Universidad Politécnica de Cartagena

Escuela Técnica Superior Ingeniería de Telecomunicación

Moreno Lorente, Sergio

Control de un protocolo anticolisión para RFID con aprendizaje reforzado

Trabajo fin de grado

para obtener el grado universitario en

Ingeniería telemática

Universidad politécnica de Cartagena

Supervisor

Dr. Juan J. Alcaraz Espín

Escuela técnica superior de ingeniería de telecomunicación

Cartagena, Septiembre 2019

Agradecimientos

A mi familia, el apoyo incondicional del que tan orgulloso me siento; nunca podré devolveros todo el cariño que me dais. Os quiero.

A la "Formación Tortuga"(Joaquín, Alfredo, Alex Gil, Alejandro Sabater, Antonio Manuel y Jesús), por la amistad construida, todos y cada uno de vosotros habéis sido esenciales en esta etapa de mi vida. Gracias.

A Juan José Alcaraz Espín que incluso en la distancia ha sabido dirigirme con rigor, aceptar mis peculiaridades y sobre todo darme la oportunidad de desarrollar un trabajo como este.

A mis abuelos, Manolo y Juan, que en paz descansen.

Índice general

Índice de figuras

Índice de figuras

1. Introducción

Al final de estas líneas descubriremos que lo que a priori parecerá magia está basado en resolver una simple ecuación, que jugar con las matemáticas nos puede llevar a conseguir cosas que ni imaginamos y que su buen uso puede facilitar y ofrecer inteligencia a cosas que aparentemente solo ejecutan órdenes.

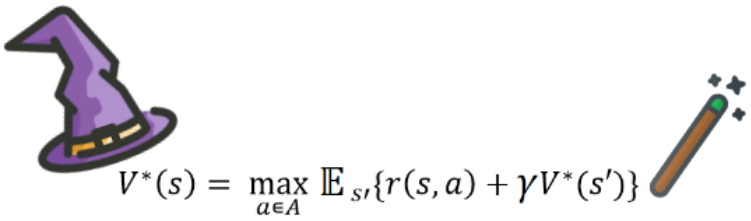

Figura 1.1.: Ecuación de Bellman animada

Podemos entender la inteligencia artificial como la realización de procesos humanos llevados a cabo por máquinas. No hablamos de procesos como digerir la comida o rascarse el brazo si te pica, hablamos de tomar decisiones de forma inteligente, es decir, frenar el coche cuando viene una curva, dejar de andar cuando estamos frente a un precipicio o tomar este o aquel atajo para llegar antes a un destino. Más adelante entraremos en conceptos técnicos de la inteligencia artificial y la parte de ésta que trataremos de exprimir en este proyecto, el *aprendizaje reforzado (reinforcement learning)*, pero de momento situaremos el contexto para empezar a trabajar.

La cantidad de aplicaciones sobre las que poder trabajar con *reinforcement learning* se nos puede escapar un poco de la razón y es que una vez que lo pones en práctica y observas su potencial solo la imaginación puede ponernos límites. Dentro de esta interminable espiral de aplicaciones nos

vamos a centrar en un sistema de lectura de tags[1] por RFID (*Radio Frequency Identification*) donde nuestro propósito será minimizar el tiempo en el que el lector identifica dichos tags. Para hacer el escenario más visual podríamos imaginarnos un proceso industrial donde varios elementos deben ser identificados a su paso por un *checkpoint* o incluso una cinta de supermercado en la que el cliente deja sus productos y el lector de códigos de barras captura estos tags para calcular el precio final de la compra. El escenario más simplificado para nuestro proyecto se presenta gráficamente a continuación y consta de una serie de tags estáticos que se presentan bajo el alcance de un lector:

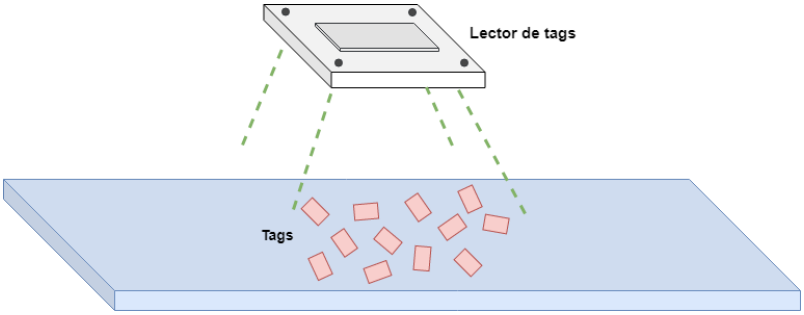

Figura 1.2.: Escenario simplificado

Una vez presentado nuestro "espacio de trabajo" vamos a hacer un pequeño recorrido introductorio por los puntos que se van a ir tocando hasta llegar finalmente a que nuestro lector de tags sea inteligente y aprenda a identificarlos de tal forma que el tiempo se minimice.

En primer lugar, estudiaremos el estándar más actual de protocolos de identificación por radiofrequecia (GS1, 2018) para que nuestra aplicación sea lo más real y precisa posible. En él nos centraremos en el estudio de todos los requerimientos físicos, lógicos y parámetros del protocolo, al igual que en la notación, símbolos y abreviaturas pertinentes.

Una vez que hayamos comprendido el estándar anteriormente introducido, será hora para crear un simulador acorde a todos los requerimientos que nos reclama el protocolo de indentificación por RFID. Este simulador será

[1] cualquier elemento pasivo activado por radiofrecuencia con el que poder mantener una comunicación.

desarrollado en lenguaje de programación PYTHON por la sencilla razón de que la mayor parte del desarrollo actual de *reinforcement learning* está basado en dicho lenguaje; además éste nos ofrece una gran cantidad de facilidades de programación que podremos exprimir al máximo para facilitar la implementación de nuestro simulador y realizarle los cambios pertinentes que introduciremos más adelante.

El siguiente paso será dotar a nuestro simulador (lector de tags por RFID) de inteligencia. Entraremos más en detalle en los términos pertenecientes a la inteligencia artificial y más concretamente en el *RL*[2], construyendo poco a poco las expresiones y relaciones matemáticas que acabarán dando vida a nuestro simulador y harán que se comporte como deseamos, la magia de la que venimos hablando.

Tras este estudio teórico y matemático preciso, será el momento de modificar nuestro simulador en PYTHON de acuerdo con los estándares creados y promovidos de OPENAI [3] https://openai.com/. Nos centraremos principalmente en un recurso que esta compañía ha desplegado para promover el uso de la inteligencia artificial como es GYM, el cual presentaremos más en profundidad cuando llegue su momento.

Podemos decir que GYM es una plataforma de desarrollo basada en PYTHON que nos ofrece librerías específicas y clases especiales que serán las que implementaremos en nuestro simulador para poder aplicar los algortimos pertinentes de RL.

Existen otros numerosos tipos de algoritmos que serán explicados en profundidad cuando llegue su turno y que serán implementados en el principal lenguaje de este proyecto para que se pueda trabajar acorde a las especificaciones que nos impone GYM.

Finalmente, será el momento de contrastar los resultados de las simulaciones entrenando nuestro simulador con los diferentes algoritmos de RL y observando que la mejora perseguida se obtiene correctamente, en nuestro

[2] *reinforcement learning* a partir de ahora

[3] es una compañía de investigación de inteligencia artificial (IA) sin fines de lucro que tiene como objetivo promover y desarrollar inteligencia artificial amigable de tal manera que beneficie a la humanidad en su conjunto. https://es.wikipedia.org/wiki/OpenAI

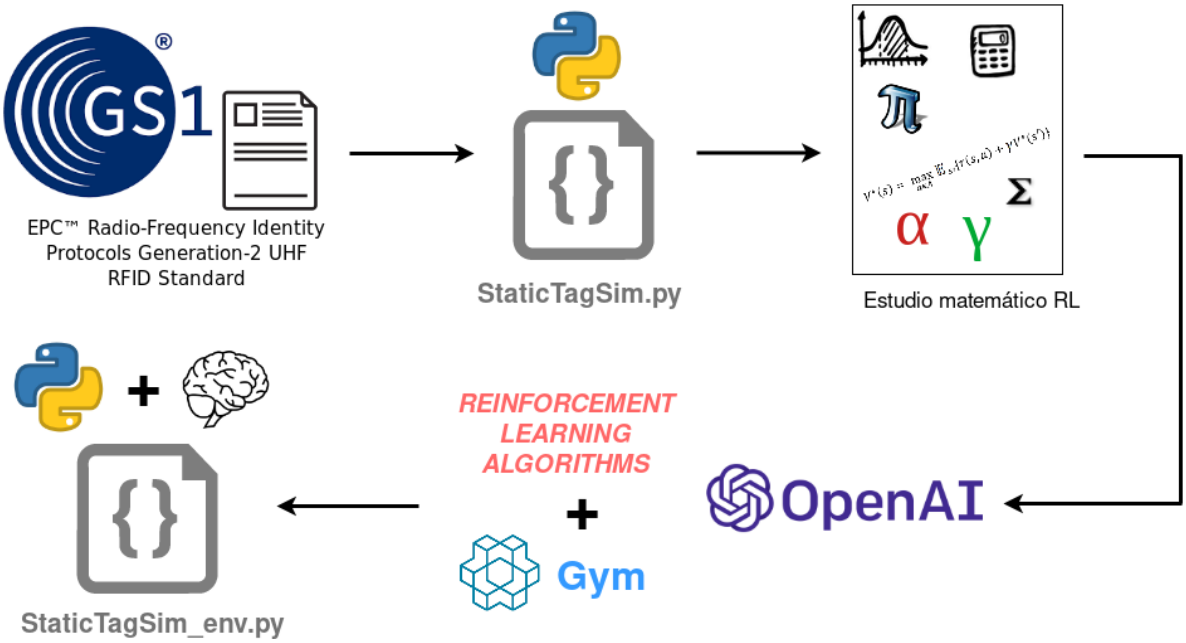

Figura 1.3.: Resumen introductorio del proceso

caso no debemos olvidar que este objetivo es minimizar el tiempo de lectura (identificación) de una serie de tags estáticos.

Pues bien, una vez se ha introducido el camino que se va a llevar a cabo y las herramientas que vamos e emplear en el mismo, es hora de ir desgranando y entrando más en detalle en todos los puntos que sutilmente se han descrito hasta ahora.

2. Estándar de protocolos para RFID

GS1 es una organización a nivel mundial que está formada por distintas empresas privadas y que ofrece una serie de productos, servicios y soluciones con el fin de mejorar la eficiencia de las cadenas de la oferta y la demanda. Posee varios sectores e industrias como son GS1 BarCodes, GS1 eCom, GS1 GDSN o EPCglobal[1]. Éste último corresponde con el "Código Electrónico de Productos" y es el que vamos a explorar para entender el protocolo de RFID y poder aplicarlo correctamente a nuestro simulador de identificacion de tags.

Las especificaciones oficiales que concretamente nos ofrece este protocolo son:

1. Interacciones físicas entre el lector y los tags
2. Proceso de operación lógico y comandos entre el lector y los tags.

2.1. Elementos y proceso general

La especificación del protocolo se realiza en el rango de frecuencias de UHF [2], más concretamente donde operan los sistemas de RFID, 860 MHz - 960 MHz. El sistema comprende a los *lectores* (también se refiere a ellos como "Interrogadores") y los *tags* (también conocidos como "etiquetas").

Un *lector* transmite información a un *tag* modulando una señal en el rango de frequencias indicado anteriormente. El *tag* recibe la energía de la señal RF

[1] Fuente: GS1, 2018

[2] Ultra-High frequency que abarca desde los 300MHz hasta los 3 GHz

junto con la información. Al tratarse de un elemento pasivo toda la energía que los *tags* necesitan para funcionar proviene del lector. La respuesta de éstos llega de vuelta al *lector*. Al tratarse de un sistema ITF[3] los *tags* modulan el coeficiente de reflexión de su antena gracias a la información que les llega del *lector* con el fin de responderles posteriormente.

La comunicación entre ambas partes no se realiza de forma simultánea, es decir, la comunicación es *half-duplex*, lo que significa que mientras uno habla el otro escucha, o viceversa.

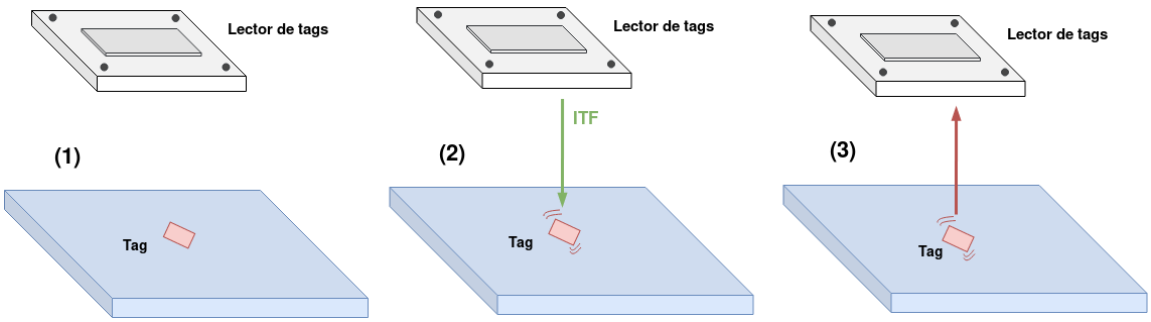

Figura 2.1.: Proceso activación *tag* con ITF y *half-duplex*

2.2. Requisitos de cumplimiento - lector y tags

Tanto el *lector* como los *tags* deben:

- cumplir los requerimientos exigidos por este protocolo.
- implementar los comandos obligatorios definidos.
- cumplir las regulaciones de radiofrecuencia locales allá donde se desplieguen.
- por parte del *lector*, modular y demodular una cantidad suficiente de señales eléctricas definidas en la capa de señalización del protocolo para comunicarse con los *tags*.
- por parte de los *tags*, modular una señal en respuesta al *lector* una vez que ha recibido los comandos apropiados.

[3]Interrogator-talks-first

Por otra parte, el *lector* y los *tags* tendrán permitido:

- implementar cualquier comando opcional definido en el protocolo.
- implementar cualquier comando especial/personalizado que sea compatible con el protocolo.

En último lugar, el *lector* y los *tags* no deberán:

- implementar ningún comando que entre en conflicto con el protocolo.
- en el caso de los *tags*, modular una señal de vuelta al *lector* a no ser que este se lo ordene y se haga bajo la capa de señalización adecuada.

2.3. Capa física

El *lector* envía información hacia uno o varios *tags* modulando una portadora RF usando una de las siguientes modulaciones:

- DSB-ASK[4]
- SSB-ASK[5]
- PR-ASK[6]

Todas ellas harán uso de "Pulse Interval Encoding - (PIE)" que es un métido de transmisión de datos a *tags* de RFID emitiendo pulsos de energía en varios intervalos para indicar con esos pulsos los unos y ceros binarios almacenados en esa etiqueta.

El *lector* recibe información de un *tag* mientras él transmite una portadora sin modular y escucha la respuesta por parte del elemento pasivo. Los *tags* se comunican mediante la modulación de la amplitud y/o la fase de la señal portadora. Como se ha comentado anteriormente, el enlace de comunicación entre ambos extremos es *half-duplex* lo que significa que los *tags* no tendrán que demodular la señal del *lector* mientras están llevando a cabo su particular respuesta.

[4]double sideband amplitude shift keying
[5]single-sideband amplitude shift keying
[6]phase-reversal amplitude shift keying

2.4. Capa de identificación de tags

Un *lector* gestionará una población de *tags* usando tres operaciones básicas:

1. **Select** - elegir una población de *tags*. Con un comando del tipo "select" es capaz de seleccionar los *tags* con los que posteriormente trabajará y tendrá que identificar. También pueden intervenir en el proceso los temas relacionados con encriptación de la información y autenticación.
2. **Invertory** - Identificar un *tag*. Se empieza una ronda de identificación transmitiendo un "*Query*". Las opciones a partir de aquí sin claras, pueden responder 0, 1 o varios *tags*. Cuando el *lector* detecta la respuesta de un *tag* le solicita su EPC[7].
3. **Access** - Comunicación con un *tag* identificado. A partir de aquí entrarían en juego numerosos procesos y comando especiales con los cuales el *lector* podrá leer, escribir, bloquear o eliminar información de los *tags*.

En el caso de nuestro futuro simulador partiremos de una población inicial de *tags* conocida por lo que el proceso de "select" no será implementado como tal. En realidad este no es un problema ya que no debemos olvidar que nuestro propósito final será la aplicación de algoritmos de RL para minimizar el tiempo de identificación en las rondas descritas en la operación "2. Inventory".

En las secciones sucesivas entraremos en detalle sobre como son y qué partes tienen estas rondas de identificación, ya que la precisión en la medida del tiempo gastado es clave para obtener unos resultados apropiados.

En la siguiente imagen vemos de forma simplificada como se llevan a cabo las tres etapas de identificación de un *tag*, partiendo de una población aleatoria. En los bocadillos superiores sobre el *lector* se incluye información que va aprendiendo de los *tags* y que pondremos de manifiesto más adelante cuando hablemos del desarrollo del simulador en Python.

[7]Electronic Product Code: información contenida en la memoria del *tag*

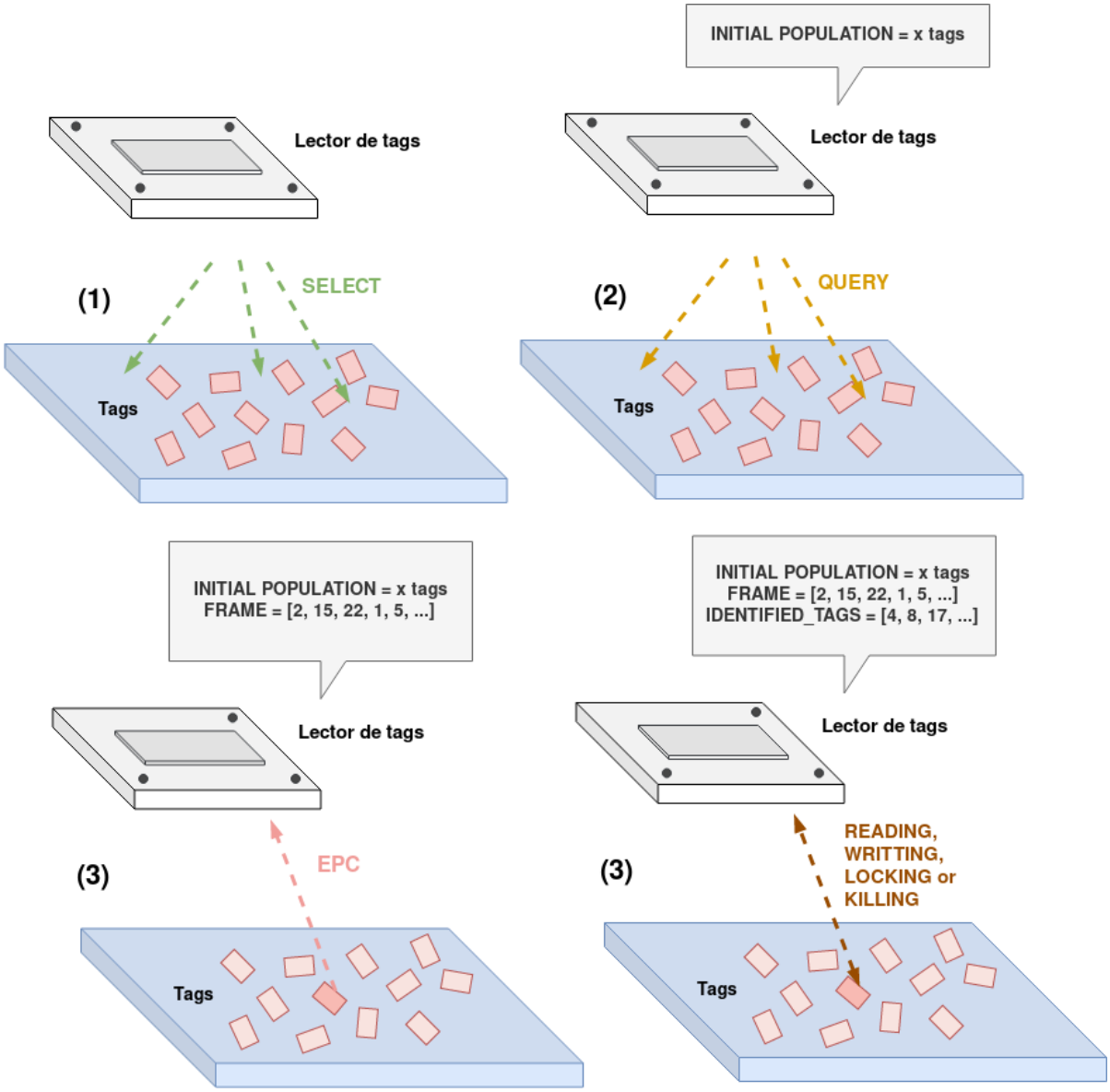

Figura 2.2.: Operaciones básicas de identificación de *tags*

2.5. Descripción básica del proceso de operación

Aunque en los puntos anteriores hemos desarrollado de forma genérica como se lleva a cabo la comunicación entre *tags* y *lector*, en esta nueva sección hablaremos de ciertas especificaciones más precisas que abarcan requerimientos físicos y lógicos del protocolo.

2.5.1. Orden de transmisión

El orden de transmisión para las comunicaciones en el sentido *tags-lector* o en el sentido *lector-tags* debe ser enviando el bit más significativo primero (MSB). Esto abarca a que dentro de cada mensaje se envíe primero la palabra más significativa y que dentro de cada palabra se envíe primero el bit más significativo.

2.5.2. Verificación de redundancia cíclica

Conocidos con las siglas CRC, los códigos de redundancia cíclica son usados para detectar errores en comunicaciones en redes digitales o en dispositivos de almacenamiento en los cuales accidentalmente pueden detectarse cambios en los datos. Gracias a estos códigos es posible detectar e incluso en ocasiones corregir estos errores en el mismo proceso de transmisión, sin necesidad de reenvío.

En nuestro caso, estos códigos servirán para asegurar la validez de los comandos en el sentido *lector-tags* y para que el *lector* verifique la validez de las respuestas por parte de los *tags*.

No entraremos más en detalle con los códigos CRC ya que no es un figura de gran interés en nuestro proyecto, pero si cabe tener en cuenta que el protocolo los precisa en su implementación.

2.5.3. Memoria de un tag

La memoria de un tag está dividida en 4 bancos de memoria:

1. Reserved memory: contiene las contraseñas de acceso o de eliminación de *tag* del proceso.
2. EPC memory: debe contener un código CRC, un código que sirve para identificar al *tag* y que sea diferenciado de los demás.
3. TID memory: (Tag-identification) contiene características de identificación que el *lector* debe conocer para llevar a cabo la identificación de dicho *tag*.
4. User memory: (opcional) sirve para almacenar en el *tag* información en forma de uno o más archivos.

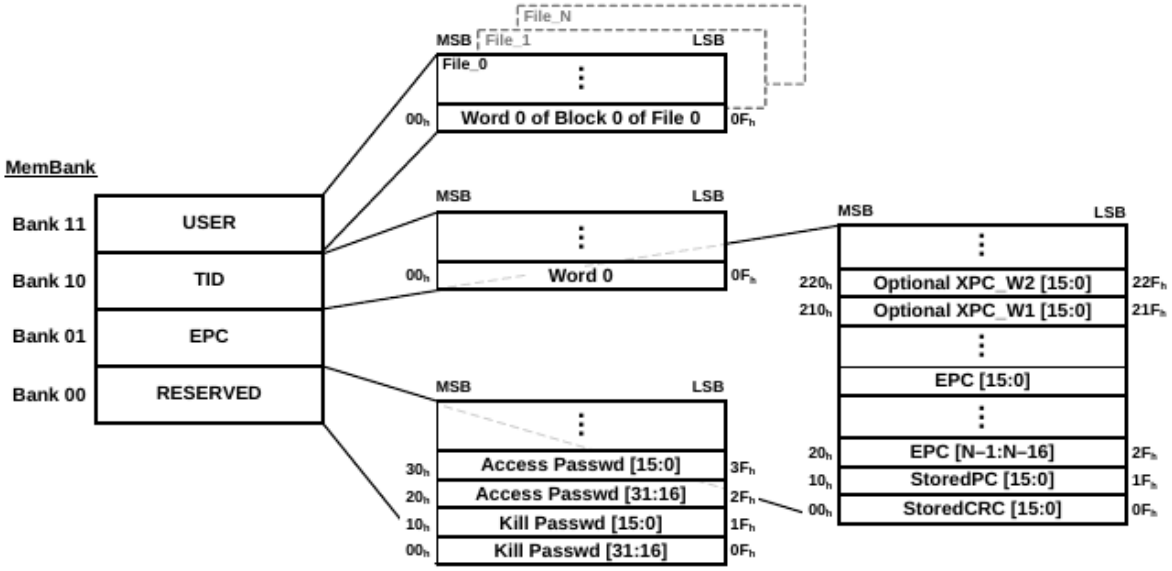

Figura 2.3.: Esquema general de la memoria de un *tag*

2.5.4. Duración de las diferentes secuencias de comandos

Gracias al trabajo desarrollado en Alcaraz y col., 2013a donde se pone de manifiesto este mismo estándar de comunicaciones y protocolo referente

a RFID, extraemos unos datos muy valiosos que nos van a servir para cuantificar en nuestro simulador el tiempo que el *lector* tarda en identificar la población de *tags* a la que se enfrente en cada episodio.

La secuencia de comandos con la que se van a estudiar las duraciones de los slots en nuestras tramas de identificación es la siguiente:

- Successful: se ha identificado correctamente un *tag*.
- Idle: no existen *tags* que identificar en este slot.
- Collision: se ha producido una colisión entre dos o más *tags* en un mismo slot.

Como se puede preveer, las duraciones de estas secuencias serán diferentes. En el caso de una colisión, este tiempo dependerá obviamente de la cantidad de *tags* que hayan colisionado. En los slots del tipo "idle" (ocioso) al no haber *tags* esperando ser identificados, el tiempo será el más corto. Por último, en el caso de un slot con identificación exitosa, el tiempo se ha calculado en función al estudio del paper citado anteriormente.

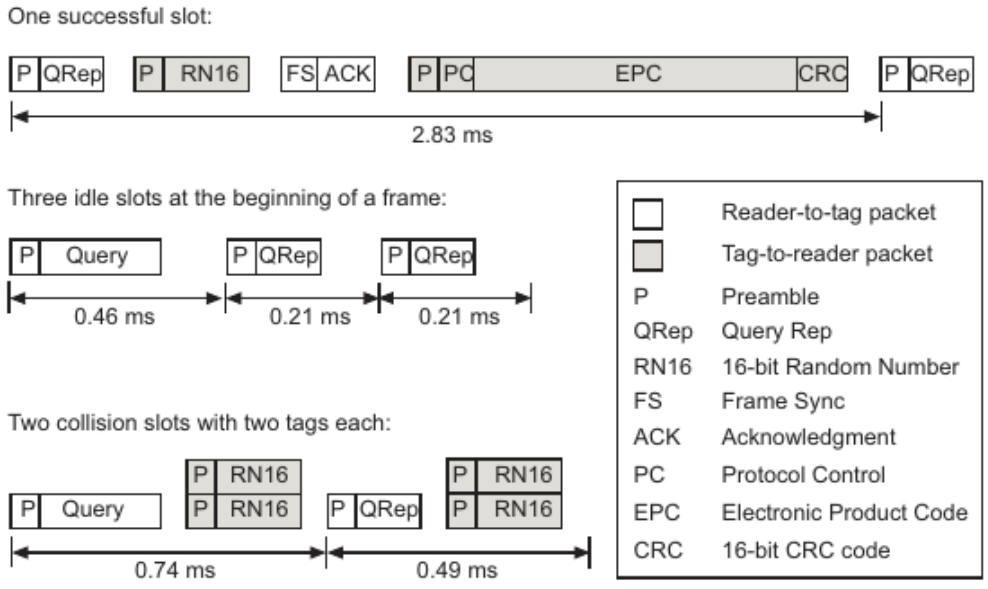

Figura 2.4.: Esquema secuencias "successful", "idle" y "collision" respectivamente

A modo de tener más accesibles los datos concretos de la duraciones buscadas, a continuación se presenta una tabla con los tiempos pertinenetes que se incluirán posteriormente en nuestro simulador en PYTHON:

Secuencia	Duración (ms)
Idle	0.21
Successful	2.83
Collision	0.49
Query	0.25

Figura 2.5.: Duración de las diferentes secuencias de comandos

3. Simulador de tags estáticos

Una vez hemos presentado las especificaciones del protocolo que rige el comportamiento de las comunicaciones RFID, es el momento de aplicarlas a nuestro simulador de *tags*. En él vamos a implementar una interfaz en la cual se nos presenta una población inicial de *tags* que nuestro lector debe identificar.

Para introducir el proceso, diremos que el *lector* poseerá inicialmente una política que, conociendo el número de *tags* por identificar, elegirá la longitud de la trama, es decir, el número de slots que dicha trama poseerá. A continuación, repartirá de forma aleatoria todos los *tags* por los diferentes slots e irá uno a uno intentando identificar el mayor número posible de *tags*. Como se ha presentado anteriormente, los posibles escenarios que se nos pueden presentar por slots son: identificar exitosamente un *tag* (”successful”), que no haya ningún *tag* por identificar en ese slot (”idle”) o que haya dos o más *tags* que respondan al lector y se produzca una colisión (”collision”). Una vez que acaba una trama, empieza una nueva con otra longitud (o con la misma) dependiendo de los *tags* que le queden por identificar. A medida que se van identificando o colisionando, el tiempo de simulación va aumentando, sumándose los tiempos que se presentaron anteriormente en 2.5.

A groso modo, podríamos decir que este es el funcionamiento básico de nuestro simulador, obviamente sin aplicar modificaciones de RL. Podríamos adelantarnos a decir que las mejoras que nos ofrecerán los algoritmos de RL se aplicarán al control de la política que en el párrafo anterior hemos dado por supuesta, aunque de momento no entraremos en ese punto.

En primer lugar y antes de entrar en el desarrollo del código, cabe explicar que se ha llevado a cabo la implementación del simulador en PYTHON a

partir de un código desarrollado en MATLAB que ha sido provisto por los autores de Alcaraz y col., 2013a y que se presenta en el ápendice A.

A continuación iremos presentando distintos puntos sobre la implementación de nuestro simulador a la vez que se introducen ciertas estructuras de programación usadas en el desarrollo del mismo y explicamos de forma detallada su funcionamiento. Esta vez trabajaremos sobre el código en PYTHON que se presenta en el apéndice B.

3.1. Modelado del sistema

En el paper "A Stochastic Shortest Path Model to Minimize the Reading Time in DFSA-Based RFID Systems" (Alcaraz y col., 2013a) se presentan las bases teóricas que sirven de punto de partida para el desarrollo del simulador y que explican como es posible minimizar el tiempo de identificación empleando la formulación de "Stochastic Shortest Path (SSP)". Este método aplicado a este problema mejora notablemente los planteamientos anteriores.

FSA (Frame Slotted Aloha) fue uno de los primeros intentos de implementación en los estándares de RFID, pero presenta limitaciones debido al tamaño fijo de la trama: dependiendo del número de tags, las tramas muy largas pueden ser ineficientes y las tramas muy cortas experimentan numerosas colisiones, en definitiva, incrementa el tiempo de identificación.

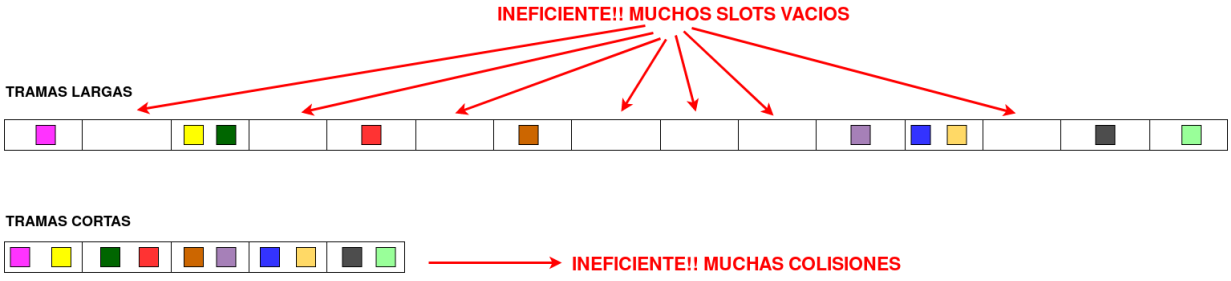

Figura 3.1.: Ineficiencia entre extremos de longitud de trama

Para corregir estas limitaciones, los estándares de RFID permiten el uso de Dinamic FSA (DFSA) donde es posible ajustar de forma dinámica la longitud de las tramas. Posee dos modos de operación, slot-by-slot y frame-by-frame. De momento nos centraremos en el segundo de ellos, pero no descartemos volver sobre nuestros pasos e implementar el primero en el futuro.

El objetivo es seleccionar, al inicio de cada ciclo/episodio, la longitud de trama que se espera que minimice el tiempo de identificación teniendo en cuenta la cantidad de *tags* sin identificar bajo la cobertura de nuestro *lector*. La principal contribución del paper en cuestión al problema, es la formulación "Stochastic Shortest Path (SSP)", que incluye el *efecto captura* (Apéndice C) y asume distintos tiempos por cada time-slot en una trama.

Continuando con el camino por nuestro simulador, es hora de decir que el sistema puede ser modelado con una Cadena de Markov en Tiempo Discreto (DTMC) donde el estado del mismo es el número de *tags* por identificar. La selección del tamaño de las tramas (cantidad de slots) vendrá determinada por un Proceso de Decisión de Markov (MDP) que nos ofrecerá las probabilidades de transición por la DTMC en términos de un parámetro de control. Este parámetro de control Q nos debe servir para calcular el tamaño de trama:

$$l = 2^Q \tag{3.1}$$

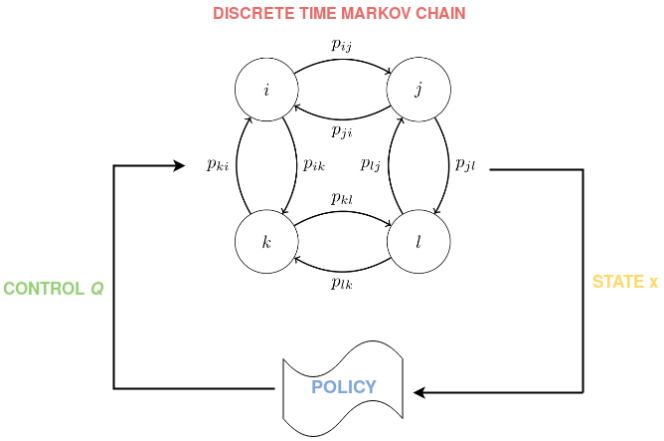

Figura 3.2.: Diagrama esquemático de un MDP

Como detalle a priori sin importancia pero de gran repercusión en nuestro trabajo, cabe destacar que si x es el estado inicial e y es el estado destino, las probabilidades de transición siempre se harán con respecto a estados futuros, nunca se podrá volver sobre nuestros pasos, ya que el estado son los *tags* sin identificar:

$$p_{xy}(l) = 0 \text{ para } y > x \tag{3.2}$$

Finalmente y una vez se ha analizado la formulación SSP para este problema, podemos concluir que la política óptima incluyendo el efecto captura antes mencionado es la siguiente:

$$\text{Política óptima con captura } \mu_c^* \text{ (ratio captura = 10 dB)} \tag{3.3}$$

	μ_C^* (capture)
Q	N **range**
0	$N = 1$
1	$1 < N \leq 2$
2	$2 < N \leq 4$
3	$4 < N \leq 10$
4	$10 < N \leq 21$
5	$21 < N \leq 43$
6	$43 < N \leq 88$
7	$88 < N \leq 100$

Figura 3.3.: Relación entre número de *tags* y parámetro Q

donde N es el número de *tags* por identificar y Q es el parámetro a partir del cual se define la longitud de la trama óptima que minimiza el coste total esperado (en nuestro caso, el tiempo de identificación).

3.2. Link budget

Dado que se ha desarrollado un simulador, es preciso que este se comporte lo más parecido posible a la realidad, es decir, cuantas más especificaciones físicas se incluyan y más cuidado se tenga en la implementación, nuestro simulador se comportará como si de un elemento real se tratara. Esto nos puede ayudar si en el futuro se decide aplicar el algoritmo de identificación a un *lector* de *tags* real en una cadena de producción, por ejemplo.

Cambiando ahora de paper (Alcaraz y col., 2013b), pero no de autores, nos centraremos en uno de sus apéndices que explica de forma precisa como es el enlace entre *lector* y *tags*, incluyendo parámetros físicos, distancia, pérdidas de propagación y demás figuras que afectan sin duda a un enlace inalámbrico de este tipo.

De los tipos de "link budgets" existentes y mencionados en el paper referenciado arriba, nos centraremos en una configuración monoestática, en la cual el *lector* usa la misma antena para la transmisión y la recepción de datos.

En este tipo usaremos dos "link budgets", uno de ellos nos ofrece la cantidad de energía que se obtiene en el circuito de radio frecuencia del *tag* y el otro nos ofrece la cantidad de energía de retrodispersión del *tag* al *lector*. En conclusión, los enlaces que vamos a calcular son los siguientes:

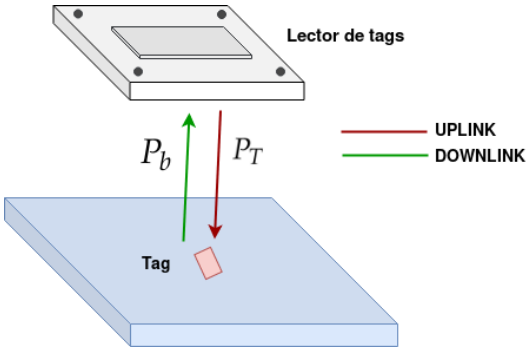

Figura 3.4.: Link budgets con potencias en sentido *lector-tag* y *tag-lector*

En la siguiente lista se muestran los parámetros que se van a emplear para el cálculo de los dos enlaces, cada uno en un sentido de la comunicación entre los dos principales elementos de nuestro escenario:

- P_T potencia transmitida por el lector [W];
- g_T ganancia de la antena del *tag*;
- g_t ganancia de la antena del *lector*;
- λ longitud de onda de la portadora [m];
- X desajuste de polarización;
- τ coeficiente de transmisión;
- r distancia de separación entre *tag* y *lector*;
- θ penalización por ganancia en la antena del *tag*;
- B pérdida por bloqueo de trayecto;
- M factor de modulación;
- h_p^2 ganancia del canal de potencia *lector-tag*;
- h_b^2 ganancia del canal de retrodispersión *tag-lector*;

Finalmente, las expresiones que completan esta explicación y que nos servirán para obtener las potencias que llegan tanto a *tags* como al *lector* en nuestros futuros experimentos con el simulador son:

$$P_t = \frac{P_T g_T g_t \lambda^2 X \tau h_p^2}{(4\pi r)^4 \theta^2 B^2} \tag{3.4}$$

$$P_b = \frac{P_T g_T^2 g_t^2 \lambda^4 X M h_b^2}{(4\pi r)^4 \theta^2 B^2} \tag{3.5}$$

3.3. Evaluación general del código en Python

Como se ha comentado anteriormente, el simulador en PYTHON ha sido desarrollado a partir de otro muy similar en lenguaje MATLAB. Aunque estos dos lenguajes de programación son muy parecidos, la realidad es que existen ciertos procesos y estructuras que no se parecen tanto y que hay que tener muy en cuenta a la hora de trasponer el simulador de uno a otro lenguaje.

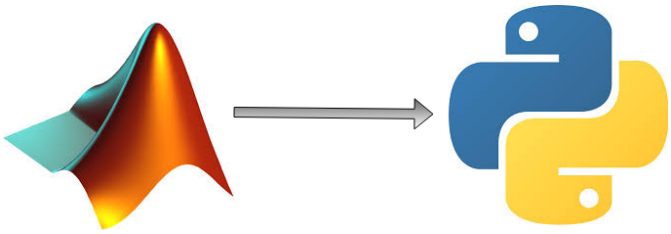

Figura 3.5.: Transición entre lenguajes de programación, MATLAB a PYTHON

En esta sección nos centraremos en las estructuras de programación más significativas que dan vida a nuestro simulador, para acabar con un punto en el que señalaremos algunas de las diferencias entre los lenguajes mencionados que más quebraderos de cabeza nos han generado en el desarrollo.

3.3.1. Reference policy

No nos pararemos mucho en este punto ya que la política de referencia se ha presentado en los puntos anteriores a partir del paper (Alcaraz y col., 2013a) y cuando lleguemos a la parte en la que apliquemos los algoritmos de RL desaparecerá ya que serán los propios algoritmos los que establezcan con su aprendizaje estas política de control. De todas formas se presenta el código ya que esta política corresponde al SSP de referencia y es de suma importancia para entender el funcionamiento del simulador.

Como podemos comprobar se trata de una lista de tamaño el número de *tags* iniciales (*Initial_Population*) donde el índice de la lista se corresponde con los *tags* y el valor que se guarda en esas posiciones correspoonde con el parámetro de control Q.

Listing 3.1: Reference policy en PYTHON

```
referencepolicy = list(range(Initial_Population))

for x in referencepolicy:
    if x == 0:
        referencepolicy[x] = 1
    elif ((0<x)and(x<=2)):
        referencepolicy[x] = 2
    elif ((2<x)and(x<=5)):
        referencepolicy[x] = 3
    elif ((5<x)and(x<=10)):
        referencepolicy[x] = 4
    elif ((10<x)and(x<=21)):
        referencepolicy[x] = 5
    elif ((21<x)and(x<=43)):
        referencepolicy[x] = 6
    elif ((43<x)and(x<=88)):
        referencepolicy[x] = 7
    elif (88<x):
        referencepolicy[x] = 8
```

3.3.2. Trama y reparto aleatorio

Saltando varias líneas triviales que se pueden consultar más en detalle en el apéndice B, llegamos a una parte bastante importante de nuestro simulador en la cual creamos una trama vacía gracias a la polivalencia de PYTHON para crear estructuras. Ésta será rellena de forma aleatoria ayudándonos con la librería importada *random*. A continuación se obtendrán los slot en los que hay algún *tag* y se ordenarán como eventos que se irán consultando en bucles posteriores para comprobar si es posible identificar o se produce alguna colisión.

Listing 3.2: Estructuras de control y reparto de *tags* en PYTHON

```
    #set frame length according to policy
    FrameLength = 2**(referencepolicy[Unident]-1)
    Frame = [ [] for i in range(FrameLength) ]

    SelectedSlotsList = []

    for x in range(Initial_Population):
```

```
        IsId = TagList[x]
        if (IsId == False): #if not identified, selected a slot for next frame
            SelectedSlot = random.randint(0,FrameLength-1) #randomly selected slot
            SelectedSlotsList.append(SelectedSlot)
            Frame[SelectedSlot].append(x)

    SelectedSlotsList = numpy.sort(SelectedSlotsList)
    SelectedSlotsList = numpy.unique(SelectedSlotsList)
    Events = len(SelectedSlotsList)
```

La variable "TagList" es una lista de booleanos que tendrá un false si el *tag* (índice/posición en la lista) está aún sin identificar o que tendrá un true si el *tag* está identificado por el *lector*.

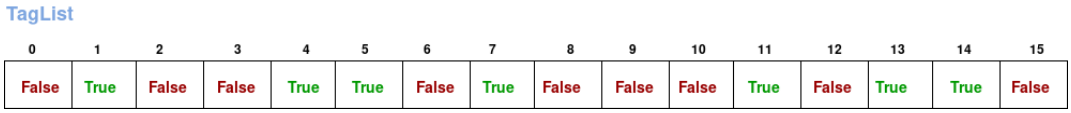

Figura 3.6.: Estructura visual de la variable "TagList" rellena con valores al azar

El siguiente proceso es muy ingenioso y es clave en el desarrollo donde se exprimen las técnicas de programación con tal de facilitarnos la implementación. La clave está en elegir las posiciones (los *tags*) sin identificar, es decir, con un valor false en "TagList". A continuación, se asocia esa posición con un slot de forma aleatoria y al final, cuando se han ubicado todos los *tags*, se lleva a cabo una ordenación que da pie de forma muy sencilla a la siguiente parte de la implementación. Se ilustra ahora un ejemplo sencillo para entender el funcionamiento del código y de los métodos de la librería NUMPY[1]

[1]Numpy es una librería de PYTHON que agrega mayor soporte para vectores y matrices, constituyendo una biblioteca de funciones matemáticas de alto nivel para operar con esos vectores o matrices. Fuente: https://es.wikipedia.org/wiki/NumPy

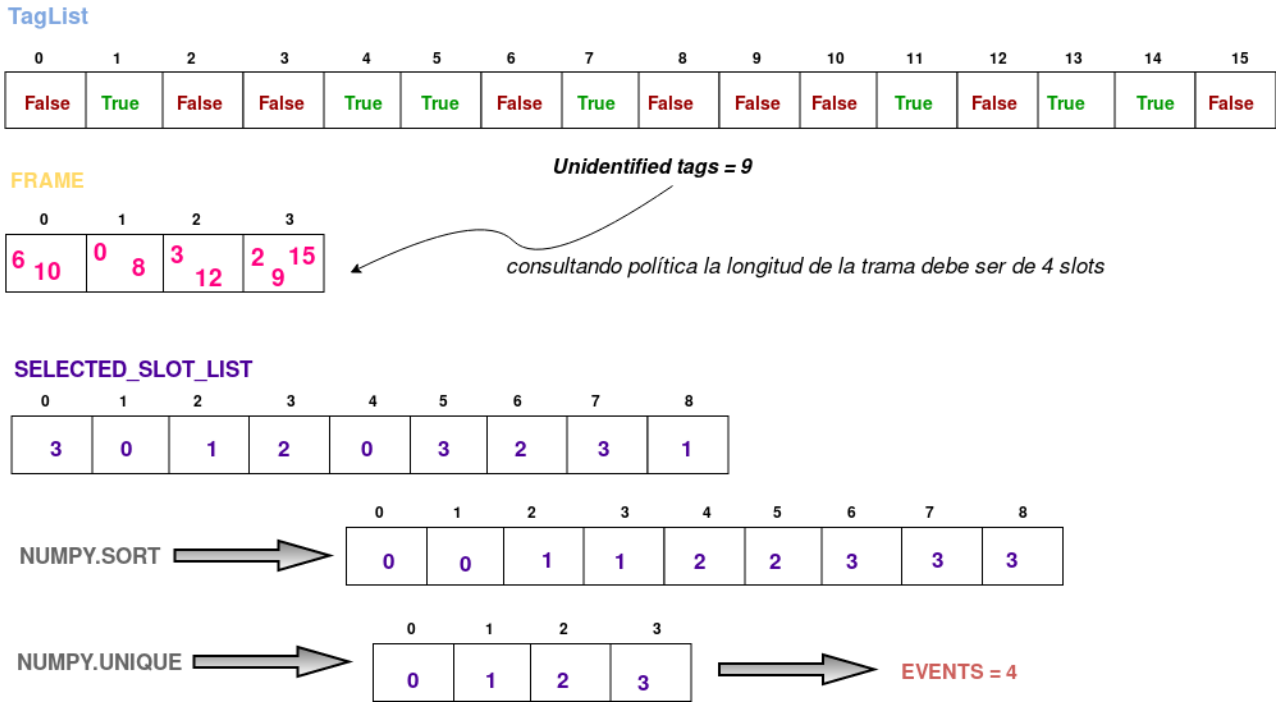

Figura 3.7.: Estructura visual del proceso de ubicación de los *tags* en la trama

3.3.3. Proceso de identificación, colisión o slot ocioso

Tras hacer un reparto aleatorio de los *tags* a través de los diferentes slots de la trama, es el momento de que vayamos iterando por los slots que poseen 1 o más *tags*, es decir, iremos iterando por "events" slots.

En cada uno de ellos se observará cuantos *tags* están compitiendo por ser identificados calculando en primer lugar la potencia con la que llegan al *lector*. En este momento pueden ocurrir tres sucesos:

- o *tags*: como los eventos se cuentan para slot con *tags*, el caso en el que no hay ninguno para identificar se contabiliza el tiempo teniendo en cuenta la diferencia con el slot anterior que si tenía *tags* para identificar, es decir, si en el slot 3 hay *tags* y hasta el 6 no hay más, se contabiliza

el tiempo de los slots 4 y 5, sumando un tiempo "idle" 2.5 por cada uno.

- 1 solo *tag*: al haber un solo *tag* la competición se anula y se identifica sin mayor problema ya que a priori al *lector* solo le llega energía de éste.
- 2 o más *tags*: se lleva a cabo una competición a partir de las potencias con las que "atacan" al *lector*. Como se explica en el apéndice C, se verifica si se cumplen los requisitos para que se produzca el efecto captura y en caso de ser así, el *tag* que llegue con mayor energía será identificado. En caso en que no se satisfaga el requisito, ninguno de los *tags* competidores en ese slot se identificarán y quedarán a la espera de serlo en la próxima ronda de identificación.

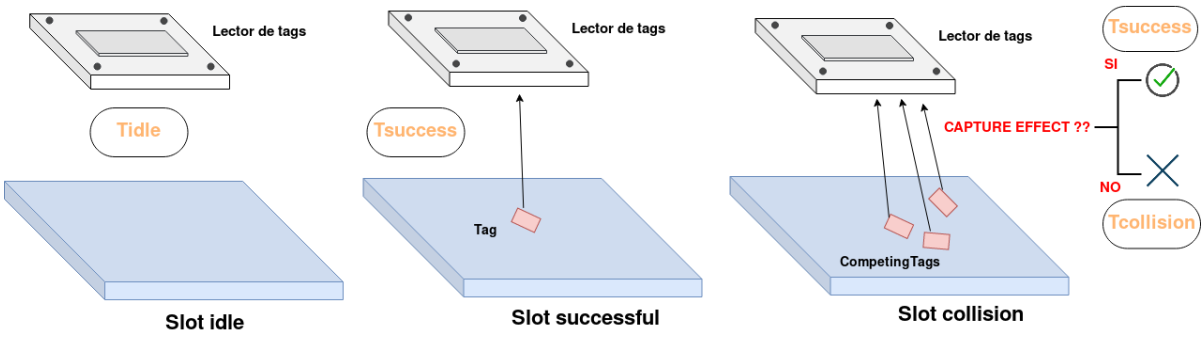

Figura 3.8.: Distintas situaciones por slot

Es importante no olvidar que cada vez que un *tag* se identifica correctamente es necesario actualizar ciertas estructuras. En primer lugar y más importante hay que cambiar el valor en "TagList" pasando la posición del *tag* identificado de false a true. En segundo lugar hay que restar 1 unidad a la variable de *tags* sin identificar *"unidentified"*.

3.3.4. Contabilización temporal y actualización de estructuras

En la figura anterior (3.8) se pueden observar los distintos tiempos que se sumarían al tiempo total de simulación "simTime" para llevar un correcto

recuento. Como todas las tramas empiezan con un mensaje de *"Query"*, este valor siempre se sumará al comienzo de cada trama. Después y en función de como estén repartidos los *tags* por los slots se sumarán tiempos *"Tidle"*, *"Tsuccess" o "Tcollision"*. Los tiempos dentro de cada trama se albergan en la variable *"ElapsedTime"* y finalmente cuando la trama ha terminado, este tiempo transcurrido se suma al tiempo de simulación total, que ya contiene "Tquery" mencionado anteriormente.

Una vez que una trama ha terminado es muy probable que aún no se hayan identificado todos los *tags*, por lo tanto, el valor del tiempo de simulación "simTime" se conserva en una posición de la variable "Jestimated". Esta variable se interpreta de la siguiente manera:

Jestimated[n]: tiempo total desde que hay n *tags* sin identificar hasta que quedan o sin identificar.

Por lo tanto, cuando se han identificado todos los *tags* se guarda en la última posición (que coincide con *"Initial_Population"*) el valor de "simTime", que es el tiempo total de simulación, es decir, el tiempo que el *lector* tarda en identificar todos los *tags* de la población bajo su cobertura.

Este es el resultado de una ejecución de nuestro simulador, donde se devuelve por pantalla la variable "Jestimated":

Listing 3.3: Jestimated

```
[ 0.     0.     0.    11.53   0.    17.61   0.     0.     0.     0.    32.95
0.
  0.     0.     0.     0.     0.     0.     0.     0.     0.     0.     0.
0.
  0.     0.     0.     0.     0.    90.26]
```

Para este ejemplo con 30 *tags* de población la interpretación es sencilla, se tardan 90.26 ms en identificar toda la población, 32.95 ms en identificar 11 *tags*, 17.61 ms en identificar 6 y 11.53 en identificar 4 *tags*.

3.4. Conclusiones sobre el simulador

Aunque este no será ni mucho menos nuestro simulador final, vamos a hacer un alto en el camino para analizar de forma superficial los resultados que nos ofrece este simulador, tal y como se expuso en Alcaraz y col., 2013a.

Se ha examinado el problema de determinar el valor del parámetro Q con el fin de minimizar el tiempo esperado de identificación en una población de *tags* sin identificar en un sistema RFID basado en DFSA al que además se ha incluido efecto captura para precisar y hacer más real su comportamiento. Se ha modelado el problema como un SSP en el que se tienen en cuenta las diferencias temporales entre los distintos tipos de slots (como ocurre en la realidad) y se ha comprobado que los resultados con la política óptima (*optimal policy*) mejoran los resultados de las pruebas pasadas en las que se trataba de maximizar el *throughtput* para minimizar el tiempo de identificación.

4. Reinforcement Learning

[1] Cuando hablamos sobre el aprendizaje de la forma más primaria y general que se nos puede ocurrir, seguramente imaginemos a un niño en sus primeros años de vida interaccionando con absolutamente todo lo que le rodea, creando todo tipo de conexiones con las vivencias que va experimentando. Estas conexiones ofrecen cada vez más información sobre el medio que rodea al aprendiz ya que hacer una u otra cosa hará que ese medio responda de una u otra forma.[2]

Figura 4.1.: Situaciones negativas de aprendizaje en niños

[1] Fuente principal: Sutton y Barto, 2018
[2] aprender desde la interacción.

Como es de esperar, en nuestro caso este proceso de aprendizaje desde la interacción será enfocado desde un punto de vista computacional, donde ciertas máquinas son capaces de resolver problemas complejos que son evaluados con análisis mateático y experimentación.

4.1. Inteligencia artificial, unas pinceladas

Podríamos escribir miles de páginas sobre este concepto tan de moda hoy en día, pero por el momento solo vamos a introducir ciertos términos básicos de la inteligencia artificial y situaremos nuestro campo, el "reinforcement learning"[3], dentro de la misma.

Muchas son las definiciones que podemos encontrar acerca de la inteligencia artificial y es que si es cierto que es un campo muy amplio que alberga numerosos subcampos, por lo tanto, cuando hablemos de la inteligencia artificial necesitaremos algo más de información.

La *inteligencia artificial* es un campo genérico que alberga a programas o máquinas que exhiban inteligencia. El concepto de "inteligencia" aquí tambíen puede dar mucho que hablar, pero en resumen un proceso o resultado inteligente se puede relacionar con las funciones cognitivas humanas como son percibir, razonar, aprender o resolver problemas.

Figura 4.2.: Partida de *AlphaGo* - campeón del mundo vs. algoritmo inteligente de *Google DeepMind*

[3]aprendizaje reforzado, en español.

Para aclarar los conceptos que se van a presentar a continuación, se añade un diagrama de Venn para ubicar correctamente los diferentes subcampos que forman el "Machine Learning" (ML), uno de los principales dentro de la inteligencia artificial (AI) y con el que trabajaremos en este proyecto.

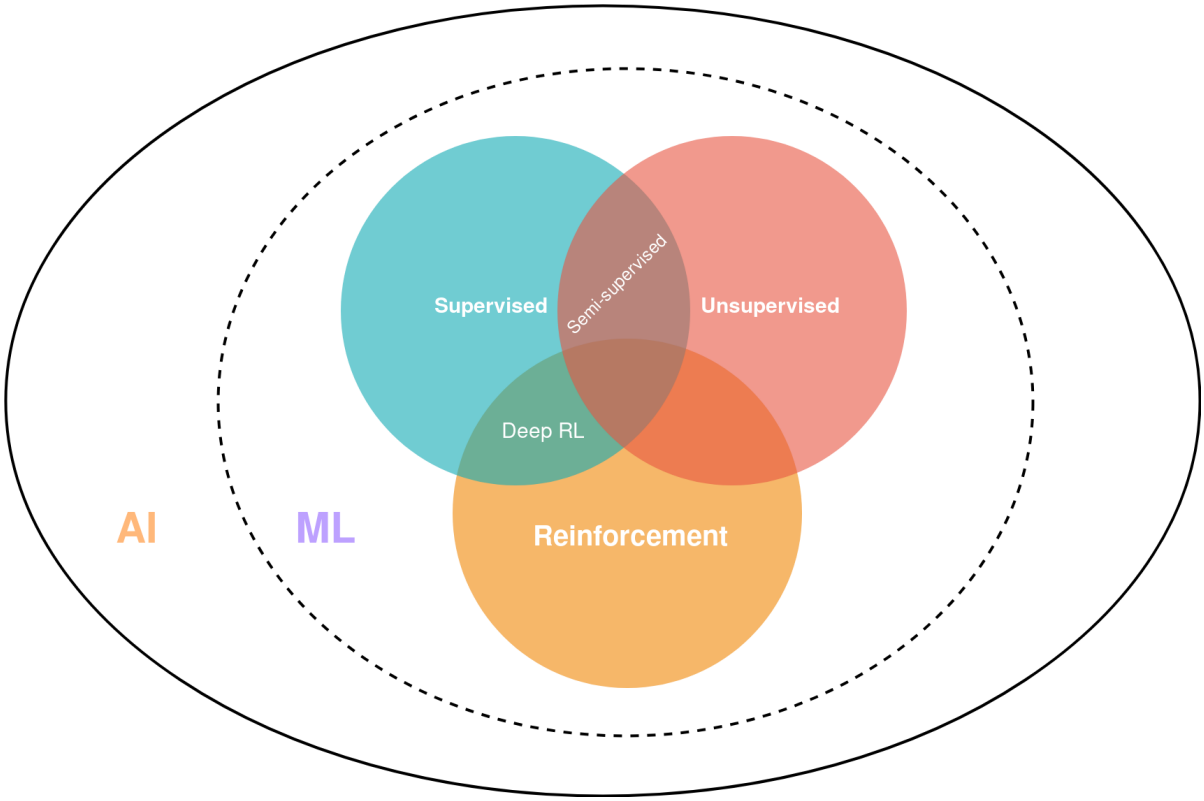

Figura 4.3.: Diagrama de Venn para ubicar el *reinforcement learning* en el mundo *AI*

Como vemos, dentro del "Machine Learning" (ML) existen numerosos campos, muchos de ellos subcampos o intersecciones entre tres principales, el aprendizaje supervisado, no-supervisado y reforzado. Nos centraremos en el último de estos el cual es el que mejor se adapta a nuestra finalidad.

4.2. Conceptos clave en Reinforcement Learning

Como hasta el momento hemos estado hablando de conceptos que rodean al *reinforcement learning* pero no hemos entrado a hablar de él de lleno, antes de hacerlo daremos una breve definición de lo que es este campo del *machine learning* y después entraremos de lleno a desgranarlo.

El *reinforcement learning* o aprendizaje reforzado es el aprendizaje acerca de lo que hacer (como relacionar ciertas situaciones con acciones particulares) con el fin de maximizar una recompensa. El aprendiz (el cual más adelante introduciremos como "agente") debe descubrir que acciones le ofrecen mayores recompensas y eso lo debe comprobar probando esas acciones.

Una definición más coloquial (obtenida de OpenAI-SpinningUp, 2019) podría enfocarse diciendo que el aprendizaje reforzado es el estudio de como un agente aprende por intento-error, es decir, como las recompensas positivas o negativas (castigos-penalizaciones) hacen que motive a repetir ciertas acciones en el futuro si le dieron buena recompensa o a no llevarlas más a cabo si le penalizaron.

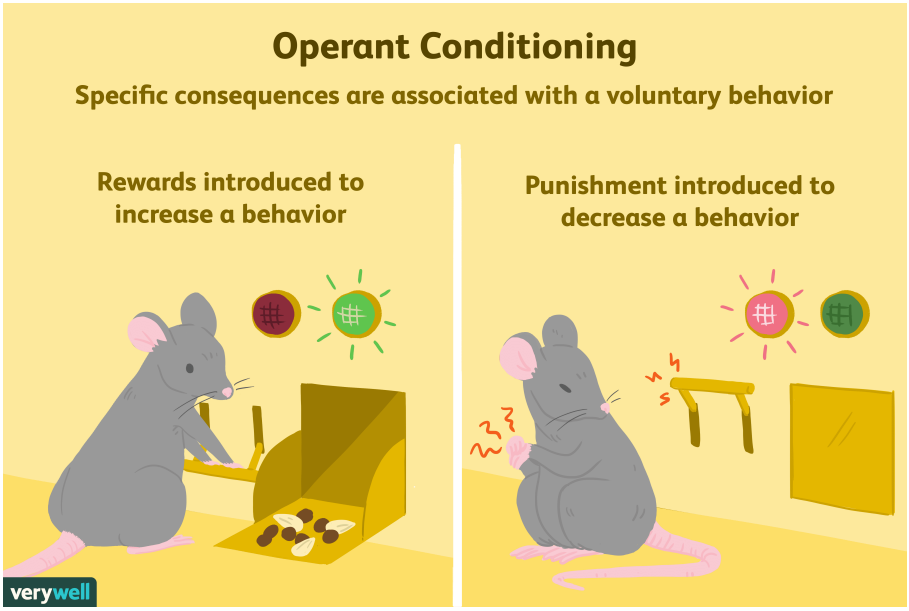

Figura 4.4.: Ejemplo animado recompensa-penalización en función de la acción ejecutada

Más adelante hablaremos de las recompensas, pero éstas no tienen porque ser positivas y las penalizaciones negativas, pueden haber recompensas mejores o peores sin necesidad de que haya una penalización negativa, dependerá del contexto del problema y la forma en la que se implemente la solución.

A continuación vamos a incluir en nuestro proyecto la formalización de los procesos de decisión de *Markov* (*MDPs*), los cuales una vez estén bien planteados trataremos de resolver implementando y aplicando correctamente los algoritmos pertinentes.

Un *MDP* es una forma matemática de representar un problema de *reinforcement learning* con el fin de poder aplicar conceptos matemáticos para resolverlos.

4.2.1. Interfaz Agent-Environment

- *Agent*: aprendiz o elemento encargado de tomar decisiones (ejecutar acciones) a través de una política.
- *Environment*: es el marco con el que el *agent* interactúa, es decir, conforma todo los que está fuera del *agent*.

El *agent* selecciona una acción y el *environment* responde a esta acción representado una nueva situación (*state*) al *agent*. A la misma vez, el *environment* devuelve una recompensa (*reward*), que es un valor numérico que el *agent* debe tratar de maximizar a través de la correcta elección de acciones.

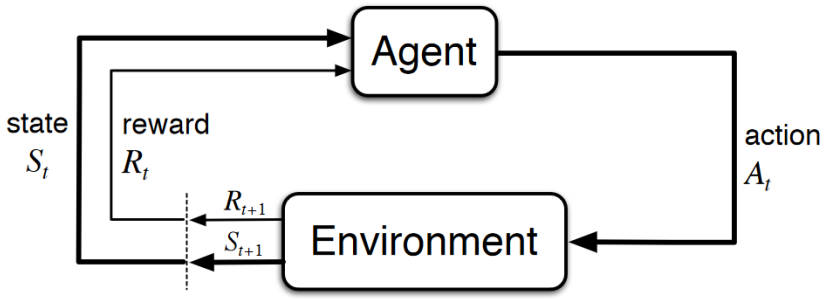

Figura 4.5.: Interacción entre *agent* y *environment* en un *MDP*

En cada paso de esta interacción, el *agent* observa el estado (*state*) del *environment* y decide qué acción tomar. El *agent* recibe una recompensa (*reward*) que nos dirá lo bueno o malo que es el *state* actual al que se ha llegado tomando dicha acción. Tras esta acción, el *environment* permuta a un nuevo *state* que el *agent* volverá a observar para elegir una nueva acción.

Al tratarse de un *MDP* discreto, el tiempo está dividido en *time-steps*, $t = 0, 1, 2, 3...$ Para entender mejor la figura de arriba, hay que saber que en cada *time-step* t:

1. El *agent* recibe una observación del *state* del *environment*, $S_t \in \mathcal{S}$.
2. Basándose en dicho *state*, el *agent* elegirá una acción, $A_t \in \mathcal{A}(s)$.

En el próximo *time-step* $t+1$:

1. Como consecuencia de la acción anterior, el *agent* recibirá un *reward* del *environment*, $R_{t+1} \in \mathcal{R} \subset \mathbb{R}$.
2. Además un nuevo *state* del *environment* será presentado al *agent*, S_{t+1}.

En el anterior proceso se han presentado ciertos parámetros y conjuntos, pero no se han especificado con claridad:

- Un *state* s es una completa descripción del estado del mundo en el que se está trabajando, el *environment*. Dicho esto, cabe destacar que en función de como se presente el *state* al *agent*, el *environment* puede ser observado totalmente o parcialmente[4]. El conjunto de todos los *states* posibles del *environment* se designa con $\mathcal{S}$.
- El espacio de acciones ejecutadas por el *agent* y aceptadas por el *environment* se recoge en $\mathcal{A}$.
- El rango de *rewards* que el *environment* puede devolver al *agent* se designa con $\mathcal{R}$ y suele estar formado por una función que depende del par *state-action* (S_t, A_t) del *time-step* anterior:

$$f(S_t, A_t) = R_{t+1} \tag{4.1}$$

[4]Del inglés *fully observed* cuando el *agent* es capaz de observar el estado completo del *environment* o bien *partially observed* cuando se obtiene una observación parcial/no completa del mismo.

4.2.2. Trayectoria

Dado que el proceso explicado en el punto anterior se lleva a cabo de forma secuencial una y otra vez, podemos presentar el término de trayectoria (*trajectory*) como el proceso iterativo de selección de una acción a partir de un estado para hacer una transición a otro estado y recibir a cambio una recompensa.

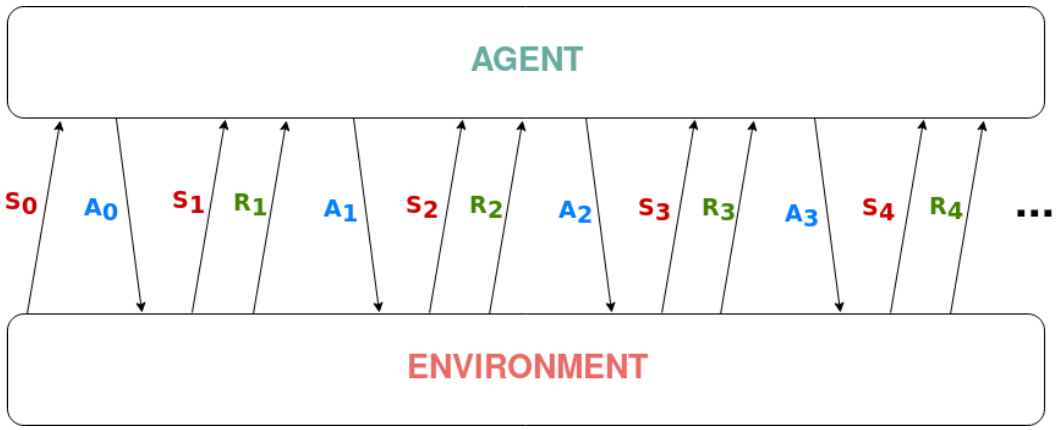

Figura 4.6.: Representación de la trayectoria entre el *agent* y el *environment*.

4.2.3. Probabilidades de transición

Todo proceso de decisión de Markov (*MDP*) posee unas probabilidades de transición que relacionan los diferentes *states* del sistema. En las próximas líneas vamos a explicar como se abordan y especifican estas probabilidades de forma precisa.

Para todo $s' \in \mathcal{S}$, $s \in \mathcal{S}$, $r \in \mathcal{R}$ y $a \in \mathcal{A}(s)$, definimos la probabilidad de transición al *state* s' con recompensa r tomando la acción a en el *state* s como:

$$p(s', r \mid s, a) = Pr\{S_t = s', R_t = r \mid S_{t-1} = s, A_{t-1} = a\} \tag{4.2}$$

donde no debemos olvidar que

$$\sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} p(s', r \mid s, a) = 1, \text{ para todo } s \in \mathcal{S}, a \in \mathcal{A}(s) \tag{4.3}$$

Con estas probabilidades el *MDP* y su dinámica (como interaccionan entre sí los estados) quedarían definidos. Como se puede observar, los posibles valores de S_t y R_t dependen solo de los valores inmediatamente anteriores S_{t-1} y A_{t-1}.

De esta forma, cada *state* debe contener información de las interacciones pasadas entre *agent* y *environment* o al menos una forma de seguimiento razonable en función del contexto del sistema.

4.2.4. Objetivo y recompensas (rewards)

En RL, el objetivo final del *agent* está regulado por la señal de recompensa (*reward*) que recibe por parte del *environment* y consiste en maximizar la cantidad total de *reward* que recibe, no el *reward* inmediato.

Por ejemplo, si nos centramos en el juego del ajedrez, si por cada pieza arrebatada al contrincante nos dan +1 de *reward*, pero por ganar la partida nos dan, por ejemplo +100, entonces podríamos pensar y jugar de tal forma que quitáramos muchas piezas al otro jugador, descuidando el objetivo real que es ganar la partida.

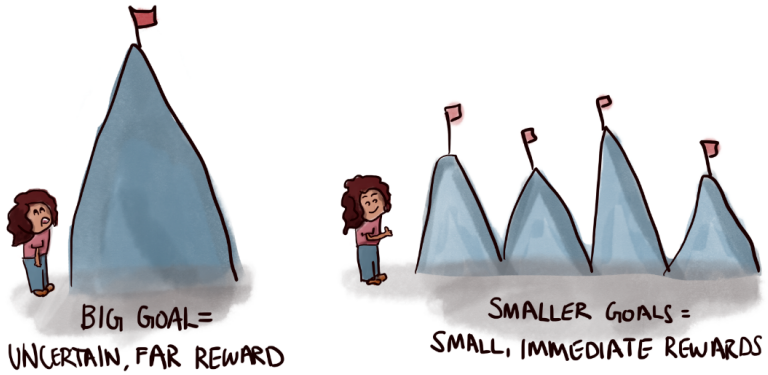

Figura 4.7.: Recompensas inmediatas vs recompensas a largo plazo.

De alguna forma esta idea se puede extrapolar con el *reinforcement learning* y es que el *reward* que nos aporta el *environment* nos debe ayudar a encontrar el camino de lo que queremos alcanzar y no el como queremos hacerlo.

Tras esta explicación informal sobre el uso de las recompensas, debemos formalizar matemáticamente, como se ha hecho con el *MDP*, el *reward* que el *agent* va obteniendo paso a paso (*step-by-step*).

Como el *reward* que pretendemos maximizar depende en cierto modo de los *rewards* a largo plazo (obtenidos en el futuro), podríamos hacer una primera aproximacón de la recompensa obtenida en una trayectoria (4.2.2) G_t como:

$$G_t = R_{t+1} + R_{t+2} + R_{t+3} + R_{t+4} + ... + R_T \tag{4.4}$$

siendo T el *time-step* final del episodio.

Este concepto de episodio se introdujo al hablar del "simulador de *tags* estáticos" y está referido a la idea de que la interacción entre *agent* y *environment* posee subsecuencias que terminan en un *state* final en el *time-step* T. Hay que destacar que los episodios son independientes entre sí y que suelen empezar en algún estado estándar o aleatorio, aunque dependerá del contexto del sistema.

Si continuamos precisando más en la formalización matemática del *reward*, es el momento de presentar un parámetro llamado "discount rate" $\gamma \sim (0, 1)$[5]. Gracias a él podemos restar importancia en el presente a las recompensas que se esperan recibir en el futuro. Así, actualizamos el concepto anterior de *expected return* a *discounted return* G_t como:

$$\begin{aligned} G_t &= R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4} + ... \\ &= R_{t+1} + \gamma(R_{t+2} + \gamma R_{t+3} + \gamma^2 R_{t+4} + ...) \\ &= R_{t+1} + \gamma G_{t+1} \end{aligned} \tag{4.5}$$

En nuestro caso, este concepto de *discounted return* no será implementado ya que nuestro *environment* posee en cada episodio un estado de terminación y no será preciso calcular como se descuenta la importancia de las recompensas que se esperan recibir.

[5]un valor cercano a 0 supondrá maximizar las recompensas más cercanas y un valor cercano a 1 tendrá más en cuenta las recompensas futuras, siendo más previsor.

Antes de cerrar este subpunto estaría bien indicar que aunque de momento solo se estén presentando estos conceptos algo abstractos, cuando sea el momento de desarrollar los algoritmos pertinentes basados en RL, volverán a aparecer y se verá como están aplicados nuestro problema real con el simulador de *tags*.

4.2.5. Políticas y funciones valor (policies y value functions)

- *Policy* es una función que mapea para un estado dado la probabilidad de elegir cada posible acción dentro de un estado dado. [π]
- *Value functions* nos dice en términos probabilisticos como de bueno es cada estado cuando se sigue una determinada política.

Por otro lado también se presenta el concepto de *Q-values*, valores que serán usados más adelante con los algoritmos tabulares:

- *Q-values* estiman como de bueno (en términos de la recompensa esperada) es para un *agent* llevar a cabo cierta acción dada en un estado dado. Por supuesto, las recompensas (*rewards*) que se esperan recibir dependen de qué acciones se toman y en qué estados se hace.

Una vez presentados estos dos conceptos, los relacionaremos matemáticamente para poder comprender más adelante como buscar los valores y política óptima adaptandolo todo a los *MDP*.

En primer lugar, presentamos el *value function* de un *state s* bajo una política de actuación π que nos devuelve la *expected return* cuando se empieza en dicho *state s* y se continúa con la política π a lo largo de los sucesivos *states*.

$$v_\pi(\mathrm{s}) = \mathbb{E}_\pi[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s] \tag{4.6}$$

Por otro lado, existe otro valor que nos devuelve la *expected return* empezando en un *state s*, tomando una acción *a* y siguiendo con una política π:

$$q_\pi(\text{s, a}) = \mathbb{E}_\pi[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s, A_t = a] \tag{4.7}$$

Ahora es el momento de volver al primer párrafo de la introducción en el que hablábamos de la magia de las matemáticas, de como jugando correctamente con ellas podríamos dar vida a cosas que a priori solo acatan ordenes. Y es que en realidad seguirán haciendo eso, ejecutar órdenes, pero aprendiendo a la misma vez. Hagamos magia.

4.3. Ecuación de Bellman

Es el momento de empezar a presentar y desarrollar alguna solución al problema que venimos introduciendo hace ya varios capítulos. Resolver un problema de *reinforcement learning* significa encontrar una política que alcance la mayor recompensa posible a lo largo del tiempo, como matemáticamente se ha introducido anteriormente.

Podemos definir que una polítca π es mejor que otra política π' siempre que para todos los *states* la *expected return* sea mayor que la que ofrece la otra política. Matemáticamente:

$$\pi \geq \pi' \text{si y solo si } v_\pi(\text{s}) \geq v_{\pi'}(\text{s}) \text{ para todos los s} \in S \tag{4.8}$$

Gracias a esta expresión podemos decir que existe una política π que es mejor que el resto de políticas, la *política óptima* π_*.

Gracias a esta política óptima π_* podemos definir el mejor *expected return* posible para todos los *states* del sistema (*MDP*) como la *optimal state-value function*:

$$v_*(\text{s}) = \max_\pi v_\pi(\text{s}) \text{ para todos los s} \in S \tag{4.9}$$

De la misma forma definimos la *optimal action-state function*, que nos ofrece la mejor *expected return* alcanzable para cualquier política π para cualquier

posible par *state-action*:

$$q_*(\text{s, a}) = \max_{\pi} q_\pi(\text{s, a}) \text{ para todos los s} \in S \text{ y para todos los a} \in A(\text{s}) \tag{4.10}$$

Simplemente hemos reescrito las ecuaciones del punto anterior, pero aplicando a cada una de ellas la acción de la política óptima, donde se devuelve la mayor recompensa posible, atendiendo a la ejecución de las acciones que la mejor política (*optimal policy*) nos rige.

Como esta última ecuación 4.10 nos devuelve la *expected return* de tomar una acción a en un *state s* y continuar con la *optimal policy*, podríamos cambiar esta ecuación y escribirla en términos de v_*:

$$q_*(\text{s, a}) = \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1}) \mid S_t = s, A_t = a] \tag{4.11}$$

En este punto y aunque el paso matemático parezca un poco sacado de la manga, si abstraemos los resultados y relaciones que estamos exponiendo, podemos decir que "el valor obtenido por estar en un *state* s y llevar a cabo la política óptima π_* debe ser igual que la *expected return* de elegir la mejor acción a en ese mismo *state s*".

Si lo pensamos fríamente este enunciado tiene todo el sentido del mundo ya que una política simplemente le va a decir al *agent* qué acción debe tomar; si esa política es la óptima, pues estaremos eligiendo la mejor acción posible de entre todas ellas. Matemáticamente:

$$\begin{aligned} v_*(\text{s}) &= \max_{a \in A(\text{s})} q_{\pi *}(\text{s, a}) \\ &= \max_{a} \mathbb{E}_{\pi_*}[R_{t+1} + \gamma G_{t+1} \mid S_t = s, A_t = a] \\ &= \max_{a} \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1}) \mid S_t = s, A_t = a] \\ &= \max_{a} \sum_{s', r} p(s', r \mid s, a)[r + \gamma v_*(s')] \end{aligned} \tag{4.12}$$

donde las dos últimas ecuaciones se corresponden con dos formas de representar la ecuación óptima de Bellman (*Bellman optimality equation para* v_*).

En el caso de la ecuación óptima de Bellman para q_*:

$$\begin{aligned} q_*(\text{s, a}) &= \mathbb{E}[R_{t+1} + \gamma \max_{a'} q_*(S_{t+1}, a') \mid S_t = s, A_t = a] \\ &= \sum_{s',r} p(s', r \mid s, a)[r + \gamma \max_{a'} q_*(s', a')] \end{aligned} \tag{4.13}$$

Para entender mejor como se llevan a cabo las acciones y estados futuros se adjunta en la siguiente figura un diagrama auxiliar:

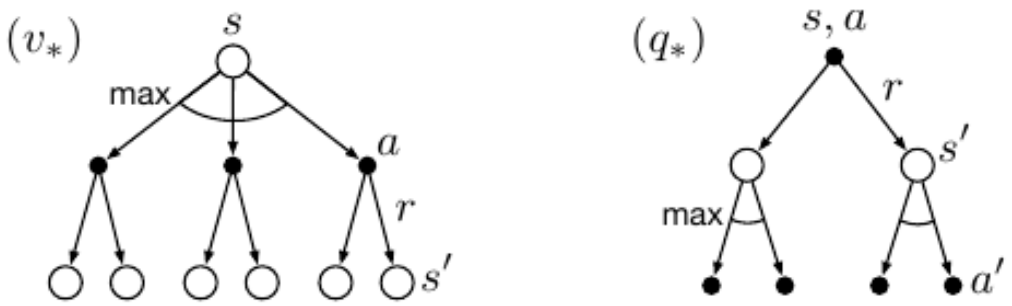

Figura 4.8.: Diagramas auxiliares para entender las ecuaciones anteriores

En el caso de la *Bellman optimality equation* (v_*) elegimos la acción que maximiza el *reward* y nos lleva del *state s* al *state s'*.

En el caso de la *Bellman optimality equation* (q_*) elegimos el mayor *reward* que estará asociado a una política específica (que elige una acción a específica dentro de un *state s* específico) y después se elegirá la acción a' que ofrece mayor recompensa (*expected return*).

Aunque pueda parecer un poco complicado de seguir, solo hace falta entender la ecuación siguiendo las flechas y pasos que se llevan a cabo en los diagramas.

Aún no vemos la magia, pero ya tenemos todo preparado para empezar a trabajar con nuestro simulador, una vez que desarrollemos los algoritmos basándonos en estas ideas, se podrá entrenar y empezarán a aparecer los resultados.

5. OpenAI-Gym, adaptación del simulador

Hasta el momento hemos explicado el protocolo de RFID implantado en la actualidad, el desarrollo de un simulador de *tags* estáticos con PYTHON y las conceptos teóricos más importantes que abarca el *reinforcement learning*. Es el momento de empezar a unir los conceptos de RL con nuestro simulador para entrenarlo con los algoritmos que más adelante se desarrollarán.

5.1. ¿Por qué esta plataforma?

La versión actual que tenemos del simulador de *tags* no es válida para la aplicación de algoritmos de *reinforcement learning*, por lo que va a ser necesaria una adaptación.

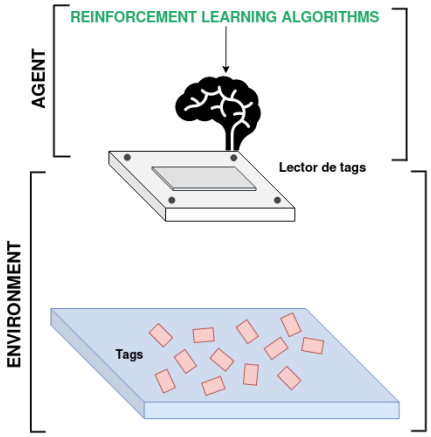

Figura 5.1.: *Agent y environment* en nuestro problema

Quizá algún lector de este proyecto no se haya dado cuenta aún, pero los dos elementos principales de un problema de *reinforcement learning* como son el *agent* y el *environment*, en nuestro caso serán: los algoritmos de RL (como si fueran el cerebro del *lector de tags*) y el simulador de *tags* propiamente dicho, respectivamente (ver 5.1).

5.1.1. OpenAI

Figura 5.2.: Logo *OpenAI*

[1]Es una compañia de investigación en el campo de la inteligencia artificial (*AI*) sin ánimo de lucro que tiene como objetivo principal promover y desarrollar inteligencia artificial de tal manera que beneficie a la humanidad en su conjunto[2].

Existen numerosos proyectos que esta compañia desarrolla y que sirven de apoyo fundamental en los estudios de varios campos de la inteligencia artificial. En nuestro caso y al estar interesados en el *machine learning*, nos centraremos en una de sus plataforma más conocidas, *OpenAI-Gym*.

5.1.2. Gym

Figura 5.3.: Logo *OpenAI-Gym*

[1]OpenAI, 2019
[2]Fuente: https://es.wikipedia.org/wiki/OpenAI

[3]Es una plataforma dentro de la compañia *OpenAI* que sirve como herramienta para el desarrollo y comparación de algoritmos de *reinforcement learning*.

Vista la definición que la compañia le da a su propia plataforma, es el lugar ideal al que acudir, ya que cumple con todas las espectativas de nuestros propósitos en este proyecto.

Aunque nuestra finalidad será la de registrar un nuevo *environment* desarrollado por nosotros en la plataforma, ésta ya ofrece varios con los que poder trabajar y observar la potencia del RL. Accediendo a la documentación oficial (OpenAI-Gym, 2019) y (Kansal y Martin, 2018) podemos observar la definición de varios de estos entornos con un video siendo entrenado; de igual modo se añade un enlace a su código fuente en *GitHub*.

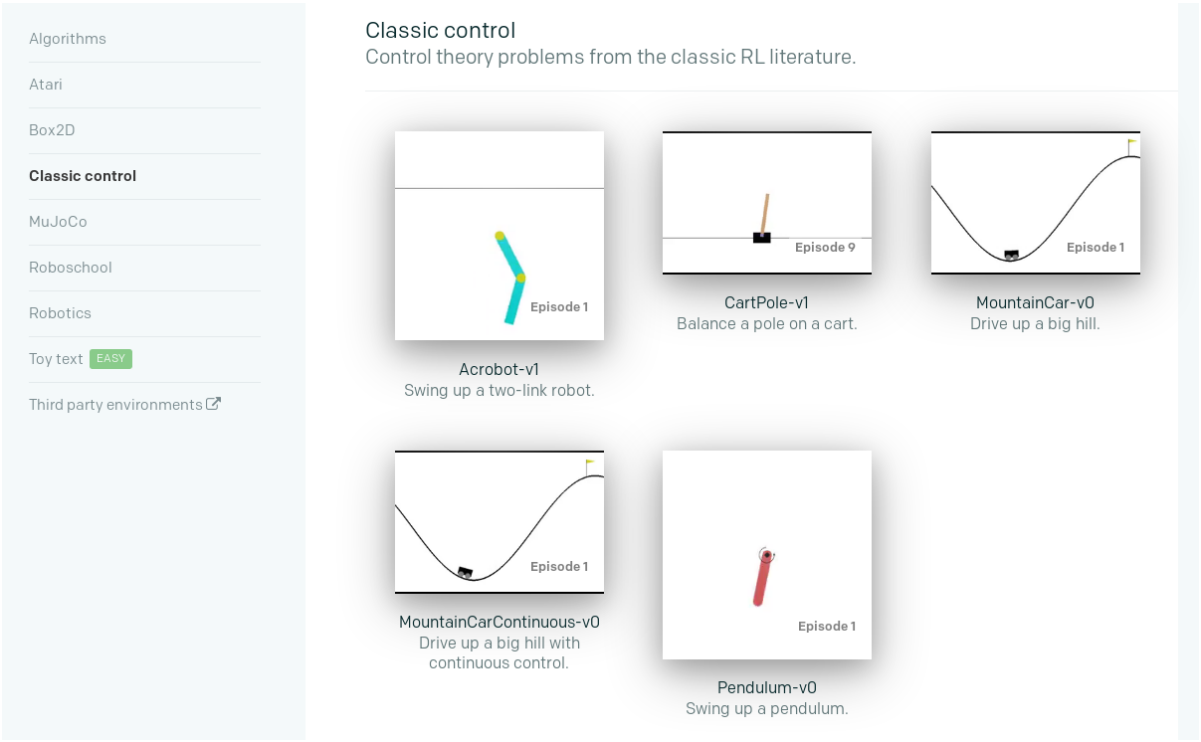

Figura 5.4.: Captura de la web de *OpenAI-Gym* con sus *environment* por defecto

[3]OpenAI-Gym, 2019

Por lo tanto, a la pregunta de esta sección seguiremos respondiendo en los siguientes puntos cuando entremos de lleno en la configuración y especificaciones que esta plataforma nos ofrece, dándonos cuenta de que nos viene como anillo al dedo para nuestro sistema de lector de *tags* inteligente.

5.2. Instalación y puesta en marcha

Tanto en la web de *OpenAI-Gym* (OpenAI-Gym, 2019) (Sonawane, 2018) como en el perfil oficial en *GitHub* ((GitHub), 2019) se puede comprobar como la instalación de la herramienta es a priori muy sencilla, aunque si es cierto que en función del sistema operativo que se tenga algunas cosas cambian.

Dada la necesidad de un editor y ejecutor de código en PYTHON, fue recomendado el uso de *Anaconda* (Anaconda, 2019) que es una distribución libre y abierta de los lenguajes PYTHON y R, la cual se usa en varios campos de *data science* y *machine learning*.

Figura 5.5.: Logos de la distribución *Anaconda* y el programa *Spyder*

En ella se alberga un editor de código llamado *Spyder*, el cual posee una consola en python donde es posible cargar los scripts y ficheros escritos en el lenguaje de programación, haciendo más sencillo llevar a cabo las ejecuciones que serán necesarias.

Mientras que *Anaconda* nos ofrece todos los paquetes y librerías de PYTHON, *OpenAI-Gym* nos aporta las librerías y métodos necesarios para la creación de *environments* y algoritmos (*agents*) con el fin de poner en práctica los conceptos y filosofía que rige el *reinforcement learning*.

Como la plataforma ya contiene varios *environments* por defecto como se ha comentado anteriormente, en sus ficheros de instalación contiene un fichero (*core.py*) en código PYTHON que posee la estructura general que deben tener los *environments* de *OpenAI-Gym*.

A continuación, vamos a ir paso a paso viendo las distintas funciones que requiere *OpenAI-Gym* y como las hemos aplicado en nuestro caso al simulador de *tags* para convertirlo en un *environment* preparado para ser entrenado con algoritmos RL.

5.2.1. Action_space y observation_space

Uno de los primeros requisitos que *OpenAI-Gym* establece es la creación de un espacion de acciones (*action_space*) y un espacio de observación (*observation_space*), los cuales describen a groso modo todas las posibles acciones y observaciones que el *environment* en desarrollo puede soportar.

Para su definición se emplea una librería específica de *OpenAI-Gym*, `gym.spaces`, el cual nos permite crear distintos tipos de datos:

`gym.spaces.Discrete`: un *environment* que use este tipo de datos contendrá n puntos discretos, por ejemplo, `Discrete(10)` nos aporta un mapeo de enteros en [0, 9].
`gym.spaces.Box`: representa un conjunto de valores continuos entre los límites que se le imponen a la función, por ejemplo, `Box(-5, 5, dtype=np.float32)` comprende todos los valores que existen entre -5 y 5.

Existe algún tipo más de datos disponible en `gym.spaces`, pero los más sencillos y los que primero presenta *OpenAI-Gym* para usar en sus implementaciones son estos dos.

Por lo tanto es el momento de llevar a cabo la primera modificación en nuestro simulador de *tags* (Apéndice B) "inanimado" y crear estas variables requeridas por la plataforma.

En primer lugar hay que destacar que al inicio de cualquier *environment* es preciso hacer una breve introducción de su funcionamiento y variables importantes; en nuestro caso se ha hecho con estas dos:

Listing 5.1: Descripción del observation_space y action_space

```
Observation:
    Type: Box(3)
    Num Observation                 Min                     Max
    0   Simulation time             0                       Inf
    1   Unidentified tags           0                       Initial_Population
    2   Frame number                0                       Inf

Actions:
    Type: Discrete(8)
    Num Action
    0   Frame length = 2 slots
    1   Frame length = 4 slots
    2   Frame length = 8 slots
    3   Frame length = 16 slots
    4   Frame length = 32 slots
    5   Frame length = 64 slots
    6   Frame length = 128 slots
    7   Frame length = 256 slots
```

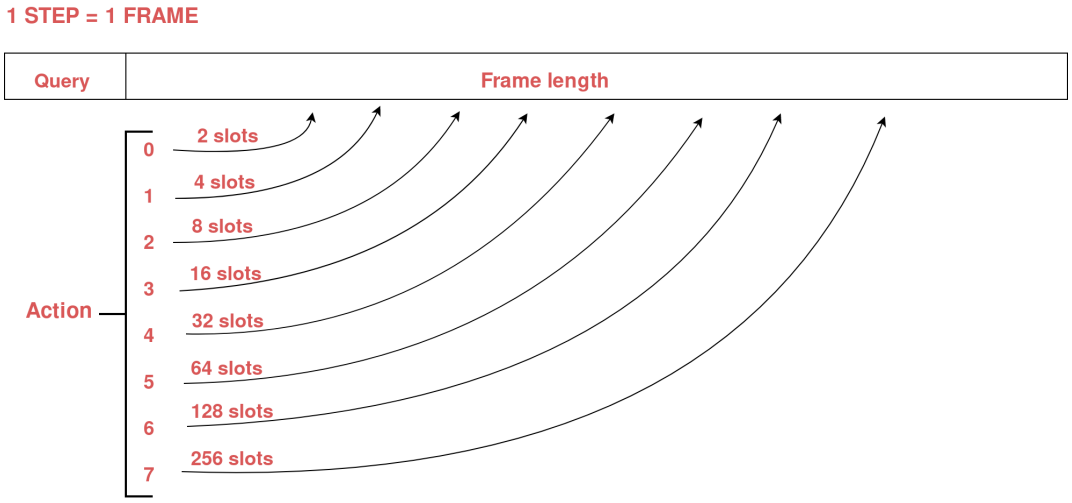

Figura 5.6.: Diagrama con las posibles acciones que permite nuestro *environment*

Aunque no se hayan comentado antes, acabamos de presentar la observación que el *environment* nos ofrecerá cada vez que el *agent* lo requiera y también las posibles acciones que el propio *agent* podrá llevar a cabo sobre

el *environment*. Como se puede comprobar, las acciones que nuestro entorno permite ejecutar sobre él mismo son la elección de la longitud de la trama.

5.2.2. Reward

Si volvemos sobre nuestros pasos y repasamos la teoría referente a este campo del *machine learning*, veremos como tanto las acciones, los estados y las recompensas son las principales variables de todas las ecuaciones que entran en juego.

Ahora es el momento de definir el *reward* y para ello hay que tener muy claro cual es nuestra finalidad, aunque estemos presentando muchos conceptos y puntos de vista, no debemos perder el rumbo del proyecto.

Como queremos minimizar el tiempo de lectura de una población de *tags*, debemos penalizar la recompensa cuando el tiempo de una trama sea demasiado grande y premiarla cuando se identifiquen *tags*. La idea sería crear una función que aumentara (mayor *reward*) cuando el tiempo de trama fuera más pequeño, o lo que es lo mismo, que el tiempo sea inversamente proporcional a la recompensa. Para el caso de los *tags* identificados ocurre justo lo contrario, siendo proporcional al *reward* (más *tags* identificados conllevan más *reward*).

$$reward = \frac{\#IdentifiedTags}{FrameTime} \tag{5.1}$$

Figura 5.7.: Animación recompensa vs castigo

5.2.3. `step, reset y render`

En primer lugar decir que el código completo del simulador modificado con las adaptaciones que precisa *OpenAI-Gym* se encuentra en el apéndice D.

En PYTHON las funciones se definen con la partícula `def`. De entre todas las posibles funciones que se podrían incluir en un *environment*, existen algunas que deben estar siempre para que el funcionamiento sea normal.

Como apunte quiero destacar que toda la cantidad de variables auxiliares que teníamos en el otro simulador (versión apéndice B), se han implementado en esta nueva versión dentro de una función llamada `def __init__(self)`. Al igual que estas variables se han incluido otras nuevas como son `action_space` y `observation_space` que han sido introducidas anteriormente.

Las dos funciones más importantes son `step` y `reset`:

`def step(self, action)`: recibe como argumento de entrada una acción y debe ser capaz de interpretarla correctamente. En nuestro caso, cuando esta función reciba el dato creará un *frame* como se hacía en el simulador pero en este caso no se empleará la política que se desarrolló en el paper (Alcaraz y col., 2013a), sino que se establecerá el tamaño de *frame* que indique la acción tomada por el *agent* (algoritmo RL).

El proceso que le sigue es el mismo que se seguía antes, se repartirán los *tags* por los distintos slots de manera aletoria y se procederá a su identificación. Al final de la trama la función `step` se encargará de devolver un *array* con:

- `observartion (object)`: se devuelve el `observation_space` que ya se introdujo antes, aunque por supuesto se devuelve actualizado según los cambios que haya experimentado el *environment*.
- `reward (float)`: se calcula y se devuelve un dato numérico que se asocia con ese paso dentro del episodio que se está llevando a cabo. En función del tiempo que haya pasado o los *tags* que se hayan identificado el *reward* será mayor o menor.

- `done (boolean)`: devuelve *True* si el episodio ha terminado p *False* si el episodio aún continua[4].
- `info (dict)`: variable que sirve para enviar información auxiliar al *agent*, pero que realmente no utiliza.

Listing 5.2: Código de `return` en la función `step`

```
self.state = (self.SimTime, self.Unident, self.Frame_number)
return np.array(self.state), reward, done, {}
```

`def reset(self)`: es una función que tiene como tarea reestablecer ciertas variables a su estado original con el fin de poder volver a empezar otro episodio en las mismas condiciones que los anteriores. En nuestro caso estas variables serán:

- `TagList`: esta estructura que contiene datos *booleanos* y nos indica qué *tags* están identificados y cuales no, se reestablecerá con todas sus posiciones a *false*.
- `SimTime`: el tiempo de simulación se establecerá al tiempo del slot de *Query*, común en todas las tramas a su comienzo.
- `Unident`: se inicializará al número de *tags* en la población, los que al inicio del nuevo episodio estarán sin identificar.
- `FrameNumber`: se inicializará a 0 ya que al inicio de un episodio aún no hay ninguna trama y por tanto este contador es nulo.

Como ocurría con la función de `step`, se devuelven varios valores, pero en este caso solo los referentes a la observación del *environment*.

Listing 5.3: Código de `return` en la función `reset`

```
self.state = (self.SimTime, self.Unident, self.Frame_number)
return np.array(self.state)
```

`def render(self, mode = 'human', 'rgb_array', 'ansi')`: como no es finalidad de nuestro trabajo, no nos hemos centrado en el desarrollo de esta función, pero simplemente nos ayuda a presentar los datos y hacerlo más visuales, desde cadenas de *strings* hasta pantallas gráficas con elementos en movimiento (como ocurre en algunos de los *environments* por defecto de *OpenAI-Gym*).

[4]La terminación de un episodio se lleva a cabo cuando se identifica a toda la población o cuando se ejecutan 100 pasos (valor establecido manualmente)

Nuestros futuros algoritmos de RL tendrán la capacidad de acceder a su antojo a las distintas funciones que les ofrece el *environment*, interactuando así con él.

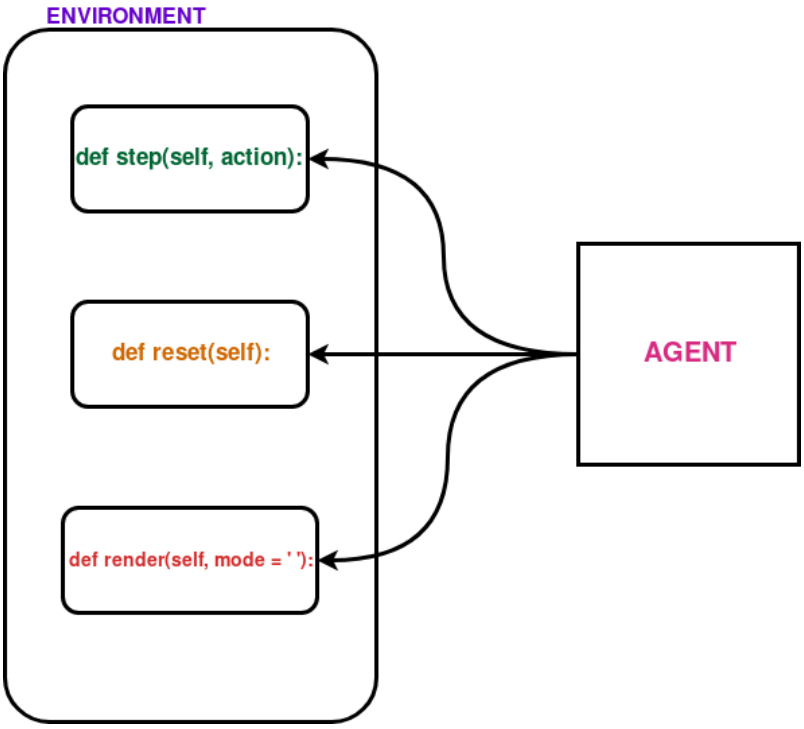

Figura 5.8.: Acceso del *agent* a las distintas funciones del *environment*

5.2.4. Registro de nuestro environment en OpenAI-Gym

Si queremos posteriormente llevar a cabo la unión entre los algoritmos de *reinforcement learning* y nuestro entorno del simulador de *tags*, es necesario registrar el *environment* en los directorios de installación de la plataforma *OpenAI-Gym* en nuestro equipo. Después de investigar bastante sobre el tema, se encontró un post que lo explica con un ejemplo genérico de una forma muy sencilla (Poddar, 2018).

Tras crear los ficheros de instalación necesarios, solo se precisaba colocar de forma adecuada la carpeta con estos ficheros y el código de nuestro *environment* (Apéndice D) en el directorio interno de *Anaconda*:

`/anaconda3/lib/python3.7/site-packages/gym_staticTagSim/`

Como se podrá comprobar a continuación, estos ficheros de los que se hablan (excepto el *environment* propiamente dicho) están formados por pequeñas porciones de código para que la plataforma *OpenAI-Gym* sea capaz de identificar nuestro entorno y lo incluya con el resto de entornos por defecto para hacer válidas las funciones y métodos que en él se usan.

La ramificación de ficheros y carpetas dentro de esta otra carpeta general se detalla en la siguiente imagen:

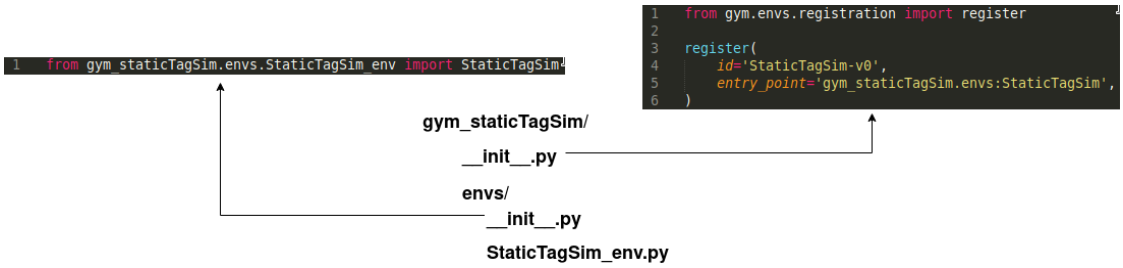

Figura 5.9.: Conjunto de ficheros y carpetas bajo `gym_staticTagSim`

Mientras uno de los ficheros se ocupa de localizar el *environment* dentro de los directorios, el otro lo registra mediante la partícula `register` con el resto de entornos de *OpenAI-Gym*.

Ha llegado el momento, nuestro simulador de *tags* está listo para empezar a trabajar con algoritmos de *reinforcement learning*.

6. Algoritmos en Reinforcement Learning

6.1. Tipos de algoritmos RL

Antes de empezar con la implementación de los algoritmos RL en PYTHON debemos hacer una diferenciación entre dos grandes grupos en función de si el *agent* tiene o no acceso al modelo del *environment*. El modelo imita el comportamiento del *environment* o permite deducir como se comportará:

- *Model-Based RL*: son modelos que se usan para planificación, es decir, conocemos como actuará el entorno al aplicar ciertas acciones antes de que se lleven a cabo.
- *Model-Free RL*: se refiere a aquellos entornos que aprenden por prueba y error (*trial-and-error*), siendo practicamente opuestos a los anteriores.

En la siguiente imagen 6.1 se trata de esbozar una pequeña clasificación de varios algoritmos de *reinforcement learning*. Empezando por la gran división que se ha explicado arriba, después se producen otras más específicas en las que no vamos a entrar.

El propósito a partir de este punto será entrar de lleno en la explicación teórica y la implementación en PYTHON con compatibilidad *OpenAI-Gym* de algunos algoritmos RL, intentando a su vez explotar todo su potencial y conseguir que el tiempo de lectura de los *tags* en nuestro simulador se minimice y que si no se llega a los valores óptimos desarrollados en el paper (Alcaraz y col., 2013a), se queden muy cerca.

En primer lugar haremos un viaje por los algoritmos de optimización de la política (*policy optimization*), donde tras una explicación de la base teórica,

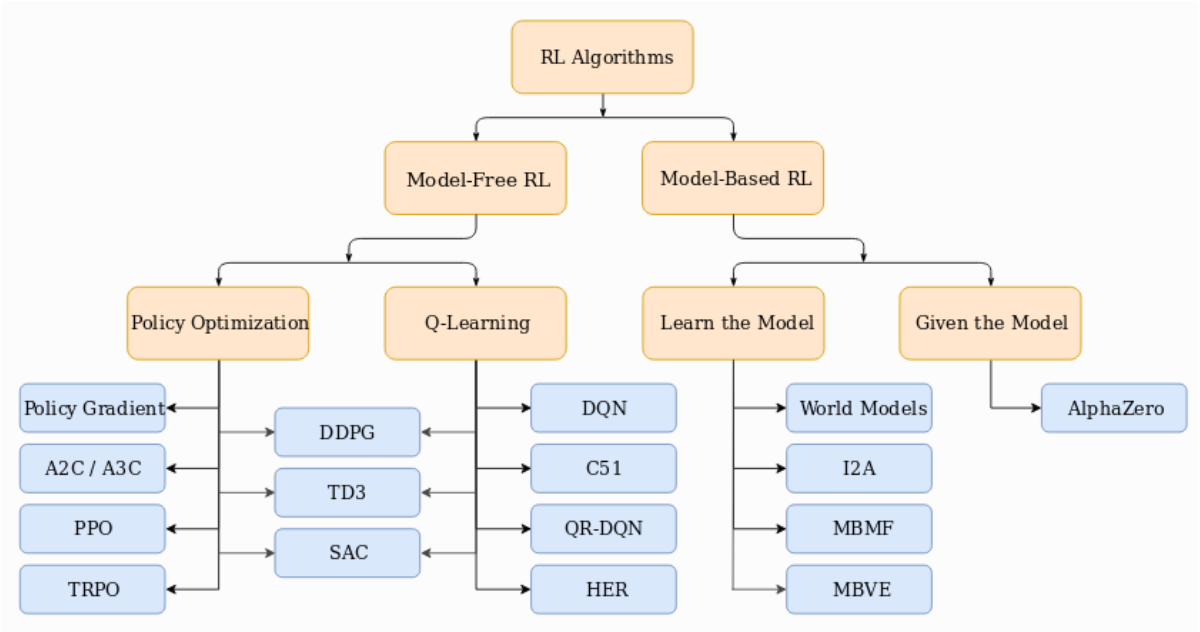

Figura 6.1.: Esquema de clasificación de algoritmos de RL

implementaremos *REINFORCE*, uno de los algoritmos más sencillos y fieles a la pura definición dentro de *policy gradient*.

A continuación pasaremos a *Q-learning* y *SARSA*, otros dos algoritmos que se desarrollan a partir de la explotación de la ecuación de *Bellman* para q_*, presentada en el punto 4.3 anterior.

6.2. Policy gradient: REINFORCE

6.2.1. Introducción

Indagando en los conceptos del *reinforcement learning* consultando el libro (Sutton y Barto, 2018) y el *post* (Hui, 2018)[1], paramos en las explicaciones de los algoritmos tipo *policy gradient*, como este primero que se presenta.

[1] otra fuente consultada: Weng, 2018

De forma introductoria a esta parte del RL, se ha decidido hacer una pequeña carta de presentación donde se sintetiza de una forma genérica como funcionan estos algoritmos, aunque el contexto real de estas palabras sea otro:

"You have to rely on the fact that you put the work in to create the muscle memory and then trust that it will kick in. The reason you practice and work on it so much is so that during the game your instincts take over to a point where it feels weird if you don't do it the right way." - Stephen Curry

Figura 6.2.: Sucesión de un lanzamiento de triple de *Stephen Curry*

Lo que vino a decir es que un buen entrenamiento es necesario para poder llegar a ejecutar las acciones de la manera perfecta. Al igual que en el deporte, en el mundo del RL una buena ejecución (y elección) de acciones es la clave para el éxito.

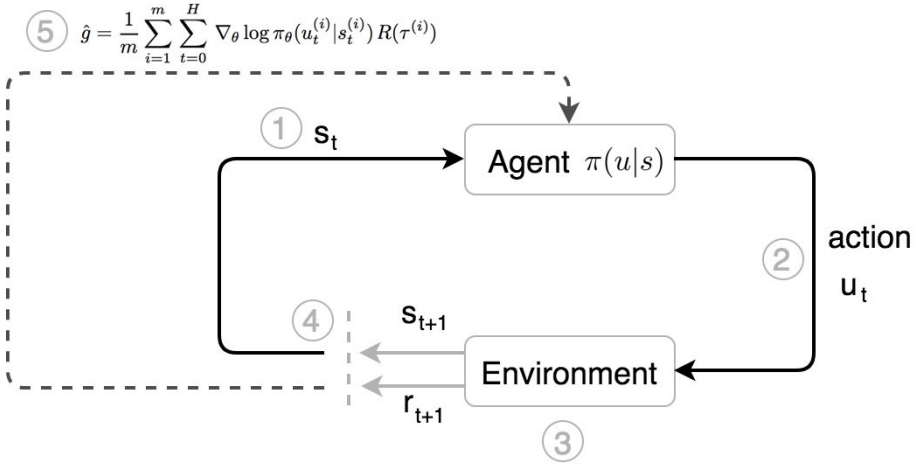

Figura 6.3.: Diagrama interacción *agent* y *environment* con *policy gradient*

En este punto, el diagrama genenal de la interacción entre *agent* y *environment* que ya se había introducido, es necesario modificarlo por este otro. Ahora entraremos con la notación, pero a grandes rasgos vemos como existe un nuevo elemento que será encargado de crear la intuición al *agent*, ayudandole a elegir las acciones.

1. El *agent* observa el estado del *environment* (S_t).
2. Toma una acción (a_t) basándose en su instinto (política π) en el estado correspondiente.
3. Tras la acción tomada, el *environment* reacciona con un nuevo estado (S_{t+1}).
4. El *agent* toma más acciones basadas en el nuevo estado observado.
5. Después de llevar a cabo una trayectoria (término introducido en 4.2.2), el *agent* ajusta su instinto (política π) basándose en las recompensas totales recibidas R_τ.

"Mantenemos lo que funciona y eliminamos lo que no"

Este instinto del que se habla se describe matemáticamente como:

$$\pi(a \mid s) : \text{probabilidad de tomar una acción } a \text{ en el estado } s. \tag{6.1}$$

6.2.2. Objetivo

Ahora es el momento de volver sobre nuestros pasos y recordar el momento en el que se presentó el término de *reward* 4.2.4. Se presentará ahora con una nomenclatura diferente pero la idea es la misma.

Las recompensas esperadas $J(\theta)$[2] se presentan como la suma de la probabilidad de que ocurra una trayectoria τ siguiendo una política π por las recompensas correspondientes a dicha trayectoria. Matemáticamente:

$$J(\theta) = \sum_{\tau} p(\tau;\theta) R(\tau) \tag{6.2}$$

Como sabemos nuestro objetivo final será minimizar el tiempo de lectura de los *tags* en el simulador, pero hablando con propiedad en *reinforcement learning*, esto se puede traducir con maximizar la recompensa que nuestro *environment* nos devolverá en cada paso de la interacción[3]. Así y continuando con la ecuación anterior, podemos traducir este enunciado a:

$$\max_{\theta} J(\theta) = \max_{\theta} [\sum_{\tau} p(\tau;\theta) R(\tau)] \tag{6.3}$$

es decir, encontrar una política π con la que se consiga llevar a cabo la trayectoria especifica que maximice la recompensa esperada J[4].

Pues bien, tras aclarar estos conceptos ya es posible enunciar el problema de optimización que formaliza los algoritmos *policy gradient*: se va a buscar modelar una política π (la modelamos modificando el vector de parámetros θ) que conlleve a una trayectoria que maximice la recompensa total del sistema.

$$\theta^* = \underset{\theta}{argmax} [\sum_{\tau} \pi_{\theta}(\tau) R(\tau)] \tag{6.4}$$

[2] el símbolo θ se usará a partir de ahora para referirnos a un vector de parámetros del que dependerá la política π

[3] en función de nuestra implementación particular del *environment*.

[4] el término $p(\tau;\theta)$ se puede intercambiar por $\pi_{\theta}(\tau)$ ya que expresan la misma idea.

Valor esperado

Cabe hacer un inciso ya que la parte importante de esta ecuación que se presentó unas líneas arriba, se puede encontrar de distintas maneras, tanto en artículos relacionados como incluso a lo largo de esta explicación más adelante.

$$\mathbb{E}_{\tau:\pi_\theta(\tau)}[r(\tau)] = \sum_{\tau} \pi_\theta(\tau) r(\tau) \tag{6.5}$$

El valor esperado es la suma de los productos del valor correspondiente por su probabilidad de que ocurra, en nuestro caso, el valor de recompensa asociado a una trayectoria por la probabilidad de que se lleve a cabo dicha trayectoria.

6.2.3. Optimización

Vamos a introducir un truco matemático que nos aportará en sucesivas ecuaciones un *logaritmo* para poder aprovechar sus características y llevar a cabo la optimización que perseguimos.

La derivada parcial de una función f(x) es igual a f(x) veces la derivada parcial del *logaritmo* de f(x). Dicho de otro modo:

$$f(x)\nabla_x log f(x) = f(x)\frac{\nabla_x f(x)}{f(x)} = \nabla_x f(x) \tag{6.6}$$

que aplicado a nuestro contexto, siendo f(x) las probabilidades de las trayectorias $\pi_\theta(\tau)$:

$$\pi_\theta(\tau)\nabla_\theta log \pi_\theta(\tau) = \nabla_\theta \pi_\theta(\tau) \tag{6.7}$$

Haciendo la aplicación inversa en la ecuación 6.2 nos queda que:

$$\begin{aligned} J(\theta) &= \sum_{\tau} \pi_\theta(\tau) R(\tau); \\ \nabla_\theta J(\theta) &= \sum_{\tau} \nabla_\theta \pi_\theta(\tau) R(\tau) \\ &= \sum_{\tau} \pi_\theta(\tau) \nabla_\theta log \pi_\theta(\tau) R(\tau) \\ &= \mathbb{E}_{\tau:\pi_\theta(\tau)}[\nabla_\theta log \pi_\theta(\tau) R(\tau)] \end{aligned} \tag{6.8}$$

Intercambiando el valor de $R(\tau)$ por la recompensa esperada tal y como se introdujo en 4.4 y desarrollando la derivada de la expresión logarítmica que hemos conseguido:

$$\nabla_\theta J(\theta) = \mathbb{E}_\pi[G_t \frac{\nabla \pi(A_t \mid S_t, \theta)}{\pi(A_t \mid S_t, \theta)}] \tag{6.9}$$

Se puede ver que la trayectoria se representa como llevar a cabo una acción en función del estado en el que se encuentre el sistema siguiendo la política θ, o lo que es lo mismo $(A_t \mid S_t, \theta)$.

Ascenso del gradiente

Los algoritmos *policy gradient* tienen este nombre porque a la hora de la optimización aplican un **ascenso/descenso del gradiente**. Este proceso es un proceso algoritmico famoso entre el aprendizaje automático y nos sirve como método general para encontrar el valor máximo/mínimo de una función. En nuestro caso nos centraremos en el **ascenso del gradiente (gradient ascent)** debido a que nuestro objetivo es maximizar la función de recompensa.

$$\theta \longleftarrow \theta + \alpha \nabla_\theta J(\theta) \tag{6.10}$$

o lo que es lo mismo

$$\theta_{t+1} = \theta_t + \alpha G_t \frac{\nabla \pi(A_t \mid S_t, \theta)}{\pi(A_t \mid S_t, \theta)} \tag{6.11}$$

En la siguiente imagen podemos ver como funciona el proceso iterativo de ascenso del gradiente donde la gráfica en verde representa la función de recompensa (en la realidad no es así, se usa esta función sencilla para la explicación).

En el caso de que estemos en la parte creciente de la función, significará que el máximo nos queda a la derecha, por lo tanto la derivada de nuestra función dará un valor ≥ 0 que junto al parámetro α [5] actualizarán la política θ hacia el valor buscado.

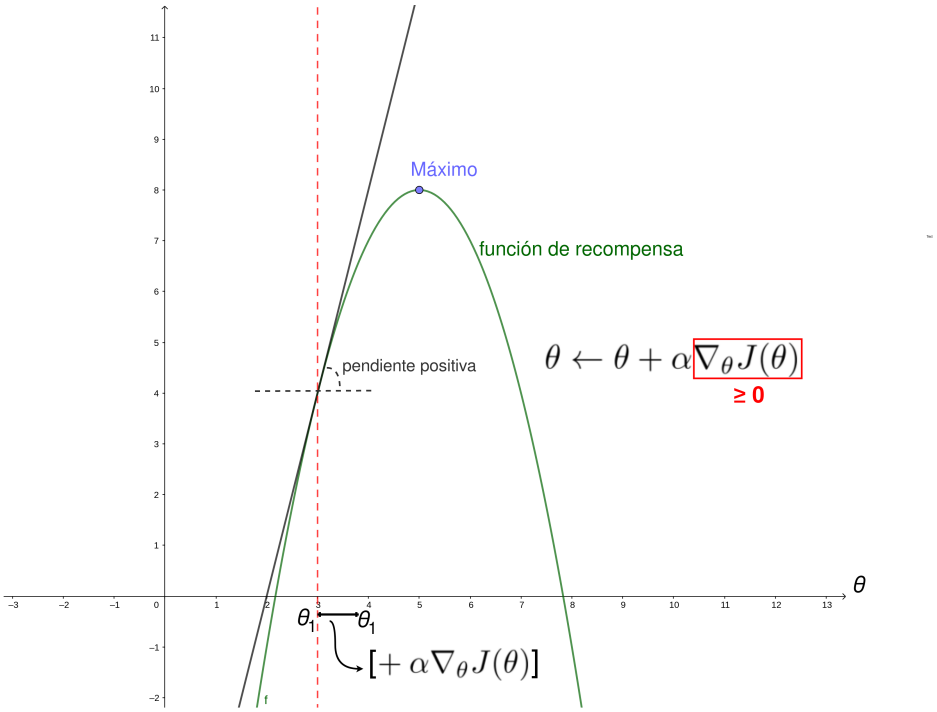

Figura 6.4.: Representación ascenso del gradiente con pendiente positiva

[5] valores muy pequeños llevan a algoritmos muy lentos; valores muy grandes llevan a divergencia/inestabilidad.

Si en cambio estamos en la parte decreciente de la función, entonces la derivada será negativa. Ahora la actualización de θ será hacia el lado izquierdo del eje (donde queda el máximo) por lo que la multiplicación de nuestra derivada y α es negativa.

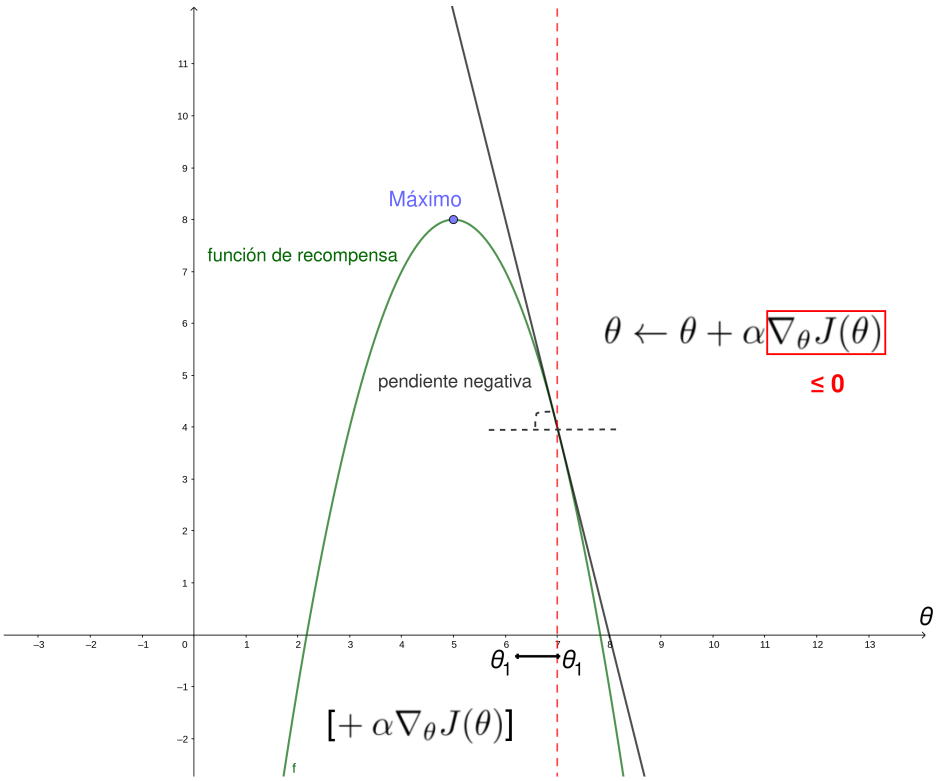

Figura 6.5.: Representación ascenso del gradiente con pendiente negativa

Como podemos observar, con este proceso iterativo nuestra política se irá actualizando y a la vez maximizando hasta conseguir una recompensa favorable a nuestro problema y consiguiendo resolver o aproximarnos a la solución del problema propuesto en 6.4.

A continuación vamos a entrar de lleno en el código PYTHON que se enlazará con el *environment* creado para poner en marcha el algoritmo.

6.2.4. Implementación en python

No debemos desviarnos de nuestro camino, queremos maximizar la función de recompensa porque una gran recompensa irá asociada a un valor pequeño de tiempo en las tramas de nuestro lector de *tags* y por tanto a una minimización en el tiempo total de identificación. De ahí que en estas gráficas y con ayuda de este proceso se busquen los valores máximos.

A raíz de la explicación anterior, el algoritmo REINFORCE (también conocido como *Monte Carlo policy gradient*) posee una estructura que es muy fiel a lo explicado en los puntos anteriores, por lo tanto este tipo de algoritmo *policy gradient* no difiere mucho de la definición general de los mismos.

REINFORCE: Monte-Carlo Policy-Gradient Control (episodic) for π_*

Input: a differentiable policy parameterization $\pi(a|s,\boldsymbol{\theta})$
Algorithm parameter: step size $\alpha > 0$
Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ (e.g., to $\mathbf{0}$)

Loop forever (for each episode):
  Generate an episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$, following $\pi(\cdot|\cdot,\boldsymbol{\theta})$
  Loop for each step of the episode $t = 0, 1, \ldots, T-1$:
    $G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$ (G_t)
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \gamma^t G \nabla \ln \pi(A_t|S_t,\boldsymbol{\theta})$

Figura 6.6.: Pseudocódigo algoritmo *policy gradient REINFORCE*

Si observamos el pseudocódigo del algoritmo vemos como en la parte principal existen dos bucles anidados donde el más interno corresponde con los pasos dentro de la interacción y el más externo con la cantidad de episodios totales de la simulación/entrenamiento. (Apéndice E)

Dentro del bucle de los pasos es donde se lleva a cabo la actualización de la función de recompensa y donde se aplica el *ascenso del gradiente* para mejorar la política.

Actualización de la función de recompensa + gradient ascent

Esta parte es la más importante del proceso ya que de ella depende que vayamos mejorando el *reward* y con él minimizando el tiempo (*simTime*) de identificación. A simple vista, el código de esta parte puede parecer un tanto confuso, pero es una técnica muy ingeniosa con la que crear los valores de G_t y actualizar la política:

Listing 6.1: Código función *update_policy* con cálculo (*discounted_rewards*)

```
def update_policy(policy_network, rewards, log_probs):
    discounted_rewards = []

    for t in range(len(rewards)):
        Gt = 0
        pw = 0
        for r in rewards[t:]:
            Gt = Gt + GAMMA**pw * r
            pw = pw + 1
        discounted_rewards.append(Gt)

    discounted_rewards = torch.tensor(discounted_rewards)

    policy_gradient = []
    for log_prob, Gt in zip(log_probs, discounted_rewards):
        policy_gradient.append(-log_prob * Gt)

    policy_network.optimizer.zero_grad()
    policy_gradient = torch.stack(policy_gradient).sum()
    policy_gradient.backward()
    policy_network.optimizer.step()
```

En la lista *rewards* se incluyen las recompensas de una trayectoria. Con el bucle más externo se recorren estas recompensas y con el más interno se van sumando de forma acumulativa con descuento (γ - variable GAMMA).

El valor de *log_prob* se computa antes con la ayuda de una librería de PYTHON llamada *torch* que incluye varios métodos muy útiles en aprendizaje automático. Esta variable representa el logaritmo de las probabilidades $\pi(A_t \mid S_t, \theta)$.

Bucle principal

Con respecto al bucle principal del algoritmo, la filosofía es clara y es la que se ve reflejada en el pseudocódigo 6.6. Incluyendo las llamadas a las funciones correspondientes *OpenAI-Gym* de nuestro *environment* - *StaticTagSim-vo* como son RESET Y STEP, interactuamos con él con el objetivo de ir recibiendo recompensas y nuevos estados.

Listing 6.2: Bucle principal del algoritmo *REINFORCE - policy gradient*

```
for episode in range(max_episode_num):
    state = env.reset()
    log_probs = []
    rewards = []

    for steps in range(max_steps):
        action, log_prob = policy_net.get_action(state)
        new_state, reward, done, _ = env.step(action)
        log_probs.append(log_prob)
        rewards.append(reward)

        if done:
            simTimes.append(new_state[0])
            update_policy(policy_net, rewards, log_probs)
            numsteps.append(steps)
            all_rewards.append(np.sum(rewards))
            if episode %1 == 0:
                sys.stdout.write("episode: {}, length: {}, simTime: {}\n"
                                 .format(episode, steps, new_state[0]))
            break

        state = new_state
```

Finalmente se lleva a cabo un almacenamiento de los datos para su posterior representación donde observaremos las mejoras encontradas tanto en los *reward* a lo largo de los episodios como en el tiempo de simulación en la identificación de los *tags* - *simTime*.

6.2.5. Resultados e interpretación

Haciendo uso de la librería *matplotlib.pyplot* para la representación de resultados en PYTHON, se han ilustrado una serie de gráficas que se desarrollan a partir de las siguientes características:

- Se ha trabajado con una población inicial de *tags* de 30 elementos.
- El valor de γ para el descuento en las recompensas se ha establecido en 1, ya que nuestro sistema tiene un estado de terminación y no afecta.
- El número de episodios con el que se va a entrenar nuestro algoritmo con la interacción con el *environment* será de 2000, en principio.
- El *learning rate* α que se usa en las actualizaciones de *ascenso del gradiente* es de 0.0001 llevando a cabo así saltos pequeños hasta converger al máximo.

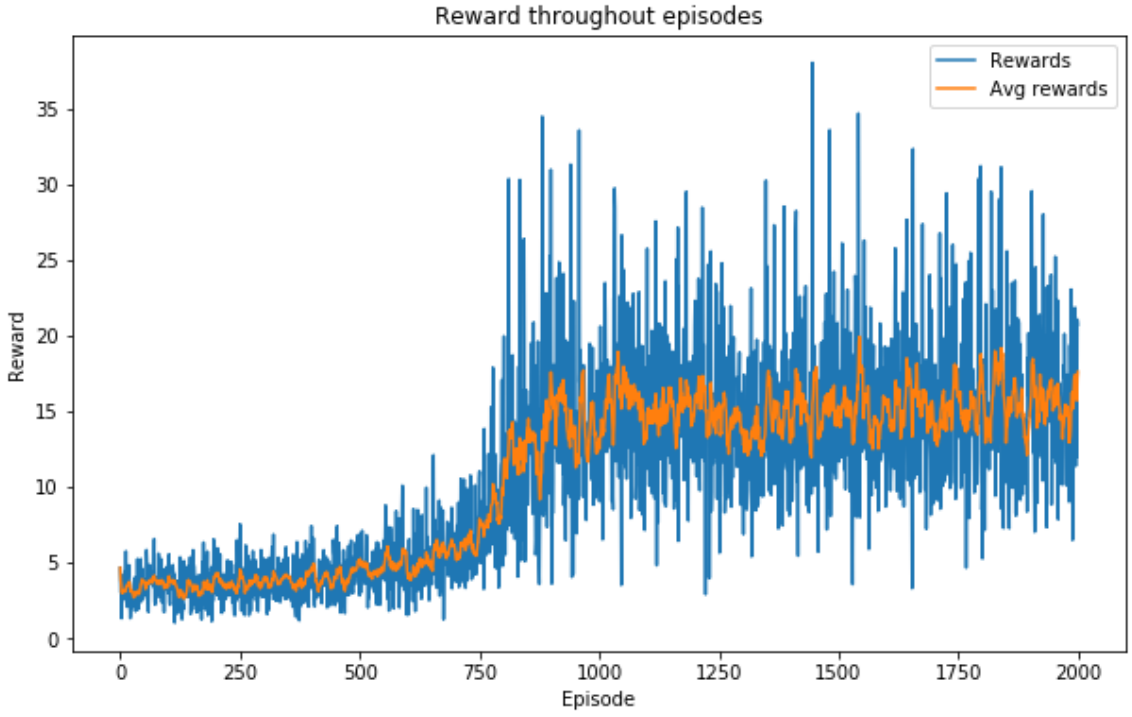

Figura 6.7.: Variación del *reward* en *policy gradient REINFORCE*

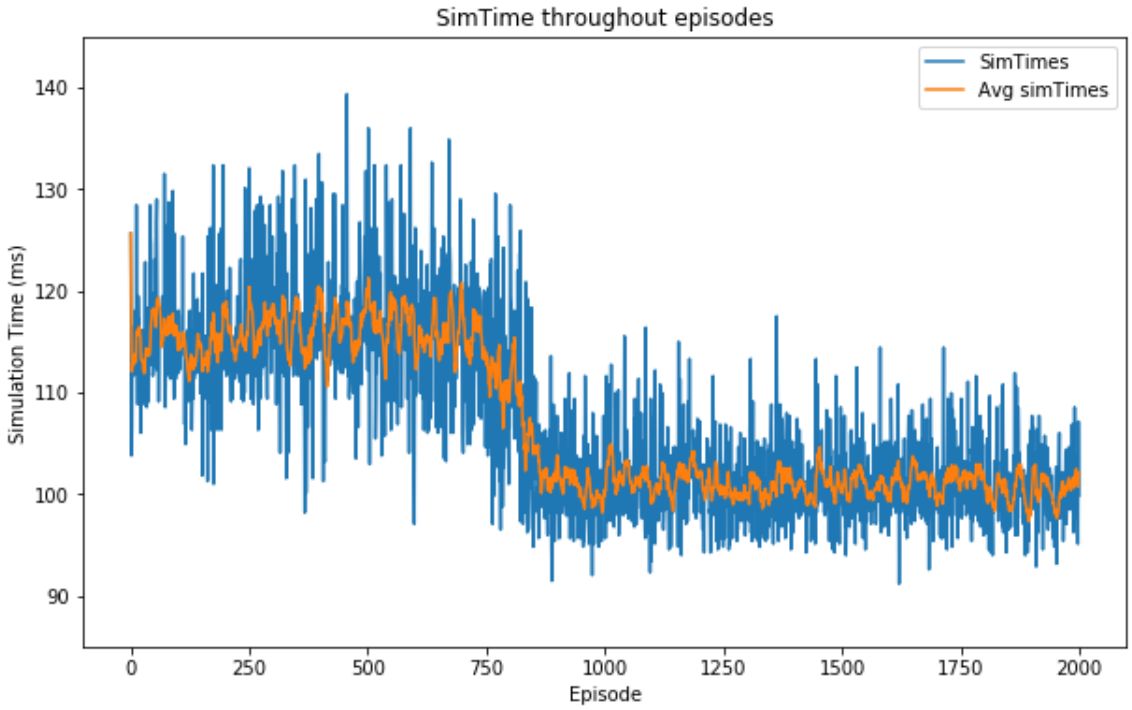

Figura 6.8.: Variación del *simTime* en *policy gradient REINFORCE*

A simple vista podemos ver la actuación del algoritmo en nuestro simulador, tenemos los primeros resultados positivos de nuestra aplicación con *reinforcement learning*. Se expresan los valores de la simulación (SimTimes y Rewards) y una media de los mismos (Avg_simTimes y Avg_rewards) con el tiempo para visualizar de forma más suave la variación.

Aunque se ha demostrado matemáticamente y los resultados eran esperados, podemos ver como a lo largo de los episodios de interacción entre *agent* y *environment* (algoritmo y simulador) se ha conseguido una mejora notable. En la primera gráfica vemos como existe un aumento del *reward* tras varios episodios de entrenamiento del algoritmo. Este incremento se hace ver en sentido opuesto con el tiempo de simulación (tiempo de identificación de *tags - simTime*); cuando el *reward* empieza a subir, el tiempo de identificación baja llegandose a conseguir mínimos cercanos a los 90 ms para la población de inicio de 30 *tags*.

6.3. Q-learning

6.3.1. Introducción

Continuando con los algoritmos del tipo *Model-free RL*, es el momento de presentar *Q-learning*, un algoritmo que como su nombre indica se desarrolla a partir de los valores Q y la ecuación de Bellman correspondiente (introducida en 4.13). Hasta el momento no se ha hablado de los valores Q, pero lo haremos más adelante cuando entremos en la explicación teórica del algoritmo.

Al igual que la anterior implementación, este algoritmo también es usado para aprender una política óptima dentro de un *MDP*. La idea de encontrar una política está referida a que esa política sea óptima en el sentido de que el valor esperado de recompensa total a lo largo de los pasos sucesivos sea la máxima alcanzable.

El modelo del sistema RL sigue siendo el mismo, un agente interacciona mediante acciones con un entorno. Éste cambia de estado con cada acción y devuelve una recompensa. Ahora bien, ¿cómo elige el agente las acciones?

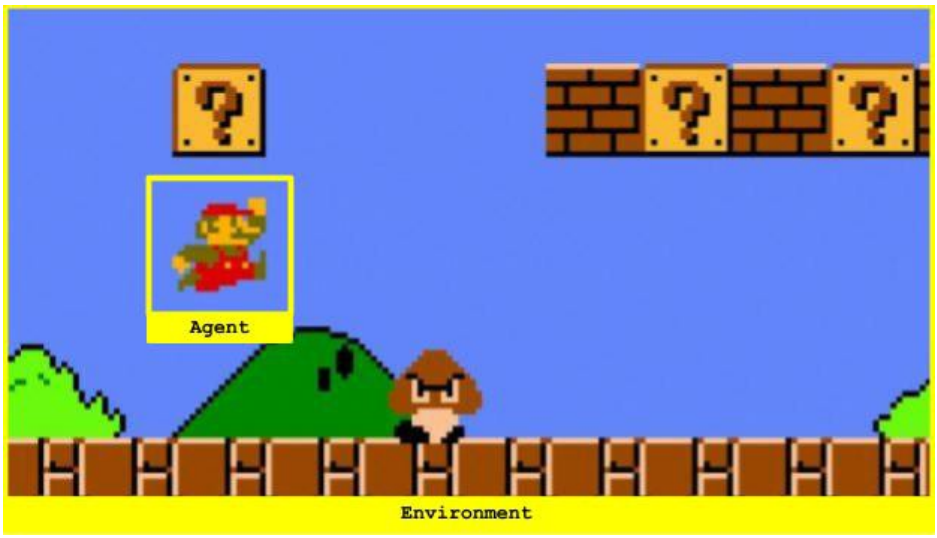

Figura 6.9.: Animación Super Mario comparación *agent-environment*

6.3.2. Objetivo y Q-table

Ahora vamos a definir el objetivo del algoritmo de una forma más precisa/ matemática. Como se ha adelantado arriba, para el desarrollo de este algoritmo vamos a partir de la ecuación de *Bellman* para q_*:

$$q_*(\text{s, a}) = \mathbb{E}[R_{t+1} + \gamma \max_{a'} q_*(S_{t+1}, a') \mid S_t = s, A_t = a] \tag{6.12}$$

En ella podemos diferenciar tres partes muy claramente:

- q_*(s, a): que corresponde con el valor esperado de recompensa máximo total (*expected return*).
- R_{t+1}: que corresponde con el valor esperado de recompensa en el instante posterior (*expected reward*).
- $\gamma \max_{a'} q_*(S_{t+1}, a')$: que corresponde con el maximo valor esperado de recompensa con descuento[6] en el futuro (*maximum expected discounted return*).

En esta expresión podemos ver como $s^{'}$ es el siguiente estado que se alcanza con la mejor acción $a^{'}$ que se puede ejecutar en tiempo t+1.

Una vez que este valor q_* se desarrolla y actualiza para cada par estado-acción, nuestro algoritmo RL encontrará la acción que maximiza $q_*(s, a)$ para cada estado posible, como veremos posteriormente en el análisis del código.

La incognita que dejamos en la introducción con respecto a los valores Q (*Q-values*) ya puede ser resuelta. Los *Q-values* no son más que estos datos que acabamos de presentar tomando la recompensa actual y las posteriores esperadas con un descuento apropiado. Se crea un *Q-value* para cada par estado-acción, por lo que será necesaria la creación de una estructura que permita albergar, acceder y editar fácilmente estos datos.

[6] el ratio de descuento ya se ha presentado con anterioridad y se usa para dar más peso a las recompensas más cercanas en el tiempo, ya que afectan más inmediatamente a la recompensa actual

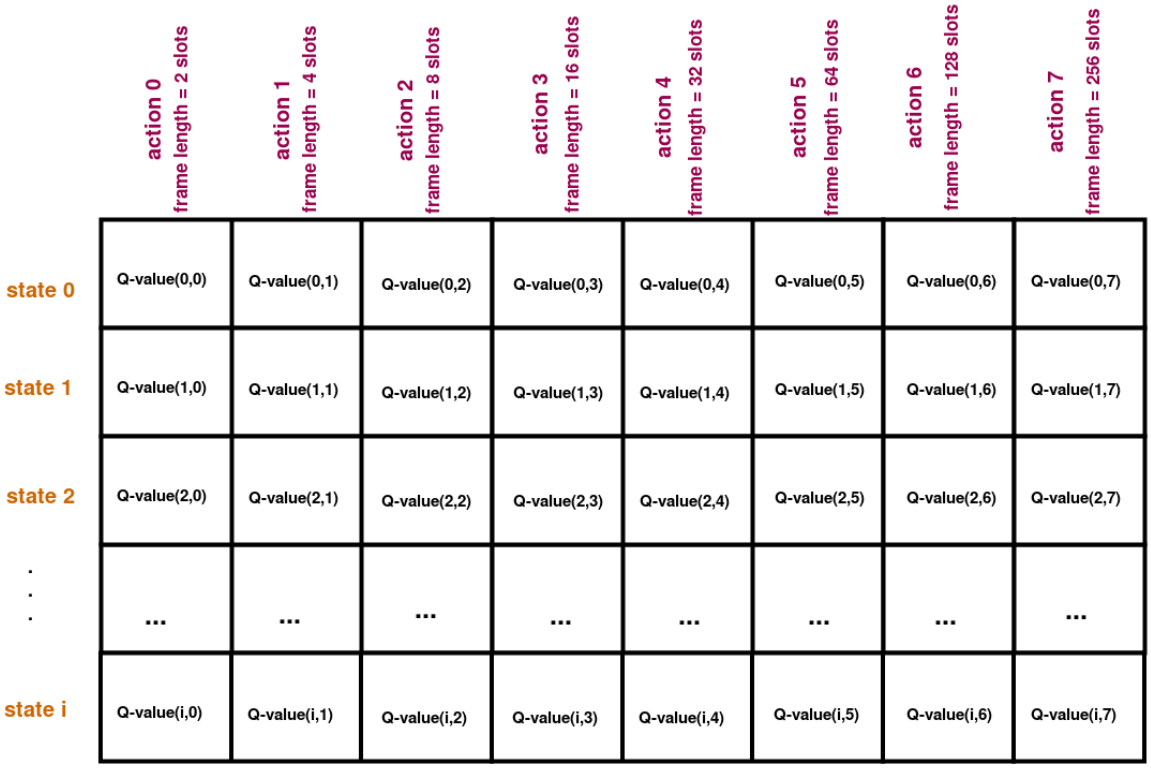

Figura 6.10.: Representación de la estructura *Q-table*

En esta figura podemos ver la estructura general de *Q-table*, una tabla que alberga los *Q-values* donde podemos mapear cada valor con las acciones (columnas) y estados (filas) correspondientes a nuestro *environment*.

A medida que nuestro algoritmo actualiza los *Q-values* (en puntos venideros explicaremos el proceso), éstos se van actualizando en la *Q-table* para que nuestro *agent* los tenga a mano y elija según le convenga las acciones que le reportan mayor *reward*.

6.3.3. Exploración vs explotación

En el contexto que nos situamos de *reinforcement learning*...

- Exploración: proceso de explorar el *environment* con la finalidad de recabar información sobre él.
- Explotación: proceso de explotar la información que ya se conoce acerca del *environment* con el fin de maximizar la recompensa que este genera.

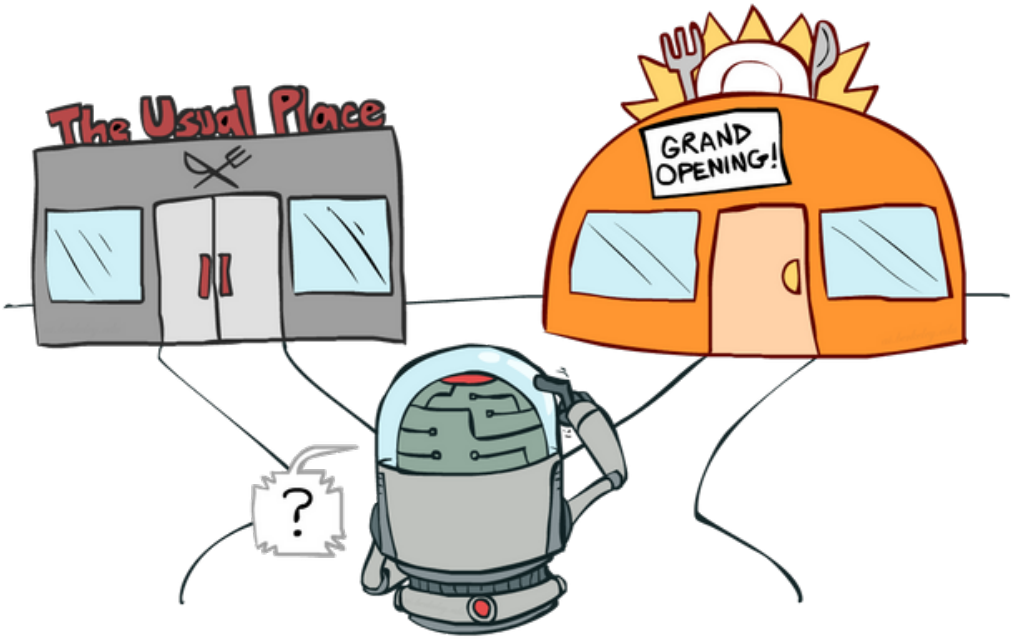

Figura 6.11.: Animación sobre el dilema de exploración vs explotación en *RL*

Este paradijma de los algoritmos de *reinforcement learning* expone la diferencia entre dos posibles modos de actuación bastante generales a los que se puede enfrentar un *agent* en un entorno cualquiera.

Mientras que con el modo de exploración el *agent* explorará nuevos estados y nuevas acciones en dichos estados que tal vez no había probado hasta el momento, con el modo de explotación el *agent* llevará a cabo trayectorias

(consecución de estados y acciones en el tiempo) similares de forma repetida. Estas trayectorias que repite suelen ser aquellas que les ofrece una mayor recompensa.

Figura 6.12.: Animación sobre el dilema de exploración vs explotación en *RL*

Si solo usáramos explotación, nuestro *agent* quizá se estaría perdiendo información (positiva y/o negativa) del *environment* y no explotando (valga la redundancia) los recursos de la forma más apropiada. En cambio si solo usamos exploración no estaríamos haciendo uso de esa información que estamos aprendiendo para obtener buenas recompensas.

Estrategia epsilon greedy

Esta estrategia es usada en el algoritmo presente para conmutar entre el modo de exploración y el modo de explotación. Aunque quedará reflejado en el código PYTHON, se presenta más abajo un pequeño pseudocódigo con esta diferenciación.

Si el caso es explorar el sistema, la acción que se llevará a cabo será aleatoria. En cambio, si el caso es explotar el sistema, sacarle el máximo partido y obtener las mejores recompensas, habrá que elegir las acciones que nos ofrecen mayores recompensas. ¿Cómo?. Preguntando a *Q-table*.

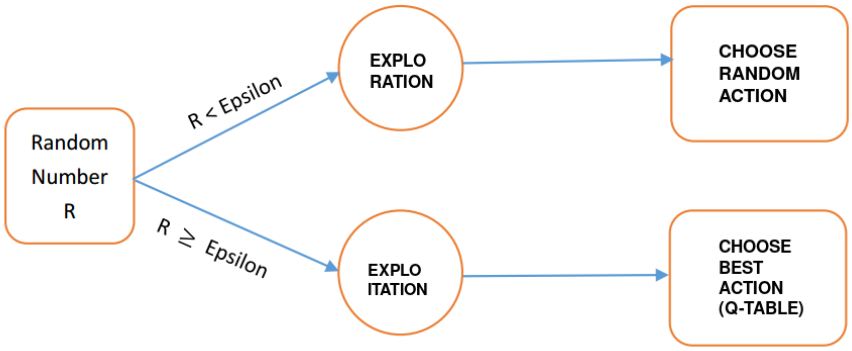

Figura 6.13.: Flujo comportamiento estrategia *epsilon greedy*

El valor de *epsilon* irá variando a lo largo de una interacción del *agent* con el *environment* de forma que en cada episodio se vaya actualizando. Como se habrá supuesto se empezará por el modo de exploración ya que al inicio del algoritmo no existe ningún entrenamiento y la *Q-table* está vacía.

6.3.4. Actualización y cálculo de los Q-values

Llegados a este punto solo nos falta presentar como actualizamos los datos que albergamos en la *Q-table*, los *Q-values*.

Obviamente nuestro objetivo es conseguir que el *Q-value* correspondiente a un par estado-acción converja al óptimo *Q-value* q_* del mismo par estado-acción. Este proceso se puede alcanzar de forma iterativa entrenando el algoritmo (explorando el *environment* y observando cuando recibimos las mejores recompensas).

Para desarrollar este proceso iterativo hay que presentar antes un parámetro de aprendizaje (*learning rate*), el cual nos indica como de rápido nuestro *agent* abandona un *Q-value* anterior de la *Q-table* y lo sustituye por uno nuevo.

$$\text{Learning rate} \equiv \alpha \tag{6.13}$$

La idea no es sobreescribir el viejo valor, sino usar el ratio de aprendizaje[7] como herramienta para determinar cuanta información mantenemos del valor previo *Q-value* para un par estado-acción frente al nuevo *Q-value* calculado para el mismo par estado-acción.

$$q^{new}(s,a) = (1-\alpha)q(s,a) + \alpha(R_{t+1} + \gamma \max_{a'} q(s',a')) \tag{6.14}$$

La parte en rojo corresponde con el valor antiguo del *Q-value* y la parte en azul con el valor aprendido.

6.3.5. Implementación en python

Vamos a destacar las partes de nuestro algoritmo implementado en PYTHON que se han considerado más importantes y que se asemejan con las explicaciones anteriormente desarrolladas. Del libro Sutton y Barto, 2018 extraemos una visión genérica sobre la implementación con este pseudocódigo:

Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

Algorithm parameters: step size $\alpha \in (0,1]$, small $\varepsilon > 0$
Initialize $Q(s,a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal,\cdot) = 0$

Loop for each episode:
  Initialize S
  Loop for each step of episode:
    Choose A from S using policy derived from Q (e.g., ε-greedy)
    Take action A, observe R, S'
    $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$
    $S \leftarrow S'$
  until S is terminal

Figura 6.14.: Pseudocódigo algoritmo *Q-learning*

En primer lugar la estrategia *epsilon greedy* para balancear la exploración y explotación se ha implementado con un par de condiciones *if* / *else* que

[7] en nuestra implementación, cuando mayor es el ratio de aprendizaje (α) más rápido se adoptarán nuevos *Q-values*.

reproducen a la perfección la idea que perseguíamos. Van ligadas a las líneas inferiores donde se actualiza el ratio de exploración (epsilon).

Listing 6.3: Condiciones para la elección entre exploración y explotación

```
# Exploration-exploitation trade-off
exploration_rate_threshold = random.uniform(0, 1)
if exploration_rate_threshold > exploration_rate:
    action = np.argmax(q_table[int(Unident),:])
else:
    action = env.action_space.sample()

# Exploration rate decay
exploration_rate = min_exploration_rate +
(max_exploration_rate - min_exploration_rate) *
np.exp(-exploration_decay_rate*episode)
```

Otra línea del código muy importante que refleja lo anunciado en la ecuación 6.17 es la siguiente donde se actualiza en la posición correcta de *Q-table* 6.10 el *Q-value* correspondiente:

Listing 6.4: Implementación de la actualización de *Q-values*

```
# Update Q-table for Q(s,a)
q_table[int(Unident), action] = q_table[int(Unident), action] *
(1 - learning_rate) + learning_rate * (reward + discount_rate *
np.max(q_table[state, :]))
```

El resto del código se compone de las llamadas correspondientes a nuestro *environment* personalizado y a los bucles *for* principales que componen el número de episodios y el número de pasos dentro de cada uno.

En el apédice F se adjunta el código completo del algoritmo implementado, un nuevo agente con el que entrenar nuestro simulador, aumentar las recompensas de los pares estados-acción y ver como todo esto se repercute en la minimización / mejora del tiempo de identificación de la población de *tags*.

6.3.6. Resultados e interpretación

En el caso de este algoritmo (*Q-learning*) hay que destacar diferencia con respecto a su predecesor en este proyecto (*REINFORCE*). Ahora necesitaremos más tiempo/episodios de entrenamiento, ya que para actualizar la *Q-table* es un proceso lento que depende de cuantos estados y acciones tengamos.

Las especificaciones con las que se ha programado el algoritmo y de las que dependerán nuestros resultados son:

- Se ha trabajado con una población inicial de *tags* de 30 elementos.
- El valor de γ para el descuento en las recompensas se establece en 1.
- El número de episodios con el que se va a entrenar nuestro algoritmo con la interacción con el *environment* será de 3000, en principio.
- El *learning rate* α usado es de 0.2.
- El valor de *epsilon* empezará en 1 e irá disminuyendo con un *"decay_rate"* de 0.0008.

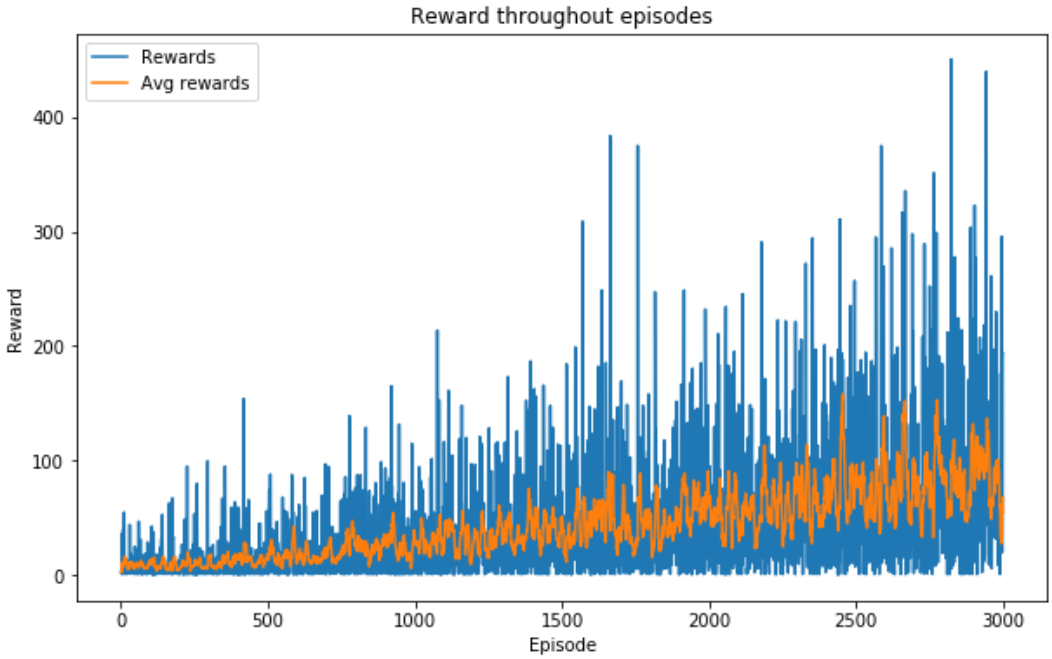

Figura 6.15.: Variación del *reward* en *Q-learning*

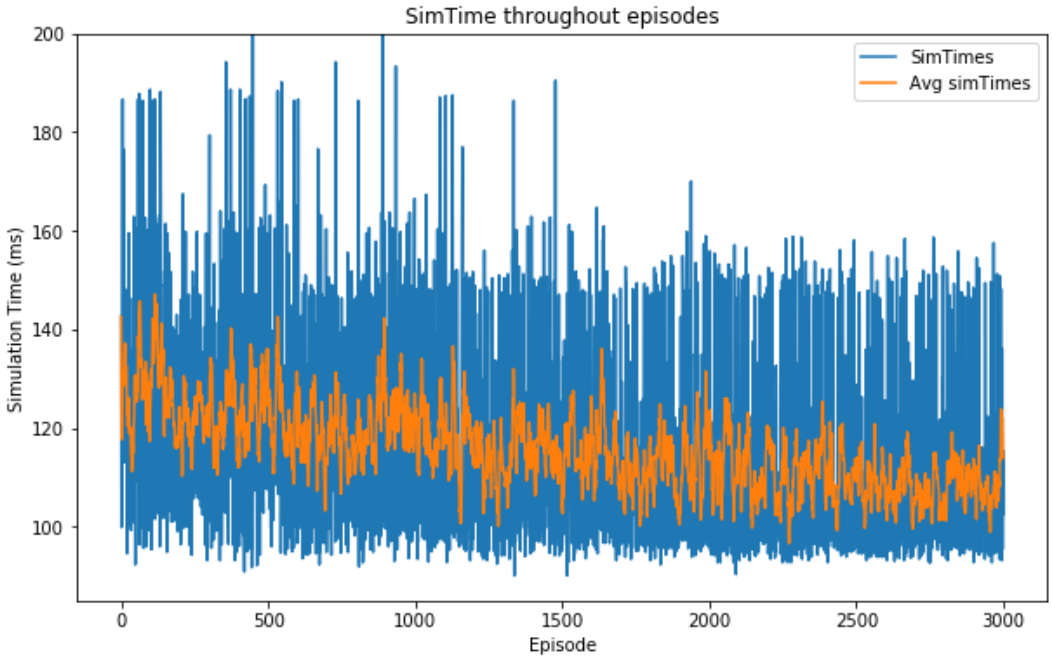

Figura 6.16.: Variación del *simTime* en *Q-learning*

En primer lugar destacar de forma inmediata la subida gradual de la recompensa a medida que los episodios transcurren y nuestro algoritmo/agente aprende más. Este incremento gradual se ve reflejado en la segunda gráfica en una disminución lenta pero efectiva del tiempo total de identificación de los *tags*.

Si nos centramos en la gráfica de tiempos, podemos ver que existe una gran diferencia entre los valores mínimos y máximos alcanzados en el proceso (zona azul). Lo normal o esperado sería que esta diferencia de tiempo fuera lo más pequeña posible porque es como mejor se vería la disminución temporal.

Esto sucede porque nuestro *environment* no refleja una situación realmente adecuada para *reinforcement learning* y más en concreto para algoritmos tabulares, es decir, deberíamos crear aún más incertidumbre y dejar aún más tiempo de entrenamiento para ver mejoras notables en el tiempo de identificación. Volverá a ocurrir con *SARSA* en la proxima sección.

Una forma de solucionar este problema sería por ejemplo que las acciones se tomen slot a slot y no trama a trama o que no se conociera de primeras el número de población.

6.4. SARSA

6.4.1. Introducción

Es el momento de presentar el último algoritmo que vamos a estudiar y desarrollar en este trabajo. Se trata de un algoritmo tipo tabular.

Los algoritmos tabulares (*Q-learning* también está incluido en esta categoría) reciben este nombre porque se refieren a la resolución de problemas con espacios de estados y acciones lo suficientemente pequeñas como para poder aproximar una función valor - representable en *arrays* o tablas.

Se adjunta este diagrama para visualizar de donde viene el nombre del algoritmo y aunque se entrará más en detalle en las transiciones que se presentan, *SARSA* no son más que las nomenclaturas de *State*, *Action*, *Reward*, *State*, *Action*.

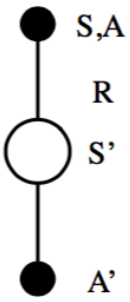

Figura 6.17.: Diagrama transiciones algoritmo *SARSA*

Como veremos en los puntos venideros, este algoritmo es muy parecido a *Q-learning*, pero se ha elegido su implementación porque los dos poseen una característica totalmente diferente y es interesante presentarla en un trabajo como este enfocado al *reinforcement learning*. El próximo punto ataja esta cuestión entre los algoritmos *On-policy* y los *Off-policy*.

6.4.2. On-policy vs Off-policy

[8]Para esta explicación será necesario introducir dos conceptos a priori nuevos, pero que hemos estado usando hasta el momento inconscientemente, es decir, sin ponerle un nombre. Se trata de dos formas de dirigirse a las políticas que siguen nuestros algoritmos:

- Política de actualización: como el agente aprende la política óptima.
- Política de comportamiento: como el agente se comporta.

En *Q-learning* el agente aprende la política óptima usando la estrategia / política *greedy*, pero se comporta siguiendo otras políticas y por ello se le conoce como un algoritmo *Off-policy*.

En *SARSA* el proceso es similar para aprender y comportarse, es decir, aprende la política óptima y se comporta llevando a cabo la misma política como veremos en el desarrollo de esta sección. Por esta razón, *SARSA* es un algoritmo *On-policy*.

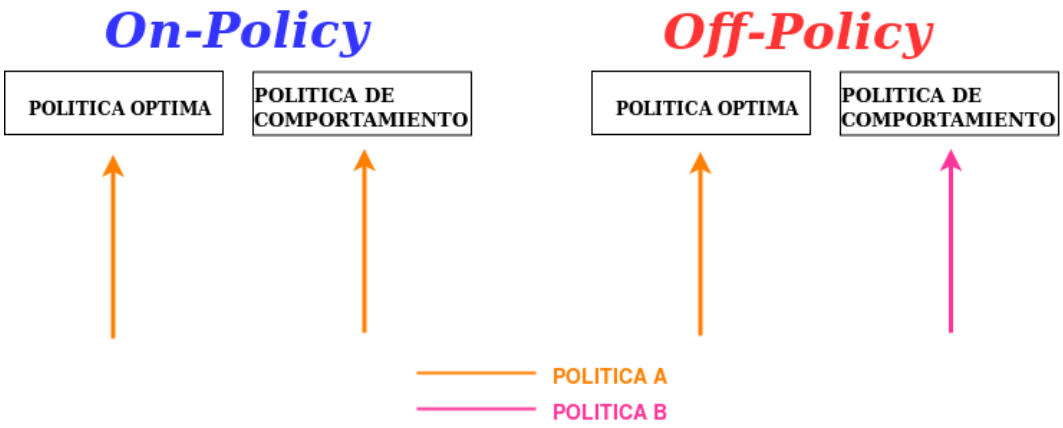

Figura 6.18.: Diagrama explicativo *On-Policy vs Off-Policy*

[8]Fuente: Mao, 2019

6.4.3. Actualización y desarrollo

No vamos a pararnos mucho en desarrollar el objetivo del algoritmo ya que se puede plantear de la misma forma que se hizo *Q-learning*. La única diferencia entre los dos algoritmos reside en el proceso de actualización de los *Q-values*.

Si en *Q-learning* teníamos esta ecuación:

$$q^{new}(s,a) = (1-\alpha)q(s,a) + \alpha(R_{t+1} + \gamma \max_{a'} q(s',a')) \tag{6.15}$$

En *SARSA* la expresión de actualización es:

$$q^{new}(s,a) = (1-\alpha)q(s,a) + \alpha(R_{t+1} + \gamma q(s_{t+1},a_{t+1})) \tag{6.16}$$

Como se ha comentado al tratarse de un algoritmo *On-Policy*, se lleva a cabo la misma política de actuación para aprender el valor óptimo que para elegir los pasos correspondientes dentro de la ejecución.

Esta política de actuación es la misma que se explicó en el apartado 6.3.3 donde se desarrollaba la estrategia *greedy*, que también es conocida en algunas bibliografías como $-$ *greedy*. En resumen el proceso seguido es el siguiente:

- Si *epsilon* es un valor grande, el valor r aleatorio dificilmente lo superará, teniendo que elegir una acción aleatoria dentro de todas las aceptadas por el *environment*. - *exploración*
- En cambio si *epsilon* es un valor pequeño, el valor aleatorio r fácilmente lo superará, actuando en función de los datos albergados en la *Q-table*, normalmente aquellos que maximizan la recompensa.

Como dato curioso y volviendo al porqué del nombre de este algoritmo podemos ver como, se leen las siglas *SARSA* fácilmente en la ecuación:

$$q^{new}(s,a) = (1-\alpha)q(s,a) + \alpha(R_{t+1} + \gamma q(s_{t+1},a_{t+1})) \tag{6.17}$$

6.4.4. Implementación en python

(Apéndice G)Para la implementación del algoritmo vamos a seguir un proceso muy sencillo ya que ésta no difiere mucho de la implementación de *Q-learning*. Al final las lineas que van a cambiar son pocas y eso se puede ver reflejado en la gran similitud entre los distintos pseudocódigos, aquí se presenta el del algoritmo en desarrollo en este punto:

Sarsa (on-policy TD control) for estimating $Q \approx q_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

Loop for each episode:
  Initialize S
  Choose A from S using policy derived from Q (e.g., ε-greedy)
  Loop 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

Figura 6.19.: Pseudocódigo algoritmo *SARSA*

Como el punto de actualización de los *Q-values* en la *Q-table* es el principal cambio en la implementación, se ha desarrollado una función que recibe como argumentos los 5 parámetros de *SARSA* y actualiza la tabla de valores correctamente de forma fiel al algoritmo.

Listing 6.5: Función actualización *Q-table* en *SARSA*

```
def learn(state, action, reward, state2, action2):
        predict = q_table[state, action]
        target = reward + gamma * q_table[state2, action2]
        q_table[state, action] = q_table[state, action] +
        lr_rate * (target - predict)
```

Se ha dividido la operación en tres líneas donde se crean dos variables (*predict y target*) con el valor antiguo y valor esperado -futuro- de la recompensa para el par estado-acción en cuestión.

6.4.5. Resultados e interpretación

Siguiendo la misma filosofía que se siguió en *Q-learning* por su parecido tipo tabular por albergar los datos en *arrays* o tablas y generando más episodios de entrenamiento que en *REINFORCE* ya que el proceso de actualización de los *Q-values* es lento, se definen las siguientes especificaciones con las que se ha programado el algoritmo y de las que dependerán nuestros resultados son (mismas que en *Q-learning*):

- Se ha trabajado con una población inicial de *tags* de 30 elementos.
- El valor de γ para el descuento en las recompensas se establece en 1.
- El número de episodios con el que se va a entrenar nuestro algoritmo con la interacción con el *environment* será de 3000, en principio.
- El *learning rate* α usado es de 0.2.
- El valor de *epsilon* empezará en 1 e irá disminuyendo con un *"decay_rate"* de 0.0008.

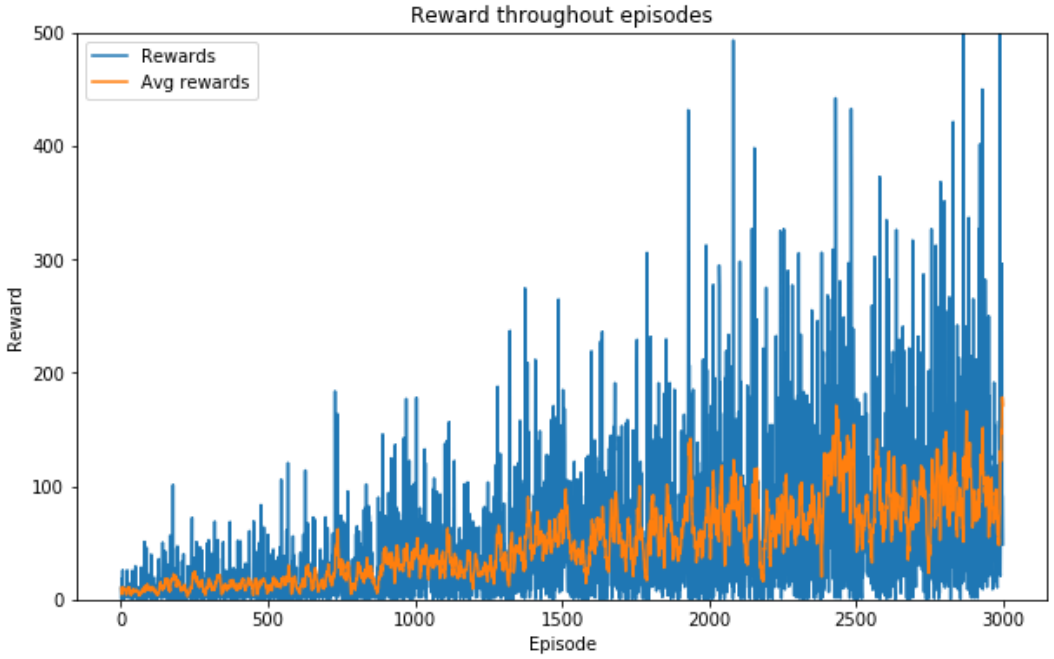

Figura 6.20.: Variación del *reward* en *SARSA*

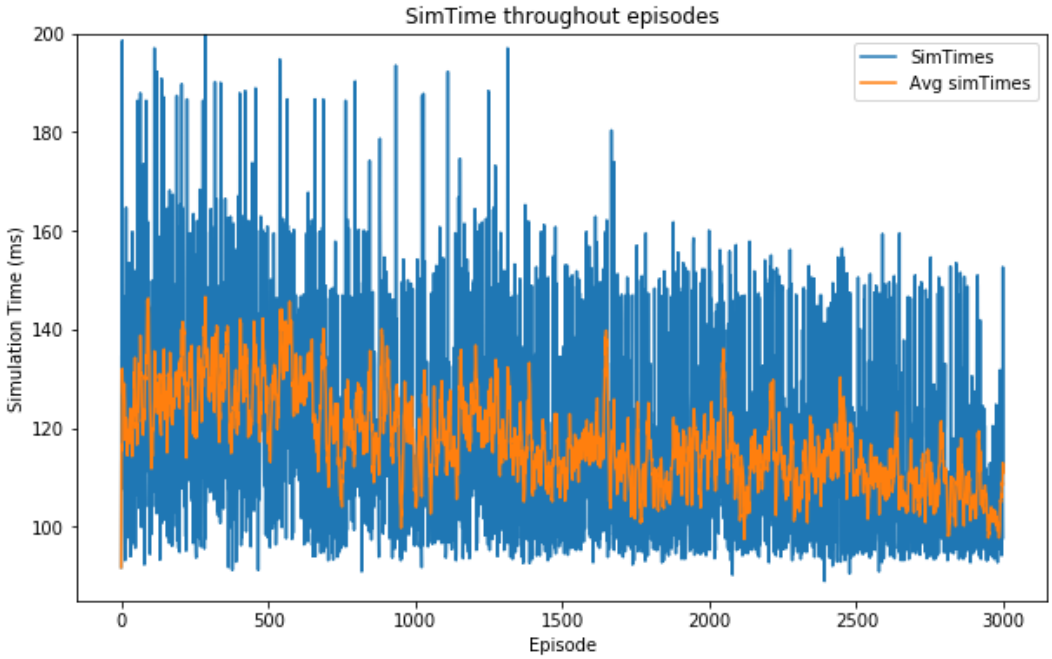

Figura 6.21.: Variación del *simTime* en *SARSA*

La interpretación de los resultados es la misma que se desarrolló para *Q-learning*, donde los más importante es ver el proceso en el que las recompensas aumentan y el tiempo de identificación, aunque de forma lenta, va disminuyendo.

7. Conclusión

A modo de cierre de este trabajo, cabe realizar una síntesis del proceso, con una valoración de resultados y posibles líneas de actuación en el futuro.

En los últimos capítulos referentes a los resultados, se ha intentado transmitir una pequeña valoración de ellos, aunque no es hasta ahora cuando podemos ver los tres algoritmos desde fuera. Como se ha comprobado, en el primero de ellos (*REINFORCE*), los resultados han sido bastante favorables, pudiendo sacar como conclusión que los algoritmos tipo *policy gradient* trabajan bien con nuestro entorno, aunque es pronto para decirlo ya que solo se ha implementado uno de todos los existentes.

Con respecto a los algoritmos tabulares decir que los resultados conseguidos no han sido tan buenos como se esperaba o al menos no se perciben de forma muy abrupta las mejoras temporales de identificación. Como se dijo en el resultado de *Q-learning*, el hecho de que nuestro entorno le dé al agente demasiada información puede provocar que el algoritmo no converja como se esperaba. En un caso en el que no tuvieramos una política óptima de referencia, seguramente el *reinforcement learning* desplegaría todo su potencial.

Y con esta discusión podríamos ligar y plantear aquí una posible línea de actuación futura en la que se modifique el entorno de trabajo de nuestro simulador de *tags* ofreciendo al agente una mayor incertidumbre de actuación teniendo así que entrenar durante más tiempo el algoritmo y aprendiendo más sobre el *environment*. De la misma forma podría desarrollarse la implantación real de algoritmos de este tipo en el mundo de la industria en incluso en el de los servicios. Todo ello iría precedido de un exhaustivo *"testbed"* donde se experimentaría con los algoritmos en un entorno parecido al real, fuera de los riesgos de hacer estas pruebas en un ambiente de producción.

7. Conclusión

Haciendo un balance general al trabajo realizado, se han tocado varios puntos donde por supuesto el principal y más importante ha sido entrar de lleno en el entendimiento, desarrollo e implementación de algoritmos de *reinforcement learning*, campo de la inteligencia artificial sobre el que se han presentado los principios básicos y se ha comprobado su potencial. Este potencial aún puede ser explotado en mayor medida de lo que se ha hecho y por ello volver a remarcar que las posibles líneas de actuación con las que se puede continuar este trabajo son muchas.

Finalmente y después de varias horas de trabajo podemos concluir este trabajo diciendo que ha sido un placer iniciarse en temas del *machine learning* de una forma tan práctica como esta y que se espera seguir indagando en el futuro.

Apéndice

Apéndice A.

Código simulador MATLAB

Listing A.1: Simulador de *tags* estáticos en MATLAB

```
function StaticTagSimulator
clear all;
clc;
%Simulator of a Single RFID Reader with static tags

%parameter definition
seed = 89456;
Initial_Population = 30;
Tidle = 0.21;
Tcollision = 0.49;
Tsuccess = 2.83;
Tquery = 0.25;
h = 2; %antenna height;
ptx = 2; %[power transmitted by the reader, W]
greader = 5; %[load-matched gain of the reader s antenna]
gtag = 1.6; %[load-matched gain of the tag s antenna]
lambda = 0.33; %[carrier-frequency wavelength, m]
X = 0.5;
M = 0.25;
Q = 5;
B = 1;
%thr = 10^(-8.3)/1000;
%tht = 10^(-1.1)/1000;
thr = 0.0;
tht = 0.0;
Kb = (ptx*greader*greader*gtag*gtag*(lambda^4)*(X^2)*M)/((4*pi())
    ^4*Q*Q*B*B);
%distance not included yet [POWER RECEIVED AT THE READER]
Kp = (ptx*greader*gtag*(lambda^2)*X*M)/((4*pi())^2*Q*B);
```

```
%distance not included yet [POWER RECEIVED AT THE TAG]
K = 1;
sigma = 3;
ro = 0.9;
c = 10^0.6;
medias = sigma*sqrt(K/2)*ones(1,4);
cov   = ro*sigma*sigma/2;
SIGMA = [sigma*sigma/2 0 cov 0; 0 sigma*sigma/2 0 cov; cov 0 sigma*sigma/2 0; -2 cov 0 sigma*sigma/2];

referencepolicy = ones(Initial_Population,1);
for i=1:Initial_Population
    if (i==1)
        referencepolicy(i) = 1;
    elseif ((1<i)&&(i<=3))
        referencepolicy(i) = 2;
    elseif ((3<i)&&(i<=6))
        referencepolicy(i) = 3;
    elseif ((6<i)&&(i<=11))
        referencepolicy(i) = 4;
    elseif ((11<i)&&(i<=22))
        referencepolicy(i) = 5;
    elseif ((22<i)&&(i<=44))
        referencepolicy(i) = 6;
    elseif ((44<i)&&(i<=89))
        referencepolicy(i) = 7;
    elseif (89<i)
        referencepolicy(i) = 8;
    end
end

Unident = Initial_Population;

rand('twister', seed); %random number generator

for j = 1:Initial_Population
        TagList(j).id = false; %is not identified yet
        TagList(j).N  = 0;
end

Jestimated = zeros(1,Initial_Population);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%simulation begins   %%%
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%
simTime = Tquery;

while Unident>0;

    ElapsedTime = 0;

    FrameLength = 2^(referencepolicy(Unident)-1); %set frame length according to policy
    Frame = cell(1,FrameLength);
    SelectedSlotsList = [];

    for j = 1:Initial_Population
        IsId      = TagList(j).id;
        if (~IsId) %if not identified, select a slot for next frame
            SelectedSlot   = ceil(FrameLength.*rand) %randomly selected slot
            SelectedSlotsList = [SelectedSlotsList, SelectedSlot]
            Frame{SelectedSlot} = [Frame{SelectedSlot}, j] %tag j has selected "SelectedSlot"
        end
    end

    SelectedSlotsList = sort(SelectedSlotsList);
    SelectedSlotsList = unique(SelectedSlotsList);
    Events            = length(SelectedSlotsList);
    PreviousSlot      = 0;

    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    %%%%decide which tags are identified %%%
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

    for e = 1:Events;
        SelectedSlot     = SelectedSlotsList(e);

        ElapsedTime      = ElapsedTime + (SelectedSlot - PreviousSlot - 1)*Tidle; %time elapsed since previous slot

        CompetingTags    = Frame{SelectedSlot};  %tags competing in this slot

        totalCompeting  = length(CompetingTags);
```

```
109
110         if (totalCompeting==0)
111             ElapsedTime = ElapsedTime + Tidle;  %no tag in this
                    slot
112         else %detection errors and capture effect
113             Pb = [];
114             PoweredCompetitors = [];
115             for(coll = 1:totalCompeting) %Generate all
                    backscattered packets
116                 distance = h;
117                 r = mvnrnd(medias,SIGMA);
118                 X = r(1);
119                 Y = r(2);
120                 W = r(3);
121                 V = r(4);
122                 RICEAN = sqrt(X*X + Y*Y);
123                 PRODUCTRICEAN = sqrt((X*W-Y*V)^2+(X*V+Y*W)^2);
124                 PowerUp = Kp*RICEAN^2/distance^2; %Power at the
                        tag
125                 if (PowerUp>tht) %only signals detected at the tag
                        are backsacattered
126                     BackscatterP = Kb*PRODUCTRICEAN^2/distance^4;
127                     Pb = [Pb BackscatterP];
128                     PoweredCompetitors = [PoweredCompetitors
                            CompetingTags(coll)];
129                 end
130             end
131             answers = length(Pb);
132             if (answers>0) %We have one or more responses
133                 [maxPb index]= max(Pb);
134                 if (maxPb > thr) %At least one is above the reader
                        threshold
135                     if (answers==1) %only one response --> tag
                            identified
136                         ElapsedTime = ElapsedTime + Tsuccess;
137                         TagList(PoweredCompetitors(index)).id =
                                true;    %IDENTIFIED TAG
138                         Unident = Unident-1;  %Decrease
                                unidentified count
139                     else  %several responses: check capture effect
140                         Pb = Pb(Pb<maxPb);
141                         Noise = sum(Pb);
142                         if (maxPb/Noise > c) %CAPTURE !!
143                             ElapsedTime = ElapsedTime + Tsuccess;
```

```
144                                 TagList(PoweredCompetitors(index)).id
                                        = true;    %IDENTIFIED TAG
145                                 Unident = Unident-1;  %Decrease
                                        unidentified count
146                             else %COLLISION
147                                 ElapsedTime = ElapsedTime + Tcollision
                                        ; %collided
148                             end
149                         end
150                     else %NO ONE is above the reader threshold
151                         ElapsedTime = ElapsedTime + Tidle;  %no tag in
                                this slot
152                     end
153                 else %NO ONE is above the tag threshold
154                     ElapsedTime = ElapsedTime + Tidle;  %no tag in
                            this slot
155                 end
156             end
157             PreviousSlot = SelectedSlot; %update PreviousSlot
158         end
159
160         ElapsedTime       = ElapsedTime + max((FrameLength -
                PreviousSlot),0)*Tidle; %time elapsed since the last
                selected slot
161
162         %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
163         %%%update data structures %%%%
164         %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
165
166         simTime = simTime + ElapsedTime;
167
168         if ((Unident<Initial_Population)&&(Unident>0))
169             if (Jestimated(Unident) == 0) Jestimated(Unident) = simTime
                    ;
170             end
171         end
172     end
173
174     Jestimated(find(Jestimated~=0)) = simTime-Jestimated(find(
            Jestimated~=0));
175     Jestimated(Initial_Population) = simTime
```

Apéndice B.

Código simulador PYTHON

Listing B.1: Simulador de *tags* estáticos en PYTHON

```
"""
SIMULATOR OF A SINGLE RFID READER WITH STATIC TAGS

@author: Sergio Moreno Lorente
"""
import numpy
import random

seed = 89456

# PARAMETER DEFINITION

Initial_Population = 30
Tidle = 0.21
Tcollision = 0.49
Tsuccess = 2.83
Tquery = 0.25

h = 2 #antenna height
ptx = 2 #[power tranasmitted by the reader, W]
greader = 5 #[load-matched gain of the reader's antenna]
gtag = 1.6 #[load-matched gain of the tag's antenna]
wave_length = 0.33 #[carrier-frequency wavelength, m]
X = 0.5
M = 0.25
Q = 5
B = 1
thr = 0.0
tht = 0.0
```

```
30
31 #[power received at the reader]
32 Kb = (ptx*(greader**2)*(gtag**2)*(wave_length**4)*(X**2)*M) /(((4*
       numpy.pi)**4)*(Q**2)*(B**2))
33
34 #[power received at the tag]
35 Kp = (ptx*greader*gtag*(wave_length**2)*X*M) /(((4*numpy.pi)**2)*Q*
       B)
36
37 K = 1
38 sigma = 3
39 ro = 0.9
40 c = 10**0.6
41 medias = sigma*numpy.sqrt(K/2)*numpy.ones(4)
42 cov = ro*sigma*sigma/2
43 SIGMA = [[sigma*sigma/2, 0, cov, 0], [0, sigma*sigma/2, 0, cov], [
       cov, 0, sigma*sigma/2, 0], [-2, cov, 0, sigma*sigma/2]]
44
45 # REFERENCE POLICY
46
47 referencepolicy = list(range(Initial_Population))
48
49 for x in referencepolicy:
50     if x == 0:
51         referencepolicy[x] = 1
52     elif ((0<x)and(x<=2)):
53         referencepolicy[x] = 2
54     elif ((2<x)and(x<=5)):
55         referencepolicy[x] = 3
56     elif ((5<x)and(x<=10)):
57         referencepolicy[x] = 4
58     elif ((10<x)and(x<=21)):
59         referencepolicy[x] = 5
60     elif ((21<x)and(x<=43)):
61         referencepolicy[x] = 6
62     elif ((43<x)and(x<=88)):
63         referencepolicy[x] = 7
64     elif (88<x):
65         referencepolicy[x] = 8
66
67 #referencepolicy = tuple(referencepolicy)
68
69 Unident = Initial_Population-1
70
```

```
71 TagList = list(range(Initial_Population))
72
73 for x in TagList:
74     TagList[x] = False
75
76 Jestimated = numpy.zeros(Initial_Population)
77
78 ##########################################
79 ############ SIMULATION BEGINS ############
80 ##########################################
81
82 simTime = Tquery
83
84 while Unident >= 0:
85
86     ElapsedTime = 0
87
88     #set frame length according to policy
89     FrameLength = 2**(referencepolicy[Unident]-1)
90     Frame = [ [] for i in range(FrameLength) ]
91
92     SelectedSlotsList = []
93
94     for x in range(Initial_Population):
95         IsId = TagList[x]
96         if (IsId == False): #if not identified, selected a slot
               for next frame
97             SelectedSlot = random.randint(0,FrameLength-1) #
                  randomly selected slot
98             SelectedSlotsList.append(SelectedSlot)
99             Frame[SelectedSlot].append(x)
100
101    SelectedSlotsList = numpy.sort(SelectedSlotsList)
102    SelectedSlotsList = numpy.unique(SelectedSlotsList)
103    Events = len(SelectedSlotsList)
104    PreviousSlot = 0
105
106    ##########################################################
107    ############ decide which tags are identified ############
108    ##########################################################
109
110    for e in range(Events):
111        SelectedSlot = SelectedSlotsList[e]
112
```

```
113             ElapsedTime = ElapsedTime + (SelectedSlot - PreviousSlot -
                    1)*Tidle #time elapsed since previous slot
114
115             CompetingTags = Frame[SelectedSlot] #tags competing in
                    this slot
116
117             totalCompeting = len(CompetingTags)
118
119             if totalCompeting == 0:
120                 ElapsedTime = ElapsedTime + Tidle #no tag in this
                        slot
121             else: #detection error and capture effect
122                 Pb = []
123                 PoweredCompetitors = []
124                 for coll in range(totalCompeting):
125                     distance = h
126                     r = numpy.random.multivariate_normal(medias, SIGMA
                            , check_valid='ignore')
127                     X = r[0]
128                     Y = r[1]
129                     W = r[2]
130                     V = r[3]
131                     RICEAN = numpy.sqrt(X*X + Y*Y)
132                     PRODUCTRICEAN = numpy.sqrt((((X*W-Y*V)**(2))+((X*V
                            +Y*W)**(2))))
133                     PowerUp = Kp*(RICEAN**2)/(distance**2) #power at
                            the tag
134                     if PowerUp > tht: #only signals detected at the
                            tag are backscattered
135                         BackscatterP = Kb*(PRODUCTRICEAN**2)/(distance
                                **4)
136                         Pb.append(BackscatterP)
137                         PoweredCompetitors.append(CompetingTags[coll])
138
139                 answers = len(Pb)
140                 if answers > 0: #we have one or more responses
141                     maxPb = max(Pb)
142                     index = Pb.index(maxPb)
143                     if (maxPb > thr): #at least one is above the
                            reader threshold
144                         if (answers == 1): #only one response --> tag
                                identified
145                             ElapsedTime = ElapsedTime + Tsuccess
```

```
146                             TagList[PoweredCompetitors[index]] = True
                                    #IDENTIFIED TAG
147                             Unident = Unident - 1 #Decrease
                                    unidentified count
148                         else: #several responses: check capture effect
149                             Pb.remove(maxPb)
150                             Noise = sum(Pb)
151                             if ((maxPb/Noise) > c): #CAPTURE!!
152                                 ElapsedTime = ElapsedTime + Tsuccess
153                                 TagList[PoweredCompetitors[index]] =
                                        True #IDENTIFIED TAG
154                                 Unident = Unident - 1 #Decrease
                                        unidentified count
155                             else: #COLLISION
156                                 ElapsedTime = ElapsedTime + Tcollision
                                        #collided
157                     else: #NO ONE is above the reader threshold
158                         ElapsedTime = ElapsedTime + Tidle #no tag in
                                this slot
159                 else: #NO ONE is above the tag threshold
160                     ElapsedTime = ElapsedTime + Tidle #no tag in this
                            slot
161
162             PreviousSlot = SelectedSlot #update PreviousSlot
163
164         ElapsedTime = ElapsedTime + max((FrameLength - PreviousSlot),
                0)*Tidle # time elapsed since the last selected slot
165
166         ##########################################
167         ######### update data structures #########
168         ##########################################
169
170         simTime = simTime + ElapsedTime
171
172         if (Unident < Initial_Population) and (Unident > 0):
173             if (Jestimated[Unident] == 0):
174                 Jestimated[Unident] = simTime
175
176     for x in range(len(Jestimated)):
177         if (Jestimated[x] != 0):
178             Jestimated[x] = simTime - Jestimated[x]
179
180     Jestimated[Initial_Population - 1] = simTime
181     print("\n"+str(Jestimated)+"\n")
```

Apéndice C.

Efecto captura

Cuando dos señales en una misma frecuencia son procesadas a través de un receptor de frecuencia modulada, solo la señal con mayor amplitud es escuchada por el receptor. Este fenómeno puede compararse con ir en coche escuchando la radio y que a medida que nos separamos de una estación de radio la señal nos llega con menos potencia hasta que finalmente hay otra señal que supera esta amplitud y se cambia de estación o emisora de radio automáticamente.[1]

Por lo tanto, el efecto captura se produce normalmente entre una señal de interés y otra señal competidora. Trasladando este fenómeno a nuestro simulador, veremos como cuando se produce colisión entre 2 o más señales aún será posible identificar alguna de ellas si llega con suficiente amplitud como para que sea notablemente más potente que el resto. Este fenómeno ha sido implementado en nuestro simulador de forma que el cociente entre nuestra señal de interés (la más potente) y el resto de señales competidoras sea mayor que un parámetro fijo dado "c", es decir, si por ejemplo tenemos tres señales de *tags* compitiendo por ser identificadas por el *lector*, se procederá excluyendo la más potente y sumando las otras dos (ruido). Si el cociente entre nuestra señal de interés y el ruido es mayor que el parámetro de captura "c", el *tag* será identificado.

[1] Fuente: Milne, 2014

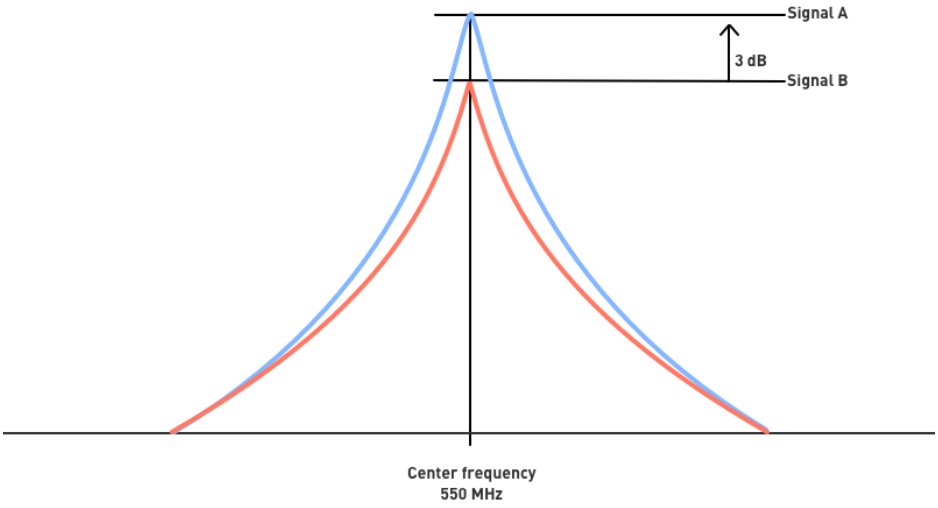

Figura C.1.: Efecto captura entre una señal de interés y otra competidora

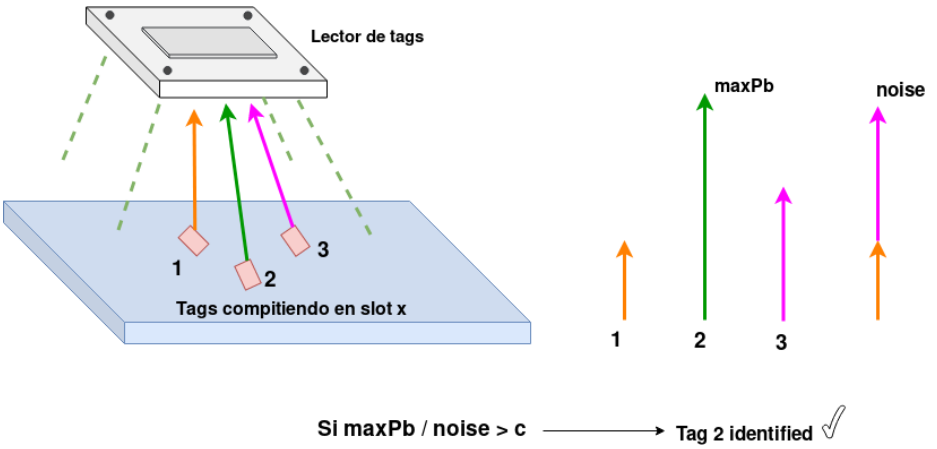

Figura C.2.: Esquema implementación efecto captura

Apéndice D.

Código environment PYTHON + OpenAI-Gym

Listing D.1: *Environment OpenAI-Gym* del simulador de *tags*

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""

@author: sergioml97

SIMULATOR OF A SINGLE RFID READER WITH STATIC TAGS
"""

import gym
import gym_staticTagSim
from gym import spaces, logger
from gym.utils import seeding
import numpy as np

class StaticTagSim(gym.Env):
    """
    Description:
        A quantity of tags (Initial_Population) are distributed in
            different slot inside a frame to be identified by a
            reader. The reader decides what is the frame length to
            minimize the total simulation time. The target is to
            identify all the tags.

    Source:
```

```
22          This environment correspond to the version of the "Static
                tag simulation" described by J. J. Alcar z Esp n in
                Matlab.
23
24      Observation:
25          Type: Box(3)
26          Num Observation                 Min                 Max
27          0   Simulation time             0                   Inf
28          1   Unidentified tags           0           Initial_Population
29          2   Frame number                0                   Inf
30
31      Actions:
32          Type: Discrete(8)
33          Num Action
34          0   Frame length = 2 slots
35          1   Frame length = 4 slots
36          2   Frame length = 8 slots
37          3   Frame length = 16 slots
38          4   Frame length = 32 slots
39          5   Frame length = 64 slots
40          6   Frame length = 128 slots
41          7   Frame length = 256 slots
42
43      Reward:
44          Reward is IdentifiedTags/FrameTime for every step taken.
45          or
46          Reward is -FrameTime for every step taken.
47
48      Starting State:
49          Simulation time = 0
50          Unidentified tags = Initial_Population
51          Frame number = 0
52
53      Episode termination:
54          All the tags have been correctly identified
55          Episode length is greater than 100 steps
56
57
58      """
59      metadata = {
60          'render.modes': ['human', 'rgb_array'],
61      }
62
63      def __init__(self):
```

```
64          # PARAMETER DEFINITION
65          self.Initial_Population = 30
66          self.frame_lengths = [2, 4, 8, 16, 32, 64, 128, 256]
67          self.Tidle = 0.21
68          self.Tcollision = 0.49
69          self.Tsuccess = 2.83
70          self.Tquery = 0.25
71          self.h = 2 #antenna height
72          self.ptx = 2 #[power tranasmitted by the reader, W]
73          self.greader = 5 #[load-matched gain of the reader's
                antenna]
74          self.gtag = 1.6 #[load-matched gain of the tag's antenna]
75          self.wave_length = 0.33 #[carrier-frequency wavelength, m]
76          self.X = 0.5
77          self.M = 0.25
78          self.Q = 5
79          self.B = 1
80          self.thr = 0.0
81          self.tht = 0.0
82          #[power received at the reader]
83          self.Kb = (self.ptx*(self.greader**2)*(self.gtag**2)*(self
                .wave_length**4)*(self.X**2)*self.M)/(((4*np.pi)**4)*(
                self.Q**2)*(self.B**2))
84
85          #[power received at the tag]
86          self.Kp = (self.ptx*self.greader*self.gtag*(self.
                wave_length**2)*self.X*self.M)/(((4*np.pi)**2)*self.Q*
                self.B)
87
88          self.K = 1
89          self.sigma = 3
90          self.ro = 0.9
91          self.c = 10**0.6
92          self.medias = self.sigma*np.sqrt(self.K/2)*np.ones(4)
93          self.cov = self.ro*self.sigma*self.sigma/2
94          self.SIGMA_vector = [[self.sigma*self.sigma/2, 0, self.cov
                , 0],[0, self.sigma*self.sigma/2, 0, self.cov],[self.
                cov, 0, self.sigma*self.sigma/2, 0],[-2, self.cov, 0,
                self.sigma*self.sigma/2]]
95
96
97          self.TagList = list(range(self.Initial_Population))
98
99          for x in self.TagList:
```

```
100             self.TagList[x] = 0
101
102         self.action_space = spaces.Discrete(8)
103         observation_min = np.array([0, 0, 0])
104         observation_max = np.array([200, self.Initial_Population,
                200])
105         self.observation_space = spaces.Box(observation_min,
                observation_max, dtype=np.float32)
106         self.seed()
107         self.SimTime = self.Tquery
108         self.Unident = self.Initial_Population-1
109         self.Frame_number = 0
110         self.state = None
111
112     def seed(self, seed = None):
113         self.np_random, seed = seeding.np_random(seed)
114         return [seed]
115
116     def step(self, action):
117         FrameLength = self.frame_lengths[action]
118         ElapsedTime = 0
119         #set frame length according to policy
120         self.Frame = [ [] for i in range(FrameLength) ]
121         SelectedSlotsList = []
122
123         for x in range(self.Initial_Population):
124             IsId = self.TagList[x]
125             if (IsId == 0): #if not identified, selected a slot
                    for next frame
126                 SelectedSlot = self.np_random.randint(0,
                        FrameLength) #randomly selected slot
127                 SelectedSlotsList.append(SelectedSlot)
128                 self.Frame[SelectedSlot].append(x)
129
130         SelectedSlotsList = np.sort(SelectedSlotsList)
131         SelectedSlotsList = np.unique(SelectedSlotsList)
132         Events = len(SelectedSlotsList)
133         PreviousSlot = 0
134
135         for e in range(Events):
136             SelectedSlot = SelectedSlotsList[e]
137
138             ElapsedTime = ElapsedTime + (SelectedSlot -
                    PreviousSlot - 1)*self.Tidle #time elapsed since
```

```
            previous slot
139
140         CompetingTags = self.Frame[SelectedSlot] #tags
                competing in this slot
141
142         totalCompeting = len(CompetingTags)
143
144         if totalCompeting == 0:
145             ElapsedTime = ElapsedTime + self.Tidle #no tag in
                    this slot
146         else: #detection error and capture effect
147             Pb = []
148             PoweredCompetitors = []
149             for coll in range(totalCompeting):
150                 distance = self.h
151                 r = np.random.multivariate_normal(self.medias,
                        self.SIGMA_vector, check_valid='ignore')
152                 X = r[0]
153                 Y = r[1]
154                 W = r[2]
155                 V = r[3]
156                 RICEAN = np.sqrt(X*X + Y*Y)
157                 PRODUCTRICEAN = np.sqrt((((X*W-Y*V)**(2))+((X*
                        V+Y*W)**(2))))
158                 PowerUp = self.Kp*(RICEAN**2)/(distance**2) #
                        power at the tag
159                 if PowerUp > self.tht: #only signals detected
                        at the tag are backscattered
160                     BackscatterP = self.Kb*(PRODUCTRICEAN**2)
                            /(distance**4)
161                     Pb.append(BackscatterP)
162                     PoweredCompetitors.append(CompetingTags[
                            coll])
163
164             answers = len(Pb)
165             if answers > 0: #we have one or more responses
166                 maxPb = max(Pb)
167                 index = Pb.index(maxPb)
168                 if (maxPb > self.thr): #at least one is above
                        the reader threshold
169                     if (answers == 1): #only one response -->
                            tag identified
170                         ElapsedTime = ElapsedTime + self.
                                Tsuccess
```

```
171                             self.TagList[PoweredCompetitors[index
                                    ]] = 1 #IDENTIFIED TAG
172                             self.Unident = self.Unident - 1 #
                                    Decrease unidentified count
173                         else: #several responses: check capture
                                effect
174                             Pb.remove(maxPb)
175                             Noise = sum(Pb)
176                             if ((maxPb/Noise) > self.c): #CAPTURE
                                    !!
177                                 ElapsedTime = ElapsedTime + self.
                                        Tsuccess
178                                 self.TagList[PoweredCompetitors[
                                        index]] = 1 #IDENTIFIED TAG
179                                 self.Unident = self.Unident - 1 #
                                        Decrease unidentified count
180                             else: #COLLISION
181                                 ElapsedTime = ElapsedTime + self.
                                        Tcollision #collided
182                     else: #NO ONE is above the reader threshold
183                         ElapsedTime = ElapsedTime + self.Tidle #no
                                tag in this slot
184                 else: #NO ONE is above the tag threshold
185                     ElapsedTime = ElapsedTime + self.Tidle #no tag
                            in this slot
186
187             PreviousSlot = SelectedSlot #update PreviousSlot
188
189         ElapsedTime = ElapsedTime + max((FrameLength -
                PreviousSlot), 0)*self.Tidle # time elapsed since the
                last selected slot
190         Identified = self.Initial_Population-self.Unident
191         self.Frame_number = self.Frame_number + 1
192         reward = Identified/ElapsedTime
193         self.SimTime = self.SimTime + ElapsedTime
194         self.state = (self.SimTime, self.Unident, self.
                Frame_number)
195         if (self.Unident == -1):
196             done = True
197         else:
198             done = False
199
200         return np.array(self.state), reward, done, {}
201
```

```
202     def reset(self):
203         self.TagList = list(range(self.Initial_Population))
204         for x in self.TagList:
205             self.TagList[x] = 0
206         self.SimTime = self.Tquery
207         self.Unident = self.Initial_Population-1
208         self.Frame_number = 0
209         self.state = (self.SimTime, self.Unident, self.
                Frame_number)
210         return np.array(self.state)
211 
212     def render(self, mode = 'human'):
213         print("SIMTIME: " + str(self.SimTime))
214         print("UNIDENT: " + str(self.Unident))
```

Apéndice E.

Código algoritmo REINFORCE policy gradient

Listing E.1: Algoritmo *REINFORCE policy gradient* para el simulador de *tags*

```
1  import sys
2  import torch
3  import time
4  import gym
5  import gym_staticTagSim
6  import numpy as np
7  import torch.nn as nn
8  import torch.optim as optim
9  import torch.nn.functional as F
10 from torch.autograd import Variable
11 import matplotlib.pyplot as plt
12
13 # Constants
14 GAMMA = 1
15
16 class PolicyNetwork(nn.Module):
17     def __init__(self, num_inputs, num_actions, hidden_size,
       learning_rate=1e-5):#3e-4):
18         super(PolicyNetwork, self).__init__()
19
20         self.num_actions = num_actions
21         self.linear1 = nn.Linear(num_inputs, hidden_size)
22         time.sleep(2)
23         self.linear2 = nn.Linear(hidden_size, num_actions)
24         self.optimizer = optim.Adam(self.parameters(), lr=
           learning_rate)
```

```
25
26      def forward(self, state):
27          x = F.relu(self.linear1(state))
28          x = F.softmax(self.linear2(x), dim=1)
29          return x
30
31      def get_action(self, state):
32          state = torch.from_numpy(state).float().unsqueeze(0)
33          probs = self.forward(Variable(state))
34          highest_prob_action = np.random.choice(self.num_actions, p
                =np.squeeze(probs.detach().numpy()))
35          log_prob = torch.log(probs.squeeze(0)[highest_prob_action
                ])
36          return highest_prob_action, log_prob
37
38  def update_policy(policy_network, rewards, log_probs):
39      discounted_rewards = []
40
41      for t in range(len(rewards)):
42          Gt = 0
43          pw = 0
44          for r in rewards[t:]:
45              Gt = Gt + GAMMA**pw * r
46              pw = pw + 1
47          discounted_rewards.append(Gt)
48
49      discounted_rewards = torch.tensor(discounted_rewards)
50
51      policy_gradient = []
52      for log_prob, Gt in zip(log_probs, discounted_rewards):
53          policy_gradient.append(-log_prob * Gt)
54
55      policy_network.optimizer.zero_grad()
56      policy_gradient = torch.stack(policy_gradient).sum()
57      policy_gradient.backward()
58      policy_network.optimizer.step()
59
60  env = gym.make('StaticTagSim-v0')
61  policy_net = PolicyNetwork(env.observation_space.shape[0], env.
        action_space.n, 128)
62
63  max_episode_num = 2000
64  max_steps = 500
65  numsteps = []
```

```
66 simTimes = []
67 avg_simtime = []
68 avg_numsteps = []
69 all_rewards = []
70 avg_reward = []
71
72 for episode in range(max_episode_num):
73     state = env.reset()
74     log_probs = []
75     rewards = []
76
77     for steps in range(max_steps):
78         action, log_prob = policy_net.get_action(state)
79         new_state, reward, done, _ = env.step(action)
80         log_probs.append(log_prob)
81         rewards.append(reward)
82
83         if done:
84             simTimes.append(new_state[0])
85             update_policy(policy_net, rewards, log_probs)
86             numsteps.append(steps)
87             avg_numsteps.append(np.mean(numsteps[-10:]))
88             avg_simtime.append(np.mean(simTimes[-10:]))
89             all_rewards.append(np.sum(rewards))
90             avg_reward.append(np.mean(all_rewards[-10:]))
91             if episode %1 == 0:
92                 sys.stdout.write("episode: {}, length: {}, simTime
                    : {}\n"
93                                  .format(episode, steps, new_state
                                     [0]))
94             break
95
96         state = new_state
```

Apéndice F.

Código algoritmo Q-learning

Listing F.1: Algoritmo *Q-learning* para el simulador de *tags*

```
import gym
import numpy as np
import random
from IPython.display import clear_output
import gym_staticTagSim
import matplotlib.pyplot as plt

env = gym.make("StaticTagSim-v0")

action_space_size = env.action_space.n
state_space_size = int(env.observation_space.high[1])

q_table = np.zeros((state_space_size, action_space_size))

num_episodes = 3000
max_steps_per_episode = 500

exploration_rate = 1
max_exploration_rate = 1
min_exploration_rate = 0.01

learning_rate = 0.2
discount_rate = 1
exploration_decay_rate = 0.0008

rewards_all_episodes = []
avg_reward = []
simTime = 0
simTimes = []
```

```
avg_simtime = []

# Q-learning algorithm
for episode in range(num_episodes):
    # initialize new episode params
    simTime, Unident, Frame_number = env.reset()
    done = False
    rewards_current_episode = 0
    for step in range(max_steps_per_episode):
        # Exploration-exploitation trade-off
        exploration_rate_threshold = random.uniform(0, 1)
        if exploration_rate_threshold > exploration_rate:
            action = np.argmax(q_table[int(Unident),:])
        else:
            action = env.action_space.sample()
        # Take new action
        new_state, reward, done, info = env.step(action)
        state = int(new_state[1])
        # Update Q-table for Q(s,a)
        q_table[int(Unident), action] = q_table[int(Unident),
            action] * (1 - learning_rate) + learning_rate * (reward
             + discount_rate * np.max(q_table[state, :]))
        # Set new state
        Unident = int(new_state[1])
        Frame_number = int(new_state[2])
        # Add new reward
        rewards_current_episode += reward

        if done == True:
            simTime = new_state[0]
            print("simTime for episode " + str(episode+1) + ": " +
                 str(simTime) + " Number of frames: " + str(step+1)
                 + "  Exp. rate: " + str(exploration_rate))
            simTimes.append(simTime)
            avg_reward.append(np.mean(rewards_all_episodes[-10:]))
            avg_simtime.append(np.mean(simTimes[-10:]))
            break

    # Exploration rate decay
    exploration_rate = min_exploration_rate + (
        max_exploration_rate - min_exploration_rate) * np.exp(-
        exploration_decay_rate*episode)
    # Add current episode reward to total rewards list
    rewards_all_episodes.append(rewards_current_episode)
```

Apéndice G.

Código algoritmo SARSA

Listing G.1: Algoritmo *SARSA* para el simulador de *tags*

```
import gym
import gym_staticTagSim
import numpy as np
from IPython.display import clear_output
import matplotlib.pyplot as plt

env = gym.make("StaticTagSim-v0")

action_space_size = env.action_space.n
state_space_size = int(env.observation_space.high[1])

q_table = np.zeros((state_space_size, action_space_size))

num_episodes = 3000
max_steps_per_episode = 1000

exploration_rate = 1
max_exploration_rate = 1
min_exploration_rate = 0.001

lr_rate = 0.1
gamma = 1
exploration_decay_rate = 0.0008

rewards_all_episodes = []

rewards_all_episodes = []
avg_reward = []
simTime = 0
```

```
30 simTimes = []
31 avg_simtime = []
32
33 def choose_action(state):
34         action=0
35         if np.random.uniform(0, 1) < exploration_rate:
36                 action = env.action_space.sample()
37         else:
38                 action = np.argmax(q_table[state, :])
39         return action
40
41 def learn(state, action, reward, state2, action2):
42         predict = q_table[state, action]
43         target = reward + gamma * q_table[state2, action2]
44         q_table[state, action] = q_table[state, action] + lr_rate
               * (target - predict)
45
46
47 # SARSA algorithm
48 for episode in range(num_episodes):
49     # initialize new episode params
50     simTime, Unident, Frame_number = env.reset()
51     state = int(Unident)
52     action = choose_action(state)
53     done = False
54     rewards_current_episode = 0
55     for step in range(max_steps_per_episode):
56         # Take new action
57         new_state, reward, done, info = env.step(action)
58         state2 = int(new_state[1])
59         action2 = choose_action(state2)
60         learn(state, action, reward, state2, action2)
61         state = state2
62         action = action2
63         Unident = int(new_state[1])
64         Frame_number = int(new_state[2])
65         # Add new reward
66         rewards_current_episode += reward
67
68         if done == True:
69             simTime = new_state[0]
70             print("simTime for episode " + str(episode+1) + ": " +
                   str(simTime) + " Number of frames: " + str(step+1)
                   + "  Exp. rate: " + str(exploration_rate))
```

```
71              simTimes.append(simTime)
72              avg_reward.append(np.mean(rewards_all_episodes[-10:]))
73              avg_simtime.append(np.mean(simTimes[-10:]))
74              break
75
76      # Exploration rate decay
77      exploration_rate = min_exploration_rate + (
            max_exploration_rate - min_exploration_rate) * np.exp(-
            exploration_decay_rate*episode)
78      # Add current episode reward to total rewards list
79      rewards_all_episodes.append(rewards_current_episode)
```

Bibliografía

Alcaraz, Juan J y col. (2013a). «A stochastic shortest path model to minimize the reading time in DFSA-based RFID systems». En: *IEEE Communications Letters* 17.2, págs. 341-344 (vid. págs. 11, 15, 20, 26, 47, 51).

Alcaraz, Juan J y col. (2013b). «RFID Reader Scheduling for Reliable Identificationof Moving Tags». En: *IEEE Communications Letters* 10.3, págs. 816-828 (vid. pág. 18).

Anaconda (2019). *Anaconda Documentation*. URL: https://www.digitalocean.com/community/tutorials/how-to-install-anaconda-on-ubuntu-18-04-quickstart (vid. pág. 43).

(GitHub), OpenAI-Gym (2019). *OpenAI Gym documentation - GitHub*. URL: https://github.com/openai/gym (vid. pág. 43).

GS1, The Global Language of Business (2018). *EPC^{TM} Radio-Frequency Identity Protocols Generation-2 UHF RFID Standard*. URL: https://www.gs1.org/sites/default/files/docs/epc/gs1-epc-gen2v2-uhf-airinterface_i21_r_2018-09-04.pdf (vid. págs. 2, 5).

Hui, Jonathan (2018). «RL — Policy Gradient Explained». En: URL: https://medium.com/@jonathan_hui/rl-policy-gradients-explained-9b13b688b146 (vid. pág. 52).

Kansal, Satwik y Brendan Martin (2018). *Reinforcement Q-Learning from Scratch in Python with OpenAI Gym*. URL: https://learndatasci.com/tutorials/reinforcement-q-learning-scratch-python-openai-gym/ (vid. pág. 42).

Mao, Lei (2019). «On-Policy VS Off-Policy in Reinforcement Learning». En: URL: https://leimao.github.io/blog/RL-On-Policy-VS-Off-Policy/ (vid. pág. 77).

Milne, Alex (2014). *Making use of the capture effect*. URL: https://www.rfvenue.com/blog/2014/12/15/making-use-of-the-capture-effect (vid. pág. 95).

OpenAI (2019). *OpenAI documentation*. URL: https://openai.com/ (vid. pág. 41).

OpenAI-Gym (2019). *OpenAI Gym documentation*. URL: https://gym.openai.com/ (vid. págs. 42, 43).

OpenAI-SpinningUp (2019). *OpenAI Spinning Up documentation*. URL: https://spinningup.openai.com/en/latest/index.html (vid. pág. 30).

Poddar, Ashish (2018). *Making a custom environment in gym*. URL: https://medium.com/@apoddar573/making-your-own-custom-environment-in-gym-c3b65ff8cdaa (vid. pág. 49).

Sonawane, Bhushan (2018). *Getting started with OpenAI Gym*. URL: https://towardsdatascience.com/getting-started-with-openai-gym-d2ac911f5cbc (vid. pág. 43).

Sutton, Richard S. y Andrew G. Barto (2018). *Reinforcement Learning, an introduction (second edition)*. URL: http://incompleteideas.net/book/the-book-2nd.html (vid. págs. 27, 52, 71).

Weng, Lilian (2018). «Policy Gradient Algorithms». En: *lilianweng.github.io*. URL: https://lilianweng.github.io/lil-log/2018/04/08/policy-gradient-algorithms.html (vid. pág. 52).