19335266 张睿敏 论文复现 Continuous control with deep reinforcement learning

一、背景

概括来说，RL 要解决的问题是：让 agent 学习在一个环境中的如何行为动作(act)， 从而获得最大的奖励值总和(total reward)。这个奖励值一般与 agent 定义的任务目标关联。agent 需要的主要学习内容：第一是行为策略(action policy)，第二是规划(planning)。其中，行为策略的学习目标是最优策略。使用这样的策略，可以让 agent 在特定环境中的行为获得最大的奖励值，从而实现其任务目标。

在 RL 领域，DDPG 主要从：PG -> DPG -> DDPG 发展而来。DDPG 是针对连续行为的策略学习方法。

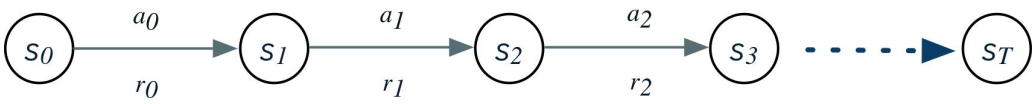
先复述一下 RL 相关的基本概念：

• s_{t}：在 t 时刻，agent 观察到的环境状态，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），通过环境执行后，环境状态由 s_{t}转换为 s_{t+1}；

• $r(s_{t} , a_{t})函数：环境在状态 s_{t}$执行行为 a_{t}后，返回的单步奖励值；

上述关系可以用一个状态转换图来表示：



• R_{t}：是从当前状态直到将来某个状态，期间所有行为所获得奖励值的加权总和，即 discounted future reward:

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

然后是 PG 的一些概念：

R.Sutton 在 2000 年提出的 Policy Gradient 方法，是 RL 中，学习连续的行为控制策略的经典方法，其提出的解决方案是：

通过一个概率分布函数来表示每一步的最优策略，在每一步根据该概率分布进行 action 采样，获得当前的最佳 action 取值；即：

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

生成 action 的过程，本质上是一个随机过程；最后学习到的策略，也是一个随机策略(stochastic policy)。

然后是 DPG：

Deepmind 的 D.Silver 等在 2014 年提出 DPG：Deterministic Policy Gradient，即确定性的行为策略，每一步的行为通过函数μ直接获得确定的值：

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

这个函数 μ 即最优行为策略，不再是一个需要采样的随机策略。

为何需要确定性的策略？简单来说，PG 方法有以下缺陷：

即使通过 PG 学习得到了随机策略之后，在每一步行为时，我们还需要对得到的最优策略概率分布进行采样，才能获得 action 的具体值；而 action 通常是高维的向量，比如 25 维、50 维，在高维的 action 空间的频繁采样，无疑是很耗费计算能力的；在 PG 的学习过程中，每一步计算 policy gradient 都需要在整个 action space 进行积分:

$$\nabla_\theta = \int_{\mathcal{S}} \int_{A} \rho(s)\pi_\theta(a|s)Q^\pi(s,a)dads$$

这个积分我们一般通过 Monte Carlo 采样来进行估算，需要在高维的 action 空间进行采样，耗费计算能力。

在这之前，业界普遍认为，环境模型无关(model-free)的确定性策略是不存在的，在 2014 年的 DPG 论文中，D.Silver 等通过严密的数学推导，证明了 DPG 的存在。

然后将 DPG 算法融合进 actor-critic 框架，结合 Q-learning 或者 Gradient Q-learning 这些传统的 Q 函数学习方法，经过训练得到一个确定性的最优行为策略函数。

最后来到这次的主角——DDPG：

Deepmind 在 2016 年提出 DDPG，全称是：Deep Deterministic Policy Gradient,是将深度学习神经网络融合进 DPG 的策略学习方法。

相对于 DPG 的核心改进是： 采用卷积神经网络作为策略函数μ和 Q 函数的模拟，即策略网络和 Q 网络；然后使用深度学习的方法来训练上述神经网络。Q 函数的实现和训练方法，采用了 Deepmind 2015 年发表的 DQN 方法 ,即 Alpha Go 使用的 Q 函数方法。

二、算法介绍

2.1 先复述一下 DDPG 相关的概念定义：

• 确定性行为策略μ：定义为一个函数，每一步的行为可以通过 a_{t} = μ(s_{t}|θ^μ)获得。

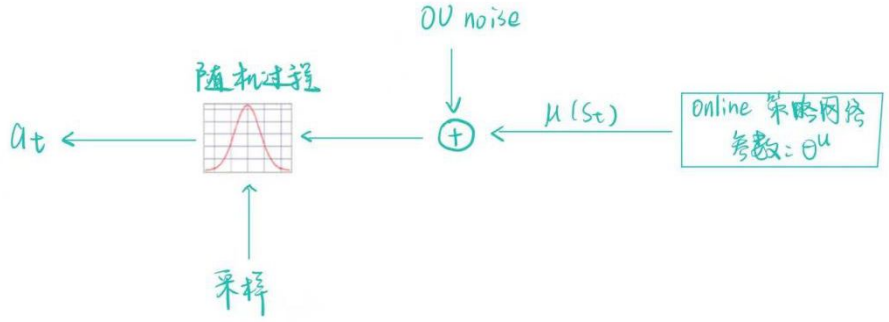
与确定性策略对应的是随机性策略，就是神经网络输出的是动作的分布，在确定每一步动作时，我们需要得到的策略分布进行采样，对于某些高纬的连续值动作，频繁的在高维空间对动作进行采样，是很耗费计算能力的。

同样，对于 DQN 算法，其只适用于低维、离散动作的问题，对于连续动作问题，DQN 要计算所有可能动作的概率，并计算可能的动作的价值，动作的数量随着自由度的数量呈指数增长，那需要非常的样本量与计算量，所以就有了确定性策略来简化这个问题。

• 策略网络：用一个卷积神经网络对μ函数进行模拟，这个网络我们就叫做策略网络，其参数为θ^μ。

• behavior policy β : 在 RL 训练过程中，我们要兼顾 2 个 e: exploration 和 exploit；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：将 action 的决策从确定性过程变为一个随机过程， 再从这个随机过程中采样得到 action，下达给环境执行.

过程如下图所示：



上述这个策略叫做 behavior 策略，用β来表示，这时 RL 的训练方式叫做 off-policy。这里与ϵ－greedy 的思路是类似的。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。

注意：

– 这个β不是我们想要得到的最优策略，仅仅在训练过程中，生成下达给环境的 action，从而获得我们想要的数据集，比如状态转换(transitions)、或者 agent 的行走路径等，然后利用这个数据集去训练策略μ，以获得最优策略。

– 在 test 和 evaluation 时，使用μ，不会再使用β。

• Q 函数: 即 action-value 函数，定义在状态 s_{t}下，采取动作 a_{t}后，且如果持续执行策略 μ 的情况下，所获得的 R_{t}期望值, 用 Bellman 等式来定义：

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

可以看到，Q 函数的定义是一个递归表达，在实际情况中，我们不可能每一步都递归计算 Q 的值，可行的方案是通过一个函数对 Bellman 等式表达进行模拟。

• Q 网络：DDPG 中，我们用一个卷积神经网络对 Q 函数进行模拟，这个网络我们就叫做 Q 网络，其参数为 Q^θ。采用了 DQN 相同的方法。

• 如何衡量一个策略μ的表现：用一个函数 J 来衡量，我们叫做 performance objective，针对 off-policy 学习的场景，定义如下：

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

其中：s 是环境的状态，这些状态是基于 agent 的 behavior 策略产生的，它们的分布函数(pdf) 为ρ^β；Q^μ(s,μ(s))是在每个状态下，如果都按照μ策略选择 acton 时，能够产生的 Q 值。也即，Jβ(μ)是在 s 根据ρ^β分布时，Q^μ(s,μ(s))的期望值。

• 训练的目标：最大化 Jβ(μ)，同时最小化 Q 网络的 Loss 函数。

• 最优行为策略μ的定义: 即最大化 Jβ(μ)的策略：

$$\mu = argmax_{\mu} J(\mu)$$

训练μ网络的过程，就是寻找μ网络参数θ^μ的最优解的过程，我们使用 SGA(stochastic

gradient ascent)的方法。

• 最优 Q 网络定义：具备最小化的 Q 网络 Loss。训练 Q 网络的过程，就是寻找 Q 网络参数 Q^θ的最优解的过程，我们使用 SGD 的方法。

2.2DDPG 实现框架和算法

• online 和 target 网络

以往的实践证明，如果只使用单个"Q 神经网络"的算法，学习过程很不稳定，因为 Q 网络的参数在频繁 gradient update 的同时，又用于计算 Q 网络和策略网络的 gradient, 参见下面等式(1),(2),(3)。基于此，DDPG 分别为策略网络、Q 网络各创建两个神经网络拷贝,一个叫做 online，一个叫做 target:

$$\text{策略网络}\begin{cases} online:\mu(s|\theta^{\mu}):gradient更新\theta^{\mu} \\ target:\mu'(s|\theta^{\mu'}):soft\ update\ \theta^{\mu'} \end{cases}$$

$$Q\text{网络}\quad\begin{cases} online:Q(s,a|\theta^{Q}):gradient更新\theta^{Q} \\ target:Q'(s,a|\theta^{Q'}):soft\ update\ \theta^{Q'} \end{cases}$$

在训练完一个 mini-batch 的数据之后，通过 SGA/SGD 算法更新 online 网络的参数，然后再通过 soft update 算法更新 target 网络的参数。soft update 是一种 running average 的算法：

$$soft\ update:_{\tau\text{一般取值}0.001}\begin{cases} \theta^{Q'} \leftarrow \tau\theta^{Q}+(1-\tau)\theta^{Q'} \\ \theta^{\mu'} \leftarrow \tau\theta^{\mu}+(1-\tau)\theta^{\mu'} \end{cases}$$

优点：target 网络参数变化小，用于在训练过程中计算 online 网络的 gradient，比较稳定，训练易于收敛。

代价：参数变化小，学习过程变慢。

• DDPG 伪代码

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:

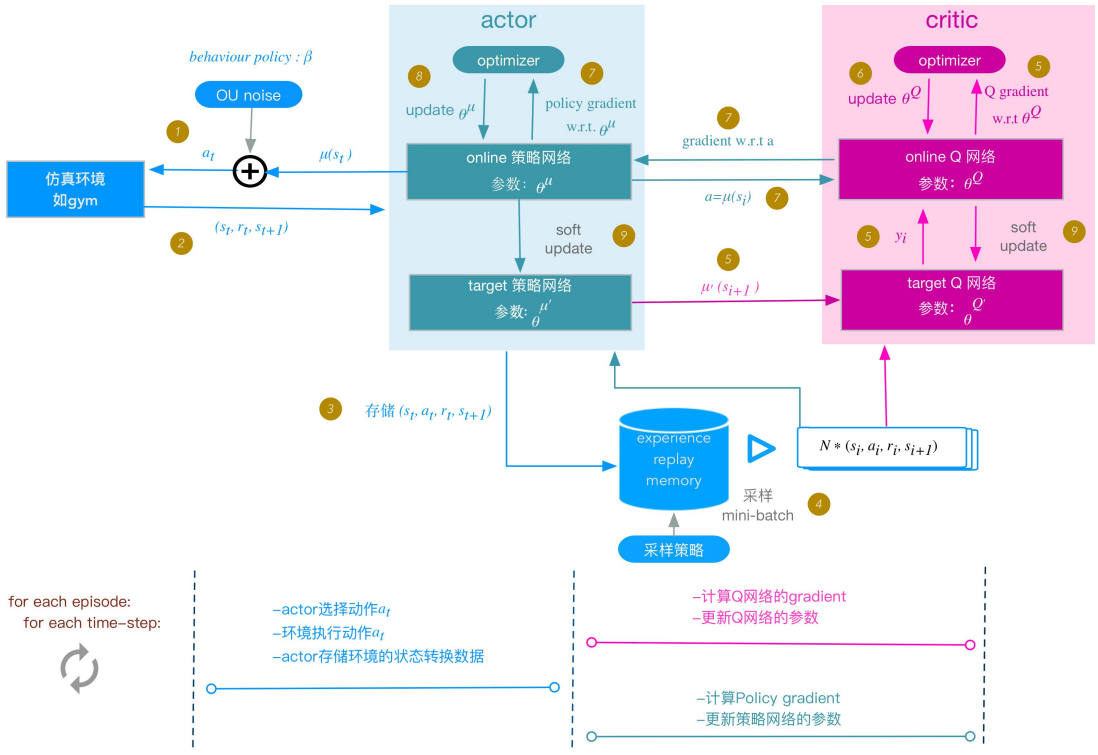
$$\nabla_{\theta^{\mu}} J \approx \frac{1}{N}\sum_i \nabla_a Q(s,a|\theta^{Q})|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^{\mu}} \mu(s|\theta^{\mu})|_{s_i}$$

    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

• 整个算法框图：



下面序号分别对应上图中的流程：

1、actor 根据 behavior 策略选择一个 a_{t}， 下达给 gym 执行该 a_{t}：

$$a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$$

behavior 策略是一个根据当前 online 策略μ和随机 UO 噪声生成的随机过程，从这个随机过程采样获得 a_{t}的值。

2、gym 执行 a_{t},返回 reward r_{t}和新的状态 s_{t+1}。

3、actor 将这个状态转换过程(transition): (s_{t}, a_{t}, r_{t}，s_{t+1}) 存入 replay memory buffer R 中，作为训练 online 网络的数据集。

4、从 replay memory buffer R 中，随机采样 N 个 transition 数据，作为 online 策略网络、online Q 网络的一个 mini-batch 训练数据。我们用(s_{i}, a_{i}, r_{i}，s_{i+1}) 表示 mini-batch 中的单个 transition 数据。

5、计算 online Q 网络的 gradient：

Q 网络的 loss 定义：使用类似于监督式学习的方法，定义 loss 为 MSE(mean squared error)：

$$L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2 \qquad (1)$$

$$y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'}) \qquad (2)$$

6、update online Q：采用 Adam optimizer 更新 Q^θ。

7、计算策略网络的 policy gradient：

policy gradient 的定义：表示 performance objective 的函数 J 针对θ^μ的 gradient。根据 2015 D.Silver 的 DPG 论文中的数学推导，在采用 off-policy 的训练方法时，policy gradient 算法如下：

$$\nabla_{\theta^\mu} J_\beta(\mu) \approx E_{s\sim\rho^\beta}[\nabla_a Q(s, a|\theta^Q)|_{a=\mu(s)} \cdot \nabla_{\theta^\mu}\mu(s|\theta^\mu)]$$

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

8、update online 策略网络：采用 Adam optimizer 更新θ^μ.

9、soft update target 网络μ'和 Q'：

使用 running average 的方法，将 online 网络的参数，soft update 给 target 网络的参数：

$$soft\ update:_{\tau一般取值0.001} \begin{cases} \theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'} \\ \theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'} \end{cases}$$

• Actor&Critic

actor-critic 框架是一个在循环的 episode 和时间步骤条件下，通过环境、actor 和 critic 三者交互，来迭代训练策略网络、Q 网络的过程。

Actor 当前网络：负责策略网络参数θ的迭代更新，负责根据当前状态 S 选择当前动作 A，用于和环境交互生成 S'和 R。

Actor 目标网络：负责根据经验回放池中采样的下一状态 S'选择最优下一动作 A'，网络参数 θ定期从θ复制。

Critic 当前网络：负责价值网络参数 w 的迭代更新，负责计算负责计算当前 Q 值 Q(S,A,w)。目标 Q 值 y i = R +γQ'(S',A',w')。

Critic 目标网络：负责计算目标 Q 值中的 Q'(S',A',w')Q 部分，网络参数 w'定期从 w 复制。

2.3 DDPG 相对于 DPG 的改进

• 使用卷积神经网络来模拟策略函数和 Q 函数，并用深度学习的方法来训练，证明了在 RL 方法中，非线性模拟函数的准确性和高性能、可收敛；而 DPG 中，可以看成使用线性回归的机器学习方法：使用带参数的线性函数来模拟策略函数和 Q 函数，然后使用线性回归的方法进行训练。

• experience replay memory 的使用：actor 同环境交互时，产生的 transition 数据序列是在时间上高度关联(correlated)的，如果这些数据序列直接用于训练，会导致神经网络的 overfit，不易收敛。

• DDPG 的 actor 将 transition 数据先存入 experience replay buffer, 然后在训练时，从 experience replay buffer 中随机采样 mini-batch 数据，这样采样得到的数据可以认为是无关联的。target 网络和 online 网络的使用， 使的学习过程更加稳定，收敛更有保障。

三、代码复现

代码使用 Pendulum-v0 连续环境，采用 tensorflow 学习框架。

配置环境：

Openai Gym Pendulum-v0, continual action space

Prerequisites

tensorflow >=2.0.0a0

tensorflow-proactionsbility 0.6.0

tensorlayer >=2.0.0

Step1 搭建网络

• Actor

def get_actor(input_state_shape):

```
    input_layer = tl.layers.Input(input_state_shape)
    layer = tl.layers.Dense(n_units=64, act=tf.nn.relu)(input_layer)
    layer = tl.layers.Dense(n_units=64, act=tf.nn.relu)(layer)
    layer = tl.layers.Dense(n_units=action_dim, act=tf.nn.tanh)(layer)
    layer = tl.layers.Lambda(lambda x: action_range * x)(layer)
    return tl.models.Model(inputs=input_layer, outputs=layer)
```

Actor 网络输入状态 ，输出动作，注意的是，连续环境的动作一般都有一个范围，这个范围在环境中已经定以好，使用 action_bound = env.action_space.high 即可获取。如果 actor 输出的动作超出范围会导致程序异常，所以在网络末端使用 tanh 函数把输出映射到[-1.0, 1.0]之间。然后使用 lamda 表达式，把动作映射到相应的范围。

- Critic

```
def get_critic(input_state_shape, input_action_shape):
    state_input = tl.layers.Input(input_state_shape)
    action_input = tl.layers.Input(input_action_shape)
    layer = tl.layers.Concat(1)([state_input, action_input])
    layer = tl.layers.Dense(n_units=64, act=tf.nn.relu)(layer)
    layer = tl.layers.Dense(n_units=64, act=tf.nn.relu)(layer)
    layer = tl.layers.Dense(n_units=1, name='C_out')(layer)
    return tl.models.Model(inputs=[state_input, action_input], outputs=layer)
```

在 DDPG 中我们把 状态和动作同时输入到 Critic 网络中，去估计 Q(s,a)。所以定义两个输入层，然后连接起来，最后模型的输入部分定义两个输入。

Step2 主流程

```
for episode in range(TRAIN_EPISODES):
    state = env.reset()
    for step in range(MAX_STEPS):
        if RENDER: env.render()
        # Add exploration noise
        action = agent.get_action(state)
        state_, reward, done, info = env.step(action)
        agent.store_transition(state, action, reward, state_)
        if agent.pointer > MEMORY_CAPACITY:
            agent.learn()
        state = state_
        if done: break
```

可以看到，DDPG 流程与 DQN 基本相同，重置状态，然后选择动作，与环境交互，获得 S’，R 后，把数据保存起来。如多数据量足够，就对数据进行抽样，更新网络参数。然后开始更新 s，开始下一步循环。

这里重点看一下 get_action()函数：

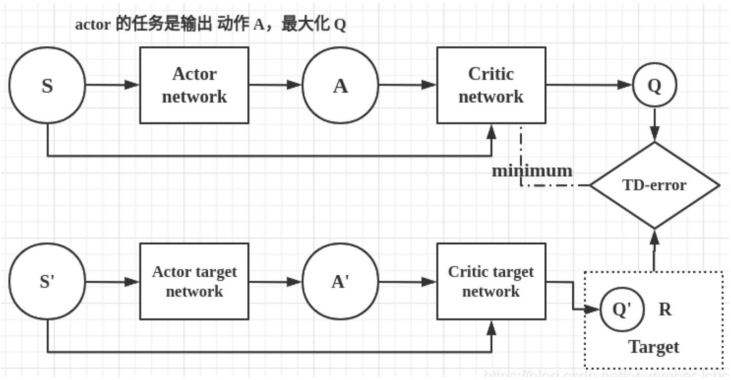
```
def get_action(self, s, greedy=False):
a = self.actor(np.array([s], dtype=np.float32))[0]
if greedy:
    return a
return np.clip(
    np.random.normal(a, self.var), -self.action_range, self.action_range)
```

get_action() 函数用以选取一个动作，然后与环境交互。为了更好的探索环境，我们在训练过程中为动作加入噪声，原始的 DDPG 作者推荐加入与时间相关的 OU 噪声，但是更近的结果表明高斯噪声表现地更好。由于后者更为简单，因此其更为常用。

我们这里就采用的后者，为动作添加高斯噪声：这里我们的 Actor 输出的 a 作为一个正太分布的平均值，然后加上参数 VAR，作为正太分布的方差，然后就可以构造一个正太分布。然后从正太分布中随机选取一个动作 ，我们知道正太分布是有很大概率采样到平均值附近的点，所以利用这个方法，就可以实现一定的探索行为。此外，我们也可以控制 VAR 的大小，来控制探索概率的大小。

当测试的时候，选取动作时就不需要探索，因为这时 Actor 要输入有最大 Q 值的动作，直接输出动作就可以。所以在 get_action() 函数中用 一个参数 greedy 来控制这两种情况。

Step3 网络更新



• Critic 更新

如上图所示，Critic 部分的更新和 DQN 是一样的，使用 td-error 来更新。用目标网络构造 target，然后和当前网络输出的 q 计算 MSE 损失，然后更新网络参数。

```
with tf.GradientTape() as tape:
    actions_ = self.actor_target(states_)
    q_ = self.critic_target([states_, actions_])
    target = rewards + GAMMA * q_
    q_pred = self.critic([states, actions])
    td_error = tf.losses.mean_squared_error(target, q_pred)
critic_grads = tape.gradient(td_error, self.critic.trainable_weights)
self.critic_opt.apply_gradients(zip(critic_grads, self.critic.trainable_weights))
```

• Actor 更新

DDPG 采用梯度上升法，Actor 作用就是输出一个动作，这个动作输入 Critic 网络能得到最大的 Q 值。由于和梯度下降方向相反，所以需要在 loss 函数前面加上负号。

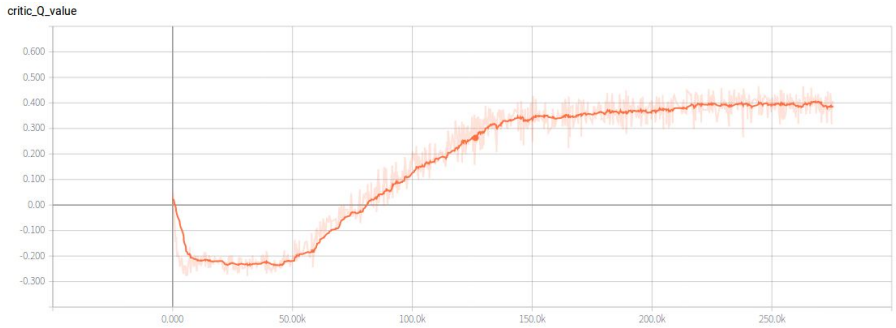
```
with tf.GradientTape() as tape:
actions = self.actor(states)
q = self.critic([states, actions])
actor_loss = -tf.reduce_mean(q)    # maximize the q
actor_grads = tape.gradient(actor_loss, self.actor.trainable_weights)
self.actor_opt.apply_gradients(zip(actor_grads, self.actor.trainable_weights))
```

完整代码详见附件。

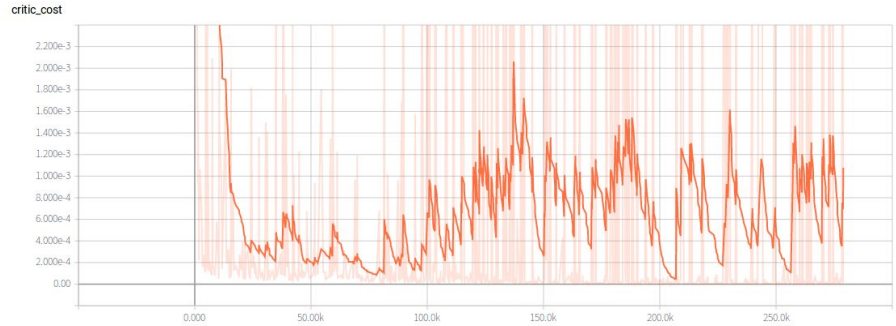
四、复现结果

4.1 使用 tensorboard 查看训练图像

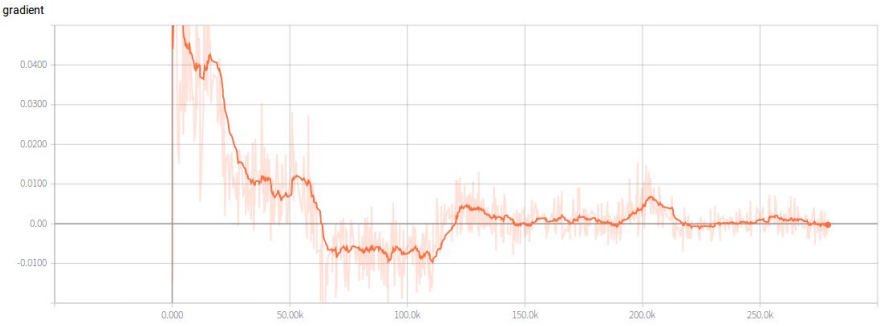
• 下图是 critic 网络输出 Q 值的总结，可以看到 Q 值从-0.24 左右（不训练 actor 网络就是这个值），一直涨到 0.4 左右。应该是我没有对动作噪声进行衰减，所以后面的策略改善速度就很慢了。



•下图是 critic 网络的损失值的总结，可以看到 critic 网络的学习频率大一些可能会更好（目前是感知 200 次，学习 25 次）

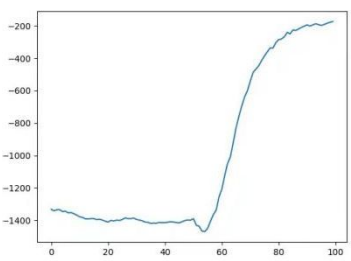


•下图是 critic 网络的梯度的总结。这个梯度参数体现了 actor 网络的学习情况。因为 actor 网络学习的直接目标，就是让 actor 网络输出的动作，在 critic 网络中有更接近 0 的梯度（极值）。



4.2 训练出的 model

actor.hdf5	2022/1/1 17:24	HDF5 文件	158 KB
actor_target.hdf5	2022/1/1 17:24	HDF5 文件	159 KB
critic.hdf5	2022/1/1 17:24	HDF5 文件	159 KB
critic_target.hdf5	2022/1/1 17:24	HDF5 文件	159 KB



五、改进与拓展