CS 486/686 Assignment 4 Sample Solutions

Blake VanBerlo & Ruixue Zhang

July 31, 2023

1 Dynamic Programming Methods

1. Your Python code (worth 10%).

2. Test your code with the problem described in TestMDP.py

   (a) Report the policy, value function and number of iterations needed by value iteration when using a tolerance of 0.01 and starting from a value function set to 0 for all states (worth 10%).

   Solutions:

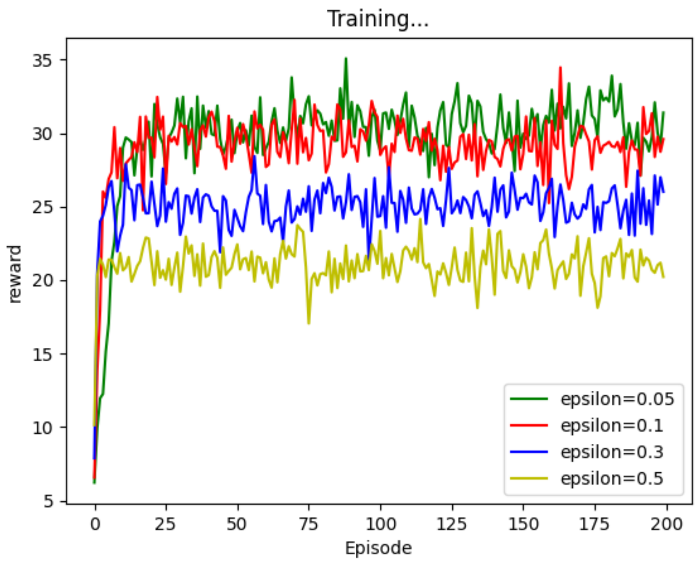
   Policy: [0. 1. 1. 1.]
   Value function:[31.49636306 38.51527513 43.935435 54.1128575]
   Number of iterations: 58

   Based on the way the number of iterations is calculated, the number of iterations may be off by 1 or 2 (still gets full points as long as the code is correct).

   (b) Report the policy, value function and number of iterations needed by policy iteration to find an optimal policy when starting from the policy that chooses action 0 in all states (worth 10%). Solutions:

   Policy: [0. 1. 1. 1.]
   Value function:[31.58448657 38.60345178 44.02359412 54.20105018]
   Number of iterations: 2. Based on the way the number of iterations is calculated, the number of iterations may be off by 1 or 2 (still gets full points as long as the code is correct).

2 Tabular Q-learning



1. Your Python code (worth 10%).

2. Test your code with the problem described in TestMDP.py. Produce a graph where the x-axis indicates the episode (from 0 to 200) and the y-axis indicates the average (based on 100 trials) of the cumulative discounted rewards per episode (100 steps). The graph should contain 4 curves corresponding to the exploration probability epsilon=0.05, 0.1, 0.3 and 0.5. The initial state is 0 and the initial Q-function is 0 for all state-action pairs. Explain the impact of the exploration probability epsilon on the cumulative discounted rewards per episode earned during training as well as the resulting Q-values and policy (worth 20%).

   Solutions:

   Q-learning results epsilon 0.05: [0 1 1 1]
   [[13.19513441 13.00054996 21.50421277 23.24771708]
   [7.47666767 20.64042701 26.54560623 37.12647296]]

   Q-learning results epsilon 0.1: [0 1 1 1]
   [[13.9690188 15.6078714 23.09073631 25.89290393]
   [9.75138335 21.36504289 27.01967352 37.65203499]]

   Q-learning results epsilon 0.3: [0 1 1 1]
   [[14.38268723 17.40593357 24.17175044 27.46974509]
   [10.95506046 21.61117662 27.1363943 37.57357703]]

Q-learning results epsilon 0.5: [0 1 1 1]
[[14.62131448 17.7737263 24.58940632 27.80613733]
[11.35880161 21.79752407 27.27107787 37.64531944]]

The optimal Q-values for this environment are (can be obtained from the optimal values in part 1):
[[31.58205721, 34.74056765, 41.58205721, 44.74056765], [28.42354678, 38.60096928 , 44.02112916, 54.19855166]]

(a) From the plot, a larger epsilon rate has poorer performance since it has a large amount of random action selections.

(b) The higher epsilon has the most correct estimation across the different epsilon values (comparing the Q estimates for each epsilon at the end of training as printed, and the optimal Q values). Since there is more exploration, all the states and actions are visited many times and updated sufficiently many times (while the lower epsilons have not updated the estimates at all state-action pairs as much, leading to still comparatively incorrect estimates).

(c) Alternative to point 2: If they just use the plot to talk about the Q-values using the definition of Q-value, without explicitly printing the Q values, that is correct too.

(d) The policy will still be the same ([0 1 1 1]) across all epsilon values as the Q-learning finds the correct optimal policy. This is a small environment with 4 states and 2 actions and Q-learning only requires few episodes to find the optimal policy irrespective of the value of epsilon.

3 Deep Q-learning

In this part, you will train a deep Q-network to solve the CartPole problem from Open AI Gym. This problem has a large state space that prevents the use of a tabular representation. Instead, you will use a neural network to represent the Q-function. Follow these steps to get started:

1. Get familiar with the CartPole problem. Read a brief description of the CartPole problem from Open AI Gym.

2. For this part of the assignment, you will use a PyTorch implementation of DQN. See the link for an explanation of different parts of the code.

3. Run the code in the file “main.py" to solve the CartPole problem with a Deep Q-Network.

Installs: Instructions available as comments in main.py file.

Submit a report containing the following:

1. Your python code along with the random seeds for all the experiments in this part.

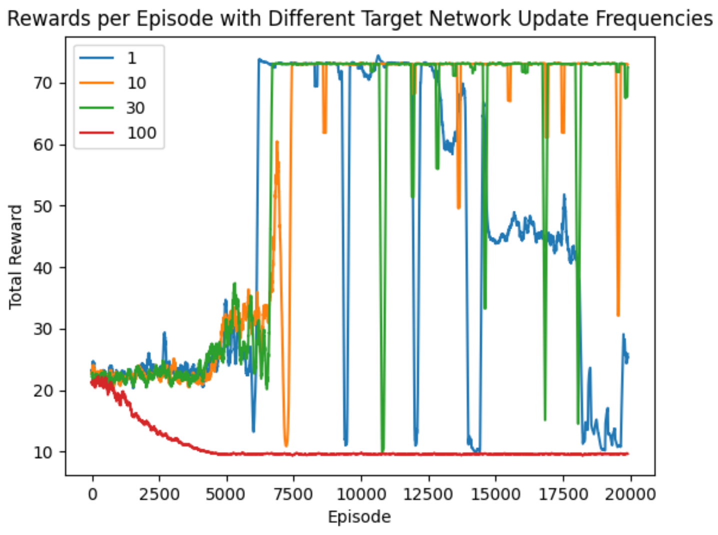
2. Modify and run the code for CartPole DQN to produce a graph where the y-axis is the running average of 100 episodes for the cumulative rewards obtained at each episode and the x-axis is the number of episodes up to a minimum of 20000 episodes. The graph should contain 4 curves corresponding to updating the target network every 1, 10 (default), 30, and 100 training step(s). To reduce stochasticity in the results, report curves that are the average of atleast 3 runs of the given code (with different random seeds). Based on the results, explain the impact of the target network and relate the target network to value iteration (worth 20%).

3. Modify and run the code for CartPole DQN to produce a graph where the y-axis is the running average of 100 episodes for the cumulative rewards obtained at each episode and the x-axis is the number of episodes up to a minimum of 20000 episodes. The graph should contain 4 curves corresponding to sampling mini-batches of 1, 16 (default), 30, and 200 experience(s) from the replay buffer. To reduce stochasticity in the results, report curves that are the average of atleast 3 runs of the given code (with different random seeds). Based on the results, explain the impact of the replay buffer and explain the difference between using the replay buffer and exact gradient descent (worth 20%).

Solutions:

1. Your random seeds. For example: 1423, 1424, & 1425.

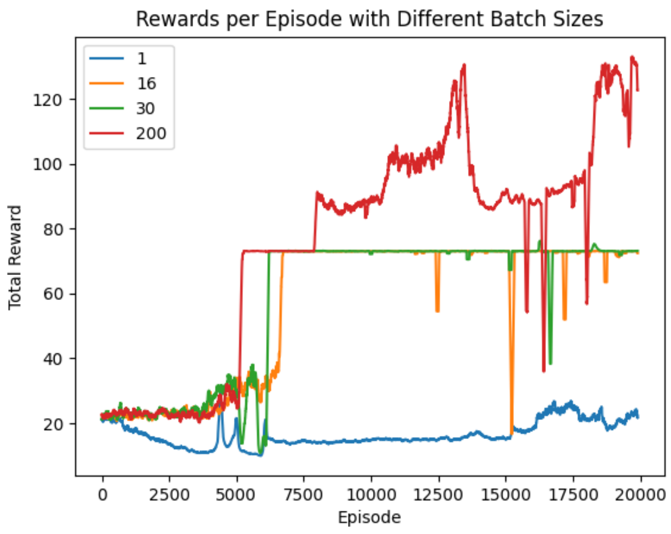
2. Below is the result of training with different target network update frequencies using the above random seeds for 3 trials with each value.



Using the target Q-values predicted by a target network to train the online network improves training stability. In tabular Q-learning, only the Q-value for the state-action pair in the current experience is updated; however, updating the weights of a DQN change Q-value estimates across the domain. Changing the parameters of the target network with every training step destabilizes the objective, initially leading to high-return scenarios and ultimately worsening over time. When the target network update frequency is increased, reward tends to increase over the course of training. Waiting to update the target network is analogous to value iteration in that the target values used to calculate value estimates are not updated with new value estimates until estimates are calculated for each state (i.e., the end of the current iteration). If the target network is updated too infrequently (e.g., every 100 steps), the online network may overfit to the naïve initial Q-value predictions by the target network, resulting in a failure to learn the task in the allotted number of episodes.

Depending on the student's choice of random seeds, their trials may not yield agents that solve the task. In this case, acceptable responses will acknowledge that the stochasticity of exploration and sampling examples from the replay buffer can lead to different results. They should still explain the expected impact of the target network update frequency, on the basis of their results.

3. Below is the result of training with different batch sizes using the above random seeds for 3 trials with each value.



For optimization via gradient descent, training samples are assumed to be independent and identically distributed. In reinforcement learning, the samples are gathered through sequential interaction with the environment. Exact gradient descent using single newly collected experiences would fail because successive experiences are highly correlated. The replay buffer overcomes this issue by providing a large collection of past experiences to randomly sample at each training step. In exact gradient descent, the error for a single experience at a time is used to update the network's parameters. As observed in the results above, exact gradient descent with a batch size of 1 can lead to poor performance because the network's weights are updated to improve performance on a single experience, at the potential expense of other areas of the domain. Increasing the batch size corrects such unintentional errors, encouraging the updates to maintain performance across a larger share of the problem domain. If the batch size is too large, learning may be slow, resulting in an inability to solve the task in the allotted number of episodes.

Again, depending on the student's choice of random seeds, their trials may not yield agents that solve the task. In this case, acceptable responses will acknowledge that the stochasticity of exploration and sampling examples from the replay buffer can lead to different results. They should still explain the beneficial role of the replay buffer and expected impact of changing the batch size, based on their results.