2020 年臺灣國際科學展覽會
研究報告

區別：北區

科別：電腦科學與資訊工程科

作品名稱：機器人自我調整強化學習演算法

關鍵詞：強化學習、自回饋、機器人

編號：

目錄

摘要

隨著仿生機器人的蓬勃發展，誕生許多足式機器人，能模仿如：豹、蜘蛛、昆蟲等生物。在控制系統方面，已取得不少成果，然而結構受損時，尚未能如生物般的快速適應。
一些研究以機器學習之方式訓練機器人行走，如 DeepMind 所展現的強化學習 AI 在模擬器中操控人形機器習得奔跑、跨越障礙以及應對外在衝擊等能力。

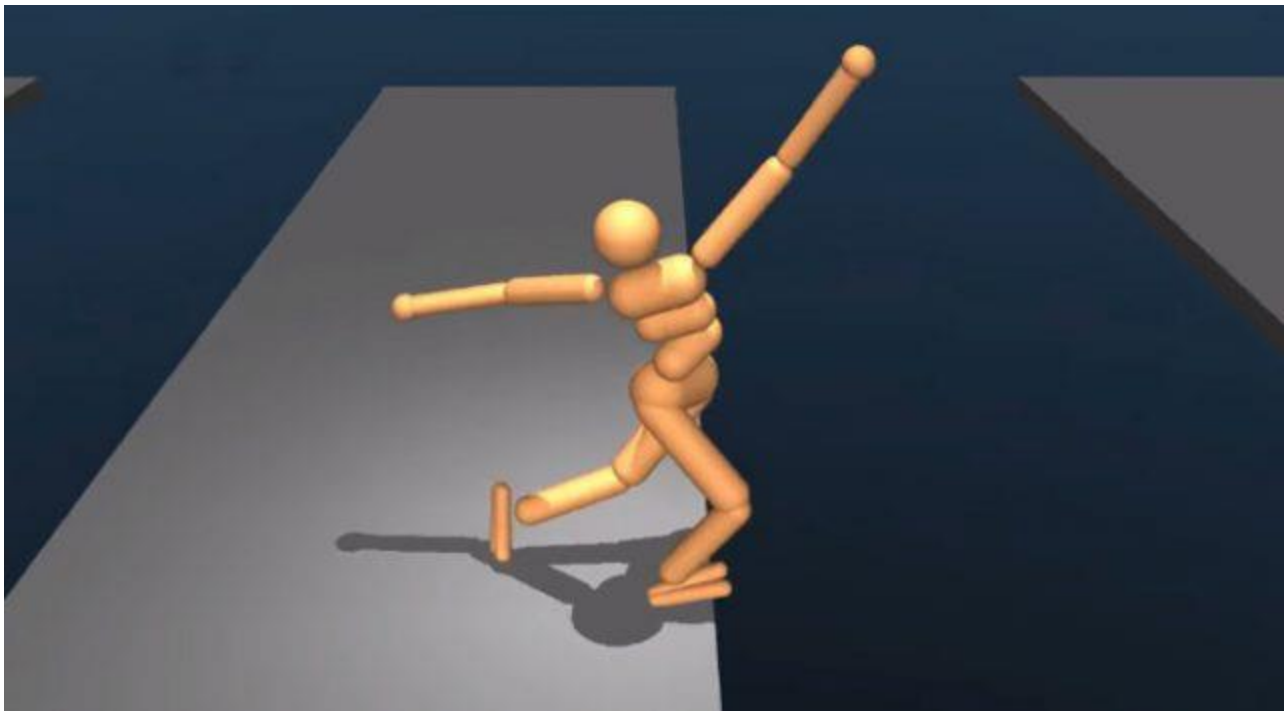

圖 1：DeepMind 訓練 AI 學會跑步(2017)

因此本研究使用強化學習算法並參考生物的記憶能力，期望運用在多足機器人上，達成適應結構受損之目的。
此方法能免於人工為不同機體編寫系統及受損之應對方案。然而強化學習本身的發展尚未能達成複雜環境下的穩定控制，是此方法的一大限制。

壹、 研究動機

一、 研究動機

組裝、操控機器人一直是我平日的一大樂趣，二足機器人格鬥賽更是令我熱血沸騰。然而對於各種不同的機體，傳統上皆需人為編寫動作，儘管只是稍微修改結構也可能花上數小時甚至數天來修改動作，因此便希望透過強化學習來讓機器人自行學會做出各種動作。

然而當機器人的零件故障或結構稍有改變時，訓練好的模型極有可能需要重新花費時間訓練。若與人類受傷時做比較，除了人類具有強大的推理能力外，若先前曾有類似的受傷經驗，便能更快速的適應。因此想到了若在訓練過程中讓機器人擁有損傷經驗，或許就能用來在未來的相似情況中達到幫助。
若能擁有適應損傷的能力，機器人便能在風險較大的環境中執行任務，例如遠距操作時，由於無法及時維修，機器便需要在有缺陷的情況下完成任務。

二. 研究目的

以往的算法中模型是固定，即模型針對一個機體訓練好後不容易處理不同情況，因此機體受損時很難即時適應，儘管有研究透過搜尋法，預先建立動作－價值對應關係，當機體複雜性增加時，仍會出現效率低落的現象。
因此本次研究希望運用人類在適應損傷時能活用經驗的能力來建立一套算法，使其能在機體損傷的狀況下利用過去經驗修正，避免重新訓練並以較短的時間適應。

貳、 研究方法或過程

一. 理論背景

強化學習(Reinforcement Learning，簡稱 RL)是機器學習中的一個領域，與一般機器學習不同的是，其著重在如何與環境的互動以取得最大利益。由於 RL 具有普適性，因此在各種領域如博弈論、控制論、群體智能等皆有研究。在控制系統中，強化學習成功在複雜、多變的情境如機器人控制中取得突出的成果。

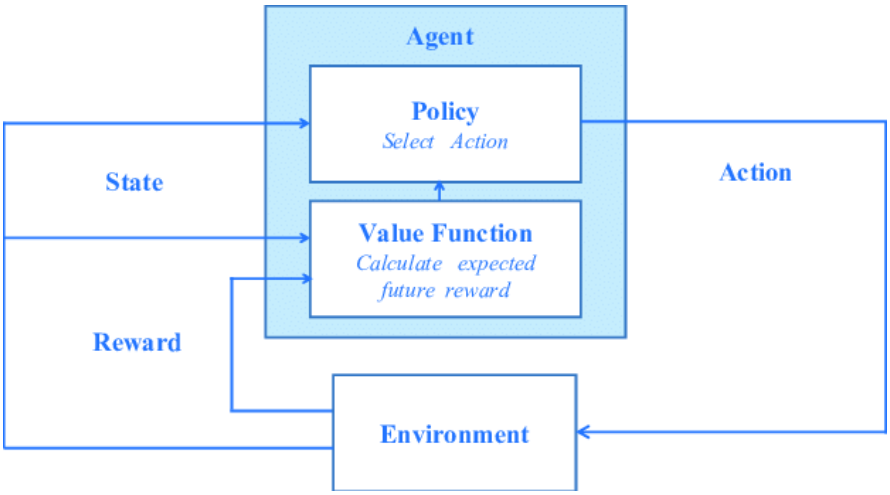

圖 2：簡易強化學習算法架構

一個簡易強化學習模型包含：代理(Agent)、環境(Environment)。在每輪的學習中，Agent 會執行一個動作(Action)，然後從 Environment 得到獎賞(Reward)與狀態(State)，接著透過價值函數(Value Function)估計未來回報，最後經策略(Policy)執行下一個 Action。

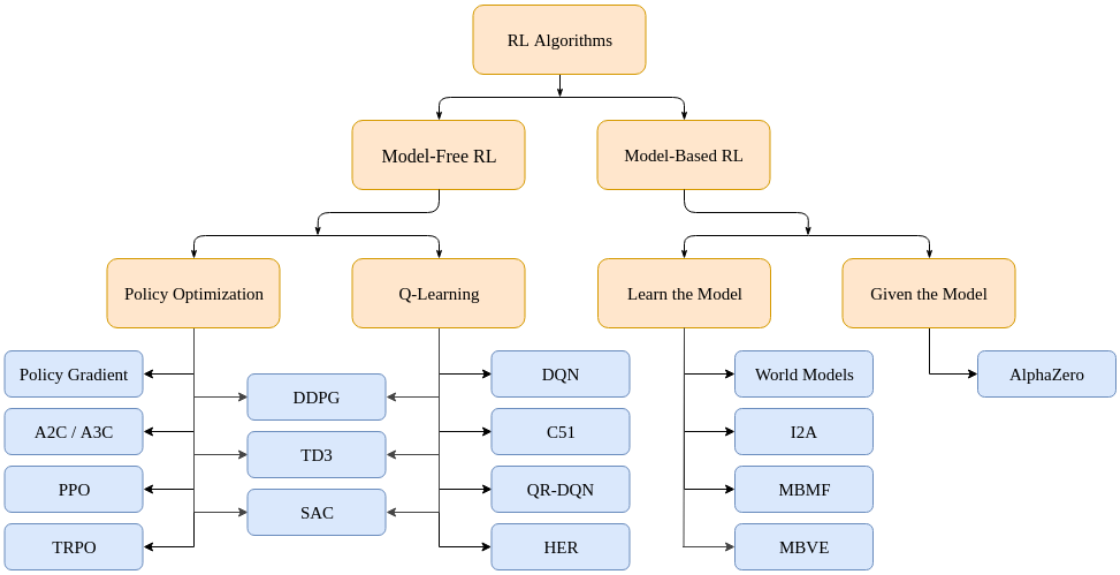

圖 3：強化學習算法生態圖

RL 算法主要可分為兩類：無關模型(Model-Free)、基於模型(Model-Based)

表 1：Model-free/Model-based 比較表

	Model-free	Model-based
說明	算法本身不了解環境狀態可能如何轉移，透過長期與環境互動的收益，仍可以評估與預測不同動作的回報	在學習的過程可觀察與學習環境在不同動作下的變化機制，預測環境狀態如何轉移
優點	由於能在未知環境中學習，因此具有更好的普適性	充分了解環境，可利用動態規劃算法求出較好的動作
缺點	學習僅能透過探索評估動作好壞，通常需較多時間收斂	大多數情況下無法提供算法對於環境的精確描述導致算法普適性較低

實際上面對複雜多變的環境難以用規劃法求解，因此多數 RL 算法 Model-free 的。而 Model-free 算法又可依據優化的對象的再區分為兩類：基於值(Value-based)、基於策略(Policy-based)

表 2：Value-based/Policy-based 比較表

	Value-based	Policy-based
說明	學習如何正確評估動作的好壞，也就是優化 Value Function	學習如何做出最好的動作，也就是優化 Policy
舉例	DQN、SARSA、QR-DQN	Policy Gradient、PPO、TRPO

一. 架構構想

(一) 算法基礎

本研究採用 DDPG(Deep Deterministic Policy Gradient)算法作為基礎，原因在於 DDPG 混和 Value-based 與 Policy-based，能處理評價與選擇動作的複雜性；且 DDPG 能處理連續動作空間，適合馬達的運作。

(二) 經驗生成

在 RL 中，為了增加訓練效率與模型泛化性，使用經驗回放(Experience Replay)機制，將模型經歷過的經驗儲存起來作為訓練樣本。因此本研究利用此機制，在訓練階段預先在受損狀態下進行定量訓練，獲取相關樣本，以利在實際受損時幫助訓練。

(三) 適應訓練

通常實際運作時的環境不會與訓練完全相同，受損情況也會有差異，因此篩選出損傷經驗後，機器仍需同時進行探索，補充實際狀況的樣本。

二. DDPG 簡述

（一） Actor-Critic

DDPG 採用 Actor-Critic 來達到同時學習 Policy 與 Value Function：

1. Critic

神經網路，作為 Value Function 使用，輸入為 Action a 與 State s，輸出為 $Q(s,\ a;\ \omega)$，ω 為網路參數。Critic 用來評估未來回報：

$$Q(s_t, a_t; \omega) = E\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k} \middle| s_t, a_t\right] = E\left[r_t + \gamma Q\left(s_{t+1}, \mu(s_{t+1}; \theta)\right) \middle| s_t, a_t\right]$$
$$Q^*(s_t, a_t) = \max_{\omega} Q(s_t, a_t; \omega) = \mathbb{E}\left[r_t + \gamma Q^*\left(s_{t+1}, \mu(s_{t+1}; \theta)\right) \middle| s_t, a_t\right] \tag{1}$$

其中 $0 < \gamma < 1$ 為折扣值，表示對於越遙遠的未來，重要性越低。
訓練時，根據 $Q(s, a; \omega)$ 的定義來設定更新目標並計算損失：

$$target_t = r_t + \gamma Q(s_{t+1}, \mu(s_{t+1}; \theta^-); \omega^-)$$
$$loss = \left(target_t - Q(s_t, a_t; \omega)\right)^2 \tag{2}$$

2. Actor

神經網路，作為 Policy 使用，輸入為狀態 s，輸出為 $\mu(s; \theta)$，θ 為網路參數。
透過策略梯度更新 Actor，令 $J(\theta)$ 為評價策略的函數：

$$J(\theta) = \mathbb{E}_s[Q\left(s, \mu(s; \theta)\right)] \tag{3}$$

策略梯度即為：

$$\Delta_\theta J(\theta) = \mathbb{E}_s[\Delta_a Q(s, a)|_{a=\mu(s)} \Delta_\theta \mu(s; \theta)] \tag{4}$$

兩個網路的交互關係為：t 時刻時，Actor 根據 s_t 作出決策 $a_t = \mu(s_t; \theta)$，接收 Reward r_{t+1} 後，用 r_{t+1} 更新 Critic，然後以 Critic 給出的 $Q(s_t, a_t; \omega)$ 更新 Actor。
由於更新網路時，若使用網路本身來計算目標，會導致以自己為目標更新。因此 Actor 與 Critic 會另外使用 Target Actor 與 Target Critic 作為更新對象，網路參數分別為 θ^-, ω^-。
Target 初始與原網路相同，訓練採 Soft-Update：

$$\begin{cases} \theta^- \leftarrow \tau\theta + (1-\tau)\theta^- \\ \omega^- \leftarrow \tau\omega + (1-\tau)\omega^- \end{cases} \text{ with } \tau \ll 1 \tag{5}$$

3. Experience Replay

令經驗庫為 $\mathcal{D}$，其中儲存樣本 (s_i, a_i, r_i, s_{i+1})，容量限制為 $\mathcal{C}$。每時間步從 $\mathcal{D}$ 取出一個 batch 的樣本 $\left(s_j, a_j, r_j, s_{j+1}\right)$更新 Actor 與 Critic。

$$L(\omega) = \frac{1}{N} \sum_{k=1}^{N} (target_k - Q(s_t, a_t; \omega))^2 \tag{6}$$

對於 Actor，沿策略梯度方向更新：

$$\Delta_\theta J(\theta) = \frac{1}{N} \sum_{k=1}^{N} \Delta_a Q(s, a)|_{s=s_k, a=\mu(s_k)} \Delta_\theta \mu(s; \theta)|_{s=s_k} \tag{7}$$

然後將當前樣本 (s_t, a_t, r_t, s_{t+1}) 存入 $\mathcal{D}$，並使用時間差分誤差(TD-error)評估樣本值得不值得學習與保留：

$$\delta_t = r_t + \gamma Q(s_{t+1}, \mu(s_{t+1}; \theta^-); \omega^-) - Q(s_t, a_t; \omega) \tag{8}$$

$|\delta_t|$ 越大表示模型對該狀態未能準確評估，越值得學習。

三. 算法設計

基於 DDPG 算法與具有優先級別的經驗回放，本研究設計出一套算法並試驗是否能達到利用經驗加速適應損傷之目的：

（一） 訓練階段

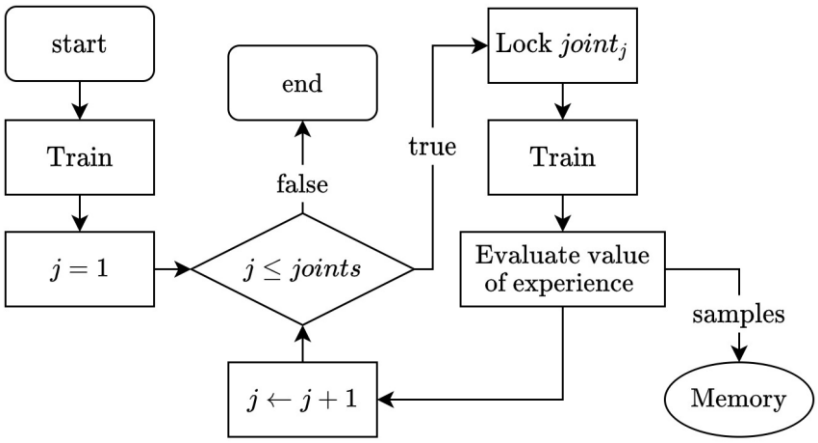

圖 4：訓練階段流程

1. 訓練(Train)

在機體完整的狀態下訓練模型。

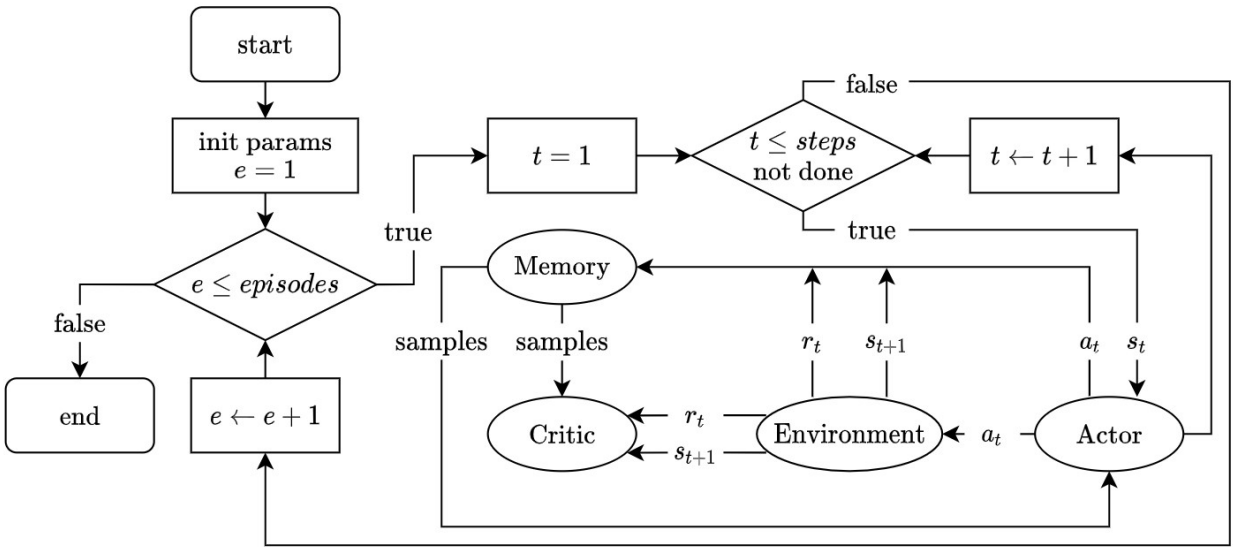

圖 5：Train 模塊流程

2. 施加損傷(Lock joint)

分別鎖定關節，並以正常模型參數為初始狀態訓練。

3. 評估適應難度(Evaluate value of experience)

評估損傷的適應難度，低則保留少量樣本；高則保留較多樣本。若訓練後平均回報高、回報波動幅度低則適應難度低；若平均回報低、回報波動幅度高則適應難度低。

波動幅度計算時，先透過計算回報的時間差分序列消除原序列趨勢，再計算整體標準差 σ_j，令 $episode$ 總數為 EP，每個 $episode$ 總回報 R_j^{ep}：

$$\sigma_j = \sqrt{\frac{1}{EP-1}\sum_{ep=2}^{EP}\left(\Delta R_j^{ep} - \mu(\Delta R_j)\right)^2}$$
$$\Delta R_j^{ep} = R_j^{ep} - R_j^{ep-1} \tag{9}$$
$$\mu(\Delta R_j) = \frac{1}{EP-1}\left(R_j^{EP} - R_j^{1}\right)$$

令平均回報為 $\overline{R_j}$，回報可為負值，故減去最小值後再計算其指標 q_j，令 $joint$ 總數為 JN：

$$q_j = e^{-\frac{\overline{R_j}-\min \bar{R}}{\mu(\bar{R})-\min \bar{R}}}$$
$$\mu(\bar{R}) = \sum_{j=1}^{JN}\overline{R_j} \tag{10}$$

適應難度 $v_j = q_j\sigma_j$

4. 分配樣本

依據適應難度分配樣本，令 c_j 為鎖定 $joint_j$ 的損傷樣本數：

$$c_j = \frac{v_j}{\sum_{i=0}^{JN} v_i}\, C \tag{11}$$

（二） 運作階段

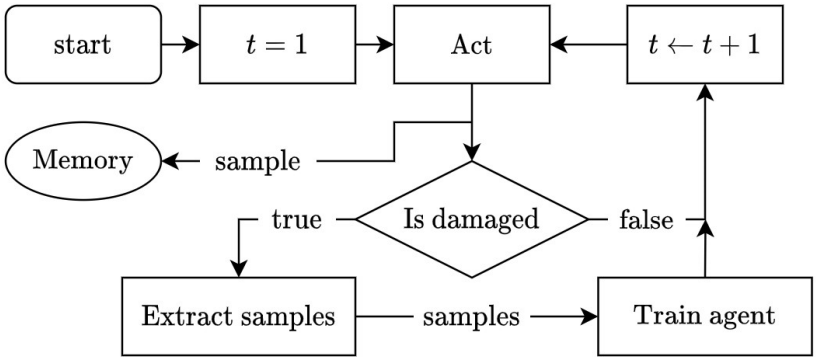

圖 6：運作流程

1. 執行(Act)

執行任務並儲存運作時的樣本。

2. 提取樣本(Extract samples)

若機體受損，從經驗庫提取損傷用於訓練 Agent 適應。

四.　實驗環境

利用 CoppeliaSim 的遠端 API 給 Python 語言進行操作，再用 TensorFlow 建立模型與訓練。

表 3：實驗環境

作業系統	Windows 10 Home
雲端運算環境	TWCC 臺灣 AI 雲
程式語言	Python 3.6.8
模型框架	TensorFlow 2.0
機器人模擬器	CoppeliaSim V4.0.0

（一）　機體設計

考慮本次研究希望觀察正常狀態至損傷狀態的適應情形，因此挑選容易訓練、損傷影響大的四足機器人。共有 8 個關節：

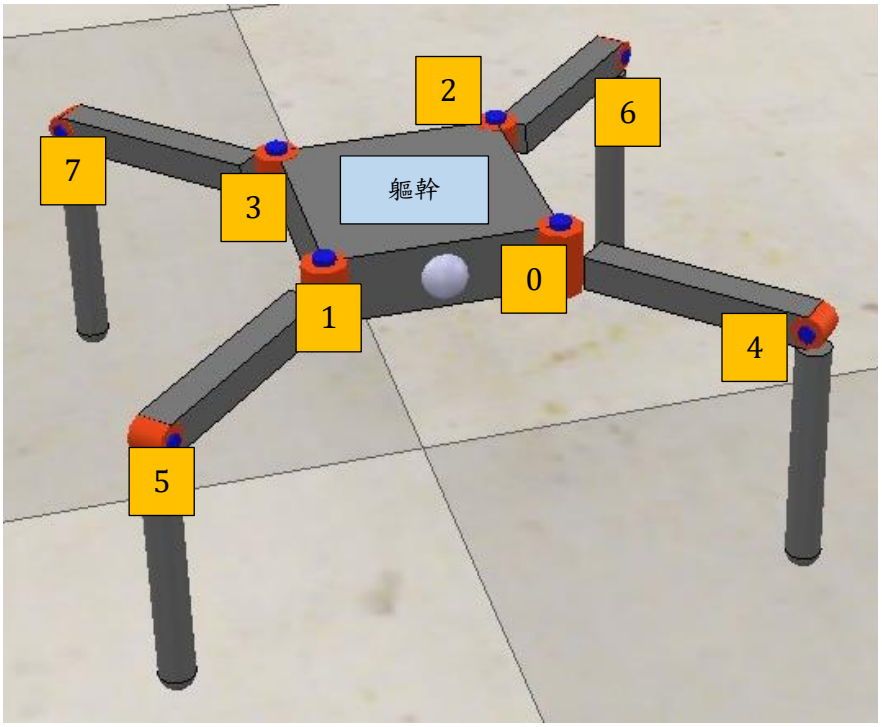

圖 7：模擬用四足機器人

表 4：機體狀態、動作資訊

	維度	資料
State	25	軀幹：座標 x,y,z、速度 v_x,y_x,z_x、傾角 ψ,θ,ϕ 馬達：角度 $q_1,q_2,\cdots,q_8$、角速度 $\omega_1,\omega_2,\cdots,\omega_8$
Action	8	馬達：輸出力矩 $\tau_1,\tau_2,\cdots,\tau_8$

（二） 場景設計

讓機器人沿著 x 正方向移動至目標處，如圖：

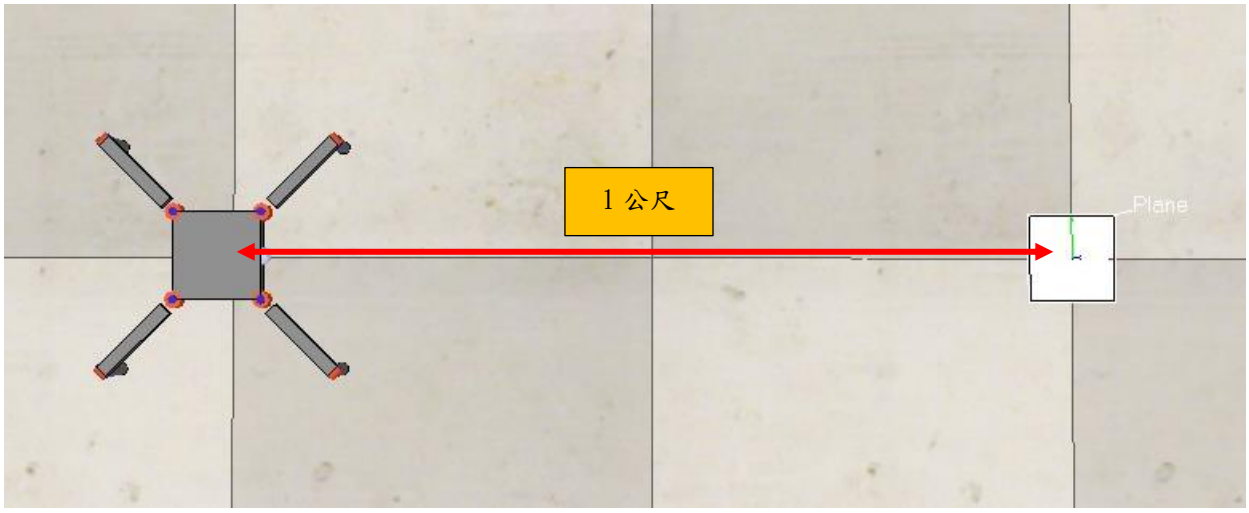

圖 8：CoppeliaSim 模擬訓練場景

（三） Reward Function

實驗中的 Reward Function 期望使機器人達成以下目標：

1. 行走平穩

向前移動，故 v_x 應保持正值；盡量沿著直線移動，故 y 應接近零；減少 y 方向上的晃動，故 v_z 應接近零。得出機體姿態對 Reward 的貢獻值：

$$w_x v_x - (w_y y^2 + w_z v_z^2) \tag{12}$$

其中 w_x, w_y, w_z 為正權重。

2. 動作精簡

希望使機器人能以較有效率的輸出達成目標，因此對馬達總輸出加入懲罰項：

$$-w_\tau \sum_{j=1}^{JN} |\tau_j| \tag{13}$$

其中 w_τ 為正權重。

$$\mathrm{r_t} = w_x v_x - (w_y y^2 + w_z v_z^2) - w_\tau \sum_{j=1}^{JN} |\tau_j| \tag{14}$$

（四） Agent 神經網路結構

Actor 輸入 State，經兩層隱藏輸出：

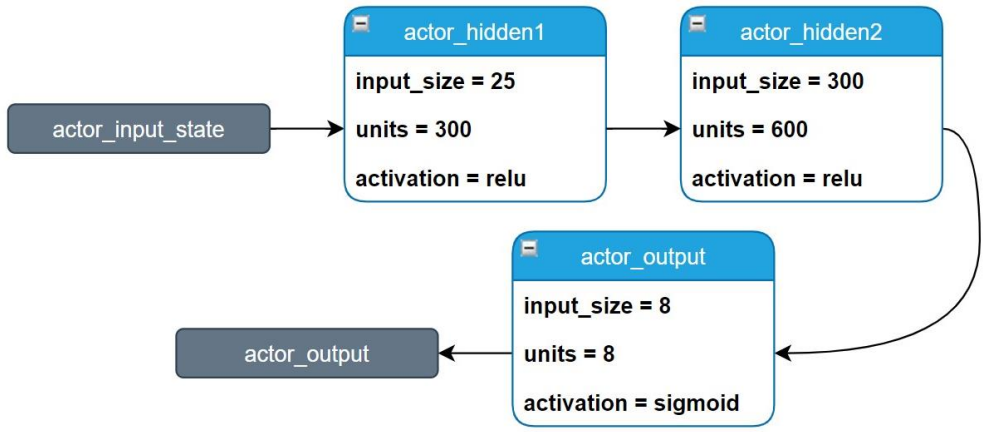

圖 9：Actor 神經網路結構

Critic 輸入 State 與 Action，分別經一層隱藏層後加總，再經一層隱藏輸出：

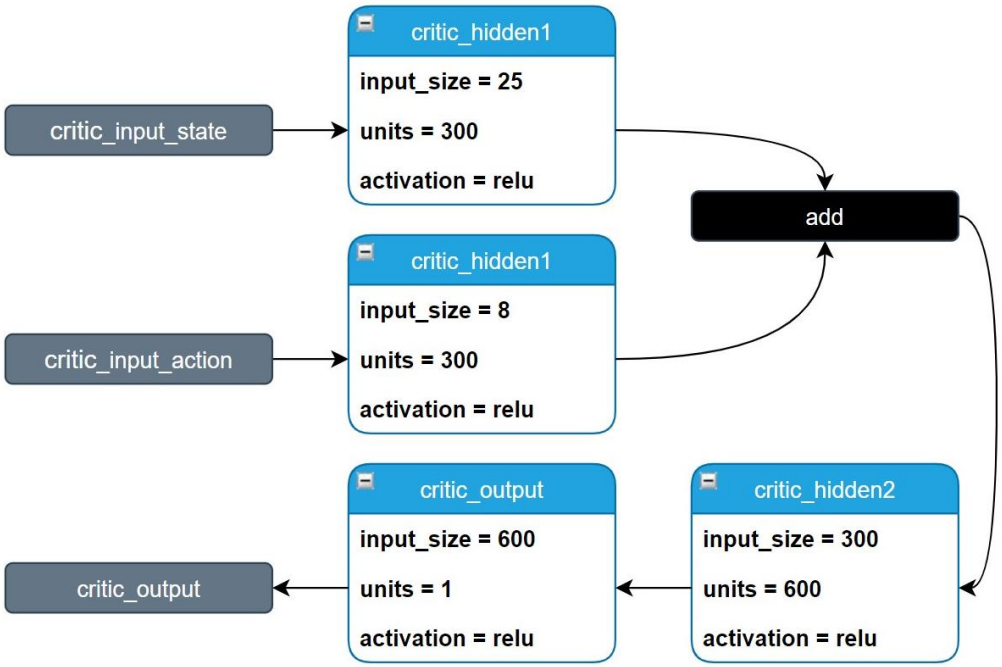

圖 10：Critic 神經網路結構

貳、 研究過程

一. 正常機體訓練

影響 RL 模型表現的最大要素在於 Reward Function 的設計，以下為對 Reward Function 進行調整的過程：

（一） Reward Function 測試

表 5：基礎參數設置

模型參數	Network Size		γ	τ	Memory Size
	Actor	Critic	0.99	0.001	1000000
	(300, 600)	(300, 600)			
訓練參數	Learning Rate		Batch Size	Episodes	Steps/Episodes
	Actor	Critic	128	200	200
	0.001	0.01			

1. 測試 1.1.1

表 6：測試 1.1.1 參數設置

w_x	w_y	w_z	w_τ
20	15	5	0.05

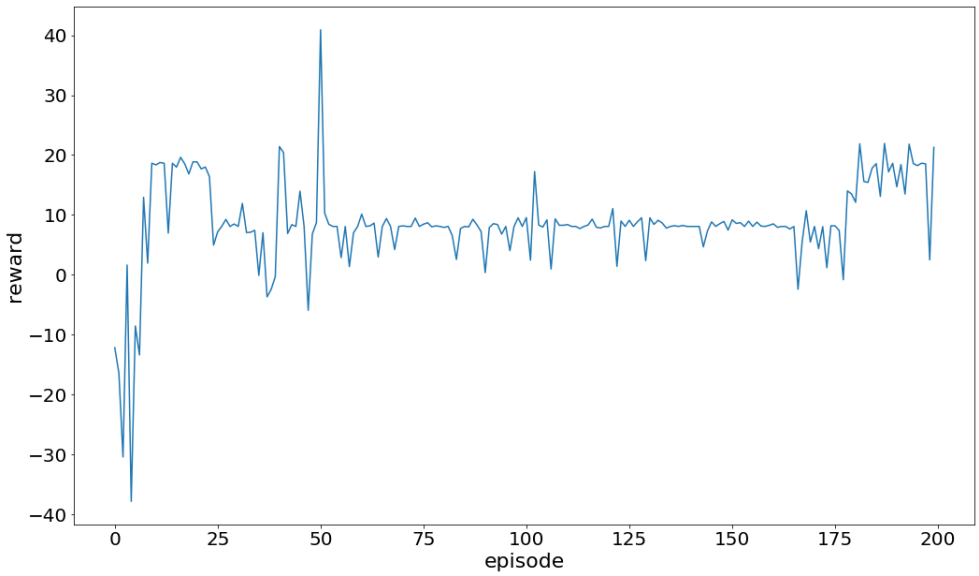

圖 11：測試 1.1.1 結果

由此次結果注意到模型發現保持不動能維持 Reward 在微小正值，因此放棄行動，儘管在 Action 上加入了雜訊來增加探索，效果仍然有限。

2. 測試 1.1.2

根據測試 1.1.1，發現應該將 Reward 修改為：

$$\mathrm{r_t} = w_x(v_x - v_{min}) - (w_y y^2 + w_z v_z^2) - w_\tau \sum_{j=1}^{JN} |\tau_j| \tag{15}$$

透過設置最低速度，確保模型能學習加速。

Reward Function 參數設置延續測試 1.1.1。

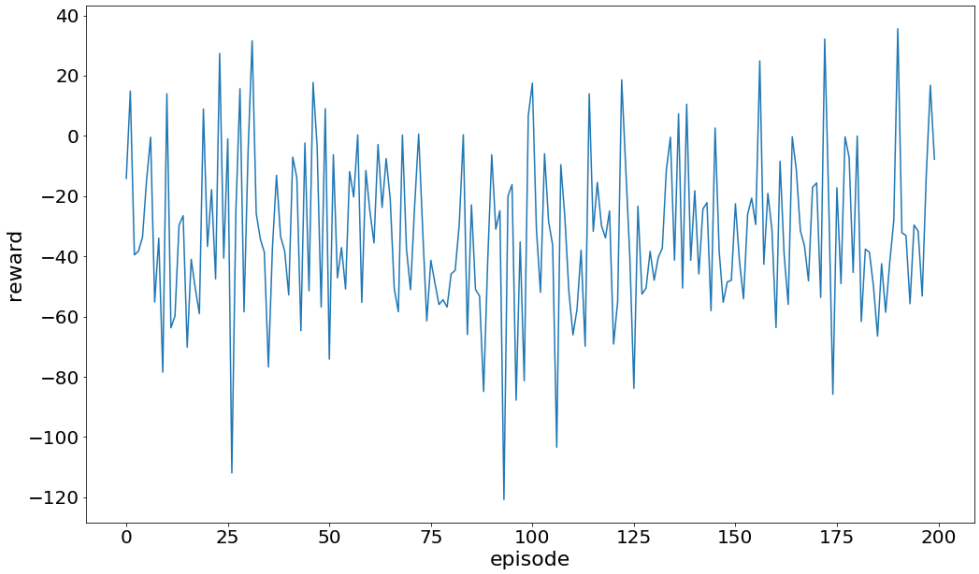

圖 12：測試 1.1.2 結果

不再陷入到靜止不動的狀態。

（二） Learning Rate 調整

1. 測試 1.2.1

模型中兩個神經網路有不同的學習率，觀察測試 1.1.2 結果，發現可能是 Learning Rate 需要調整，因此延續測試 1.1.2，紀錄 Actor 與 Critic 每個 Episode 中的神經元平均梯度大小：

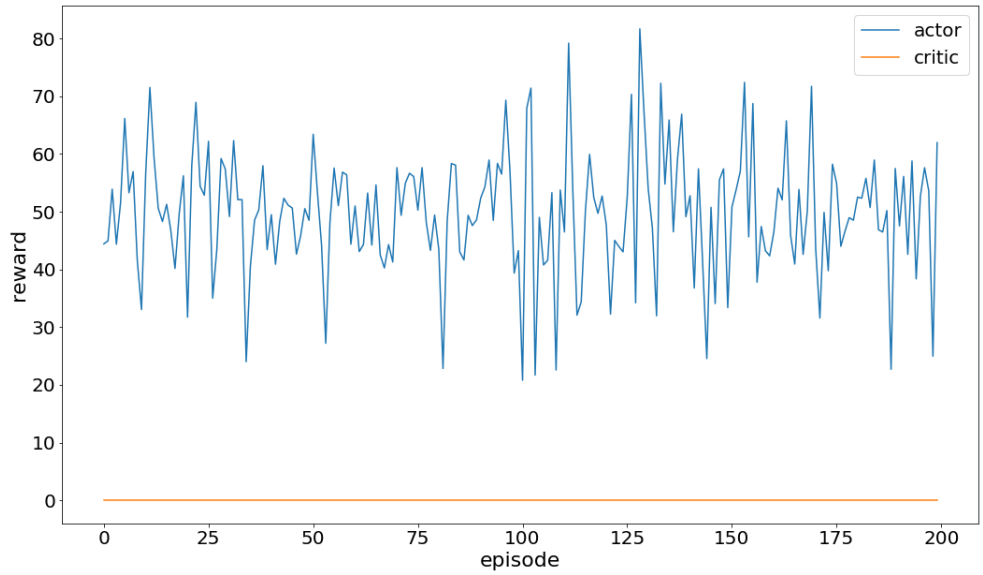

圖 13：測試 1.2.1 神經元平均梯度量值

Actor 平均梯度量值乘上學習率後數量級約為 10^{-2}，在可接受範圍。然而 Critic 平均梯度量值乘上學習率後數量級約為 10^{-6}，收斂速度極慢。因此設置新的學習率為：

表 7：修正後學習率

Actor	Critic
0.001	5

2. 測試 1.2.2

延續測試 1.2.1，利用修正過的學習率重新訓練得：

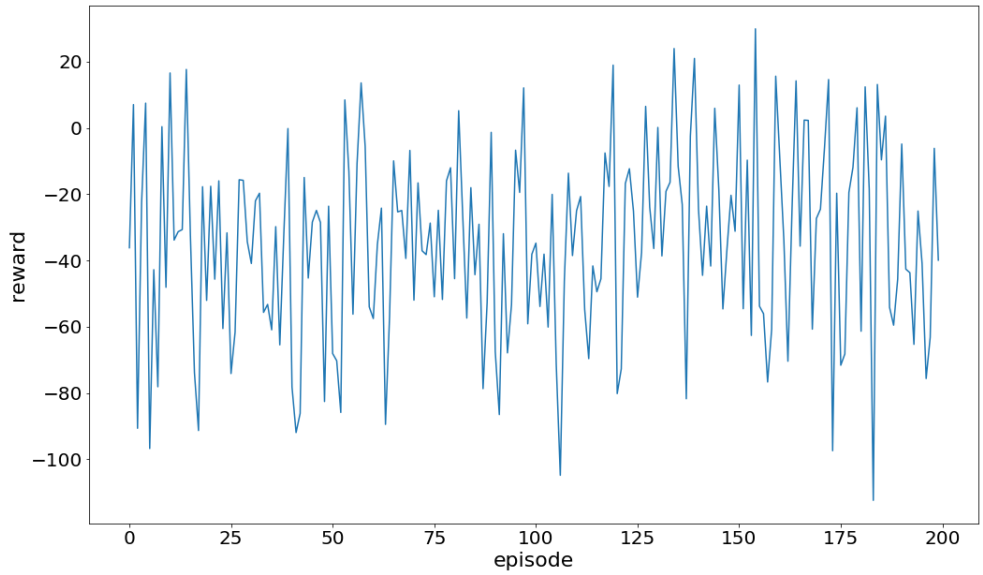

圖 14：測試 1.2.2

參、 研究結果與討論

初步研究結果如下：

表 8：初步研究結果

演算法建構	確立生成經驗之方法
	完成演算法流程設計
	完成模型架構
	建構評估經驗價值之方法
模型訓練	分析 Reward Function 之影響並修正
	分析 Learning Rate 之影響並修正

研究階段規劃如下：

表 9：研究階段規劃

	階段研究內容
正常訓練	分析神經網路結構之影響並修正
經驗生成	分析生成經驗時各參數之影響並修正
	分析經驗價值評估法之影響並修正
適應訓練	測試適應效果

肆、 結論與應用

伍、 參考文獻

一. 英文文獻

[1] Jeff Clune, Antoine Cully, Jean-Baptiste Mouret, & Danesh Tarapore, Robots that can adapt like animals, 2015. URL https://arxiv.org/abs/1407.3501

[2] Kevin Chavez, Augustus Hong, & Hao Yi Ong , Distributed Deep Q-Learning, 2015. URL https://arxiv.org/abs/1508.04186

[3] Shixiang Gu, Ethan Holly, Sergey Levine, & Timothy Lillicrap, Deep Reinforcement Learning for Robotic Manipulation with Asynchronous Off-Policy Updates, 2016. URL https://arxiv.org/abs/1610.00633

[4] Ioannis Antonoglou, John Quan, Tom Schaul, & David Silver, Prioritized Experience Replay, 2015. URL https://arxiv.org/abs/1511.05952

[5] Tom Erez, Nicolas Heess, Jonathan J. Hunt, Timothy P. Lillicrap, Alexander Pritzel, David Silver, Yuval Tassa, & Daan Wierstra, Continuous control with deep reinforcement learning, 2015. URL https://arxiv.org/abs/1509.02971

[6] Ioannis Antonoglou, Alex Graves, Koray Kavukcuoglu, Volodymyr Mnih, Martin Riedmiller, David Silver, & Daan Wierstra, Playing Atari with Deep Reinforcement Learning, 2013. URL https://arxiv.org/abs/1312.5602

二. 網路資源

[1] <Wikipedia. Reinforcement learning>, URL https://en.wikipedia.org/wiki/Reinforcement_learning

陸、 附錄

一. Python 程式碼

（一） DDPG

```
import tensorflow as tf
import numpy as np

from .networks.actor import Actor
from .networks.critic import Critic
from utils.memory import Memory

from tensorflow.keras.losses import MSE
from tensorflow.keras.optimizers import Adam

class DDPG(object):

    def __init__(self,
                 state_shape, action_shape,
                 action_range, state_range,
                 noise_std=0.05,
                 actor_lr=1e-4, critic_lr=1e-3,
                 actor_hidden_units=(300, 600), critic_hidden_units=(300, 600),
                 batch_size=128, discount=0.99, memory_size=1e6, tau=0.001):

        self.discount = discount
        self.batch_size = batch_size
        self.tau = tau

        self.state_shape = state_shape
        self.action_shape = action_shape

        self.action_range = action_range
        self.state_range = state_range

        self.noise_std = noise_std

        # create actor
        self.actor = Actor(state_shape=state_shape, action_shape=action_shape, hidden_units=a
ctor_hidden_units)
        self.target_actor = Actor(state_shape=state_shape, action_shape=action_shape, hidden_
units=actor_hidden_units)
        # create critic
        self.critic = Critic(state_shape=state_shape, action_shape=action_shape, hidden_units
=critic_hidden_units)
        self.target_critic = Critic(state_shape=state_shape, action_shape=action_shape, hidde
n_units=critic_hidden_units)

        self.actor_optimizer = Adam(learning_rate=actor_lr)
        self.critic_optimizer = Adam(learning_rate=critic_lr)

        self.memory = Memory(int(memory_size), action_shape, state_shape)

    def store_transition(self, state, action, reward, next_state, done):

        self.memory.append(state, action, reward, next_state, done)

```

```
49.     def step(self, state, apply_noise=True):
50.
51.         action = self.actor(state)
52.         if apply_noise:
53.             action += tf.random.normal(self.action_shape, stddev=self.noise_std)
54.         action = tf.clip_by_value(action, self.action_range[0], self.action_range[1])
55.
56.         return action
57.
58.     def train(self):
59.
60.         if len(self.memory) >= self.batch_size:
61.             states, actions, rewards, next_states, done = self.memory.sample(self.batch_size)
62.             ct = self.train_critic(states, actions, next_states, rewards, done)
63.             at = self.train_actor(states)
64.             self.update_target_models()
65.
66.             return ct, at
67.
68.     def train_critic(self, states, actions, next_states, rewards, done):
69.
70.         # q target
71.         next_actions = self.target_actor(next_states)
72.         next_q = self.target_critic(next_states, next_actions)
73.         q_targets = rewards + (1 - done) * self.discount * next_q
74.
75.         weights = self.critic.get_trainable_weights()
76.         with tf.GradientTape() as tape:
77.             tape.watch(weights)
78.             # compute MSE loss
79.             q_values = self.critic.model([states, actions])
80.             loss = MSE(q_targets, q_values)
81.         # compute gradients
82.         grads = tape.gradient(loss, weights)
83.
84.         tot = 0
85.         for w in grads:
86.             tot += tf.math.reduce_mean(tf.abs(w))
87.         tot /= len(grads)
88.
89.         self.critic_optimizer.apply_gradients(zip(grads, weights))
90.         return np.asarray(tot)
91.
92.     def train_actor(self, states):
93.
94.         weights = self.actor.get_trainable_weights()
95.         with tf.GradientTape() as tape:
96.             tape.watch(weights)
97.             # compute policy value
98.             actions = self.actor(states)
99.             q_values = self.critic(states, actions)
100.                 loss = -tf.math.reduce_mean(q_values)
101.             # compute policy gradients
102.             grads = tape.gradient(loss, weights)
103.
104.             self.actor_optimizer.apply_gradients(zip(grads, weights))
105.             return np.asarray(tot)
106.
```

```
    def initialize(self):

        actor = self.actor.get_trainable_weights()
        target_actor = self.target_actor.get_trainable_weights()
        for weight, target_weight in zip(actor, target_actor):
            target_weight.assign(weight)

        critic = self.critic.get_trainable_weights()
        target_critic = self.target_critic.get_trainable_weights()
        for weight, target_weight in zip(critic, target_critic):
            target_weight.assign(weight)

    def update_target_models(self):

        actor = self.actor.get_trainable_weights()
        target_actor = self.target_actor.get_trainable_weights()
        for weight, target_weight in zip(actor, target_actor):
            target_weight.assign((1. - self.tau) * target_weight + self.tau * weight)

        critic = self.critic.get_trainable_weights()
        target_critic = self.target_critic.get_trainable_weights()
        for weight, target_weight in zip(critic, target_critic):
            target_weight.assign((1. - self.tau) * target_weight + self.tau * weight)
```

（二） Actor

```
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Dense, Input

class Actor(object):

    def __init__(self, state_shape, action_shape, hidden_units=(300, 600)):

        # store parameters
        self.state_shape = state_shape
        self.action_shape = action_shape
        self.hidden = hidden_units

        # generate model
        self.model = self.generate_model()

    def __call__(self, state):

        return self.model(state)

    def get_weights(self):

        return self.model.weights

    def get_trainable_weights(self):

        return self.model.trainable_weights

    def set_weights(self, weights):

        self.model.set_weights(weights)

```

```
    def generate_model(self):

        # input state
        input_layer = Input(shape=self.state_shape,
                            name="actor_input_state")
        # hidden layer1
        layer = Dense(self.hidden[0], activation='relu',
                      kernel_initializer='random_normal',
                      bias_initializer='zeros',
                      name="actor_hedden1")(input_layer)
        # hidden layer2
        layer = Dense(self.hidden[1], activation='relu',
                      kernel_initializer='random_normal',
                      bias_initializer='zeros',
                      name="actor_hedden2")(layer)
        # output layer
        output_layer = Dense(self.action_shape[0], activation='sigmoid',
                             kernel_initializer='random_normal',
                             bias_initializer='zeros',
                             name="actor_output")(layer)
        # create model
        model = Model(inputs=input_layer, outputs=output_layer)

        return model
```

（三） Critic

```
from tensorflow.keras.layers import Dense, Input, Add
from tensorflow.keras.models import Model

class Critic(object):

    def __init__(self, state_shape, action_shape, hidden_units=(300, 600)):

        # store parameters
        self.state_shape = state_shape
        self.action_shape = action_shape
        self.hidden = hidden_units

        # generate model
        self.model = self.generate_model()

    def __call__(self, state, action):

        return self.model([state, action])

    def get_trainable_weights(self):

        return self.model.trainable_weights

    def set_weights(self, weights):

        self.model.set_weights(weights)

    def get_weights(self):

        return self.model.weights

```

```
    def generate_model(self):

        # input state
        s_input_layer = Input(shape=self.state_shape,
                              name="critic_input_state")
        # input action
        a_input_layer = Input(shape=self.action_shape,
                              name="critic_input_action")
        # state hidden layer
        s_layer = Dense(self.hidden[0], activation='relu',
                        kernel_initializer='random_normal',
                        bias_initializer='zeros',
                        name="critic_state_hidden_layer")(s_input_layer)
        # action hidden layer
        a_layer = Dense(self.hidden[0], activation='linear',
                        kernel_initializer='random_normal',
                        bias_initializer='zeros',
                        name="critic_action_hidden_layer")(a_input_layer)
        # add state hidden layer and action hidden layer
        hidden = Add()([s_layer, a_layer])
        # hidden layer2
        hidden = Dense(self.hidden[1], activation='relu',
                       kernel_initializer='random_normal',
                       bias_initializer='zeros',
                       name="critic_hidden_layer2")(hidden)
        # output layer
        output_layer = Dense(1, activation='linear',
                             kernel_initializer='random_normal',
                             bias_initializer='zeros',
                             name="critic_output")(hidden)
        # create model
        model = Model(inputs=[s_input_layer, a_input_layer],
                      outputs=output_layer)

        return model
```

（四） Environment

```
import numpy as np
from .simAPI import sim

_oneshot = sim.simx_opmode_oneshot
_blocking = sim.simx_opmode_blocking

PI = 3.14159265358979
DEG2RAD = PI / 180

```

```python
class Environment(object):

    def __init__(self, state_shape, action_shape,
                 dt=0.010, frames=10,
                 min_vel = 0.10):

        self.state_shape = state_shape
        self.action_shape = action_shape

        self.dt = dt
        self.frames = frames

        self.min_vel = min_vel

        self.body_name = "QuadSpider"

        self.joint_num = 8
        self.max_joint_torque = 10
        self.max_joint_velocity = 360 * DEG2RAD

        self.w_vx = 20
        self.w_y = 15
        self.w_z = 5

        self.w_tor = 0.05

    def open(self):

        # clear communications
        sim.simxFinish(-1)
        # create communication
        self.client_ID = sim.simxStart('127.0.0.1', 19997, True, True, 5000, 5)

        self.stop_sim()

        # read body's handle
        _, self.body_handle = sim.simxGetObjectHandle(self.client_ID, self.body_name, _blocki
ng)

        # read joints' handless
        self.joint_handles = []
        for i in range(self.joint_num):
            _, res = sim.simxGetObjectHandle(self.client_ID, "joint" + str(i), _blocking)
            self.joint_handles.append(res)

        sim.simxGetPingTime(self.client_ID)

    def close(self):

        # stop simulation before close communication
        self.stop_sim()
        # close communication
        sim.simxFinish(self.client_ID)

        sim.simxGetPingTime(self.client_ID)

    def start_sim(self):

        # set synchronous
        sim.simxSynchronous(self.client_ID, True)
        # start simulation
        sim.simxStartSimulation(self.client_ID, _oneshot)
        # trigger once
        sim.simxSynchronousTrigger(self.client_ID)

        sim.simxGetPingTime(self.client_ID)

```

```
    def stop_sim(self):

        sim.simxStopSimulation(self.client_ID, _oneshot)

        sim.simxGetPingTime(self.client_ID)

    def initialize(self):
        for i in range(self.joint_num):
            # set the direction of target velocity
            sim.simxSetJointTargetVelocity(self.client_ID, self.joint_handles[i],
                                           self.max_joint_velocity, _oneshot)
            sim.simxSetJointForce(self.client_ID, self.joint_handles[i],
                                  self.max_joint_torque, _oneshot)

        _, position = sim.simxGetObjectPosition(self.client_ID, self.body_handle, -
1, _blocking)
        self.height = position[2]

        self.time = 0

    def act(self, act):

        # pause communication until all commands are sent
        sim.simxPauseCommunication(self.client_ID, True)

        act = act[0]
        # set target position
        for i in range(self.joint_num):
            tor = (act[i] - 0.5) * 2 * self.max_joint_torque
            # set the direction of target velocity
            if tor >= 0:
                 sim.simxSetJointTargetVelocity(self.client_ID, self.joint_handles[i],
                                                self.max_joint_velocity, _oneshot)
                 sim.simxSetJointForce(self.client_ID, self.joint_handles[i],
                                       tor, _oneshot)
            else:
                 sim.simxSetJointTargetVelocity(self.client_ID, self.joint_handles[i],
                                                -self.max_joint_velocity, _oneshot)
                 sim.simxSetJointForce(self.client_ID, self.joint_handles[i],
                                       -tor, _oneshot)

        # restart communication
        sim.simxPauseCommunication(self.client_ID, False)
        # trigger once
        for i in range(self.frames):
            sim.simxSynchronousTrigger(self.client_ID)
            sim.simxGetPingTime(self.client_ID)
            self.time += self.dt

    def get_reward(self, velocity, position, torques):

        r_vx = self.w_vx * velocity[0]
        r_y = self.w_y * position[1] * position[1]
        r_z = self.w_z * (position[2] - self.height) * (position[2] - self.height)

        r_tor = self.w_tor * np.square(torques).sum()

        return r_vx - r_y - r_z - r_tor

```

```
    def is_done(self, velocity, position, orientation):

        if position[0] >= 3:
            return True

        if abs(orientation[0]) >= PI / 2:
            return True
        if abs(orientation[1]) >= PI / 2:
            return True
        if abs(orientation[2]) >= PI / 2:
            return True

        if self.time >= 2:
            if position[0] < 0:
                return True
            if abs(position[1]) > 0.2:
                return True
            if velocity[0] < 0.05:
                return True

        return False

    def get_state(self):

        # read body position
        _, position = sim.simxGetObjectPosition(self.client_ID, self.body_handle, -
1, _blocking)
        position = np.array(position)
        # read body velocity
        _, velocity, _ = sim.simxGetObjectVelocity(self.client_ID, self.body_handle, _blockin
g)
        velocity = np.array(velocity)
        # read body orientation
        _, orientation = sim.simxGetObjectOrientation(self.client_ID, self.body_handle, -
1, _blocking)
        orientation = np.array(orientation)

        joint_torques = []
        joint_velocities = []
        for i in range(self.joint_num):
            # read joint force
            _, tor = sim.simxGetJointForce(self.client_ID, self.joint_handles[i], _blocking)

            # read joint velocity
            _, vel = sim.simxGetObjectFloatParameter(self.client_ID, self.joint_handles[i], s
im.sim_jointfloatparam_velocity, _blocking)
            vel *= DEG2RAD
            # record joint state
            joint_torques.append(tor)
            joint_velocities.append(vel)

        joint_torques = np.array(joint_torques)
        joint_velocities = np.array(joint_velocities)

        sim.simxGetPingTime(self.client_ID)

        return np.concatenate([position, velocity, orientation,
                               joint_torques, joint_velocities]).reshape((1, *self.state_shap
e)), \
               self.get_reward(velocity, position, joint_torques), \
               self.is_done(velocity, position, orientation)
```