ECE433/COS435 Introduction to RL
Assignment 4: Q-learning, DQN, and Approximate Dynamic Programming
Spring 2024

Fill me in

Your name here.

Due March 4, 2024

Collaborators

Fill me in

Please fill in the names and NetIDs of your collaborators in this section.

Instructions

Writeups should be typesetted in Latex and submitted as PDFs. You can work with whatever tool you like for the code, but **please submit the asked-for snippet and answer in the solutions box as part of your writeup. We will only be grading your write-up.** Make sure still also to attach your notebook/code with your submission.

Question 1. Q Learning

Tabular setting

If the state and action spaces are sufficiently small, we can simply maintain a table containing the value of $Q(s,a)$ – an estimate of $Q^*(s,a)$[1] – for every (s,a) pair. In this *tabular setting*, given an experience sample (s,a,r,s'), the update rule is

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left(r + \gamma \max_{a' \in \mathcal{A}} Q(s',a') - Q(s,a) \right) \tag{1}$$

where $\alpha > 0$ is the learning rate and $\gamma \in [0,1]$ is the discount factor.

[1] Here, $Q^*(s,a)$ refers to optimal Q value function.

Question 1.a: Regular Q-Learning

Why is it difficult to extend this learning rule to the game of Tetris or similar Atari games?

Solution

Due to the scale of Atari environments, we cannot reasonably learn and store a Q value for each state-action tuple.

Approximation setting

Here, we instead represent our Q values as a function $\hat{q}(s, a; \mathbf{w})$, where $\mathbf{w}$ are parameters of the function (typically a neural network's weights and bias parameters). In this *approximation setting*, the update rule becomes

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha \left(r + \gamma \max_{a' \in \mathcal{A}} \hat{q}(s', a'; \mathbf{w}) - \hat{q}(s, a; \mathbf{w}) \right) \nabla_{\mathbf{w}} \hat{q}(s, a; \mathbf{w}). \tag{2}$$

In other words, given current parameters at iteration i, w_i, the we aim to minimize the loss at $\mathbf{w}_{i+1}$ which is:

$$L(\mathbf{w}_{i+1}) = \mathbb{E}_{s,a,r,s' \sim \mathcal{D}} \left[\left(r + \gamma \max_{a' \in \mathcal{A}} \hat{q}(s', a'; \mathbf{w}_i) - \hat{q}(s, a; \mathbf{w}_{i+1}) \right)^2 \right] \tag{3}$$

Question 1.b: Action spaces

We can represent a Q-function $Q(s, a)$ as either a function $Q(s; w) : S \to R^A$ outputting the vector of Q-values $[Q(s, a_1), \dots, Q(s, a_{|A|})]$ all at once, or a function $Q(s, a; w) : S \times A \to R$ outputting a single Q-value $Q(s, a)$. What is a benefit of implementing the former over the latter? What is one drawback?

Solution

A benefit of the former representation over the latter is that it is easier to compute the max over actions with just one forward pass. A drawback of the former representation over the latter is that by representing the output as a power of $|A|$, the Q function is hard to scale in a setting with many actions.

Question 1.c: Continuous actions

Consider an environment such as Mountain Car Continuous where the action space is $[-1, 1] \in \mathbb{R}$. How might our representation of the Q-function described in (1.a) change?

Solution

In environments where the action space is discrete, the representation of the Q-function involves taking the maximum over Q-values corresponding to discrete actions. In a continuous action space, you cannot simply enumerate all possible actions and take the maximum Q-value because there are infinitely many possible actions. Hence, instead of maintaining a table of Q-values for each state-action pair, we need to use function approximation to represent the Q-function. A common approach is to use neural networks to approximate the Q-function. These networks can take a state and action as input and output a Q-value, thereby allowing us to work with continuous action spaces.

Question 1.d: Policy iteration

Policy iteration is a model-based (i.e. we have access to the transition probabilities) reinforcement learning algorithm that provably improves a policy. In policy iteration, there is a step called "policy evaluation" that estimates the "value" of a state under the current policy π being learned:

$$V^{\pi}(s) = \sum_{s' \in S} P_{\pi(s)}(s' \mid s)\ [r(s,a,s') + \gamma\ V^{\pi}(s')]$$

The Q-learning update as described in Equation 1, on the other hand, models an entirely different value function. Other than the fact Q-learning learns a Q function and policy evaluation learns a V function, how do these learned value functions differ from each other and why?

Solution

The main difference between the two methods is that policy iteration requires a model to compute the full expected value of a policy, while Q-learning updates its estimates based only on the observed rewards and the maximum value of the next state. Policy iteration systematically evaluates and improves a policy (learning V^{π}), whereas Q-learning directly learns the value of the optimal policy (V^*).
The reason for this difference is due to the max operator used in Q-learning. This max operator is what ensures that Q-learning is learning the optimal value function V^* as it always backs up the value of the best possible future action. This is opposed to policy evaluation in policy iteration, which only considers the expected value under the current policy π without necessarily looking for the maximum value.

Question 1.e: Two Learning Rules for Q-Values

Assuming that you're given a dataset of transitions (s_t, a, r, s_{t+1}) collected from a policy $\beta(a|s)$. Consider learning a Q-function using one of these two learning rules:
Learning Rule 1

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \Big(R_{t+1} + \gamma \max_{a \in \mathcal{A}} Q(s_{t+1}, a) - Q(s_t, a_t) \Big) \tag{4}$$

Learning Rule 2

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha\Big(R_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)\Big) \tag{5}$$

Assume that states and actions are discrete, so the Q-function is just a table. In what settings will these two learning rules converge to the same policy?

Solution

Q-learning will converge to the optimal policy as long as all state-action pairs are visited an infinite number of times and the learning rate α decreases over time according to certain conditions. The optimal policy that Q-learning converges to is one that always selects the action with the highest Q-value in every state.
SARSA will converge to the optimal policy under the same conditions as Q-learning if the policy used to update the Q-values is greedy. However, if the policy used is not greedy then SARSA will converge to a near-optimal policy that is influenced by the level of exploration specified by ϵ.
These converge to similar values when, as described above, SARSA follows a greedy policy and thus effectively becomes Q-learning, as the action taken will always be the one that maximizes the Q-value in the next state (or more specifically SARSA is used with a decay schedule for ϵ that goes to zero, ensuring that the policy becomes greedy in the limit).