

# Reinforcement Learning Assignment

January 24, 2020

## 1 Introduction

The policy gradient section is inspired from the assignments of the course [1] at UC Berkeley.

Some definitions:

- A *trajectory*  $\tau$  is a sequence  $(s_1, a_1, s_2, a_2 \dots s_T, a_T)$  of the visited states and actions. We have  $p_\pi(\tau) = \pi(\tau) = p(s_1) \prod_{t=1}^T \pi(a_t|s_t)p(s_{t+1}|s_t, a_t)$ .
- The discounted cumulative reward is  $R_t = \sum_{k=0}^T \gamma^k r(s_{t+k}, a_{t+k})$ .
- The *Q-function*  $Q^\pi(s_t, a_t) = \mathbb{E}_{p_\pi}\{R_t|s_t, a_t\} = \sum_{t'=t}^T \mathbb{E}_{p_\pi}\{\gamma^{t'-t} r(s_{t'}, a_{t'})|s_t, a_t\}$  is the reward-to-go from state  $s_t$  if we pick the action  $a_t$  and then follow  $\pi$ .
- The *value function*  $V^\pi(s_t) = \mathbb{E}_{p_\pi}\{R_t|s_t\} = \sum_{t'=t}^T \mathbb{E}_{p_\pi}\{\gamma^{t'-t} r(s_{t'}, a_{t'})|s_t\} = \mathbb{E}_{a_t \sim \pi(a_t|s_t)}\{Q^\pi(s_t, a_t)\}$  is the reward-to-go from state  $s_t$  if we follow the policy  $\pi$ .
- The *advantage function*  $A^\pi(s_t, a_t) = Q^\pi(s_t, a_t) - V^\pi(s_t)$  represents the value of a particular action  $a_t$  with respect to the average action taken by  $\pi$  at state  $s_t$ .

Practical tips:

- Don't forget to enable the GPU in the notebook.
- 'YOUR CODE STARTS HERE' appears wherever you are expected to write your own code, or modify hyper-parameters.
- There are some recommended hyper-parameters written in the code. These are just some of the parameters that we tested, but they might not be optimal so feel free to tweak them.
- Try not to stop a cell that is currently training, as it might mess up the rendering of the agent for future runs.
- The 'cheetah' environment is the longest to train and fails to render in the provided notebook, so it is recommended to avoid it.

## 2 Policy gradient

The idea is to do stochastic gradient ascent on the objective  $J(\pi) = \mathbb{E}_{\tau \sim p_\pi(\tau)}\{R_1\}$  to find the best parameterized stochastic policy  $\pi_\theta$ . A simple example is the Reinforce algorithm [4].

The derivation is straight-forward. The idea is to use a logarithm to get rid of the terms depending on the dynamics:  $\nabla_\theta J(\theta) = \mathbb{E}_{\tau \sim p_\theta(\tau)}\{\nabla_\theta \log p_\theta(\tau) r(\tau)\} = \mathbb{E}_{\tau \sim p_\theta(\tau)}\{\nabla_\theta \sum_t \log \pi_\theta(a_t|s_t) r(\tau)\}$ . Step 2 of the algorithm uses samples to approximate the expectation. Step 3 is gradient ascent, making good sampled trajectories more likely, and bad trajectories less likely.

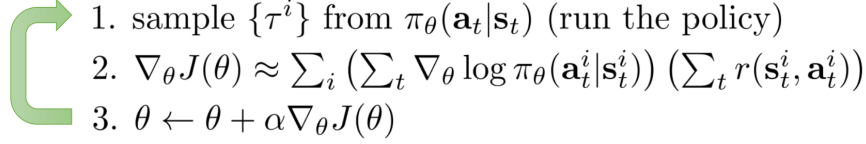


Figure 1: Reinforce algorithm (source: [1])

## 2.1 CartPole - Experiments

Run some experiments in the discrete [cartpole](#) environment. You should be able to reach a maximum score of 200. Two interesting parameters to tune are:

**reward\_to\_go.** We usually replace  $\sum_{t=1}^T \gamma^{t-1} r(s_t^i, a_t^i)$  with  $\sum_{t'=t}^T \gamma^{t'-t} r(s_{t'}^i, a_{t'}^i)$ , i.e. we hold policies accountable for future rewards only, which reduces the variance but might bring a bias when  $\gamma < 1$ .

**batch\_size.** Number of state-action pairs sampled while acting according to the current policy at each iteration. A bigger batch size should also reduce the variance.

## 2.2 LunarLander - Implement a neural network baseline

For this part you should use **reward\_to\_go**. You should be able to reach a maximum score of around 180 in the continuous [lunar\\_lander](#) environment.

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^N \sum_{t=1}^T \nabla_\theta \log \pi_\theta(a_t^i | s_t^i) \left( \sum_{t'=t}^T \gamma^{t'-t} r(s_{t'}^i, a_{t'}^i) - b \right) \quad (1)$$

Reinforce uses the MC (Monte Carlo) estimator of the Q-function with a single sample  $\hat{Q}^\pi(s_t, a_t) = \sum_{t'=t}^T \gamma^{t'-t} r(s_{t'}^i, a_{t'}^i)$ , hence the high variance. To reduce the variance, we can add a state-dependant baseline that approximates the value function  $b = \hat{V}^\pi(s_t)$ , which gives us an approximation of the advantage function  $\hat{A}^\pi(s_t, a_t)$ .

The goal is to approximate the value function with a neural network  $V_\phi$ . To do so, use the MC targets  $y_i = \sum_{t'} \gamma^{t'-t} r(s_{t'}^i, a_{t'}^i)$  and minimize the supervised regression loss  $\frac{1}{2} \sum_i \|V_\phi^\pi(s_t^i) - y_i\|^2$ .

- (Optional) Hint: It will be easier for the network to predict returns with zero mean and standard deviation of one. Use the function 'normalize' to compute normalized targets. Don't forget to scale the predicted returns to match the statistics of the un-normalized targets before computing the advantage.
- (Optional) Hint: Use the PyTorch function [detach\(\)](#) when computing the advantage to stop gradients from propagating back to the value function network when updating the policy.

## 3 Q-learning

The goal is to recover the optimal Q-function which verifies the Bellman equation:

$$Q^*(s_t, a_t) = \mathbb{E}_{s_{t+1} \sim p(\cdot | s_t, a_t)} \{ r(s_t, a_t) + \gamma \max_{a_{t+1}} Q^*(s_{t+1}, a_{t+1}) \} \quad (2)$$

To do so, we minimize the TD (temporal difference) error  $\sum_i \|Q_\phi(s_i, a_i) - y_i\|^2$  where  $y_i = r(s_i, a_i) + \gamma \max_{a'_i} Q_\phi(s'_i, a'_i)$ . A very popular Q-learning algorithm is DQN [3]. Let's cover the two main ideas:

- *Replay buffer.* As we interact with the environment sequentially, the samples are strongly correlated which can lead to local over-fitting. The idea is to have a buffer  $\mathcal{B}$  where we add the collected data, and then update the Q-function from data sampled uniformly from  $\mathcal{B}$ .
- *Target network.* We do not regress the Q-function to the TD targets  $y$  until convergence because the optimal estimate from the available data will be very poor in early stages. This means that the targets  $y$  are going to change very frequently, which doesn't help with stability. The idea is to do regular backups of the parameters  $\phi$  into a new target network  $Q_{\phi'}$  used to compute these targets.

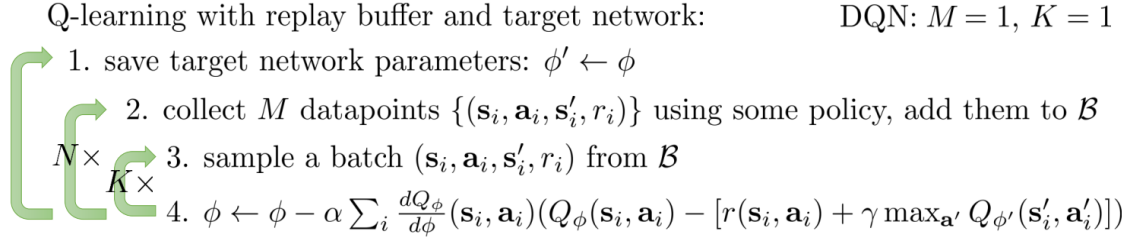


Figure 2: General Q-learning algorithm (source: [1])

### 3.1 MountainCar - Experiments

Solving the [mountain car](#) task is very difficult with the original reward, which gives -1 at every timestep until the car reaches the flag. In order to make any progress, the car has to reach the flag by exploring, and random exploration might not be good enough.

A better strategy would be to reward an increase in mechanical energy (potential plus kinetic) at every time step. To do so, simply uncomment the code that replaces the reward and compare the results. With this new reward you should be able to reach good original rewards (around -120). Note that the logs will still report the original rewards.

### 3.2 LunarLander - Implement double DQN

One problem with DQN is overestimation when computing the targets. Since  $Q_{\phi'}$  is not perfect,  $\max_{a'} Q_{\phi'}(s', a')$  will be an overestimation if there is any 'upwards' noise. If we could select the action according to a different network with decorrelated noise the problem would go away. Therefore, the idea is to use  $y = r + \gamma Q_{\phi'}(s', \arg \max_{a'} Q_{\phi}(s', a'))$  rather than  $y = r + \gamma Q_{\phi'}(s', \arg \max_{a'} Q_{\phi'}(s', a'))$ .

You can compare the stability of the learning process with or without DDQN. You should reach rewards of around 150.

## 4 Actor Critic

DDPG [2] is a popular actor-critic algorithm. It is an interesting example because it was introduced as an alternative to policy gradient for deterministic policies, and can also be seen as an alternative to Q-learning for continuous actions. A key feature is that it works off-policy.

### 4.1 Implement the actor update

- Hint: Thanks to the chain rule we have  $\nabla_{\theta} Q_{\phi}(s, \mu_{\theta}(s)) = \nabla_a Q_{\phi}(s, a = \mu_{\theta}(s)) \nabla_{\theta} \mu_{\theta}(s)$ .


- 
1. take some action  $\mathbf{a}_i$  and observe  $(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)$ , add it to  $\mathcal{B}$
  2. sample mini-batch  $\{\mathbf{s}_j, \mathbf{a}_j, \mathbf{s}'_j, r_j\}$  from  $\mathcal{B}$  uniformly
  3. compute  $y_j = r_j + \gamma \max_{\mathbf{a}'_j} Q_{\phi'}(\mathbf{s}'_j, \mu_{\theta'}(\mathbf{s}'_j))$  using *target* nets  $Q_{\phi'}$  and  $\mu_{\theta'}$
  4.  $\phi \leftarrow \phi - \alpha \sum_j \frac{dQ_\phi}{d\phi}(\mathbf{s}_j, \mathbf{a}_j)(Q_\phi(\mathbf{s}_j, \mathbf{a}_j) - y_j)$
  5.  $\theta \leftarrow \theta + \beta \sum_j \frac{d\mu}{d\theta}(\mathbf{s}_j) \frac{dQ_\phi}{d\mathbf{a}}(\mathbf{s}_j, \mathbf{a})$
  6. update  $\phi'$  and  $\theta'$

Figure 3: DDPG: deep deterministic policy gradient (source: [1])

## References

- [1] Sergey Levine. *CS285: Deep Reinforcement Learning*. Fa2019. URL: <http://rail.eecs.berkeley.edu/deeprlcourse/>.
- [2] Timothy P Lillicrap et al. “Continuous control with deep reinforcement learning”. In: *arXiv preprint arXiv:1509.02971* (2015).
- [3] Volodymyr Mnih et al. “Human-level control through deep reinforcement learning”. In: *Nature* 518.7540 (2015), pp. 529–533.
- [4] Ronald J Williams. “Simple statistical gradient-following algorithms for connectionist reinforcement learning”. In: *Machine learning* 8.3-4 (1992), pp. 229–256.