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 (30 points)

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 (6 points)

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

Solution

Your solution here...

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, $\mathbf{w}_i$, 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 (6 points)

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 $[Q(s, a_1), \ldots, Q(s, a_{|A|})]$ 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

Your solution here...

Question 1.c: Continuous actions (6 points)

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

Your solution here...

Question 1.d: Policy iteration (6 points)

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. Describe this difference and the reason why.

Solution

Your solution here...

Question 1.e: Two Learning Rules for Q-Values (6 points)

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

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

Learning Rule 2

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

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

Your solution here...

Question 2. Deep Q-Networks (50 points)

You will implement DQN with experience replay and use it to solve the `Cartpole-v0` in OpenAI Gym. In this implementation, there is a target network (frozen for TD updates, used in the update function in $\max_{a \in \mathcal{A}} Q(s_{t+1}, a)$) and a Q-network being trained. Every so often, the target network is updated by copying over the weights from the network being trained, but no gradients updates are made to it.

While there are many (fantastic) implementations of DQN on Github, the goal of this question is for you to implement DQN from scratch *without* looking up code online. Please write your code in the provided notebook.

How to measure if I "solved" the environment?

You should achieve the test reward of around 200 averaged over 20 games (we provide this printing functionality for you) (`Cartpole-v0`).

Runtime Estimation

To help you better manage your schedule, we provide you with reference runtime of DQN on MacBook Pro 2018. For `Cartpole-v0`, it takes 5 minutes to first reach a reward of 200 which is approximately 100 episodes.

Question 2.a (10 points)

First, read through the QNetwork and DQNAgent classes to get a better sense of what is going on. You will implement the neural network in PyTorch used to represent the Q-values. Fill in DQN() including both defining the network and the forward() method.

Hint. While the original DQN paper uses a convolutional architecture, a neural network with 3 fully-connected layers should suffice for the low-dimensional environments that we are working with.

Solution

```
class DQN:
    # FILL ME IN
```

Question 2.b.i (5 points)

You will implement a few functional components of the exploration policy. You will focus on the ϵ-greedy and greedy policies.

Implement the logic for the epsilon greedy policy:

Solution

```
def epsilon_greedy_policy(self, q_values, epsilon):
  # FILL ME IN
```

Question 2.b.ii (5 points)

Implement the logic for the greedy policy:

Solution

```
def greedy_policy(self, q_values):
  # FILL ME IN
```

Question 2.c.i (10 points)

Implement `td_estimate()` and `td_target()`, which are used to compute the loss. These functions represent your model's estimate of its Q-values and the target-estimate using the target network.

Solution

```
 def td_estimate (self, state, action):
    # FILL ME IN

def td_target (self, reward, next_state, done):
    # FILL ME IN
```

Question 2.c.ii (10 points)

Implement the logic for learning the neural network using the error between the TD estimate and TD target. We used `smooth_l1_loss` as our loss function (you can use others if it works).

Solution

```
def train_dqn(self):
   # FILL ME IN
```

Question 2.c.iii (10 points)

Plot the **average cumulative test reward** and **TD error** throughout training and add the image here (no need to add plotting code). Note that in this case we are interested in total reward without discounting or truncation. Does the TD error decrease when the reward increases? Suggest a reason why this may or may not be the case.

Solution

Your plots here...

Question 3 (20 points)

Course Feedback

What would you change so far about the course?

Solution

Your answer here...

Midterm Questions

For this year's midterm, the course staff is making questions, but also allowing students to create (a public bank) of potential questions we may put on the midterm. This will also form a nice study guide for the midterm itself.

Propose 3 simple questions that you would want to see on the midterm. There will be a Google Form to submit your questions, which we will publicly release as a study guide prior to the midterm. **Use the Google Form.**

The spreadsheet with all other student's questions can be found here.