Introduction to Machine Learning

Recitation attendance check

Type in your section passcode to get attendance credit (within the first fifteen minutes of your scheduled section).

Passcode: Enter

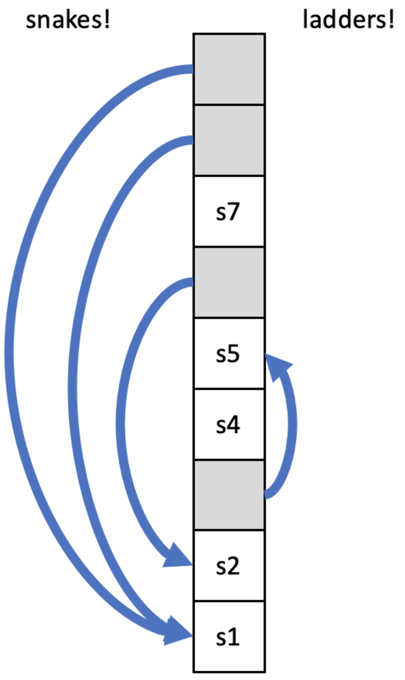
Reinforcement Learning

Due: Wednesday, May 10, 2023 at 11:00 PM

Heads up: All upcoming assignments are due on/before May 12. While submissions are accepted after the 12th, the 20-day automatic extensions will not be applicable beyond May 17 (i.e. submissions after the 17th can still help your learning and are encouraged; but they won't directly count towards your grade). Be sure to plan accordingly.

1) Snakes and Ladders

You are playing a simplified version of a classic game, sometimes called "Snakes and Ladders" in English.



- You are moving up a 1-dimensional track with squares s_1, s_2, s_4, s_5, s_7, as shown above.
- You have two actions: **climb** and **quit**.
- If you **climb** from state s_i then with probability 0.5 you go up one square, and with probability 0.5 you go up two squares.

- **However!** if the square you land on has a ladder going *up* from it, then you automatically, instantaneously, with probability 1 move to the square the ladder goes to.
- **And!** if the square you land on has a snake going *down* from it, then you automatically, instantaneously, with probability 1 move to the square the snake goes to.
- **So:** for example, in our case, if you start in state s_5 and **climb** there is a .5 chance you'll end up in square s_2 (because you move up one but hit a snake and fall back down) and a .5 chance you'll end up in square s_7. If you **climb** from s_7 then you will *go back to square 1* with probability 1.0.
- If you **quit** then the game is over and you get to take no further actions.
- Each new episode starts in state s_1.
- The reward for choosing **climb** in any state is 0.
- The reward for choosing **quit** in state s_i is i.

In the following, you need to supply actions or Q-values for the states s_1, s_2, s_4, s_5 and s_7.

1.1)

What is the optimal horizon 1 policy?

Give your answer as a list of five strings, each being either "climb" or "quit", corresponding to the optimal action for s_1, s_2, s_4, s_5, and s_7, in that order.

['quit', 'quit', 'quit', 'quit', 'quit']

Check Formatting Submit View Answer 100.00%

You have infinitely many submissions remaining.

1.2)

If you initialize the Q values of all the states to 0, and do one iteration of undiscounted ($\gamma = 1$) value iteration, what is the resulting Q value function?

Give your answer as a 2 x 5 array of numbers, in the form

$$[[Q(s_1, \mathbf{quit}), \dots, Q(s_7, \mathbf{quit})], [Q(s_1, \mathbf{climb}), \dots, Q(s_7, \mathbf{climb})]].$$

[[1, 2, 4, 5, 7], [0, 0, 0, 0, 0]]

Check Formatting Submit View Answer 100.00%

You have infinitely many submissions remaining.

1.3)

If $\gamma = 1$ (that is, there is no discounting) what is the optimal infinite-horizon policy?

Give your answer as a list of five strings, each being either "climb" or "quit", corresponding to the optimal action for s_1, s_2, s_4, s_5, and s_7, in that order.

['climb', 'climb', 'climb', 'climb', 'quit']

100.00%

You have infinitely many submissions remaining.

Solution: `['climb', 'climb', 'climb', 'climb', 'quit']`

Explanation:

We only get rewards when we quit, so the maximum reward of 7 is attained by quitting in s_7. Because there is no discounting, the highest reward is obtained by climbing in every state that is not s_7 and then quitting when we get to s_7. Note that this policy induces a finite state markov chain, and the steady-state vector of this markov chain tells us that the long-term probability that we are in s_7 is 1.

2) Sneaking Snakes and Lurking Ladders

What if we have to play snakes and ladders, but we don't know where the snakes and ladders actually are? Assume we know the rewards, but not the transition model. We'll try Q learning to address this problem.

During learning, after executing the **quit** action and getting a reward, the current learning episode terminates. We then initialize a new learning episode, and *we always start the new episode in square 1. We will use a discount factor of $\gamma = 1$ throughout.*

2.1)

When might it be sensible to use value iteration as part of a solution to this problem?

- ☐ When we are using a policy-search approach to reinforcement learning.
- ☑ When we are using a model-based approach to reinforcement learning and have an estimated model for the transition and reward functions.
- ☐ When we are using Q-learning to solve the problem.

100.00%

You have infinitely many submissions remaining.

Solution:

✗ When we are using a policy-search approach to reinforcement learning.

✓ When we are using a model-based approach to reinforcement learning and have an estimated model for the transition and reward functions.

✗ When we are using Q-learning to solve the problem.

Explanation:

From the class notes on reinforcement learning: "The conceptually simplest approach to RL is to model R and T from the data we have gotten so far, and then use those models, together with an algorithm for solving MDPs (such as value iteration) to find a policy that is near-optimal given the current models. Given empirical models for the transition and reward functions, we can now solve the MDP to find an optimal policy using value iteration, or use a finite- depth expecti-max search to find an action to take for a particular state."

2.2)

If we do purely greedy action selection *during* Q-learning (that is $\epsilon = 0$), starting from all 0's in our Q table and where ties are broken in favor of the **climb** action, what (roughly) will the Q function be after 1000 steps?

[✓
$$\begin{array}{cccccc} & s_1 & s_2 & s_4 & s_5 & s_7 \\ \textbf{quit} & 0 & 0 & 0 & 0 & 0 \\ \textbf{climb} & 0 & 0 & 0 & 0 & 0 \end{array}$$
]

$$\begin{array}{cccccc} & s_1 & s_2 & s_4 & s_5 & s_7 \\ \textbf{quit} & 1 & 0 & 0 & 0 & 0 \\ \textbf{climb} & 0 & 0 & 0 & 0 & 0 \end{array}$$

$$\begin{array}{cccccc} & s_1 & s_2 & s_4 & s_5 & s_7 \\ \textbf{quit} & 1 & 2 & 4 & 5 & 7 \\ \textbf{climb} & 0 & 0 & 0 & 0 & 0 \end{array}$$

$$\begin{array}{cccccc} & s_1 & s_2 & s_4 & s_5 & s_7 \\ \textbf{quit} & 1 & 2 & 4 & 5 & 7 \\ \textbf{climb} & 7 & 7 & 7 & 7 & 7 \end{array}$$

100.00%

You have infinitely many submissions remaining.

Solution:
$$\begin{array}{cccccc} & s_1 & s_2 & s_4 & s_5 & s_7 \\ \textbf{quit} & 0 & 0 & 0 & 0 & 0 \\ \textbf{climb} & 0 & 0 & 0 & 0 & 0 \end{array}$$

Explanation:

Because the reward is zero for climbing in any state, $Q(s, \text{climb})$ will be zero for all states. $Q(s, \text{quit})$ will also be zero for all states because we will never take a quit action, so we won't ever update it.

2.3)

If we do purely greedy action selection *during* Q-learning (that is $\epsilon = 0$), starting from all 0's in our Q table and where ties are broken in favor of the **quit** action, what (roughly) will the Q function be after 1000 steps?

○
$$\begin{array}{rccccc} & s_1 & s_2 & s_4 & s_5 & s_7 \\ \textbf{quit} & 0 & 0 & 0 & 0 & 0 \\ \textbf{climb} & 0 & 0 & 0 & 0 & 0 \end{array}$$

[✓]
$$\begin{array}{rccccc} & s_1 & s_2 & s_4 & s_5 & s_7 \\ \textbf{quit} & 1 & 0 & 0 & 0 & 0 \\ \textbf{climb} & 0 & 0 & 0 & 0 & 0 \end{array}$$

○
$$\begin{array}{rccccc} & s_1 & s_2 & s_4 & s_5 & s_7 \\ \textbf{quit} & 1 & 2 & 4 & 5 & 7 \\ \textbf{climb} & 0 & 0 & 0 & 0 & 0 \end{array}$$

○
$$\begin{array}{rccccc} & s_1 & s_2 & s_4 & s_5 & s_7 \\ \textbf{quit} & 1 & 2 & 4 & 5 & 7 \\ \textbf{climb} & 7 & 7 & 7 & 7 & 7 \end{array}$$

100.00%

You have infinitely many submissions remaining.

Solution:
$$\begin{array}{rccccc} & s_1 & s_2 & s_4 & s_5 & s_7 \\ \textbf{quit} & 1 & 0 & 0 & 0 & 0 \\ \textbf{climb} & 0 & 0 & 0 & 0 & 0 \end{array}$$

Explanation:

Assuming when we start in s_1 when we start a new episode, we will never visit any states other than s_1, so the Q values for any states other than s_1 will never change. $Q(s_1, \text{quit}) = 1$ because $R(s_1, \text{quit}) = 1$ and the episode ends with a quit action.

2.4)

Assume we start with a tabular Q function on states s_1, s_2, s_4, s_5 and s_7, initialized to all 0's. Then we get the following experience, consisting of three trajectories, shown below as (state, action, reward) tuples.

Using learning rate $\alpha = 0.5$, what is the Q function after each of these trajectories? *Do not reset all your Q values back to 0 after each trajectory --- assume this is all a single execution of the Q learning algorithm on three episodes of experience.*

For each trajectory, give your answer as a 2 x 5 array of numbers, in the form

$$[[Q(s_1, \textbf{quit}), \ldots, Q(s_7, \textbf{quit})], [Q(s_1, \textbf{climb}), \ldots, Q(s_7, \textbf{climb})]].$$

After $((s_1, \mathbf{climb}, 0), (s_2, \mathbf{climb}, 0), (s_5, \mathbf{climb}, 0), (s_7, \mathbf{quit}, 7))$, what is the Q function?

[[0, 0, 0, 0, 3.5], [0, 0, 0, 0, 0]]

100.00%

You have infinitely many submissions remaining.

Solution: `[[0, 0, 0, 0, 3.5], [0, 0, 0, 0, 0]]`

Explanation:

$$Q(s_1, \text{climb}) = 0.5 * 0 + 0.5 * (0 + \gamma * 0) = 0$$
$$Q(s_2, \text{climb}) = 0.5 * 0 + 0.5 * (0 + \gamma * 0) = 0$$
$$Q(s_5, \text{climb}) = 0.5 * 0 + 0.5 * (0 + \gamma * 0) = 0$$
$$Q(s_7, \text{quit}) = 0.5 * 0 + 0.5 * (7 + 0) = 3.5$$

After $((s_1, \mathbf{quit}, 1)$, what is the Q function?

[[.5, 0, 0, 0, 3.5], [0, 0, 0, 0, 0]]

Check Formatting Submit View Answer 100.00%

You have infinitely many submissions remaining.

After $((s_1, \mathbf{climb}, 0), (s_2, \mathbf{climb}, 0), (s_5, \mathbf{climb}, 0), (s_7, \mathbf{quit}, 7))$, what is the Q function?

[[.5, 0, 0, 0, 1.75], [0, 0, 0, 0, 0]]

Check Formatting Submit View Answer Ask for Help 80.00%

You have infinitely many submissions remaining.

3) Exploration

In this problem, we will work with the "tiny" MDP from exercise 11 (repeated below for convenience), and we will look at the effects of exploration on the success of Q-learning.

Consider an MDP with states (0, 1, 2, 3) and actions ('b', 'c'). The reward function is:

$$R(s,a) = \begin{cases} 1 & \text{if } s = 1 \\ 2 & \text{if } s = 3 \\ 0 & \text{otherwise} \end{cases}$$

You get the reward associated with a state on the step when you take an action in that state. The transition function for each action is below, where $T[i, x, j]$ is the $P(s_{t+1} = j | a = x, s_t = i)$.

$$T(s_t, \text{‘b’}, s_{t+1}) = \begin{bmatrix} 0.0 & 0.9 & 0.1 & 0.0 \\ 0.9 & 0.1 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.1 & 0.9 \\ 0.9 & 0.0 & 0.0 & 0.1 \end{bmatrix}$$

$$T(s_t, \text{‘c’}, s_{t+1}) = \begin{bmatrix} 0.0 & 0.1 & 0.9 & 0.0 \\ 0.9 & 0.1 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.1 & 0.9 \\ 0.9 & 0.0 & 0.0 & 0.1 \end{bmatrix}$$

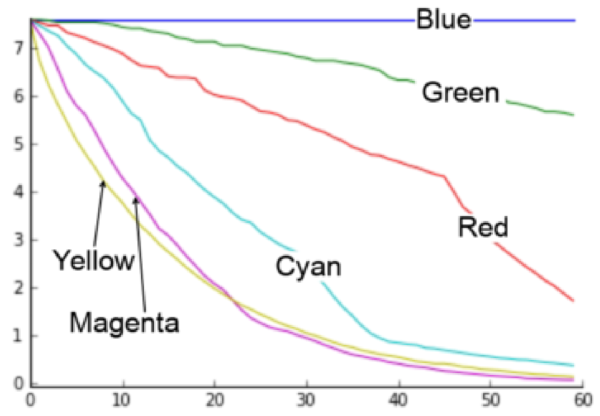
Note that the **only** difference in the effect of the actions is in the transition probabilities out of state 0 (the first row of the transition matrix).

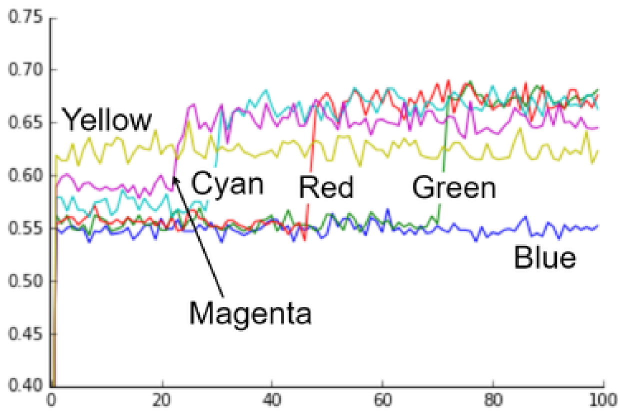
Consider the two plots below. In both cases, the x-axis is the number of steps of Q-learning, plotted in the unit of 1000's (so the actual number of steps is 60,000 in plot 1 and 100,000 in plot 2).

1. In the first plot, the y-axis is the *max norm difference* between the optimal Q function, Q^*, and the learned Q function, $\hat{Q}$:

$$\max_{s,a} |Q^*(s,a) - \hat{Q}(s,a)|$$

2. In the second plot, the y-axis is the average per-step reward, averaged over 1000 steps.





We use an epsilon-greedy action-selection strategy at each iteration. With probability $1-\epsilon$, we greedily select the best action under $\hat{Q}$ (the current Q-value estimates): $\arg\max_a \hat{Q}(s,a)$. With probability ϵ, we select an action uniformly at random (for exploration purposes).

In each plot, we have used a different value of ϵ:

- 0.00 : Blue
- 0.05 : Green
- 0.10 : Red
- 0.25 : Cyan
- 0.50 : Magenta
- 1.00 : Yellow

Here are some points/facts about the algorithm:

1. When there's a tie between action values, our implementation of argmax always returns the first element, which in this case is action 'b'.
2. Even when Q^* is known very well, behavior can be suboptimal due to the exploration.
3. Even when Q^* is not correct, all that matters for optimal behavior (if we don't explore) is that the relative **ordering** of action values is correct.
4. For any state, it is necessary to execute action 'c' from that state some number of times before its estimated Q-value is higher than the estimated Q-value for action 'b' from that state.

For each of the following observations about the plots, mark all the points listed above that are relevant to (help explain) that observation.

3.1)

The blue line is perfectly constant in plot 1 and roughly constant in plot 2.

Mark all the points listed below that are relevant to (help explain) this observation:

- [✓] When there's a tie between action values, our implementation of argmax always returns the first element, which in this case is action 'b'.
- [ ] Even when $Q^\star$ is known very well, behavior can be suboptimal due to the exploration.
- [ ] Even when $Q^\star$ is not correct, all that matters for optimal behavior (if we don't explore) is that the relative **ordering** of action values is correct.
- [✓] For any state, it is necessary to execute action 'c' from that state some number of times before its estimated Q-value is higher than the estimated Q-value for action 'b' from that state.

100.00%

You have infinitely many submissions remaining.

Solution:

✓ When there's a tie between action values, our implementation of argmax always returns the first element, which in this case is action 'b'.

✗ Even when $Q^\star$ is known very well, behavior can be suboptimal due to the exploration.

✗ Even when $Q^\star$ is not correct, all that matters for optimal behavior (if we don't explore) is that the relative **ordering** of action values is correct.

✓ For any state, it is necessary to execute action 'c' from that state some number of times before its estimated Q-value is higher than the estimated Q-value for action 'b' from that state.

Explanation:

The first step is to understand when we do and don't explore. We select a greedy action with probability $1 - \epsilon$, and select an action uniformly at random with probability ϵ. Therefore, the extreme cases are blue ($\epsilon = 0.0$), which always selects the greedy action, and yellow ($\epsilon = 1.0$), which always explores.

We also want to understand the two plots. Plot 1 shows a max norm difference between learned $\hat{Q}$ and optimal $Q^\star$ at each learning step, so it tells us about convergence of our learning algorithm. Plot 2 gives an averaged per-step reward at each learning step. Note that learning steps are plotted in 1000s.

Initially, all Q-values are 0, so the policy that always acts greedily (the blue line) will see a tie initially and pick action B, then continue to always pick action 'b' from then on. Thus, the max norm difference is dominated by all the Q-values for action 'c' being wrong (i.e. 0). See why exploration is important?

3.2)

The yellow line goes down the fastest in plot 1, but doesn't have the best performance in plot 2.

Mark all the points listed below that are relevant to (help explain) this observation:

- ☐ When there's a tie between action values, our implementation of argmax always returns the first element, which in this case is action 'b'.
- ☑ Even when $Q^\star$ is known very well, behavior can be suboptimal due to the exploration.
- ☐ Even when $Q^\star$ is not correct, all that matters for optimal behavior (if we don't explore) is that the relative **ordering** of action values is correct.
- ☐ For any state, it is necessary to execute action 'c' from that state some number of times before its estimated Q-value is higher than the estimated Q-value for action 'b' from that state.

100.00%

You have infinitely many submissions remaining.

Solution:

✗ When there's a tie between action values, our implementation of argmax always returns the first element, which in this case is action 'b'.

✓ Even when $Q^\star$ is known very well, behavior can be suboptimal due to the exploration.

✗ Even when $Q^\star$ is not correct, all that matters for optimal behavior (if we don't explore) is that the relative **ordering** of action values is correct.

✗ For any state, it is necessary to execute action 'c' from that state some number of times before its estimated Q-value is higher than the estimated Q-value for action 'b' from that state.

Explanation:

While the yellow line approaches the optimal Q most quickly (in plot 1), it does not receive the highest average per-step reward (in plot 2), and maintains a relatively constant reward, because its policy is to just always explore. Thus, we can see that even when $Q^\star$ is known very well, exploration can cause suboptimal behavior.

3.3)

The red line goes down slowly in plot 1, and leaps up in the middle of plot 2.

Mark all the points listed below that are relevant to (help explain) this observation:

- ☐ When there's a tie between action values, our implementation of argmax always returns the first element, which in this case is action 'b'.
- ☐ Even when $Q^\star$ is known very well, behavior can be suboptimal due to the exploration.
- ☑ Even when $Q^\star$ is not correct, all that matters for optimal behavior (if we don't explore) is that the relative **ordering** of action values is correct.
- ☑ For any state, it is necessary to execute action 'c' from that state some number of times before its estimated Q-value is higher than the estimated Q-value for action 'b' from that state.

Submit View Answer 100.00%

You have infinitely many submissions remaining.

3.4)

The magenta, cyan, red, and green lines each leap up at some point (each one after the other) in the average rewards plot.

Mark all the points listed below that are relevant to (help explain) this observation:

- ☐ When there's a tie between action values, our implementation of argmax always returns the first element, which in this case is action 'b'.
- ☐ Even when $Q^\star$ is known very well, behavior can be suboptimal due to the exploration.
- ☑ Even when $Q^\star$ is not correct, all that matters for optimal behavior (if we don't explore) is that the relative **ordering** of action values is correct.
- ☑ For any state, it is necessary to execute action 'c' from that state some number of times before its estimated Q-value is higher than the estimated Q-value for action 'b' from that state.

Submit View Answer 100.00%

You have infinitely many submissions remaining.

4) Implement Q-Learning

We'll work up to implementing the Q-learning algorithm by extending our `MDP` and `TabularQ` code from MDP Homework. Use this **Q-learning colab notebook** to answer these questions:

4.1) epsilon_greedy

Write the `epsilon_greedy` method, which takes a state `s` and a parameter `epsilon`, and returns an action. With probability `1 - epsilon` it should select the greedy action and with probability `epsilon` it should select an action uniformly from the set of possible actions.

- You should use `random.random()` to generate a random number to test against eps.
- You should use the `draw` method of `uniform_dist` to generate a random action. (Recall that `uniform_dist` is a function that takes `elts`, and returns an instance of `DDist` with a uniform distribution over `elts`.)
- You can use the `greedy` function we wrote in Homework 10, which takes `(q, s)` and returns a greedy action.

```
def epsilon_greedy(q, s, eps = 0.5):
    """ Returns an action.

    >>> q = TabularQ([0,1,2,3],['b','c'])
    >>> q.set(0, 'b', 5)
    >>> q.set(0, 'c', 10)
    >>> q.set(1, 'b', 2)
    >>> eps = 0.
    >>> epsilon_greedy(q, 0, eps) #greedy
    'c'
    >>> epsilon_greedy(q, 1, eps) #greedy
    'b'
    """
```

Run Code Submit View Answer **100.00%**

You have infinitely many submissions remaining.

4.2) update

Write the `update` method of the class `TabularQ`, which takes a list `data` of `(s, a, t)` experience tuples and a learning rate `lr` and updates `self` with the data from each experience.

Recall that we wrote the `TabularQ` class in MDP Homework. To update a Q value, given a state and action (s, a) with corresponding target t, we change $Q(s, a)$ to be:

$$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha t,$$

where α is the learning rate.

As this is a method of the `TabularQ` class, you should use the other methods to get and set the Q values. Use `self.get(s, a)` to get the value of $Q(s, a)$, and `self.set(s, a, v)` to set $Q(s, a) = v$.

A test for the `update` method has been provided in the Colab. Note that you need to include your `update` method in the `TabularQ` class for the test to run successfully.

```
def update(self, data, lr):
    # Your code here
    for d in data:
        new = self.get(d[0], d[1]) + lr * (d[2] - self.get(d[0], d[1]))
        self.set(d[0], d[1], new)
```

100.00%

You have infinitely many submissions remaining.

Here is the solution we wrote:

```
def update(self, data, lr):
    for (s, a, t) in data:
        old_value = self.get(s, a)
        new_value = (1 - lr) *old_value + lr* t
        self.set(s, a, new_value)
```

4.3) Q_learn

Write a `Q_learn` function that takes an MDP `mdp` and a Q function `q`, runs `iters` iterations of the Q-learning algorithm, and returns the updated `q`. In this version, you will do **one** update in every iteration, i.e., we will collect a single `(s, a)` tuple, and update `Q(s, a)`.

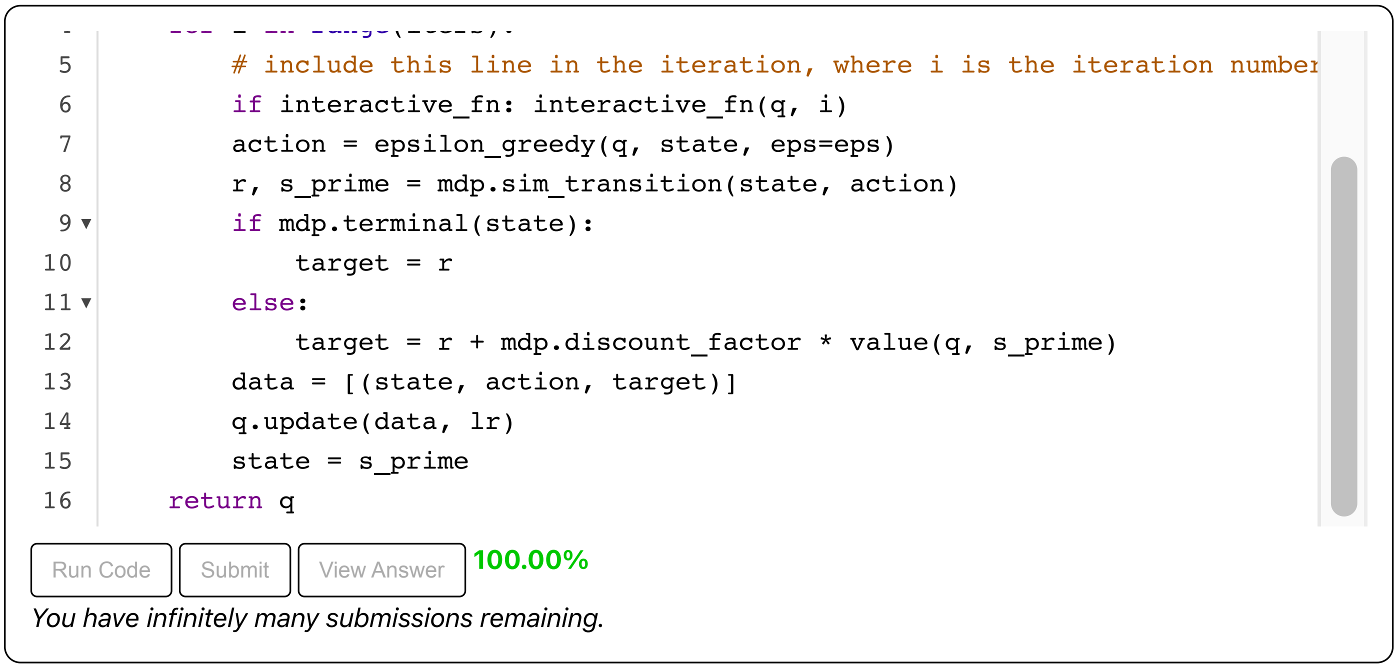
Here's pseudocode for what `Q_learn` should be doing:

```
def Q_learn(...):
    # Let s be the MDP's initial state

    repeat iters times:
        # Choose an action a with epsilon_greedy
        # In the MDP, take action a from state s; let r, s' be reward and next state
        # Compute t, the new Q target, as r plus the future value
        # Update q with (s, a, t)
        # Let s be the new state s'

    return q
```

Some notes:

- The methods of the class `MDP` will be used to choose the initial state and taking actions from states; see the functions `mdp.init_state` and `mdp.sim_transition`.
- The important part here is computing `t`. To compute the future expected value, you will need the discount factor, `mdp.discount_factor`, and the largest Q value of the next state `s'`.
    - To find the largest Q value of a state, use the `value` function you wrote in hw10.
    - If `s` is terminal, then the state `s'` returned by `mdp.sim_transition` will *not* be the actual next state, and instead be a new initial state. In this case, the future expected value should be set to zero. You can check if a state is terminal using `mdp.terminal`.
- To update `q`, call the `update` function you wrote using `q.update`. Note that `q.update` takes a *list* of experience tuples, but here you're updating it with only one tuple.

A test for your `Q_learn` method has been provided in the Colab, where we set up a tiny MDP problem.



```
        # include this line in the iteration, where i is the iteration numbe
        if interactive_fn: interactive_fn(q, i)
        action = epsilon_greedy(q, state, eps=eps)
        r, s_prime = mdp.sim_transition(state, action)
        if mdp.terminal(state):
            target = r
        else:
            target = r + mdp.discount_factor * value(q, s_prime)
        data = [(state, action, target)]
        q.update(data, lr)
        state = s_prime
    return q
```

100.00%

You have infinitely many submissions remaining.

4.4) Q_learn_batch

Write a `Q_learn_batch` function that takes an MDP `mdp` and a Q function `q`, runs `iters` iterations of a *batch* Q-learning algorithm, and returns the updated `q`. In this version, you will do **many** updates in every iteration.

In each iteration, we generate several *episodes*. An episode is a list of experience tuples, each of which is a state, action, reward, next state. The experience tuples happen one after another. An example episode (of length three) will look like `[(s0, a0, r0, s1), (s1, a1, r1, s2), (s2, a2, r2, s3)]`.

Then, we update the Q function using *all* the episodes, even from previous iterations. But we have to *recompute* the target Q values in each iteration. (What will happen if we didn't recompute the target Q values?)

Here's pseudocode for what `Q_learn_batch` should be doing:

```
def Q_learn_batch(...):
    # Let all_experiences be an empty list
```

```
repeat iters times:
    repeat n_episodes times:
        # Let episode be an episode generated by sim_episode
        # Add each (s, a, r, s') experience in episode to all_experiences

    # Let all_q_targets be an empty list

    for each experience in all_experiences:
        # Let (s, a, r, s') be the experience
        # Compute t, the new Q target, as r plus the future expected value
        # Add (s, a, t) to all_q_targets

    # Update q with all_q_targets

return q
```

Some notes:

- To simulate an episode, call the function `sim_episode`. Read its source code, which is in the tests downloaded from Colab notebook.
    - Its `policy` argument needs to be a function that takes a state, and returns the action to take. We want to pick the next action using `epsilon_greedy`. But passing in `epsilon_greedy` won't work, because `epsilon_greedy` takes three arguments, and `policy` needs to have only one argument. That means we need to create a new function, say, `p`, that we *can* pass into `sim_episode`. One way to do this would be to define `p = lambda s: epsilon_greedy(q, s, eps)`.
    - `sim_episode` returns three things: the total reward, the episode itself, and an animation. You can ignore the first and third results.
- Be careful, once again, about computing `t`; its value depends on whether `s` is a terminal state.
    - Unlike `Q_learn`, in this case, if `s` is a terminal state, then `s'` would be `None`, and not an initial state.

A test for your `Q_learn_batch` method has been provided in the Colab, where we set up a tiny MDP problem.

```
            all.extend(ep)

        all_targets = []

        for e in all:
            t = e[2]
            if e[3] is not None:
                t += mdp.discount_factor * value(q, e[3])
            all_targets.append((e[0], e[1], t))

        q.update(all_targets, lr)
    return q
```

Run Code Submit View Answer 100.00%

You have infinitely many submissions remaining.

5) NNQ: Using neural networks to store the Q function

We would like to operate in large or continuous state and/or action spaces where it is not possible (or effective) to store the Q values in a table as we did with the `TabularQ` class. Instead, we will "store" them by training a neural network to do regression for us, taking s, a as input and generating (an approximation of) $Q^*(s, a)$ as output.

To train the network, we will use *squared Bellman error* as the loss function:

$$\left(\left[R\left(s_t, a_t\right) + \gamma \max_{a'} Q\left(s_{t+1}, a'; \theta\right)\right] - Q\left(s_t, a_t; \theta\right)\right)^2$$

where θ stands for the current weights in the neural network and $Q(s, a; \theta)$ stands for the output of the network with weights θ when (s, a) is the input.

There are many choices of neural network architecture for storing Q values. In this problem, we will:

- Focus on the case where we have a small set of possible actions, so we make **one neural network for each possible action** a;
- Design that network with two **hidden** layers with ReLU units and a single linear output unit (although a deeper network could be useful, and our model code should be able to handle different numbers of hidden layers and output units); and
- Use mean squared error (MSE) as the loss function, more specifically, use the MSE error defined in the equation above. (Since we are predicting continuous Q values, which is a regression problem.)

To use a neural net to store Q values, for a given action, we will need to have a mapping from states to fixed-length vectors. We will assume that the `MDP` class has a `state2vec` method that maps states to vectors. For the simple discrete-state MDPs we have seen so far, this simply returns a one-hot representation of the state. For reference, this is our implementation of `state2vec` (note the shape of its returned array):

```
    def state2vec(self, s):
        '''
        Return a vector encoding of state s; used in neural network agent implementations
        '''
        v = np.zeros((1, len(self.states)))
        v[0,self.states.index(s)] = 1.
        return v
```

Now, all we need to do is write a new class, called `NNQ` to implement neural-network version of Q-function storage; then we can pass an `NNQ` instance instead of a `TabularQ` instance into `Q_learn` or `Q_learn_batch`, and we will automatically have reinforcement learning with neural networks!

There are three methods to implement in our `NNQ` class. Here are some ideas for how to do that:

- `__init__`: Create one neural network for each action, and store them in `self.models`. The parameters `num_layers` and `num_units`, passed to `NNQ` when initializing it are important for the construction of these neural networks. Also, note that `actions` is a list that may consist of integers or strings or other objects. As a reminder, here's how to make a new feed-forward network using PyTorch (you can assume this `make_nn` procedure exists and can be called by your code):

```
from torch import nn
def make_nn(state_dim, num_layers, num_units):
    '''
    state_dim  = (int) number of states
    num_layers = (int) number of fully connected hidden layers
    num_units  = (int) number of dense relu units to use in hidden layers
    '''
    model = []
    model += [nn.Linear(state_dim, num_units), nn.ReLU()]
    for i in range(num_layers-1):
        model += [nn.Linear(num_units, num_units), nn.ReLU()]
    model += [nn.Linear(num_units, 1)]
    model = nn.Sequential(*model)
    return model
```

- `get(self, s, a)`: Use the neural network you have stored for action `a` to predict a Q value for state `s`. Feel free to consult a list of PyTorch code examples.
- `update(self, data, lr, epochs = 1)`: As in `TabularQ`, `data` is a list of `(s, a, t)` tuples, where `t` is a target Q value. **(However, you should not expect t to be provided as a scalar. It may be passed in as either a scalar or a 2D array.)** For each action `a`, you will need to:
    - Construct a training set `X, Y` of data that is relevant to action `a`. The input values are states (encoded as vectors) and the output values are the target Q values. `np.vstack` may be useful here. And note that `X, Y` should have shapes `(n,d), (n,1)` where `n` is the total number of data points and `d` is the length of each state vector.
    - Use the provided method `fit(self, model, X, Y, epochs=epochs)` to update the weights in the associated network.

Note: The `update` method of `NNQ` has the same signature as the `update` method of `tabularQ` to allow easy switching between the two classes. However, you **should not** use the `lr` input parameter of the `update` method in `NNQ`. Instead, the learning rate that is provided when initializing `NNQ` and stored as `self.lr` should be used to construct the Pytorch SGD optimizer in the `fit` method.)

The colab and code file contain two tests that use small MDP examples to test your `NNQ` implementation: `test_NNQ` and `test_NNQ2`. For `test_NNQ2`, you should run the code as follows and enter the result below:

Enter the list returned by test_NNQ2: [0.0789480209350586, -0.2069215178489685, 0.0800420

100.00%

You have infinitely many submissions remaining.

Solution: [0.0789480209350586, -0.2069215327501297, 0.0800420418381691, -0.1686033010482788, 0.09363521635532379, -0.19312027096748352, 0.06085893511772156, -0.24304574728012085, 0.04742565006017685, -0.26113757491111755]

Explanation:

Here's the solution we wrote:

```
class NNQ:
    def __init__(self, states, actions, state2vec, num_layers, num_units, lr=1e-
2, epochs=1):
        self.actions = actions
        self.states = states
        self.state2vec = state2vec
        self.epochs = epochs
        self.lr = lr
        state_dim = state2vec(states[0]).shape[1] # a row vector
        self.models = {a : make_nn(state_dim, num_layers, num_units)for a in
actions}
        # Your code here
        self.running_loss = 0.
        self.running_one = 0.
        self.num_running = 0.001

    def predict(self, model, s):
      return model(torch.FloatTensor(self.state2vec(s))).detach().numpy()

    def get(self, s, a):
        return self.predict(self.models[a],s)

    def fit(self, model, X,Y, epochs=None, dbg=None):
      if epochs is None: epochs = self.epochs
      train = torch.utils.data.TensorDataset(torch.FloatTensor(X),
torch.FloatTensor(Y))
      train_loader = torch.utils.data.DataLoader(train,
batch_size=256,shuffle=True)
      opt = torch.optim.SGD(model.parameters(), lr=self.lr)
      for epoch in range(epochs):
        for (X,Y) in train_loader:
          opt.zero_grad()
          loss = torch.nn.MSELoss()(model(X), Y)
          loss.backward()
          self.running_loss = self.running_loss*(1.-self.num_running) +
loss.item()*self.num_running
```

```
            self.running_one = self.running_one*(1.-self.num_running) +
self.num_running
            opt.step()
        if dbg is True or (dbg is None and np.random.rand()< (0.001*X.shape[0])):
          print('Loss running average: ', self.running_loss/self.running_one)

    def update(self, data, lr, dbg=None):
        for a in self.actions:
          if [s for (s,at,t) in data if a==at]:
            X = np.vstack([self.state2vec(s) for (s, at, t) in data if a==at])
            Y = np.vstack([t for (s, at, t) in data if a==at])
            self.fit(self.models[a],X,Y, epochs=self.epochs, dbg=dbg)
```

6) No Exit: Learning to Play

In this part, we are going to test these various methods we experimented with using a very simple game we call *No Exit*, which you experienced in this week's lab.

You can see the game working by running `test_solve_play()` and just hitting return to watch a policy that was found using value iteration and the known model.

The raw representation of a state, for the purposes of No Exit, is

```
((ball_row, ball_column), (ball_row_velocity, ball_column_velocity), paddle_row,
paddle_row_velocity)
```

with a special state for `game_over`. There is a method `state2vec` which will take a state of the game and return a seven-dimensional numpy row vector (the last feature indicates whether the game is over) which you can transform to a FloatTensor to use as an input vector for PyTorch.

In order to implement this problem we have added an additional `interactive_fn` argument to both `Q_learn` and `Q_learn_batch`.

At the end of the colab notebook are a set of test cases that run the various approaches for learning the game.

- Value Iteration - using an explicit model of the MDP
- Q-learning - using a tabular representation in Q_learn
- Batch Q-learning - using a tabular representation in Q_learn_batch
- NN Q-learning - using neural nets to approximate Q in Q_learn
- Fitted Q-learning - using neural nets to approximate Q in Q_learn_batch

For each of the learning method and Q model combinations below, solve the game so that it reliably gets to a score of 100 (that is, the learned game reliably plays 100 steps without missing the ball, earning a score of 100).

During learning, you should see a sequence of lines like: `score (5000, 37.5)`, which indicates that after 5000 iterations the average reward over 10 games is 37.5. We are checking whether you reach a solution that gets an average reward 100 at least one time (i.e. you should see `score (iteration_num, 100)` at least once).

Try playing around with the number of iterations (an argument to `test_learn_play`) until you achieve this point. Note that we will need fewer iterations for Q_learn_batch, in general (check for yourself: why?).

After learning, the code prints a long "upload string" in HEX code. Enter the upload strings in the question boxes below.

If you do not see the `score` printouts it's because you did not call the `interactive_fn` at every iteration. Look at the code for `Q_learn` we originally provided to see how this should be done.

Note on computation time: the training stage can take a relatively long time (sometimes tens of minutes) for the default dimension of the game, `d = 5`. For debugging, you might pass smaller values such as `d = 4` into the `test_solve_play` or `test_learn_play` functions.

Once submitted, the questions below print the decoded information in the pasted-in string. This consists of a list of three elements: boolean (True if TabularQ is used, False if NNQ is used); boolean (True if batch Q_learn_batch is used, False if Q_learn is used); a list of (iteration_number, score) tuples resulting during learning. A successful submission will have a 100.0 score achieved during the learning.

6.1)

Enter the string printed by tabular Q_learn (double check that your game gets to a score of 100):

"286c70300a4930310a614930300a61286c70310a284c304c

Check Formatting Submit View Answer **100.00%**

You have infinitely many submissions remaining.

6.2)

Enter the string printed by tabular Q_learn_batch (double check that your game gets to a score of 100):

"286c70300a4930310a614930310a61286c70310a284c304c

Check Formatting Submit View Answer **100.00%**

You have infinitely many submissions remaining.

6.3)

Enter the string printed by NNQ Q_learn (double check that your game gets to a score of 100):

"286c70300a4930300a614930300a61286c70310a284c304c

Check Formatting Submit View Answer **100.00%**

You have infinitely many submissions remaining.

6.4)

Enter the string printed by Fitted Q_learn (NNQ in Q_learn_batch, and double check that your game gets to a score of 100): "286c70300a4930300a614930310a61286c70310a284c304c

Check Formatting | Submit | View Answer 100.00%

You have infinitely many submissions remaining.

6.5)

Which Q model generally required fewer iterations to solve the game?

[✓ NNQ]
TabularQ

Submit | View Answer 100.00%

You have infinitely many submissions remaining.

6.6)

Which learning algorithm generally required fewer iterations to solve the game?

Q_learn
[✓ Q_learn_batch]

Submit | View Answer 100.00%

You have infinitely many submissions remaining.

Think about why more or fewer iterations are required in these different cases.

Bottom line: Why did we want to use Neural Nets for Q Learning?

6.7) Optional Challenges

There are a lot of possible extensions!

- You can try bigger game instances.
- You can change the ball speed.
- You can change the game behavior.
- You can try different network architectures, e.g. one network with multiple outputs, one for each action.
- You can try to use a CNN using the board directly as an image input. Talk to a staff member for pointers on how to do this.

Survey

(The form below is to help us improve/calibrate for future assignments; submission is encouraged but not required. Thanks!)

How did you feel about the **length** of this homework?

[✓ Too long.]

About right.

Too short.

How did you feel about the **difficulty** of this homework?

Too hard. We should tone it down.

[✓ About right.]

Too easy. I want more challenge.

Do you have any feedback or comments about any questions in this homework? Anything else you want us to know?

Submit

Thanks for your feedback!