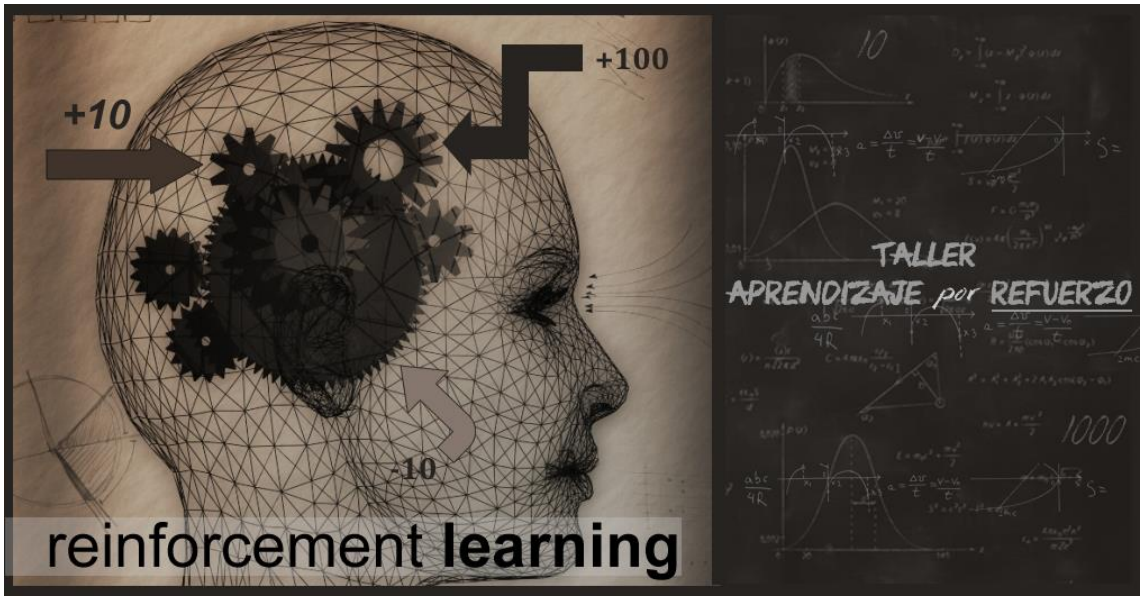


[Parte 1] – Introducción al RL.

Con esta primera parte se busca introducir al lector en el Reinforcemente Learning (Aprendizaje por refuerzo) de una manera muy práctica, con ejemplos reales y sin que sea necesario tener unos conocimientos previos sobre IA o Machine Learning. Al finalizar el artículo se pretende que el lector pueda comprender cómo funciona y cómo se comporta un agente inteligente y cuáles son los problemas iniciales en el diseño de estos algoritmos.

Índice

1.- Introducción.

El objetivo.

Este primer capítulo no busca introducir al lector definitivamente en el **Machine Learning** (aprendizaje automático) a través del **Reinforcement Learning** (Aprendizaje por refuerzo), eso vendrá después. Este primer capítulo pretende fomentar la curiosidad necesaria para adentrarse en el apasionante mundo del **Reinforcement Learning**, por lo que dejaremos a un lado gran parte de la teoría, la matemática y la estadística para centrarnos únicamente en ejemplos prácticos a través de pruebas de concepto y así poder experimentar con ellos, sacar conclusiones y aprender de los problemas que irán apareciendo.

La curva de aprendizaje.

¿Cuál es el motivo de empezar así? En mi experiencia personal, cuando decidí adentrarme en estos menesteres me topé con una gran cantidad de información que explicaba el **Reinforcement Learning** (RL) desde un enfoque muy teórico a través de la lógica, las matemáticas y la estadística con un lenguaje muy técnico y donde se asumían muchos conceptos importantes. Los ejemplos prácticos y las pruebas de concepto brillaban por su ausencia. A mayores, muchos de estos textos están enfocados hacia algo muy concreto dentro del RL.

Después de recolectar diferentes fuentes de conocimiento lo más importante parecía ser clasificar toda la información y buscar un punto de partida por el que comenzar. Me resulto bastante complicado encontrar ese punto de partida.

Soy un gran defensor de que el mejor sistema de aprendizaje es el que combina la teoría con ejemplos prácticos. Considero que es una fuente de realimentación constante en ambos sentidos.

Cuando eres autodidacta lo primero que debes de hacer es *aprender a aprender*. Siguiendo este consejo, intenté llevar a la práctica, al mundo real, los pocos conocimientos que había absorbido mediante *papers*, libros y videos… ¡y la cosa mejoró!

Lo primero es aprender a aprender.

La programación es en sí un sistema de aprendizaje muy bueno y muy gratificante. Te obliga a enfrentarte a problemas, pensar, buscar soluciones y desarrollar nuevas ideas. Por lo que al final, sin darte cuenta, adquieres unos conceptos muy importantes que más tarde, cuando se estudian en la teoría, como por arte de magia, aparecen, se relacionan y cobran sentido.

Por lo que contestando a la pregunta anterior; simplemente porque considero que el aprendizaje resulta más sencillo y aumenta el interés.

Al empezamos por un enfoque práctico irán apareciendo problemas que luego en la teoría cobrarán sentido y, por extensión, serán más sencillos de asimilar.

Requisitos.
Para poder seguir esta introducción se recomienda al lector:

- Disponer de unos conocimientos previos sobre programación. En los ejemplos se utiliza el lenguaje Python. Python es un lenguaje de programación muy intuitivo y sencillo, con una curva de aprendizaje muy pequeña. Por lo que, si el lector desconoce este lenguaje, pero está familiarizado con los *paradigmas de la programación* no va a ser un inconveniente.

- Disponer de un entorno de desarrollo para python para poder probar cada uno de los ejemplos mencionados.

- Interés y curiosidad. Esto sobretodo. Aunque si estás aquí seguramente sea por eso…

2.- Inteligencia Artificial, Machine Learning y Reinforcement Learning.

IA- Inteligencia Artificial.

La wikipedia define la *IA* o *Inteligencia artificial* es:

> *La inteligencia exhibida por máquinas. En ciencias de la computación, una máquina inteligente ideal es un agente racional flexible que percibe su entorno y lleva a cabo acciones que maximicen sus posibilidades de éxito en algún objetivo o tarea. Coloquialmente, el término inteligencia artificial se aplica cuando una máquina imita las funciones «cognitivas» que los humanos asocian con otras mentes humanas, como, por ejemplo: "aprender" y "resolver problemas". [...]*
>
> *En 1956, **John McCarthy** acuñó la expresión «inteligencia artificial», y la definió como: "...la ciencia e ingenio de hacer máquinas inteligentes, especialmente programas de cómputo inteligentes".*

Podéis obtener más información a partir del enlace a la wikipedia.

ML - Machine Learning.

Según la wikipedia el *Machiner Learning* (de ahora en adelante ML) o *Aprendizaje automático* es:

> *El subcampo de las ciencias de la computación y una rama de la inteligencia artificial cuyo objetivo es desarrollar técnicas que permitan a las computadoras aprender. De forma más concreta, se trata de crear programas capaces de generalizar comportamientos a partir de una información suministrada en forma de ejemplos. Es, por lo tanto, un proceso de inducción del conocimiento.[...] El aprendizaje automático puede ser visto como un intento de automatizar algunas partes del método científico mediante métodos matemáticos.*
>
> *El aprendizaje automático tiene una amplia gama de aplicaciones, incluyendo motores de búsqueda, diagnósticos médicos, detección de fraude en el uso de tarjetas de crédito, análisis del mercado de valores, clasificación de secuencias de ADN, reconocimiento del habla y del lenguaje escrito, juegos y robótica.*

Nuevamente, en el enlace de la wikiepdia podéis profundizar más en el tema.

RL - Reinforcement Learning.

El *Reinforcement Learning* (RL) o *Aprendizaje por Refuerzo* es una rama del *Machine Learning* que inspirado en la *psicología conductista* tiene por objetivo que un *agente* -software- aprenda a tomar decisiones inteligentes a través de la iteración con el entorno mediante la obtención de recompensas (positivas o negativas).

Considero suficientes estas tres definiciones para aclarar donde se ubica el *Reinforcement Learning* (RL) dentro de la IA y en que consiste.

3.- Practica 1.

***Let's Make a Deal.* Problema de Monty Hall.**

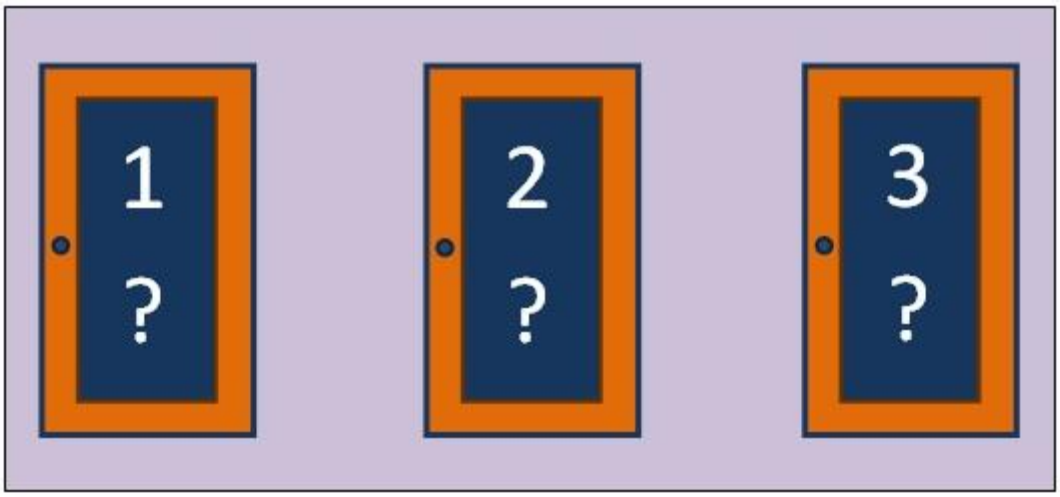
Como se ha mencionado en páginas anteriores, el objetivo es utilizar un problema real e intentar llevarlo a la práctica para experimentar con él.

Para esta primera prueba de concepto he decido utilizar el ***problema de Monty Hall.***

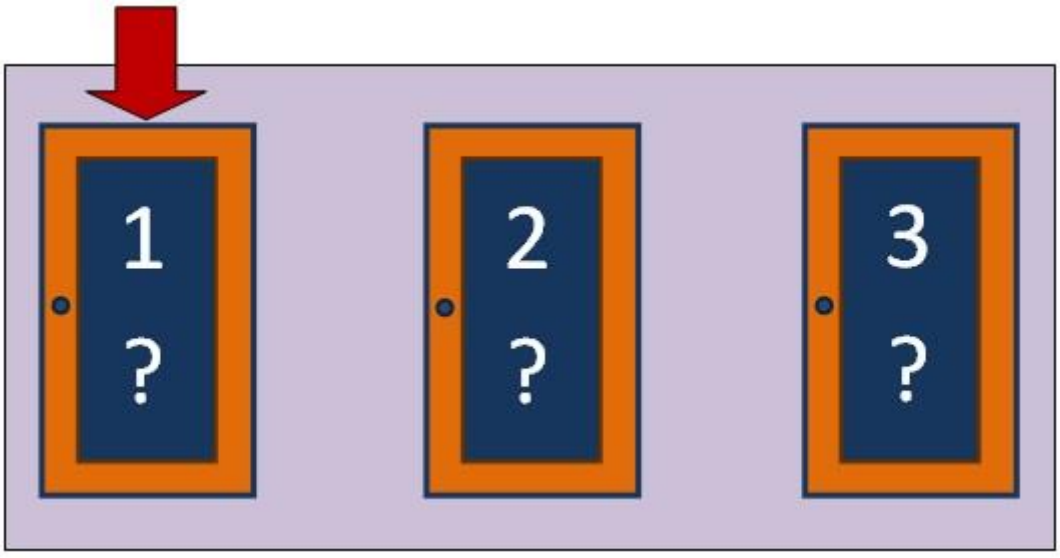
El ***problema de Monty Hall*** es un problema matemático de probabilidad basado en el concurso televisivo estadounidense *Let's Make a Deal - Hagamos un trato* -. El problema fue bautizado con el nombre del presentador de dicho concurso: *Monty Hall*.

A continuación, se explica en que consiste este concurso televisivo.

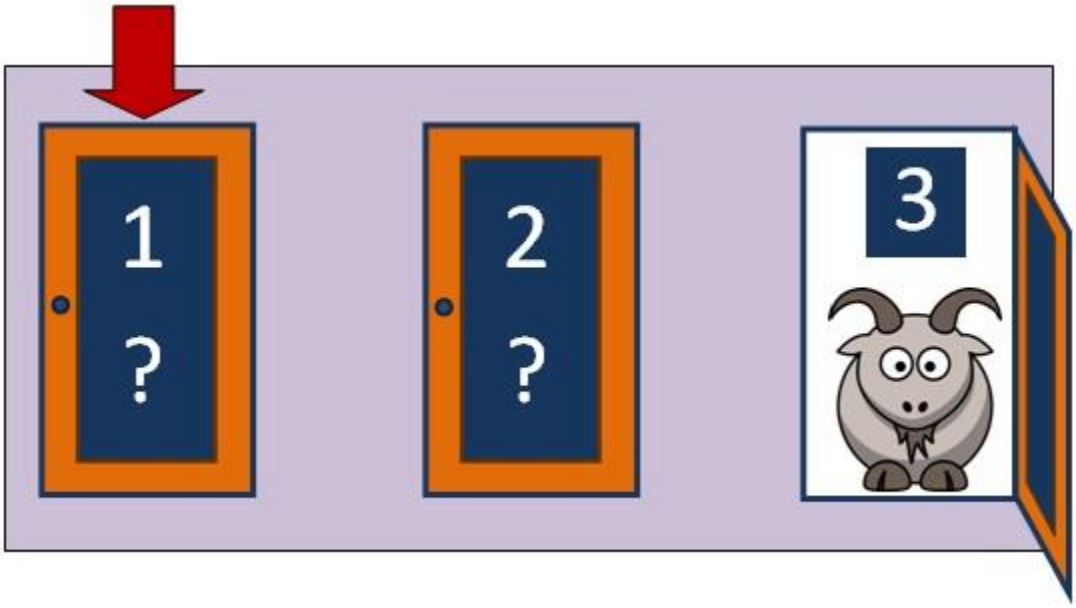
1. El presentador muestra al concursante tres puertas numeradas del 1 al 3. Detrás de una de las puertas hay un coche, mientras que en las dos restantes hay dos cabras.



2. El concursante escoge una de las 3 puertas con el objetivo de encontrar el coche. Supongamos que escoge la puerta número 1.



3. El presentador – *Monty Hall* – conoce en todo momento cuál es la puerta donde se encuentra el coche. Acto seguido, ***Monty Hall***, abre una de las dos puertas que no ha elegido el concursante y en la que sabe que no se encuentra el coche. Por lo que detrás de la puerta que abra siempre habrá una cabra. Para este ejemplo, imaginemos que la puerta abierta es la número 3.



4. Acto seguido. El presentador le pregunta al concursante: - *¿Quieres cambiar de puerta?*
   El concursante debe elegir entre dos opciones:
   Cambiar de puerta, es decir, elegir la puerta número 2.
   Seguir con la puerta elegida desde el principio, es decir, la puerta número 1.

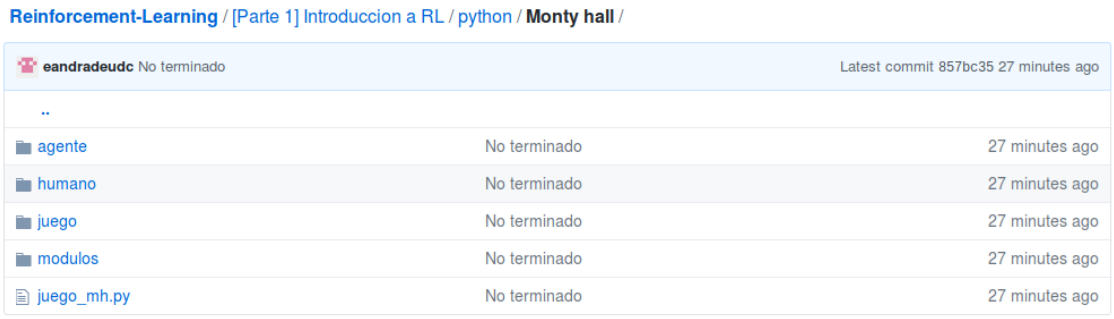
   El jugador gana si el coche se encuentra detrás de la puerta que finalmente ha elegido. En caso contrario, pierde si detrás de la puerta elegida se encuentra una cabra.

Esta es toda la información que necesitan conocer los jugadores para participar en este juego.

El juego de Mony Hall en Python.
En mi repositorio en GitHub disponéis de todo el material necesario para seguir este *paper*.
https://github.com/blogNetting/Reinforcement-Learning

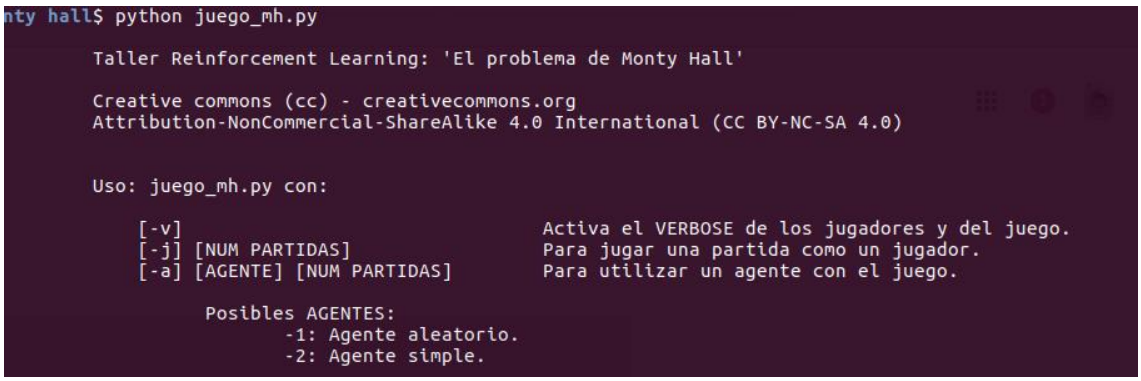
El código fuente para seguir este ejemplo se encuentra en la siguiente URL:
https://github.com/blogNetting/Reinforcement-Learning/tree/master/%5BParte%201%5D%20Introduccion%20a%20RL/python/Monty%20hall

Dentro del directorio ***Monty Hall*** se encuentra la siguiente estructura de ficheros.



Para poder interactuar con el juego se utilizará el fichero ***juego_mh.py***.

A continuación, una imagen que explica el uso del programa:



Para realizar pruebas como Jugador (persona):

python juego_mh.py -j 2

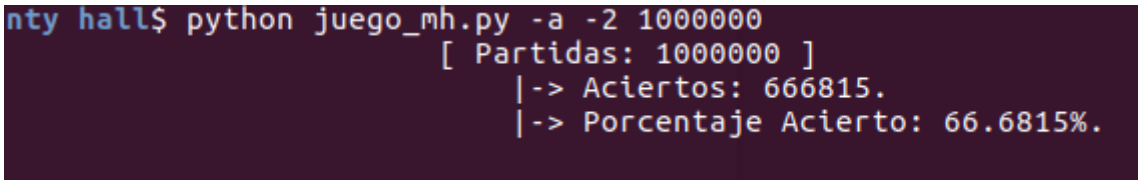
El juego se compone de varias partidas repetidas. Al finalizar, se muestran los aciertos y su probabilidad.

Se reta al lector a interactuar con el juego e intentar conseguir un porcentaje de acierto superior al 50% con el mayor número de partidas posibles.

¡Complejo! ¿verdad?

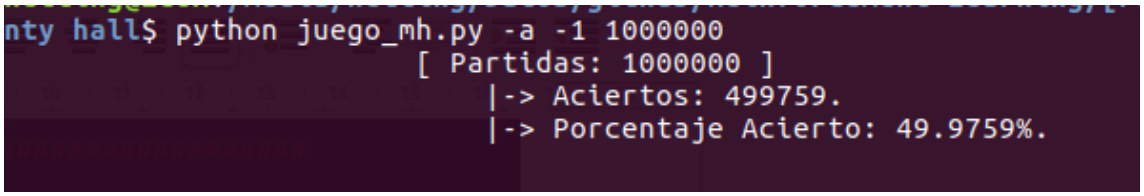

+--------+--------+--------+
PUERTA	PUERTA	PUERTA
1	2	3
		C
+--------+--------+--------+
¿Quieres cambiar de puerta (N=No, S=Si)?: s
##
¡Gano!
##
---------- [Partida 2] ----------
+--------+--------+--------+
PUERTA	PUERTA	PUERTA
1	2	3
+--------+--------+--------+
¿Que puerta eliges?: 1
+--------+--------+--------+
PUERTA	PUERTA	PUERTA
1	2	3
	C	
+--------+--------+--------+
¿Quieres cambiar de puerta (N=No, S=Si)?: n
##
¡Perdio!
##
[Partidas: 2]
|-> Aciertos: 1.
|-> Porcentaje Acierto: 50.0%.

Antes de comenzar con las explicaciones se comprobará la probabilidad de acierto del **agente** que he creado.



El **agente** obtiene una **probabilidad de acierto** que se aproxima al **66.66%.** Lo correcto en este problema (más adelante se desmostará esta afirmación).

También, para comparar varios casos, se ha desarrollado otro "**agente**" que toma todas las decisiones (elegir una puerta y realizar el cambio, o no) de forma aleatoria.



El "**agente**" aleatorio consigue una **probabilidad de acierto** cercano al **50%**.

Crea tu primer agente.
- *¿Cuál es el objetivo de esta primera prueba de concepto?.*
Crear un ***agente inteligente*** que a través de la obtención de recompensas, ya sean buenas o malas, aprenda a tomar las mejores decisiones para el problema de ***Monty Hall***.

El agente, debe de tomar las decisiones en función de su propia experiencia. Aprendiendo de sus errores y de sus aciertos.

Se anima al lector a crear un agente que pueda conseguir un **porcentaje de acierto** entorno al **66%**, cumpliendo las reglas anteriores de independencia y sin utilizar soluciones estadísticas.

Como se ha comentado anteriormente, el lenguaje de programación utilizado es ***Python***.

Para la creación del juego como para los **agentes** se ha utilizado la programación ***orientación a objetos***.
El Juego ***monty_hall.py*** como los dos agentes: ***aleatorio.py*** y ***simple.py*** son clases.

El presente artículo no pretende profundizar en aspectos como la programación orientada a objetivos ni en la explicación del juego.

El lector puede encontrar la clase ***AgenteLector*** en el fichero ***lector.py*** en la ruta:

/Reinforcement-Learning/[Parte 1] Introduccion a RL/python/Monty hall/agente.py

https://github.com/blogNetting/Reinforcement-Learning/blob/master/%5BParte%201%5D%20Introduccion%20a%20RL/python/Monty%20hall/agente/lector.py

Se recomienda al lector utilizar como recompensas:
- Positivas: +1.
- Negativas: -1.

```python
#!/usr/bin/env python
# -*- coding: utf-8 -*-

# Autor:        Enrique Andrade González - NeTTinG
# Web:          blog.netting.es
# email:        netting(at)netting(dot)es

# .::Reinforcement Learning::.
# [Parte 1] Introduccion a RL - Taller
# Practica 1 - "El problema de Monty Hall".

# Licencia:
# Creative commons (cc) - creativecommons.org
# Attribution-NonCommercial-ShareAlike 4.0 International (CC BY-NC-SA 4.0)


from random import randint
import sys
#sys.path.append('../juego')
from juego.monty_hall import MontyHall
#sys.path.append('../modulos')
from modulos.estadisticas import mostrar_estadistica, calcular_estadistica

class AgenteLector:

    def __elegir_puerta(self):
        # Completar #
        pass
        #return


    def __decidir_cambio(self):
        # Completar #
        pass
        #return


    def jugar(self, num_i, v=False):
        aciertos = 0
        porcentaje_acierto = 0.0
        for i in range(num_i):
            if(v): print "\t\t      ----------------- [ Partida " + str(i+1) + " ] -----------------"
            es_ganador = MontyHall().jugarIA(self.__elegir_puerta(), self.__decidir_cambio(), debug=v, guardar=False)
            aciertos, porcentaje_acierto = calcular_estadistica(es_ganador, aciertos, i)
            if(v): mostrar_estadistica(aciertos, porcentaje_acierto, i)
        if (not v):mostrar_estadistica(aciertos, porcentaje_acierto, i)

```

Se deben de completar los siguientes métodos:

- **__elegir_puerta():**
- **__decidir_camnio():**
- **__calcular_recompensas():**

Para realizar las pertinentes pruebas se debe de utilizar el fichero: ***juego_mh.py*** con los siguientes parámetros:

python juego_mh.py -a -3 10

Cambiando el número 10 por el número de partidas deseadas.

El lector también podrá utilizar la opción ***verbose*** (-v) para que el programa pueda mostrar más información por pantalla.

Agente para el problema de Monty Hall.
En este apartado se explicará el agente que he creado con el objetivo de acercarse al concepto de ***agente inteligente*** y comprender los problemas que existen para futuros *papers* donde se profundizará ya en conceptos importantes de RL.

Por lo que se pasará directamente a explicar la clase ***AgenteAleatorio*** a la que se puede acceder mediante la siguiente URL:

https://github.com/blogNetting/Reinforcement-Learning/blob/master/%5BParte%201%5D%20Introduccion%20a%20RL/python/Monty%20hall/agente/simple.py

o la siguiente ruta de directorio:

/Reinforcement-Learning/[Parte 1] Introduccion a RL/python/Monty hall/simple.py

Como se ha mencionado en el apartado anterior, un agente debe de:
- Ser independiente. Que no siga unas reglas o condiciones establecidas por el programador.
- Autodidacta. Aprender de sus decisiones mediante prueba y error.
- Capitalizar recompensas en función de su valor, negativo o positivo.
- Adaptarse a los cambios del entorno. Es decir, sin cambia el entorno es posible que el agente deba de adaptarse a los nuevos cambios.

El agente no conoce nada del entorno. Tan solo sabe que tiene que:
- Elegir una puerta entre 3 posibles.
- Tomar la decisión, o no, de cambiar de puerta.

Por eso se ha divido el problema en tres funciones.
- **__elegir_puerta():**
- **__decidir_camnio():**
- **__calcular_recompensas():**

En el constructor del ***AgenteSimple*** se inicializan tres variables.

La lista de recompensas:

self.recompensa = [-1, 1]

Una lista para almacenar las recompensas de cada puerta:

self.s = [0, 0, 0]

Por último, una lista de listas por cada una de las tres puertas. En la primera posición de la lista se guardan las recompensas obtenidas cuando no se realiza un cambio y en el segundo las recompensas obtenidas cuando se realiza el cambio

self.a_puertas = [[0,0], [0,0], [0,0]]

El método **__elegir_puerta()** devuelve la posición de la lista, es decir, el número de la puerta que tiene el mayor número de aciertos. En caso de empate, devuelve la primera puerta.

El método **__decidir_cambio()** accede la lista de listas donde se almacenan las recompensas de la puerta seleccionada en el método anterior. La variable *puerta* se le pasa al método.
Se compara las recompensas obtenidas en busca de la acción (cambio – no cambio) que tiene una mayor recompensa.
Si decide no hacer cambio devuelve 0 y si decide realizar un cambio devuelve 1.
En caso de empate, elige una acción al azar.

Como se observa hasta aquí, nada complicado.

Pasemos ahora a explicar el método que asigna las recompensas.

El método ***__calcular_recompensas(ganador, puerta, cambio)*** recibe tres parámetros:

- **Ganador:** Le indica al método si ha acertado y, por extensión, si va a recibir una recompensa positiva o negativa.
- **Puerta:** La puerta que ha elegido al comienzo del juego. Para poder guardar las recompensas en la puerta pertinente.
- **Cambio:** Para conocer si el agente ha realizado un cambio y guardar la recompensa en la posición de la lista de listas correspondiente.

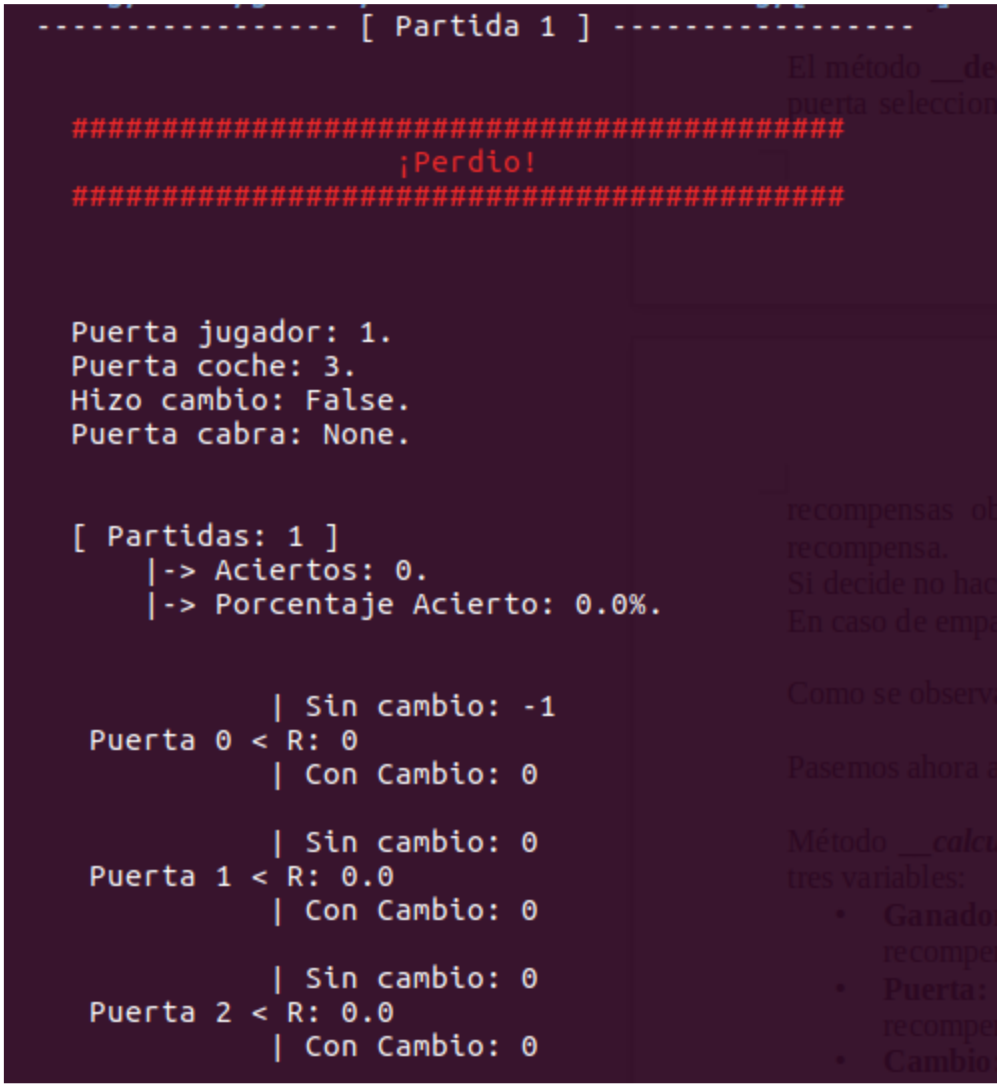
A la puerta seleccionada se le asigna el máximo valor de recompensa entre las dos acciones.

A continuación, se muestra cómo funciona todo esto. Para ello ejecutamos el fichero ***juego_mh.py*** con los siguientes parámetros:

```
python juego_mh.py -a -2 100 -v
```

Simplemente se ha activado el *verbose* (-v) para que se muestre más información por pantalla.

Se procede a explicar cómo actúa el agente en una partida real pasa o paso:

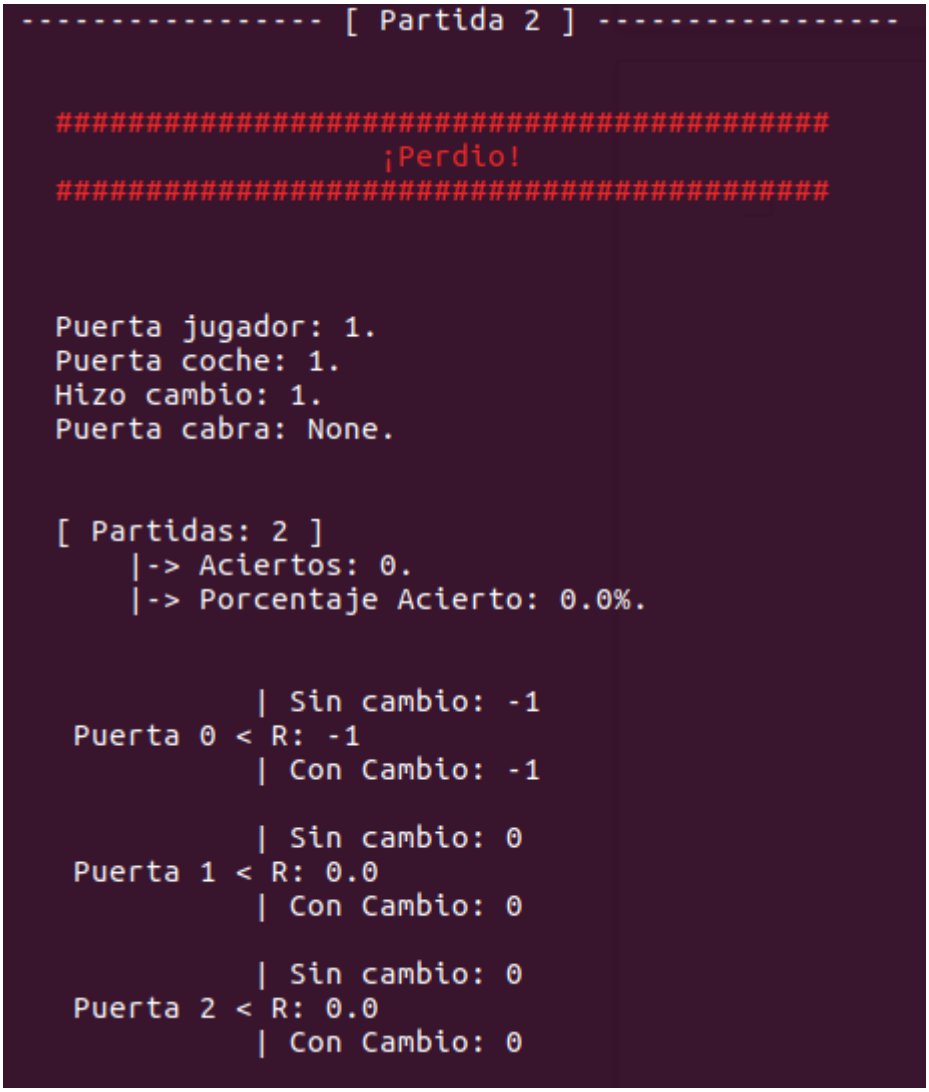


En la imagen de la **Partida 1** se puede observar que:

- El juego esconde el coche en la puerta 3.
- El agente elige la puerta 1.
- El agente ha decidido no hacer cambio.

Como el **agente** no ha acertado, el método asigna una recompensa negativa (-1) a la acción "*No cambio*" de la **Puerta 1** (posición de la lista 0).

El valor de la **Puerta 1** sigue siendo 0, puesto que la mayor recompensa entre las acciones "*No cambio*" = -1 y "*Cambio*" = 0, es 0.

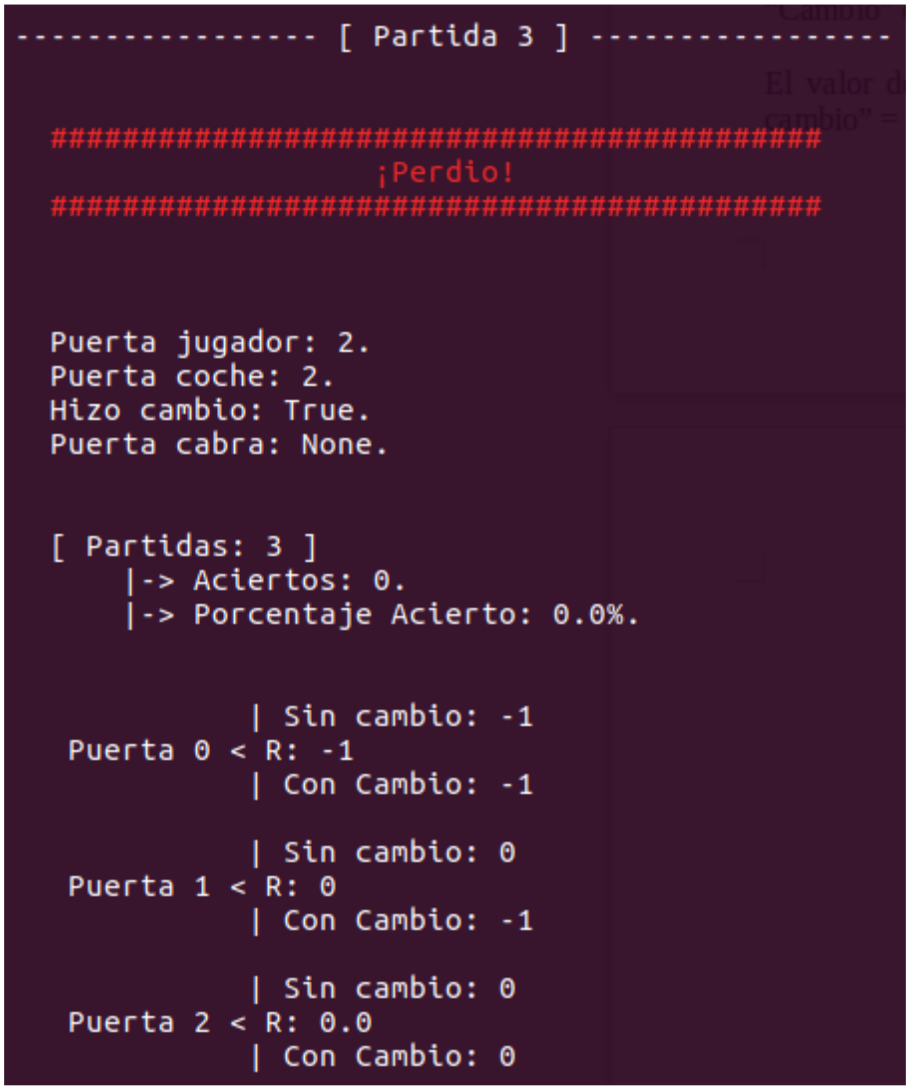


En la imagen de la **Partida 2** se puede observar que:

- El juego esconde el coche en la puerta 1.
- El agente elige la puerta 1.
- El agente ha decidido no hacer cambio.

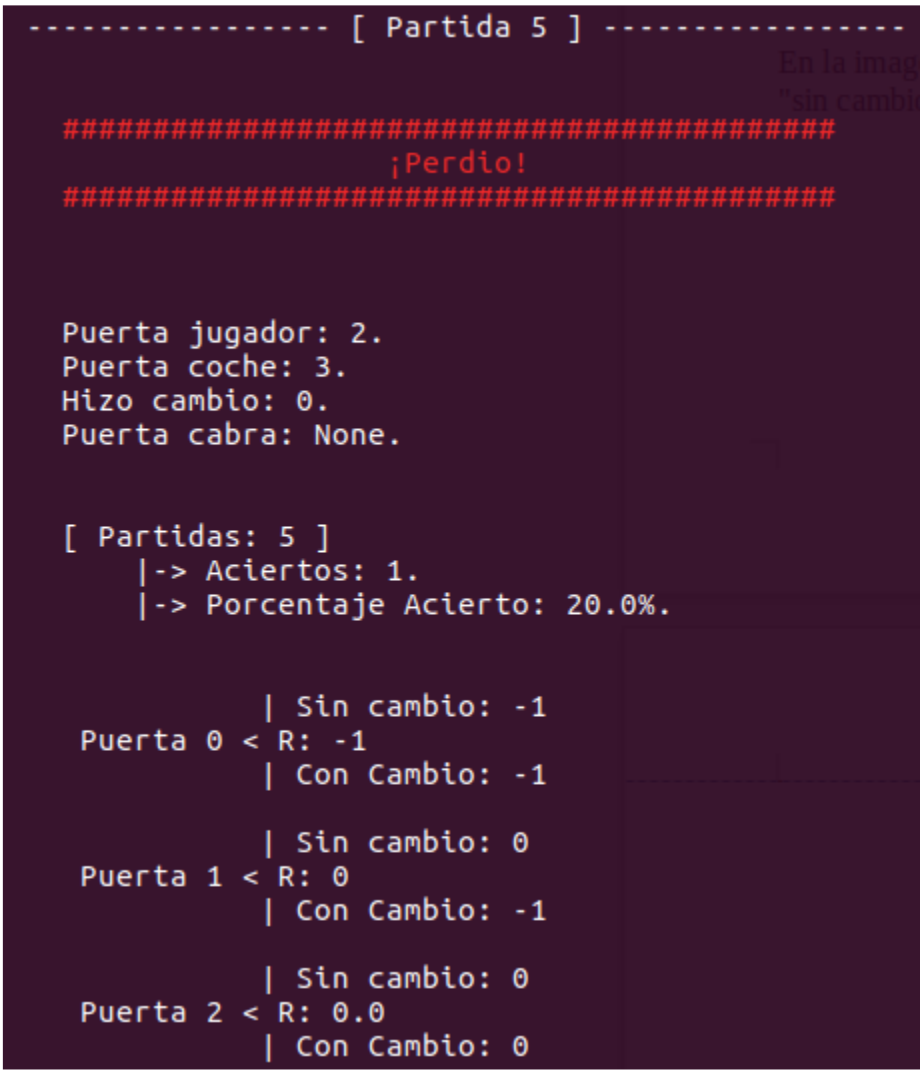
Como el **agente** no ha acertado, el método asigna una recompensa negativa (-1) a la acción "*Cambio*" de la **Puerta 1** (posición de la lista 0).

El valor de la **Puerta 1** pasa a ser -1, puesto que la mayor recompensa entre las acciones "*No cambio*" = -1 y "*Cambio*" = -1, es -1.



En la imagen de la **Partida 3** se observa que ocurre la misma situación que en la **Partida 2**, pero en este caso con la acción "*Cambio*" y con la **Puerta 2**.

En la imagen de la **Partida 4** el **agente** gana. Por lo que asigna una recompensa positiva a la acción "*sin cambio*" de la **Puerta 2**. La **Puerta 2** obtiene la recompensa de la acción "*sin cambio*."



En la **Partida 5**. El **agente** se equivoca. Por lo que se le asigna una recompensa negativa a la acción "*No cambio*" de la **Puerta 2**.
Se recuerda que anteriormente esta recompensa era 1. Pasa a ser 0 dado que 1 – 1 = 0. De igual forma, la máxima recompensa obtenida para la **Puerta 2** es 0. Máximo entre 0 y -1.

El **agente** va tomando decisiones en función de las recompensas obtenidas en deciones anteriores, por lo que irá almacenando y modificando sus listas.

A partir de un número determinado de partidas, el **agente** converge hacia una solución acertada, es decir, hacia una **política óptima**. Como se puede apreciar a partir de la **Partida 10**.

Esto se observa mejor en juegos con mayor número de partidas, donde el **agente** converge hacia una solución correcta.

En la imagen de la **Partida 1.000**. Se puede apreciar que la **Puerta 2** con la acción "*Con cambio*" obtiene considerablemente el mejor número de recompensas. Esto tiene un inconveniente. Se explicará más adelante…

Nuevamente, se aprecia que el **porcentaje de acierto** está entorno al **66%**. **Por lo que estamos en una solución correcta.**

La estadística en el problema de Monty Hall.
Seguramente, a estas alturas, el lector se pregunte porque razón la solución correcta debe de ser un **porcentaje de acierto** entorno al **66%**.

Sin entrar en profundidad en métodos estadísticos se procede a dar una justificación del ***problema de Monty Hall***.

Cuando el concursante o el agente elige una puerta, tiene una probabilidad de acertar del 33.33% (1/3) y una probabilidad de fallar del 66,67% (2/3).

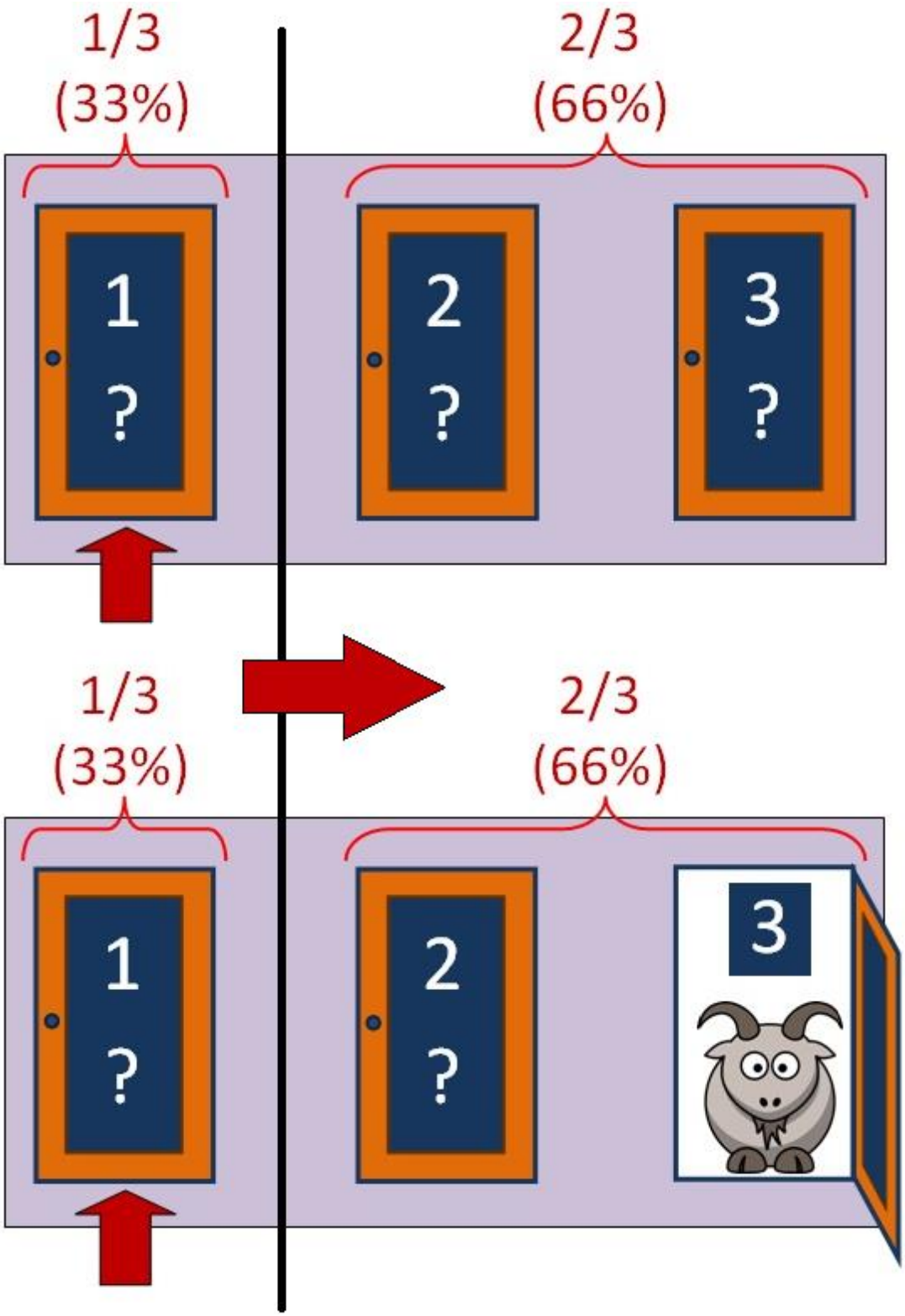
Pongamos un ejemplo:

El **coche** se esconde en la **Puerta 1**.
El **concursante** elige la **Puerta 2**.
El **presentador** abre la **Puerta 3** y le da la posibilidad de cambiar de puerta al concursante.

Si el concursante realiza el cambio, está cambiando también la **probabilidad de acierto**, es decir, pasa del 33.33% al 66,66%. Por lo que, siguiendo el ejemplo, ganarla partida.

Muchas personas piensan que la probabilidad de acertar en la situación de elegir el cambio de puerta es del 50% para ambas puertas, es decir, que tienen la misma probabilidad de acierto. Esto es erróneo. Puesto que hubo una elección posterior y la probabilidad sigue siendo la misma: un 33.33% frente a un 66.66%.

Por eso de que una imagen vale más que mil palabras se puede apreciar esta explicación en la siguiente imagen.


1/3
(33%)
2/3
(66%)
1
?
2
?
3
?
1/3
(33%)
2/3
(66%)
1
?
2
?
3

Se procede a explicar el problema exagerando enormemente el número de puertas. Las reglas del juego singuen siendo las mismas.

En esta ocasión el número de puertas totales será 100.
La probabilidad de acertar es de un 1/100, es decir, un 1%.
La probabilidad de fallar es de un 99/100, es decir, un 99%.

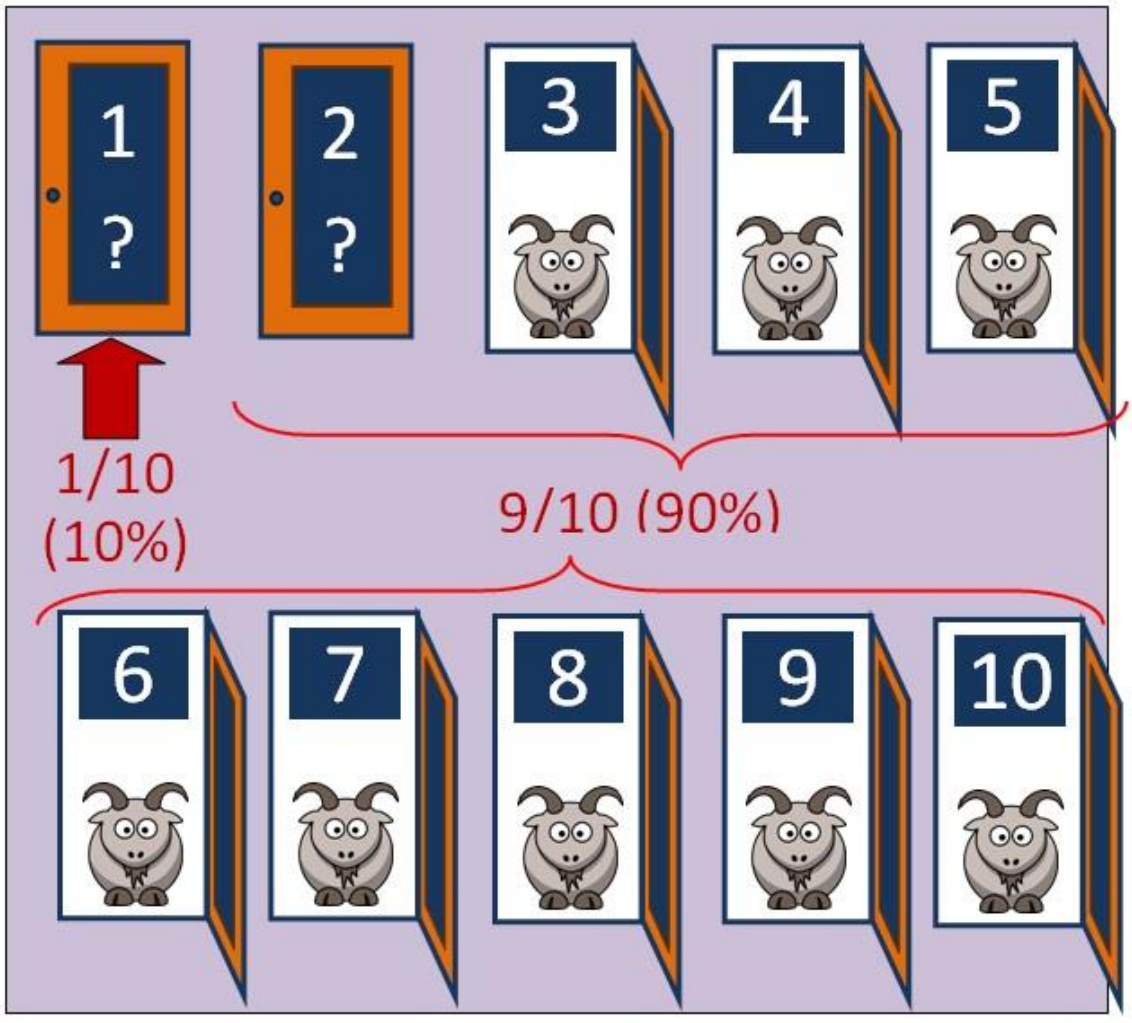
Si escogemos la **Puerta 1**, tenemos una probabilidad de acertar del 1%.
Vamos a obviar el hecho de que el presentador abre las puertas 98 puertas restantes y deja una sola, en su lugar vamos a pensar que el presentador dice:

- *¿Quieres la puerta número 1 o prefieres las otras 99 puertas restantes?*
Obviamente, para ganar, debemos de cambiar de opinión y quedarnos con las 99 puertas.

Ahora, se lanza la siguiente pregunta para hacer reflexionar al lector.

¿Qué diferencia existe entre cambiar las 99 puertas por la puerta 1 a eliminar las 98 puertas restantes y dejar una abierta?

Si se comprende bien la explicación se observa que el cambio de puerta, aunque sea solo una, tiene una probabilidad de acierto del 99%.



En el siguiente video de la película ***21BlackJack*** se explica este problema:
https://www.youtube.com/watch?v=THGJQuYFFpg

Retomando el juego de las tres puertas... Siempre que se hace el cambio, se obtiene una **probabilidad de acierto** del **66.66%** frente al **33.33%** de fallo. Por esta razón, si el agente arroja un **porcentaje de acierto** aproximado al **66,66%** estará tomando decisiones correctas. A mayor número de partidas más aproximado será el resultado final.

PoC - Cambiando las reglas.
Al principio de esta prueba de concepto se comentó que el **agente** debería de seguir unas determinadas reglas:

- Ser independiente. Que no siga unas reglas o condiciones establecidas por el programador.
- Autodidacta. Aprender de sus decisiones mediante prueba y error.
- Capitalizar recompensas en función de su valor, negativo o positivo.
- Adaptarse a los cambios del entorno. Es decir, sin cambia el entorno es posible que el agente deba de adaptarse a los nuevos cambios.

Entre otras cosas, estas características indican que si se cambian ciertos mecanismos del juego, también debe de cambiar la **política óptima**.

En esta prueba de concepto el juego (***monty_hall.py***) siempre va a esconder el coche detras de la **Puerta 3**.

```
class MontyHall:

    def __esconder_coche(self):
        #serie = [3,1,2,3,2,1,1,3,2,1,2,3,2,3,1,2,1,3,3,2,1,1,3,2,2,1,3]
        #return serie[randint(0, len(serie)-1)]
        return 3
```

Se procede a comprobar cómo se comporta el agente. Debería de arrojar una **probabilidad de acierto** próximo al **99,99%**.

```
[ Partidas: 100 ]
    |-> Aciertos: 99.
    |-> Porcentaje Acierto: 99.0%.

ting/Datos/github/Reinforcement-Learn
[ Partidas: 100 ]
    |-> Aciertos: 100.
    |-> Porcentaje Acierto: 100.0%.

ting/Datos/github/Reinforcement-Learn
[ Partidas: 100 ]
    |-> Aciertos: 99.
    |-> Porcentaje Acierto: 99.0%.

ting/Datos/github/Reinforcement-Learn
[ Partidas: 100 ]
    |-> Aciertos: 99.
    |-> Porcentaje Acierto: 99.0%.
```

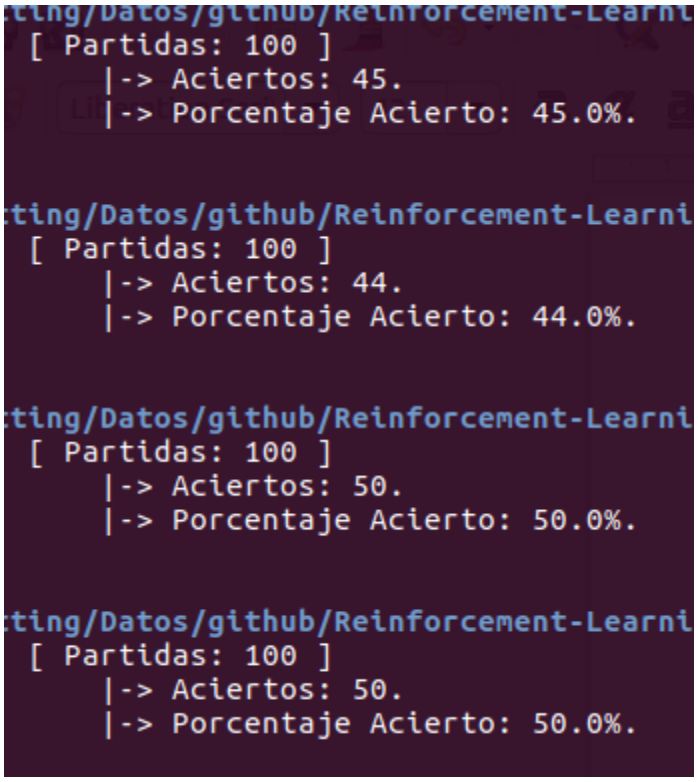
Dado este escenario el **agente** puede encontrar tres **políticas óptimas**.

- Elegir la puerta 1 y hacer el cambio.
- Elegir la puerta 2 y hacer el cambio.
- Elegir la puerta 3 y no hacer el cambio.

El agente descubrirá una de las tres posibles **políticas óptimas** en función del primer acierto.

Por lo que se puede concluir que el **agente** toma nuevamente decisiones correctas en función de la experiencia.

Con respecto al **agente aleatorio**, consigue una **probabilidad de acierto** entre el **45%** y el **50%**.

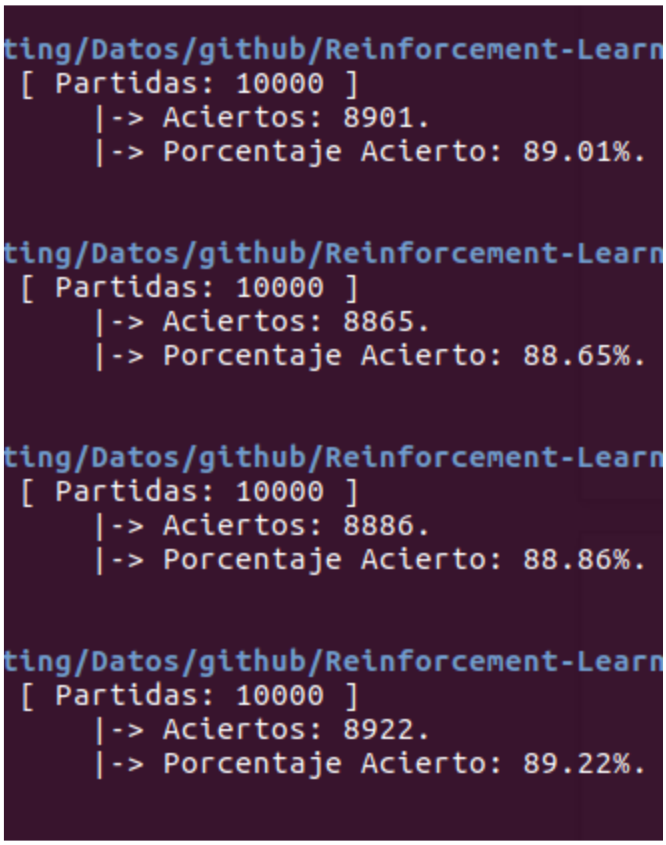


*¿Se daría cuenta el ser humano que **Monty Hall** esconde siempre el coche en la **Puerta 3**?*
Podría ser un experimento interesante...

PoC - Números calientes.
Para esta ocasión se va a cambiar el vector de números aleatorios, para que exista una mayor probabilidad de que el coche se esconda en la **Puerta 3**.

serie = [3,3,2,3,2,3,3,3,2,1,2,3,2,3,3,2,3,3,3,2,3,1,3,2,2,1,3]

Dada esta situación, el **agente** consigue una **probabilidad de acierto** del **88%**.



El **agente aleatorio**, nuvamente, obtiene un **porcenta de acierto** entorno al **50%**.

No es todo oro lo que reluce.
Hasta ahora, sin conocimientos previos, se ha creado un **agente** que tiene un comportamiento muy bueno ante diferentes escenarios de un mismo problema.

Lo habitual en estos casos es **entrenar** el **agente**, es decir, realizar una cantidad de partidas suficientes para que el **agente** converja y obtenga la **política óptima**.
Una vez que se obtiene la **política óptima**, esta se guarda, con sus "*experiencias*" y sus "*decisiones*". En este caso, las variables importantes serían:

- self.s.
- self.a_puertas.

De esta manera, cuando se utilizase nuevamente el **agente**, este cargaría dichas variables y no se cometería errores por "*desconocimiento*" o falta de "*experiencia*". Dicho de otra forma, no sería necesario que experimentase y cometiera errores, como, por ejemplo, no realizar el cambio de puerta.

El **agente** de las pruebas de concepto tiene varios problemas.
En el caso de tomar una decisión correcta, es decir, elegir una puerta *x* y cambiar de puerta, pero con la mala suerte de obtener una recompensa negativa, el **agente** penalizará muy drásticamente esa decisión y tardará bastante tiempo en volver a darle otra oportunidad.
Esto ocurre dado que el **agente** tiene una política muy conservadora, valora muchísimo las experiencias positivas y valora muy negativamente las experiencias negativas (aun siendo estas las correctas).

En la siguiente imagen, se puede ver un buen ejemplo:

Observando la **tabla de recompensas** se puede intuir las siguientes acciones:

- El **agente** se equivocó al menos una vez en la **Puerta 1** con la acción "*sin cambio*".
- El **agente** convergió muy rápidamente por la decisión: **Puerta 1** acción "*con cambio*". Esto provocó que el **agente** fuera conservador y que ni siquiera experimentara con las demás opciones.

PoC – La venganza de Monty Hall.
Dada la situación expuesta en la imagen anterior, podría ocurrir que el presentador decidiera empezar a esconder el coche en la **Puerta 1**, puesto que sabe que el agente siempre elige esta puerta.

En este escenario el **agente** cometería al menos 3.319 errores (la recompensa negativa tan solo reduciría las recompensas de 1 en 1) antes de cambiar de decisión o de política y converger hacia una nueva **política óptima**.

Es decir, hasta que la **recompensa** llegase al valor 0 el **agente** no tomaría la decisión de experimentar otras posibilidades.

Esto, sin lugar a dudas, es un problema importante y a tener en cuenta. Por lo que no estaríamos ante un **agente** eficiente.

No cumpliría la regla de: "*Adaptarse a los cambios del entorno.*"

Más adelante se explicará una posible solución a este problema.

PoC – (No) Todos los caminos llevan a Roma.
Otro problema que hay que tener en cuenta en estos casos es que la política descubierta por el **agente** no tiene por qué ser la mejor, la más óptima…

Un ejemplo:
Tenemos 10 caminos. 5 de los cuáles terminan con una recompensa negativa y los otros 5 en una recompensa positiva.
Recompensa negativa: -1.
Recompensa positiva: +1.
Todos los caminos tienen distancias diferentes entre ellos.

Es posible que el **agente** descubra uno de los caminos que arrojan una recompensa positiva, es decir, que encuentre una solución correcta. Pero eso no quiere decir que esa solución sea la mejor.
El **agente** siempre tiene que descubrir la **política óptima**.

Esto ocurre del mismo modo en seres humanos.

Un ejemplo:
Goretti quiere regalarle una ***Loop Tiebow*** a su novio. Por lo que visita la tienda online https://loopsco.es y compra el modelo "***Rock 'n' roll Tiebow***".

Cuando su amiga Marla le pregunta: - *¿Qué le has comprado a tu novio?*
Ella responde: - ¡*La pajarita más bonita y más original de todas las Loops Tiebow!*.

(Os recomiendo esta tienda para fardar de pajarita, ir modernos y elegantes. Sobre todo, ahora que se acerca el *fin de año*).

Goretti ha llegado a la conclusión de que está pajarita es la más bonita de todas… pero... ¿es esta una **política óptima**?

Pues depende de la siguiente pregunta: *¿Ha visto Goretti todo el catalogo?*

- Si la respuesta es sí: Hemos llegado a una **política óptima**.
- Si la respuesta es no: Hemos llegado a una política, pero sin ver todo el catalogo no podremos decir que se trata de una **política óptima**.

Podemos trasladar el ejemplo a un restaurante, solo podremos opinar de cuál es el mejor plato de la carta cuando los probemos todos.

Con el **agente** sucede lo mismo. Si el **agente** se ciñe única y exclusivamente a una única política no experimentará con otras posibilidades y no estará en condiciones de tomar la mejor elección.

4.- Soluciones.

Posibles soluciones:

Para solucionar los problemas expuestos en las dos últimas pruebas de concepto:

- PoC – La venganza de Monty Hall.
- PoC – (No) Todos los caminos llevan a Roma.

Hay que hablar del ***dilema exploración-explotación***. Preguntarse:

- ¿Cuándo **explotar** lo que ya sabemos?
- y… ¿cuándo seguir **explorando** mejores opciones?

Mediante la ***explotación-exploración*** buscamos que el **agente** descubra todas las posibles soluciones a base de experiencias (ya sean positivas o negativas). Es decir, el agente no convergerá tan solo a una política conservadora, en ciertas ocasiones, a través de cierta estrategia el agente en vez de tomar la decisión más correcta hasta el momento se aventurará en nuevas posibilidades y decisiones.

Si aplicamos estos conocimientos a nuestro **agente**, teóricamente, este, debería de repartir las recompensas positivas (a partir de una convergencia) entre las 3 puertas con la acción "*cambio*" de manera equitativa.

Si utilizamos como ejemplo lo ocurrido en la última imagen (imagen XXX) el **agente** debería de repartir las 3.319 recompensas entre las 3 puertas con la acción "*cambio*" equilibradamente, puesto que estas serías las mejores opciones.

En este caso, las recompensas positivas en estas decisiones estarían entorno a las 1.106 recompensas positivas. Seguramente unas puertas tendrían menos recompensas que otras. Dependerá de la "*mala suerte*" que haya tenido el **agente** en errar en una decisión correcta.

Si utilizamos la "***PoC – La venganza de Monty Hall***", donde el presentador a partir de las 10.000 partidas decide esconder el coche siempre en la puerta donde el **agente** tiene la mayor recompensa y aplicamos en el **agente** la estrategia de ***exploración-explotación*** el **agente** no tardaría mucho tiempo en encontrar una decisión más correcta, incluso casi sin cometer errores ni tener que explorar demasiado.

Esto sucede dado que, al ser las recompensas equilibradas entre las 3 mejores opciones, saltaría a la siguiente opción más correcta en función de experiencias pasadas. Con lo que terminaría convergiendo hacia una nueva **política óptima** en poco tiempo y con un menor números de errores.

Veamos un ejemplo:

Vamos a utilizar la siguiente nomenclatura:

- SC: Sin cambio.
- CC: Con cambio.

Vamos a utilizar estos dos agentes, con sus correspondientes **políticas óptimas** con un entrenamiento de 1.000 partidas.

Agente Simple:
Puerta1: 3.319
SC: -1
CC: 3.319
Puerta2: 0
SC: 0
CC: 0
Puerta3: 0
SC: 0
CC: 0

AgenteExploraciónExplotación:
Puerta1: 1.090
SC: -2
CC: 1.090
Puerta2: 1.080
SC: 2
CC: 1.075

Puerta3: 1.088
SC: 5
CC: 1.080

Comportamiento de los dos agentes en la "***PoC – La venganza de Monty Hall***".
El coche se esconde siempre en la **Puerta 1**.

AgenteSimple: Este **agente** ha realizado una convergencia conservadora. Utiliza siempre la decisión que mejor funcione sin explorar otras opciones. Las recompensas tienden a infinito según el número de partidas.
Como el coche se encuentra escondido en la **Puerta 1**, lo que va a suceder es que el **agente** va a seguir utilizando la decisión: **"*Puerta 1 – Con cambio*"**. Porque según su experiencia es con creces la mejor opción. Esto quiere decir que, hasta que las recompensas de esta decisión no sean igual a 0, el **agente** no va a plantearse otras posibilidades. Por lo que el **agente** estaría condenado a equivocarse 3.319 veces.

AgenteExploraciónExplotación: Este **agente** (aunque todavía no sabemos cómo) ha encontrado una **política óptima** a través de la ***exploración-explotación*** por lo que conoce las tres mejores decisiones al problema.
Dada esta situación, las recompensas positivas fueron asignadas de forma uniforme entre las tres mejores decisiones. Con diferencias menores debido a decisiones correctas con "*mala fortuna*".
En este caso, dado que las tres decisiones correctas están muy próximas, en el peor de los casos, el **agente** estaría condenado a equivocarse en tan 10 ocasiones, las recompensas de **Puerta 1** menos las recompensas de la **Puerta 3**.

R_Puerta1 – R_Puerta2 = 1.090 – 1.080= 10

Es importante fijarse en que digo "*en el peor de los casos*" puesto que como veremos más adelante, en próximas partes, esto no tiene por qué ser así, depende de la estrategia que utilicemos para *explorar* y de otros factores.

Con los números encima de la mesa vemos que el ***AgenteSimple*** es tremendamente ineficiente ante esta situación.

Existen otros inconvenientes. No es una buena opción utilizar recompensas que tiendan a infinito. Es mejor utilizar otro tipo de estrategias. Esto será en futuros artículos.

5.- Conclusiones.

Conclusiones finales.

Esta introducción sobre RL tiene como objetivo introducir al lector de forma muy sencilla, con un enfoque práctico, en los primeros conceptos de lo que pretende ser el **aprendizaje por refuerzo**.

Se ha buscado que el lector entienda como funcionan y cómo se comportan estos agentes inteligentes ante ciertos entornos.

No se ha estudiado conceptos ni terminología adecuada, más allá de ciertos conceptos en los que tampoco se ha profundizado mucho ni se ha dado una definición correcta.

Al finalizar esta parte, el lector debería ser capaz de resolver problemas sencillos utilizando el **aprendizaje por refuerzo**, estudiar el **agente**, y detectar ciertos problemas y debilidades.

Los estrategias y códigos que se han utiliza en el **AgenteSimple** están muy lejos de lo que proponen los algoritmos y estrategias utilizadas en el verdadero **aprendizaje por refuerzo**. El **AgenteSimple** presentado en esta parte no debe tomarse como algo definitivo. Tan solo como un primer contacto sencillo para ayudar al lector a introducirse en el **aprendizaje por refuerzo** a través de ejemplos prácticos.

En futuras partes empezaremos a definir y estudiar conceptos fundamentales en el RL.

Autor: Enrique Andrade – NeTTinG
http://blog.netting.es