(O)Unit 18 DDPG

Deterministic Policy Gradient (DPG)=DQN+AC

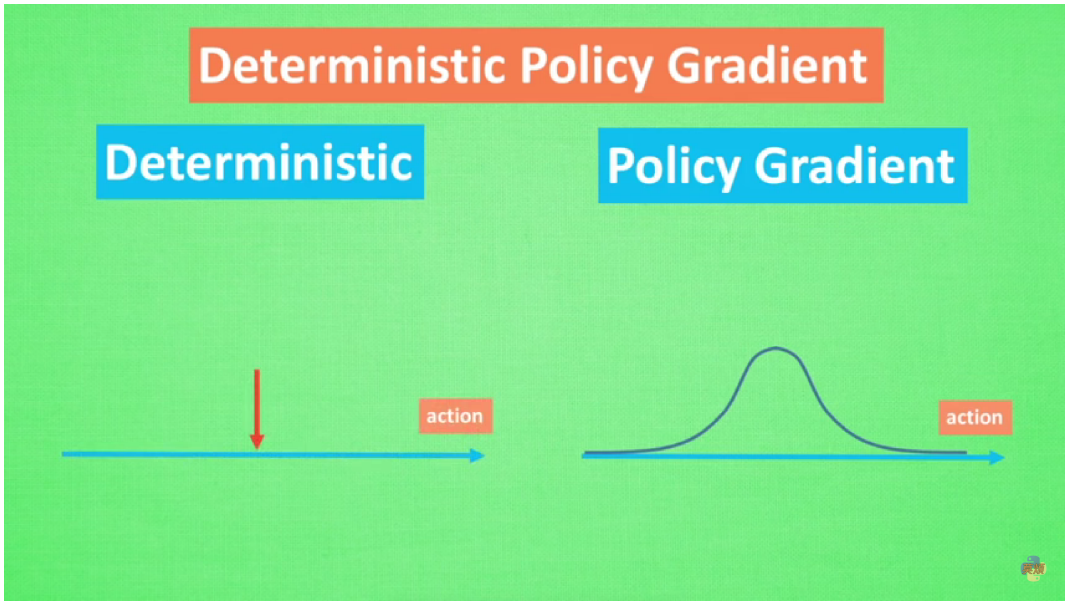
- 最大優勢：在連續動作上更有效學習，增加actor critic的穩定性加速收斂
- 如何做到：吸收 Actor critic 讓 policy gradient 單步更新精華及讓機器學著玩遊戲DQN的精華。

▼ 和 DQN 的共同點：

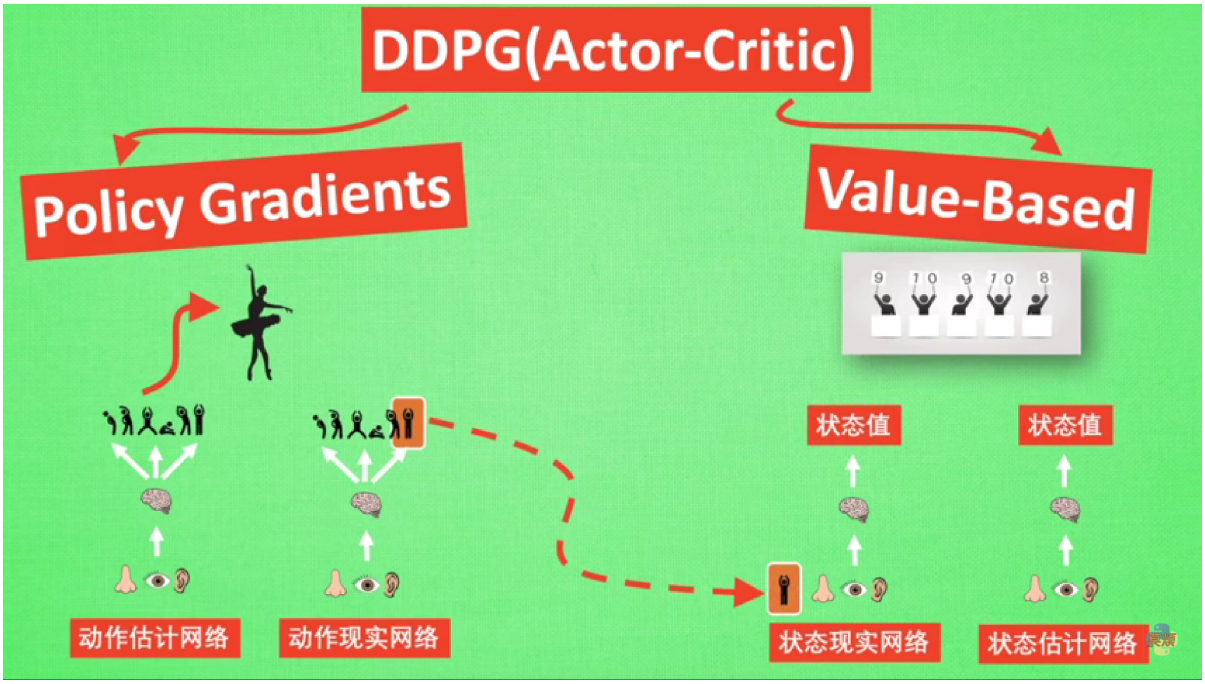
一樣架構為記憶庫 + 兩個結構相同但參數更新頻率不同的神經網路(估計網路 eval_net (持續更新)& 現實網路 target_net (延遲更新)) 一個延遲一個持續更新 >> 更有效學習。

▼ 和 policy gradient 差別：

將 policy gradient 在連續動作上輸出隨機概率動作 改成 輸出確定(Deterministic) 的一個動作，因最後也只會挑一個動作。



▼ 如何運作更新



▼ 分為 Actor & Critic 各別像 DQN 一樣有兩個網路

1. 估計網路 evel_net 持續更新 2. 現實網路 target_net 延遲更新

	估計網路(eval_net)	現實網路(target_net)
Actor 價值系統	輸出實質的動作，供 actor 在現實中實行	更新Critic價值網絡系統
Critic 價值系統	同 : 輸出狀態的價值	同 : 輸出狀態的價值
	不同 : 輸入從 Actor 現實網路(target_net) 來的動作加上狀態(state) 的觀測值加以分析	不同 : 拿著當時 Actor 施加的動作當做輸入

- Actor 的估計網路(eval_net)用來輸出實質的動作，供 actor 在現實中實行，而Actor 的現實網路(target_net)則是用來更新Critic價值網絡系統的。
- Critic 價值系統的現實網路(target_net)和估計網路(eval_net)，都在輸出這個狀態的價值，而輸入端卻有不同。
- Critic 狀態現實網路(target_net)這邊會拿著從Actor動作現實網路(target_net)來的動作加上狀態(state)的觀測值加以分析，而狀態估計網路(eval_net)則是拿著當時 Actor 施加的動作當做輸入，在實際運用中，DDPG 的這種做法的確帶來了更有效的學習過程。

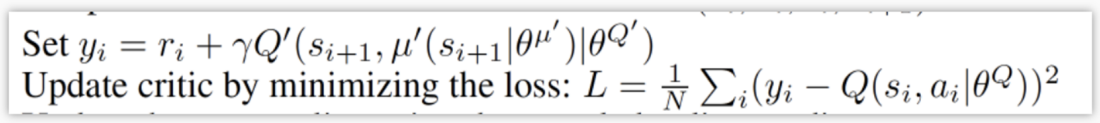
▼ Actor 參數如何更新

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

Actor 參數更新	來源	意涵
前半部	grad[Q] 是從 Critic 來	Actor 的動作要怎麼移動能獲得更大的 Q
後半部	grad[u] 是從 Actor 來	Actor 要怎麼樣修改自身參數，使得 Actor 更有可能做這個動作

兩者合起來就是在說： Actor 要朝著更有可能獲取大 Q 的方向修改動作參數了。

▼ Critic 如何更新



兩個計算 Q 的神經網絡，Q_target 中依據下一狀態，用 Actor 來選擇動作，而這時的 Actor 也是一個 Actor_target (有著 Actor 很久之前的參數)。使用這種方法獲得的 Q_target 能像 DQN 那樣切斷相關性，提高收斂性。

▼ 參考影片：

- 莫煩

https://www.youtube.com/watch?v=3e6pfzux1x0

https://youtu.be/TQE4OLSz2BE

https://mofanpy.com/tutorials/machine-learning/reinforcement-learning/DDPG

▼ HW05 DDPG+ stable-baseline3

(1) Source code

▼ stable baseline3

(https://stable-baselines3.readthedocs.io/en/master/modules/ddpg.html).

```
# 使用 Stable Baselines3 中的 DDPG 演算法來訓練和運行一個強化學習代理，以解決 OpenAI Gym 中的 Pendulum-v1 問題

# 導入 OpenAI Gym 環境。
import gymnasium as gym
# 導入 NumPy 庫，用於數值計算
import numpy as np
# 從 Stable Baselines3 中導入 DDPG 演算法
from stable_baselines3 import DDPG
# 從 Stable Baselines3 中導入用於添加動作噪音的類別
from stable_baselines3.common.noise import NormalActionNoise, OrnsteinUhlenbeckActionNoise
# 創建名為 Pendulum-v1 的 OpenAI Gym 環境，並設置渲染模式為 "rgb_array"
env = gym.make("Pendulum-v1", render_mode="rgb_array")
```

```
# The noise objects for DDPG
# 獲取環境的動作空間大小
n_actions = env.action_space.shape[-1]
# 創建一個正態分布的動作噪音對象，用於 DDPG 演算法
action_noise = NormalActionNoise(mean=np.zeros(n_actions), sigma=0.1 * np.ones(n_actions))
# 創建 DDPG 模型，使用 "MlpPolicy" 神經網路架構，設置環境和動作噪音，並設置詳細程度為 1
model = DDPG("MlpPolicy", env, action_noise=action_noise, verbose=1)
# 訓練 DDPG 模型，總共進行 10000 步，每 10 步記錄一次訓練信息
model.learn(total_timesteps=10000, log_interval=10)
# 將訓練後的模型保存命名為 "ddpg_pendulum.zip"
model.save("ddpg_pendulum")

vec_env = model.get_env()

del model # remove to demonstrate saving and loading
# 從 ddpg_pendulum.zip 加載模型
model = DDPG.load("ddpg_pendulum")
# 重置環境並獲取初始觀測值
obs = vec_env.reset()
# 進入無限循環，運行代理(agent)與環境(env)的互動。
while True:
    # 使用模型預測下一個動作
    action, _states = model.predict(obs)
    # 在環境中執行動作，獲取新的觀測值、回報、終止標誌和其他信息
    obs, rewards, dones, info = vec_env.step(action)
    # 以 "human" 模式渲染環境，讓你可以看到代理的互動。
    env.render("human")
```

▼ RL zoo

https://rlzoo.readthedocs.io/en/latest/algorithms/dppo.html

```
from rlzoo.common.env_wrappers import build_env
 from rlzoo.common.utils import call_default_params
 from rlzoo.algorithms import DDPG

 AlgName = 'DDPG'
 EnvName = 'Pendulum-v0'  # only continuous action
 EnvType = 'classic_control'

 # EnvName = 'BipedalWalker-v2'
 # EnvType = 'box2d'

 # EnvName = 'Ant-v2'
 # EnvType = 'mujoco'

 # EnvName = 'FetchPush-v1'
 # EnvType = 'robotics'

 # EnvName = 'FishSwim-v0'
 # EnvType = 'dm_control'

 # EnvName = 'ReachTarget'
 # EnvType = 'rlbench'

 env = build_env(EnvName, EnvType)
 alg_params, learn_params = call_default_params(env, EnvType, AlgName)
 alg = eval(AlgName+'(**alg_params)')
```

```
alg.learn(env=env, mode='train', render=False, **learn_params)
alg.learn(env=env, mode='test', render=True, **learn_params
```