deep_reinforcement_learning_project2

Overview

For this project, you will work with the Reacher environment.

In this environment, a double-jointed arm can move to target locations. A reward of +0.1 is provided for each step that the agent's hand is in the goal location. Thus, the goal of your agent is to maintain its position at the target location for as many time steps as possible.

The observation space consists of 33 variables corresponding to position, rotation, velocity, and angular velocities of the arm. Each action is a vector with four numbers, corresponding to torque applicable to two joints. Every entry in the action vector should be a number between -1 and 1.

Getting Started

For this project, you will not need to install Unity if you work on udacity - this is because udacity have already built the environment for you

Download the Unity Environment

For this project, you will not need to install Unity - this is because we have already built the environment for you, and you can download it from one of

the links below. You need only select the environment that matches your operating system:

Linux: click here

Mac OSX: click here

Windows (32-bit): click here

Windows (64-bit): click here

Then, place the file in the p2_continuous-control/ folder in the DRLND GitHub repository, and unzip (or decompress) the file.

(For Windows users) Check out this link if you need help with determining if your computer is running a 32-bit version or 64-bit version of the Windows operating system.

(For AWS) If you'd like to train the agent on AWS (and have not enabled a virtual screen), then please use this link (version 1) or this link (version 2) to obtain the "headless" version of the environment. You will not be able to watch the agent without enabling a virtual screen, but you will be able to train the agent. (To watch the agent, you should follow the instructions to enable a virtual screen, and then download the environment for the Linux operating system above.)

Install Dependencies

Please install the following python packages:

tensorflow==1.7.1

Pillow>=4.2.1

matplotlib

numpy>=1.11.0

jupyter

pytest>=3.2.2

docopt

pyyaml

protobuf==3.5.2

grpcio==1.11.0

torch==0.4.0

pandas

scipy

ipykernel

Code Architecture

Continuous_Control.ipynb: jupyter notebook based solution

ddpg_agent.py: DQN agent code

model.py: Q-Network based model

actor.pth: weights of the actor model

critic.pth: weights of the critic model

Instructions

In this project, we could run Continuous_Control.ipynb within the env, train the model and get the results.

DDPG Algorithm Description

Deep Deterministic Policy Gradient (DDPG) is an algorithm which concurrently learns a Q-function and a policy. It uses off-policy data and the Bellman equation to learn the Q-function, and uses the Q-function to learn the policy.

This approach is closely connected to Q-learning, and is motivated the same way: if you know the optimal action-value function $Q^*(s,a)$, then in any given state, the optimal action $a^*(s)$ can be found by solving

$$a^*(s) = \arg\max_a Q^*(s,a).$$

DDPG interleaves learning an approximator to $Q^*(s,a)$ with learning an approximator to $a^*(s)$, and it does so in a way which is specifically adapted for environments with continuous action spaces. But what does it mean that DDPG is adapted *specifically* for environments with continuous action spaces? It relates to how we compute the max over actions in $\max_a Q^*(s,a)$.

When there are a finite number of discrete actions, the max poses no problem, because we can just compute the Q-values for each action separately and directly compare them. (This also immediately gives us the action which maximizes the Q-value.) But when the action space is continuous, we can't exhaustively evaluate the space, and solving the optimization problem is highly non-trivial. Using a normal optimization algorithm would make calculating $\max_a Q^*(s,a)$ a painfully expensive subroutine. And since it would need to be run every time the agent wants to take an action in the environment, this is unacceptable.

Because the action space is continuous, the function $Q^*(s,a)$ is presumed to be differentiable with respect to the action argument. This allows us to set up an efficient, gradient-based learning rule for a policy $\mu(s)$ which exploits that fact. Then, instead of running an expensive optimization subroutine each time we wish to compute $\max_a Q(s,a)$, we can approximate it with $\max_a Q(s,a) \approx Q(s,\mu(s))$. See the Key Equations section details.

1) Description refers to [Deep Deterministic Policy Gradient — Spinning Up documentation (openai.com)](openai.com).

2) The code implementation referenced the code provided by Udacity.

DDPG Algorithm Pseudocode

Algorithm 1 Deep Deterministic Policy Gradient

Input: initial policy parameters θ, Q-function parameters ϕ, empty replay buffer $\mathcal{D}$
Set target parameters equal to main parameters $\theta_{\text{targ}} \leftarrow \theta$, $\phi_{\text{targ}} \leftarrow \phi$
repeat
  Observe state s and select action $a = \text{clip}(\mu_\theta(s) + \epsilon, a_{Low}, a_{High})$, where $\epsilon \sim \mathcal{N}$
  Execute a in the environment
  Observe next state s', reward r, and done signal d to indicate whether s' is terminal
  Store (s, a, r, s', d) in replay buffer $\mathcal{D}$
  If s' is terminal, reset environment state.
  if it's time to update **then**
    for however many updates **do**
      Randomly sample a batch of transitions, $B = \{(s, a, r, s', d)\}$ from $\mathcal{D}$
      Compute targets

$$y(r, s', d) = r + \gamma(1 - d)Q_{\phi_{\text{targ}}}(s', \mu_{\theta_{\text{targ}}}(s'))$$

      Update Q-function by one step of gradient descent using

$$\nabla_\phi \frac{1}{|B|} \sum_{(s,a,r,s',d) \in B} \left(Q_\phi(s, a) - y(r, s', d)\right)^2$$

      Update policy by one step of gradient ascent using

$$\nabla_\theta \frac{1}{|B|} \sum_{s \in B} Q_\phi(s, \mu_\theta(s))$$

      Update target networks with

$$\phi_{\text{targ}} \leftarrow \rho\phi_{\text{targ}} + (1 - \rho)\phi$$
$$\theta_{\text{targ}} \leftarrow \rho\theta_{\text{targ}} + (1 - \rho)\theta$$

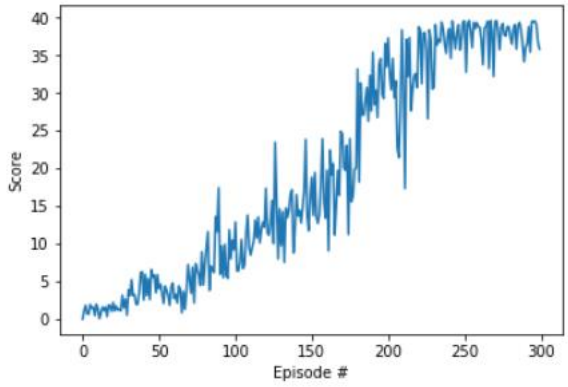
    end for
  end if
until convergence

*) The DDPG algorithm is from [Deep Deterministic Policy Gradient — Spinning Up documentation (openai.com)]

Result

```
Episode 10      Average Score: 1.17
Episode 20      Average Score: 1.16
Episode 30      Average Score: 1.33
Episode 40      Average Score: 1.96
Episode 50      Average Score: 2.50
Episode 60      Average Score: 2.69
Episode 70      Average Score: 2.78
Episode 80      Average Score: 3.13
Episode 90      Average Score: 3.83
Episode 100     Average Score: 4.26
Episode 110     Average Score: 5.07
Episode 120     Average Score: 6.08
Episode 130     Average Score: 7.32
Episode 140     Average Score: 8.22
Episode 150     Average Score: 9.30
Episode 160     Average Score: 10.57
Episode 170     Average Score: 12.02
Episode 180     Average Score: 13.41
Episode 190     Average Score: 15.30
Episode 200     Average Score: 17.66
Episode 210     Average Score: 19.81
Episode 220     Average Score: 21.76
Episode 230     Average Score: 23.89
Episode 240     Average Score: 26.27
Episode 250     Average Score: 28.48

Environment solved in 157 episodes!     Average Score: 30.02
Episode 260     Average Score: 30.66
Episode 270     Average Score: 32.57
Episode 280     Average Score: 34.46
Episode 290     Average Score: 35.38
Episode 300     Average Score: 35.97
```



It takes 157 episodes to reach the average score 30.02, please refer to the results and picture above.

Ideas for future work

Try new algorithms such as dd4g in the future.