CMPSCI 390A Homework 5

Assigned: April 12 2022; Due: April 22 2022 @ 11:59pm

Abstract

This assignment will cover the basics of solving a reinforcement learning problem using a simple algorithm. To submit this assignment, upload two files to Gradescope. **File 1:** A `.pdf` containing your responses to the written response questions below. **File 2:** a file named `my_hw5.py` with your solution to the programming portion. Failure to name the file `my_hw5.py` will will result in subtraction of 5 points from the grade for this assignment. An auto-grader will check your code for the correct output. As such, your program must meet the requirements specified below. We will also be using cheating detection software, so, as a reminder, you are allowed to discuss the homework with other students, but you must write the code on your own.

1 Written Questions (20 points)

In this part of the assignment, you will be asked to answer questions that will help with the assignment's coding portion. In this homework, you will be implementing the Simple RL algorithm covered in recent lectures. The Simple RL algorithm covered in lectures requires computing two quantities: $\frac{\partial}{\partial \theta_i^a} \pi_\theta(s, a)$ and the sum of discounted rewards. The policy representation used in this assignment will be a linear softmax with a Fourier basis. Recall that a linear softmax policy specifies the probability of action a in state s as

$$\pi_\theta(s, a) = \frac{e^{\sum_j \theta_j^a \phi_j(s)}}{\sum_{a'} e^{\sum_j \theta_j^{a'} \phi_j(s)}},$$

where θ_j^a is the jth weight in the vector for action a, and $\phi_j(s)$ is the jth feature produced by the basis function ϕ. For the coding portion ϕ will be the Fourier basis and you will be given the code for it. The policy can be broken down into two steps. The first step is to compute output values for each action, i.e., $o_a = \sum_j \theta_j^a \phi_j(s)$. This is equivalent to the linear function approximation we did in the regression problems, e.g., $o_a = f_{\theta^a}(s)$. The second step is to perform the softmax operation on each action output o_a, i.e.,

$$\pi_\theta(s, a) = \frac{e^{o_a}}{\sum_{a'} e^{o_{a'}}}.$$

To answer the questions below, provide an expression for the quantity specified. For derivatives, your answer should not contain any derivative symbols but can contain computed quantities such as $\pi_\theta(s, a)$ and $\phi_j(s)$.

1. (5 points) Compute the partial derivative for the action output o_a when a is the chosen action.

$$\frac{\partial}{\partial o_a} \pi_\theta(s, a) = \qquad \text{Text}$$

2. (5 points) Compute the partial derivative for the action output o_a when another action b is chosen, i.e., $b \neq a$.

$$\frac{\partial}{\partial o_a} \pi_\theta(s, b) =$$

3. (2 points) Compute partial derivative for the weights of the chosen action.

$$\frac{\partial}{\partial \theta_k^a} \pi_\theta(s, a) =$$

4. (1 points) Compute the partial derivative for the weights of an action a when another action b was chosen, i.e., $b \neq a$.

$$\frac{\partial}{\partial \theta_k^a} \pi_\theta(s, b) =$$

5. (5 points) Compute the partial derivative of the natural log probability of the chosen action. Side note: this is the derivative used in most policy gradient algorithms.

$$\frac{\partial}{\partial \theta_k^a} \ln \pi_\theta(s, a) =$$

6. (2 points) Let $G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k}$. Write a recursive expression for G_t using R_t, γ, and G_{t+1}. Side note: This makes it easy to compute G_t starting at the last reward in an episode.

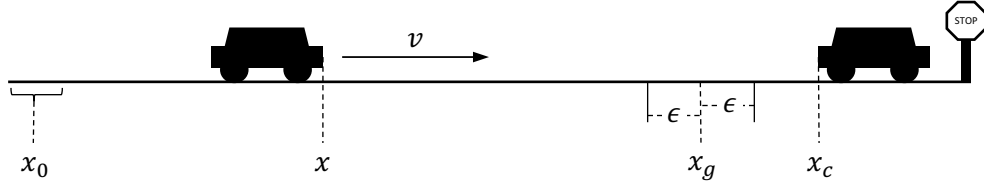$$G_t =$$

2   The Stopped Car Problem

Figure 1: Depiction of the Stopped Car problem.

In this assignment, you will be tasked with optimizing a policy to solve a reinforcement learning task. This assignment's task will be the *StoppedCar* problem, which is illustrated in Figure 1. We have implemented this environment for you, so you only need to understand the environment to interpret the policy's performance.

In the StoppedCar problem, there is a car at position x moving a constant velocity, v, towards another car at position x_c that is stopped at a stop sign. The agent's job is to learn to apply the breaks, so the car stops within a goal region centered at x_g. The car is equipped with a distance sensor to measure the distance to the back of the stopped car, i.e., $x_c - x$. The state of the environment at time t is represented by the distance to the stopped car $x_c - x_t$ and the velocity of the car v_t at time t, i.e., $s_t = [x_c - x_t, v_t]$.

The position of the car always transitions based on the current velocity, i.e., $x_{t+1} = x_t + \Delta t v_t$, where Δt is the amount of simulation time between time steps t and $t + 1$. There are two actions available to the agent, *do-nothing* and *apply-breaks*. When the do-nothing action is taken the car continues moving at the current velocity, i.e., $v_{t+1} = v_t$. When the apply-breaks action is taken, the velocity is decreased by a constant factor κ of the current velocity, i.e., $v_t = v_t - \Delta t \kappa v_t$. If the velocity becomes smaller than a constant c, then the velocity is set to zero. The episode terminates when the velocity reaches zero or a collision with the stopped car occurs, i.e, $x_{t+1} \geq x_c$. The agent receives a reward of 0 at all time steps except at the last time step. On the last time step, the agent will receive a reward of $+10$, -10, or -1 depending on if it is stopped in the goal region, collided with the car, or stopped anywhere else.

3   Implementing a Simple RL Algorithm (50 points)

For this assignment, you will implement the Simple RL algorithm and solve the stopped car problem using the code provided in the template file `my_hw6.py`. We have provided complete code for the environment (`stoppedcar.py`), the Fourier basis (`fourierbasis.py`), and sampling an episode from the environment. You are responsible for finishing the implementation of the class `LinearSoftmax` and completing functions for the Simple RL algorithm. The requirements for each of these are outlined below.

Implement the following functions:

- `LinearSoftmax.get_action_probabilities(self, x)`. This function computes each action's probability given a vector x, where x_j is the output of the basis function $\phi_j(s)$. This function should compute the following for each action:

$$\pi_\theta(s, a) = \frac{e^{\sum_j \theta_j^a x_j}}{\sum_{a'} e^{\sum_j \theta_j^{a'} x_j}}.$$

- `LinearSoftmax.gradient_logprob(self, state, action)`. This function computes and returns two quantities: $\frac{\partial}{\partial \theta} \ln \pi_\theta(s, a)$ and $\ln \pi_\theta(s, a)$, where s and a are the variables `state` and `action`. This function should return a tuple of both quantities with the derivative terms first. This also requires compute the basis functions for the state input. See the code in `LinearSoftmax.get_action` for an example on how to do this.

- `LinearSoftmax.gradient_prob(self, state, action)`. This function computes and returns the quantity $\frac{\partial}{\partial \theta}\pi_\theta(s, a)$.

- `compute_sum_of_discounted_rewards(rewards, gamma)`. This function computes and returns the list of all discounted sum of rewards for one episode, i.e., return the list:

$$[G_0, G_1, \ldots, G_{T-1}],$$

where $T - 1$ is the last step of the episode.

- `update_policy(policy, states, actions, rewards, alpha, gamma)`. This function performs the policy update specified by the MENACE algorithm Version 4.0, e.g., Algorithm 15.1 in the course notes. The algorithm is shown below and the `update_policy` function performs the computation of the last for loop.

Algorithm 1: A simple RL algorithm inspired by MENACE, Version 4.0

---

for *each episode* **do**
    `// We provide code for this for loop.`
    `// Run one episode (play one game).`
    for *each time t in the episode* **do**
        Agent observes state S_t;
        Agent selects action A_t according to the current policy, π_θ;
        Environment responds by transitioning from state S_t to state S_{t+1} and emitting reward R_t;
    end
    `// You are implementing this update.`
    `// Learn from the outcome of the episode.`
    for *each time t in the episode* **do**
        $\forall i, \theta_i \leftarrow \theta_i + \alpha \left(\sum_{k=0}^{\infty} \gamma^k R_{t+k}\right) \frac{\partial \pi_\theta(S_t, A_t)}{\partial \theta_i}$;
    end
end

Import details on how to correctly implement these functions are specified in the template file's comments. For your convince, the code contains *TODO* comments to mark where you need to implement these functions along with other assignment-related aspects. However, you should read the entire file to understand the greater scope of the function calls. An autograder will check these functions, and you will be able to see the outputs of the autograder every time you upload your file.

4 Running the Algorithm and Optimizing Hyperparameters (30 points)

With the above functions correctly implemented, the next step is to tune the algorithm's hyperparameters. This algorithm has three hyperparameters: the step size α, discount factor γ, and Fourier basis order. These hyperparameters need to be tuned to find a setting that makes the algorithm reliably achieve good performance.

The performance will be measured by the sum of rewards without discounting (i.e., $\gamma = 1$). The algorithm will be run for 100 trials (lifetimes), each containing 400 episodes. The algorithm's average performance will be represented by a learning curve plot as shown in Figure 2. This plot is automatically generated and saved when the function `main` is run. In addition, to the written questions above, your written response should contain the learning curve plot and the hyperparameters you found.

5 Hints and Tips

Quickly Optimizing Hyperparameters: When tuning hyperparameters, it will take a long time to determine if a particular hyperparameter worked well if you run all the lifetimes and episodes. A faster way is to set the number of lifetimes to one and increase the number as you find good values for the hyperparameters. You can also set the number of episodes lower, but by doing so you will not be able to tell how the algorithm will perform when it is run for longer episodes. Make sure to set both of these to 100 lifetimes and 400 episodes when you perform your final run.

Scaling Hyperparameters: Tuning multiple hyperparameters can be tricky since changing one usually affects how good the value is for another. A good example of this is changing the size of the basis function and the step size. Typically, as the number of basis functions grows (order increases), the step size needs to be decreased. A common heuristic for scaling the step size to account for the size of the basis function is to scale it by the number of basis functions, e.g., $\alpha = \alpha_0/m$, where
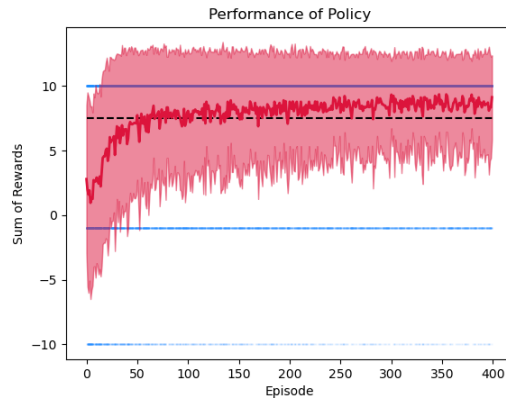
Figure 2: This plot represents the acceptable performance of the algorithm for this assignment. The red line represents the mean performance for each episode averaged over 100 lifetimes, and the red shaded area is the standard deviation. The black dashed line represents the minimum performance the algorithm must achieve on average. The blue dots are all of the sums of rewards from each episode and lifetime. Each dot has the α value (translucency not step size) set to 0.01, so as the dot becomes more or less opaque, it means that the algorithm optimized the policy so that that level of performance is more or less frequent. The change in frequency is easy to see on the best and worst performance levels, where the top performance becomes frequent and the bottom rare.

α_0 is the step size you specify, and m is the number of basis functions. The `getNumFeatures()` function of the FourierBasis class will return the number of basis functions used.

Easy and Efficient Python Coding: As many have of you might have noticed, Python code can run very slow. This slowness is because Python cannot perform code optimization when each command gets compiled to machine code. Python libraries like NumPy write much of their code in C/C++/Fortran, and the Python components act as a convenient interface to this code. In doing so, these libraries can have enormous speeds over a similar implementation written directly in pure Python. We outline a couple of useful NumPy operations that make code both simpler (fewer lines) and faster.

```python
x = np.array(range(100))
y = np.random.rand(100)

# scalar times each value in the array
# Pure Python:
z = np.zeros_like(x)
s = 20   # an arbitrary value
for i in range(x.shape[0]):
    z[i] = s * x[i]

# NumPy way (called broadcasting)
z = s * x   # super simple and equivalent to the above for loop
# this also works with +,-,/ and some other operations.
# in-place operations also work
x *= s   # same as x = s * x

# add the value of each element of x and y together
# Pure Python:
z = np.zeros_like(x)
for i in range(x.shape[0]):
    z[i] = x[i] + y[i]

# NumPy way
z = x + y
# also works with -,*,/ and in-place operations.


# sum of the products of each element in x and y (called a dot product)
# Pure Python:
```

```python
z = 0.0
for i in range(x.shape[0]):
    z += x[i] * y[i]

# NumPy way
z = np.dot(x,y)

# The function np.dot is a powerful function for NumPy and can handle multidimensional arrays, but you need
# to be careful with the shape of each array. See the NumPy documentation for more information.
```