Reproducibility of CONTINUOUS CONTROL WITH DEEP REINFORCEMENT LEARNING

丛鹏彧20337185

ABSTRACT

In this report, the main works are: We will compare these two noises in our reproduction. I will constructe a variety of challenging physical control problems to evaluate the method DDPG which is raised in the paper.

1 INTRODUCTION

DQN solves problems with high-dimensional observation spaces, However, as shown in the previous literature ,it can only handle discrete and low-dimensional action spaces. Many tasks of interest, most notably physical control tasks, have continuous (real valued) and high dimensional action spaces. DQN cannot be straight forwardly applied to continuous domains since it relies on a fifinding the action that maximizes the action-value function, which in the continuous valued case requires an iterative optimization process at every step.

To overcome the challenge,the literature provides model-free approach which we call Deep DPG (DDPG) can learn competitive policies for all of our tasks using low-dimensional observations (e.g. cartesian coordinates or joint angles) using the same hyper-parameters and network structure. In many cases, we are also able to learn good policies directly from pixels, again keeping hyperparameters and network structure constant 1 . A key feature of the approach is its simplicity: it requires only a straightforward actor-critic architecture and learning algorithm with very few “moving parts”, making it easy to implement and scale to more diffificult problems and larger networks. For the physical control problems we compare our results to a baseline computed by a planner (Tassa et al., 2012) that has full access to the underlying simulated dynamics and its derivatives (see supplementary information). Interestingly, DDPG can sometimes fifind policies that exceed the performance of the planner, in some cases even when learning from pixels (the planner always plans over the underlying low-dimensional state space).

In this paper,i accomplish the Reproducibility Reports of'CONTINUOUS CONTROL WITH DEEP REINFORCEMENT LEARNING '

2 BACKGROUND

In the paper, author pointed out that DPG (DDPG) can learn competitive policies for all of our tasks using low-dimensional observations using the same hyper-parameters and network structure. In many cases, we are also able to learn good policies directly from pixels, again keeping hyperparameters and network structure constant. DDPG can sometimes find policies that exceed the performance of the planner.

In this report, the main works are: We will compare these two noises in our reproduction. I will constructe a variety of challenging physical control problems to evaluate the method DDPG which is raised in the paper.

3 APPROACH

3.1 ALGORITHM

DPG

It is not possible to straightforwardly apply Q-learning to continuous action spaces, because in continuous spaces finding the greedy policy requires an optimization of a at every timestep; this optimization is too slow to be practical with large, unconstrained function approximators and nontrivial action spaces. Instead, here we used an actor-critic approach based on the DPG algorithm

The DPG algorithm maintains a parameterized actor function $\mu(s|\theta\,\mu)$ which specifiies the current policy by deterministically mapping states to a specifific action. The critic $Q(s, a)$ is learned using the Bellman equation as in Q-learning. The actor is updated by applying the chain rule to equation 3 with respect to the actor parameters:

$$\begin{aligned}\nabla_{\theta^\mu}\mu &\approx \mathbb{E}_{\mu'}\left[\nabla_{\theta^\mu}Q(s,a|\theta^Q)|_{s=s_t,a=\mu(s_t|\theta^\mu)}\right]\\ &= \mathbb{E}_{\mu'}\left[\nabla_a Q(s,a|\theta^Q)|_{s=s_t,a=\mu(s_t)}\nabla_{\theta_\mu}\mu(s|\theta^\mu)|_{s=s_t}\right]\end{aligned}$$

DDPG

DDPG algorithm is an online deep reinforcement learning algorithm under the Actor-Critic (AC) framework. Therefore, the algorithm includes Actor network and Critic network, and each network updates according to its own update law, so as to maximize the cumulative expected return. DDPG algorithm is an online deep reinforcement learning algorithm under the Actor-Critic (AC) framework. Therefore, the algorithm includes Actor network and Critic network, and each network updates according to its own update law, so as to maximize the cumulative expected return.

As with Q learning, introducing non-linear function approximators means that convergence is no longer guaranteed. However, such approximators appear essential in order to learn and generalize on large state spaces. NFQCA (Hafner & Riedmiller, 2011), which uses the same update rules as DPG but with neural network function approximators, uses batch learning for stability, which is intractable for large networks. A minibatch version of NFQCA which does not reset the policy at each update, as would be required to scale to large networks, is equivalent to the original DPG, which we compare to here. Our contribution here is to provide modifications to DPG, inspired by the success of DQN, which allow it to use neural network function approximators to learn in large state and action spaces online. We refer to our algorithm as Deep DPG.

As in DQN, we used a replay buffer to address these issues. The replay buffer is a fifinite sized cache *R*. Transitions were sampled from the environment according to the exploration policy and the tuple (***st**, **at**, **rt**, s***t*+1) was stored in the replay buffer. When the replay buffer was full the oldest samples were discarded. At each timestep the actor and critic are updated by sampling a minibatch uniformly from the buffer. Because DDPG is an off-policy algorithm, the replay buffer can be large, allowing the algorithm to benefifit from learning across a set of transitions uncorrelated transitions.

When learning from low dimensional feature vector observations, the different components of the observation may have different physical units (for example, positions versus velocities) and the ranges may vary across environments. This can make it diffifficult for the network to learn effectively and may make it diffifficult to fifind hyper-parameters which generalise across

environments with different scales of state values

A major challenge of learning in continuous action spaces is exploration. An advantage of off policies algorithms such as DDPG is that we can treat the problem of exploration independently from the learning algorithm. We constructed an exploration policy μ by adding noise sampled from a noise process N to our actor policy

$$\mu'(s_t) = \mu(s_t|\theta_t^\mu) + \mathcal{N}$$

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 gradient:

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

    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

```
class DDPG(object):
    def __init__(self, state_dim, action_dim, device, env, args):
        self.actor = Actor(state_dim, action_dim).to(device)
        self.actor_target = Actor(state_dim, action_dim).to(device)
        self.critic = Critic(state_dim, action_dim).to(device)
        self.critic_target = Critic(state_dim, action_dim).to(device)
        self.actor_optim = torch.optim.Adam(self.actor.parameters(),
lr=args.actor_lr)
        self.critic_optim = torch.optim.Adam(self.critic.parameters(),
lr=args.critic_lr)
        self.replay_memory = Replay_Memory(args.pool_size, args.batch_size)
        self.args = args
        self.device = device
        self.actor_target.load_state_dict(self.actor.state_dict())
        self.critic_target.load_state_dict(self.critic.state_dict())
        self.env = env
        self.noise = None
        self.eps = args.eps
        if args.noise == "OU":
            self.noise = OUNoise(action_dim)
        elif args.noise == "Gauss":
```

```
            self.noise = GaussNoise(action_dim)
```

3.2 Model

Empirical replay

Empirical replay is a technique to stabilize the empirical probability distribution and improve the stability of training. There are two key steps of experiential playback: "storage" and "playback".

Storage: will experience to $(s**i, a**i, r**i, s**i+1)$is stored in the experience in the pool.

Playback: Sampling one or more experience data from the experience pool according to certain rules.

```
class ReplayMemory(object):
    def __init__(self, pool_size, batch_size):
        self.pool = []
        self.max_size = pool_size
        self.index = 0
        self.batch_size = batch_size

    def append(self, state, action, reward, next_state, done):
        if len(self.pool) != self.max_size:
            self.pool.append((state, action, reward, next_state, done))
            self.index = (self.index + 1) % self.max_size
        else:
            self.pool[self.index] = (state, action, reward, next_state, done)
            self.index = (self.index + 1) % self.max_size

    def sample(self):
        rand_indexs = random.sample(range(0, len(self.pool)), self.batch_size)
        rand_data = [self.pool[i] for i in rand_indexs]
        return rand_dataclass ReplayBuffer:
```

target network

The algorithm update mainly updates the parameters of Actor network and Critic network, among which the Actor network updates by maximizing the cumulative expected return and the Critic network updates by minimizing the error between the evaluation value and the target value

Actor network update :

```
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
```

Critic network update :

```
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
```

For update of target networks, DDPG algorithm adopts a soft update approach, which is also called Exponential Moving Average (EMA). That is, a learning rate (or momentum) is introduced to make a weighted average of the old target network parameters and the new corresponding network parameters, and then assign a value to the target network.

Target Actor network update :

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

Target Critic network update :

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

Noise exploration

Exploration is essential for agents, and deterministic strategies "naturally" lack exploration, so we need to artificially add noise to the output action to make the agents capable of exploration .

we can choose two different types of action noise .

GaussNoise:

```
class GaussNoise:
    def __init__(self, action_dimension, mu=0, sigma=0.2):
        self.action_dimension = action_dimension
        self.mu = mu
        self.sigma = sigma

    def noise(self):
        tmp = np.random.normal(self.mu, self.sigma, size=self.action_dimension)
        return tmp
```

OUNoise:

```
class OUNoise:
    def __init__(self, action_dimension, mu=0, theta=0.15, sigma=0.2):
        self.action_dimension = action_dimension
        self.mu = mu
        self.theta = theta
```

```
        self.sigma = sigma
        self.state = np.ones(self.action_dimension) * self.mu
        self.reset()

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

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

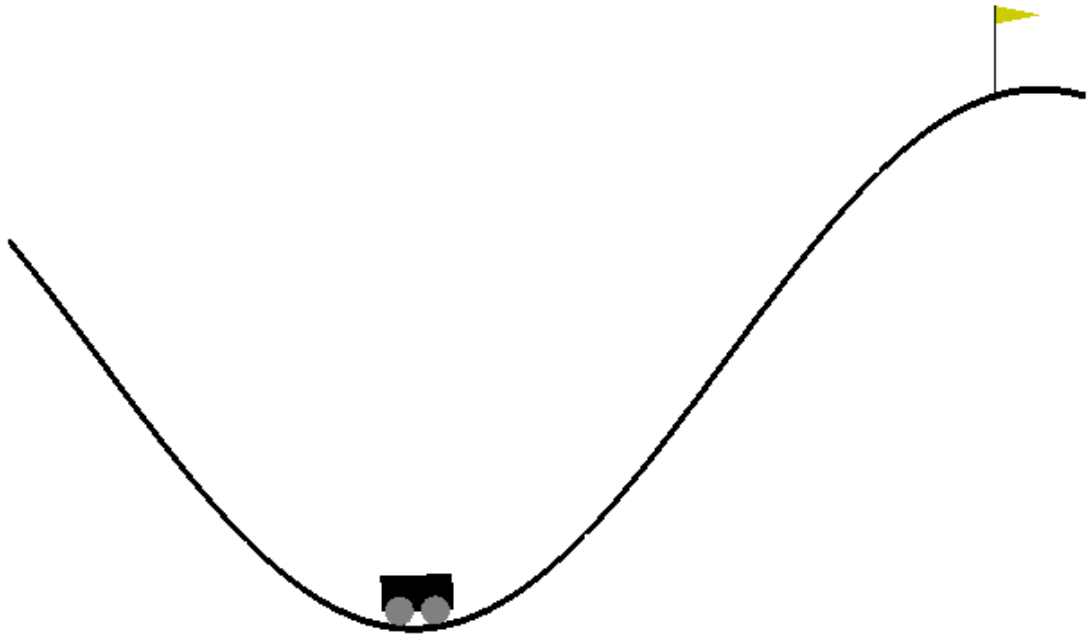
3.3 Test Game

Gym

Using 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.

Mountain Car

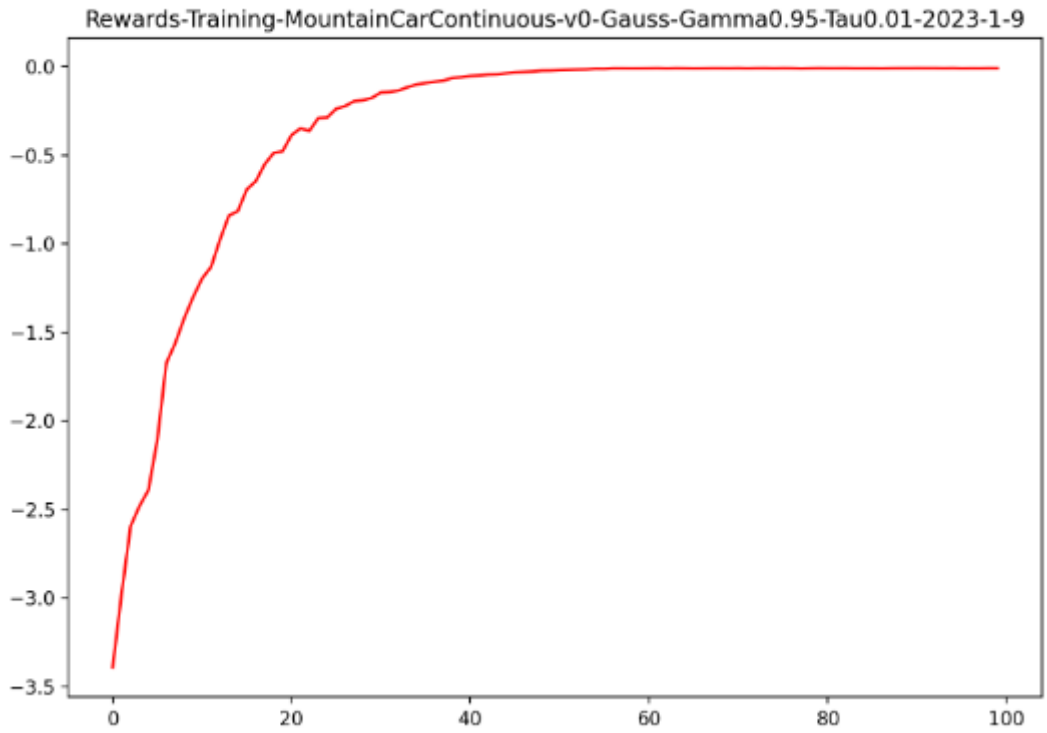
A car is on a one-dimensional track, positioned between two "mountains". The goal is to drive up the mountain on the right; however, the car's engine is not strong enough to scale the mountain in a single pass. Therefore, the only way to succeed is to drive back and forth to build up momentum. The reward is greater if you spend less energy to reach the goal.

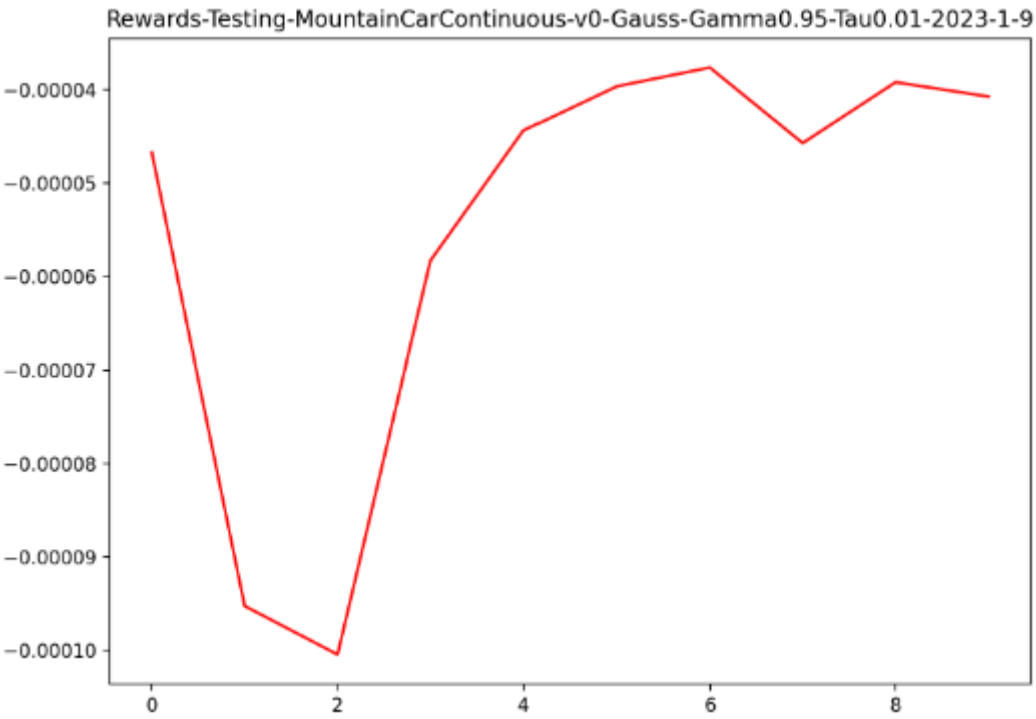


4 RESULTS

At the beginning the car almost remained in the valley initially, and after some learning, it can achieve the goal perfectly.

MountainCar rewards with Gauss in training and testing:





It is proved that the DDPG algorithm achieves good results in the continuous control problem.

5 CONCLUSION AND DISCUSSIONS

This report reproduces the principle of DDPG algorithm and code implementation. DDPG is a pioneering algorithm in reinforce learning. It spans the control problem in the continuous space and give a practice solution. In many fifields, DDPG can play a vital role, such as robot control, automatic driving, modern team

game and so on. Although it is a very good algorithm, there are still some problems that need to be improved, such as the overestimation problem, and many optional details, such as types of action noise.

References

[1] Timothy P. Lillicrap,CONTINUOUS CONTROL WITH DEEP REINFORCEMENT LEARNING

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