Reinforcement Learning

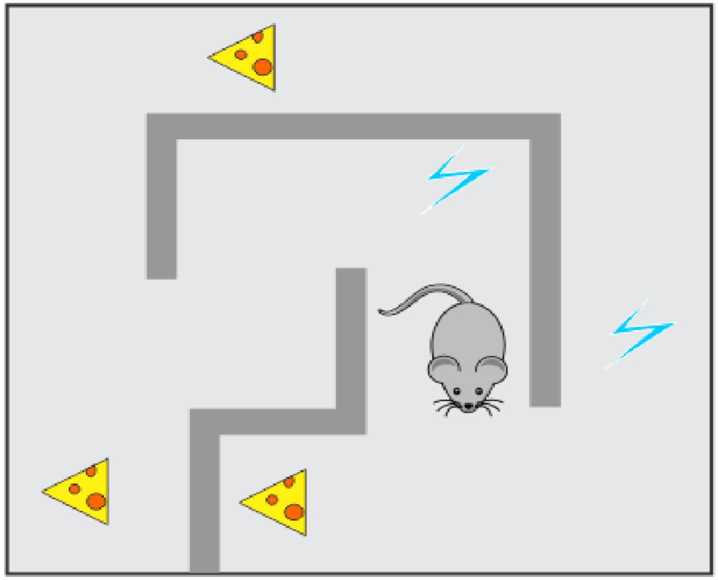
Outline

- Introduction
- RL for maze problem
- The Markov Decision Process (MDP)
- SARSA method
- Q-Learning
- Deep Q Network (DQN)
- Keras RL model

Introduction

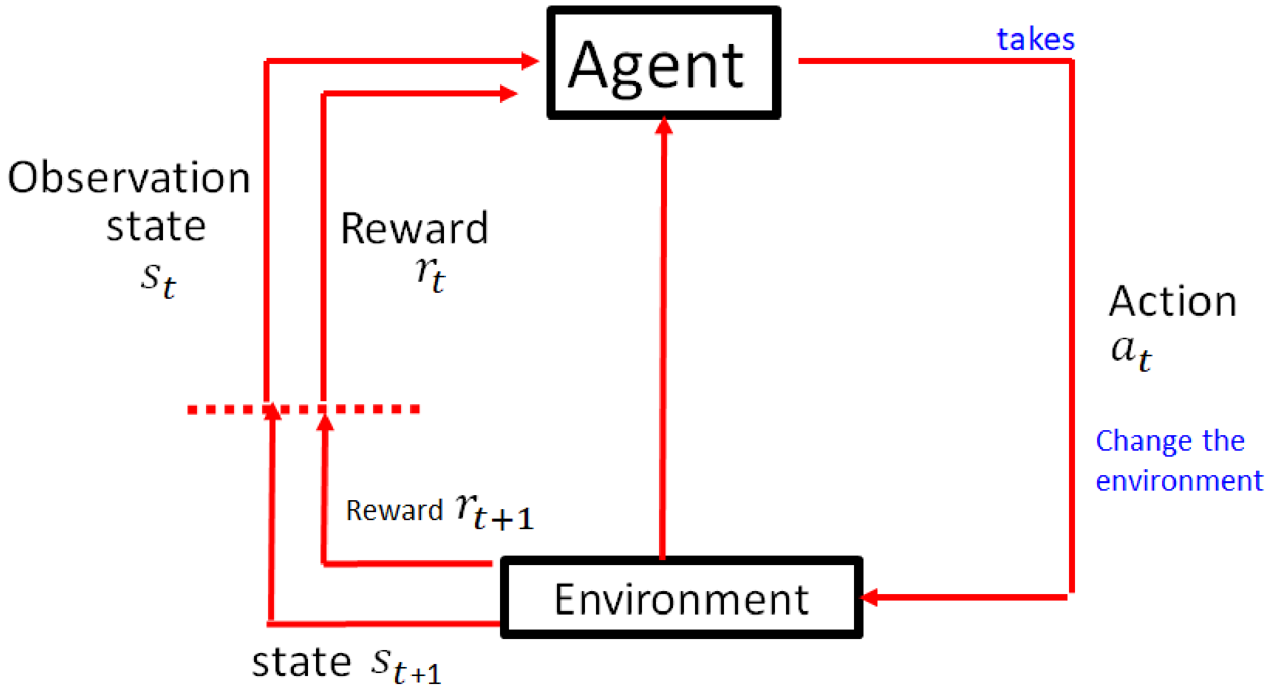
Reinforcement learning (RL)

- A subfield of **machine learning** (**ML**),
- Addresses the problem of **the automatic learning** of optimal decisions over time,
- Based on the **operant conditioning** (操作制約學習) Skinner Box 史金納箱
- This is a general and common problem that has been studied in many scientific and engineering fields.



Reinforcement Learning agent

- interacts with its environment,
- The environment itself could demonstrate multiple states.
- The agent acts upon the environment to change the environment's **state**, thereby also receiving a **reward** or **penalty** as determined by the achieved state and the objective of the agent.



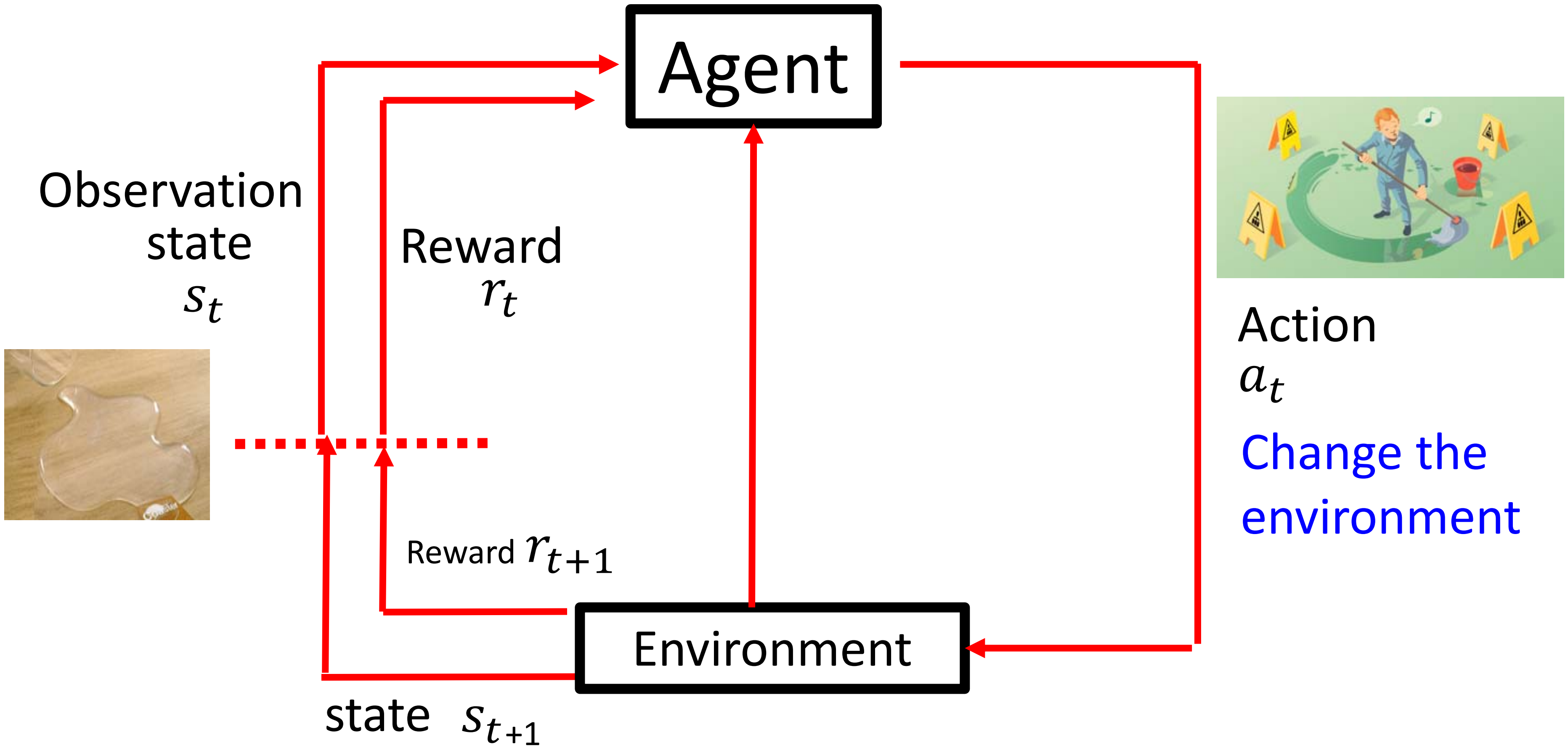
The Reward and a Good Reward Function

- **Future reward**
  - The actual reward for a right action taken in a particular state may not be realized **immediately.**
  - For example: Go out to play vs. sit and study for exam
    - Play get immediate reward (may be taking action).
    - Study seem boring but well in the **long run**, the reward is realized only in the future.
  - using a discounting-factor to discount the future rewards to present time
- **The probabilistic nature of the rewards or uncertainty in the rewards**.

Reward

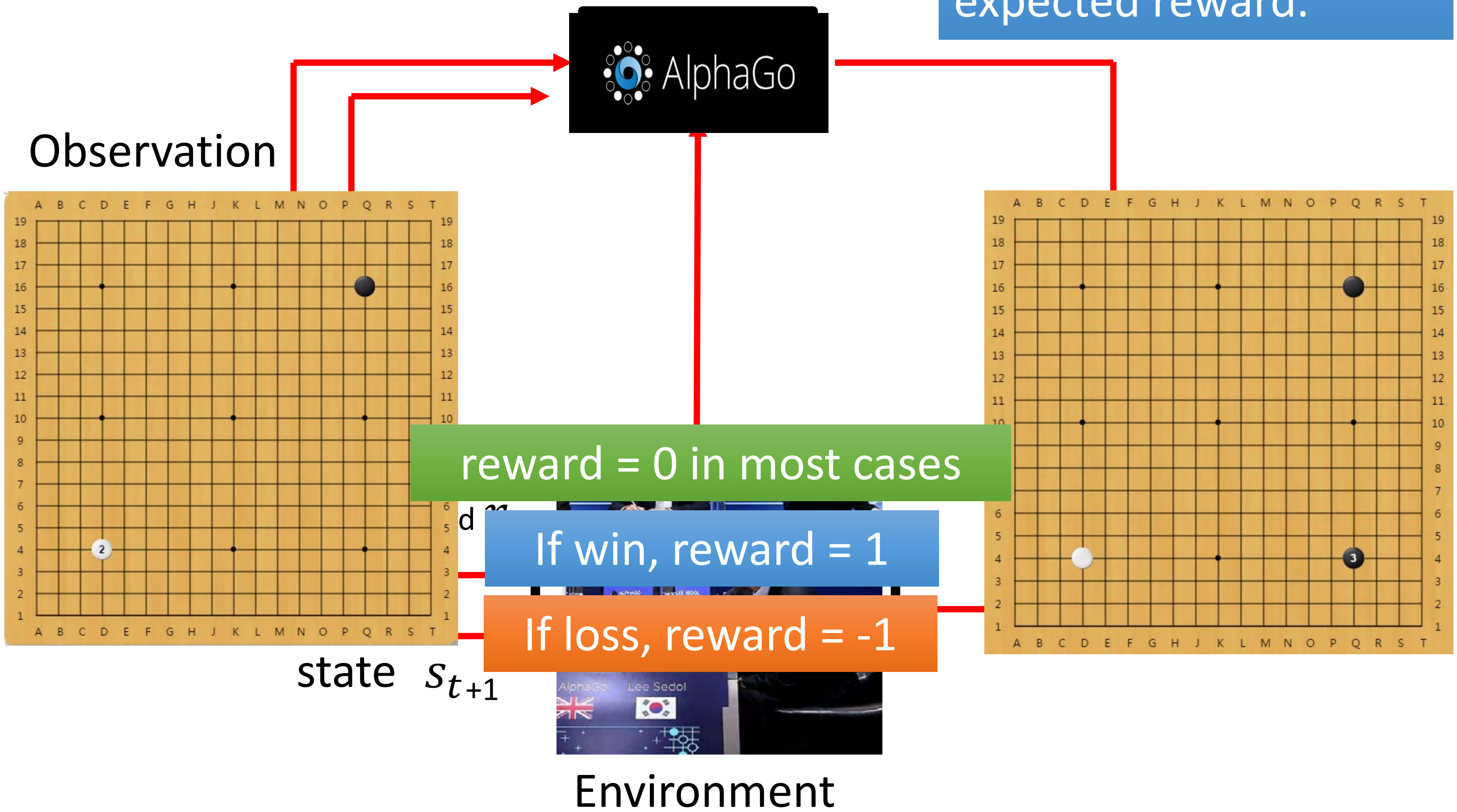
- Attribution of rewards to **different actions** taken in the past
- Determining a **good reward function**
  - Absolute and percentage scores
- Dealing with **different types** of reward
  - Different unit and scale
  - Device a conversion function between two scales
- Domain aspects and solutions to the reward problem
  - Real-time problem to RL model

Reinforcement Learning



Learning to play Go

Agent learns to take actions maximizing expected reward.



The **Agent** in Reinforcement Learning

- The agent needs to decide which is **the best action** it can take when facing a specific state.
  - It focuses on identifying which is the next best state to be in (reachable from the current state) as determined from the history of present and future rewards that the agent has received when it was in this particular state earlier.
  - We could extend this logic to similar state, and that is where a lot of learning to convert a state into a representative function will come.
  - We are trying to **predict the "Value" (or utility) of any state (or state–action combination**), even the unseen ones, based on the ones that we have seen.
  - This value could be a function of all present and (discounted) future rewards that could be attributed to being in this state.

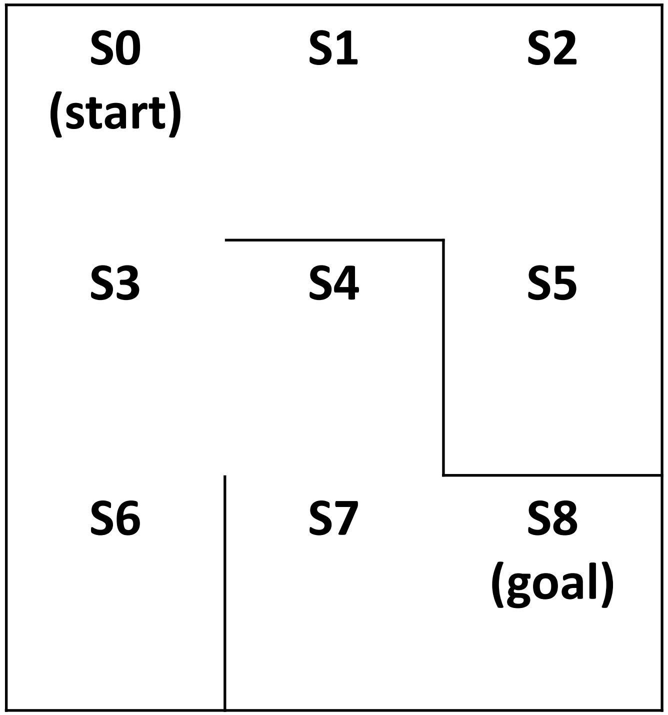
The (State) Value Function *V*(s),

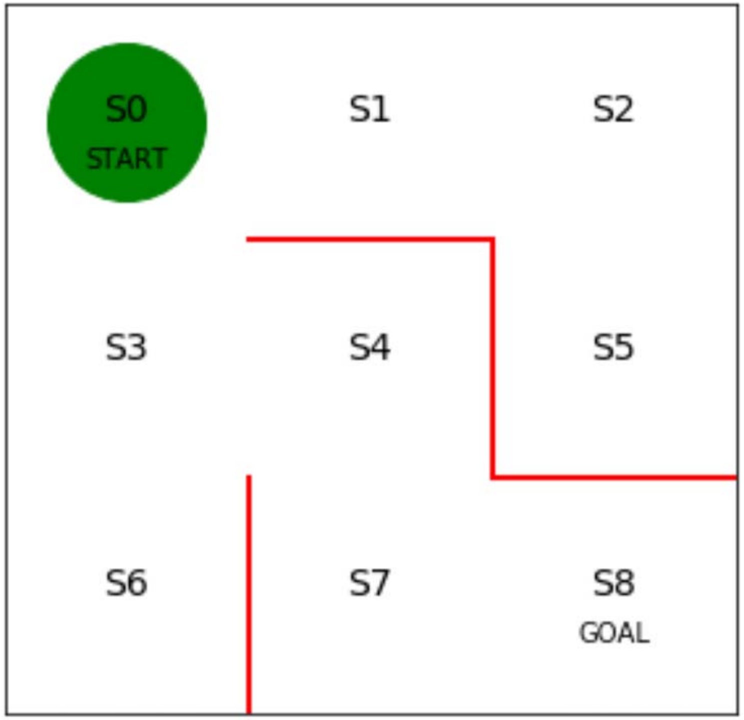
- Value *V(s)* is a function of the state s.
- Dealing with future/delayed and probabilistic/uncertain rewards by converting the different rewards into one homogenous function tied to a state that the agent will try to learn.
- Then the agent recommends an action that will transition it to the identified most lucrative state from where it will have to decide again on the next most lucrative state reachable from this new state
- The underlying training of the agent tries to learn this very important "Value Function" that could represent the most accurate possible "Value" of each state using the training data/experiments.

以強化學習建置迷宮

Maze Environment 3x3

- Agent can reach the goal.





Permissible State Transitions in Grid-World

- In the example of grid world, the agent from a given state/cell could move only to the **adjacent**, **non-diagonal** cells within the grid from a particular cell in a given turn.
- From a given state, the agent can move in either of the four directions, **UP (U), DOWN (D), LEFT (L), or RIGHT (R)**.
- If there exists a valid state on taking these actions, the state is **transitioned** to that valid state, else it remains the same.

Recipes to Build Our Own Custom Environment Class

- These two functions are the "step" and "reset" methods/functions which are explained below.
- **The Step () Method**
  - We need a mechanism to present to it a **state**; the agent then takes the best possible **action** possible for that state.
  - We need a mechanism to give the agent a **reward/penalty** corresponding to that action, and to change the state that occurs because of the action.
- The step method on receiving the above inputs processes them to returns a tuple in the format **(observation, reward, done, info).**

(observation, reward, done, info).

- **Observation (object)**
  - This variable constitutes the **(new/next) state** that is returned from the environment on taking the particular action by the agent (as sent in the step method's input).
  - This state could have observations in the way best represented in the environment.
- **Reward (float)**
  - This is the **instantaneous reward** received by the agent for reaching the particular new state on taking the action (input).
- **Done (boolean)**
  - which deals with episodes.
  - An **episode** is a series of experiments/turns that has a beginning and an end.
- **Info (dict)**
  - This is an optional parameter and is used to share the information required for debugging.

The Reset () Method

- Whenever the environment is **instantiated for the first time**, or whenever a **new episode starts**, the state of the environment needs to be reset.
- The reset function takes no argument and returns an observation/state corresponding to the start of a fresh episode.
- Depending upon specific environments, other internal variables that need to be reset/instantiated for a fresh episode's start are also reset in this function.

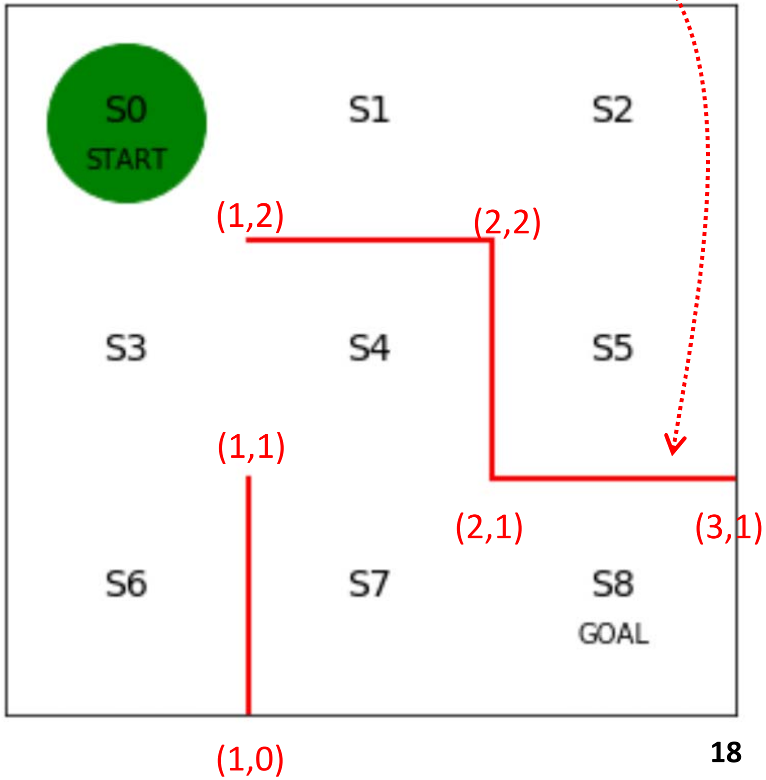
建構迷宮

```
# parameters
# plot(x, y, color='green', linestyle='dashed', marker='o',
#  markerfacecolor='blue', markersize=12).
[1,1], [0,1]代表座標 (1,0)->(1,1)
```

```python
# 宣告使用的套件
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
```

```python
# 迷宮的初始狀態

# 宣告圖的大小與圖的變數名稱
fig = plt.figure(figsize=(5, 5))
ax = plt.gca()

# 繪製紅色牆壁
plt.plot([1, 1], [0, 1], color='red', linewidth=2)
plt.plot([1, 2], [2, 2], color='red', linewidth=2)
plt.plot([2, 2], [2, 1], color='red', linewidth=2)
plt.plot([2, 3], [1, 1], color='red', linewidth=2)

# 繪製代表狀態的文字S0~S8
plt.text(0.5, 2.5, 'S0', size=14, ha='center')
plt.text(1.5, 2.5, 'S1', size=14, ha='center')
plt.text(2.5, 2.5, 'S2', size=14, ha='center')
plt.text(0.5, 1.5, 'S3', size=14, ha='center')
plt.text(1.5, 1.5, 'S4', size=14, ha='center')
plt.text(2.5, 1.5, 'S5', size=14, ha='center')
plt.text(0.5, 0.5, 'S6', size=14, ha='center')
plt.text(1.5, 0.5, 'S7', size=14, ha='center')
plt.text(2.5, 0.5, 'S8', size=14, ha='center')
plt.text(0.5, 2.3, 'START', ha='center')
plt.text(2.5, 0.3, 'GOAL', ha='center')
```

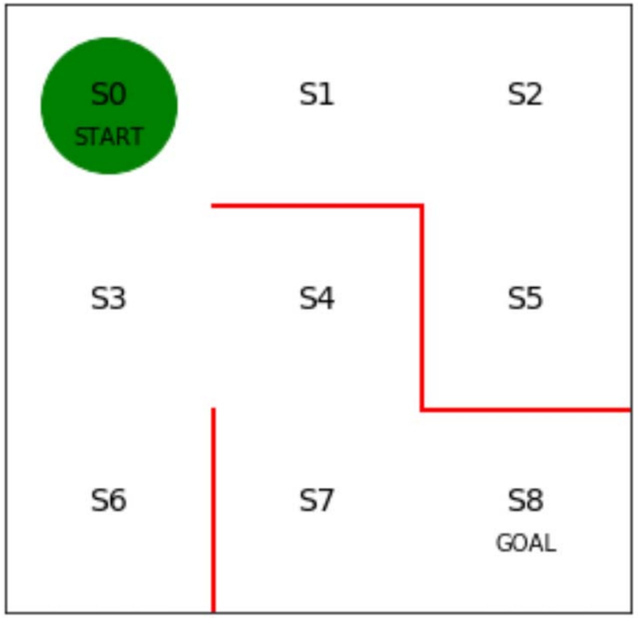
當前的圖形 width/height 5 inches
當前的軸



建構迷宮

參數設定參考
https://kknews.cc/zh-tw/code/ea286yz.html

```
# 設定繪圖範圍與塗銷刻度
ax.set_xlim(0, 3)          # Set the x-axis view limits.
ax.set_ylim(0, 3)          # Set the x-axis view limits.
plt.tick_params(axis='both', which='both', bottom='off', top='off',
                labelbottom='off', right='off', left='off', labelleft='off')
#參數刻度線顯示設定 軸，刻度線，
# 於目前位置S0繪製綠色圓形
line, = ax.plot([0.5], [2.5], marker="o", color='g', markersize=60)
```



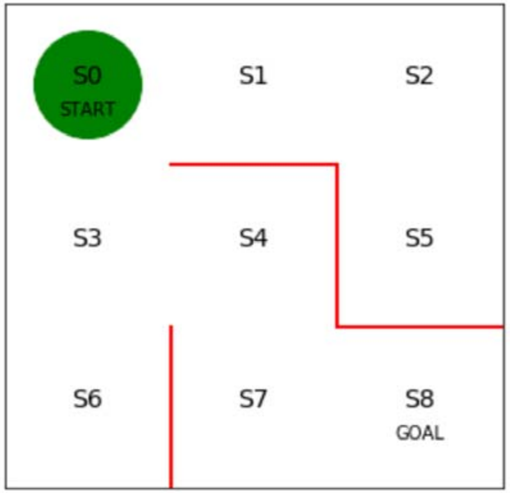
Policy: π_θ(s, a)

- Policy: π_θ(s, a), is a probability 於狀態s時採用行動a之機率。
  - s: state
  - a: action
  - θ: parameter of the policy
- Policy can be represented by a **table**, **function** or a **deep neural network (DNN)**

θ and possible actions of state

- Agent can move within maze until it reaches the goal state.
- 每一個state (s0-s7) 可以有四種actions (依序為)上、右、下、左，以一個陣列表示。
  - 依所給定的環境，以陣列表示，
  - **1**代表該行可以執行，np.nan代表不可行(空白缺損值)。
- S8為goal state不需要actions
- **Policy**(策略代表行動的方式，此例為random)

```
# 設定一開始採用何種策略的參數theta_0

# 列為狀態0~7、欄移動方向的↑、→、↓、←
theta_0 = np.array([[np.nan, 1, 1, np.nan],  # s0
                    [np.nan, 1, np.nan, 1],  # s1
                    [np.nan, np.nan, 1, 1],  # s2
                    [1, 1, 1, np.nan],  # s3
                    [np.nan, np.nan, 1, 1],  # s4
                    [1, np.nan, np.nan, np.nan],  # s5
                    [1, np.nan, np.nan, np.nan],  # s6
                    [1, 1, np.nan, np.nan],  # s7、※s8是終點，所以不需採用任何策略
                    ])
```



Using θ to find policy

```
([[np.nan, 1, 1, np.nan],  # s0
  [np.nan, 1, np.nan, 1],  # s1
  [np.nan, np.nan, 1, 1],  # s2
  [1, 1, 1, np.nan],  # s3
  [np.nan, np.nan, 1, 1],  # s4
  [1, np.nan, np.nan, np.nan],  # s
  [1, np.nan, np.nan, np.nan],  # s
  [1, 1, np.nan, np.nan],  # s7、※
  ])
```

```
# 自訂策略的參數theta轉換成行動策略pi的函數


def simple_convert_into_pi_from_theta(theta):
    '''單純地計算比例'''

    [m, n] = theta.shape  # 取得theta的矩陣大小
    pi = np.zeros((m, n)) # 設定陣列初值均為0
    for i in range(0, m):
        pi[i, :] = theta[i, :] / np.nansum(theta[i, :])  # 計算比例

    pi = np.nan_to_num(pi)  # 將nan轉換成0   # 計算總和(有缺值nan)

    return pi
```

```
# 算出初始策略pi_0
pi_0 = simple_convert_into_pi_from_theta(theta_0)
```

- print(pi_0)

```
[[0.         0.5        0.5        0.        ]
 [0.         0.5        0.         0.5       ]
 [0.         0.         0.5        0.5       ]
 [0.33333333 0.33333333 0.33333333 0.        ]
 [0.         0.         0.5        0.5       ]
 [1.         0.         0.         0.        ]
 [1.         0.         0.         0.        ]
 [0.5        0.5        0.         0.        ]]
```

於policy θ中選出state s的action並傳回下一個狀態 s_next

```
# 自訂計算1step移動後的狀態s的函數

def get_next_s(pi, s):
    direction = ["up", "right", "down", "left"]

    next_direction = np.random.choice(direction, p=pi[s, :])
    # 根據pi[s,:]的機率、選定direction

    if next_direction == "up":
        s_next = s - 3  # 往上移動時，讓代表狀態的數字減少3
    elif next_direction == "right":
        s_next = s + 1  # 往右移動時，讓代表狀態的數字加1
    elif next_direction == "down":
        s_next = s + 3  # 往下移動時，讓代表狀態的數字加3
    elif next_direction == "left":
        s_next = s - 1  # 往左移動時，讓代表狀態的數字減1

    return s_next
```

numpy.random.choice(a, size=None, replace=True, p=None)
Generates a random sample from **a** given 1-D array

s所在的列

Goal Test function

```python
# 自訂代理器在迷宮之內不斷移動，直到抵達終點為止的函數


def goal_maze(pi):
    s = 0  # 起點
    state_history = [0]  # 記錄代理器移動軌跡的list

    while (1):  # 持續移動，直到抵達終點的迴圈
        next_s = get_next_s(pi, s)
        state_history.append(next_s)  # 在記錄list追加下一個狀態（代理器的位置）

        if next_s == 8:  # 若抵達終點就結束程式
            break
        else:
            s = next_s

    return state_history
```

```python
# 在迷宮內部往終點移動
state_history = goal_maze(pi_0)
```

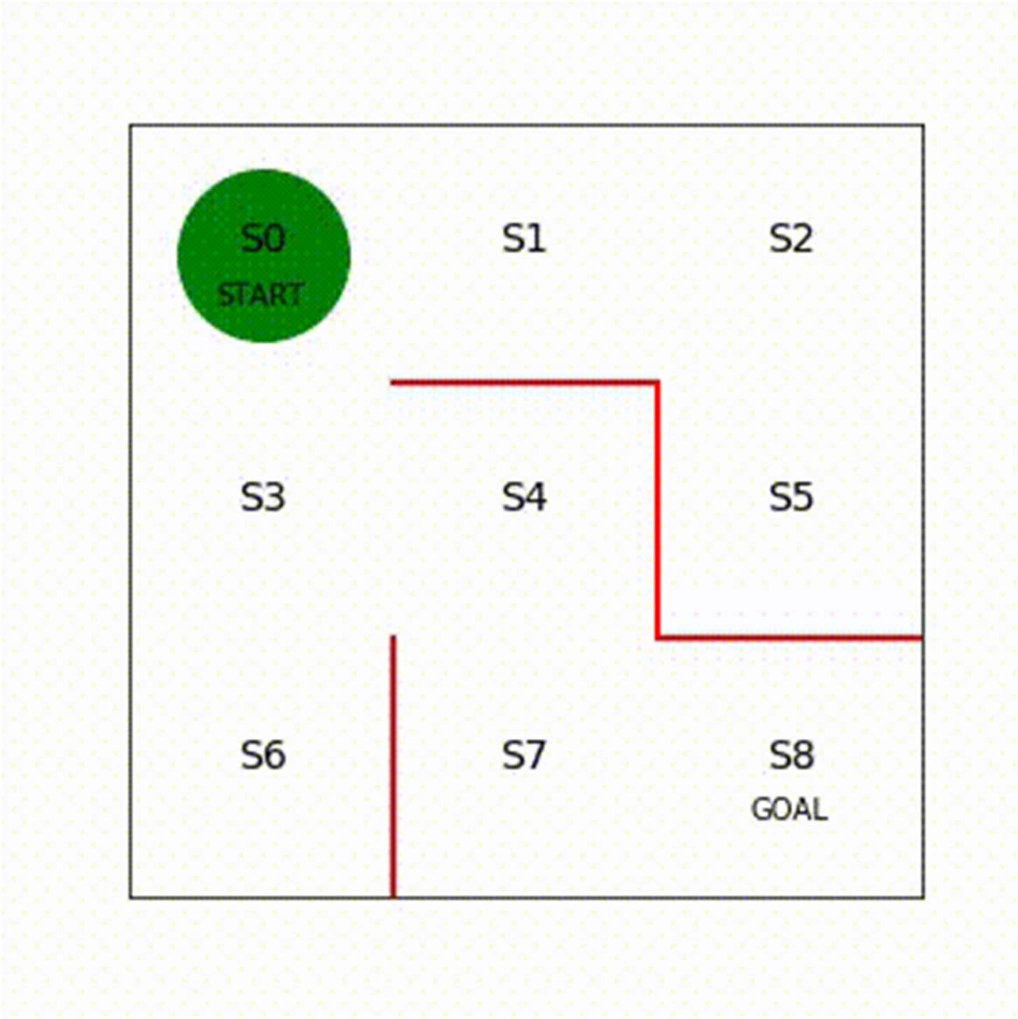
```
[0, 3, 4, 7, 4, 7, 4, 7, 8]
走出迷宮的總步數為8喲
```

```python
print(state_history)
print("走出迷宮的總步數為" + str(len(state_history) - 1) + "喲")
```

將軌跡轉成動畫gif

```
# 將代理器移動軌跡畫成動畫
# 參考URL http://louistiao.me/posts/notebooks/embedding-matplotlib-animations-in-jupyter-notebooks/
from matplotlib import animation
from IPython.display import HTML


def init():
    '''初始化背景影像'''
    line.set_data([], [])
    return (line,)


def animate(i):
    '''每一個的繪圖內容'''
    state = state_history[i]  # 繪製目前的位置
    x = (state % 3) + 0.5  # 狀態的x座標以3除之，再於得到的餘數+0.5
    y = 2.5 - int(state / 3)  # y座標以3除之，再以2.5減去商數
    line.set_data(x, y)
    return (line,)


#  利用初始化函數與每格影格的繪圖函數繪製動畫
anim = animation.FuncAnimation(fig, animate, init_func=init, frames=len(
    state_history), interval=200, repeat=False)

HTML(anim.to_jshtml())
```

https://riptutorial.com/zh-TW/matplotlib/example/23558/使用funcanimation的基本動畫


S0
START
S1
S2
S3
S4
S5
S6
S7
S8
GOAL

Policy-iteration Methods

- 使得agent 可以學得朝終點直接前進
  - **策略迭代法(Policy iteration)**：關注儘早到達終點的**行動(action)**，不斷採用較佳的行動來更新策略。
  - **價值迭代法(Value iteration)**：從終點逆推，依序將agent誘導到**終點前一個或前兩個狀態**。終點以外的狀態，賦予一個值(或優先度)，依據狀態優先選擇。

- **策略梯度法(Policy Gradient Method)**
  - 為**策略迭代法(Policy iteration)**的一種。
  - Policy: π_θ(s, a), is a probability 於狀態s時採用行動a之機率。
    - s: state,
    - a: action,
    - θ:parameter of the policy
  - 採用轉換函數softmax P(θ_i)= $\frac{\exp(\beta\theta_i)}{\sum_{j=1}^{N_a}\exp(\beta\theta_j)}$
  - N_a代表可以執行的動作(actions)數量

Policy Gradient Method

$$P(\theta_i)=\frac{\exp(\beta\theta_i)}{\sum_{j=1}^{N_a}\exp(\beta\theta_j)}$$

```
# 定義利用softmax函數將策略參數theta轉換成行動策略pi的手法

def softmax_convert_into_pi_from_theta(theta):
    '''以softmax函數計算比例'''

    beta = 1.0
    [m, n] = theta.shape  # 取得theta的矩陣大小
    pi = np.zeros((m, n))

    exp_theta = np.exp(beta * theta)  # 將theta轉換成exp(theta)

    for i in range(0, m):
        # pi[i, :] = theta[i, :] / np.nansum(theta[i, :])
        # 於simpleに計算比例的情況

        pi[i, :] = exp_theta[i, :] / np.nansum(exp_theta[i, :])
        # 以softmax計算的情況

    pi = np.nan_to_num(pi)  # 將nan轉換成0

    return pi
```

policy

```
([[np.nan, 1, 1, np.nan],  # s0
  [np.nan, 1, np.nan, 1],  # s1
  [np.nan, np.nan, 1, 1],  # s2
  [1, 1, 1, np.nan],  # s3
  [np.nan, np.nan, 1, 1],  # s4
  [1, np.nan, np.nan, np.nan],  # s
  [1, np.nan, np.nan, np.nan],  # s
  [1, 1, np.nan, np.nan],  # s7、※
  ])
```

```
[[0.    0.5   0.5   0.   ]
 [0.    0.5   0.    0.5  ]
 [0.    0.    0.5   0.5  ]
 [0.333 0.333 0.333 0.   ]
 [0.    0.    0.5   0.5  ]
 [1.    0.    0.    0.   ]
 [1.    0.    0.    0.   ]
 [0.5   0.5   0.    0.   ]]
```

get_action_and_next_s

```
# 定義計算行動a與1step移動後的狀態s的函數


def get_action_and_next_s(pi, s):
    direction = ["up", "right", "down", "left"]
    # 依照pi[s,:]的機率選擇direction
    next_direction = np.random.choice(direction, p=pi[s, :])

    if next_direction == "up":
        action = 0
        s_next = s - 3  # 往上移動時，代表狀態的數字減3
    elif next_direction == "right":
        action = 1
        s_next = s + 1  # 往右移動時，代表狀態的數字加1
    elif next_direction == "down":
        action = 2
        s_next = s + 3  # 往下移動時，代表狀態的數字加3
    elif next_direction == "left":
        action = 3
        s_next = s - 1  # 往左移動時，代表狀態的數字減1

    return [action, s_next]
```

RANDOM SAMPLING

Goal_maze_ret_s_a(pi)

```python
# 定義走出迷宮的函數，輸出狀態與行動的履歷


def goal_maze_ret_s_a(pi):
    s = 0  # 起點
    s_a_history = [[0, np.nan]]  # 記錄智能體移動軌跡的list

    while (1):  # 抵達終點之前不斷執行的迴圈
        [action, next_s] = get_action_and_next_s(pi, s)
        s_a_history[-1][1] = action
        # 代入目前狀態（是最後一個狀態，所以是index=-1）的行動

        s_a_history.append([next_s, np.nan])
        # 代入下一個狀態。還不知道會採取什麼行動，所以先設定為nan

        if next_s == 8:  # 若抵達終點就結束執行
            break
        else:
            s = next_s

    return s_a_history
```

判斷是否goal state
紀錄中間經過的狀態與行動

[State, action]

```python
# 以初始策略走出迷宮
s_a_history = goal_maze_ret_s_a(pi_0)
print(s_a_history)
print("走出迷宮的步數為" + str(len(s_a_history) - 1) + "喲")
```

```
[[0, 1], [1, 3], [0, 1], [1, 3], [0, 1], [1, 3], [0, 2], [3, 0], [0, 2], [3, 1], [4, 3], [3, 0], [0, 1], [1, 3], [0, 2], [3, 1], [4, 3], [3, 1], [4, 3], [3, 0], [0, 2], [3, 2], [6, 0], [3, 2], [6, 0], [3, 0], [0, 2], [3, 0], [0, 1], [1, 3], [0, 1], [1, 1], [2, 2], [5, 0], [2, 2], [5, 0], [2, 3], [1, 3], [0, 2], [3, 2], [6, 0], [3, 2], [6, 0], [3, 0], [0, 2], [3, 1], [4, 3], [3, 2], [6, 0], [3, 2], [6, 0], [3, 0], [0, 2], [3, 1], [4, 3], [3, 2], [6, 0], [3, 1], [4, 2], [7, 0], [4, 2], [7, 1], [8, nan]]
走出迷宮的步數為62喲
```

```
Press any key
[[0, 2], [3, 1], [4, 2], [7, 0], [4, 2], [7, 1], [8, nan]]
走出迷宮的步數為6喲
```

策略梯度法更新策略

- θ_{s_i,a_i}代表在狀態 s_i 時採用action a_i 的機率。
- 參數更新 $\theta_{s_i,a_i} = \theta_{s_i,a_i} + \eta \times \Delta\theta_{s_i,a_i}$,
- η 為learning rate, $0 \le \eta \le 1$
  - 太小無法學習，太大學習粗糙

- N(s_i, a_j)在狀態 s_i 時採用action a_j 的次數。
- P(s_i, a_j)在狀態 s_i 時採用action a_j 的機率。
- N(s_i,a)在狀態 s_i 時採用action 的總和次數。
- T為抵達終點的總步數。
- $\Delta\theta_{s_i,a_i} = \dfrac{N(s_i,a_j) - P(s_i,a_j)N(s_i,a)}{T}$

(實際發生次數−期望次數)/T

Update_theta function

```
def update_theta(theta, pi, s_a_history):
    eta = 0.1 # 學習率
    T = len(s_a_history) - 1  # 抵達終點的總步數

    [m, n] = theta.shape  # 取得theta的矩陣大小
    delta_theta = theta.copy()  # 由於要製作Δtheta的來源與指標參照、所以不能直接寫成delta_theta = theta

    # 於每個元素計算delta_theta
    for i in range(0, m):
        for j in range(0, n):
            if not(np.isnan(theta[i, j])):  # 當theta不為nan的情況

                SA_i = [SA for SA in s_a_history if SA[0] == i]
                # 從履歷取出狀態i的list包含式

                SA_ij = [SA for SA in s_a_history if SA == [i, j]]
                # 取出於狀態i採用行動j

                N_i = len(SA_i)  # 於狀態i採取行動的總次數
                N_ij = len(SA_ij)  # 於狀態i採取行動j的次數

                # 初版的符號正負有誤（修正日期：180703）
                #delta_theta[i, j] = (N_ij + pi[i, j] * N_i) / T
                delta_theta[i, j] = (N_ij - pi[i, j] * N_i) / T

    new_theta = theta + eta * delta_theta

    return new_theta
```

SA=[[State, action], [s,a], …[s, goal]]

取出sub-list

取出sub-list

（實際發生次數-期望次數)/T

更新 $\theta_{s_i,a_i} = \theta_{s_i,a_i} + \eta \times \Delta\theta_{s_i,a_i}$

```
# 更新策略
new_theta = update_theta(theta_0, pi_0, s_a_history)
pi = softmax_convert_into_pi_from_theta(new_theta) 轉成機率矩陣
print(pi)
```

```
[[0.         0.49919355 0.50080645 0.        ]
 [0.         0.49798388 0.         0.50201612]
 [0.         0.         0.50040323 0.49959677]
 [0.3335125  0.33297501 0.3335125  0.        ]
 [0.         0.         0.49879032 0.50120968]
 [1.         0.         0.         0.        ]
 [1.         0.         0.         0.        ]
 [0.5        0.5        0.         0.        ]]
```

學習與學習結束

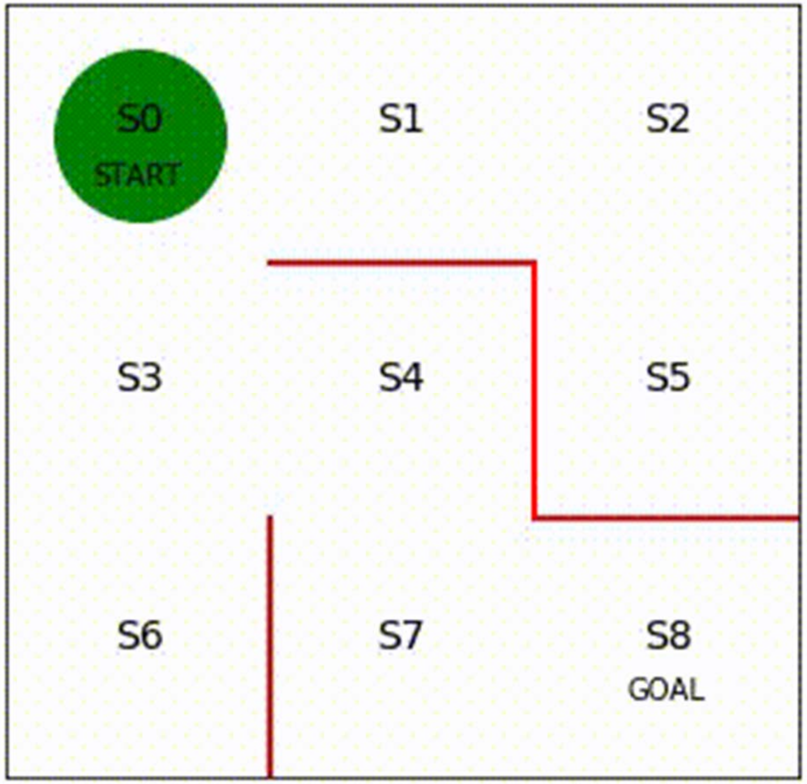
```python
# 以策略梯度法走出迷宮

# 初版的def update_theta有錯，所以調整結束執行的條件（修正日期：180703）
#若策略的改變比stop_epsilon = 10**-8  # 10^-8還少就結束學習
stop_epsilon = 10**-4  # 若策略的改變小於10^-4就結束學習


theta = theta_0
pi = pi_0

is_continue = True
count = 1
while is_continue:  # 在is_continue變成False之前持續執行
    s_a_history = goal_maze_ret_s_a(pi)  # 計算以策略π探索迷宮的履歷
    new_theta = update_theta(theta, pi, s_a_history)  # 更新參數θ
    new_pi = softmax_convert_into_pi_from_theta(new_theta)  # 更新策略π

    print(np.sum(np.abs(new_pi - pi)))  # 輸出策略的變化
    print("出走迷宮的總步數為" + str(len(s_a_history) - 1) + "喲")

    if np.sum(np.abs(new_pi - pi)) < stop_epsilon:
        is_continue = False
    else:
        theta = new_theta
        pi = new_pi
```

Console 1/A

```
出走迷宮的總步數為4喲
0.00010055087480081298
出走迷宮的總步數為4喲
0.00010038701237899753
出走迷宮的總步數為4喲
0.00010022356722585654
出走迷宮的總步數為4喲
0.00010006053787758141
出走迷宮的總步數為4喲
9.989792287384142e-05
出走迷宮的總步數為4喲
[[0.    0.014 0.986 0.   ]
 [0.    0.273 0.    0.727]
 [0.    0.    0.425 0.575]
 [0.011 0.973 0.016 0.   ]
 [0.    0.    0.987 0.013]
 [1.    0.    0.    0.   ]]
```

繪製動畫

```
# 將智能體的移動軌跡製作成動畫
# 參考URL http://louistiao.me/posts/notebooks/embedding-matplotlib-animations-in-jupyter-notebooks/
from matplotlib import animation
from IPython.display import HTML


def init():
    # 初始化背景影像
    line.set_data([], [])
    return (line,)


def animate(i):
    # 每個影格的繪製內容
    state = s_a_history[i][0]  # 繪製目前的位置
    x = (state % 3) + 0.5  # 狀態的x座標為以3除之的餘數+0.5
    y = 2.5 - int(state / 3)  # y座標為2.5減掉以3除之的商數
    line.set_data(x, y)
    return (line,)


#  以初始化函數與繪製每格影格內容的繪圖函數繪製動畫
anim = animation.FuncAnimation(fig, animate, init_func=init, frames=len(
    s_a_history), interval=200, repeat=False)

HTML(anim.to_jshtml())
```


S0
START
S1
S2
S3
S4
S5
S6
S7
S8
GOAL

The Markov Decision Process (MDP)

The Markov Decision Process (MDP)

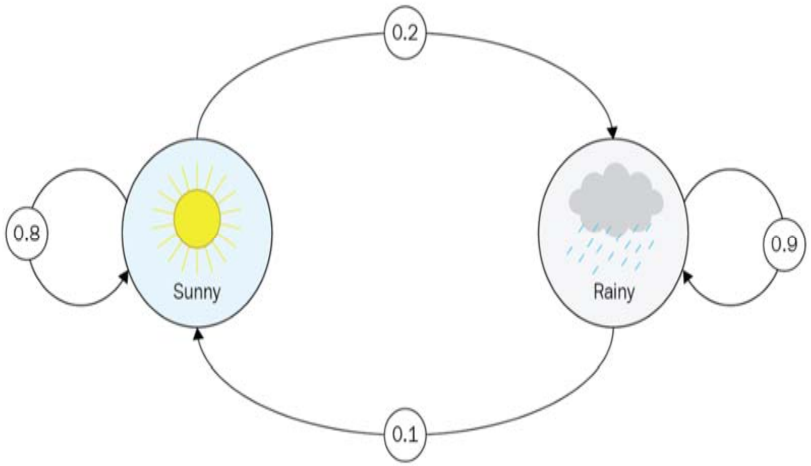
- **Markov Decision Process (MDP)** is the underlying basis of any Reinforcement Learning Process
- "**Markov Property**" which is the underlying principle of the "Markov Chain" phenomena of which MDP is a form.
  - Also called the "memoryless" property for stochastic (or probabilistic/uncertain in simpler words) processes.
  - The conditional probability distribution of the **probable next state** would depend **only** on the present state, irrespective of the sequence of states the process has gone through to reach this specific current state.
  - The conditional probability distribution of next states from this specific state remains the same.
  - Markov property implies **stationarity**: the underlying transition distribution for any state does **not** change over time.

Markov Chain (Markov Process)

- Applies the Markov Property to a sequence of stochastic events.
- It refers to a stochastic model which comprises a sequence of events such that the probability of next event is based solely on the state achieved in the previous event.
- Can be discrete or continuous process
- Example
  - States/states space (finite)
  - Observations form a sequence of states or a **chain**
  - A sequence of observations over time forms a chain of states, and this is called **history.**

Markov Process (MP)

- You can capture transition probabilities with a **transition matrix**, which is a square matrix of the size $N \times N$, where N is the number of states in our model.
- Every cell in a row, *i*, and a column, *j*, in the matrix contains **the probability of the system to transition from state *i* to state *j***.
- The formal definition of an MP is as follows:
  - A set of states (***S***) that a system can be in
  - A transition matrix (***T***), with transition probabilities, which defines the system dynamics



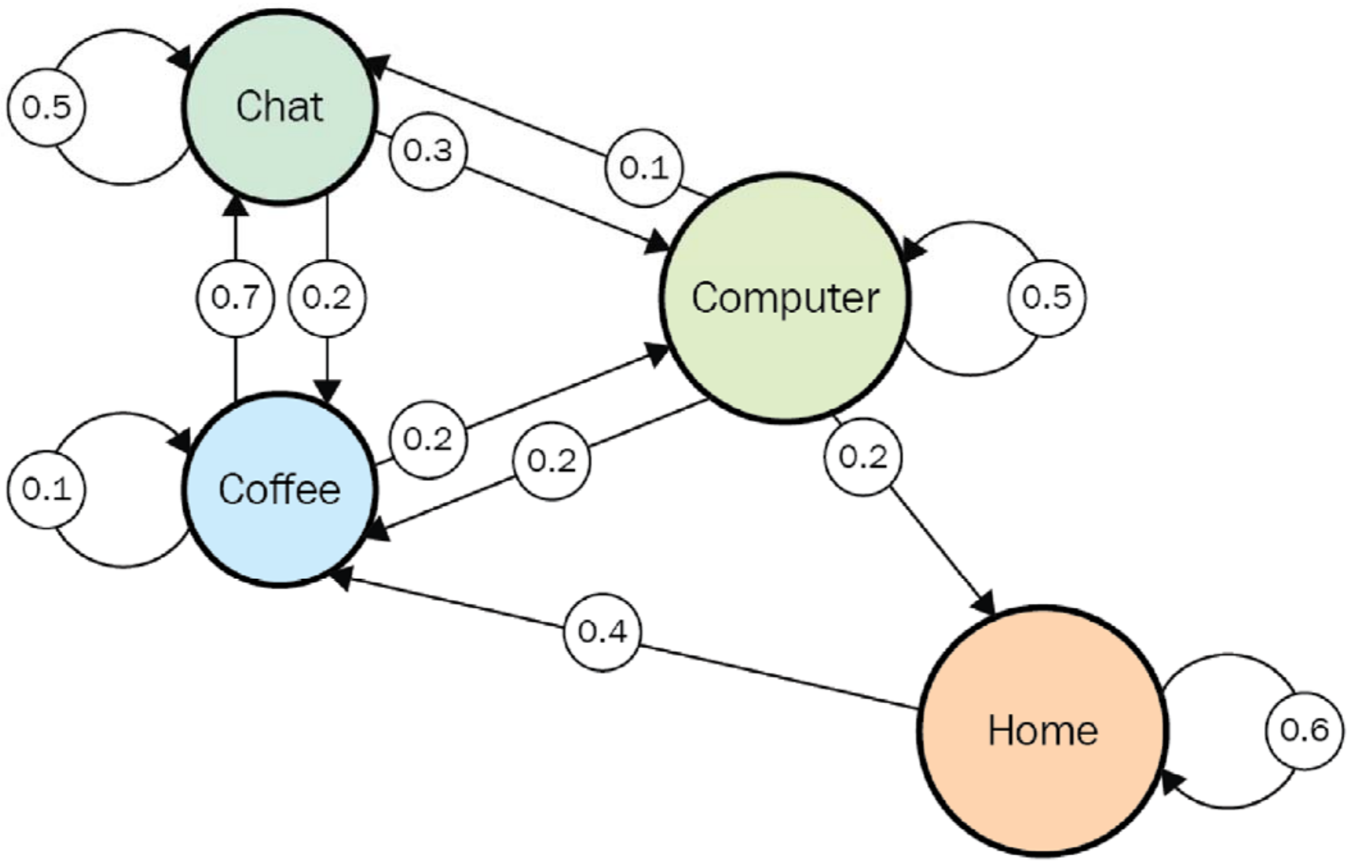
transition graph

transition matrix

	Sunny	Rainy
Sunny	0.8	0.2
Rainy	0.1	0.9

Example

- The state transition graph with transition probabilities



Markov reward processes

- Extend Markov Process model includes **reward**
- The most general way is to have another square matrix, similar to the transition matrix, with reward given for transitioning from state *i* to state *j*, which reside in row *i* and column *j*.
- Discount factor γ (gamma), which is a single number from 0 to 1 (inclusive).
- For every episode, we define **return** at the time, *t*, as this quantity: (accumulated rewards with discount factor)

$$G_t = R_{t+1} + \gamma R_{t+2} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

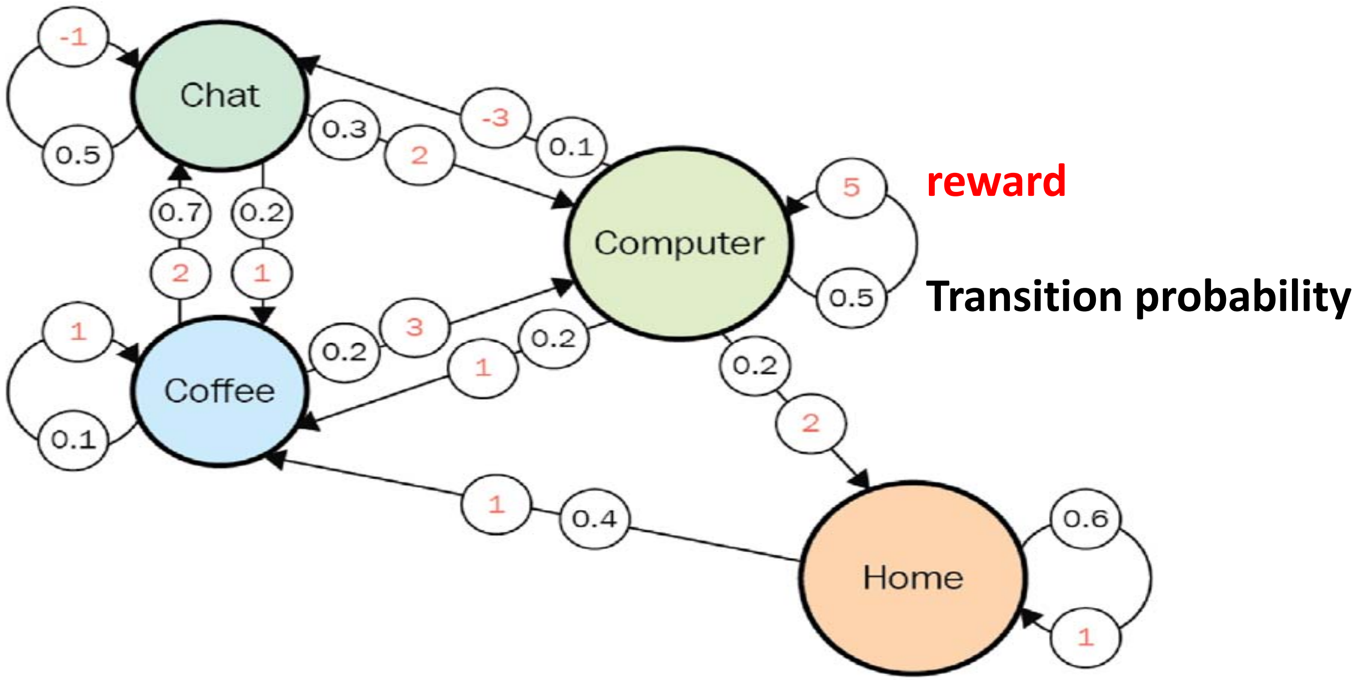
- If $\gamma = 1$, then return, G_t, just equals a sum of all subsequent rewards and corresponds to the agent that has perfect visibility of any subsequent rewards.
- If $\gamma = 0$, G_t will be just **immediate reward** without any subsequent state and will correspond to absolute short-sightedness.

Value of state

- If we go to the extreme and calculate the **mathematical expectation** of return for any state (by averaging a large number of chains), we will get a much more useful quantity, which is called the **value of the state**: $V(s) = E[G|S_t=s]$
- For every state, *s*, the value, $V(s)$, is the average (or expected) return we get by following the Markov reward process.

Value of states

- $V(chat) = -1 * 0.5 + 2 * 0.3 + 1 * 0.2 = 0.3$
- $V(coffee) = 2 * 0.7 + 1 * 0.1 + 3 * 0.2 = 2.1$
- $V(home) = 1 * 0.6 + 1 * 0.4 = 1.0$
- $V(computer) = 5 * 0.5 + (-3) * 0.1 + 1 * 0.2 + 2 * 0.2 = 2.8$



Policy

- **Policy** is some set of rules that controls the agent's behavior.
- Even for fairly simple environments, we can have a variety of policies.
- Policy is defined as the probability distribution over actions for every possible state:

  $\pi(a|s)=P[A_t = a|S_t = s]$

- Example of the robot can perform the following actions:
  - Blindly move forward regardless of anything.
  - Try to go around obstacles by checking whether that previous *forward* action failed.
  - Funnily spin around to entertain its creator.
  - Choose an action by **randomly** modeling a drunk robot in the grid world scenario.

Markov Decision Process (MDP)

- Markov Decision Process (MDP)
  - is defined as a discrete time stochastic control process.
  - applies the Markov Chain property to a given Decision Process.
  - The decision process in context of Reinforcement Learning implies to the "**Policy**" $\pi(s)$ which helps the agent determine the best action to take or transition to make when it is in a specific current state s.
  - The Markov Decision Process provides a mathematical basis for modeling the decision process where the outcomes are partly in our control and are partly random

MDP Notations in Tuple Format

- The state transition probability function, i.e., the probabilities of transitioning from the current state—***s***, to any of the next possible state—***s'***, by taking an action—***a***.
- The **state transition probability function** is conditioned on the action that is taken and is denoted as $P_a(s, s')$.
- $R_a(s, s')$ defines the **reward function** for the rewards received on attaining (transitioning to) state—s' from the current state—s, conditioned on the action—a taken.
- The probability of attaining the new state—s' from the previous state—s on taking an action—a under $P_a(s, s')$ is given by $\mathbf{P_a(s, a\ s')}$;
- The **instantaneous reward** achieved on attaining the new state—s' from the previous state —s on taking an action—a could be computed from the reward function $\mathbf{R_a(s, s')}$ as $\mathbf{R_a(s, a, s')}$.

Markov Decision Process or the MDP

- **MDP (S, A, P_a, R_a, γ),**
  - ***S*** is the present/current state,
  - ***A*** is the action taken,
  - ***P_a*** and ***R_a*** are abbreviations for $P_a(s, a, s')$ the next state probability, and
  - ***R_a***(s, a, s') the reward achieved on transitioning from the current to the new state.
  - γ: Discounted factor (a real number between 0 and 1)
  - In terms of the discounting rate r, the discounting factor γ is given by $\gamma = 1/(1 + r)$.
  - To discount a reward attained n steps ahead to the present step, the future reward is discounted by a factor of γ^n to account for it to the present time step.

Mathematical Objective

- The objective is to **maximize the sum total of all discounted rewards**.
- Maximizing the sum total of all discounted rewards may in turn require to find a policy that may do so.
- The subsequent action—a_t (at any time t) taken in any state—s is given by the policy which is denoted by $\pi_{(s)}$.
- Under this policy we have the discounted reward at time t is given as

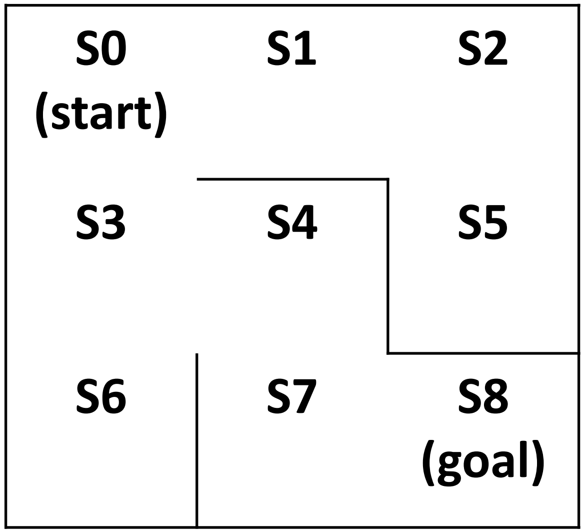
$$\gamma^t R_{a_t}(s_t, s_{t+1})$$

- Accumulating the rewards at all time steps, the total reward under this policy is given by

$$\sum_{t=0}^{\infty} \gamma^t R_{a_t}(s_t, s_{t+1})$$

Action Value $Q^{\pi}(s,a)$ example

- 對於Policy π
- **Value of action (行動價值)** can be represented by $Q^{\pi}(s,a)$
  - S=7, a=1 (右) 到達S8, thus $Q^{\pi}(s=7,a=1)=R_{t+1}=1$
  - S=7, a=0 (上) 到達S4, 要到終點需要S7-S4-S7-S8, thus $Q^{\pi}(s=7,a=0)=\gamma^2 R_{t+1}=\gamma^2$ ，
  - 多花兩步，reward要打折。



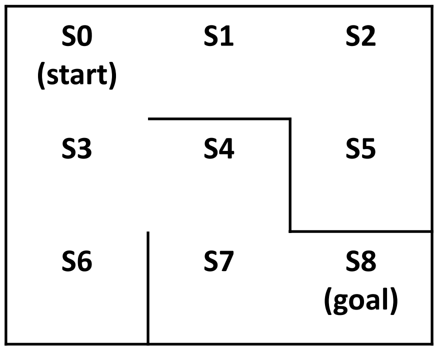
State Value $V^{\pi}(s)$

- 狀態價值 State Value $V^{\pi}(s)$
  - 於狀態s時，依照所採用的Policy π行動後，後續累積的折扣報酬總和 G_t。

$$G_t = R_{t+1} + \gamma R_{t+2} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

- S=7, a=1 (右) 到達S8, thus $V^{\pi}(s=7) = R_{t+1} = 1$
- 當agent 在S4, 要到終點需要S4-S7-S8, thus $V^{\pi}(s=4) = \gamma$
- $V^{\pi}(s=4) = R_{t+1} + \gamma\, V^{\pi}(s=7)$
  - 其中 R_{t+1}代表進入S7後的即時報酬，故$R_{t+1} = 0$

- $V^{\pi}(s=4) = 0 + \gamma\, V^{\pi}(s=7)$=$\gamma$



Bellman equation and MDP

$$V^{\pi}(s) = \max_{a} E_{\pi}\left[R_{s,a} + \gamma V^{\pi}(s(s,a))\right]$$

- State s 之狀態價值(state-value) or optimal value 定義為當採用使得右式的期望價值變得最大的action a的狀態價值。
- $R_{s,a}$代表在狀態s採取行動a所獲得的即時報酬R_{t+1} ，
- 而s(s, a)代表狀態s採取行動a所到達的新狀態s_{t+1}
- The right hand side unknown, should be estimated
- Recursive equation, the dynamic programming method should be used to calculate.

SARSA method
value-iteration method

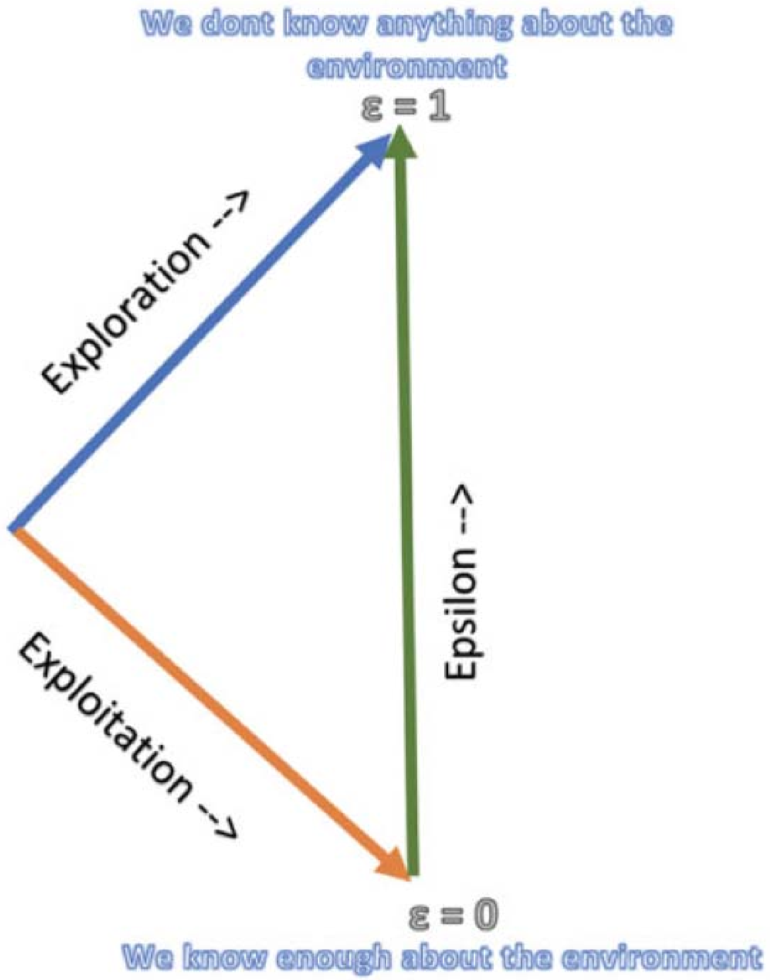
Epsilon-Greedy (ϵ-Greedy)

Epsilon-Greedy (ϵ-Greedy)

- **Epsilon-Greedy** is the most popular and the simplest algorithm to strike the trade-off between the "**exploration**" and "exploitation" phases.
- A constant "epsilon" (ϵ), which represents the probability with which the agent decides to "explore" in every turn.
- So, for example, if the value of $\epsilon = 0.1$,
  - then there is **10% probability** in any given turn that the agent will take a **random** action (explore), and
  - **90% probability** that it will "exploit" the existing **Q function** estimates that greedily chooses the action as per the best value estimates from the Q function as updated until that iteration.

Epsilon-Greedy (ϵ-Greedy)

- The larger the "epsilon", the greater the number of times the agent is likely to "explore" random actions, and
- the smaller the "epsilon" the greater the number of times the agent is likely to greedily "exploit" the estimated value/Q function.



SARSA

- **State-Action-Reward-State-Action**, or to be more precise in terms of steps, it stands for **State$_{(t)}$-Action$_{(t)}$-Reward$_{(t)}$-State$_{(t+1)}$- Action$_{(t+1)}$.**
- It uses the same principal for **value function** (/ table) updates the **action-value function** (**Q Function**) updates.
- SARSA works on "control" side of the problem.
- Given that the action-value function $Q^{\pi}(s, a)$ works on a pair of state and action, i.e., (s, a) or action when the agent is in a given state, the **SARSA** acronym could be grouped as **[(s, a), r, (s', a')],** or further augmented by the correct **action-value** notation Q as **[$\boldsymbol{Q^{\pi}(s, a)}$, r, Q(s', a')].**

SARSA update method

- SARSA updates the Q value of a given (s, a) combination,
  - using the **instantaneous rewards R** that the agent receives in **any step** and
  - the **Q value** of the resulting state-action pair, i.e., (s', a').
- The symbols α, γ, s, s',
  - α is the learning rate,
  - γ is the discounting factor,
  - s is the current state and
  - s' is the subsequent state when the agent takes an action - a in the state - s.
- Action value function (in learning)
  - $Q(s, a)=R_{t+1}+ \gamma\, Q(s', a'))$
- **Temporal difference error (TD error)** defined as
  - $\boxed{R_{t+1}+ \gamma\, Q(s', a'))}$ - $Q(s, a)$

SARSA algorithm

Initialize $Q(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state}, \cdot) = 0$
Repeat (for each episode):
  Initialize S
  Choose A from S using policy derived from Q (e.g., ϵ-greedy)
  Repeat (for each step of episode):
    Take action A, observe R, S'
    Choose A' from S' using policy derived from Q (e.g., ϵ-greedy)
    $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma Q(S',A') - Q(S,A)\big]$
    $S \leftarrow S'$; $A \leftarrow A'$;
  until S is terminal

- If **TD error =0** then learning complete
- Thus the action-value updated rule is given as
  - $Q(s, a)=Q(s, a)+ \alpha(R_{t+1} + \gamma Q(s', a') - Q(s, a))$ or
  - $Q(s, a)=(1- \alpha)Q(s, a)+ \alpha(R_{t+1} + \gamma Q(s', a'))$

“on-policy” learning algorithm

- The “on-policy”, and “off-policy” differences the algorithms could be classified into one of these depending upon **whether the algorithm uses the same mechanism (policy) for taking an action (behavior) and updating (estimating)/exploring the functions on the basis of which the best action is determined or different mechanisms for both**.
- **SARSA** follows the **same policy** to take the actions that it uses to update the action-value function. SARSA is an “**on-policy**” learning algorithm
- In **SARSA**, the **initial Q value** space is initialized with a **very low initial value (random)**, also known as “**optimistic-initial-condition**”.

Initial Q for Maze problem

```
[[0.         0.5        0.5        0.        ]
 [0.         0.5        0.         0.5       ]
 [0.         0.         0.5        0.5       ]
 [0.33333333 0.33333333 0.33333333 0.        ]
 [0.         0.         0.5        0.5       ]
 [1.         0.         0.         0.        ]
 [1.         0.         0.         0.        ]
 [0.5        0.5        0.         0.        ]]
```

```python
# 自訂策略的參數theta轉換成行動策略pi的函數


def simple_convert_into_pi_from_theta(theta):
    '''單純地計算比例'''

    [m, n] = theta.shape  # 取得theta的矩陣大小
    pi = np.zeros((m, n))
    for i in range(0, m):
        pi[i, :] = theta[i, :] / np.nansum(theta[i, :])  # 計算比例

    pi = np.nan_to_num(pi)  # 將nan轉換成0

    return pi

# 求得隨機採取行動的策略pi_0
pi_0 = simple_convert_into_pi_from_theta(theta_0)

# 設定初始的動作價值函數Q

[a, b] = theta_0.shape  # 將列與欄的數字分別存入a與b
Q = np.random.rand(a, b) * theta_0
# * theta0可乘上每個元素，在Q為朝向牆壁的值，將該值設定為nan
```

設定Q的初值 採用亂數產生

Select the next action with epsilon

```python
# 建置ε-greedy法


def get_action(s, Q, epsilon, pi_0):
    direction = ["up", "right", "down", "left"]

    # 決定動作
    if np.random.rand() < epsilon:
        # 根據ε的機率隨機移動
        next_direction = np.random.choice(direction, p=pi_0[s, :])
    else:
        # 採用Q為最大值的動作
        next_direction = direction[np.nanargmax(Q[s, :])]

    # 將動作存入index
    if next_direction == "up":
        action = 0
    elif next_direction == "right":
        action = 1
    elif next_direction == "down":
        action = 2
    elif next_direction == "left":
        action = 3

    return action
```

"explore"

"exploit"

#有nan值取max_value之函數

依action a 找出下一個state s’

```python
def get_s_next(s, a, Q, epsilon, pi_0):
    direction = ["up", "right", "down", "left"]
    next_direction = direction[a]  # 動作a的方向

    # 根據動作決定下一個狀態
    if next_direction == "up":
        s_next = s - 3  # 向上移動時，狀態的數字減3
    elif next_direction == "right":
        s_next = s + 1  # 向右移動時，狀態的數字加1
    elif next_direction == "down":
        s_next = s + 3  # 向下移動時，狀態的數字加3
    elif next_direction == "left":
        s_next = s - 1  # 向左移動時，狀態的數字減1

    return s_next

```

SARSA 更新action value 的方法

```python
# 以Sarsa更新動作價值函數Q


def Sarsa(s, a, r, s_next, a_next, Q, eta, gamma):

    if s_next == 8:  # 抵達終點的情況
        Q[s, a] = Q[s, a] + eta * (r - Q[s, a])

    else:
        Q[s, a] = Q[s, a] + eta * (r + gamma * Q[s_next, a_next] - Q[s, a])

    return Q

# 定義以Sarsa走出迷宮的函數，輸出狀態、動作的履歷與更新之後的Q

```

由 Bellman equation得 $Q(s_t, a_t) = R_{t+1} + \gamma Q(s_{t+1}, a_{t+1})$

The TD error (Temporal difference error) 為 $R_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)$

當TD error =0，代表學習完畢

故Q-value的更新公式為 $Q(s_t, a_t) = Q(s_t, a_t) + \alpha \times \left(R_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)\right)$

```python
# 定義以Sarsa走出迷宮的函數，輸出狀態、動作的履歷與更新之後的Q
def goal_maze_ret_s_a_Q(Q, epsilon, eta, gamma, pi):
    s = 0  # 起點
    a = a_next = get_action(s, Q, epsilon, pi)  # 初始的行動
    s_a_history = [[0, np.nan]]  # 記錄代理器移動軌跡的list

    while (1):  # 在抵達終點之前不斷執行的迴圈
        a = a_next  # 更新動作
        s_a_history[-1][1] = a
        # 將動作代入目前的狀態（由於是最後一個動作，所以index=-1）

        s_next = get_s_next(s, a, Q, epsilon, pi)
        # 儲存下一個狀態

        s_a_history.append([s_next, np.nan])
        # 代入下一個狀態。由於還不知道會是什麼動作，所以先設定為nan

        # 給予報酬，計算下一個動作
        if s_next == 8:
            r = 1  # 若已抵達終點就給予報酬
            a_next = np.nan
        else:
            r = 0
            a_next = get_action(s_next, Q, epsilon, pi)
            # 計算下一個動作a_next

        # 更新價值函數
        Q = Sarsa(s, a, r, s_next, a_next, Q, eta, gamma)
        # 結束條件
        if s_next == 8:  # 若已抵達終點，就結束程式
            break
        else:
            s = s_next
    return [s_a_history, Q]
```

主程式

```python
# 以Sarsa攻克迷宮

eta = 0.1  # 學習率
gamma = 0.9  # 時間折扣率
epsilon = 0.5  # ε-greedy法的初始值
v = np.nanmax(Q, axis=1)  # 計算價值在每個狀態之下的最大值
is_continue = True
episode = 1

while is_continue:  # 不斷執行，直到is_continue等於False為止
    print("回合:" + str(episode))

    # 遞減ε-greedy的值
    epsilon = epsilon / 2

    # 以Sarsa走出迷宮，得出移動軌跡與更新之後的Q
    [s_a_history, Q] = goal_maze_ret_s_a_Q(Q, epsilon, eta, gamma, pi_0)

    # 狀態價值的變化
    new_v = np.nanmax(Q, axis=1)  # 計算價值在每個狀態之下的最大值
    print(np.sum(np.abs(new_v - v)))  # 輸出狀態價值的變化
    v = new_v

    print("走出迷宮的總步數為" + str(len(s_a_history) - 1) + "步")

    # 重覆執行100回合
    episode = episode + 1
    if episode > 100:
        break
```

更新epsilon 之值

```
0.03403793554313328
走出迷宮的總步數為4步
回合:17
0.033654957599576374
走出迷宮的總步數為4步
回合:18
0.0331494362741121
走出迷宮的總步數為4步
回合:19
0.032537014859977675
走出迷宮的總步數為4步
回合:20
0.031832560054249226
走出迷宮的總步數為4步
回合:21
0.031050062470035056
```

Q-Learning

Value – Iteration Method

SARSA vs. Q-learning

- **SARSA update**

$$\boldsymbol{Q(s_t, a_t)} \\ = \boldsymbol{Q(s_t, a_t)} + \alpha \times \left(\boldsymbol{R_{t+1}} + \boldsymbol{\gamma Q(s_{t+1}, a_{t+1})} - \boldsymbol{Q(s_t, a_t)}\right)$$

- **Q-learning update**

$$\boldsymbol{Q(s_t, a_t)} \\ = \boldsymbol{Q(s_t, a_t)} + \alpha \times \left(\boldsymbol{R_{t+1}} + \boldsymbol{\gamma} \max_{\boldsymbol{a}} \boldsymbol{Q(s_{t+1}, a)} - \boldsymbol{Q(s_t, a_t)}\right)$$

Q-Learning

- ***Q-Learning*** also use ***Temporal Difference Learning*** (TD Learning) for the estimation side of the problem.
- Q-Learning provides solution for the “control” part of the problem and tries to **estimate the action-value/Q Function to take the best possible action** (this is called the “control”).
- So, the estimation part for Q-Learning is similar to that of SARSA and it also **updates the Q Function iteratively in every step**.
- Q-Learning is an “**Off-Policy**” approach and **does not** use the Q Function to decide the behavior (or the policy to determine the next action).
- Therefore, unlike SARSA, the initialization of the Q-Table/Variable **could be done using all zeros**.
- Convergent more quickly

Q-learning for Maze problem

```python
# 求得隨機採取行動的策略pi_0
pi_0 = simple_convert_into_pi_from_theta(theta_0)

## 設定初始的動作價值函數Q

[a, b] = theta_0.shape  # 將列與欄的數字分別存入a與b
Q = np.random.rand(a, b) * theta_0 * 0.1
# * theta0可乘上每個元素，在Q為朝向牆壁的值，將該值設定為nan

# 建置ε-greedy法


def get_action(s, Q, epsilon, pi_0):
    direction = ["up", "right", "down", "left"]

    # 決定動作
    if np.random.rand() < epsilon:
        # 根據ε的機率隨機移動
        next_direction = np.random.choice(direction, p=pi_0[s, :])
    else:
        # 採用Q為最大值的動作
        next_direction = direction[np.nanargmax(Q[s, :])]

    # 將動作存入index
    if next_direction == "up":
        action = 0
    elif next_direction == "right":
        action = 1
    elif next_direction == "down":
        action = 2
    elif next_direction == "left":
        action = 3

    return action
```

比SARSA更小

E-greedy method

Q-learning update value

```python
def get_s_next(s, a, Q, epsilon, pi_0):
    direction = ["up", "right", "down", "left"]
    next_direction = direction[a]  # 動作a的方向

    # 根據動作決定下一個狀態
    if next_direction == "up":
        s_next = s - 3  # 向上移動時，狀態的數字減3
    elif next_direction == "right":
        s_next = s + 1  # 向右移動時，狀態的數字加1
    elif next_direction == "down":
        s_next = s + 3  # 向下移動時，狀態的數字加3
    elif next_direction == "left":
        s_next = s - 1  # 向左移動時，狀態的數字減1

    return s_next

# 以Q學習更新動作價值函數Q的部分
```

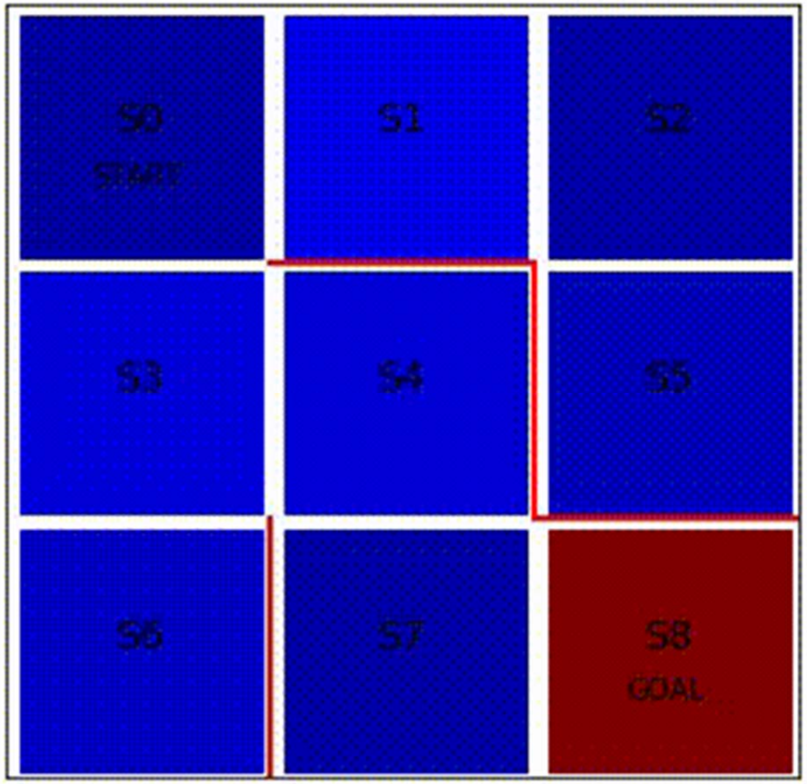
$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha \times \left(R_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t) \right)$$

```python
def Q_learning(s, a, r, s_next, Q, eta, gamma):

    if s_next == 8:  # 抵達終點的情況
        Q[s, a] = Q[s, a] + eta * (r - Q[s, a])

    else:
        Q[s, a] = Q[s, a] + eta * (r + gamma * np.nanmax(Q[s_next,: ]) - Q[s, a])

    return Q
```

```python
def goal_maze_ret_s_a_Q(Q, epsilon, eta, gamma, pi):
    s = 0  # 起點
    a = a_next = get_action(s, Q, epsilon, pi)  # 初始的行動
    s_a_history = [[0, np.nan]]  # 記錄代理器移動軌跡的list

    while (1):  # 在抵達終點之前不斷執行的迴圈
        a = a_next  # 更新動作

        s_a_history[-1][1] = a
        # 將動作代入目前的狀態（由於是最後一個動作，所以index=-1）

        s_next = get_s_next(s, a, Q, epsilon, pi)
        # 儲存下一個狀態

        s_a_history.append([s_next, np.nan])
        # 代入下一個狀態。由於還不知道會是什麼動作，所以先設定為nan

        # 給予報酬，計算下一個動作
        if s_next == 8:
            r = 1  # 若已抵達終點就給予報酬
            a_next = np.nan
        else:
            r = 0
            a_next = get_action(s_next, Q, epsilon, pi)
            # 計算下一個動作a_next

        # 更新價值函數
        Q = Q_learning(s, a, r, s_next, Q, eta, gamma)

        # 結束條件
        if s_next == 8:  # 若已抵達終點，就結束程式
            break
        else:
            s = s_next

    return [s_a_history, Q]
```

```
# 以Q學習走出迷宮

eta = 0.1  # 學習率
gamma = 0.9  # 時間折扣率
epsilon = 0.5  # ε-greedy法的初始值
v = np.nanmax(Q, axis=1)  # 計算價值在每個狀態之下的最大值
is_continue = True
episode = 1

V = []  # 儲存每回合的狀態價值
V.append(np.nanmax(Q, axis=1))  # 計算動作價值在每個狀態下的最大值

while is_continue:  # 不斷執行，直到is_continue等於False為止
    print("回合:" + str(episode))

    # 遞減ε-greedy的值
    epsilon = epsilon / 2

    # 以Q學習走出迷宮，得出移動軌跡與更新之後的Q
    [s_a_history, Q] = goal_maze_ret_s_a_Q(Q, epsilon, eta, gamma, pi_0)

    # 狀態價值的變化
    new_v = np.nanmax(Q, axis=1)  # 計算動作價值在每個狀態下的最大值
    print(np.sum(np.abs(new_v - v)))  # 輸出狀態價值的變化
    v = new_v
    V.append(v)  # 追加在這個回合結束時的狀態價值函數

    print("走出迷宮的總步數為" + str(len(s_a_history) - 1) + "步")

    # 重覆執行100回合
    episode = episode + 1
    if episode > 100:
        break
```

```python
def animate(i):
    # 每一格影格的繪圖內容
    # 在每一格繪製與狀態價值相同大小的彩色四邊形
    line, = ax.plot([0.5], [2.5], marker="s",
                    color=cm.jet(V[i][0]), markersize=85)  # S0
    line, = ax.plot([1.5], [2.5], marker="s",
                    color=cm.jet(V[i][1]), markersize=85)  # S1
    line, = ax.plot([2.5], [2.5], marker="s",
                    color=cm.jet(V[i][2]), markersize=85)  # S2
    line, = ax.plot([0.5], [1.5], marker="s",
                    color=cm.jet(V[i][3]), markersize=85)  # S3
    line, = ax.plot([1.5], [1.5], marker="s",
                    color=cm.jet(V[i][4]), markersize=85)  # S4
    line, = ax.plot([2.5], [1.5], marker="s",
                    color=cm.jet(V[i][5]), markersize=85)  # S5
    line, = ax.plot([0.5], [0.5], marker="s",
                    color=cm.jet(V[i][6]), markersize=85)  # S6
    line, = ax.plot([1.5], [0.5], marker="s",
                    color=cm.jet(V[i][7]), markersize=85)  # S7
    line, = ax.plot([2.5], [0.5], marker="s",
                    color=cm.jet(1.0), markersize=85)  # S8
    return (line,)


#  利用初始化函數與每格影格的繪圖函數繪製動畫
anim = animation.FuncAnimation(
    fig, animate, init_func=init, frames=len(V), interval=200, repeat=False)

HTML(anim.to_jshtml())
```


S0
START
S1
S2
S3
S4
S5
S6
S7
S8
GOAL

Deep Q Network (DQN)

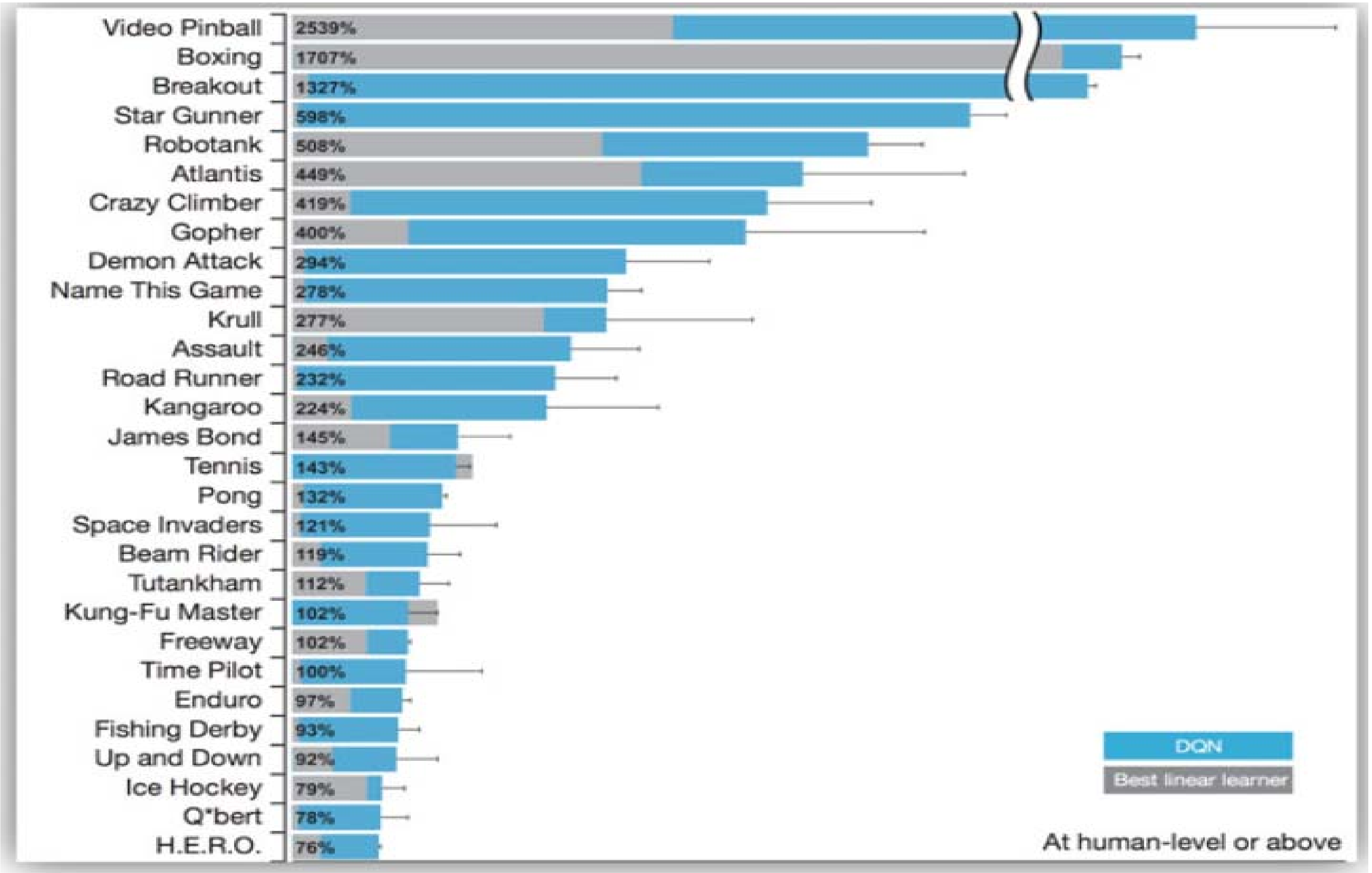
General Artificial Intelligence

- Until recently the Reinforcement Learning agents were handcrafted and tuned to perform individual and specific tasks.
- Recently with AI gym and some other initiatives opening their platforms to Reinforcement Learning academician and enthusiast to work on standardized problems (in the form of exposed standard environments) and compare their results and enhancements for these problems with the community

“Google Deep Mind” and “AlphaGo”

- Researchers at Google’s “**Deep Mind**” (“Deep Mind” was acquired by Google sometime back) developed this algorithm called as the **Deep Q Network**.
- Combined the **Q-Learning algorithm** in Reinforcement Learning with the ideas in **Deep Learning** to enable the concept of **Deep Q Networks** (DQN).
- A single DQN program could teach itself how to play 49 different games from the “**Atari**” titles (“Atari” used to be a very popular gaming console in the era of ’80s and beyond.
- RL+DL =Deep Q-Network (DQN)
  - We seek a single agent which can solve any human level task
  - RL defines the objective (Q-value function)
  - DL gives the mechanism
  - Use deep network to represent value function

DQN application on Atari game



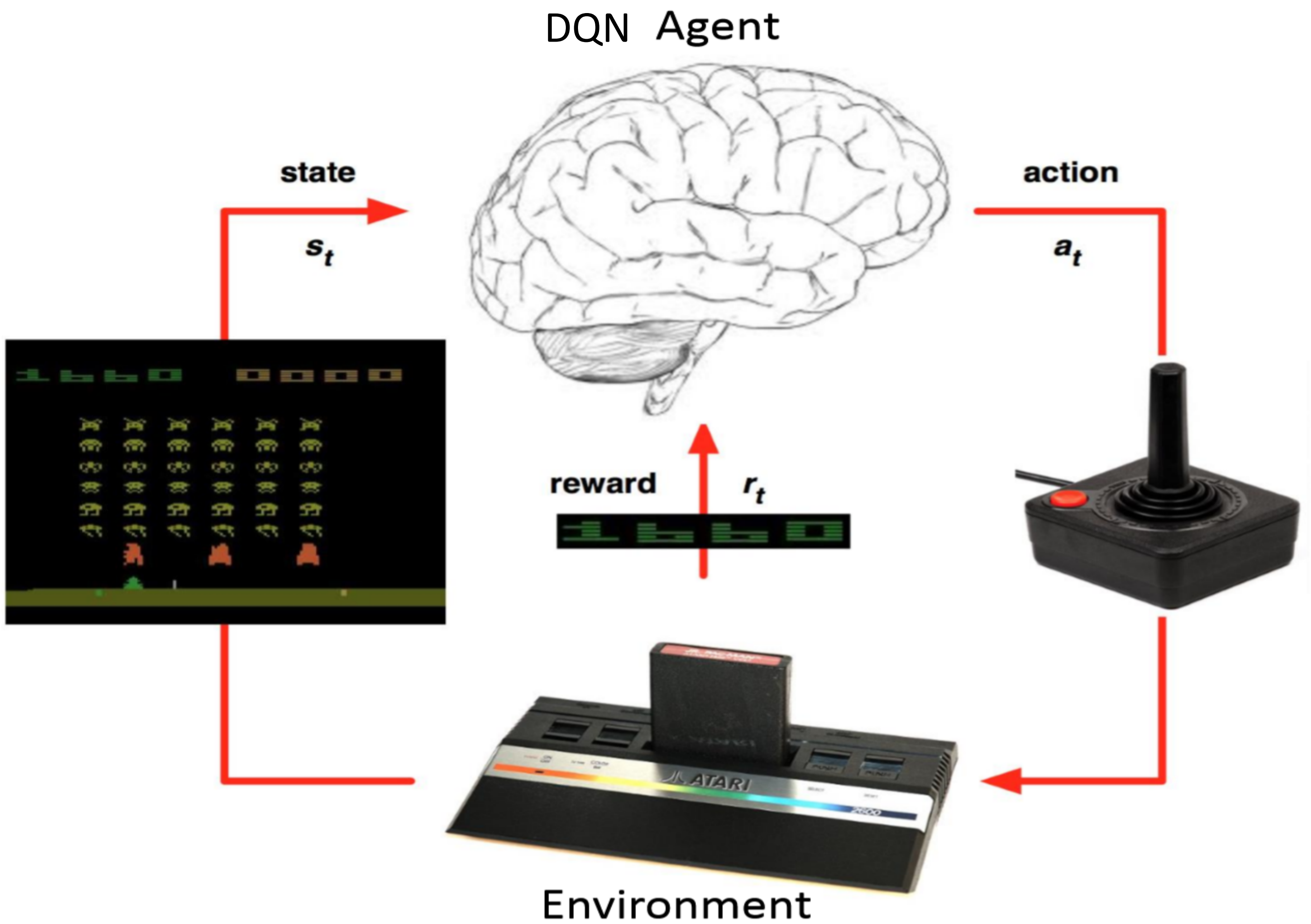
The DQN Algorithm

- The term “Deep” in the “Deep Q Networks” (DQN) refers to the use of “**Deep**” “**Convolutional Neural Networks**” (**CNN**) in the DQNs.
- **Convolutional Neural Networks** (CNN) are deep learning architectures inspired by the way human’s **visual cortex area** of the brain works to understand the images that the sensors (eyes) are receiving.
- The state could either be humanly abstracted or the agent could be made intelligent enough to make sense of these states.

DQN

- The specific DQN performed well on 49 Atari titles simultaneously,
  - used an architecture having a **CNN with 2 convolutional layers**, followed by two fully connected layers,
  - terminating into an **18 class "SoftMax" classification**.
  - These 18 classes represent the 18 actions possible from an **Atari controller** (Atari had a single 8-direction joystick, and just one button for all the games) that the game input could act on.
- These 18 classes (as used in the specific DQN by DeepMind for Atari) are
  - **Do-Nothing** (i.e., don't do anything), then
  - **8-classes representing the 8 directions of the joystick** (Move-Straight-Up, Move-Diagonal-Right-UP, Move-Straight-Right, Move-Diagonal-Right-Down, Move-Straight-Down, Move-Diagonal-Left-Down, Move-Straight-Left, Move-Diagonal-Left-Up), Press-Button (alone without moving), then
  - another **8 actions corresponding to simultaneously pressing the button and making one of the joystick-movement**.

DQN in Atari



DQN

- Atari gives a 60 FPS video output. It means that every second the game generates and displays/sends 60 images as an input.
- **One drawback** of using raw image pixels and working directly with all consecutive frames at such high frame rate to train a Q-Learning-Network is that the training of the Q-Learning-Network may not be very stable.
- Not only the **training** might take a lot of time to converge, but at times instead of **converging** the loss function may actually diverge or get stuck into a hunting loop.
- To overcome these challenges while working on **high frame rate**, high-dimension, **correlated image** data the DQN had to implement the following **three enhancements** to ensure descent convergence and practical applicability.

Keras RL model

- We will use the *keras-rl* library here which lets us implement deep Q-learning out of the box.
- **Step 1: Install keras-rl library**
  - **git clone https://github.com/matthiasplappert/keras-rl.git**
  - **cd keras-rl**
  - **python setup.py install**
- **Step 2: Install dependencies for the CartPole environment**
  - **pip install h5py**
  - **pip install gym**

Example code keras-rl DQN for carpole

Install keras-rl library
Pip install keras-rl
pip install h5py
pip install gym

- import numpy as np
- import gym

- from keras.models import Sequential
- from keras.layers import Dense, Activation, Flatten
- from keras.optimizers import Adam

- from rl.agents.dqn import DQNAgent
- from rl.policy import EpsGreedyQPolicy
- from rl.memory import SequentialMemory

- ENV_NAME = 'CartPole-v0'

https://www.analyticsvidhya.com/blog/2019/04/introduction-deep-q-learning-python/

- # Get the environment and extract the number of actions available in the Cartpole problem
- env = gym.make(ENV_NAME)
- np.random.seed(123)
- env.seed(123)
- nb_actions = env.action_space.n

- model = Sequential()
- model.add(Flatten(input_shape=(1,) + env.observation_space.shape))
- model.add(Dense(16))
- model.add(Activation('relu'))
- model.add(Dense(nb_actions))
- model.add(Activation('linear'))
- print(model.summary())

- policy = EpsGreedyQPolicy()
- memory = SequentialMemory(limit=50000, window_length=1)
- dqn = DQNAgent(model=model, nb_actions=nb_actions, memory=memory, nb_steps_warmup=10,
- target_model_update=1e-2, policy=policy)
- dqn.compile(Adam(lr=1e-3), metrics=['mae'])

- # Okay, now it's time to learn something! We visualize the training here for show, but this slows down training quite a lot.
- dqn.fit(env, nb_steps=5000, visualize=True, verbose=2)

- dqn.test(env, nb_episodes=5, visualize=True)