DDPG

Introduction：

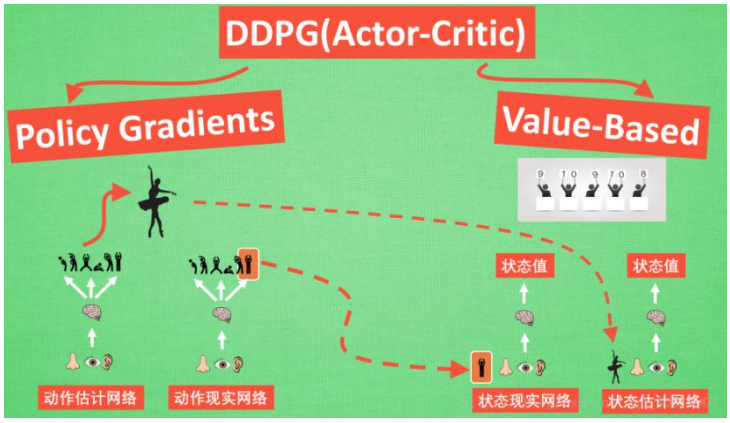
Deep：首先 Deep 我们都知道，就是更深层次的网络结构，我们之前在 DQN 中使用两个网络与经验池的结构，在 DDPG 中就应用了这种思想。但 DQN 的缺陷在于只适用于离散低维动作空间。

Policy Gradient：顾名思义就是策略梯度算法，能够在连续的动作空间根据所学习到的策略（动作分布）随机筛选动作。

Deterministic：它的作用就是用来帮助 Policy Gradient 不让他随机选择，只输出一个动作值。在连续动作空间中动作是无穷大，要从中进行输出每个动作的概率是不可能的，因此对于一个状态，需要输出最可能的动作。

- 随机性策略，∑π(a|s)=1，∑π(a|s)=1 策略输出的是动作的概率，使用正态分布对动作进行采样选择，即每个动作都有概率被选到；优点，将探索和改进集成到一个策略中；缺点，需要大量训练数据。
- 确定性策略，π(s)，S→A，策略输出即是动作；优点，需要采样的数据少，算法效率高；缺点，无法探索环境。然而因为我们引用了 DQN 的结构利用 off Policy 采样，这样就解决了无法探索环境的问题，还有对采取的动作增加一点噪声也能探索。

从 DDPG 网络整体上来说：他应用了 Actor-Critic 形式的，所以也具备策略 Policy 的神经网络 和基于 价值 Value 的神经网络，因为引入了 DQN 的思想，每种神经网络我们都需要再细分为两个，Policy Gradient 这边，我们有估计网络和现实网络，估计网络用来输出实时的动作，供 actor 在现实中实行，而现实网络则是用来更新价值网络系统的。再看另一侧价值网络，我们也有现实网络和估计网络，他们都在输出这个状态的价值，而输入端却有不同，状态现实网络这边会拿着从动作现实网络来的动作加上状态的观测值加以分析，而状态估计网络则是拿着当时 Actor 施加的动作当做输入。



DDPG 在连续动作空间的任务中效果优于 DQN 而且收敛速度更快，但是不适用于随机环境问题。

公式推导：

我们来说说 Critic 这边，Critic 这边的学习过程跟 DQN 类似，我们都知道 DQN 根据下面的损失函数来进行网络学习，即现实的 Q 值和估计的 Q 值的平方损失：

$$R + \gamma \max_a Q(S', a) - Q(S, A)$$

上面式子中 Q(S,A)是根据状态估计网络得到的，A 是动作估计网络传过来的动作。而前面部分 R + gamma * maxQ(S',A')是现实的 Q 值，这里不一样的是，我们计算现实的 Q 值，不在使用贪心算法，来选择动作 A',而是动作现实网络得到这里的 A'。总的来说，Critic 的状

态估计网络的训练还是基于现实的 Q 值和估计的 Q 值的平方损失，估计的 Q 值根据当前的状态 S 和动作估计网络输出的动作 A 输入状态估计网络得到，而现实的 Q 值根据现实的奖励 R，以及将下一时刻的状态 S'和动作现实网络得到的动作 A' 输入到状态现实网络 而得到的 Q 值的折现值加和得到(这里运用的是贝尔曼方程)。

再说一下 Actor 部分，

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

这个式子看上去很吓人，但是其实理解起来很简单。假如对同一个状态，我们输出了两个不同的动作 a1 和 a2，从状态估计网络得到了两个反馈的 Q 值，分别是 Q1 和 Q2，假设 Q1>Q2,即采取动作 1 可以得到更多的奖励，那么 Policy gradient 的思想是什么呢，就是增加 a1 的概率，降低 a2 的概率，也就是说，Actor 想要尽可能的得到更大的 Q 值。所以我们的 Actor 的损失可以简单的理解为得到的反馈 Q 值越大损失越小，得到的反馈 Q 值越小损失越大，因此只要对状态估计网络返回的 Q 值取个负号就好啦。是不是很简单。

2、公式推导

再来啰唆一下前置公式

s_t：在t时刻，agent所能表示的环境状态，比如观察到的环境图像，agent在环境中的位置、速度、机器人关节角度等；

a_t：在t时刻，agent选择的行为（action）

$r(s_t,a_t)$：函数: 环境在状态st 执行行为at后，返回的单步奖励值；

R_t：是从当前状态直到将来某个状态中间所有行为所获得奖励值的之和当然下一个状态的奖励值要有一个衰变系数 γ 一般情况下可取0到1的小数

$R_t = \sum_{i=t}\gamma^{i-t} r(s_i,a_i)$

Policy Gradient:

通过概率的分布函数确定最优策略，在每一步根据该概率分布获取当前状态最佳的动作，产生动作采取的是随机性策略

$a_t \sim \pi_\theta(s_t|\theta^\pi)$

目标函数：$J(\pi_\theta) = \int_S \rho^\pi(s)\int_A \pi_\theta(s,a)r(s,a)dads = E_{s\sim\rho^\pi,a\sim\pi_\theta}[r(s,a)]$（注意dads不是什么未知的符号，而是积分的 da ds)

梯度：$\nabla_\theta J(\pi_\theta) = \int_S \rho^\pi(s)\int_A \nabla_\theta\pi_\theta(s,a)Q^\pi(s,a)dads = E_{s\sim\rho^\pi,a\sim\pi_\theta}[\nabla_\theta log\pi_\theta(a|s)Q^\pi(s,a)]$

Deterministic Policy Gradient:

因为Policy Gradient是采取随机性策略，所以要想获取当前动作action就需要对最优策略的概率分布进行采样，而且在迭代过程中每一步都要对整个动作空间进行积分，所以计算量很大

在PG的基础上采取了确定性策略，根据行为直接通过函数μ确定了一个动作，可以吧μ理解成一个最优行为策略

$a_t = \mu(s_t|\theta^\mu)$

performance objective为

$J(\mu_\theta) = \int_S \rho^\mu(s)r(s,\mu_\theta(s))ds$

$J(\mu_\theta) = E_{s\sim\rho^\mu}[r(s,\mu_\theta(s))]$

deterministic policy梯度

$\bigtriangledown_\theta J(\mu_\theta) = \int_S \rho^\mu(s)\bigtriangledown\theta\mu_\theta(s)Q^\mu(s,a)|_{a=\mu_\theta}ds = E_{s\sim\rho^\beta}[\bigtriangledown\theta\mu_\theta(s)Q^\mu(s,a)|_{a=\mu_\theta}]$

DDPG 就是用了**确定性策略**在 DPG 基础上结合 DQN 的特点建议改进出来的算法。

Deep Deterministic Policy Gradient

所以基于上述两种算法

DDPG采用确定性策略μ来选取动作 $a_t = \mu(s_t|\theta^\mu)$ 其中θ^μ是产生确定性动作的策略网络的参数。根据之前提到过的AC算与PG算法我们可以想到，使用策略网络μ来充当actor，使用价值网络来拟合(s,a)函数，来充当critic的角色，所以将DDPG的目标函数就可以定义为

$$J(\theta^\mu) = E_{\theta^\mu}[r_1 + \gamma r_2 + \gamma^2 r_3 + \cdots]$$

此时Q函数表示为在采用确定性策略μ下选择动作的奖励期望值，在DDPG我们就采用DQN的结构使用Q网络来拟合Q函数

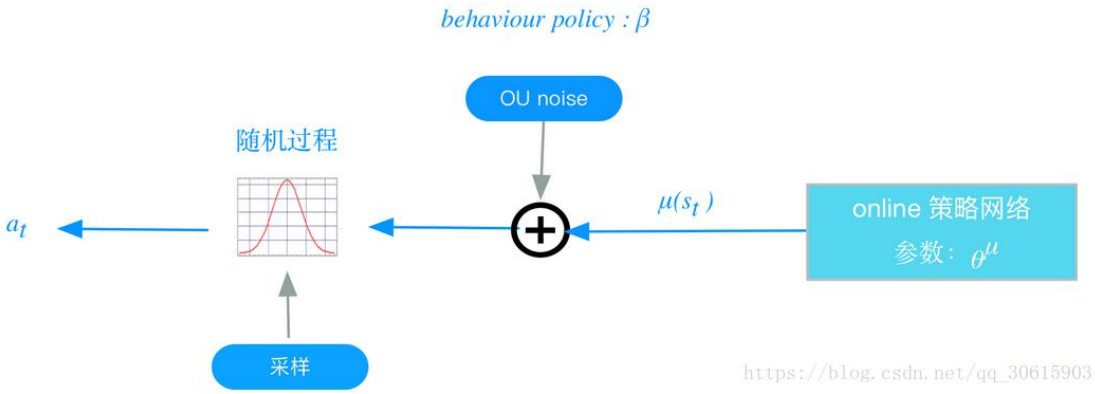
$$Q^\mu(s_t,a_t) = E[r(s_t,a_t) + \gamma Q^\mu(s_{t+1},\mu(s_{t+1}))]$$

Q网络中的参数定义为θ^Q，$Q^\mu(s,\mu(s))$表示使用μ策略在s状态选选取动作所获取的回报期望值，又因为是在连续空间内所以期望可用积分来求，则可以使用下式来表示策略μ的好坏

$$J_\beta(\mu) = \int_S \rho^\beta(s)Q^\mu(s,\mu(s))ds = E_{s\sim\rho^\beta}[Q^\mu(s,\mu(s)]$$

behavior policy β: 在常见的RL训练过程中存在贪婪策略来平衡exploration和exploit与之类似，在DDPG中使用Uhlenbeck-Ornstein随机过程（下面简称UO过程），作为引入的随机噪声：UO过程在时序上具备很好的相关性，可以使agent很好的探索具备动量属性的环境exploration的目的是探索潜在的更优策略，所以训练过程中，我们为action的决策机制引入随机噪声：

过程如下图所示：



Silver大神证明了目标函数采用μ策略的梯度与Q函数采用μ策略的期望梯度是等价的：

$$\frac{\partial J(\theta^{\mu})}{\partial \theta^{\mu}} = E_s[\frac{\partial Q(s,a|\theta^Q)}{\partial \theta^{\mu}}]$$

因为是确定性策略$a = \mu(s|\theta^{\mu})$所以得到Actor网络的梯度为

$$\frac{\partial J(\theta^{\mu})}{\partial \theta^{\mu}} = E_s[\frac{\partial Q(s,a|\theta^Q)}{\partial a}\frac{\partial \pi(s|\theta^{\mu})}{\partial \theta^{\mu}}]$$

$$\bigtriangledown_{\theta} J_{\beta}(\mu_{\theta}) = \int_S \rho^{\beta}(s) \bigtriangledown_{\theta} \mu_{\theta}(s) Q^{\mu}(s,a)|_{a=\mu_{\theta}} ds = E_{s\sim\rho^{\beta}}[\bigtriangledown_{\theta}\mu_{\theta}(s) Q^{\mu}(s,a)|_{a=\mu_{\theta}}]$$

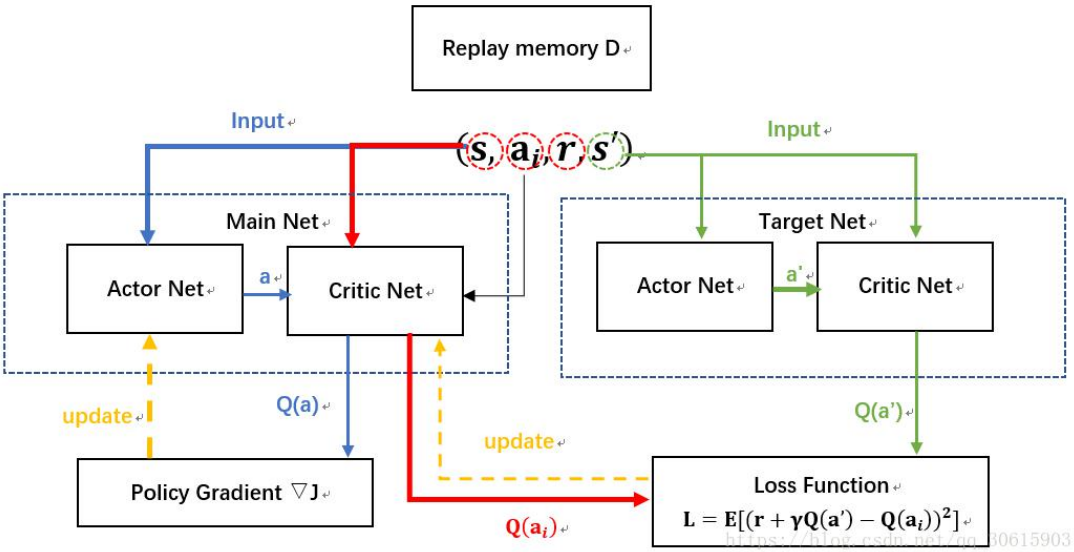
在另一方面Critic网络上的价值梯度为

$$\frac{\partial L(\theta^Q)}{\partial \theta^Q} = E_{s,a,r,s'\sim D}[(TargetQ - Q(s,a|\theta^Q)\frac{\partial Q(s,a|\theta^Q)}{\partial \theta^Q}]$$

$$TargetQ = r + \gamma Q'(s', \pi(s'|\theta^{\mu'})|\theta^{Q'})$$

损失函数采取均方误差损失MSE，另外在计算策略梯度期望的时候仍然选择蒙特卡罗法来取无偏估计（随机采样加和平均法）

我们有了上述两个梯度公式就可以使用梯度下降进行网络的更新
网络结构图如下因为引用了DQN的结构，所以就多了一个target网络



在 DDPG 中，采用的是一种 soft 模式的 target-net 网络参数更新，即每一步都对 target-net 网络中的参数更新一点点，这种参数更新方式经过试验表明可以大大的提高学习的稳定性。

论文中提到的另一个小 trick 是对采取的动作增加一定的噪声。

Background：

这里，我们假定一个完全观察环境 xt=st，实用 MDP 过程分析。

RL 的目标是从初始化分布 J 开始找到一个策略使得最大化期望累计回报。

期望依赖于环境，意味着学习由随机策略β产生的带有转移方法的 Qμ是可能的。

经验回放方法的使用使得 target 网络能独立计算 Q 和 V。

Algorithm:

DPG 维护一个参数化动作函数μ(s|θμ)使得当前策略下能确定的从一个状态中选择动作。

使用链式法则，将 DPG 的目标函数化为以下形式：

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

带有小 batch 版本的 NFQCA 不会在每一次更新时重设策略，它需要扩展到大的网络，它等价于原始 DPG。结合 DQN 优点的 DPG 算法即 DDPG，流程如下：

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

大多数优化都假定样本独立，但其实 trajectory 中的动作不会是独立的，因此破坏了许多优化，使用经验回放技巧可以尽可能抓住数据的独立性，是有限大小的 cache，buffer 满了会解散以前的数据，一般被小批量利用，DDPG 也使用这一技巧。

使用 soft 更新方法，每一步都更新一点点代替固定回合直接复制参数更新 target Net 减少不稳定性和发散。这意味着 target Net 的参数会受限于缓慢更新，以改善其稳定性。

在实验中，先将特征小批量归一化处理，这样可以最小化协方差。

在连续动作空间中有个关键的挑战的就是探索阶段。DDPG 将探索和学习分开，设计一个 μ '= μ +N 作为探索策略，即在动作选择上加一些噪声。这种噪声也需要一定技巧性。

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

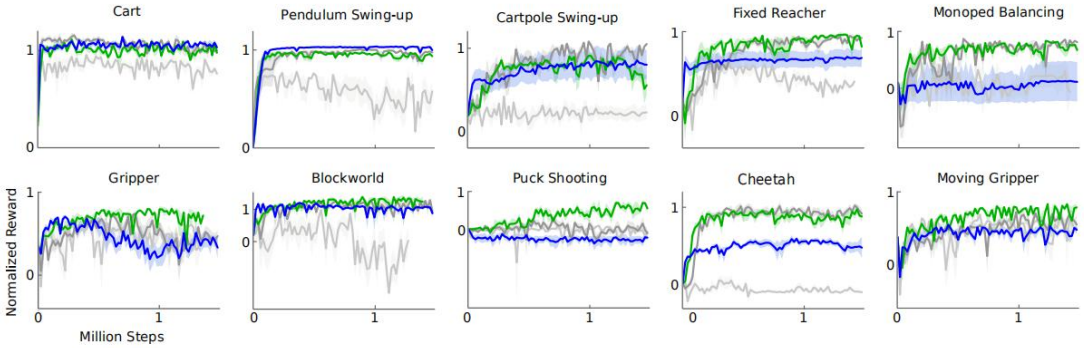


Figure 2: Performance curves for a selection of domains using variants of DPG: original DPG algorithm (minibatch NFQCA) with batch normalization (light grey), with target network (dark grey), with target networks and batch normalization (green), with target networks from pixel-only inputs (blue). Target networks are crucial.

environment	$R_{av,lowd}$	$R_{best,lowd}$	$R_{av,pix}$	$R_{best,pix}$	$R_{av,cntrl}$	$R_{best,cntrl}$
blockworld1	1.156	1.511	0.466	1.299	-0.080	1.260
blockworld3da	0.340	0.705	0.889	2.225	-0.139	0.658
canada	0.303	1.735	0.176	0.688	0.125	1.157
canada2d	0.400	0.978	-0.285	0.119	-0.045	0.701
cart	0.938	1.336	1.096	1.258	0.343	1.216
cartpole	0.844	1.115	0.482	1.138	0.244	0.755
cartpoleBalance	0.951	1.000	0.335	0.996	-0.468	0.528
cartpoleParallelDouble	0.549	0.900	0.188	0.323	0.197	0.572
cartpoleSerialDouble	0.272	0.719	0.195	0.642	0.143	0.701
cartpoleSerialTriple	0.736	0.946	0.412	0.427	0.583	0.942
cheetah	0.903	1.206	0.457	0.792	-0.008	0.425
fixedReacher	0.849	1.021	0.693	0.981	0.259	0.927
fixedReacherDouble	0.924	0.996	0.872	0.943	0.290	0.995
fixedReacherSingle	0.954	1.000	0.827	0.995	0.620	0.999
gripper	0.655	0.972	0.406	0.790	0.461	0.816
gripperRandom	0.618	0.937	0.082	0.791	0.557	0.808
hardCheetah	1.311	1.990	1.204	1.431	-0.031	1.411
hopper	0.676	0.936	0.112	0.924	0.078	0.917
hyq	0.416	0.722	0.234	0.672	0.198	0.618
movingGripper	0.474	0.936	0.480	0.644	0.416	0.805
pendulum	0.946	1.021	0.663	1.055	0.099	0.951
reacher	0.720	0.987	0.194	0.878	0.231	0.953
reacher3daFixedTarget	0.585	0.943	0.453	0.922	0.204	0.631
reacher3daRandomTarget	0.467	0.739	0.374	0.735	-0.046	0.158
reacherSingle	0.981	1.102	1.000	1.083	1.010	1.083
walker2d	0.705	1.573	0.944	1.476	0.393	1.397
torcs	-393.385	1840.036	-401.911	1876.284	-911.034	1961.600

Table 1: Performance after training across all environments for at most 2.5 million steps. We report both the average and best observed (across 5 runs). All scores, except Torcs, are normalized so that a random agent receives 0 and a planning algorithm 1; for Torcs we present the raw reward score. We include results from the DDPG algorithn in the low-dimensional ($lowd$) version of the environment and high-dimensional (pix). For comparision we also include results from the original DPG algorithm with a replay buffer and batch normalization ($cntrl$).

Results：

In all tasks, we ran experiments using both a low-dimensional state description (such as joint angles and positions) and high-dimensional renderings of the environment.

Related Work：

DPG 拥有数据有效性的优点，同时 off-policy、on-policy 都可以适用。但是 DDPG 将其扩展，使其可以适用于高维复杂的状态和动作空间。

标准 PG 难以扩展到复杂问题，学习不稳定或者学习过慢。

TRPO 使用近似单调学习 policy，而不是 Q 动作，因此数据有效性较弱。

Conclusion：

DDPG 也没有严谨的理论保证来表明网络的非线性近似器可以收敛，只是用实验证明。

DDPG 主要解决连续动作空间和高维负责问题和未处理的图像输入。

DDPG 缺陷，收敛速度慢，

Source Code：

```
import tensorflow as tf
import numpy as np
import gym
import time


#####################  hyper parameters  ####################

MAX_EPISODES = 200
MAX_EP_STEPS = 200
LR_A = 0.001        # learning rate for actor
LR_C = 0.002        # learning rate for critic
GAMMA = 0.9         # reward discount
TAU = 0.01          # soft replacement
MEMORY_CAPACITY = 10000
BATCH_SIZE = 32

RENDER = False
ENV_NAME = 'Pendulum-v0'


###############################  DDPG  ####################################


class DDPG(object):
    def __init__(self, a_dim, s_dim, a_bound,):
        self.memory = np.zeros((MEMORY_CAPACITY, s_dim * 2 + a_dim + 1), dtype=np.float32)
```

```
        self.pointer = 0
        self.sess = tf.Session()

        self.a_dim, self.s_dim, self.a_bound = a_dim, s_dim, a_bound,
        self.S = tf.placeholder(tf.float32, [None, s_dim], 's')
        self.S_ = tf.placeholder(tf.float32, [None, s_dim], 's_')
        self.R = tf.placeholder(tf.float32, [None, 1], 'r')

        self.a = self._build_a(self.S,)
        q = self._build_c(self.S, self.a, )
        a_params = tf.get_collection(tf.GraphKeys.TRAINABLE_VARIABLES, scope='Actor')
        c_params = tf.get_collection(tf.GraphKeys.TRAINABLE_VARIABLES, scope='Critic')
        ema = tf.train.ExponentialMovingAverage(decay=1 - TAU)          # soft replacement

        def ema_getter(getter, name, *args, **kwargs):
            return ema.average(getter(name, *args, **kwargs))

        target_update = [ema.apply(a_params), ema.apply(c_params)]      # soft update operation
        a_ = self._build_a(self.S_, reuse=True, custom_getter=ema_getter)   # replaced target parameters
        q_ = self._build_c(self.S_, a_, reuse=True, custom_getter=ema_getter)

        a_loss = - tf.reduce_mean(q)    # maximize the q
        self.atrain = tf.train.AdamOptimizer(LR_A).minimize(a_loss, var_list=a_params)

        with tf.control_dependencies(target_update):    # soft replacement happened at here
            q_target = self.R + GAMMA * q_
            td_error = tf.losses.mean_squared_error(labels=q_target, predictions=q)
            self.ctrain = tf.train.AdamOptimizer(LR_C).minimize(td_error, var_list=c_params)

        self.sess.run(tf.global_variables_initializer())

    def choose_action(self, s):
        return self.sess.run(self.a, {self.S: s[np.newaxis, :]})[0]

    def learn(self):
        indices = np.random.choice(MEMORY_CAPACITY, size=BATCH_SIZE)
        bt = self.memory[indices, :]
        bs = bt[:, :self.s_dim]
        ba = bt[:, self.s_dim: self.s_dim + self.a_dim]
        br = bt[:, -self.s_dim - 1: -self.s_dim]
        bs_ = bt[:, -self.s_dim:]
```

```
            self.sess.run(self.atrain, {self.S: bs})
            self.sess.run(self.ctrain, {self.S: bs, self.a: ba, self.R: br, self.S_: bs_})

        def store_transition(self, s, a, r, s_):
            transition = np.hstack((s, a, [r], s_))
            index = self.pointer % MEMORY_CAPACITY    # replace the old memory with new memory
            self.memory[index, :] = transition
            self.pointer += 1

        def _build_a(self, s, reuse=None, custom_getter=None):
            trainable = True if reuse is None else False
            with tf.variable_scope('Actor', reuse=reuse, custom_getter=custom_getter):
                net = tf.layers.dense(s, 30, activation=tf.nn.relu, name='l1', trainable=trainable)
                a = tf.layers.dense(net, self.a_dim, activation=tf.nn.tanh, name='a', trainable=trainable)
                return tf.multiply(a, self.a_bound, name='scaled_a')

        def _build_c(self, s, a, reuse=None, custom_getter=None):
            trainable = True if reuse is None else False
            with tf.variable_scope('Critic', reuse=reuse, custom_getter=custom_getter):
                n_l1 = 30
                w1_s = tf.get_variable('w1_s', [self.s_dim, n_l1], trainable=trainable)
                w1_a = tf.get_variable('w1_a', [self.a_dim, n_l1], trainable=trainable)
                b1 = tf.get_variable('b1', [1, n_l1], trainable=trainable)
                net = tf.nn.relu(tf.matmul(s, w1_s) + tf.matmul(a, w1_a) + b1)
                return tf.layers.dense(net, 1, trainable=trainable)    # Q(s,a)


##################################    training    ####################################

env = gym.make(ENV_NAME)
env = env.unwrapped
env.seed(1)

s_dim = env.observation_space.shape[0]
a_dim = env.action_space.shape[0]
a_bound = env.action_space.high

ddpg = DDPG(a_dim, s_dim, a_bound)

var = 3    # control exploration
t1 = time.time()
```

```
for i in range(MAX_EPISODES):
    s = env.reset()
    ep_reward = 0
    for j in range(MAX_EP_STEPS):
        if RENDER:
            env.render()

        # Add exploration noise
        a = ddpg.choose_action(s)
        a = np.clip(np.random.normal(a, var), -2, 2)    # add randomness to action selection for exploration
        s_, r, done, info = env.step(a)

        ddpg.store_transition(s, a, r / 10, s_)

        if ddpg.pointer > MEMORY_CAPACITY:
            var *= .9995    # decay the action randomness
            ddpg.learn()

        s = s_
        ep_reward += r
        if j == MAX_EP_STEPS-1:
            print('Episode:', i, ' Reward: %i' % int(ep_reward), 'Explore: %.2f' % var, )
            # if ep_reward > -300:RENDER = True
            break

print('Running time: ', time.time() - t1)
```