CS6700 : Reinforcement Learning
Programming Assignment #2

Control Algorithms — Deadline: 7th March 2020, 11:55 pm

- This is an individual assignment. Collaborations and discussions are strictly prohibited. *Do **not** put up the code for any part of the assignment in public repositories online.*
- You have to turn in the well-documented code along with a detailed report of the results of the experiments. Typeset your report in LaTeX.
- Be precise with your explanations. Unnecessary verbosity will be penalized.
- Check the Moodle discussion forums regularly for updates regarding the assignment.
- *Any kind* of plagiarism will be dealt with extremely seriously. Acknowledge any and every resource used.
- **Please start early.**

The aim of the exercise is to familiarize you with various learning control algorithms. The goal is to solve variants of the given problems.

We will be using python and OpenAI Gym to solve the problems in this assignment. The above link leads to the page on 'Getting started with Gym'. Please install OpenAI Gym before starting.

1 Q-learning and SARSA

Environment details

1. This is a typical grid world, with 4 stochastic actions. The actions might result in movement in a direction other than the one intended with a probability of 0.1. For example, if the selected action is N (north), it will transition to the cell one above your current position with probability 0.9. It will transition to one of the other neighbouring cells with probability 0.1/3.
2. Transitions that take you off the grid will not result in any change.
3. There is also a gentle Westerly blowing, that will push you one additional cell to the east, regardless of the effect of the action you took, with a probability of 0.5 [1].
4. The episodes start in one the start states in the first column, with equal probability.
5. There are three variants of the problem, A, B, and C, in each of which the goal is in the square marked with the respective alphabet. There is a reward of +10 on reaching the goal.

[1] You might want to turn this off for Goal C

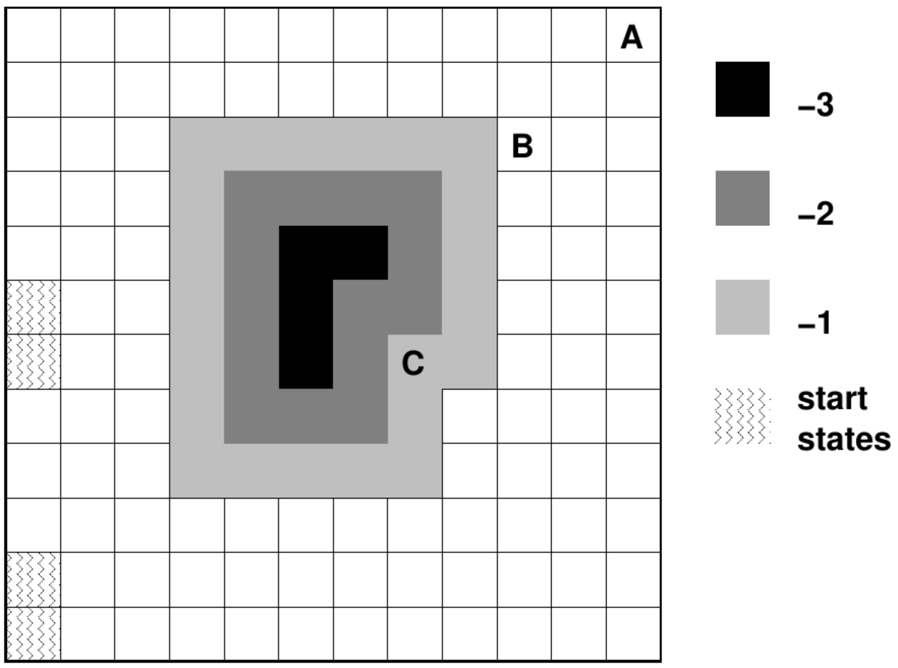

Figure 1: The puddle-world.

6. There is a puddle in the middle of the gridworld which the agent would like to avoid. Every transition into a puddle cell gives a negative reward depending on the depth of the puddle at that point, as indicated in the figure.

Questions

1. Create this gridworld in OpenAI-gym (documentation and code) You can refer to some more instructions in Part 1 of the next question for coding up new environments. We highly recommend getting used to Gym since it is offers a rich suite of flexible, open-source environments that you might be using for any potential future research.

Q-learning

2. Implement *Q-learning* to solve each of the three variants of the problem. For each variant run experiments with a gamma value of 0.9. Compute the averages over 50 independent runs. The deliverables are the following:

   (a) Turn in two learning curves for each experiment, one that shows how the average number of steps to goal changes over learning trials and the other that shows how the average reward per episode changes.

   (b) Indicate the optimal policies arrived at in each of the experiments.

Sarsa

3. Repeat with *Sarsa*. The deliverables remain the same.

4. Now solve this with *Sarsa*(λ), for values of $\lambda = \{0, 0.3, 0.5, 0.9, 0.99, 1.0\}$. Apart from the plots mentioned above, also turn in a plot/graph that shows the average performance of the algorithms (in terms of average steps and reward obtained), for the various values of λ after 25 learning trials.

2 Policy Gradients

In this question, we would learn Policy Gradients to perform continuous control. By continuous control we mean both the state and the action spaces are continuous. The question has three main parts :

- Part 1: The environment implementation
- Part 2: The rollout function
- Part 3: The policy gradient implementation

Part 1: The environment implementation

We are going to implement two simple 2D environments using gym. We call them `chakra` and `vishamC`. Contour plots of the rewards from these environments are shown in Figures 2 and 3.

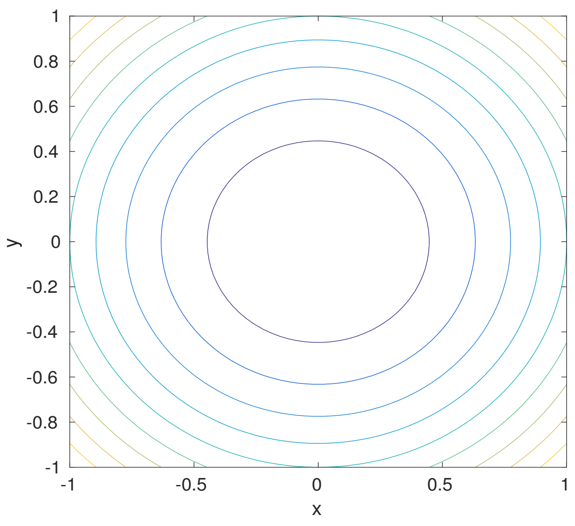

Figure 2: `chakra` world

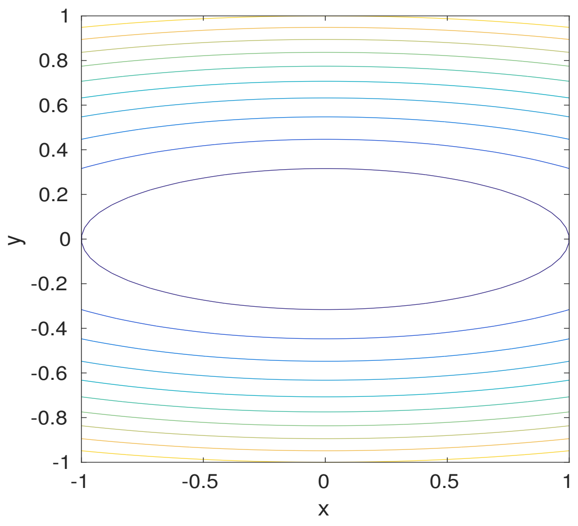

Figure 3: `vishamC` world

In both the environments, a point will be hurled at a random location at the beginning of each episode. The goal of the task is to move the point to the center as soon as possible.

A state from the environment is the current position of the point : (x, y), limited to [-1,1] along any of the axes. The action space is a 2-dimensional continuous vector that denotes the desired amount of movement in x, y coordinates: (dx, dy), clipped to [-0.025,0.025] along any of the directions. The environment is reset every time you move out of the designated area. The only difference between `chakra` and `vishamC` is their reward function. For `chakra` reward function is simply the distance of current state from the origin. For `vishamC`, the reward function is a little skewed, and given by : $0.5x^2 + \gamma 0.5y^2$, where γ is 10.

You are required to code these environments in OpenAI Gym. Name the folder you are doing this assignments in as `rlpa2`. In this part we are working with `chakra.py`. An environment class gives a complete specification of the environment using the following methods:

1. Initializer : To initialize the state and the action spaces
2. Random seed generator
3. Step : This method is the most important part of this code. It generates the next state and action, given the current state. You are required to fill this method.
4. Reset : Method to reset an episode
5. Render : Method to render the environment. Pass it for now.
6. Everytime you create a new environment with Gym, you need to register it in order to import and use it. The second answer in this link is how you do it.

Once you're done with this, you are ready with the environment for `chakra`. Next, you should create the environment for `vishamC`, on the same lines as the previous environment. Now that you have your environment ready, we need to know how a policy would interact with this environment. So, we move to the rollout function.

Part 2: The rollout function

This is an important part to understand how practical RL algorithms work. You need to understand the flow of the code. The main function is the entry point of the entire program. It sets corresponding environment-specific variables. Then, it initializes theta to some random value, and runs an infinite loop. In this loop, we alternate between calling get action to sample an action given the current state, and calling `env.step` to advance the state of the environment given the sampled action.

The first function in the file `rollout.py`, `chakra_get_action`, is to obtain the action that needs to be taken in a given world state, given the current set of parameters and the state. The overall code runs a linear policy on randomly initialized parameters.

Part 3: The policy gradient implementation

This is the part where you write bulk of the code. The flow of the code would be similar to the rollout function you saw above. We would not be providing you with any skeleton code for this part. Please follow the following instructions to go about writing your code.

Theory of gradient computation of the policy

To revise, we would first quickly go over standard policy gradient algorithm. To learn a policy, you first choose a policy parameterization. Once that is in control, you need a cost function to assess the quality of your policy. There are two such cost functions formulations: Average reward and discounted reward with a particular start state. The latter is what we use for our purpose of understanding policy gradients. We calculate the gradient of this cost function to update our current policy.

We would implement a batch algorithm, which means that we will collect a large number of samples per iteration, and perform a single update to the policy using these samples.

The formula for policy gradient is given by:

$$\nabla_\theta \, \mathbb{E}_{\pi_\theta}\left[\sum_{t=0}^{T} \gamma^t r_t\right] = \mathbb{E}_{\pi_\theta}\left[\sum_{t=0}^{T} \nabla_\theta \log_{\pi_\theta}(a|s)\,(R_t - b(s_t))\right], \tag{1}$$

where

- π_θ is a stochastic policy parameterized by θ;
- γ is the discount factor;
- s_t, a_t and r_t are the state, action, and reward at time t, respectively;
- T is the length of a single episode;
- $b(s_t)$ is a baseline function which only depends on the current state s_t (but does not depend on, say, the action a_t or future states s_t for $t' > t$);
- R_t is the discounted cumulative return, given by $R_t = \sum_{t'=t}^{T} \gamma^{t'-t} r_t$. Strictly speaking, the unbiased gradient should use $\gamma^{t'}$ instead of $\gamma^{t'-t}$. However, the gradient of the former term typically has a very weak signal from later time steps, and hence we typically use the latter term. The quantity $R_t - b(s_t)$ is an estimator of the *advantage* of taking action a_t in s_t. This is also denoted as $\widehat{A}(s_t, a_t)$.

Since this formula is an expectation, we cannot use it directly. It is more useful to consider a sampling-based estimator:

$$\nabla_\theta \, \mathbb{E}_{\pi_\theta}\left[\sum_{t=0}^{T} \gamma^t r_t\right] \approx \frac{1}{N}\sum_{i=1}^{N}\sum_{t=0}^{T} \nabla_\theta \log_{\pi_\theta}\left(a^i|si\right) \widehat{A}\left(s^i, a^i\right), \tag{2}$$

where we assume that in each iteration, N trajectories are sampled: $\tau_1, \tau_2, ...\tau_N$ and each trajectory τ_i is a list of states, actions, and rewards.

Computing policy gradient

We are going to use stochastic policies. How do you think we should parameterize the action distribution in such as case? Well, since the action space is continuous, we choose to use multivariate Gaussian distribution with identity covariance matrix. The mean is given by $\mu = \theta^T s'$, where s' is the vector obtained by appending 1 to the state s, so that we don't need a separate bias term. It returns a randomly sampled action from the distribution whose mean is μ.

You need to write two functions here. First one computes $\log \pi(a|s)$ for the above mentioned policy parameterization. Second, to calculate the gradient of that policy : $\bigtriangledown_\theta \log \pi(a|s)$.

Accumulating policy gradient

In the section above, you implemented only a part of the overall policy gradient update. In this section, you calculate the whole term represented by the sampling-based estimator of Eqn. 2.

You can follow the following pseudo code for this implementation:

```
Algorithm 1: Policy training loop
   Result: The learned policy
 1 initialization;
 2 while $itr \leq max\_itr$ do
 3     Collect trajectory;
 4     Store cumulative returns for each time step;
 5     while $n\_samples \leq batch\_size$ do
 6         Collect a new trajectory;
 7         while $t \leq T$ do
 8             Go back in time to compute returns and accumulate gradient;
 9             Compute the gradient along this trajectory;
10             Normalize the gradient;
11         end
12     end
13     Update the parameters with the gradient : theta += learning_rate * grad;
14     Print the itr , Average reward , parameter;
15 end
```

For rough normalization of the gradient, use :
`grad = grad / (np.linalg.norm(grad) + 1e-8)`

Questions and Deliverables

Once you are done with a working implementation, answer the following:

1. Carry out hyperparameter tuning: Tune the hyperparameters, like batch_size, learning_rate, discount factor, *etc.* What do you observe? Can you learn a policy faster?

2. Can you visualize a rough value function for the learned policy? If no, too bad. If yes, how would you go about it? Turn in the plots.

3. What do you observe when you try plotting the policy trajectories for the learned agent? Is there a pattern in how the agent moves to reach the origin?

Submission Guidelines

Submit a single tar/zip file containing the following files in the specified directory structure. Use the following naming convention: `rollno_PA2.tar.gz` or `rollno_PA2.zip`
A sample submission would look like this:

```
rollno_PA2
├── Code
│   ├── Q1
│   │   └── ...
│   └── Q2
│       └── ...
├── report.pdf
└── README
```