Project

gym-Pendulum-v0 with **DDPG** by **@smileandyxu (3180106206 徐正浩)**

Project Introduction

Environment

Virtual Game Environment

gym-Pendulum-v0

Response

Observation & State (dim=2):

cos(theta): [Real Number] (Min=-1.0, Max=1.0)

sin(theta): [Real Number] (Min=-1.0, Max=1.0)

thetadot: [Real Number] (Min=-8.0, Max=8.0)

Actions (dim=1):

jointeffort: [Real Number] (Min=-2.0, Max=2.0)

Reward:

$ reward=-(\theta^2+0.1*\theta_{dt}^2+0.001*action^2)\in[-16.2736044, 0]$

Initial State:

theta: [Real Number] (Min=-π, Max=π)

velocity: [Real Number] (Min=-1, Max=1)

Episode Termination:

None

Work

Goal

Learn to control the pendulum to get maximized reward during each episode within fixed steps.

Algorithm

DDPG - Can deal with continuous action space; convergence result is stable.

DDPG contains an Actor-Critic model. It has a Actor network used to predict what action to take under particular state, and a Critic network used to evaluate the Q-value of a given state-action pair. It also uses a delayed gradient update policy by holding two set of networks, 'current networks' and 'target networks', which make it produce relatively stable performance.

Develop Environment

Tools:

Python 3.7.3

PyTorch 0.4.1

Libs:

gym 0.13.1

Runtime Requirement

Windows 10 & Debian

Technical Details

Theory

- Actor-Critic Model

  Actor-Critic Model is an approach to solve DRL problems combining the advantages of both Value-Based approaches like DQN and Policy-Based approaches like Policy Gradient. It uses an Actor network to give a policy function P(s) (which tells the agent what to do when at state s) and a Critic network to give an approximation of Q-function Q(s, a) (which tells the potential reward if one takes action a when at state s). We train both networks to get result. Details can be seen in reference[2]

- Determined Policy Gradient (DPG)

  Since with plain DQN we can only take discrete values as actions, and 'Pendulum' has action value which can be real numbers. So we use DPG here, not to output a probability distribution of actions, but to output real values which can be considered as particular actions. We transform classification mission in DQN to a regression problem in DPG. In each state, we have a determined action to take. We can add a noise on the output of the regression model to get variant result.

- Deep Determined Policy Gradient (DDPG)

  Directly using neural networks in DPG is proved to be unstable, since our estimate of Q-function is continuously being updated during each optimization step, thus changing our evaluation of current (s, a) pairs. So we have to let it update, not that frequently. We use another pair of Actor-Critic networks, called target networks, to maintain relatively stable parameters. During each time we pull out some memories to feed the current (or pioneer) model, then use current model to update stable model. This helps us to get a convergent result. Details can be seen in the original paper [1].

Algorithm

See reference [1]:

Algorithm 1 DDPG algorithm

Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights θ^Q and θ^μ.
Initialize target network Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer R
for episode = 1, M **do**
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial observation state s_1
  for t = 1, T **do**
    Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise
    Execute action a_t and observe reward r_t and observe new state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in R
    Sample a random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from R
    Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
    Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
    Update the actor policy using the sampled policy gradient:

$$\nabla_{\theta^\mu} J \approx \frac{1}{N} \sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s_i}$$

    Update the target networks:

$$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$
$$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$

  end for
end for

Implementation

The project contains `main.py`, `brain.py`, `DDPG.py`, `utils.py` and `config.py`.

`main.py` is the entrance of the project, which starts the DDPG algorithm process.

In `brain.py` we inherit `nn.Modules` in PyTorch to build up `Actor` network and `Critic` network. They both have two hidden layers with dimension of 64 and 32, connections are all full.

`DDPG.py` is the algorithm process, contains a class `DDPG`. It mainly consists of two pairs of Actor-Critic networks `actor`, `critic`, `target_actor`, `target_critic`, the latter two of these are delayed networks which update their parameters with the former two. And their are two optimizers `actor_opt` and `critic_opt` which inherit AdamOptimizer. It has a list of shape (batch_size, batch_size, batch_size, batch_size, batch_size) `memory` to store training data. It has a gym environment `env` to control the play. There are some other members to fully complete the algorithm.

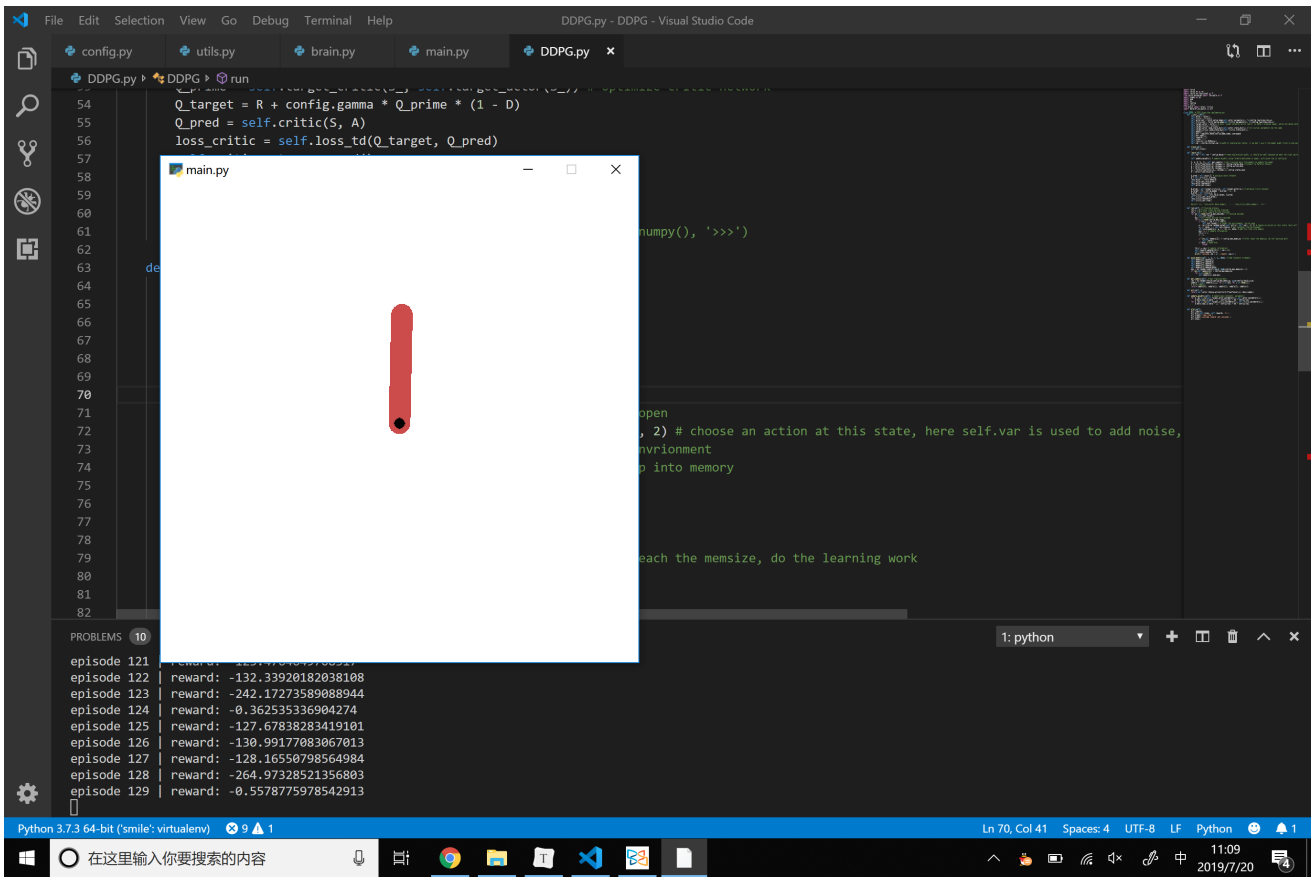
`utils.py` contains some useful functions that are not on the spot. Some of them have been removed during our development process. Now it only has a function `inin_nns(layers)` to initialize our model.

`config.py` contains some configurations and hyper-parameters used in the project. We use `lr=0.001` for both Actor-Net and Critic-Net. We use `gamma=0.9` to develop Actor-Critic, `tau=0.01` to develop DDPG, `initial_var=3.0` and `decay=0.995` for exploration noise. `max_episode=500`, `max_step`=200. Memory size is `max_memsize=8192`, and training batch `batch_size=32`.

Details can be seen in codes. There are explicit annotations.

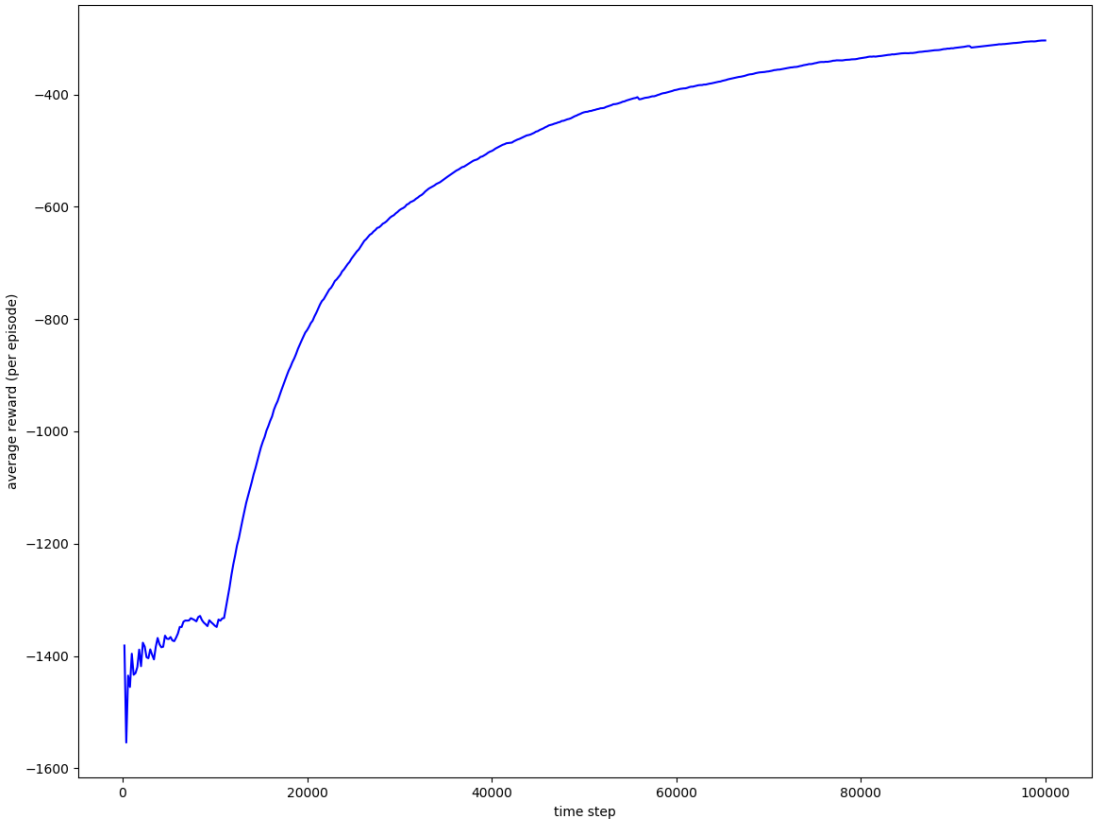
Experiment Result

Runtime:



More can be seen in `./video`.

Performance:



References

[1] Continuous control with deep reinforcement learning, T. P. Lillicrap et al.,ICLR, 2016.

[2] https://zhuanlan.zhihu.com/p/29486661 Actor-Critic算法小结

[3] https://github.com/talebolano/example_of_reinforcement_lreaning_by_pytorch example_of_reinforcement_lreaning_by_pytorch