CMPSCI 687 Homework 2 - Fall 2022

Due **October 16, 2022**, 11:55pm Eastern Time

Note: we have fixed a minor typo in one of the equations in Question 3. Here, the definition of J is identical to the one presented in class/slides/lecture notes. All parts of this document that were changed with respect to the original one are shown in blue.

1 Instructions

This homework assignment consists of a written portion and a programming portion. While you may discuss problems with your peers (e.g., to discuss high-level approaches), you must answer the questions on your own. In your submission, do explicitly list all students with whom you discussed this assignment. Submissions must be typed (handwritten and scanned submissions will not be accepted). You must use LaTeX. The assignment should be submitted on Gradescope as a PDF with marked answers via the Gradescope interface. The source code should be submitted via the Gradescope programming assignment as a .zip file. Include with your source code instructions for how to run your code. You **must** use Python 3 for your homework code. You may not use any reinforcement learning or machine learning-specific libraries in your code, e.g., TensorFlow, PyTorch, or scikit-learn. You *may* use libraries like numpy and matplotlib, though. The automated system will not accept assignments after 11:55pm on October 16. The tex file for this homework can be found here.

Before starting this homework, please review this course's policies on plagiarism by reading Section 10 of the syllabus.

Part One: Written (40 Points Total)

1. (**14 Points**) Recall that the action-value function of a policy π is defined as

$$q^\pi(s,a) := \mathbb{E}\big[G_t | S_t = s, A_t = a; \pi\big]. \tag{1}$$

Start from this equation and use the definitions and properties of probability distributions discussed in Homework 1, as well as the Markov Property (when appropriate), to show from "first principles" the complete derivation proving that the following is a Bellman Equation for q^π:

$$q^\pi(s,a) = R(s,a) + \gamma \sum_{s'} p(s,a,s') \sum_{a'} \pi(s',a') q^\pi(s',a'). \tag{2}$$

Hint: review the strategy we used in class to prove, from first principles, one of the Bellman equations for v^π.

2. (**6 Points**) Recall that the value function of a policy π is defined as

$$v^\pi(s) := \mathbb{E}\big[G_t | S_t = s; \pi\big]. \tag{3}$$

Use the different variants of the Bellman equation studied in class to prove that the following is also a Bellman equation for v^π:

$$v^\pi(s) = \sum_a \pi(s,a)\, q^\pi(s,a). \tag{4}$$

3. (**12 Points**) In class, we presented two ways of defining what an optimal policy, π^*, is:

- **Definition 1.**
  π^* is an optimal policy iff $\pi^* \geq \pi$, for all $\pi \in \Pi$. This holds iff $v^{\pi^*}(s) \geq v^\pi(s)$, for all $\pi \in \Pi$ and for all $s \in \mathcal{S}$.
- **Definition 2.**
  π^* is an optimal policy iff $\pi^* \in \arg\max J(\pi)$, where $J(\pi) = \mathbb{E}[\sum_{k=0}^{\infty} \gamma^k R_k \,|\, \pi]$. ~~$J(\pi) = \mathbb{E}[\sum_{k=0}^{\infty} \gamma^k R_{t+k} | \pi]$.~~

(Question 3a. 5 Points) Let π_1^* be an optimal policy with respect to Definition 1. Let π_2^* be an optimal policy with respect to Definition 2. Is it true that $J(\pi_1^*) = J(\pi_2^*)$? If so, prove it. If not, show a counter-example.

(Question 3b. 7 Points) In class, we proved that if two policies, π_A and π_B, are optimal with respect to Definition 1, then they share the same value function: $v^{\pi_A}(s) = v^{\pi_B}(s) = v^*(s)$, for all $s \in \mathcal{S}$. Along similar lines, consider π_1^* and π_2^* as defined in Question 3a: π_1^* is optimal with respect to Definition 1, and π_2^* is optimal with respect to Definition 2. Is it true that $v^{\pi_1^*}(s) = v^{\pi_2^*}(s)$, for all $s \in \mathcal{S}$? If so, prove it. If not, show a counter-example.

4. (**8 Points**) In class, we studied different ways in which we may want to quantify the expected return of a policy. In some cases, we are interested in the expected return if the agent starts in state s and follows a given policy π; that is, $v^\pi(s)$. In other cases, we are interested in the expected return if the agent starts in state s, executes action a, and then follows a given policy π; that is, $q^\pi(s, a)$. Consider yet another possible way in which we may want to quantify expected return: via a function $w^\pi(s, a, s')$ that represents the expected return if the agent starts in state s, executes action a, transitions to a particular next state s', and then follows policy π.

   (Question 4a. 3 Points) Show the formal definition of w^π in terms of an expectation, similarly to how v^π and q^π were defined.

   (Question 4b. 5 Points) Use the definitions and properties of probability distributions discussed in Homework 1, as well as the Markov Property (when appropriate), to show from "first principles" how v^π can be written/expressed in terms of w^π. Your expression for v^π in terms of w^π should only use w^π, $\mathcal{S}, \mathcal{A}, p, R, d_0, \gamma$, and π.

Part Two: Programming (60 Points Total)

Implement the Mountain Car domain (discussed in class). You will find, below, a complete description of this domain:

- **State**: $s = (x, v)$, where $x \in \mathbb{R}$ is the position of the car and $v \in \mathbb{R}$ is the velocity.
- **Actions**: $a \in \{\texttt{reverse}, \texttt{neutral}, \texttt{forward}\}$. These actions are mapped to numerical values as follows: $a \in \{-1, 0, 1\}$.
- **Dynamics**: The dynamics are *deterministic*—taking action a in state s always produces the same next state, s'. Thus, $p(s, a, s') \in \{0, 1\}$. The dynamics are characterized by:

$$\begin{aligned} v_{t+1} &= v_t + 0.001 a_t - 0.0025 \cos(3 x_t) \\ x_{t+1} &= x_t + v_{t+1}. \end{aligned}$$

  After the next state, $s' = [x_{t+1}, v_{t+1}]$ has been computed, the value of x_{t+1} is clipped so that it stays in the closed interval $[-1.2, 0.5]$. Similarly, the value v_{t+1} is clipped so that it stays in the closed interval $[-0.07, 0.07]$. If x_{t+1} reaches the left bound (i.e., the car is at $x_{t+1} = -1.2$), or if x_{t+1} reaches the right bound (i.e., the car is at $x_{t+1} = 0.5$), then the car's velocity is reset to zero: $v_{t+1} = 0$. This simulates inelastic collisions with walls at -1.2 and 0.5.
- **Terminal States**: If $x_t = 0.5$, then the state is terminal (it always transitions to s_∞). The episode may also terminate due to a timeout; in particular, it terminates if the agent runs for more than 1000 time steps.
- **Rewards**: $R_t = -1$ always, except when transitioning to s_∞ (from s_∞ or from a terminal state), in which case $R_t = 0$.
- **Discount**: $\gamma = 1.0$.
- **Initial State**: $S_0 = (X_0, 0)$, where X_0 is an initial position drawn uniformly at random from the interval $[-0.6, -0.4]$.

Next, you will implement a Black-Box Optimization (BBO) algorithm to search for optimal policies to control the car. In particular, you will be implementing the Cross-Entropy algorithm, which is described below. **Notice that you may not use existing RL code for this problem: you must implement the agent and environment entirely on your own and from scratch.**

1.1 Cross-Entropy Method

The *Cross-Entropy* (CE) method for policy search is a simple BBO algorithm that has achieved remarkable performance on domains like playing Tetris. We present, below, a variant of CE based on the work of Stulp and Sigaud (2012). **You should read Section 2.1 of their paper before implementing the CE method, to understand how (and why) it works and how to set its hyperparameters.**

Intuitively, CE starts with a multivariate Gaussian distribution over policy parameter vectors. This distribution has mean θ and covariance matrix Σ. The method samples K policy parameter vectors from this distribution. Let $\theta_1, \ldots, \theta_K$ denote these samples. CE evaluates these K sampled policies by running each one for N episodes and averaging the resulting returns. It then picks the K_e best performing policy parameter vectors (for some user-defined, constant K_e) and fits a multivariate Gaussian to these parameter vectors. The mean and covariance matrix for this fit are stored in θ and Σ and this process is repeated. We present pseudocode for CE in Algorithm 1, which uses the `estimate_J` function defined in Algorithm 2. Notice that `estimate_J` is very similar to the function you implemented in Homework 1 to estimate the performance, $\hat{J}$, of a given policy by averaging different sampled returns.

Algorithm 1: `Cross-Entropy` (CE) for Policy Search.
Input:
1) Initial mean policy parameter vector, $\theta \in \mathbb{R}^n$
2) "Initial exploration" parameter, $\sigma \in \mathbb{R}$
3) Population, $K \in \mathbb{N}_{>1}$ [for example, $K = 20$]
4) Elite population, $K_e \in \mathbb{N}_{>0}$, where $K_e < K$ [for example, $K_e = 10$]
5) Number of episodes to sample per policy, $N \in \mathbb{N}_{>0}$ [for example, $N = 10$]
6) Small numerical stability parameter $\epsilon \in \mathbb{R}$ [for example, $\epsilon = 0.0001$]

Let $\Sigma = \sigma I$ be the initial covariance matrix, where I is the $n \times n$ identity matrix;
while `true` **do**
  for $k = 1$ **to** K **do**
    $\theta_k \sim N(\theta, \Sigma)$;
    $\hat{J}_k = \texttt{estimate\_J}(\theta_k, N)$;
  $\texttt{sort}((\theta_1, \hat{J}_1), (\theta_2, \hat{J}_2), \ldots, (\theta_K, \hat{J}_K), \text{descending order})$;
  $\theta = \frac{1}{K_e} \sum_{k=1}^{K_e} \theta_k$;
  $\Sigma = \frac{1}{\epsilon + K_e} \left(\epsilon I + \sum_{k=1}^{K_e} (\theta_k - \theta)(\theta_k - \theta)^\intercal \right)$;

Algorithm 2: `estimate_J`
Input:
1) Policy parameter vector, $\theta \in \mathbb{R}^n$
2) Number of episodes to sample, $N \in \mathbb{N}_{>0}$ [for example, $N = 10$]

Run the parameterized policy using policy parameters θ for N episodes;
Compute the resulting N returns, $G^1, G^2, \ldots, G^N$, where $G^i = \sum_{t=0}^{\infty} \gamma^t R_t^i$;
Return $\frac{1}{N} \sum_{i=1}^{N} G^i$;

1.2 Policy Representation

To solve the Mountain Car problem, we need to select a policy representation that works with continuous state spaces and discrete actions. In order to allow the following experiments to run faster, we will be using a *deterministic* policy representation that picks actions according to the following overall strategy:

1. We assume that a policy is represented by a vector of weights, $\theta \in \mathbb{R}^n$.
2. When in state s, the agent computes a vector of *state features*, $\phi(s) := [\phi_1(s), \ldots, \phi_n(s)]$, where each ϕ_i is a domain-dependent state feature function. Below, we will describe the particular state feature functions that you should use in this experiment—the Fourier basis features.

3. A *threshold* value is then computed as the dot product between $\phi(s)$ and θ. In other words: the threshold value corresponds to the weighted average of state features, where the weights are given by the corresponding policy parameters.

4. If the threshold value is less than or equal to zero, the policy commands the car to drive left (reverse).

5. If the threshold value is larger than zero, the policy commands the car to drive right (forward).

6. Notice that using this particular policy representation simplifies the problem since the action Neutral is never used/considered by the policy.

The procedure that implements this type of deterministic policy is presented in Algorithm 3.

1.2.1 State Features: the Fourier Basis

The Fourier basis, introduced by Konidaris et al. (2011), is a simple and principled way of constructing state features when learning value functions for problems with continuous state spaces. It has performed well over a wide range of problems, despite its simplicity. For a high-level introduction to the Fourier basis, please see `http://irl.cs.brown.edu/fb.php`.

In this homework, however, we will not be using the Fourier basis to learn value functions. Instead, we will be using them to represent parameterized policies over continuous states—in particular, to define each state feature function, ϕ_i, used by the policy representation described above. Let M be the *order* of the Fourier basis we wish to use. In this homework, you may choose to define the state feature vector associated with a given state s, in the Mountain car domain, in two different ways:

1. $\phi(s) = [1, \cos(1\pi x), \cos(2\pi x), \ldots, \cos(M\pi x), \cos(1\pi v), \cos(2\pi v), \ldots, \cos(M\pi v)]^\top$.
2. $\phi(s) = [1, \sin(1\pi x), \sin(2\pi x), \ldots, \sin(M\pi x), \sin(1\pi v), \sin(2\pi v), \ldots, \sin(M\pi v)]^\top$.

Important: prior to computing $\phi(s)$, you should normalize each component of the state (i.e., x and v). If you choose to use the feature representation based on *cosines*, we suggest that you normalize each component of the state so that it is in the interval $[0, 1]$. Alternatively, if you choose to use the feature representation based on *sines*, we suggest that you normalize each component of the state so that it is in the interval $[-1, 1]$. To normalize the state features, remember that car positions, x, are originally in the interval $[-1.2, 0.5]$, and that car velocities, v, are originally in the interval $[-0.07, 0.07]$. You can/should experiment with these two different types of state features to identify which one works best.

The procedure that implements the deterministic policy described in this section is presented in Algorithm 3.

Algorithm 3: Deterministic policy, π, based on the Fourier basis, for use in the Mountain Car domain.
Input:
1) Policy parameter vector, $\theta \in \mathbb{R}^n$
2) Current state, $s = [x, v]$
3) The order, M, of the Fourier Basis features

```
Normalize each component of the state (i.e., x and v) to the appropriate interval;
Compute φ(s) according to one of the two definitions presented in Section 1.2.1;
threshold = φ(s)^⊤ θ;
if threshold ≤ 0 then
    Return a = −1;
else
    Return a = +1;
```

2. Questions (Programming)

Note: there exists a near-optimal policy for Mountain Car that "solves" the problem in approximately 60 steps. In the questions below you should try to get your algorithm's performance as close as possible to this goal. Notice also that, given the particular deterministic policy representation we choose to use, you may not be able to replicate this performance exactly.

1. (**28 Points**) You will first use the CE algorithm to learn efficient policies to control the car in the Mountain Car domain. Doing so requires setting many hyperparameters of the algorithm: N, K, K_e, σ, and M. Search the space of hyperparameters for hyperparameters that work well. Choosing the range from which to sample/select hyperparameters is challenging. As an example, one could heuristically guess that K is some value in $[20, 300]$; that K_e might be in $[3, 50]$; σ may be in $[5, 200]$; and M might be in $[2, 10]$. Notice that these ranges/intervals are just *examples*, and we are *not* suggesting they are necessarily the best ones to use. You are encouraged to test other types of values as well. You should also think carefully about how to tune N, by considering the stochasticity in the domain and the policy.

   Report how you searched the hyperparameters, what hyperparameters you found that worked best **(including which state feature representation you chose to use)**. Show a learning curve plot (i.e., return as a function of the number of iterations/updates performed by CE) for a few selected hyperparameters that you found to be relevant (or surprising) in terms of how they affected your algorithm. Each learning curve plot should present average performance results computed over 5 trials/runs (see *Question 2*, below, for details on how to do this).[1] You should report results for at least 5 settings of hyperparameters and discuss why you think the algorithm behaved like it did in each corresponding case.

2. (**12 Points**) Consider the hyperparameters you identified, in the previous question, as hyparameters that work well. Present their values **(and indicate which state feature representation you chose to use)** and show a learning curve plot of your algorithm when using these hyperparameters and when evaluated over 20 trials/runs.[2] In particular, you will run your algorithm 20 times and plot the mean/average return (computed over the 20 trials/runs) after 1 iteration of CE, after 2 iterations of CE, and so on. When creating this graph, also show the standard deviation of the return for each of these points.

3. (**8 Points**) Did your algorithm reach near-optimal performance/return? If not, why do you think that is the case? In general, reflect on this problem: was it easier or harder than you expected to get the method working? What were the main challenges?

4. (**12 Points**) The type of analysis described above, where we check how different hyperparameters affect learning curves, can be used to fine-tune your algorithm to optimize different performance criteria. You could, for example, be interested in finding hyperparameters that *(a)* maximize convergence speed; or that *(b)* make the algorithm's performance more stable over time (i.e., keep the mean return from fluctuating wildly as a function of the number of iterations/updates performed by CE); or *(c)* allow the algorithm to find better policies, though possibly causing its runtime to increase significantly.

   Discuss your findings regarding how different types of hyperparameters affect each of these three performance criteria. For instance, are there different sets of hyperparameters that cause the algorithm to converge to the same mean return, but such that one causes returns to fluctuate more strongly over time, and the other causes returns to fluctuate less strongly over time? Are there hyperparameters that cause the algorithm to converge faster, though possibly to a worse solution? Can you find any patterns in how different values for each of the hyperparameters affect these different performance criteria?

References

Freek Stulp and Olivier Sigaud. Path integral policy improvement with covariance matrix adaptation. *CoRR*, abs/1206.4621, 2012.

G. D. Konidaris, S. Osentoski, and P. S. Thomas. Value function approximation in reinforcement learning using the Fourier basis. In *Proceedings of the Twenty-Fifth Conference on Artificial Intelligence*, pages 380–395, 2011.

[1] Evaluating hyperparameters over just 5 trials/runs is, in general, certainly not sufficient. You are allowed to do that in this question to lower the computation time required to perform these experiments and analyses. You will evaluate your selected solution/policy more accurately in a subsequent question.

[2] In real-life applications/research, we need to average results over many more runs. Using 20 runs here is acceptable since there is almost no stochasticity in the MDP nor the policy, and CE's performance (in this particular domain) is not highly dependent on its initial conditions.