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.**

Question 1. Q-Learning (40 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 (8 points)

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, $\mathbf{w}_i$, 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 (8 points)

We can represent a Q-function $Q(s, a)$ as either a function $Q(s; w) : S \to \mathbb{R}^{|A|}$ 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

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 (8 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 to support our use case (Hint: considering the maximum operation)?

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 (8 points)

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

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

Step "policy improvement" improves the current policy π based on the evaluation. 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 two methods differ from each other and explain in details? (Considering the policy and the environment transition function)

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 (Q^*).
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 Q^* 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 (8 points)

Assuming that you're given a dataset of transitions (s_t, a_t, r_t, 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-learning

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

Learning Rule 2 – SARSA

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

Algorithm 1 Q-learning

Initialize $Q(s, a)$ arbitrarily for all $s \in \mathcal{S}, a \in \mathcal{A}$
for each episode **do**
  Initialize s_t
  while s_t is not terminal **do**
    Choose a_t using ϵ-greedy policy w.r.t. $Q(s, a)$
    Take action a_t, observe r_t, s_{t+1}
    $Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha\big(r_t + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)\big)$
    $s_t \leftarrow s_{t+1}$
  end while
end for

Algorithm 2 SARSA

Initialize $Q(s, a)$ arbitrarily for all $s \in \mathcal{S}, a \in \mathcal{A}$
for each episode **do**
  Initialize s_t
  Choose a_t using ϵ-greedy policy w.r.t. $Q(s, a)$
  while s_t is not terminal **do**
    Take action a_t, observe r_t, s_{t+1}
    Choose a_{t+1} using ϵ-greedy policy w.r.t. $Q(s, a)$
    $Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha\big(r_t + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)\big)$
    $s_t \leftarrow s_{t+1}$, $a_t \leftarrow a_{t+1}$
  end while
end for

The pseudo-code of two algorithms are also displayed above. Assume that states and actions are discrete, so the Q-function is just a table.

Will these two methods converge to the same policy under the same hyperparameters and training dataset? Explain why. What are the conditions for each to converge to optimal 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).

Question 2. Leanring Value Functions (55 points)

We will implement three RL algorithms:

- 1. Q-Learning (Off-policy TD control)
- 2. SARSA (On-policy TD control)
- 3. Monte Carlo (Episodic control)

We will use a simple neural network to approximate $Q(s, a)$ in a discrete-action environment (e.g., "CartPole-v1").

Question 2.a (10 points)

Paste the code block implementing `QNetwork` below.

Solution

```
    ############################
    # YOUR IMPLEMENTATION HERE #

    ############################
class QNetwork(nn.Module):
    def __init__(self, state_dim, action_dim, hidden_dim=64):
        super(QNetwork, self).__init__()
        self.fc1 = nn.Linear(state_dim, hidden_dim)
        self.fc2 = nn.Linear(hidden_dim, hidden_dim)
        self.out = nn.Linear(hidden_dim, action_dim)

    def forward(self, x):
        x = torch.relu(self.fc1(x))
        x = torch.relu(self.fc2(x))
        return self.out(x)

```

Question 2.b (5 points)

Paste the code block implementing `epsilon-greedy policy` below.

Solution

```
    ############################
    # YOUR IMPLEMENTATION HERE #

```

```
    ############################
def epsilon_greedy_policy(q_values, epsilon):
    if np.random.rand() < epsilon:
        return np.random.choice(action_size)
    else:
        # Detach from computation graph and convert to numpy
        return q_values.detach().argmax().item()

```

Question 2.c.1 (15 points)

Paste the code block implementing `q_learning_update` below.

Solution

```
    ############################
    # YOUR IMPLEMENTATION HERE #

    ############################
def q_learning_update(network, optimizer, state, action, reward,
    next_state, done, gamma=0.99):
    # Convert everything to tensors
    state_t = torch.FloatTensor(state).unsqueeze(0)
    next_state_t = torch.FloatTensor(next_state).unsqueeze(0)
    reward_t = torch.FloatTensor([reward])
    done_t = torch.FloatTensor([float(done)])

    # Current Q value
    q_values = network(state_t)
    q_value = q_values[0, action]

    # Target Q value
    with torch.no_grad():
        next_q_values = network(next_state_t)
        next_max_q = next_q_values.max(1)[0]
        target = reward_t + gamma * next_max_q * (1 - done_t)

    # Compute loss and update
    loss = (target - q_value)**2

    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    return loss.item()

```

Question 2.c.2 (15 points)

Paste the code block implementing `sarsa_update` below.

Solution

```
    ############################
    # YOUR IMPLEMENTATION HERE #

    ############################
def sarsa_update(network, optimizer, state, action, reward,
    next_state, next_action, done, gamma=0.99):
    state_t = torch.FloatTensor(state).unsqueeze(0)
    next_state_t = torch.FloatTensor(next_state).unsqueeze(0)

    q_values = network(state_t)
    q_value = q_values[0, action]

    with torch.no_grad():
        next_q_values = network(next_state_t)
        next_q = next_q_values[0, next_action]
        target = reward + (0.0 if done else gamma * next_q)

    loss = (target - q_value)**2

    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    return loss.item()


```

Question 2.c.3 (10 points)

Paste the plotted comparison of learning curves from three methods (Q-learning, SARSA, Monte Carlo regression) below, and analyze the performances of these methods.

Then, justify whether the empirical results here verify the conclusion you made in previous question Q 1.e.

Solution

Your answer here...

Question 3 (5 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.

The midterm problems for last year is here. You can use it as a reference, but please to not copy any question from there as your submitted questions. 2024 Midterm.

Please answer "yes" if have finished submitting the three questions to the google form: # TO FILL (yes or no)