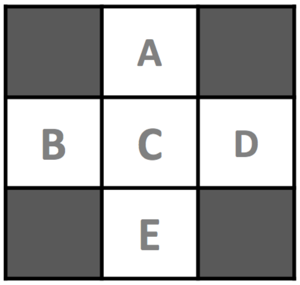
   In model-based learning, we estimate the transition and reward functions by taking some actions, then solve the MDP using them. In model-free learning, we don't attempt to model the MDP, and instead just try to learn the values directly.

   (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.

3 Temporal Difference Learning and Q-Learning

Consider the Gridworld example that we looked at in lecture. We would like to use TD learning to find the values of these states.



Suppose we use an ϵ-greedy policy and observe the following $(s, a, s', R(s, a, s'))$* transitions and rewards:

$(B$, East, C, 2), (C, South, E, 4), (C, East, A, 6), (B, East, C, 2)

**Note that the $R(s, a, s')$ in this notation refers to observed reward, not a reward value computed from a reward function (because we don't have access to the reward function).*

The initial value of each state is 0. Let $\gamma = 1$ and $\alpha = 0.5$.

1. What are the learned values for each state from TD learning after all four observations?

   For (B, East, C, 2), we update $V^\pi(B)$:
   $V^\pi(B) \leftarrow V^\pi(B) + \alpha(R(s, a, s') + \gamma V^\pi(C) - V^\pi(B)) = 0 + 0.5(2 + 1 * 0 - 0) = 1.$
   Following the same computation, we get final values: $V(B) = 3.5$; $V(C) = 4$; $V(s) = 0\ \forall s \in \{A, D, E\}$
   Here are our intermediate computations - the values of each state after each transition are shown below:

Transitions	A	B	C	D	E
(initial)	0	0	0	0	0
$(B, East, C, 2)$	0	1	0	0	0
$(C, South, E, 4)$	0	1	2	0	0
$(C, East, A, 6)$	0	1	4	0	0
$(B, East, C, 2)$	0	3.5	4	0	0

2. In class, we presented the following two formulations for TD-learning:

$$V^\pi(s) \leftarrow (1-\alpha)V^\pi(s) + (\alpha)sample$$

$$V^\pi(s) \leftarrow V^\pi(s) + \alpha(sample - V^\pi(s))$$

Mathematically, these two equations are equivalent. However, they represent two conceptually different ways of understanding TD value updates. How could we intuitively explain each of these equations?

The first equation takes a weighted average between our current values and our new sample. We can think of this as computing an expected value.

The second equation updates our current values towards the new sample value, scaled by a factor of our learning rate, α. This is where the "temporal difference" term is motivated (for those of you familiar, this is gradient descent, where $(sample - V^\pi(s))$ is the gradient.).

3. What are the learned Q-values from Q-learning after all four observations? Use the same $\alpha = 0.5, \gamma = 1$ as before.

$Q(C, South) = 2; Q(C, East) = 3; Q(B, East) = 3;$ $Q(s,a) = 0$ for all other Q-states (s,a).

We use the following Q-value update rule to find what the new value should be (note what is inside of the brackets may also be referred to as *sample* in the slides):

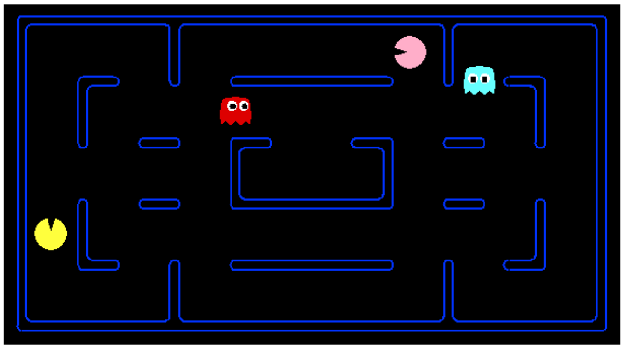
$$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + (\alpha)[R(s,a,s') + \gamma \max_{a'} Q(s'a')]$$

Here are our intermediate computations - the values of each Q-state after each transition are shown below (Q-states for which values did not change are omitted):

Transitions	$(B, East)$	$(C, South)$	$(C, East)$
(initial)	0	0	0
$(B, East, C, 2)$	1	0	0
$(C, South, E, 4)$	1	2	0
$(C, East, A, 6)$	1	2	3
$(B, East, C, 2)$	3	2	3

4 Approximate Q-Learning

Adabelle and Elizabeth are training agents to play AI eTag, which is totally different from Pacman. In this game, the player must find the other player in a maze. However, there are phantoms (note: these are not ghosts) that both players must avoid. However, it is unknown what the scores are for staying alive, for being caught by a phantom, or for finding the other player. Here's a sample board:

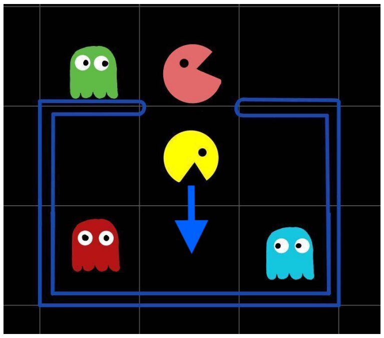


We want to apply Q-learning to this game to train our agents, but it takes a lot of memory to hold the entire grid. To remedy this, we switch to feature-based representation.

1. What features would you extract from an eTag board to judge the expected outcome of the game?

   Lots of possible answers! Can include distances or data on neighboring squares, etc.

2. Say our two features are the number of phantoms in one step of our agent (F_p) and the Manhattan distance to our friend (F_d). Consider the state from the perspective of the yellow agent (the bottom agent). What are the feature values for the following board and action (note: the gridlines are drawn to help measure distance by eye):



   $F_p = f_p(s, South) = 2$. $F_d = f_d(s, South) = 2$.

3. When we get to this state, we have learned weights $w_p = 100$ and $w_d = 10$. Using a linear approximator, calculate the approximate Q-value for the state and action above.

$$Q_w(s, a) = w_p * F_p + w_d * F_d = 200 + 20 = 220$$

4. We have received an episode, which is a start state s, the action $South$, and an end state s', and a reward $R(s, South, s')$. Now, we must update the values. The start state is the state above, and the next state has feature values $F_p = 3$ and $F_d = 1$, and the reward was 20. Assuming discount factor of 0.5, calculate the new estimate of the Q-value for s' **based on the sample**, i.e. $R(s, South, s') + \gamma \max_{a'} Q(s', a')$.

$$\begin{aligned} Q_{new}(s, South) &= R(s, South, s') + \gamma * max_{a'} Q(s', a') && (1) \\ &= 20 + 0.5 * (100 * 3 + 10 * 1) = 20 + (155) = 175 && (2) \end{aligned}$$

5. Now let's update the weights for each feature, given that our learning rate α is 0.5.

   $w_p = w_p + \alpha(Q_{new}(s, South) - Q(s, South)) * F_p(s, South) = 100 + 0.5(175 - 220) * 2 = 100 - 45 = 55$

   $w_d = w_d + \alpha(Q_{new}(s, South) - Q(s, South)) * F_d(s, South) = 10 + 0.5(175 - 220) * 2 = 10 - 45 = -35$

6. At a high-level, what did our approximate Q-learning agent learn here? Once we finish learning, how can we evaluate our agents' performances?

We learned feature weights, which tell us how good or bad certain feature values are. For example, looking at the signs of the weights, it's better to be a small distance to your friend, and larger distance from phantoms.
(Think back to the evaluation functions from P1 - this time, our agent learned how to evaluate the given features on its own!)
After learning, we will be able to run the game and see the rewards the agents end up getting.