CS5500 : Reinforcement Learning

Assignment № 3

Due Date : 12/11/2021

Teaching Assistants : Sai Srinivas Kancheti and Dupati Srikar Chandra

---

Easwar Subramanian, IIT Hyderabad 25/10/2021

This assignment predominantly involves coding the Deep Q Network (DQN) algorithm and policy gradient (PG) algorithm and testing the implementation on several Open AI Gym environments. Read all the instructions and questions **carefully** before you start working on the assignment. The assignment policies remain same as in Assignment 1 and 2.

Additional Guidelines

- For questions that involve programming, do provide a short write-up for all (sub) questions asked. These can include performance reports, learning curves, hyper-parameter choices, other insights or justifications that a (sub) question may warrant.
- We expect that the training for the Pong game using DQN may take good amount of compute time. For example, on a modest laptop, it could take about two to three nights. Hence, please start the assignment early.
- For additional insights, some questions will require you to perform multiple runs of Q- learning, different hyper-parameter values or alternate learning rate or exploration schedules. Each of which can take a long time, if you choose work on complex environments.
- Due to high training time involved in some experiments, consider using tools like **screen** to run jobs in background.
- Due to computational constraints that some of you may have due to not being in campus, apart from the question involving the Pong game, all other questions involve training on simple Open AI Gym environments. Although, we have included a version number of the environment in the question, it is possible that you might find a slightly different version of the environment in the Open AI Gym repo.
- Although it is not required, those who have access to decent computational resources are encouraged to try out their code to train on other complex environments available in Gym. For example, you could test your DQN code on Atari games like Breakout or Space Invaders. And the PG code be tried out on continuous control tasks like inverted pendulum or half-cheetah. You may also want to try PG implementation on Atari games like Pong. Interesting additional experiments or extensions can be considered for bonus points.

Problem 1 : Deep Q Learning : Preliminaries

Problem 1 : Deep Q Learning : Programming

This problem involves coding the DQN algorithm and one of its variant, namely, DDQN. We will consider two environments from Open AI Gym for this question, namely, **MountainCar-v0** and **Pong-v0**. Although, the crux of the DQN algorithm was covered in lectures (Lectures 14,15 and 16 from Piazza), we encourage you to read the original papers, namely, [1, 2] . You may also want to look at numerous other materials available on the internet.

About Environments

- **MountainCar-v0**

  A car is on a one-dimensional track, positioned between two "mountains". The goal is to drive up the mountain on the right; however, the car's engine is not strong enough to scale the mountain in a single pass. Therefore, the only way to succeed is to drive back and forth to build up momentum. Here, the reward is greater if you spend less energy to reach the goal. This is depicted in Figure 1.

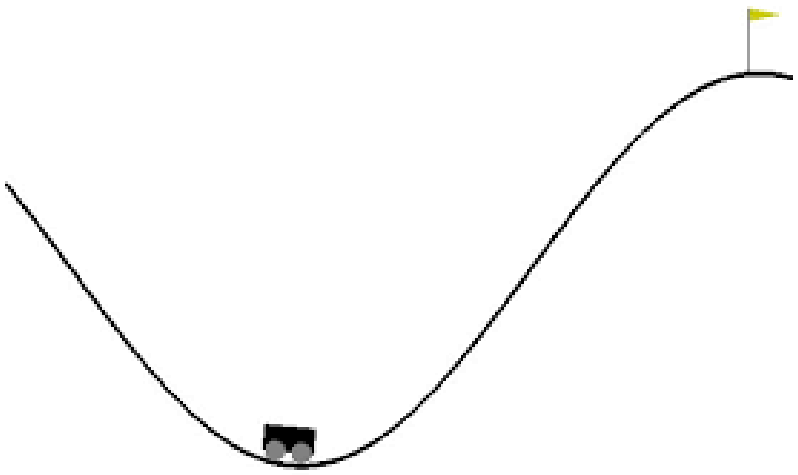

Figure 1: Mountain Car Environment

  The position and velocity of the car are the state (or input) variables. While the action consists of push left, push right and no push. For each time step that the car does not reach the goal, located at position 0.5, the environment returns a reward of -1. Depending on the version of the environment, an episode may end after 200 time-steps.

- **Pong-v0**

  Pong is a two-dimensional sports game that simulates table tennis. The agent that we intend to develop usually controls the paddle on the right. It competes with another hard-coded AI agent who controls the second paddle on the opposite side. Players use the paddles to hit a ball back and forth. The goal is to develop an agent that can beat the

hard-coded AI player on the other side. A screen shot of the game is provided in Figure 2.

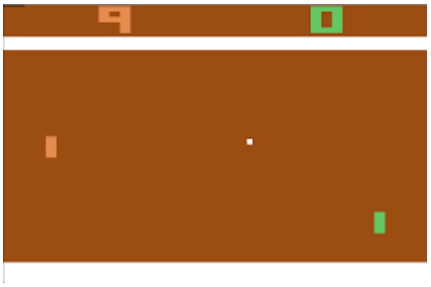

Figure 2: Pong Environment

As raw pixels of a frame are fed as input to the problem, there is no unique way formulate a state space. Usually, state space formulation involves some preprocessing of at least two past frames and combining them in suitable way. Suggested actions for the agent involve moving the paddle up (action 2), down (action 3) or do-nothing (action 0). In Pong, one player wins a volley if the ball passes by the other player. Winning a volley gives a reward of 1, while losing gives a reward of -1. An episode (or a game) is over when one of the two players reaches 21 wins.

(a) Develop a small code snippet to load the corresponding Gym environment(s) and print out the respective state and action space. Develop a random agent to understand the reward function of the environment. Record your observations. (6 Points)

(b) Implement the DQN algorithm to solve the tasks envisioned by the two environments. For the **Pong-v0** environment consider pre-processing of the frames (to arrive at state formulation) using the following step(s).

- Convert the RGB image into grayscale and downsample the resultant grayscale image.
- Further, one could also perform image subtraction between two successive frames

Any other pre-processing framework could also be used. Although, due to computational constraints, one may not be able to develop 'best' pong player, a reasonable pong player can be developed with a modest laptop (say in 2 to 4 million steps). Include a learning curve plot showing the performance of your implementation on the game **Pong-v0** and **MountainCar-v0**. The x-axis should correspond to number of time steps (suitably scaled) and the y-axis should show the mean n-episode reward (for a suitable choice of n) as well as the best mean reward. For the pong game, you could implement using normal feed forward networks or convnets. Accordingly, you may have to do suitable preprocessing. In the case of **MountainCar-v0** environment do plot a graph (one graph) that explains the action choices of the trained agent for various values of position and velocity. (15 Points)

(c) The DQN implementation in the above question involves choosing suitable values for a number of hyper-parameters. Choose one hyper-parameter of your choice and run at least

three other settings of this hyper-parameter, in addition to the one used in sub-question 1, and plot all four learning curves on the same graph. Examples include: learning rates, neural network architecture, replay buffer size, mini-batch size, target update interval, exploration schedule or exploration rule (e.g. you may implement an alternative to ϵ-greedy), etc. You are advised to choose a hyper-parameter that makes a nontrivial difference on performance of the agent. If computational resource is a constraint, consider answering this question only on **MountainCar-v0** environment. (8 Points)

(d) Implement the double DQN estimator on the **MountainCar-v0** environment and compare the performance of DDQN and vanilla DQN. Since there would be considerable variance between runs, you must run at least three random seeds for both DQN and DDQN and plot the learning curve as described in sub-question (a). If computational resource is a constraint, consider answering this question only on **MountainCar-v0** environment. (6 Points)

Problem 3 : Policy Gradient

This problem involves coding the policy gradient algorithm with variance reduction techniques. Consider the following tricks that will be used in the PG implementation.

- **Temporal Structure or Reward to go**

  Consider the following formulations of the policy gradient

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}\left[\sum_{t=0}^{\infty} \nabla_\theta \log \pi_\theta(a_t|s_t)\Psi_t\right]$$

  1. $\Psi_t = \sum_{k=0}^{\infty} \gamma^k r_{k+1} = G_{0:\infty}$, Total reward of the trajectory
  2. $\Psi_t = \sum_{t'=t}^{\infty} \gamma^{t'} r_{t'+1} = G_{t:\infty}$,Total reward following action a_t (reward to-go)

  where for a path $\tau \sim \pi_\theta$

$$G_{a:b}(\tau) = \sum_{t=a}^{b} \gamma^t r_{t+1}.$$

  The first formulation involves computing the total reward of the trajectory and the second formulation involves computing the **reward to go** that results from taking action a_t from state s_t. Supposedly, the second formulation is expected to lower the variance of the gradient estimate and would thus help in stabilizing the policy gradient algorithm.

- **Advantage Normalization**

  Advantage normalization involves centering advantage estimates (computed as reward-to-go minus a suitable baseline) by normalizing them to have mean of 0 and a standard deviation of 1. This technique is known to boost empirical performance of the policy gradient by lowering the variance of the advantage estimator thereby stabilizing the policy gradient algorithm.

We will now code a generic policy gradient algorithm and test it on two different environments from Open AI Gym, namely, **Cartpole-v0** and **Lunarlander-v2**.

About Environments

- **Cartpole-v0**

  A pole is attached by an unactuated joint to a cart, which moves along a friction-less track as shown in Figure 3. The input (or state) to the system consists of cart position, cart

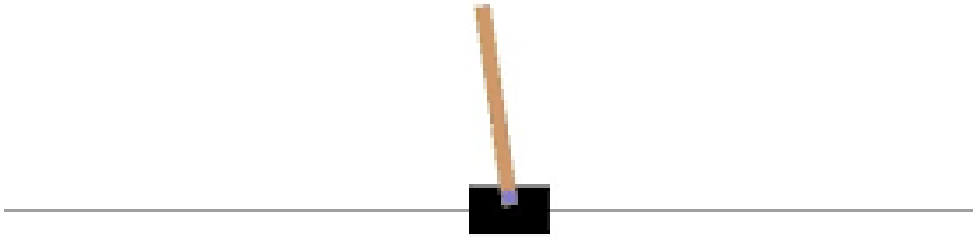

Figure 3: Cartpole Environment

  velocity, pole angle and pole velocity at tip. The system is controlled by applying a force that either pushes the cart to the left or right. The pendulum starts upright, and the goal is to prevent it from falling over. A reward of +1 is provided for every time-step that the pole remains upright. The episode ends when the pole is more than 15 degrees from vertical, or the cart moves more than 2.4 units from the center.

- **Lunarlander-v2**

  The goal of a Lunar lander, as you can imagine, is to land on the moon. Landing pad is always at coordinates (0,0). Coordinates are the first two numbers in state vector. Landing outside landing pad is possible. Fuel is infinite, so an agent can learn to fly and then land on its first attempt. Figure 4 depicts a schematic lunar lander.

  There are four discrete actions available: do nothing, fire left orientation engine, fire main engine, fire right orientation engine. The state constitute the coordinates and position of the lander. This reward function is determined by the Lunar Lander environment. The reward is mainly a function of how close the lander is to the landing pad and how close it is to zero speed, basically the closer it is to landing the higher the reward. But there are other things that affect the reward include firing the main engine deducts points on

Figure 4: Lunar Lander Environment

every frame, moving away from the landing pad deducts points, crashing deducts points, etc. The game or episode ends when the lander lands, crashes, or flies off away from the screen.

(a) Develop a small code snippet to load the corresponding Gym environments and print out the respective state and action space. Develop a random agent to understand the reward function of the environment. Record your observations. (2 Points)

(b) Implement the policy gradient algorithm with and without the reward to go and advantage normalization functionality. For the advantage normalization functionality, use a suitable baseline function (Slide 41, Lecture 17 : Policy Gradient Methods). Test them on **Cartpole-v0** and **Lunarlander-v2** environments. The code developed must be able to run from command line using options such as environment name, reward-to-go functionality (true or false), advantage normalization functionality (true or false), number of iterations, batch size for policy gradient and other command line variables as you deem fit. Attach in your reports, a comparison of learning curve graphs (average return at each iteration) with and with out advantage normalization and reward-to-go functionality. (12 Points)

(c) Study and report the impact of batch size on the policy gradient estimates on these environments. (6 Points)

(d) In the policy gradient lecture, we saw that adding a baseline to the policy gradient estimate does not create bias. The policy gradient formulation with temporal structure and baseline is given by,

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}\left[\sum_{t=0}^{\infty} \nabla_\theta \log \pi_\theta(a_t|s_t)\Psi_t\right]$$

where $\Psi_t = \sum_{t'=t}^{\infty} \gamma^{t'} r_{t'+1} - b(s_{t'}) = G_{t:\infty} - b(s_t)$. Prove that the policy gradient estimate $\nabla_\theta J(\theta)$ has minimum variance for the following choice of baseline b, given by,

$$b = \frac{\mathbb{E}_{\pi_\theta}(\nabla_\theta \log \pi(a_t|s_t)^2 G_{t:\infty}(\tau))}{\mathbb{E}_{\pi_\theta}(\nabla_\theta \log \pi(a_t|s_t)^2)}$$

(5 Points)

References

[1] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013.

[2] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, February 2015.