Reproducibility report of "Continuous control with deep Reinforcement Learning"

19335010 ZENG LINGZHE

2022-1-19

1 Introduction

In previous courses, we learned the basic algorithms of reinforce learning, such as Q-learning, Sarsa, DQN. These algorithms usually perform well when they are used to solve the discrete problems. But the problems with a continuous action space act like a obstacle. One traditional method is to convert the continuous problems into the discrete problems by select several fixed values in the action space to approximate the whole action space. But beyond all doubt, it has a low performance due to the smaller domain. Besides, if the action space is large and has more dimensions, we have to choose more discrete action to get a better approach to the origin problem, resulting in a huge Q-table to compute. The paper "Continuous control with deep Reinforcement Learning", which we will reproduce, proposes the ideas that adapt the DQN to the continuous action domain. It mainly contains the ideas about actor-critic, deterministic policy gradient, replay memory and so on.We reproduce the DDPG algorithm presented in this paper with Pytorch and use the simple games in Gym Library to validate our implementation. Then we will discuss the problems in the process of implementation. The implementation is in `https://github.com/zeng798473532/DDPG-RL-final/`

2 Scope of reproducibility

DDPG is an acronym for "Deep Deterministic Policy Gradient", mainly used to settle the problems that receive continuous variables as the input. Deep means that we use

a deep neural network to build our model, and Deterministic Policy implies that we will get a deterministic action output from our model based on the experience from the histories and the current state. A significant mission of learning in the continuous space is exploration. The model will output the deterministic policy, signifying that this is the best policy that the model has learned. If we just make it into use, we maybe trap into the local optimum. So exploration is a indispensable process in our model training. In the paper, author pointed out that the Ornstein-Uhlenbeck process has a better exploration efficiency in the physical control problems with inertia. Other available choice is Gauss. In this report, our main works are:
1. We will compare these two noises in our reproduction.
2. we will use multiple game environment to test our model.

3 Methodology

The first method that combines Q-learning and deep neural network is NIPS DQN. It's mainly used to solve the problems with the discrete input space. It uses a deep neural network to get an approximation of the Q-table. Its core formulas are shown below:

$$\epsilon\text{-greedy for action } a_t = \max_a Q^*(\phi(s_t), a; \theta)$$

$$y_j = \begin{cases} r_j & \phi_{j+1} \text{ is done.} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \phi_{j+1} \text{ is not done.} \end{cases}$$

$$Loss = (y_i - Q(\phi_j, a; \theta))^2$$

which ϕ_j is the state at time j and Q is the function that estimates the q-value with neural network. Then various improved algorithms are put forward and have better performance, such as the Natural DQN, Double DQN and so on.
As a contrast, the DDPG use a similar loss function to optimize the model when facing with the continuous problems. The original algorithm is just as follows.

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

Fig. 1. DDPG Algorithm

The main difference between DDPG and traditional DQN are the selection of action and the model structures. We will talk about those in more detail.

3.1 Model Descriptions

Actor-Critic Network. The DDPG network contains two parts. One is Critic network, responsible for generating reward values and evaluating the input action, which acts like the Q-table and DQN network said above. The other one is Actor network, taking charge of generating the upcoming action by the current state. First, the Actor network get the state and output an action. Then the Critic network will output the reward, Q-value in another word, after processing the current state and the action. These two network is defined by several linear layers together with specific activation functions. The codes below clearly illustrate it.

```
class Actor(nn.Module):
    def __init__(self, state_dim, action_dim):
        super().__init__()
        self.linear1 = nn.Linear(state_dim, 256)
        self.linear2 = nn.Linear(256, 64)
        self.linear3 = nn.Linear(64, action_dim)

    def forward(self, state):
        y = F.relu(self.linear1(state))
        y = F.relu(self.linear2(y))
        y = torch.tanh(self.linear3(y))
        return y
```

```
class Critic(nn.Module):
    def __init__(self, state_dim, action_dim):
        super().__init__()
        self.linear1 = nn.Linear(state_dim + action_dim, 256)
        self.linear2 = nn.Linear(256, 64)
        self.linear3 = nn.Linear(64, 1)

    def forward(self, state, action):
        x = torch.cat((state, action), dim=1)
        x = F.relu(self.linear1(x))
        x = F.relu(self.linear2(x))
        x = self.linear3(x)
        return x
```

Target Network. If we only use single network, then we will find that the target value y and the network we need to update is the same one, making the model easier to diverge. So a standalone target network is necessary. In another word, we have 4 networks in our model: Actor, Critic, Actor-target, Critic-target. Actor, Critic network can also be call policy network. The author made a small modification that the target network should be soft-update rather than directly copy the weights from the policy network. This will lead to tiny change to the target network and make the training more steady. The process of soft-update is as follows. First we need to define a parameter τ, then we update all the weight in target network by calculate $\boldsymbol{w}_{target} = \tau \boldsymbol{w}_{policy} + (1 - \tau)\boldsymbol{w}_{target}$, where τ is 0.01 in our implementation.

Replay Memory Pool. In traditional deep learning, we usually have a training set to learn the model weights. While in DDPG, we also need to "build"a training set, called Replay Memory Pool. At every step, we store the current state, action, reward, next action (s_i, a_i, r_i, s_{i+1}) into the Replay Memory Pool. When learning, we sample a random mini-batch of N transitions (s_i, a_i, r_i, s_{i+1}) from the Replay Memory Pool and use them to train our model. It needs to point out that the pool size should be adjusted to fit with the job. If the pool size is too small, it means that we just learn from the recent transitions. If the pool size is too big, then it will waste storage space and learn slowly.

3.2 Test Toolkit

We use gym toolkit to assist us in training and testing. The gym library is a collection of test problems —environments —that we can use to work out the reinforcement learning algorithms, from basic physics control problems to robotic simulation. The environments have a shared interface, allowing us to write general algorithms. We choose two classic control problems to test our model's performance.

Pendulum. The inverted pendulum swing-up problem is a classic problem in the control literature. The pendulum starts in a random position, and the goal is to swing it up so it stays upright.

Mountain Car. 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. The reward is greater if you spend less energy to reach the goal.

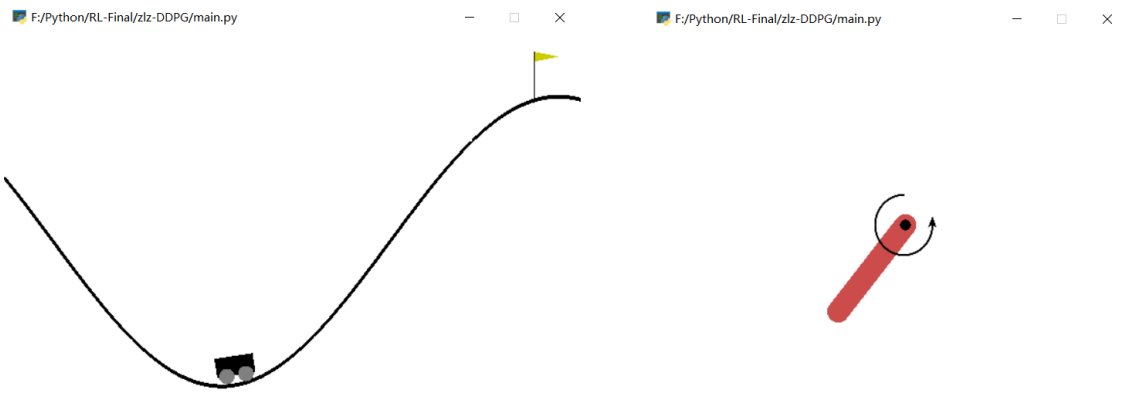

Fig. 2. Gym game: left is MountainCar, right is Pendulum

3.3 Basic Improvement

Warm-up stage. At the beginning of the training, the Replay Memory Pool is too empty to satisfy a mini-batch. So we add a warm-up stage to fill in the Replay Memory Pool in advance. Owing to the untrained model weights, we suppose that random action is better in warm-up stage than using Actor output.

Noise decrease progressively. As we said above, noise is an essential part in the exploration period. It can prevent the agent to trap into local optimization and help it to find out the better weights. However, when learned for sufficient episodes, noise will have a negative effect on convergence. So, we add a parameter epsilon ϵ to control the noise weight. In early episodes, the noise is loud and as the train goes on, it weights will decrease until the minimum value is arrived. We use a simple exponential function to implement this process. The ϵ change curve is shown below.

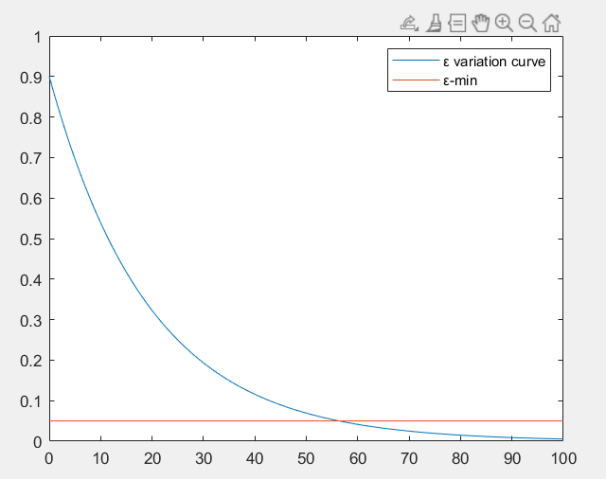

Fig. 3. x-axis is episode, y-axis is epsilon.

3.4 Experiment Environment

Pytorch is an open source machine learning framework that can help us simplify the coding and accelerate the training process. It provides many defined network layers and network optimizer. We use Linear layer to build our model and Adam optimizer to learn the model weights. Other software and hardware environments are listed below.
Python:3.8.12
Pytorch: 1.4.0+cu92
Gym:0.21.0
Hardware: Intel 9750H + Nvidia GTX 1050 3G in Win10

4 Results

The result demos can be found in the GitHub repository. In the Pendulum game, we can find that the agent makes action badly at the very beginning, but after

some episodes, it learned a better policy to do the game. It can make the pendulum swing up in a short time. As for the Mountain-Car game, the car almost remained in the valley initially, and after some learning, it can achieve the goal perfectly. The rewards in the training and testing period are show below.

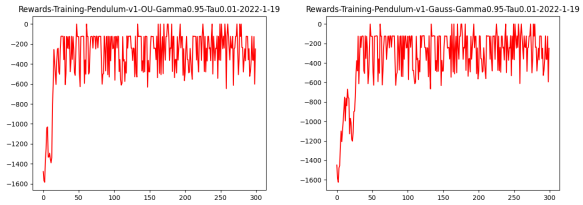

Fig. 4. Pendulum rewards in training with OU and Gauss

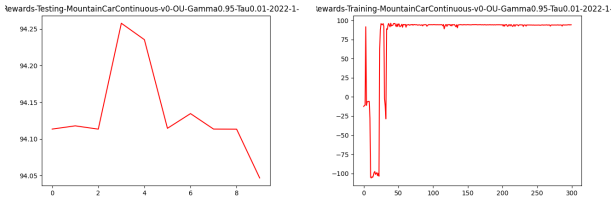

Fig. 5. MountainCar rewards with OU in training and testing

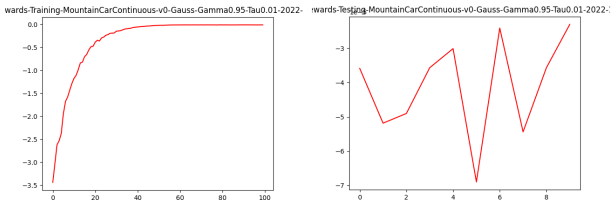

Fig. 6. MountainCar rewards with Gauss in training and testing

As for the noise factor, we use Gauss and OU noise to compare the results. When considering the Pendulum game, we find that Gauss noise will make the action more wiggle, just as the demo video shows. That is, the action will change frequently to maintain the pendulum swinging up. In regard of Mountain-Car, Gauss noise perform worse just like Fig.6. It is too bad to learn a available policy. Gauss is a fully random noise, where every two noises are independent. Imagine trying to swim by nervously shaking your arms and legs in every direction in some chaotic and out of sync manner. That wouldn’ t be very efficient to find a method to swim, would it? But OU noise has the characteristic that the next noise is based on the previous noise. It will make exploration in one direction for a distance. This is useful in the physics environment with momentum.

5 Summary and Discussion

In summary, DDPG is a pioneering algorithm in reinforce learning. It spans the control problem in the continuous space and give a practice solution. In many fields, DDPG can play a vital role, such as robot control, automatic driving, modern team game and so on.

In the process of our reappearance, we also find some problems. For example, there is some probability that the rewards would suddenly slump when training. Second, some parameters should be changed to satisfy different missions. Third, there is still a question that whether these noise can help to escape from the local optimal solutions. We have to do more experiments to explore.

References

[1] Mnih, Volodymyr et al. "Playing Atari with Deep Reinforcement Learning." ArXiv abs/1312.5602 (2013): n. pag.

[2] Mnih, Volodymyr, Kavukcuoglu, Koray, Silver, David, Rusu, Andrei A, Veness, Joel, Bellemare, Marc G, Graves, Alex, Riedmiller, Martin, Fidjeland, Andreas K, Ostrovski, Georg, et al. Humanlevel control through deep reinforcement learning. Nature, 518(7540):529–533, 2015.

[3] Sutton, R. S. , et al. "Policy Gradient Methods for Reinforcement Learning with Function Approximation."Submitted to Advances in Neural Information Processing Systems 12(1999).

[4] Silver, David , et al. "Deterministic Policy Gradient Algorithms."JMLR.org.