Reproducibility report of "Continuous Control with Deep Reinforcement Learning"

0.1 Reproducibility summary

The code is open source at

`https://github.com/EstherBear/Reproduction-of-Continuous-Control-with-Deep-Reinforcement-Learning-`

Scope of reproducibility

The original work by lillicrap et al.[1] claimed (1) that DDPG can learn policies in high-dimensional, continuous action spaces. and (2) that the components (batch normalization, target network and generating exploration noise from Ornstein-Uhlenbeck process) of DDPG is necessary. In addition, lillicrap et al.[1] stated that (3) in simple tasks DDPG estimates returns accurately without systematic biases while for harder tasks the Q estimates are worse, but DDPG is still able to learn good policies.

Methodology

Since the original paper's code was not published officially, nearly all of the codebase was written from scratch based on the description in the paper. The implementation of DQN and Ornstein-Uhlenbeck process online is referenced. I mostly used a Nvidia's TITAN Xp as the GPU and a Intel(R) Xeon(R) E5-2637 v4 as the CPU. The runtime of each physical environment was highly dependant on the architecture used. The runtimes for each environment can be found in Tabel 2.

Results

DDPG is able to learn good policies on various of tasks with continous action space in different environment including Classic control, Box2D and MuJoCo which supports the first claim.

The claim that components of DDPG is necessary is only partially supported by the results. There is no doubt that the target network is critical. However, the necessity of batch normalization and Ornstein-Uhlenbeck process is questionable according to the results.

Additionally the claim that DDPG estimates returns accurately without systematic biases in simple tasks and for harder tasks the Q estimates are worse is not supported by the results. But DDPG is still able to learn good policies regardless of the accuracy of estimated Q value.

What was easy

Different physical environments are easy to get access to with the gym package. And some techniques used by DDPG are well documented and can be found online.

What was difficult

Some hyper-parameters and computing methods used in figures were not included in the publication. Additionally, some physical environments with continuous space presented in the original paper are not open source.

Communication with original authors

We did not contact the original authors of the publication.

0.2 Introduction

At the time of this article's publication, reinforcement learning has been combined with deep learning to handle complex tasks. Among them, a relatively important work is "Deep Q Network"(DQN)[2]. DQN uses deep neural network to fit action-value function in reinforcement learning.

However, DQN can only handle discrete, low-dimensional action space, while many physical control tasks have continuous, high-dimensional action space. The Deterministic Policy Gradient (DPG) algorithm proposed by Silver et al.[3] can deal with continuous action space. Therefore, Deep DPG(DDPG), proposed by Lillicrap et al.[1], combines insights in DQN and DPG to learn policy in high-dimensional and continuous action space. This paper aims at reproducing DDPG.

lillicrap et al.[1] verified DDPG performance in different challenging physical control problems with continuous action space. In addition to following the DQN:

1. The network is trained off-policy with samples from a replay buffer to minimize correlations between samples;
2. The network is trained with a target Q network to give consistent targets during temporal difference backups;

batch normalization is also used by the authors of DDPG to improve performance of DDPG. The authors also proposes to use the Ornstein-Uhlenbeck [5] to generate exploration noise which can be added to actor policies to generate exploration policies.

0.3 Scope of reproducibility

Lillicrap et al.[1] mainly solved the problem of using deep neural network to learn policy in high-dimensional and continuous action space. Besides, experiments were carried out to analyze estimated Q value and real returns.

In this paper, I test the following concrete claims:

1. DDPG can learn policies in high-dimensional, continuous action spaces.
2. The various components (including batch normalization, target network and noise from Ornstein-Uhlenbeck process) of DDPG is necessary.
3. In simple tasks DDPG estimates returns accurately without systematic biases. For harder tasks the Q estimates are worse, but DDPG is still able to learn good policies.

Tabel 1: Modes of variants of DPG

mode	variants of DPG
bn	with batch normalization and exploration noise generated by Ornstein-Uhlenbeck process
tn	with target network and exploration noise generated by Ornstein-Uhlenbeck process
bntn	with batch normalization, target network and exploration noise generated by Ornstein-Uhlenbeck process
normalnoise	with target network and exploration noise generated by Normal Distribution

0.4 Methodology

Network architecture

DDPG adopts actor-critic framework including four networks: Critic, Target Critic, Actor, and Target Actor.

Among them, critic network, as Q function, maps state and action to Q-value while actor network acts as policy network, mapping state directly to action.

The target network is updated softly with the parameters of its original network. To improve the stability of training, the original network is trained with a target network to give consistent targets during temporal difference backups.

According to the description of the supplementary in the paper, both actor network and critic network have 2 hidden layers with 400 and 300 units respectively. All hidden layers are followed by ReLU and Batch Normalization, and the final output layer of the actor is a tanh layer to bound the actions.

To verify the necessity of each part in the DDPG, I add the mode parameter to the actor and critic network implementation to distinguish between different modes. (See Tabel 1 for all modes in this reproduction) Listing 1 is a concrete implementation of actor and critic network.

```
1  class Actor(torch.nn.Module):
2
3      def __init__(self, state_dimension, action_dimension, max_action):
4          super(Actor, self).__init__()
5          self.linear1 = torch.nn.Linear(state_dimension, 400)
6          self.linear2 = torch.nn.Linear(400, 300)
7          self.linear3 = torch.nn.Linear(300, action_dimension)
8          self.bn1 = nn.BatchNorm1d(400)
9          self.bn2 = nn.BatchNorm1d(300)
10         self.max_action = max_action
11
12     def forward(self, state, mode):
13         if mode == 'bn' or mode == 'bntn':
```

```
14              a = F.relu(self.bn1(self.linear1(state)))
15              a = F.relu(self.bn2(self.linear2(a)))
16          else:
17              a = F.relu(self.linear1(state))
18              a = F.relu(self.linear2(a))
19          return self.max_action * torch.tanh(self.linear3(a))
20
21
22  class Critic(torch.nn.Module):
23
24      def __init__(self, state_dimension, action_dimension):
25          super(Critic, self).__init__()
26          self.linear1 = torch.nn.Linear(state_dimension +
        action_dimension, 400)
27          self.linear2 = torch.nn.Linear(400, 300)
28          self.linear3 = torch.nn.Linear(300, 1)
29          self.bn1 = nn.BatchNorm1d(400)
30          self.bn2 = nn.BatchNorm1d(300)
31
32      def forward(self, state, action, mode):
33          if mode == 'bn' or mode == 'bntn':
34              q = F.relu(self.bn1(self.linear1(torch.cat([state, action
        ], 1))))
35              q = F.relu(self.bn2(self.linear2(q)))
36          else:
37              q = F.relu(self.linear1(torch.cat([state, action], 1)))
38              q = F.relu(self.linear2(q))
39          return self.linear3(q)
```

Listing 1: models.py

Learning

Referring to the pseudo code described in "Continuous Control With Deep Reinforcement Learning", for each step in the training process, the algorithm should:

1. Select an action based on actor network and exploration noise: $a_t = \mu(s_t|\theta^{\mu'})$.
2. Execute action a_t and observe new reward r_t and new state s_{t+1}.
3. Add new experience (s_t, a_t, r_t, s_{t+1}) into replay buffer.
4. Sample a minibatch of samples from replay buffer: (s_i, a_i, r_i, s_{i+1}).
5. Compute $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$.
6. Update the actor policy using the sampled policy gradient:

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

7. Update the target networks softly:

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

Several major parts and their implementation details are described below.

Generating exploration noise

Two methods for generating exploration noise are implemented here, the Ornstein-Ulhenbeck Process used in the original paper and simple Normal Distribution as a comparison counterpart to verify the necessity of the Ornstein-Ulhenbeck Process used in the original paper.

The implementation of the Ornstein-Ulhenbeck Process uses the version implemented in RLkit. And the implementation of Normal Distribution uses the function provided in Numpy. The concrete implementation is shown as Listing 2.

```
# Ornstein-Ulhenbeck Process
# https://github.com/vitchyr/rlkit/blob/master/rlkit/exploration_strategies/ou_strategy.py
class OUNoise(object):
    def __init__(self, action_space, mu=0.0, theta=0.15, max_sigma=0.2, min_sigma=0.2, decay_period=100000):
        self.mu           = mu
        self.theta        = theta
        self.sigma        = max_sigma
        self.max_sigma    = max_sigma
        self.min_sigma    = min_sigma
        self.decay_period = decay_period
        self.action_dim   = action_space.shape[0]
        self.low          = action_space.low
        self.high         = action_space.high
        self.reset()

    def reset(self):
        self.state = np.ones(self.action_dim) * self.mu

    def evolve_state(self):
        x  = self.state
        dx = self.theta * (self.mu - x) + self.sigma * np.random.randn(self.action_dim)
        self.state = x + dx
        return self.state

    def get_action(self, action, t=0):
        ou_state = self.evolve_state()
        self.sigma = self.max_sigma - (self.max_sigma - self.min_sigma) * min(1.0, t / self.decay_period)
        return np.clip(action + ou_state, self.low, self.high)

# normal distribution noise
def normal_noise_action(action, max_action, action_dim):
    # print(action, max_action)
    action = (action + np.random.normal(0, max_action * 0.1, size=action_dim)).clip(-max_action, max_action)
    return action
```

Listing 2: utils.py

Replay buffer

The Replay Buffer was originally proposed by DQN. I reference some DQN implementations, using an numpy array and a tail pointer to implement a fixed size queue as the Replay Buffer here. The concrete implementation is shown as Listing 3.

```
class ReplayBuffer(object):

    def __init__(self, state_dimension, action_dimension, max_size=int(1e6)):
        # initialize the parameter of the memory buffer
        self.capacity = max_size
        self.tailptr = 0
        self.size = 0

        # initialize the momory
        self.state = np.zeros((max_size, state_dimension))
        self.action = np.zeros((max_size, action_dimension))
        self.next_state = np.zeros((max_size, state_dimension))
        self.reward = np.zeros((max_size, 1))
        self.not_done = np.zeros((max_size, 1))

        self.device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

    # update the replay buffer with experience learnt by each step
    def update(self, state, action, next_state, reward, done):
        self.state[self.tailptr] = state
        self.action[self.tailptr] = action
        self.next_state[self.tailptr] = next_state
        self.reward[self.tailptr] = reward
        self.not_done[self.tailptr] = 1. - done

        self.tailptr = (self.tailptr + 1) % self.capacity
        self.size = min(self.size + 1, self.capacity)

    # randomly sample experience of batch_size
    def sample(self, batch_size):
        indice = np.random.randint(0, self.size, size=batch_size)
        return (
            torch.FloatTensor(self.state[indice]).to(self.device),
            torch.FloatTensor(self.action[indice]).to(self.device),
            torch.FloatTensor(self.next_state[indice]).to(self.device),
            torch.FloatTensor(self.reward[indice]).to(self.device),
            torch.FloatTensor(self.not_done[indice]).to(self.device)
        )
```

Listing 3: utils.py

Critic and actor learning

The update of the critic needs to approximate the new Q value calculated by the Bellman equation with next-state Q value calculated with the target critic and target

actor, that is, the gap between the original Q value and the new Q value calculated should be minimized.

The update goal of actor is to maximize estimated Q value, which is the mean of a mini-batch of estimated Q value.

The concrete implementation is shown as Listing 4. Again, there are different branches for different modes here.

```
# Q'(si+1, '(si+1))
if self.mode == 'bn':
    self.actor.eval()
    self.critic.eval()
    Q_target = self.critic(
        next_state, self.actor(next_state, mode=self.mode), mode=self.mode)
    self.actor.train()
    self.critic.train()
else:
    Q_target = self.target_critic(
        next_state, self.target_actor(next_state, mode=self.mode), mode=self.mode)
# ri+Q'(si+1, '(si+1))
Q_target = reward + (not_done * self.discount * Q_target).detach()
Q_value = self.critic(state, action, mode=self.mode)

# update critic
critic_loss = F.mse_loss(Q_value, Q_target)
self.critic_optimizer.zero_grad()
critic_loss.backward()
self.critic_optimizer.step()

# update actor
actor_loss = -self.critic(state, self.actor(state,  mode=self.mode), mode=self.mode).mean()
estimated_reward = (-actor_loss).detach()
self.actor_optimizer.zero_grad()
actor_loss.backward()
self.actor_optimizer.step()
```

Listing 4: agents.py

Target network updating

Finallyunlike hard updating in DQN, DDPG updates target network softly using τ to control update speed.The concrete implementation is shown as Listing 5.

```
def soft_update(current_network, target_network, tau):
    for current_param, target_param in zip(current_network.parameters(), target_network.parameters()):
        target_param.data.copy_(tau * current_param.data + (1.0 - tau) * target_param.data)
```

Listing 5: agents.py

Tabel 2: Runtime for each environment

Environment	Runtime
Pendulum-v1	2h 12m
LunarLanderContinuous-v2	2h 34m
Hopper-v2	2h 28m

Tabel 3: Performance after training across all environments for 1 million steps.

Environment	$R_{av,lowd}$	$R_{best,lowd}$	$R_{av,ctrl}$	$R_{best,ctrl}$
Pendulum-v1	-188.8200	-188.3583	-207.0115	-197.4855
LunarLanderContinuous-v2	293.5735	306.8242	-120.9383	-41.0218
Hopper-v2	3517.9395	3710.1439	1005.0862	1044.4058

Physical environment

In order to verify the generality of the conclusion in this paper, tasks in 3 different kinds of physical environment are selected for reproduction: Pendulum-v1, LunarLanderContinuous-v2 and Hopper-v2, respectively from Classic Control, Box2D and MuJoCo, action space of which have 1, 2 and 3 dimensions respectively.

Hyperparameters

The hyperparameters provided by Lillicrap et al.[1] are detailed, but two of them are needed to adjust by myself. The warm-up step used to initialize the replay buffer before starting training is not provided. I conduct serveral experiments and find that setting warmSteps=2500 works well. In addition, since the original paper was published in 2016, when the cost of using GPU was relatively high, the batch size in the paper was only set to 64. Through experiments, a good reward can be achieved in different tasks by setting batchSize=256.

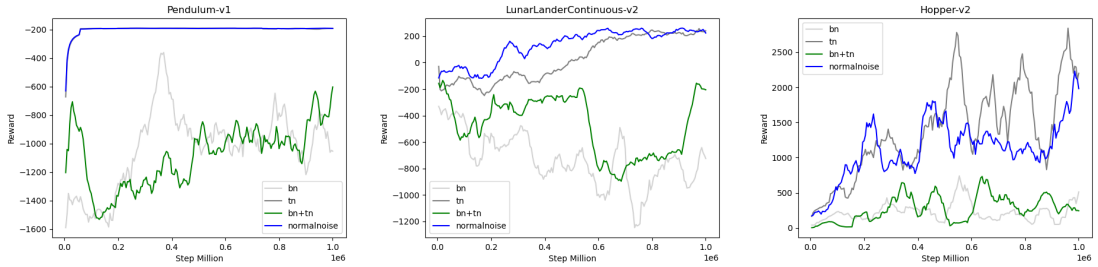
Experimental setup and code

The set up and complete code can be viewd here:

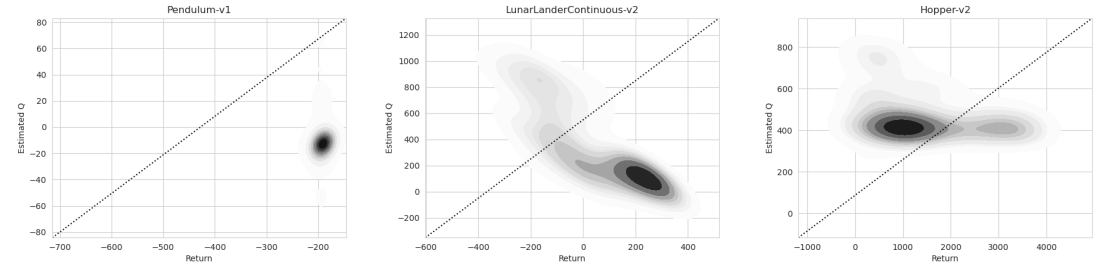
`https://github.com/EstherBear/Reproduction-of-Continuous-Control-with-Deep-Reinforcement-Learning-`

Computational requirements

I mostly used a TITAN Xp for the GPU and a Intel(R) Xeon(R) E5-2637 v4 for the CPU. The runtime of each physical environment was highly dependant on the architecture used. The runtimes for 1 million steps for each physical environment can be found in Tabel 2.



Figur 1: Performance curves for a selection of domains using variants of DPG



Figur 2: Density plot showing estimated Q values versus observed returns sampled from test episodes on 5 replicas

0.5 Results

Results reproducing original paper

DDPG can operate over continuous action spaces.

Tabel 3 summarizes DDPGs performance across all of the environments (results are averaged over 5 replicas). DDPG is able to learn good policies on various of tasks with continous action space in different environment including Classic control, Box2D and MuJoCo. And in all cases DDPG algorithn in the low-dimensional (lowd) version can learn policies which are superior to those found by original DPG algorithm with a replay buffer and batch normalization (cntrl).

Necessity of components of DDPG.

I evaluated the policy periodically (5000 steps) during training by testing it without exploration noise. Figur 1 shows the performance curve for the 3 chosen environments. I also report results with differnt combination of components of the DDPG algorithm (i.e. the target network, batch normalization and method of exploration noise generation, the details of the setting can be viewd in Tabel 1). In this section, let's focus on target network and batch normalization. I'll leave results of different methods of exploration noise generation for sektion 0.5.

In order to perform well across all tasks, target network are necessary. Learning without a target network, as in the original work with DPG, is very poor in many environments. However, batch normalization are not necessary and even hurt the performance which is inconsistent with the conclusion of the original paper.

Comparison of estimated Q values and observed returns.

It can be challenging to learn accurate value estimates. Q-learning, for example, is prone to over estimating values (Hasselt, 2010). I examined DDPG's estimates empirically by comparing the values estimated by Q (average over minibatch of samples) after training with the true returns seen on test episodes.

The original paper claim that "In simple tasks DDPG estimates returns accurately without systematic biases. For harder tasks the Q estimates are worse."However Figur 2 shows that accuracy has nothing to do with the complexity of the task. In the simplest task, Q value is over estimated while in the most complex task Q value is under estimated. In LunarLanderContinuous-v2, Q value is estimated accurately.

But as the original paper proposed, DDPG is still able to learn good policies even with inaccurate Q estimates.

Results beyond original paper

Necessity of exploration noise generated from Ornstein-Uhlenbeck process.

The original paper used an Ornstein-Uhlenbeck process (Uhlenbeck & Ornstein, 1930) with $\theta = 0.15$ and $\sigma = 0.2$ to generate exploration noise. The purpose of Ornstein-Uhlenbeck process is to generate temporally correlated noise in order to explore well in physical environments that have momentum.

However, as shown in Figur 1, the Ornstein-Uhlenbeck process is not necessary. A simple normal distribution can achieve comparable results with Ornstein-Uhlenbeck process in all 3 environments with inertia.

0.6 Discussion

DDPG can operate over continuous action spaces.

The claim that DDPG can learn policies in high-dimensional, continuous action spaces is supported by the results. I have trained DDPG in 3 environments with continous action space including Classic control, Box2D and MuJoCo. DDPG can be trained stably and converged to a good policy in all 3 environments. In this sense, DDPG can be applied generically and easily for different tasks with continous action space.

Necessity of components of DDPG.

The claim that components of DDPG is necessary is only partially supported by the results. There is no doubt that the target network is critical. However, the necessity of batch normalization and Ornstein-Uhlenbeck process is questionable according to

the results.

Batch normalization makes the training process unstable and hurt the performance of all 3 tasks severely. Unlike supervised training, the agent of reinforcement learning needs to interact with the environment through "experience replay"techniques to obtain training data that cannot be prepared before training. Therefore, reinforcement learning cannot provide enough stable training data for batch noramalization. Whenever the training data changes (the agent collects a large number of new states), batch noramalization cannot adapt to the new data, resulting in the collapse of the critic network and the actor network successively. And from the current point of view, the simple feed feedforward layer does not need batch noramalization. [4] also reports that results with batch normalization often are substantially worse than those without normalization.

Compared with the simple normal distribution, the Ornstein-Uhlenbeck process does not show advantages. According to Occam's Razor, it's better to use normal distribution with less parameters to be adjusted and less cost here.

Comparison of estimated Q values and observed returns.

The claim that DDPG estimates returns accurately without systematic biases in simple tasks and for harder tasks the Q estimates are worse is not supported by the results. But DDPG is still able to learn good policies regardless of the accuracy of estimated Q value.

The author draws a hasty conclusion about the relationship between complexity and estimated value without sufficient experiments (various tasks and plenty of episodes). This conclusion is easily disproved by the 3 counterexamples. However, it's not critical since the gap between estimated Q values and true returns have negligible influence on the quality of policy learnt by DDPG.

What was easy?

Overall I found DDPG quite difficult to understand and reproduce, but what did make it a lot easier was the different implementations of some techniques used by DDPG that were already online. For the DQN structure and Ornstein-Uhlenbeck process DDPG based on I found many useful implementations online that I could use.

- Different physical environments are easy to get access to with the gym package.
- Some techniques used by DDPG are well documented and can be found online.

What was difficult?

Some hyper-parameters were not included in the publication, making it difficult to properly replicate the results with sufficient precision. One of the main hyper-parameters that were missing were warm up steps. Without this hyper-parameter,

I have to conduct lots of experiments to find a proper one which is probably still different from the original paper's.

The computing methods used in some figures were not specified in detail, such as the method for calculating estimated Q values. So I can only guess at the details. For example, I take the average value of a mini-batch to obtain estimated value purely by guessing which may lead to inaccurate conclusions.

Additionally, some physical environments with continuous space presented in the origin paper are not open source. So I can only conduct experiments on limited environments.

Litteratur

[1] Lillicrap T P, Hunt J J, Pritzel A, et al. Continuous control with deep reinforcement learning[J]. arXiv preprint arXiv:1509.02971, 2015.

[2] Mnih V, Kavukcuoglu K, Silver D, et al. Human-level control through deep reinforcement learning[J]. nature, 2015, 518(7540): 529-533.

[3] Silver D, Lever G, Heess N, et al. Deterministic policy gradient algorithms[C]//International conference on machine learning. PMLR, 2014: 387-395.

[4] Bhatt A, Argus M, Amiranashvili A, et al. Crossnorm: Normalization for off-policy td reinforcement learning[J]. arXiv preprint arXiv:1902.05605, 2019.

[5] Uhlenbeck G E, Ornstein L S. On the theory of the Brownian motion[J]. Physical review, 1930, 36(5): 823.