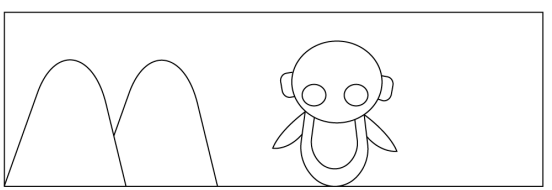
什麼是 RL ？

假設你想教狗狗接球，但無法透過直接示範接球來教會牠。你只能把球丟出去，狗狗只要接到球，就給牠一塊餅乾；如果沒接到球的話，就沒有餅乾。這樣狗狗就會知道哪個動作會收到餅乾，只要重複做這個動作就好。

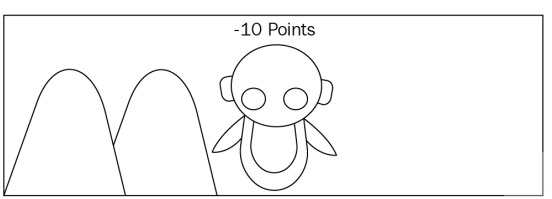
同樣地，在 RL 環境中，你不是去教導代理要做什麼以及怎麼做，反之是對代理所做的每個動作都給一個獎勵。獎勵可能為正向或負向。代理就會去執行那些會讓它得到正向獎勵的動作。因此，這是一個試誤的過程。在上個比喻中，狗狗就是代理。當狗狗接到球就給一片餅乾，這是正向獎勵，不給餅乾則是負向獎勵。

獎勵可能延遲發放。你可能不會在每一步驟都得到獎勵，而可能是在完成一項任務之後才給。在某些情況下，你在每一步驟都會得到獎勵，這樣就能知道有沒有犯錯。

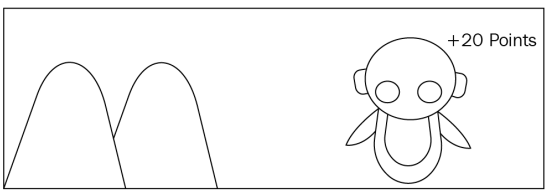
想像一下，你要教導機器人如何走路且不會撞到小山，但做法不是直接告訴機器人不要朝著山前進：



反之，如果機器人撞到山並且卡住的話，你會扣 10 分來讓機器人理解到撞到山會導致一個負面獎勵，它就不會再朝這個方向走了：



當機器人沿著正確方向前進且不會被卡住的話，它會得到 20 分。這樣一來，機器人就知道哪條才是正確的道路，它會嘗試沿著正確的道路移動，期望得到最多的獎勵：



RL 代理會去**探索（explore）**各種可能會讓它得到正向獎勵，或者它會再次**運用（exploit）**前一個可導致正向獎勵的動作。如果 RL 代理又一直去探索不同動作的話，代理很有可能得到相當差的獎勵，因為後續的動作很難是最好的。但如果 RL 代理只去做它已知的最佳動作，也相當有可能漏掉實際上能提供更棒獎勵的最佳動作。在探索新動作與運用既有動作，兩者之間一定有取捨。我們無法同時讓探索與運用都最佳化。後續章節會深入討論這個探索 - 運用難題。

RL 演算法

常見的 RL 演算法包含了以下步驟：

1. 首先，代理會執行某個動作來與環境互動。
2. 代理執行某個動作，並從一個狀態轉移到下一個狀態。
3. 根據所執行的動作，代理會收到一個獎勵。
4. 代理會根據獎勵來理解到這個動作是好是壞。

5. 如果動作結果很好，也就是代理收到了一個正向獎勵，代理就會傾向於執行這個動作，不然代理就會試著去做其他會產生正向獎勵的動作。因此，這基本上就是個試誤型的學習流程。

RL 與其他 ML 方法有何不同？

在監督式學習中，機器（代理）是經由一組已標記的訓練輸入資料來學習並輸出結果。目標是讓模型能學會如何推斷與歸納，好讓它能適用於未見過的資料。在此會有一位具備完整環境知識的外部監督者，它會監督代理來完成任務。

回想一下之前的狗狗接球範例；在監督式學習中如果要教狗狗接球的話，我們會明確地告訴狗狗：左轉、右轉、前進五步、接球等等。但換成 RL 的話，我們就是把球丟出去而已，每次狗狗接到球，我們給它一片餅乾（獎勵）。這樣一來狗狗就會藉由得到餅乾來學會如何接球。

在非監督式學習中，我們只提供模型以及只有幾組輸入的訓練資料；模型會學習如何去判斷輸入中隱藏的型態。常見的誤解是把 RL 視為非監督式學習，但事實上不是。在非監督式學習中，模型會學習隱藏的架構，但 RL 的作法是讓模型藉由獎勵最大化來學習。假設我們想對使用者推薦新電影，非監督式學習會去分析使用者看過的類似電影來推薦，RL 則是不斷接收來自使用者的回饋、理解他們對於電影的喜好，並以此建立一個知識庫來推薦新的電影。

另外還有稱為半監督式學習（semi-supervised learning）的學習方式，基本上就是監督式與非監督式學習的組合，它需要對已標記與未標記的資料進行函數估計，在此 RL 就是代理與其所處環境的互動方式。因此，RL 與其他的機器學習法是完全不同的。

RL 所包含的重要元素

RL 所包含的元素一一介紹如下：

◉ 代理

代理是指一個能執行智能決策的軟體程式，就是 RL 中的學習者。代理藉由與環境的互動來作出某些動作，並根據所採取的動作來收到獎勵，例如，超級瑪利在遊戲中移動。

◉ 策略函數

策略（policy）定義了代理在環境中的行為。代理會根據策略來決定到底要做哪個動作。假設你想要從家裡出發到辦公室，去辦公室的路有很多條，有些路是捷徑，有些則比較長。這些路徑就稱為策略，因為它們代表了我們選擇某個動作來達到目標的方式。策略通常是用符號 π 來表示，而策略可能是個查找表或更複雜的搜尋流程。

◉ 價值函數

價值函數（Value Function）代表代理在某個狀態中到底有多好。它與某個策略是相依的，且常用 $v(s)$ 來表示。它等於代理從初始狀態開始之後所收到的總期望獎勵。價值函數可以有多個；最佳價值函數代表這個函數與其他函數相比，在所有的狀態中都得到了最高的價值。同樣地，最佳策略就是擁有最佳價值函數的那一個策略。

◉ 模型

模型是代理在某個環境的表現。學習有兩種類型：模型式（model-based）學習與無模型（model-free）學習。在模型式學習中，代理會運用先前所學到的資訊來完成一項任務，另一方面在無模型學習中，代理只仰賴執行正

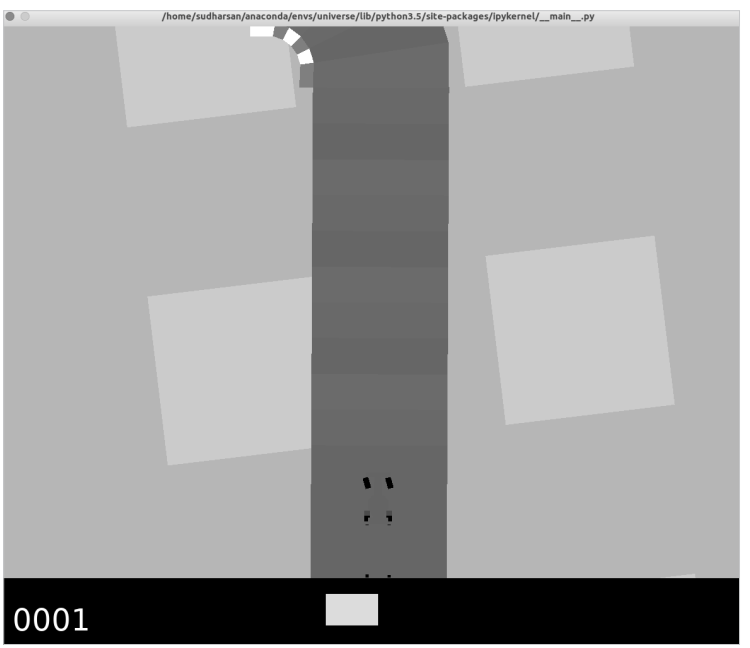
OpenAI Gym 提供了相當多的模擬環境來訓練、評估與建置各種代理。我們可以到環境開發者的網站上看看資料，或使用以下程式來列出所有可用的環境：

```
from gym import envs
print(envs.registry.all())
```

既然 Gym 已經提供了這麼多不同的有趣環境，那麼來模擬一個賽車環境，程式碼如下：

```
import gym
env = gym.make('CarRacing-v0')
env.reset()
for _ in range(1000):
    env.render()
    env.step(env.action_space.sample())
```

執行後會看到以下的輸出畫面：



◉ 訓練機器人走路

本段要運用 Gym 搭配其他基礎功能來訓練機器人走路。

策略是當機器人前進時，給它 X 分，但當機器人失敗的話，就扣 Y 分。如此一來，機器人就能以獎勵最大化的前提來學會走路。

首先要匯入函式庫，接著用 make 指令來建立一個模擬實例。Open AI Gym 提供一個稱為 BipedalWalker-v2 的環境，可以在簡易的地形上訓練機器人代理：

```
import gym
env = gym.make('BipedalWalker-v2')
```

接著針對每一世代（episode，代表從初始到最終狀態這段期間，代理與環境之間的互動），使用 reset 方法來初始化環境：

```
for episode in range(100):
  observation = env.reset()
```

接著，透過迴圈來產生環境：

```
 for i in range(10000):
  env.render()
```

我們從環境的動作空間來取樣隨機動作。每個環境都有各自的動作空間，其中包含了可用的有效動作：

```
action = env.action_space.sample()
```

對每個動作步驟來說，會記錄 observation、reward、done 與 info 等四個值：

```
observation, reward, done, info = env.step(action)
```

observation 是一個呈現環境觀測結果的物件。例如機器人在地形中的狀態。

reward 代表上一個動作所得到的獎勵。例如機器人成功前進所收到的獎勵。

done 則是一個布林值；如果為真，表示 episode 已完成（代表機器人已學會走路或完全失敗）。一旦 episode 完成，就能使用 env.reset() 來初始化下一個世代的環境。

info 是除錯相關的資訊。

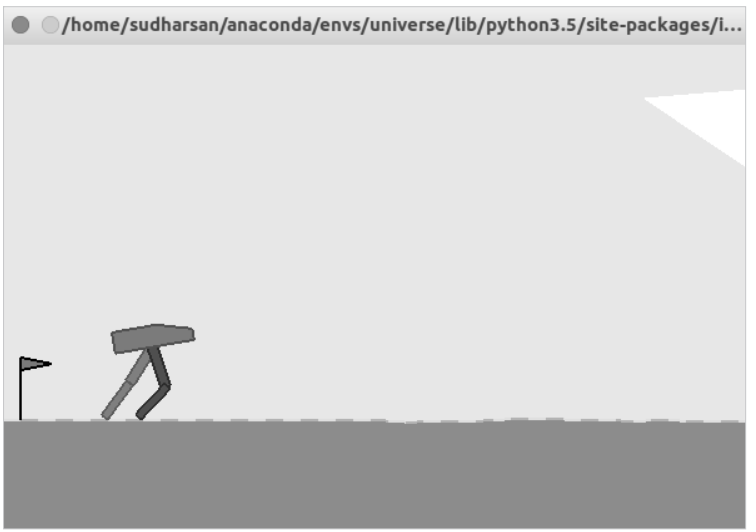
當 done 為真時，我們把這個世代所花的時間步驟顯示出來並中斷當下的世代：

```
if done:
  print("{} timesteps taken for the Episode".format(i+1))
  break
```

完整程式碼如下：

```
import gym
env = gym.make('BipedalWalker-v2')
for i_episode in range(100):
  observation = env.reset()
  for t in range(10000):
      env.render()
      print(observation)
      action = env.action_space.sample()
      observation, reward, done, info = env.step(action)
      if done:
          print("{} timesteps taken for the episode".format(t+1))
          break
```

輸出畫面如下：



OpenAI Universe

OpenAI Universe 提供非常豐富的擬真遊戲環境。OpenAI Universe 是 OpenAI Gym 的延伸版，提供了各種環境來訓練與評估代理的效能，從簡單到超複雜的即時環境都有，它也能完全控制許多遊戲環境。

◉ 打造電玩機器人

現在，要來看看怎麼做一個會玩賽車遊戲的電玩機器人。我們的目標是讓小車一直前進，並且不會撞到任何障礙物或其他小車。

首先匯入所需的函式庫：

```
import gym
import universe  # 註冊 universe 環境
import random
```

使用 make 函式來模擬賽車環境：

```
env = gym.make('flashgames.NeonRace-v0')
env.configure(remotes=1)  # 自動建立本地端的 docker 容器
```

建立用來控制小車的各個變數：

```
# 向左
left = [('KeyEvent', 'ArrowUp', True), ('KeyEvent', 'ArrowLeft', True),
        ('KeyEvent', 'ArrowRight', False)]

# 向右
right = [('KeyEvent', 'ArrowUp', True), ('KeyEvent', 'ArrowLeft', False),
         ('KeyEvent', 'ArrowRight', True)]

# 前進
forward = [('KeyEvent', 'ArrowUp', True), ('KeyEvent', 'ArrowRight', False),
           ('KeyEvent', 'ArrowLeft', False), ('KeyEvent', 'n', True)]
```

初始化其他變數：

```
# 使用變數來決定是否要轉彎
turn = 0

# 把所有獎勵存入清單
rewards = []

# 將緩衝區大小作為閾值
buffer_size = 100

# 一開始設定動作為 forward，讓小車前進而不會轉彎
action = forward
```

現在讓遊戲代理運用無窮迴圈來一直玩遊戲，根據與環境的互動來選定某個動作：

```
while True:
    turn -= 1
# 假設一開始只會直走而不轉彎
# 接著會檢查 turn 變數值，如果小於 0
# 代表不需要轉彎而繼續直走
    if turn <= 0:
        action = forward
        turn = 0
```

解 Bellman 方程式

只要解出 Bellman 最佳性方程式就能找到最佳策略。但我們需要一個稱為動態規劃的特殊技巧，才能順利解出 Bellman 最佳性方程式。

◉ 動態規劃

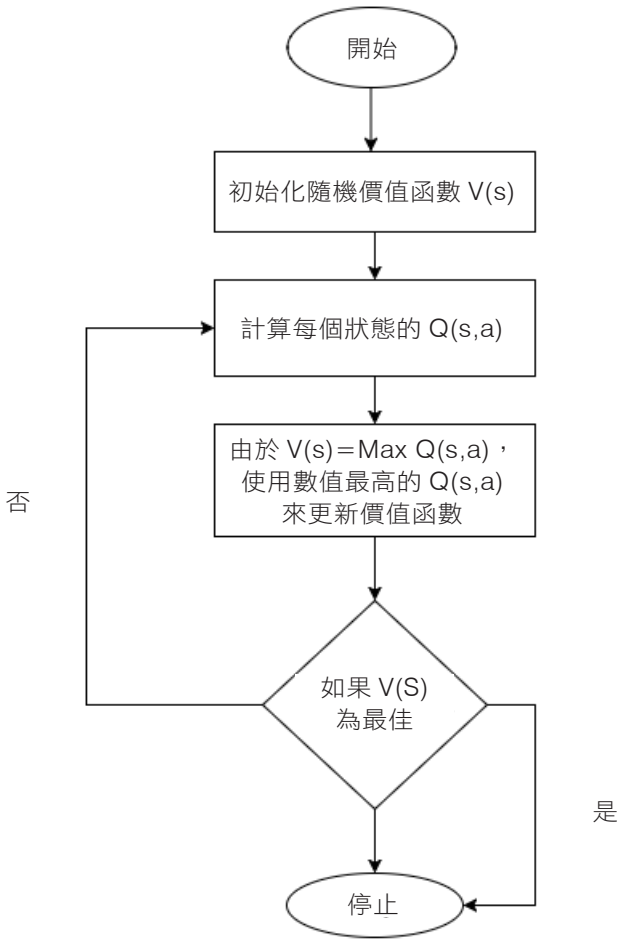
動態規劃（Dynamic programming，DP）是一種用於處理複雜問題的技巧。在 DP 中並非逐一搞定一整個複雜問題，而是把問題拆解成較簡單的子問題，並計算並儲存每個子問題的解決方案。如果發生了同樣的子問題，我們不再重新計算而會採用已經計算好的方案。因此，DP 就能大幅降低運算時間。它已被普遍應用於各種領域，包含電腦科學、數學與生物資訊學等等。

我們使用兩個很厲害的演算法來解 Bellman 方程式：

- 價值迭代
- 策略迭代

價值迭代

價值迭代層從隨機的價值函數開始，這個隨機價值函數顯然不一定會是最好的，所以要用遞迴的方式來尋找更新更好的價值函數，直到找到最佳的為止。一旦找到最佳的價值函數，很容易就能從它身上找出最佳策略：

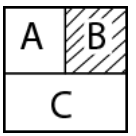


價值迭代的步驟如下：

1. 首先，初始化隨機價值函數，就是各個狀態的隨機值。
2. 計算所有狀態動作組 *Q(s, a)* 的 *Q* 函數。
3. 使用 *Q(s,a)* 的最大值來更新價值函數。
4. 重複以上步驟直到價值函數的變化非常小為止。

現在一步步來執行價值迭代，好讓你能更直觀地理解。

請看以下的格子。假設我們處於狀態 **A**，目標是在不造訪狀態 **B** 的前題下到達狀態 **C**，共有兩種動作：0—左 / 右，與 1—上 / 下：



想得出來這裡的最佳策略是什麼嗎？現在的最佳策略是在狀態 **A** 中執行動作 1，這樣就不會造訪狀態 **B** 並順利到達狀態 **C**。但如何找到這個最佳策略呢？現在就來看看吧。

初始化隨機價值函數，也就是給予所有狀態一個隨機數值。現在把所有狀態都設為 **0**：

狀態	價值
A	0
B	0
C	0

計算所有狀態 - 動作組的 Q 值。

Q 值會指出一個動作在各狀態中的數值。首先算出狀態 **A** 的 Q 值。回想一下 Q 函數的方程式。要算出這個結果需要轉移機率與獎勵機率。狀態 **A** 的轉移機率與獎勵機率如下：

狀態 (s)	動作 (a)	下一個動作 (s')	轉移機率 $(P^a_{ss'})$	獎勵機率 $(R^a_{ss'})$
A	0	A	0.1	0
A	0	B	0.4	-1.0
A	0	C	0.3	1.0
A	1	A	0.3	0
A	1	B	0.1	-2.0
A	1	C	0.5	1.0