etsit
escuela técnica superior de
ingeniería de telecomunicación

Escuela Técnica
Superior
de Ingeniería de
Telecomunicación

UNIVERSIDAD POLITÉCNICA DE CARTAGENA

Escuela Técnica Superior de Ingeniería de Telecomunicación

Demostrador de aprendizaje reforzado en plataforma de juego

TRABAJO FIN DE GRADO

GRADO EN INGENIERÍA TELEMÁTICA

Autor: Pedro Dimas Martínez

Director: Javier Vales Alonso

Cartagena a 19-06-2020

Universidad
Politécnica
de Cartagena

Resumen:

Palabras claves: aprendizaje por refuerzo, openAI Gym, policy gradient, q-learning, frozen lake, qbert

En este proyecto confeccionaremos e implementaremos un demostrador de aprendizaje máquina. Concretamente, diseñaremos dos modelos de aprendizaje por refuerzo, el primero mediante el algoritmo de Q-learning aproximado y el segundo con el algoritmo Policy-gradient. Ambos son algoritmos bastante populares en este campo debido a los buenos resultados que obtienen. Como entorno de aprendizaje, estudiaremos los resultados de los algoritmos mediante la librería de Python OpenAI Gym. Además de explicar ambos algoritmos y sus bases teóricas, explicaremos el uso de diferentes sistemas de aproximación de funciones de hipótesis. Concretamente, utilizaremos regresores lineales y árboles de decisión para el algoritmo de Q-Learning aproximado, y una red neuronal para resolver policy gradient.

Para resolver el algoritmo de Q-Learning aproximado, haremos uso del entorno de OpenAI Gym Frozen lake. Este entorno se presenta como un mapa de 4x4 donde el agente deberá llegar a la meta, esquivando los agujeros que se encuentre en el camino. Para este entorno, nuestro agente ha llegado a ganar una de cada tres partidas. Para el segundo modelo, utilizaremos el entorno de Atari QBert. Este videojuego se presenta como una forma intuitiva de demostrar el potencial de policy-gradient. Consiste en una pirámide de plataformas por las que el agente deberá pasar para conseguir completar la pantalla. Sin embargo, no será tan fácil puesto que habrá enemigos que perseguirán al agente para mermar sus vidas. En este entorno de entrenamiento, nuestros resultados muestran una muy buena progresión, comenzando con una recompensa media de -578 y alcanzando recompensas medias de +320. El agente adquiere con relativa facilidad la capacidad de eliminar enemigos para maximizar las recompensas.

Concluiremos analizando el rendimiento de ambos algoritmos en base a los resultados mencionados anteriormente, además de comentar diferentes líneas de ampliación del proyecto y posibles campos que se podrían mejorar. El objetivo primordial de este proyecto es proporcionar, a aquellos perfiles no expertos, una base para comenzar la andadura en este mundo del aprendizaje por refuerzo.

Tabla de contenidos

CAPÍTULO 1

Introducción

Estoy seguro de que todos hemos escuchado hablar sobre la **inteligencia artificial**, pero muchas veces lo nombramos casi como un ente superior que apenas conocemos. En este documento explicaremos estos conceptos para que resulten entendibles por un público inexperto.

Así que, comenzamos con la pregunta raíz que se nos viene a la cabeza: ¿qué es la inteligencia artificial? Se trata de una tecnología que nace con el objetivo de emular la capacidad del ser humano de aprender acerca de aquello que le rodea, es decir, la capacidad innata de cualquier ser vivo para adaptarse dinámicamente al medio en el que vive. Según es enseñado en los primeros niveles educativos, la diferencia entre cualquier ser humano y el resto de animales es que nosotros tenemos la habilidad de razonar. Y así, a medio camino entre la pereza y la eficiencia, obtenemos un objetivo: crear máquinas que imiten nuestra capacidad de "pensar". Dicho así parece un objetivo bastante ambicioso, pero el camino parte de reducir los comportamientos de la naturaleza a módulos lo más independientes posibles, para tratar de reproducirlos con sistemas que podamos fabricar. Entonces, ¿qué es aprender? Bueno, los humanos hemos hecho un esfuerzo bastante grande en documentar nuestro saber y por tanto vamos a ver qué dice nuestra primera fuente que define los conceptos, el DRAE. La RAE define aprender como *'adquirir el conocimiento de algo por medio del estudio o la experiencia'*. Y con este propósito nace la inteligencia artificial. Algo que a priori se puede imaginar como robots humanoides interactuando con nosotros como en la película de Will Smith, va más allá, ya que actualmente está siendo usado en nuestro día a día. Un ejemplo de lo integrada que se encuentra la inteligencia artificial en nuestra vida, es cuando dejamos que Youtube vaya reproduciendo canciones. El sistema trata de interiorizar tus gustos y de recomendarte artistas acordes a tus preferencias, basándose, entre otras cosas, en patrones que va estudiando atendiendo a comportamientos sociales similares.

Y ahora, la siguiente pregunta que nos viene a la mente suele ser, ¿cómo lo hace? Una de las ramas de la inteligencia artificial es el **machine learning** que, a diferencia de otras ramas, trata de llegar a una conclusión mediante ensayo y error, obteniendo características de los datos que introducimos al sistema. Hasta que se desarrolló esta tecnología, se programaba desarrollando todos los casos posibles y dando una solución general para aquellos casos no contemplados. El punto innovador del machine learning es que es el algoritmo el que decide qué características debe observar para aprender de los datos. Por ejemplo, cuando antiguamente se quería clasificar tipos de flores, el programador definía

un algoritmo con una estructura que podría ser de este estilo: comenzar observando el tamaño de la flor, para luego, centrarse el color y en función de esos resultados, descartar estos tipos de flores. Sin embargo, en las técnicas de machine learning el programador ofrece al programa los datos y es propio el algoritmo el que decide qué características influyen y con qué peso. Para explicar mejor su forma esencial de funcionamiento procederemos a entrar en materia más técnica. Como siempre, nuestras mentes se ofuscan en encasillar nuestros conocimientos, buscando formas de organizarlos para poder memorizarlos. Dentro del machine learning tenemos tres grupos de técnicas principales: el aprendizaje supervisado, el no supervisado y el aprendizaje por refuerzo.

1.1. Tipos de machine learning

El **aprendizaje supervisado** se basa en aprender a través de un maestro omnisciente que conoce perfectamente los resultados que queremos obtener. Es el programador el que aporta un conjunto de entrenamiento etiquetado con la respuesta correcta que queremos obtener del algoritmo. Para que se entienda más fácilmente, voy a poner el ejemplo de lo que técnicamente se conoce como un problema de clasificación. Supongamos que yo quiero que un niño aprenda la diferencia entre un elefante y un ciervo. Para ello, le enseñaré muchas fotos de elefantes y decirle son elefantes. Luego le mostraré varias fotos de ciervos y le diré que son ciervos. Al final, estamos etiquetando las fotos según sean de la primera clase de animales (elefantes) o de la segunda (ciervos) y conseguimos que el niño aprenda por asociación. Lo más importante en este tipo de problemas es que conocemos de antemano el resultado que queremos obtener, sabemos si la imagen que le estamos enseñando es un ciervo o un elefante. Por desgracia, esta condición no siempre se cumple ya que hay veces en las que el sistema se construye con el fin de averiguar la respuesta a algo. Este tipo de técnicas se utilizan para conseguir identificar elementos y discernir unas cosas de otras, pero sin embargo no aportan información nueva propiamente dicha.

Por otro lado, tenemos el **aprendizaje no supervisado**. Es bastante sencillo confundirlo con el aprendizaje supervisado puesto que este tipo de algoritmos también se suelen utilizar para clasificar datos. Por ejemplo, imaginad que estamos en una habitación con un cerdo, una silla, un pollo, una mesa, un libro y un pato. Nosotros no conocemos nada de este conjunto variopinto, pero queremos clasificarlos de alguna forma. Los observamos y vemos que hay cosas que no se mueven y que simplemente están ahí. También observamos que hay otro tipo de elementos que hacen ruidos, se mueven y probablemente se terminen comiendo al resto de la sala si les dejamos suficiente tiempo. En esencia, lo que estamos haciendo es obtener características de los elementos y clasificarlos en grupos que comparten esas características, que en este caso podrían ser objetos inanimados y seres vivos. La diferencia con el aprendizaje supervisado reside en que yo no sé si esta clasificación es buena o no, o sea, no se usa el resultado esperado para el aprendizaje. Otra persona podría llegar y clasificarlos por colores o por tamaños y esta clasificación sería igualmente válida, dependiendo de para qué la queramos utilizar.

En última instancia, tenemos lo que desde mi punto de vista se fundamenta en una aproximación más psicológica a nuestra forma de pensamiento, el **aprendizaje por refuerzo**. Tratando de reducir mucho su funcionamiento, podríamos decir que trata de observar experiencias, decisiones tomadas para esas experiencias y una valoración lo más objetiva

posible de lo positivas que han sido las respuestas. Podríamos definir esas experiencias como el conjunto de variables que definen una situación concreta, o sea, una forma de representar el estado del entorno que queremos estudiar para un momento concreto. Imaginemos ahora que tenemos una tostadora antigua que no conocemos su funcionamiento y observamos que tiene un piloto apagado. Pulsamos el botón que pone ON y vemos que el se enciende piloto. Para estudiar la tostadora la tocamos y observamos que no pasa nada. Bien, una respuesta no negativa. Pasado un rato, el piloto se apaga y decidimos volver a tocarla, porque somos así de empíricos. Nos quemamos, una respuesta negativa. Pues ya podemos archivar en nuestra memoria que ante un estado formado por una tostadora y un piloto encendido, la respuesta ante la decisión de tocarla no es negativa, mientras que ante un estado formado por una tostadora y el piloto apagado, la respuesta que obtenemos al tocar la tostadora es bastante negativa. Si analizamos intuitivamente esa experiencia aprenderemos, en teoría, que no hay que tocar esa tostadora cuando el piloto esté encendido, pese a que más de uno nos haga falta repetir el experimento porque hay algoritmos más eficientes que otros.

Es en esta rama del machine learning donde se centra este proyecto, pues nuestro objetivo es realizar un algoritmo de aprendizaje reforzado que sea capaz de jugar a un videojuego. A menudo, los videojuegos presentan un escenario idóneo para demostrar la efectividad de los algoritmos de reinforment learning (RL) porque puedes reproducir una situación incontables veces. Además, es relativamente fácil caracterizar lo que consideras una recompensa positiva y una negativa ya que los juegos tienen un objetivo y habitualmente una cantidad de vidas. Así mismo, las acciones son limitadas y concretas: tienes cuatro botones que puedes pulsar y cada uno cumple una función. También, puedes entrenar varios algoritmos en paralelo puesto que se trata de una simulación. Y por último, resulta muy interesante poder comparar el desarrollo humano cuando se enfrenta a un videojuego desconocido y el de un agente inteligente.

1.2. Entornos de entrenamiento: OpenAI Gym

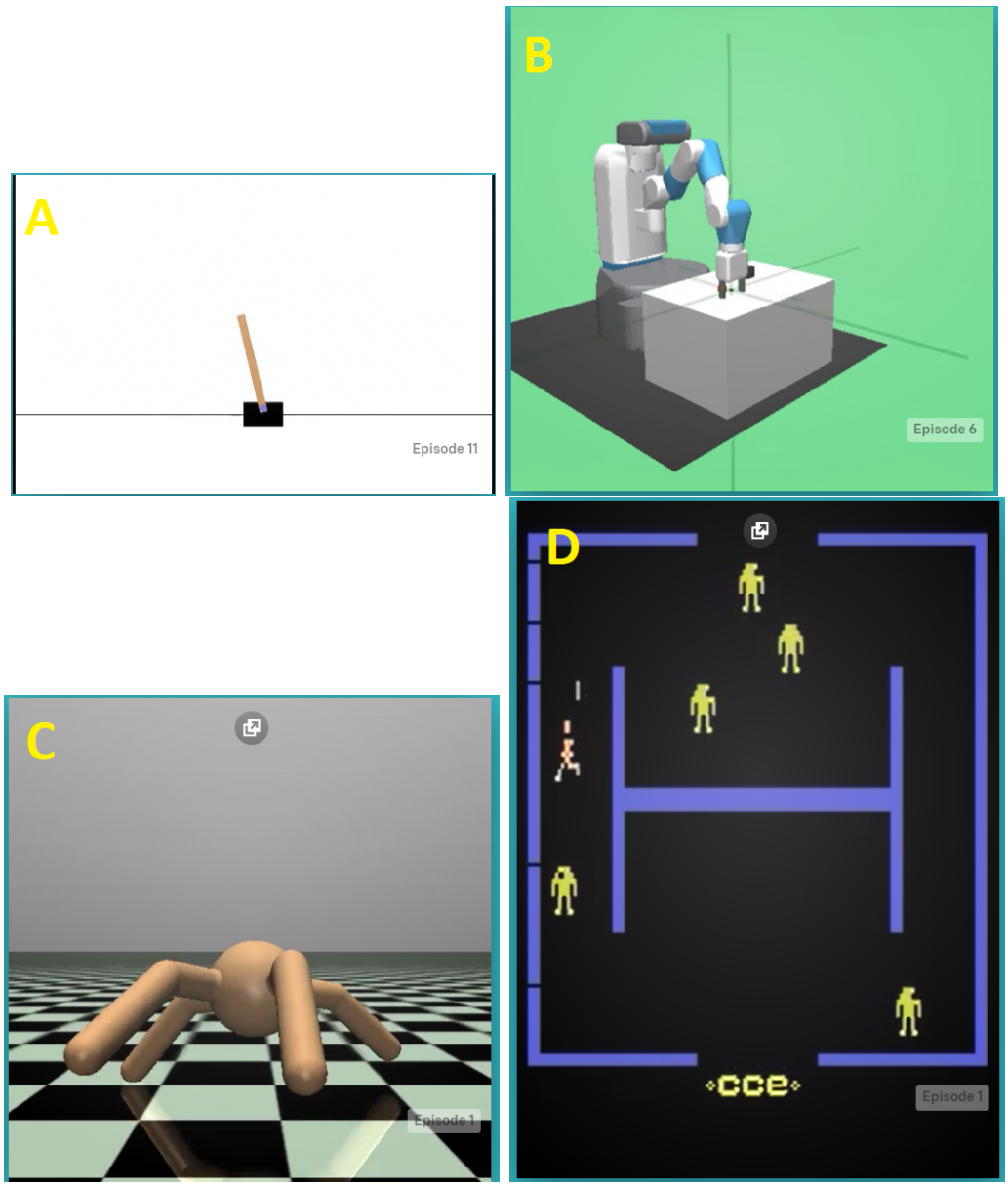


Figura 1.1: Diferentes entornos de entrenamiento de OpenAI Gym: (A) Classic control - CartPole, (B) Robotics - FetchPickAndPlace, (C) MuJoCo - Ant, (D) Atari - Berzerk

Como entorno de entrenamiento hemos escogido la librería OpenAI Gym, que resulta tremendamente útil a la hora de probar algoritmos de aprendizaje por refuerzo. Uno de sus puntos fuertes es que recopila multitud de entornos 1.1, desde videojuegos de Atari hasta trabajo con brazos mecánicos o simuladores para articular el movimiento de robots. A su vez, esta gran colección se complementa perfectamente con la sencilla interfaz que proporciona, que la hace realmente fácil de usar. La librería, proporciona la función `step(action)`, con la que obtendremos la nueva observación del entorno junto a la recompensa obtenida tras ejecutar esa acción. Además, OpenAI Gym proporciona el método `render()` que permite obtener una representación del entorno de una manera

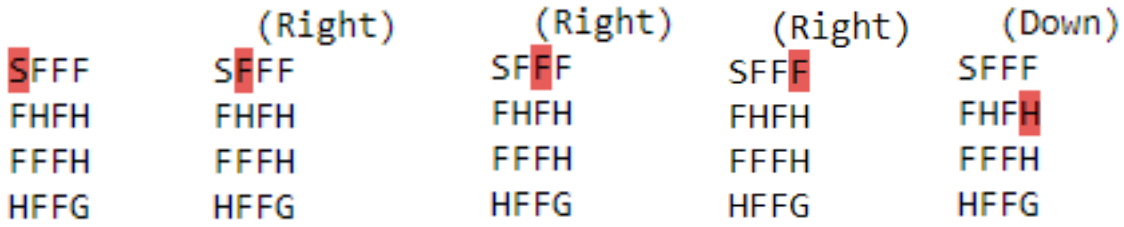


Figura 1.2: Secuencia de acciones en Frozen Lake

relativamente sencilla, al menos si lo ejecutas desde local. Más adelante comentaremos los problemas que hemos tenido con esta funcionalidad a la hora de ejecutarlo en un servidor remoto.

Para este proyecto utilizaremos Frozen Lake y QBert como entornos de entrenamiento de nuestros algoritmos. Ambos videojuegos están escogidos por dos razones principales. La primera es que la cantidad de acciones posibles es relativamente reducida. Si los comparamos con otros videojuegos, podemos intuir que es más fácil aprender a controlar los mandos de los juegos que menos acciones tienen. Si solo puedes pulsar dos botones, (arriba y abajo en el Pong, por ejemplo), es más fácil adivinar qué hace cada acción. Por otro lado, también se puede deducir que cuantos menos elementos tenga la pantalla más sencillo será aprender el objetivo del juego. Inicialmente escogimos el videojuego Berzerk que consiste en ir moviéndose por un escenario, eliminando enemigos para conseguir llegar hasta una zona concreta del mapa. Este entorno se descartó puesto que el set de acciones que contempla este entorno es bastante grande. Además de moverse en las cuatro direcciones, tienes la posibilidad de disparar hacia cada una de esas cuatro direcciones, lo que elevaría la dificultad del proyecto.

1.2.1. Frozen Lake

Frozen Lake, inicialmente plantea un mapa muy sencillo de 16 posiciones. Partimos de una salida (S) y el objetivo es llegar una meta (G). Por el camino, hay agujeros (H) que nos harán perder la partida y terrenos seguros (F) por donde debemos movernos para alcanzar la salida. El funcionamiento de este entorno es bastante intuitivo, vas saltando de casilla en casilla marcando la dirección en la que te gustaría ir.

Sin embargo, una de las peculiaridades de este entorno es que "está resbaladizo" y la elección de la acción no es determinista. En otras palabras, si eliges ir hacia arriba al agente igual se le antoja hacerte caso o tirarse al vacío por el agujero que tiene a la izquierda. Esto se puede desactivar pero hemos preferido mantenerlo para aumentar la dificultad del problema. Otro *handicap* que nos hemos impuesto es que el mapa a resolver sea aleatorio, manteniendo un número determinado de agujeros.

La idea era utilizar este simulador como un entorno sencillo para poder afianzar conceptos y estudiar el funcionamiento de OpenAI Gym. Pese a esta concepción inicial, este entorno lo hemos terminado atajando mediante un algoritmo de QLearning aproximado, conpcetualmente más complejo que el utilizado para QBert.

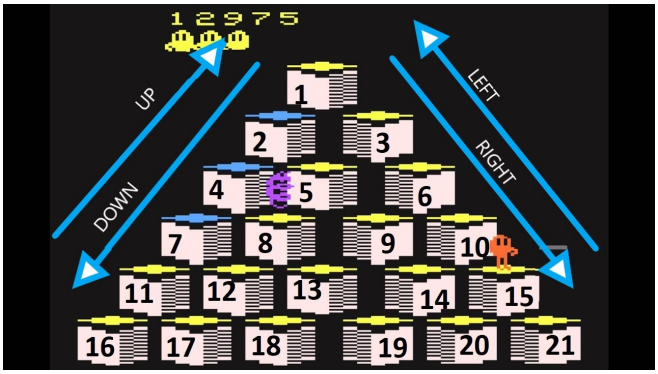


Figura 1.3: Pantalla de ejemplo de QBert, incluyendo las bases y las acciones disponibles.

1.2.2. QBert

Por otro lado, tenemos **QBert**. Este es un videojuego propiamente dicho que quizá algún lector conozca. Se podía jugar en la Atari 2600, aunque yo recientemente me lo encontré con sorpresa en un minijuego de la PlayStation 4. Consiste en una pirámide con varias plataformas y el objetivo es marcarlas todas, pasando por encima de ellas. Pero no estamos solos, encontraremos varios enemigos que tratarán de alcanzarnos para quitarnos vidas, aunque también podemos perderlas si nos caemos de la pirámide. Más adelante veremos que cada vida perdida será una recompensa negativa y cada vez que consigamos pasar por un plataforma nueva obtendremos una recompensa positiva.

Como mostramos en la figura 1.3, los controles de este videojuego pueden resultar extraños al principio, puesto que sólo se avanza en diagonal, pero al menos en este juego el personaje se mueve siempre hacia donde tú le indicas. El videojuego se puede jugar en la web retrogames, ¿podrás mejorar a nuestro agente? Sin embargo, a los desarrolladores de Gym les gusta jugar en modo difícil y cada acción elegida se realiza durante una cantidad aleatoria de frames. Esto lo hemos modificado utilizando la versión determinística de Qbert de la librería, como ya comentaremos cuando analicemos el código. Otra peculiaridad de este entorno es que los enemigos y las bases cambian de color con cada pantalla que superas. A la hora de implementar el código para modelar el estado, esto suponía una dificultad añadida. Por este motivo, haremos que el agente juegue siempre a la primera pantalla.

1.3. Estructura del documento

Una vez concluida esta primera aproximación al proyecto, sólo nos resta presentar la estructura global de este documento. A continuación, estudiaremos **trabajos relacionados** donde se apliquen los algoritmos de aprendizaje por refuerzo que hemos utilizado para resolver los videojuegos nombrados anteriormente. Una vez vistas estas investigaciones y el potencial de los algoritmos, explicaremos el **modelo de los sistemas** que hemos implementado. En este apartado, hablaremos más en detalle de cómo hemos mapeado los estados, considerado las recompensas y qué aproximaciones hemos utilizado para obtener el resultado. Seguidamente, continuaremos exponiendo la **implementación** del modelo que hemos programado, donde describiremos paso por paso las funciones que hemos utilizado. Tras esto, **analizaremos los resultados** obtenidos atendiendo a la evolución

de las recompensas tras el paso de las iteraciones de entrenamiento y probaremos la mejor política en una partida real. Por último, presentaremos nuestras **conclusiones** y comentaremos qué nuevas líneas pueden seguirse para ahondar en este trabajo.

CAPÍTULO 2

Trabajos relacionados

En este capítulo comentaremos algunos artículos académicos sobre el mundo del reinforment learning, que ayuden a dar una visión más enfocada a proyectos reales. En primer lugar comentaremos un artículo de los diseñadores de **OpenAI Gym**, que explica su pretensiones respecto a la librería. En segundo lugar, discutiremos un artículo sobre el uso de **policy gradient en redes 5G** para la planificación de la asignación de recursos.

2.1. Artículo sobre OpenAI Gym

El primer paper que discutiremos será el artículo [1] que define nuestro entorno de entrenamiento: OpenAI Gym. Este artículo está publicado por Brockman *et al.* Los investigadores, explican que OpenAI Gym nace para proporcionar una infraestructura que permita desarrollar y comparar fácilmente algoritmos de aprendizaje por refuerzo. Expone que la comunidad desarolladora en este campo reclamaba un sistema de referencias para medir las prestaciones de sus algoritmos. Gym nace con este propósito y para ello aporta su página web, donde fomentan que sus usuarios suban sus códigos fuente. Así mismo, este paquete aporta una interfaz genérica para todos los entornos. En cada `step`, el agente toma una una acción y recibe una recompensa del entorno. Todos estos entornos están formalizados como procesos de decisión de Markov parcialmente observables [2], donde el objetivo será maximizar la recompensa de cada estado.

2.1.1. Decisiones de diseño

Este artículo, comenta los motivos que llevaron a los diseñadores a tomar ciertas decisiones, como la de incluir solo entornos de entrenamiento y no agentes, para adaptarse a las necesidades de la mayor cantidad de tareas. Además, queda patente el objetivo de crear una comunidad abierta donde sus usuarios cooperen. Pese a que los mismos investigadores mencionan que está inspirado en la famosa página web Kaggle, con su librería no buscan la competición. En su lugar, persiguen que los usuarios compartan sus soluciones en un *Writeup*. En entornos de aprendizaje supervisado, es relativamente sencillo medir la precisión de un algoritmo mediante un set de datos desconocido. Sin embargo, en el campo del RL, entrenar los diferentes algoritmos en varios entornos sería computacionalmente costoso. Aportando muchas soluciones de distintos usuarios se consigue medir las

prestaciones de los algoritmos a un coste mucho menos significativo. Para medir los diferentes algoritmos, los desarrolladores decidieron destacar no solo el nivel de recompensa final que adquiere el algoritmo entrenado, sino también el tiempo necesario para alcanzar cierto umbral. Esta medición se antoja sencilla, puesto que cada vez que se aplica una acción se dice que se ha dado un `step`, y cada vez que se llega a un estado terminal, se completa un episodio. El tiempo de entrenamiento se puede medir de forma inmediata contando el número de episodios para alcanzar cierto nivel de recompensa. Por defecto, la interfaz proporciona un monitor capaz de grabar un vídeo del entrenamiento periódicamente, así como obtener gráficas que muestren el progreso del algoritmo. Finalmente, la última decisión que comentan los autores en el artículo es la de mantener un control estricto de las versiones. Cada cambio de cada entorno está detallado con un número de versión. El motivo reside en garantizar que los resultados sean siempre comparables, incluso si cambia el entorno.

2.1.2. Entornos actuales y aspiraciones futuras

Los autores mencionan los diferentes entornos que incorpora esta librería. Si bien es cierto que a lo largo de los años se han ido incorporando otros entornos como el juego Doom, en este artículo se destacan los siguientes:

- Classic control y toy text: Son tareas pequeñas del campo del RL.
- Algorithmic: Para realizar computaciones como añadir dígitos ó extraer secuencias.
- Atari: Los clásicos juegos de la Atari 2600, utilizando Arcade Learning Enviroment [3].
- Board games: Juegos de mesa como el Go en tableros de 9x9 y 19x19, utilizando el motor Pachi como oponente.
- 2D y 3D robots: Para controlar un robot con el motor de físicas de MuJoCo.

Para concluir el artículo, los autores destacan ciertas características que les gustaría implementar en un futuro, como la integración de Gym con hardware robótico para validar los algoritmos en el mundo real. Además, planean completar su librería con funciones que permitan la colaboración de varios agentes en un entorno o la encadenación de tareas para que el algoritmo aprenda mediante transfer learning.

2.2. Artículo sobre Policy Gradient en 5G

Sheng-Chia Tseng, Zheng-Wei Liuy, Yen-Cheng Chouy y Chih-Wei Huangy consiguieron que su artículo [4] fuese publicado en la conferencia sobre telecomunicaciones ICC Workshops. Los investigadores del 5G se vuelven a dar de bruces con un problema presente en cada estándar de telecomunicaciones: ¿cómo repartir los recursos entre los diferentes usuarios? Conforme se evoluciona la tecnología, este problema resulta más tedioso de resolver utilizando métodos convencionales, puesto que el número de parámetros aumenta. Para resolverlo, los autores han diseñado una arquitectura valiéndose de una variación del algoritmo que usaremos nosotros para resolver QBert. Este algoritmo se conoce como

Deep Deterministic Policy Gradient. Como bien expone el artículo, otros problemas relativos a este campo se han resuelto utilizando técnicas de machine learning. Los autores citan otros proyectos del campo del 5G como el documento [5], donde se propone un sistema de RL para resolver tareas de calidad de servicio (QoS). En este caso, se centrarán en la planificación de la asignación de recursos de radio en en redes RAN (radio access network) 5G. Como en cualquier sistema de RL, se debe modelar el estado, la acción, la recompensa y una forma de medir el rendimiento del algoritmo. Estos autores modelan esas características del sistema de la siguiente manera.

2.2.1. El estado

El equivalente a lo que nosotros catalogamos como estado vendrá dado por lo que en artículo se nombra como la parte del escenario. En él, se describen parámetros como el tiempo de medio de llegdas de UEs (User Equipment). Estos UEs serán los usuarios de la red a los que se les tendrán que asignar recursos. También, se tiene en cuenta las interferencias, la latencia o el grado de QoS exigido por cada servicio.

2.2.2. La acción

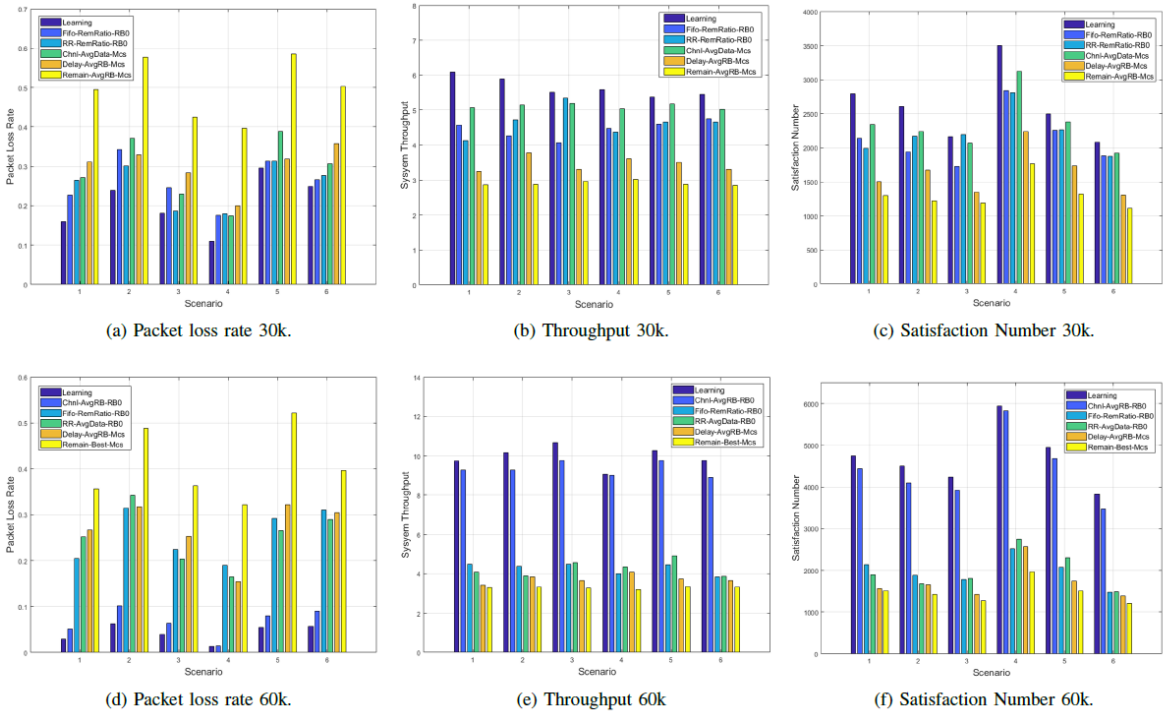
Habitualmente, el proceso de asignación de recursos se divide en tres etapas, atendiendo al propósito de cada tarea. La primera etapa consiste en ordenar los clientes que quieren acceder al canal, atendiendo a la demanda de QoS y a la justicia en el reparto. En segundo lugar, se decide la cantidad de recursos que se le va a asignar a cada cliente. Finalmente, se asigna una localización en el canal a cada cliente, cuidando la eficiencia de uso del espectro. Los posibles resultados de este proceso, compondrán la lista de acciones disponibles del sistema de aprendizaje por refuerzo.

2.2.3. La recompensa

Una vez establecidas las posibles acciones, la recompensa es el valor que otorga una intuición sobre los resultados de aplicar una acción en un TTI (Time transfer interval) concreto. Estos TTIs actuarán como los steps de nuestros algoritmos. La recompensa, queda modelada mediante un valor negativo que marca la satisfacción de cada UE. Si por ejemplo, un UE necesita transmitir dos paquetes en un periodo, tendrá una recompensa negativa si el sistema le asigna capacidad para enviar menos de esos dos paquetes. La recompensa obtenida en ese TTI vendrá dada por la suma de las recompensas de cada cliente.

2.2.4. Los resultados del experimento

Una vez definido el entorno, se procede a poner en marcha el experimento. Para hacer un experimento realista, los autores definen seis tipos de tráfico representando diferentes aplicaciones como conversaciones de voz y vídeo, streaming de vídeo o videojuegos en tiempo real. Una de ellas representa a tráfico de baja latencia y alta tasa de datos, que son las aplicaciones donde brillarán las redes 5G. Así mismo, componen seis diferentes escenarios, cada uno con una proporción diferente de cada tipo de tráfico. En el artículo, también describen la arquitectura de la red neuronal utilizada así como los parámetros de los algoritmos como el learning rate, el facotr de descuento o el tamaño de la memoria.



(a) Packet loss rate 30k. (b) Throughput 30k. (c) Satisfaction Number 30k.

(d) Packet loss rate 60k. (e) Throughput 60k (f) Satisfaction Number 60k.

Figura 2.1: Resultados del experimento del artículo Radio Resource Scheduling for 5G NR via Deep Deterministic Policy Gradient"

Resulta curioso ver en la figura 2.1 cómo el algoritmo de reinforment learning supera al resto de métodos convencionales para cualquier parámetro medido en el experimento, así como en todos los escenarios descritos.

CAPÍTULO 3

Modelo del sistema

En este apartado vamos a tratar de describir los modelos y algoritmos utilizados para resolver los problemas de **Frozen Lake** y **QBert**. Para ello, haremos uso del libro citado en [6], que nos ha servido de guía para confeccionar este proyecto. La idea general es jugar partidas para obtener recompensas asociadas a acciones. En ese momento definiremos una política que nos haga valorar si esas acciones son beneficiosas o no para superar el juego. Asociados a esta idea nos surgen diferentes dificultades que comentaremos a continuación.

3.0.1. El problema a la hora de evaluar las acciones

Pensemos en cierta situación. Vas paseando y llegas a una bifurcación con dos caminos, uno a la izquierda y otro a la derecha. Eliges ir por la derecha y encuentras un sendero paradisíaco que te lleva hasta tu meta, con lo que puedes catalogar esa experiencia como positiva. En otro momento, llegas a la misma intersección y como eres un explorador nato esta vez tomas el de la izquierda. Transcurrido cierto tiempo te conduce a un acantilado digno de un relato de Tolkien. Tratas de bajar escalando y te caes. La ley de la gravedad nos anticipa que vas a tener una recompensa bastante negativa. A la vista de estos sucesos, ¿Tiene toda la culpa la acción de bajar escalando, o influyó también la decisión en la bifurcación? Por lo tanto, ¿Debemos concluir que la acción que precede a la respuesta negativa debe ser archivada como mala y nunca más volver a hacerse? Esto es lo que se conoce como el problema de asignación de recompensas.

En definitiva, lo que podemos deducir de esta experiencia es que las acciones que escogemos influyen en las acciones futuras. Por lo tanto, la recompensa obtenida tras una acción no depende exclusivamente de la última acción. Debemos ajustar el problema para que la recompensa de una acción contemple las recompensas de las acciones posteriores. Para abordar ese problema, es común utilizar la suma de las recompensas futuras ponderadas por el factor de descuento γ.

$$R_t = \sum_{i=0}^{\infty} \gamma^i R_{t+i}, \text{ donde } \gamma \in [0, 1] \tag{3.1}$$

- R_t = Recompensa en el momento t

- γ = factor de descuento
- i = steps del episodio

Esta ecuación, lo que viene a decir es que la recompensa R en el momento t es la suma de las recompensas de las acciones siguientes (t+i), desde el step actual (i=0) hasta el final del episodio (i=inf). Cualitativamente, γ mide en qué grado se debe tomar en consideración el histórico posterior de acciones. De esta forma, si γ es cercano a 1 supondrá que el histórico de las acciones que tomemos determinarán en gran medida el resultado final.

3.0.2. Exploración vs explotación

Este problema se asemeja a un argumento irrefutable al que muchos nos hemos tenido que enfrentar de pequeños: ¿cómo sabes que no te gusta esta comida si no la has probado? Supongamos que estamos en un restaurante y en la carta hay platos que nos gustan, otros que no nos gustan y platos que directamente no hemos probado. Si únicamente pidiésemos nuestro plato preferido nos llevaríamos una grata experiencia. El punto flaco de seguir siempre esta política es que podemos desechar una opción mejor que nuestro plato preferido actual. Esto es lo que se conoce en el campo del machine learning como el balance entre explorar contenido nuevo frente a explotar la experiencia que hemos acumulado.

Para llevar a cabo este balance, hemos utilizado una función exponencial llamada ϵ-*greedy*, que devuelve una probabilidad de explorar en función de las iteraciones de entrenamiento. Esta probabilidad irá decreciendo conforme vaya pasando el tiempo, es decir, conforme más experiencia sobre el problema acumulemos. De esta forma, al principio elegiremos acciones aleatorias casi con total probabilidad, con las que el algoritmo visualizará hacia dónde conlleva cada cadena de acciones. Sin embargo, conforme el algoritmo vaya aprendiendo del entorno tratará de escoger la mejor acción. Concretamente, nosotros hemos utilizado la función que se detalla en la imagen 3.1.

$$expRate = minRate + (maxRate - minRate)\ e^{-decayRate\ \cdot\ iteration} \quad (3.2)$$

- $minExpRate = 0{,}5 \rightarrow$ probabilidad inicial de exploración.
- $maxExpRate = 0{,}1 \rightarrow$ probabilidad final de exploración.
- $decayRate = 0{,}1 \rightarrow$ ratio de descenso. Marca la pendiente de la exponencial.
- $iteration \rightarrow$ número de iteraciones de entrenamiento.

Y una vez estudiados estos problemas generales, vamos a analizar ambas soluciones a los problemas propuestos: QBert y Frozen Lake. Comenzaremos con QBert puesto que el algoritmo general es más sencillo de explicar. Continuaremos con Frozen Lake, que pese a tener un algoritmo más complejo, los métodos de estimación de probabilidades son más simples.

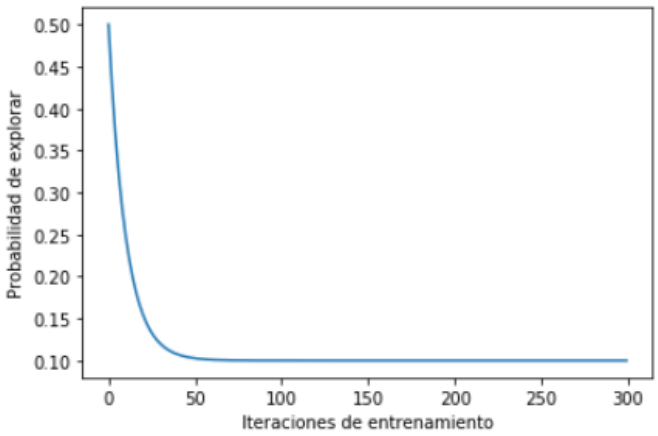


Figura 3.1: Pregresión de la función ϵ-greedy

3.1. Modelo de QBert

Este algoritmo se conoce como *'Policy gradient'* y consiste en optimizar los parámetros de una política de acciones para maximizar recompensa obtenida. Utilizaremos una red neuronal para obtener las probabilidades de elegir cada acción. De la misma forma que antes, es necesario sentar unas bases sobre cómo funcionan las redes neuronales antes de poder explicar cómo hemos resuelto este problema.

3.1.1. Función de hipótesis

En cualquier sistema de machine learning, la función de hipótesis es aquella función con la que podemos modelar el sistema, de forma que ante nuevas entradas somos capaces de predecir el valor de sus salidas. Esta función utiliza unos pesos (θ) para que al introducir una entrada determinada (X) nos produzca una salida deseada (Y). Vamos a resolver un ejemplo trivial. Supongamos que tenemos el siguiente conjunto de entrada $X = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]$ y sabemos que cada valor se corresponde con el siguiente conjunto de etiquetas $Y = [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]$. La función de hipótesis que caracteriza estos datos es muy sencilla:

$$h_\theta(x) = \theta_0 + \theta_1 x = 2 + 2x \tag{3.3}$$

Por ejemplo, supongamos que queremos predecir la probabilidad de tener cáncer de pulmón y nuestras variables son con la edad del sujeto, la frecuencia con la que fuma y el deporte que practica. Bien pues seguramente terminaremos teniendo un peso positivo para la edad y los cigarrillos, porque a más cantidad de estas características mayor probabilidad de cáncer de pulmón hay. Así mismo, acabaremos con un peso negativo para el deporte porque a más cantidad de deporte, menos probabilidad hay.

3.1.2. Función de coste

Una función de coste es una expresión matemática que depende de ciertos pesos θ y que pretendemos que represente lo lejos que estamos de una solución óptima del problema.

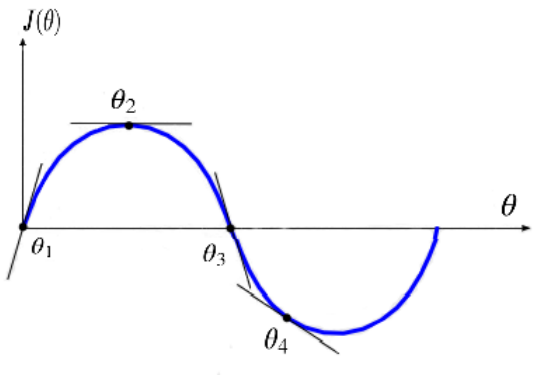


Figura 3.2: Función de coste aleatoria con un solo peso

Matemáticamente, mide la diferencia entre el valor obtenido de aplicar una función hipótesis $h_\theta(x)$ y el valor de la salida esperada Y. Los pesos acabarán siendo ajustados para potenciar o disminuir las variables en función de lo que estas aporten a la solución óptima. Dentro del ejemplo anterior, si obtenemos como salida que la probabilidad de tener cáncer es del 0 % ante un paciente de 40 años, fumador diario y sedentario, tendremos una función de coste elevada, puesto que esa probabilidad no se corresponde con la realidad. Dándole sentido dentro de nuestros entornos, buscamos que si estamos perdiendo muchas partidas el valor de la función de coste sea muy alto, ya que significará que estamos tomando malas acciones. Sentada esta premisa, nuestro problema se reduce a minimizar esta función de coste para perder las menos partidas posibles.

En nuestro caso, para resolver QBert, hemos hecho uso de una función muy común en este campo: la función *'cross-entropy'*. Esta función, es muy conveniente cuando se analizan probabilidades de diferentes hipótesis, sobretodo porque tiene muy buenos resultados en problemas de clasificación binarios. como veremos próximamente, nosotros hemos usado una codificación de entrada *'one-hot'* para potenciar esta característica de nuestra función de pérdidas. Matemáticamente, cross-entropy se formula de la siguiente manera:

$$loss = -\sum_{i}^{C} t_i \log\left(s_i\right) \tag{3.4}$$

- $loss$ = Pérdida obtenida
- $i \in C$ = Cada clase del sistema. Por ejemplo cada posible acción del sistema
- t_i = Valor deseado para la clase i (label o target)
- s_i = valor de salida (predicho) para la clase i

3.1.3. Descenso del gradiente

Este es un método matemático que permite obtener el mínimo de una función de coste determinada. Este método se basa en calcular el gradiente de la función de coste, es decir, las derivadas parciales de cada uno de los pesos. Estas derivadas parciales matemáticamente nos indican la pendiente de la función de coste en ese punto. Imaginemos que tenemos la función de la figura 3.2 de coste con un solo peso, es decir, una sola variable.

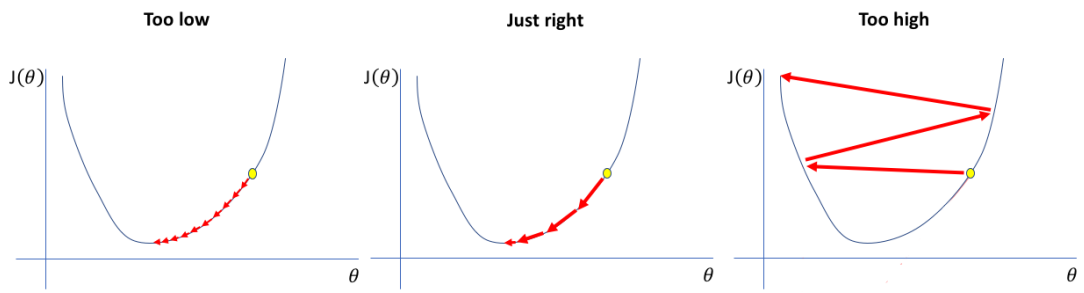


Figura 3.3: Minimización de la función de hipótesis con diferentes learning rates

La pendiente para el peso θ_1 de esta función es positiva por lo que implica que conforme la característica θ aumenta, la función de coste también aumenta. Sin embargo, en el punto θ_4, tenemos una pendiente negativa, lo que implica que conforme sube el valor de θ, el valor de la función de coste disminuye. En cada iteración k, el valor del peso θ_i se actualizará según esta ecuación:

$$\theta_i^k = \theta_i^{k-1} - \alpha \cdot \frac{\partial}{\partial \theta_i} J(\theta) \text{ para todo } \theta_i \tag{3.5}$$

- θ_i^{k-1} = Valor del peso i en la iteración anterior
- $J(\theta)$ = Función de coste
- α = Learning rate

De esta forma, si la pendiente es positiva, disminuiremos el valor del peso, por lo que iremos hacia un valor menor de la función de coste. Sin embargo, si la pendiente es negativa, estaremos aumentando el valor del peso para seguir avanzando hacia una función de coste menor. Como la función de coste mide lo cerca que está la solución actual de la salida esperada, con este método se converge a una solución óptima. En nuestro problema particular, QBert, ejecutaremos una etapa de descenso de gradiente en cada iteración de entrenamiento, calculando los gradientes de los pesos de los enlaces entre las neuronas.

Sin embargo, antes de entrar en materia con las redes neuronales, queda por explicar el último parámetro de esta formulación: el **'learning rate'**. Este parámetro descrito como α, hace referencia a la velocidad de convergencia del algoritmo, es decir, la proporción en la que aumentamos o disminuimos el peso en cada iteración del descenso del gradiente. Como se muestra en la figura 3.3, con un learning rate bajo, disminuiríamos los pesos muy poco a poco, lo que desembocaría en un algoritmo que tardaría mucho tiempo en encontrar una solución óptima. No obstante, poner un learning rate muy alto, supondría aumentar demasiado los pesos, tanto que podrían saltarse el mínimo y no llegar a converger nunca. En el caso de QBert, el learning rate lo hemos establecido a 0.01.

3.1.4. Funcionamiento de la red neuronal

Ya hemos estudiado un método de Una red neuronal es un modelo computacional supuestamente inspirado en el funcionamiento del cerebro humano. Biológicamente, nuestro

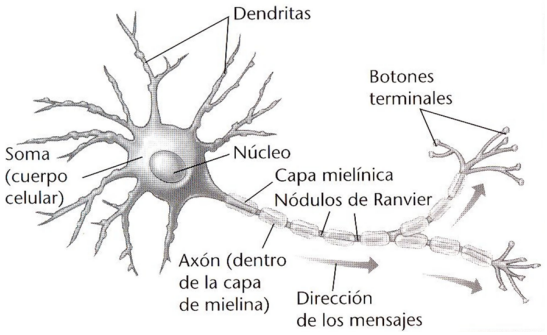


Figura 3.4: Neurona biológica

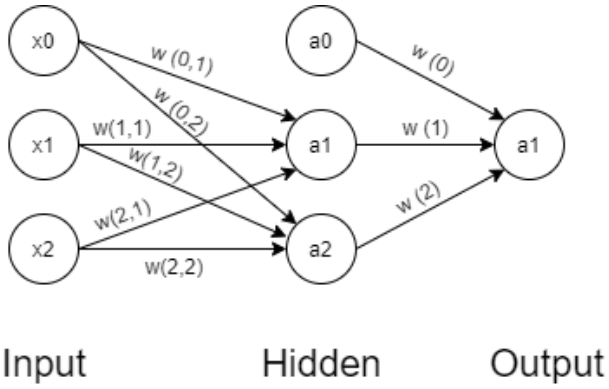


Figura 3.5: Arquitectura de una ANN con una capa oculta

cerebro está constituido por neuronas, unidades que actúan como sistemas de transformación de información. A cada neurona le llega un impulso nervioso a través de las dendritas que es transformado parcialmente y propagado a las siguientes neuronas a las que está conectada. Estas neuronas están distribuidas en capas y cada una se enfoca en una función dentro del sistema biológico.

De manera análoga al cerebro humano, las redes neuronales artificiales (*ANN*) están compuestas por unidades de análisis de información dispuestas en capas, las neuronas 3.4. Como mínimo, una red neuronal tiene una capa de entrada y otra de salida. Adicionalmente, puede tener una capa intermedia llamada capa oculta, para quitarle rigidez al modelo y que sea capaz de obtener mejores respuestas. Pongamos el ejemplo de una red neuronal sencilla de la figura 3.5

En esta red neuronal tenemos tres capas. En la capa de entrada tenemos tres neuronas y cada de ellas representa una característica de entrada del sistema. Siguiendo el ejemplo anterior, cada neurona de entrada representaría una característica de un sujeto de estudio: la edad, la cantidad que fuma y el deporte. En la capa oculta tenemos otras tres neuronas, cada una con su función de activación. Estas neuronas, operan con las entradas y propagan un resultado a la siguiente capa. La primera es la neurona se conoce como *'bias'*, y se añade para dar flexibilidad matemática al problema. La tercera capa, la de salida, nos da la probabilidad de que, dado un paciente concreto (con su edad, su consumo y su deporte), tenga cáncer.

Al principio, la red neuronal se inicializa con pesos aleatorios y se aplica el algoritmo *'forward propagation'*. En este algoritmo, cada neurona salvo las bias, reciben las entradas

de la capa anterior, multiplican cada valor por un peso determinado y aplican la función de activación, que da un resultado numérico concreto. Iterando este proceso en cada capa, al final de la red neuronal, obtenemos la probabilidad deseada. No obstante, al comenzar con los pesos aleatorios difícilmente estas probabilidades serán fiables. El siguiente paso consiste en aplicar el algoritmo *'backpropagation'* para entrenar la red, es decir, hacer que estas probabilidades se ajusten al conjunto de datos que se maneja. Este algoritmo, calcula el error de la neurona de salida, obtenido mediante la función de coste. Este error se calcula para las neuronas de las capas intermedias, mediante variaciones de las derivadas parciales que contemplan el error de la capa anterior. Una vez que tienes el error, se modifican los pesos de cada enlace en función del signo, de forma parecida al descenso del gradiente.

Abstrayéndonos de la parte matemática detrás del funcionamiento de estos sistemas, la mayor complejidad que presentan las redes neuronales, reside en definir una arquitectura que sea capaz de aproximar correctamente los resultados deseados. En función de la función de activación o de la naturaleza de las capas que utilicemos, tendremos una ANN que cumpla mejor ciertos propósitos. Por ejemplo, si utilizamos una red con capas convolucionales, funcionarán mejor para tareas de procesamiento de imágenes. Antes de estudiar nuestra arquitectura concreta de la red neuronal, necesitamos conocer cómo hemos modelizado el estado, porque de ella dependerá el número de neuronas de entrada de la red.

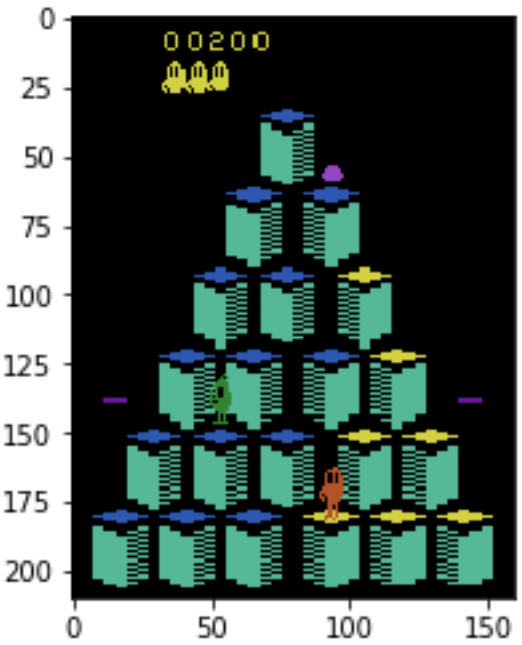
3.1.5. Codificación del estado

En primer lugar comentaremos cómo hemos parametrizado el estado. Como hemos explicado anteriormente, QBert es un videojuego que nos sitúa en una pirámide con bases y el objetivo es pasar por todas ellas. Hay dos tipos de enemigos y ambos nos harán perder una vida al colisionar contra el personaje principal. Como demuestra la figura 3.6, nuestro estado se compone de cuatro vectores que corresponden a esas tres variables: bases, posición del personaje principal y posición de los enemigos.

El primer vector está compuesto de 21 posiciones, que tendrán un 0 si hemos de pasar por la base (es decir, si la base no está coloreada) y un 1 en caso contrario. Es importante resaltar que hay enemigos que cambian de color las bases por las que ya hemos pasado, por lo que puede suceder que tengamos que pasar por alguna base varias veces para completar el nivel. El resto de vectores corresponde a la posición del agente y de los enemigos, utilizando una codificación *'one-hot'*. Habrá un 1 en la posición de la base en la que se encuentre el personaje o el enemigo, y un cero en todas las demás.

La modelación de nuestro estado, tendrá una longitud de $21 \cdot 4 = 84$ vectores, que es lo que tomaremos como entradas de la red neuronal. Esta representación, pese a que para nosotros los humanos sea compleja por su longitud, funciona bastante bien con los algoritmos puesto que solo estamos utilizando variables categóricas binarias.

Casi al final del entrenamiento, nos percatamos de que el agente trataba de utilizar siempre las barras laterales. Esto, es un problema porque las barras laterales solo se pueden usar una vez y el agente acababa muriendo. Por ello, decidimos que sería buena opción añadir dos variables binarias que mostrasen si se puede utilizar la barra lateral o no. Sin embargo, esto acarrea problemas que comentaremos a continuación.




```
Base colours [0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 1, 0, 0, 0, 1, 1, 1]
PJ position  [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0]
Enemy position 0 [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0]
Enemy position 1 [0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

Figura 3.6: Modelación del estado de QBert

3.1.6. Arquitectura de la red neuronal

Y una vez conocidas las dimensiones de las características que vamos a manejar, procederemos a explicar el sistema que actúa de ALU de nuestro algoritmo: la red neuronal. Como hemos comentado anteriormente, este sistema será el encargado de realizar los cálculos matemáticos que aproximarán las probabilidades de escoger una acción u otra. Para ello, debemos estudiar cuáles queremos que sean nuestras salidas. En nuestro entorno, tenemos cinco acciones posibles: izquierda, derecha, arriba, abajo y, más útil de lo que aparenta inicialmente, no hacer nada. El número de acciones posibles determina el número de neuronas de salida que deberá tener nuestra red neuronal. Para la entrada, nos fijamos en la modelización del estado que hemos comentado anteriormente. Finalmente, el número de neuronas ocultas suele corresponder con el orden de las neuronas de entrada.

Y la figura 3.7 es como se nos queda esta arquitectura, con 84 neuronas de entrada, 84 neuronas en la capa intermedia y 5 de salida. Para nuestro algoritmo, hemos usado la función de activación *'elu'* para las neuronas de la capa intermedia. Otra opción muy común en este campo es utilizar la función de activación *'relu'*. Hemos escogido *'elu'* porque permite salidas negativas. Para las neuronas de la capa de salida hemos utilizado la función *'softmax'*, que comentaremos más adelante puesto que hemos implementado una variación para Frozen Lake. Escogemos esta función para obtener como salida de la red la probabilidad de tomar cada acción.

Sin embargo, como hemos comentado antes, prácticamente al final del entrenamiento decidimos tratar de añadir las barras laterales al estado. Esto, pese a que parezca una modificación trivial en cuanto a concepto, tuvo bastante repercusiones: había que cambiar

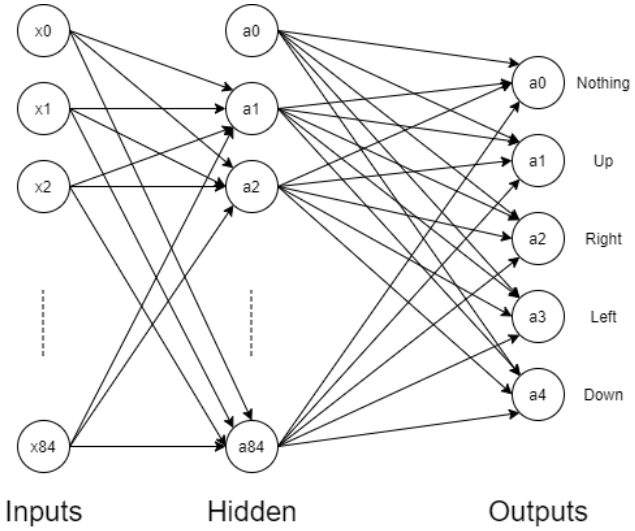


Figura 3.7: Arquitectura ANN QBert

la red neuronal. Por ello, cuando hemos puesto en marcha la última versión del algoritmo, hemos utilizado 86 neuronas de entrada y 86 neuronas en la capa oculta.

3.1.7. Adaptación de las recompensas

Inicialmente, QBert asigna una recompensa de 25 por cada plataforma que pasas, pero no otorga recompensa por perder vidas. Para facilitarle las cosas a nuestro agente, hemos establecido una recompensa de -250 cada vez que se pierde una vida y una recompensa de 1000 por completar la pantalla. Con estos cambios, al agrandar más las recompensas y hacerlas más extremas, el algoritmo encontrará más rápido la política de acciones correcta. También por defecto en QBert, otorga una recompensa de 300 por eliminar al enemigo verde y de 400 por eliminar al enemigo rosa. Es curioso porque esto no lo sabíamos inicialmente, lo descubrimos mientras el algoritmo jugaba. Así mismo, en cada iteración de entrenamiento, descontaremos las recompensas como describimos al principio de este capítulo. En este caso, hemos utilizado un discount rate de 0.95. Tras este proceso, normalizaremos utilizando la típica fórmula para normalizar parámetros estadísticos:

$$r = \frac{r - \mu}{\sigma} \tag{3.6}$$

- r = Recompensa descontada
- μ = Recompensa media de la iteración de entrenamiento
- σ = Desviación estándar de la iteración de entrenamiento

3.1.8. Estructura del algoritmo general

Y una vez sentadas las bases de estos sistemas concluiremos esta sección comentando el algoritmo general, que comentaremos a continuación. Estos pasos los ejecutaremos en bucle hasta que la red neuronal obtenga una solución que consideremos aceptable para resolver el problema. Cada vez que completemos estos pasos, lo consideraremos una iteración de entrenamiento.

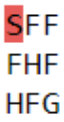


Figura 3.8: Mapa de ejemplo 3x3 de Frozen Lake

1. La red neuronal jugará 500 partidas, y en cada step calcularemos los gradientes de los pesos de la red neuronal. También, guardaremos la recompensa de cada de acción.
2. Cada 500 partidas, pasaremos a una fase de entrenamiento, en la que obtendremos las recompensas normalizadas y descontadas, fijando el discount rate (γ) a 0.95.
3. Multiplicaremos cada gradiente por la recompensa descontada y normalizada. Así, si una recompensa ha sido muy grande (positiva), se aumentarán los gradientes en la misma proporción.
4. Finalmente, se aplicará un paso de descenso del gradiente. Como los gradientes están potenciados por las recompensas, los pesos serán incrementados (o disminuidos) en proporción a lo que hayan contribuido a obtener buenas recompensas.

3.2. Modelo de Frozen Lake

Para solventar el problema de Frozen Lake, hemos utilizado una técnica de **QLearning aproximado**. Sin embargo, antes de pasar a describir este algoritmo es necesario sentar unas bases. Primero, explicaremos lo que son las cadenas de Markov. Continuaremos con el QLearning, entendiéndolo como una forma de adaptar las cadenas de Markov al reinforment learning. Finalmente llegaremos a la adaptación práctica del QLearning: el QLearning aproximado.

3.2.1. Cadenas de Markov

Una **cadena de Markov** es una forma de modelar sistemas en el que se contemplan tuplas de estado y probabilidades de pasar al siguiente estado. Un estado puede ser de dos tipos, terminal o transitorio. Se cataloga como terminal si cuando llegamos a ese estado no podemos salir de él, es decir, se acaba el episodio. En Frozen Lake serían los agujeros y meta. Matemáticamente, la probabilidad de pasar a otros estados es 0 y la probabilidad de volver al mismo estado es del 100 %. Los estados transitorios serán todos los demás, es decir, por donde se puede pasar a otro a estado (salida y terrenos). Una característica notable de las cadenas de Markov es que son sistemas sin memoria. Esto quiere decir que la probabilidad de pasar de un estado a otro no depende de los estados anteriores, solo del estado en que te encuentras y del estado al que vas a saltar. Por ejemplo, imaginemos que tenemos el mapa 3x3 de la figura 3.8 de Frozen Lake:

En Frozen Lake tenemos 4 acciones: izquierda (0), abajo (1), derecha (2) y arriba (3). El mapa tiene unos límites, osea que si el agente está en la casilla de salida y va hacia la izquierda no se moverá. Para hacerlo más didáctico, vamos a eliminar la condición de

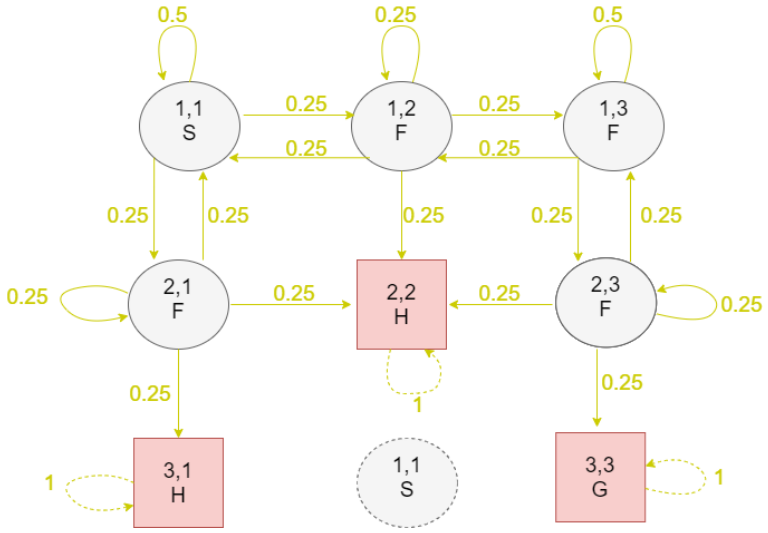


Figura 3.9: Cadena de Markov asociada al ejemplo

que el suelo está resbaladizo. Es decir, que siempre que elegimos una acción, vamos a desplazarnos una casilla en esa dirección. Si en cada casilla elegimos una acción al azar, la representación de este mapa sería la cadena de Markov de la figura 3.9.

Cada casilla posible del mapa está representada mediante un estado, en el que los estados terminales están representados como cuadrados y los transitorios como círculos. En este diagrama parece intuitivo que pasaremos de un estado a otro tomando una acción. Por ejemplo, desde el estado de salida (1,1), podemos ir o hacia abajo o hacia a la derecha con la misma probabilidad: 1 acción de 4 posibles. Sin embargo, si fuésemos hacia la izquierda o hacia arriba, al chocar con el límite del mapa, nos quedaríamos donde estamos. Por lo tanto, la probabilidad de quedarnos en el mismo estado es 2 de 4, el 50 %.

3.2.2. Procesos de decisión de Markov

Y esto, ¿qué relación guarda con el reinforment learning? Fue en 1950 cuando Richard Bellman, un matemático estadounidense, adaptó estas cadenas a un modelo de acciones y recompensas, los **procesos de decisión de Markov**. Resulta irónico pensar que esta tecnología es futurista cuando lleva siendo estudiada casi tres cuartos de siglo. Esta adaptación dio como resultado la **ecuación de optimización de Bellman**, que computa el "valor" de un estado para el sistema, suponiendo que se siguen las acciones óptimas, es decir, aquellas que consiguen una mejor recompensa.

$$V^*(s) = \max_a \sum_{s' \epsilon\ S} prob(s, a, s')[R(s, a, s') + \gamma V^*(s')] \text{ para todo } s \qquad (3.7)$$

- s = estado actual
- s' = siguiente estado
- S = Conjunto de estados posibles a los que podemos pasar desde el estado s
- $prob(s, a, s')$ = Probabilidad de transición del estado s al estado s', habiendo escogido la acción a

- $R(s, a, s')$ = Recompensa de pasar del estado s al estado s', habiendo escogido la acción a
- γ = factor de descuento
- $V^*(s')$ = Valor del estado s' escogiendo la acción óptima

Vamos a diseccionar esta pequeña ecuación para tratar de facilitar su comprensión. En primer lugar, definamos las variables s, s' y S. La variable s hace referencia al estado en el que estamos actualmente, por ejemplo el primer estado (1,1). La variable S implica al conjunto de todos los estados a los que podemos pasar, en este caso serían el estado (1,1), el (1,2) o el estado (2,1). La variable s' sería cada uno de esos estados a los que podemos saltar. El término $R(s, a, s') + \gamma V*(s')$ nos otorga el valor de la transición desde el estado s al s'. Este término suma la recompensa de pasar del estado s a s' junto al valor que se espera del estado s'. Finalmente, al sumar el valor de cada transición multiplicado por la probabilidad de esa transición, estaremos promediando las transiciones posibles desde el estado s. Esta ecuación se calculará para cada estado del sistema, tratando de escoger en cada estado la acción que maximice el sumatorio.

Analizándolo de una manera más cualitativa, el resultado de la ecuación para el estado (2,1) tendría que tener mucho menos valor que el estado (2,3), ya que estar en el estado (2,3) significa que estamos más cerca de la meta y que tenemos menos probabilidad de caernos.

3.2.3. Algoritmo de iteración de Q-Values

Sin embargo, conocer el valor de cada estado puede ser útil pero no le dice al agente qué decisión tomar. Para esto, necesitamos un algoritmo iterativo con una función que dependa de la acción. Esto es lo que se conoce como el **algoritmo de iteración de Q-Values**. Los Q-Values actúan igual que $V^*(s)$, simbolizando el valor de la transición.

$$Q_{k+1}(s, a) \leftarrow \sum_{s' \epsilon\ S} prob(s, a, s')[R(s, a, s') + \gamma \max_{a'} Q_k(s', a')] \text{ para todo } (s, a) \qquad (3.8)$$

- $prob(s, a, s')$ = Probabilidad de transición del estado s al estado s', habiendo escogido la acción a
- $R(s, a, s')$ = Recompensa de pasar del estado s al estado s', habiendo escogido la acción a
- γ = factor de descuento
- $Q_k(s', a')$ = Q-Value del estado s' escogiendo la acción a' en la iteración k

Este algoritmo es muy similar a la ecuación anteriormente descrita. Básicamente, la diferencia que introduce es que especifica concretamente cómo iterar. Se trabaja con una tabla que almacena para cada estado, el Q-Value asociado a cada acción. Para calcular este valor, se estudiarán los Q-values del siguiente estado y se escogerá el mayor. Recursivamente, en cada iteración se actualizará un Q-Value y acabará convergiendo a los Q-Values óptimos. Una vez obtenidos estos valores, resolver el problema es trivial: en cada estado sólo tenemos que escoger la acción con mayor Q-Value.

3.2.4. Q-Learning

Y ya estamos muy cerca de llegar a la solución de nuestro problema, pero aún quedan cosas por perfilar. Hay un término en el que no hemos hecho demasiado hincapié y es el que nos transporta desde una dimensión teórica maravillosa donde todo encaja, a la devastadora realidad donde nada funciona. Ese término, es el que hemos definido como la probabilidad de transición de un estado a otro, dada una acción concreta. ¿Recordáis que inicialmente el suelo estaba resbaladizo? Cambiar la variable `is_slippery` a `True`, genera que pese a que el agente escoja ir hacia a la derecha, el entorno puede llevarlo en cualquier dirección. Bien pues este hecho complica la ecuación, porque justamente hace que $prob(s, a, s')$ sea desconocida. Así mismo, cuando el agente comienza a evaluar un entorno, no conoce qué recompensas va a obtener tras las acciones que desempeña. Primero, necesita experimentar las respuestas a sus actos para determinar cuánto valor tiene una acción o un estado. Para subsanar este problema con las probabilidades de transición y las recompensas, se desarrolló el **algoritmo de Q-Learning**, que se puede resumir con esta ecuación:

$$Q_{k+1}(s, a) \leftarrow (1 - \alpha)Q_k(s, a) + \alpha(r + \gamma \max_{a'} Q_k(s', a')) \tag{3.9}$$

- r = Recompensa obtenida en ese step
- α = Learning rate
- γ = factor de descuento
- $Q_k(s', a')$ = Q-Value del estado s' escogiendo la acción a' en la iteración k

Este algoritmo, vuelve a ser una adaptación de los anteriores pero tomando más independencia del estado asociado. Con esta ecuación solo se contempla la recompensa obtenida en el último step y se introduce el concepto de learning rate (α).

Analizando la ecuación, vemos que consta de dos términos contrapuestos. El primero de ellos, es el valor Q de ese estado que teníamos en la iteración de entrenamiento anterior. Este término representa el crédito que la vamos a dar a la experiencia pasada. Por otro lado, el segundo caracteriza la cantidad de nuevo conocimiento que vamos a tener en cuenta. Imaginemos que desde la infancia estamos acostumbrados a tener clases por la mañana y estudiando por las tardes. Sin embargo, llegas a la universidad y cambias el horario, estudias por la mañana y clases por la tarde y recibes una buena nota en un examen. Un learning rate cercano a 1, tomaría muy en consideración el primer término. Esto, supondría atender mucho al conocimiento aportado por la nueva experiencia, y seguir explorando el estudiar por las mañanas. Correlativamente, un learning rate cercano a 0 le daría mucho más peso al conocimiento que has obtenido hasta la fecha, el segundo término, y seguiría tratando de estudiar por la tarde. Se podría decir que este segundo término mide lo conservador que es un algoritmo.

3.2.5. Q-Learning aproximado

Y concluimos ya con esta exposición teórica del recorrido histórico del Q-Learning, llegando a la tecnología que hemos utilizado para intentar completar Frozen Lake. Es cierto que todos los sistemas de reinforment learning se pueden representar como procesos de

decisión de Markov, pero si contemplamos un videojuego como el Pong, la definición de estado no se presenta tan fácilmente. ¿Debería ser todas las posibles posiciones donde puede estar cada una de las palas y la bola? Serían demasiados estados y almacenar los posibles Q-Values de cada estado para cada acción sería computacionalmente muy costoso. ¿Adaptamos por franjas del mapa dónde esté la bola? Con este procedimiento se perdería demasiada información y el agente no sería capaz de aprender a jugar. Para resolver este tipo de problemas, la solución reside en encontrar una función que sea capaz de **aproximar los Q-Values**. Pese a que multitud de estudios demuestran que las redes neuronales son tremendamente eficientes en este campo, hay otras formas de aproximarlos. Nosotros hemos obtenido la aproximación de los Q-Values de dos formas computacionalmente menos exigentes. Hemos utilizado **regresores lineales** y **árboles de decisión**.

3.2.6. Aproximación de los Q-Values: Regresión lineal y árbol de decisión

En machine learning, un **regresor lineal** es un algoritmo utilizado principalmente en aprendizaje supervisado. El objetivo es tomar como función de hipótesis una recta, y conseguir los valores de los pesos que mejor ajustan esa recta a las salidas esperadas, en este caso las ecuaciones de Bellman. Como función de coste, utilizaremos una variante del error cuadrático medio, también llamado 'mse'.

$$J(\theta) = \frac{1}{2m}\sum_{i=0}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2 \text{ para todo } x^{(i)} \in \ X, \ y^{(i)} \in \ Y \qquad (3.10)$$

- m = Número de ejemplos de entrenamiento.
- $x^{(i)} \epsilon \ X$ = Muestra de entrada iésima del conjunto de entrenamiento X.
- $y^{(i)} \epsilon \ Y$ = Valor deseado de salida iésimo del conjunto de entrenamiento Y.
- $h_\theta(X)$ = Función de hipótesis del modelo.

De esta forma, una vez obtenida la función de hipótesis óptima, seremos capaces de predecir la salida y ante cualquier nueva entrada x. Es importante incluir el término independiente (bias), ya que si no lo ponemos estaríamos forzando a que todas las soluciones pasasen por el (0,0) y eso limita con creces la capacidad de cómputo del modelo. El principal problema de estos sistemas es que son limitados, ya que las funciones que pueden tomarse como función de hipótesis son rectas.

Un **árbol de decisión** consiste en una estructura que va analizando los datos de entrada y creando un diagrama de flujo en función de reglas de decisión simples inferidas a partir de las características de estos datos. Su principal ventaja es que son fáciles de interpretar. La figura 3.10 es un ejemplo de un árbol de decisión que clasifica variedades de iris.

En él, se intuye fácilmente que si el pétalo mide menos de 2.45cm de largo, con total probabilidad es una iris setosa. Si no, en función del ancho del pétalo, se clasificará en iris versicolor o virgínica. Un parámetro importante en estos árboles de decisión es el gini. Muestra el nivel de impureza, es decir, lo seguro que estamos de la clase del dato a

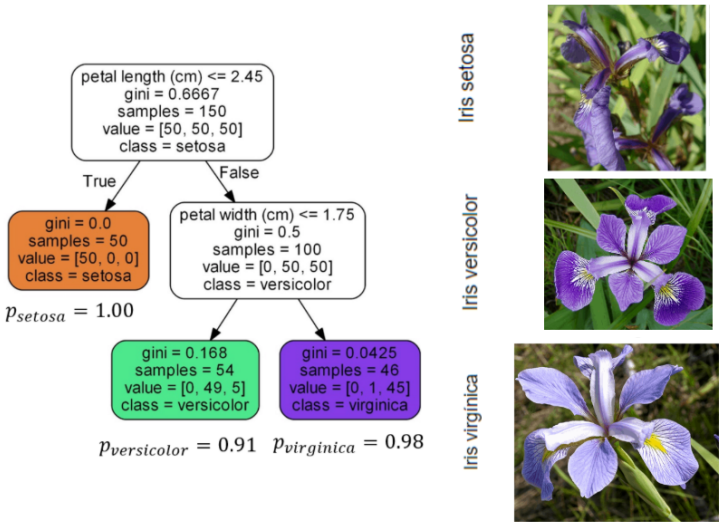


Figura 3.10: Árbol de decisión para clasificación de fotos de iris

predecir. En nuestro caso, en lugar de un árbol de decisión para clasificar, hemos utilizado un árbol para regresión. En estos modelos, el parámetro utilizado como nivel de confianza es el MSE.

Tanto cuando hemos usado un regresor lineal como cuando hemos usado el árbol de decisión, para aplicarlo a Frozen Lake manejamos un conjunto que contiene un regresor por cada acción. Ambos modelos tienen dos métodos principales. El método `fit` recogerá los datos de entrada y sus etiquetas para aproximar la función de hipótesis. Y el método `predict`, que utilizará la función de hipótesis para estimar las etiquetas de un dato nuevo.

3.2.7. Elección de la acción: softMax

En la línea comentada anteriormente de establecer un correcto balance entre exploración y explotación, hemos introducido la función `softMax`. El objetivo de esta función es transformar los Q-Values en probabilidades de coger una acción un otra. De esta forma, si por ejemplo nos encontramos con una bifurcación en la que tenemos dos acciones con Q-Values muy parecidos, ambas acciones serán casi equiprobables, por lo que unas veces se escogerá la de mayor Q-Value y otras veces la de menor. El procedimiento que hemos seguido para aplicar esta función sería el siguiente:

1. Normalizar los Q-Values de cada estado. Primero sumaremos 10000 para evitar contemplar negativos y calcularemos el logaritmo de este valor, para evitar que se distancien demasiado. Crearemos un vector que contenga el valor anterior si son positivos o 0 si no.

2. Aplicaremos la fórmula de softMax, con la que obtendremos las probabilidades de cada acción en relación a su Q-Value:

$$prob(qValue) = \frac{e^{qValue}}{\sum e^{qValuesNorm}}$$

3. Elegiremos una acción muestreando según las probabilidades obtenidas en el paso anterior.

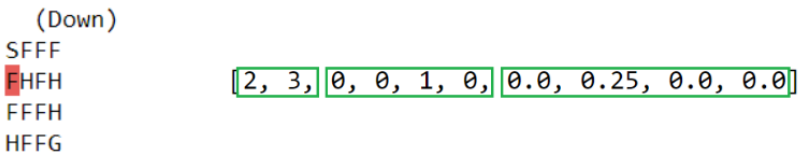


Figura 3.11: Representación del estado en QBert

3.2.8. Codificación del estado

Y una vez que ya tenemos el criterio para decidir las acciones estamos en posición de describir el estado. El estado es un conjunto de parámetros y variables que tratan de presentar al algoritmo el momento en el que se encuentra el agente. Para la mente humana, el estado podría ser el mapa, con cada casilla caracterizada según su función (agujero, meta o terreno). También forma parte del estado la casilla señalada que interpretamos como la posición en la que nos encontramos. En nuestro problema, el estado se modela como un vector de 10 posiciones que contemplan datos relevantes para resolver el problema. El significado de cada uno de los valores que componen el estado es el siguiente:

- La primeras dos posiciones codifican la distancia a la que estamos de la meta. La primera muestra a cuántas filas de distancia está, mientras que la segunda cuenta las columnas a las que está la meta.
- Las cuatro siguientes posiciones consisten en un valor binario que representa si hay agujero o no en las casillas contiguas (izquierda, abajo, derecha, arriba).
- Las últimas cuatro calculan la probabilidad de que haya un agujero a dos casillas de distancia. Para cada casilla contigua, se cuentan los agujeros alrededor y los divide entre las 4 opciones a donde podría ir (izquierda, abajo, derecha, arriba).
- Adicionalmente, en el caso del regresor lineal se añade una posición al principio del vector que codifica el término bias.

Un ejemplo de representación del estado sería el de la figura 3.11. El objetivo de esta representación es aportar al algoritmo parámetros más trabajados que faciliten aprender las reglas de decisión. En el ejemplo anterior, con este tipo de modelización debe facilitar al algoritmo aprender que disminuir los dos primeros valores hasta 0 conlleva una recompensa positiva. Así mismo, esta modelización debería ayudar a identificar que se deben evitar las acciones con un 1 en el segundo grupo del vector.

3.2.9. Adaptación de las recompensas

Con el fin de incrementar la eficiencia del algoritmo, hemos cambiado el sistema de recompensas original de Frozen Lake. El entorno por defecto, asigna una recompensa de +1 cuando consigues llegar a la meta y 0 en cualquier otro caso. Nosotros lo hemos cambiado para que se asigne una recompensa de +1000 cuando consigues llegar a la meta y de -1000 cuando te caes por un agujero. De igual forma que en QBert, polarizar las recompensas facilitará el trabajo al agente. En este caso, también hemos asignado una recompensa de -1 en cada paso que da el agente, para valorar que escoja el camino que lleve a la salida más rápidamente. Como veremos más adelante, el factor de descuento lo hemos fijado a 0.8.

3.2.10. Estructura del algoritmo general

El algoritmo tiene **dos etapas** fácilmente distinguibles. La primera etapa es en la que el agente va jugando partidas y almacenando las experiencias que obtenemos. La segunda etapa consistirá en entrenar los regresores en función de las experiencias que hemos obtenido. Estas dos etapas se repetirán en bucle hasta que el algoritmo esté correctamente entrenado.

Así mismo, la primera vez que ejecutamos el algoritmo deberemos inicializar los regresores. En este caso hemos utilizado un regresor por cada acción, es decir, manejaremos 4 regresores. Otra opción era hacerlo con un regresor multi-output, que nos diese 4 salidas y asignar cada una a una acción. Nosotros hemos elegido utilizar cuatro regresores porque resulta más simple ya que no hay que transformar las entradas.

Este algoritmo, se basa en dos modelos de 4 regresores: el modelo$\widehat{\ }$ y el modelo*. Ambos son conceptualmente iguales, pero la diferencia radica en que el modelo* es el que se utilizará para jugar partidas, mientras que el modelo$\widehat{\ }$ será el que entrenaremos. El objetivo del entrenamiento, será obtener los Q-Values óptimos, que vienen definidos por la siguiente función:

$$y^{(i)} = r^{(i)} + \gamma \max_{a'} Q(s'^{(i)}, a', \theta^*) \tag{3.11}$$

- $y^{(i)}$ = valor deseado del Q-Value (etiqueta) para la experiencia i
- γ = facor de descuento
- $r^{(i)}$ = recompensa de la experiencia i
- $\max_{a'} Q(s'^{(i)}, a', \theta^*)$ = Valor máximo del Q-Value del siguiente estado de la experiencia i, uilizando el modelo*

Esta función será la que usaremos como etiquetas (y) que utilizaremos para entrenar nuestros modelos, mientras que como entrada (x) utilizaremos los estados modelados como hemos descrito anteriormente.

En la **primera fase**, el objetivo será jugar muchas partidas para completar la replay-memory, donde almacenaremos los resultados de cada experiencia. Cada vez que avanzamos una casilla se seguirá el siguiente procedimiento:

1. Codificaremos el estado, de forma que obtendremos el vector explicado anteriormente.
2. Calcularemos los Q-Values para ese estado modelado, haciendo uso del modelo*. Obtendremos la acción a aplicar, primero llamando a la función ϵ-greedy y luego aplicando el procedimiento softMax.
3. Aplicamos esa acción y obtenemos el siguiente estado, la recompensa y si ha concluido la partida (episodio).
4. Modelaremos el siguiente estado y procesaremos la recompensa obtenida.
5. Guardaremos la experiencia obtenida en la replay-memory. Esta memoria contendrá el estado modelado, la acción que hemos escogido, la recompensa procesada y el

siguiente estado modelado.

6. Por último, si se ha acabado el episodio volveremos a inicializar el entorno con otro mapa que sea aleatorio.

Este algoritmo se repetirá hasta que se completen todos los pasos de entrenamiento (25000 o 12000). Cuando ya tengamos todas estas experiencias en la memoria, pasaremos a la **segunda fase**: el entrenamiento. El objetivo es que el algoritmo devuelva el valor óptimo de los Q-Values. Con este fin, utilizaremos el siguiente algoritmo:

1. Muestrear la replay-memory para obtener las experiencias. Nosotros las utilizamos todas, pero también es habitual usar solo una parte de las experiencias, por ejemplo el 30 %.
2. Para cada experiencia, calcular los Q-Values del siguiente estado, utilizando el modelo*. De entre esos Q-Values, almacenamos el mayor.
3. Por cada experiencia, guardar el valor del estado s.
4. Calcularemos la etiqueta de cada experiencia mediante la ecuación 3.11.
5. Entrenaremos el modelo$\widehat{\ }$ para que ante el estado guardado s trate de obtener el valor de la etiqueta y.
6. Finalmente, copiaremos el modelo$\widehat{\ }$ al modelo*, para continuar jugando con el modelo entrenado.

Una vez concluida la fase de entrenamiento propiamente dicha, queda asegurarnos de no perder el progreso. Para ello, utilizaremos un sistema de checkpoints que eviten oscilaciones. Para ello, se obtendrá el ratio de victorias obtenidas en esa iteración de entrenamiento. Este ratio se calcula como la fracción de partidas ganadas frente al total de partidas jugadas. Una vez tenemos este valor, si es el que mejor ratio tiene a lo largo de todo el entrenamiento, guardaremos el modelo$\widehat{\ }$. En caso contrario, entrenaremos el sistema hasta cinco veces más, buscando un ratio mejor. Si no conseguimos unos coeficientes que nos den mejor ratio de victorias, volveremos a utilizar el modelo que mejor resultados nos ha dado.

CAPÍTULO 4

Implementación

Una vez descritos los modelos que hemos utilizado para resolver ambos problemas, procederemos a explicar la implementación detallada de cada uno de los modelos. Comenzaremos comentando el algoritmo de Frozen Lake, puesto que es más simple a la hora de implementar.

4.1. Implementación de Frozen Lake

En este momento ya estamos preparados para ponernos manos a la obra con la programación. Comencemos analizando el entorno Frozen Lake, en concreto la pantalla y el resultado de ejecutar una acción. Estableceremos una semilla para hacer el notebook reproducible. Es importante destacar que se facilita el código utilizando un árbol de decisión. Si se quisiesen usar regresores lineales habría que cambiar algunos detalles que vienen indicados en el código.

[1]:
```
import gym
seed = 9

env = gym.make('FrozenLake-v0')
env.seed(seed)
env.reset()
env.render()

numActions = env.action_space.n
for action in range(numActions):
    observation, reward, done, info = env.step(action)
    print("observation", observation, "; reward", reward, "; done",␣
 ↪done)
    env.render()
```

```
SFFF
FHFH
```

```
FFFH
HFFG
observation 4 ; reward 0.0 ; done False
  (Left)
SFFF
FHFH
FFFH
HFFG
observation 8 ; reward 0.0 ; done False
  (Down)
SFFF
FHFH
FFFH
HFFG
observation 4 ; reward 0.0 ; done False
  (Right)
SFFF
FHFH
FFFH
HFFG
observation 4 ; reward 0.0 ; done False
  (Up)
SFFF
FHFH
FFFH
HFFG
```

Lo primero que obtenemos cuando empezamos con el entorno es un mapa 4x4. Si llamamos a `env = gym.make('FrozenLake-v0')` varias veces vemos que la distribución del mapa no cambia. Si analizamos las posibles acciones vemos que hay cuatro posibles: left, down, right y up. Sin embargo, la acción y el resultado no siempre concuerdan. Esto es debido a lo que hemos comentado antes, el suelo está resbaladizo y la dirección es aleatoria. Si nos centramos en la observación, vemos que es un solo número que representa la casilla en la que estamos. Vemos que cuando caemos en una casilla `H`, un agujero, perdemos y done se pone a `True`. También podemos ver que la recompensa ha sido siempre 0, por lo que conviene conocer cuál es la recompensa por ganar.

[2]:
```
import numpy as np

mapFlat = np.array([b"S", b"F", b"F", b"F", b"H", b"F", b"H", b"F",␣
 ↪b"G"])
map3x3 = mapFlat.reshape(3,3)
env = gym.make('FrozenLake-v0', desc=map3x3, is_slippery=False)
env.seed(seed)
obs = env.reset()
env.render()
policy = [2, 2, 2, 1, 1]
```

```
for step in range(len(policy)):
    obs, reward, done, info = env.step(policy[step])
    env.render()
    if done:
        print("Reward =", reward)
        break
```

```
SFF
FHF
HFG
  (Right)
SFF
FHF
HFG
  (Right)
SFF
FHF
HFG
  (Right)
SFF
FHF
HFG
  (Down)
SFF
FHF
HFG
  (Down)
SFF
FHF
HFG
Reward = 1.0
```

Bien, ahora ya sabemos que la recompensa por ganar, es 1.0. Por el camino, hemos forzado que la dirección no sea aleatoria con `is_slippery` y hemos enseñado cómo cambiar el mapa utilizando `env.desc`.

Una vez analizado el entorno, procederemos a analizar el código del algoritmo de reinforment learning. En primer lugar, importaremos las librerías necesarias.

- gym proporcionará el entorno de reinforcement learning.
- numpy para facilitar las operaciones matemáticas.
- sklearn para generar los regresores.
- matplotlib la usaremos para representar nuestros resultados. Matplotlib por defecto abre las representaciones en una ventana aparte. Con %matplotlib inline forzamos a que se muestren en la misma superficie que muestra los outputs. Esto es necesario al estar ejecutando el notebook en un servidor.

- time nos ayudará a ver con más claridad los pasos que va dando nuestro algoritmo, pudiendo introducir un delay en el código para observar bien el estado de la pantalla.
- display para poder limpiar la superficie de salida.
- os para poder acceder a los ficheros del equipo y guardar nuestro modelo.
- copy para copiar los elementos creando elementos nuevos, en lugar de utilizar referencias.
- pickle para serializar y guardar los modelos entrenados

[3]:
```
import gym
import numpy as np

from sklearn import tree
from sklearn.linear_model import LinearRegression

import matplotlib
%matplotlib inline
import matplotlib.pyplot as plt

import time
from IPython import display
import os
import copy
import pickle
```

A continuación, definiremos la función con la que obtendremos la posición de los agujeros, de la salida y de la meta. Tomará como argumento el entorno y devolverá un vector que contendrá tres vectores. El primero es la posición de los agujeros. El segundo es la posición de la meta y el tercero la de la salida. Las posiciones están expresadas como coordenadas `[fila, columna]`.

[4]:
```
#Returns the position of the holes and the position of the goal
def getHolesGoalStart(env):
    holes = []
    for row in range(len(env.desc)):
        for col in range(len(env.desc[row])):
            if env.desc[row][col] == b'H':
                holes.append([row,col])
            elif env.desc[row][col] == b'G':
                goal = [row, col]
            elif env.desc[row][col] == b'S':
                start = [row,col]
    return holes, goal, start
```

[5]:
```
env.render()
holes, goal, start = getHolesGoalStart(env)
print("holes:", holes)
```

```
print("goal:", goal)
print("start:", start)
```

```
  (Down)
SFF
FHF
HFG
holes: [[1, 1], [2, 0]]
goal: [2, 2]
start: [0, 0]
```

Seguidamente, implementaremos una función para obtener el mapa aleatorio. Este mapa tendrá cuatro agujeros, una meta y una salida, como el mapa original. El formato de salida es el que usa Gym para establecer su mapa.

[6]:
```
def getRandomMap(env):
    c = copy.deepcopy(env.desc)

    # Delete holes, goal, start from the original map
    holesPositions, goal, start = getHolesGoalStart(env)
    c[goal[0]][goal[1]] = b'F'
    c[start[0]][start[1]] = b'F'
    for position in holesPositions: c[position[0]][position[1]] = b'F'

    # Insert randomly holes, goal and start
    c = c.reshape(1,-1)
    numStates = env.observation_space.n
    numHoles = 4
    numPos = numHoles + 2 #x Holes + 1 goal + 1 start
    positions = np.random.choice(numStates, numPos, replace=False)

    c[0][positions[-1]] = b'S' #Start
    c[0][positions[-2]] = b'G' #Goal
    for position in positions[:-2]: c[0][position] = b'H' #Holes

    numRows, numCols = int(np.sqrt(numStates)), int(np.sqrt(numStates))␣
 ↪ #square map
    c = c.reshape(numRows, numCols)

    return c
```

[7]:
```
env = gym.make("FrozenLake-v0")
env.seed(seed)
getRandomMap(env)
```

[7]:
```
array([[b'F', b'F', b'F', b'F'],
       [b'F', b'F', b'H', b'H'],
       [b'H', b'F', b'S', b'H'],
```

```
       [b'F', b'F', b'F', b'G']], dtype='|S1')
```

Continuaremos con la modelación del estado. Definiremos una función que tome por argumento una posición variable y las posiciones de los agujeros. Se observará cada casilla colindante a la posición dada y devolverá un 1 si en esa casilla hay un agujero y un 0 si no.

[8]:
```
#Returns an array. Actions are ordered by L;D;R;U. 1 if there is a␣
 ↪hole in the position u go after taking that action, 0 else
def getHolesRound(position, holesPositions):
    holes = []
    #if there is a hole after taking an action this feature of the␣
 ↪state is 1, if there is something else then will be 0
    h = 1 if [position[0], position[1]-1] in holesPositions else 0 ␣
 ↪#Left
    holes.append(h)
    h = 1 if [position[0]+1, position[1]] in holesPositions else 0 ␣
 ↪#Down
    holes.append(h)
    h = 1 if [position[0], position[1]+1] in holesPositions else 0 ␣
 ↪#Right
    holes.append(h)
    h = 1 if [position[0]-1, position[1]] in holesPositions else 0 ␣
 ↪#Up
    holes.append(h)
    return holes
```

[9]:
```
env.reset()
numStates = env.observation_space.n
numRows, numCols = int(np.sqrt(numStates)), int(np.sqrt(numStates)) ␣
 ↪#square map
aux = np.arange(numStates).reshape(numRows, numCols)
obs, reward, done, info = env.step(1)

result = np.where(aux == obs)
position = [result[0][0], result[1][0]]
env.render()
print("position:", position, "; holes:", getHolesRound(position, holes))
```

```
  (Down)
SFFF
FHFH
FFFH
HFFG
position: [0, 1] ; holes: [0, 1, 0, 0]
```

Estamos preparados para ver en detalle la función que codifica el estado en sí. Como ya hemos comentado en el capítulo anterior, los dos primeros valores marcan la distancia a

la meta. Los cuatro siguientes valores representan los agujeros alrededor de la posición del agente. Los cuatro últimos representan la probabilidad de tener un agujero a dos pasos vista. Cabe destacar que si la aproximación se hace mediante regresores lineales, se debe añadir el término de bias al estado, es decir, un 1 como valor de la primera característica.

[10]:
```
def preprocessObservation(env, observation):
    numStates = env.observation_space.n
    numRows, numCols = int(np.sqrt(numStates)), int(np.sqrt(numStates))␣
 ↪ #square map
    aux = np.arange(numStates).reshape(numRows, numCols)
    result = np.where(aux == observation)
    position = [result[0][0], result[1][0]]


    holesPositions, goal, start = getHolesGoalStart(env)
    #Distance from goal
    dx, dy = goal[0] - position[0], goal[1] - position[1]
#     state = [1, dx, dy]  #Uncomment for linear regression (bias)
    state = [dx, dy]   #Uncomment for tree


    for h in getHolesRound(position, holesPositions):
        state.append(h)


    nextPosition = [position[0], position[1]-1] #Next position after␣
 ↪moving left
    prob = np.mean(getHolesRound(nextPosition, holesPositions))␣
 ↪#proabability of having a hole in this state
    state.append(prob)

    nextPosition = [position[0]+1, position[1]] #Next position after␣
 ↪moving down
    prob = np.mean(getHolesRound(nextPosition, holesPositions))␣
 ↪#proabability of having a hole in this state
    state.append(prob)

    nextPosition = [position[0], position[1]+1] #Next position after␣
 ↪moving down
    prob = np.mean(getHolesRound(nextPosition, holesPositions))␣
 ↪#proabability of having a hole in this state
    state.append(prob)

    nextPosition = [position[0]-1, position[1]] #Next position after␣
 ↪moving down
    prob = np.mean(getHolesRound(nextPosition, holesPositions))␣
 ↪#proabability of having a hole in this state
```

```
    state.append(prob)

    return state
```

[11]:
```
env.render()
state = preprocessObservation(env, obs)
print(state)
```

```
  (Down)
SFFF
FHFH
FFFH
HFFG
[3, 2, 0, 1, 0, 0, 0.0, 0.0, 0.0, 0.0]
```

Seguidamente, procederemos a explicar la parte relativa a los Q-Values y sus aproximadores. En cualquier algoritmo de aprendizaje supervisado, el procedimiento habitual suele comenzar con entrenar el sistema. Para ello, se le introduce un set de datos etiquetado con los valores esperados, por ejemplo, fotos de animales especificando si son ciervos o elefantes. Después, se pone a prueba el modelo pasándole datos nuevos de validación para estudiar su tasa de acierto. Para implementar los regresores, hemos utilizado la librería `sklearn` y ésta te obliga a llamar al método `fit`, para entrenar, antes de poder llamar al método `predict`. Es por esto que necesitamos la función `initRegressors` para inicializar los regresores con unos pesos aleatorios. Otra restricción de `sklearn` es que no trabaja bien cuando sólo hay una muestra en el conjunto. Tiene lógica, ya que es difícil aprender con un solo ejemplo. Como inicialmente los pesos van a ser aleatorios, simplemente copiamos el estado un número aleatorio de veces y lo inicializamos con una etiqueta aleatoria. Para implementar el modelo de predicciones, inicializamos cuatro regresores, uno por cada acción, y éstos serán los que usaremos para predecir. También se podría utilizar un solo regresor multiOutput.

[12]:
```
def initRegressors(state, numActions):
    regressors = []
    for action in range(numActions):
        y = [np.random.uniform() for _ in range(80)] #initialize␣
 ↪regressor randomly
        X = [state for _ in range(80)] #One single sample doesnt works

        model = tree.DecisionTreeRegressor(random_state = 0)
#         model = LinearRegression()
        model.fit(X, y)
        regressors.append(model)
    return regressors
```

A continuación, necesitaremos una función que pueda obtener el Q-Value dado un estado. Al llamar al método `predict` de `sklearn`, hay que especificar que es sólo una muestra con varias características. Finalmente, una vez obtenidos los Q-Values para un estado, implementaremos el método para obtener la acción por `softMax` como comentamos en el capítulo anterior.

[13]:
```
def getAllQValues(state, regressors):
    qValues = []
    for regressor in regressors:
        qVal = regressor.predict(np.array(state).reshape([1, -1]))
        qValues.append(qVal[0])  #all values are equals so get the␣
↪first
    return qValues
```

[14]:
```
def getActionSoftMax(qValues, debug=False):
    #+10000 for avoiding negative. Log for avoid large distances␣
↪between qVals
    qValuesNormalized = np.maximum(0, np.log([qVal+10000 for qVal in␣
↪qValues]))
    probsSoftMax = [np.exp(qVal)/np.sum(np.exp(qValuesNormalized)) for␣
↪qVal in qValuesNormalized]
    if debug:
        print("Probabilities:", probsSoftMax)
    actions = np.arange(len(qValues))
    action = np.random.choice(actions, p=probsSoftMax)
    return action
```

[15]:
```
regressors = initRegressors(state, numActions)
print("Regressors: ", regressors, "\n###############")
qValues = getAllQValues(state, regressors)
print("Q-Values: ", qValues, "\n###############")
action = getActionSoftMax(qValues, True)
print("Action: ", action)
```

```
Regressors:  [DecisionTreeRegressor(criterion='mse', max_depth=None,
max_features=None,
                      max_leaf_nodes=None, min_impurity_decrease=0.0,
                      min_impurity_split=None, min_samples_leaf=1,
                      min_samples_split=2, min_weight_fraction_leaf=0.0,
                      presort=False, random_state=0, splitter='best'),
DecisionTreeRegressor(criterion='mse', max_depth=None, max_features=None,
                      max_leaf_nodes=None, min_impurity_decrease=0.0,
                      min_impurity_split=None, min_samples_leaf=1,
                      min_samples_split=2, min_weight_fraction_leaf=0.0,
                      presort=False, random_state=0, splitter='best'),
DecisionTreeRegressor(criterion='mse', max_depth=None, max_features=None,
                      max_leaf_nodes=None, min_impurity_decrease=0.0,
                      min_impurity_split=None, min_samples_leaf=1,
                      min_samples_split=2, min_weight_fraction_leaf=0.0,
                      presort=False, random_state=0, splitter='best'),
DecisionTreeRegressor(criterion='mse', max_depth=None, max_features=None,
                      max_leaf_nodes=None, min_impurity_decrease=0.0,
                      min_impurity_split=None, min_samples_leaf=1,
                      min_samples_split=2, min_weight_fraction_leaf=0.0,
```

```
                        presort=False, random_state=0, splitter='best')]
###############
Q-Values:  [0.5454636354151157, 0.5209242989022127, 0.4524626410851324,
0.5262700081143794]
###############
Probabilities: [0.2500008545435473, 0.25000024109149904, 0.
 ↪24999852963755687,
0.2500003747273968]
Action:  2
```

Una vez que tenemos el sistema de asignación de Q-Values, aplicaremos ϵ-greedy para mantener un balance entre exploración y explotación. De esta forma, se garantiza que se exploran todos los caminos del mapa.

[16]:
```
#Epsilon greedy: Exploration vs Exploitation
expDecayRate = 0.1  #0.001
maxExpRate = 0.5
minExpRate = 0.1
epsGreedy = []
def epsilonGreedy(trainIter, qValues):
    expRate = minExpRate + (maxExpRate-minExpRate) * np.
 ↪exp(-expDecayRate * trainIter)
    if np.random.random() > expRate:
#         action = np.argmax(qValues)
        action = getActionSoftMax(qValues)
    else:
        action = np.random.randint(len(qValues)) #exploration
    return action, expRate
```

[17]:
```
probs = [1, 2, 3, 5, 6]
rates = [epsilonGreedy(i, probs)[1] for i in range(500)]

plt.ylabel("Probabilidad de explorar")
plt.xlabel("Iteraciones de entrenamiento")
plt.plot(rates)
```

[17]:
```
[<matplotlib.lines.Line2D at 0x7f6ef2445b50>]
```

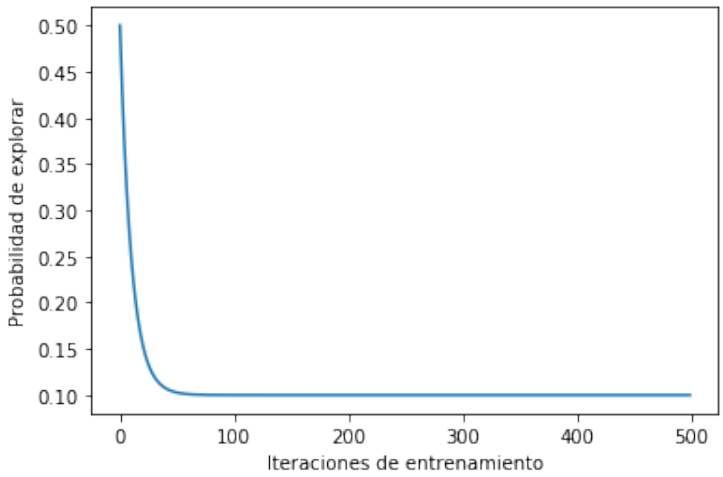


Figura 4.1: Probabilidad de explorar conforme se entrena la red

En siguiente lugar, definiremos una función para trabajar con la memoria. Necesitaremos que sea capaz de almacenar experiencias, compuestas por el estado actual, la acción que se toma, la recompensa obtenida y el estado en el que desemboca. Es habitual en problemas de este campo que tengamos almacenadas muchas experiencias. Computacionalmente puede ser más eficiente analizar sólo un conjunto de ellas y no la memoria completa. Para ello, hemos definido esta función, aunque luego en realidad no le llegamos a dar uso.

[18]:
```
def sampleMemory(memory, batchSize):
    shuffledMemory = memory[:]
    np.random.shuffle(shuffledMemory)

    memorySample = [] #state, action, reward, next_state
    #Sample not empty experiences
    for experience in shuffledMemory:
        if len(memorySample) < batchSize and experience:
            memorySample.append(experience)

    states, actions, rewards, nextStates = [],[],[],[]
    for experience in memorySample:
        states.append(experience[0])
        actions.append(experience[1])
        rewards.append(experience[2])
        nextStates.append(experience[3])

    return states, actions, rewards, nextStates
```

[19]:
```
observation = env.reset()
memory, actions = [], []
for i in range(100):
    state = preprocessObservation(env, observation)
    qValues = getAllQValues(state, regressors)
    action = getActionSoftMax(qValues)
```

```
    observation, reward, done, info = env.step(action)
    nextState = preprocessObservation(env, observation)
    memory.append([state, action, reward, nextState])
    actions.append(action)

plt.hist(actions)
plt.grid(axis = "y")
plt.ylabel("Frecuencia")
plt.xlabel("Acción")
plt.show()

batchSize = len(memory)*0.3
sampledActions = sampleMemory(memory, batchSize)[1]
plt.hist(sampledActions)
plt.grid(axis = "y")
plt.ylabel("Frecuencia")
plt.xlabel("Acción")
plt.show()
```

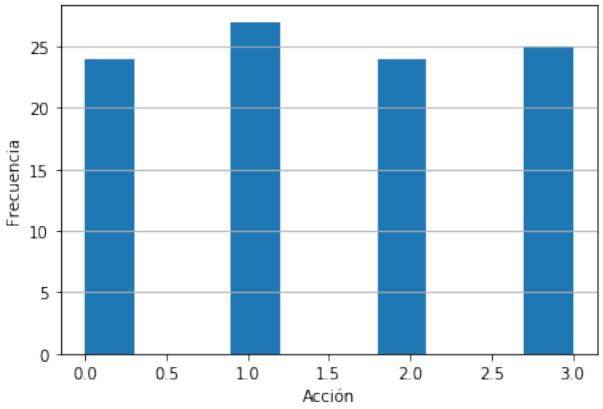


Figura 4.2: Histograma de acciones almacenadas en la memoria completa

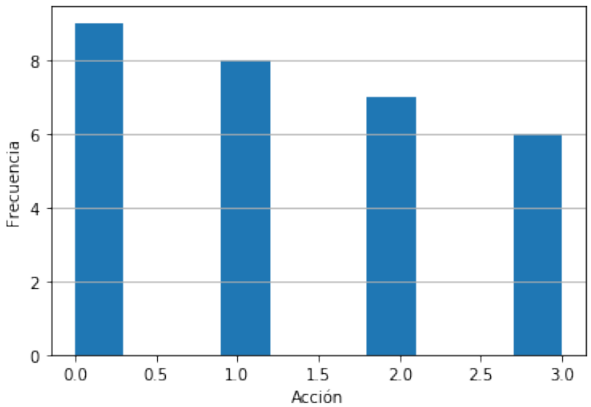


Figura 4.3: Histograma de acciones almacenadas en la memoria muestreda

A continuación, definiremos una función para depurar la salida y comprobar que todo

está funcionando perfectamente. Esta función no se utiliza en el bucle pero puede ser de utilidad para realizar variaciones al algoritmo y comprobarlas.

[20]:
```
#Shows the qvalue for each action for each state
def debugQ(env, regressors):
    numStates = env.observation_space.n
    numRows, numCols = int(np.sqrt(numStates)), int(np.sqrt(numStates))
 ↪ #square map
    aux = np.arange(numStates).reshape(numRows, numCols)
    print(aux)
    env.render()
    print("\t\tL \t\t\tD \t\t\tR \t\t\tU")
    for observation in range(numStates):
        state = preprocessObservation(env, observation)
        s = "position {}: {}".format(observation, getAllQValues(state,
 ↪regressors))
        print(s)
```

[21]:
```
debugQ(env, regressors)
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
 [12 13 14 15]]
  (Left)
SFFF
FHFH
FFFH
HFFG
                L                       D                       R
U
position 0: [0.5454636354151157, 0.5209242989022127, 0.4524626410851324,
0.5262700081143794]
position 1: [0.5454636354151157, 0.5209242989022127, 0.4524626410851324,
0.5262700081143794]
position 2: [0.5454636354151157, 0.5209242989022127, 0.4524626410851324,
0.5262700081143794]
position 3: [0.5454636354151157, 0.5209242989022127, 0.4524626410851324,
0.5262700081143794]
position 4: [0.5454636354151157, 0.5209242989022127, 0.4524626410851324,
0.5262700081143794]
position 5: [0.5454636354151157, 0.5209242989022127, 0.4524626410851324,
0.5262700081143794]
position 6: [0.5454636354151157, 0.5209242989022127, 0.4524626410851324,
0.5262700081143794]
position 7: [0.5454636354151157, 0.5209242989022127, 0.4524626410851324,
0.5262700081143794]
position 8: [0.5454636354151157, 0.5209242989022127, 0.4524626410851324,
```

```
0.5262700081143794]
position 9: [0.5454636354151157, 0.5209242989022127, 0.4524626410851324,
0.5262700081143794]
position 10: [0.5454636354151157, 0.5209242989022127, 0.4524626410851324,
0.5262700081143794]
position 11: [0.5454636354151157, 0.5209242989022127, 0.4524626410851324,
0.5262700081143794]
position 12: [0.5454636354151157, 0.5209242989022127, 0.4524626410851324,
0.5262700081143794]
position 13: [0.5454636354151157, 0.5209242989022127, 0.4524626410851324,
0.5262700081143794]
position 14: [0.5454636354151157, 0.5209242989022127, 0.4524626410851324,
0.5262700081143794]
position 15: [0.5454636354151157, 0.5209242989022127, 0.4524626410851324,
0.5262700081143794]
```

Una vez analizadas todas las funciones, vamos a dar el último paso antes de ver el bucle principal. Consiste en la inicialización de diferentes parámetros, como el ratio de descuento a 0.8. El número de steps para cada iteración de entrenamiento lo fijaremos a 12000 si estamos usando un árbol de decisión y 25000 cuando estimemos con regresores lineales. También, inicializaremos vectores para sacar estadísticas del rendimiento. Como en cualquier proyecto de esta magnitud, es importante tener un sistema de gestión de checkpoints. Para serializar los modelos, nosotros hemos usado la librería `pickle`. También, establecemos un contador para evitar oscilaciones.

[22]:
```python
# Create enviroment, using a random map
env = gym.make('FrozenLake-v0')
randomMap = getRandomMap(env)
env = gym.make('FrozenLake-v0', desc = randomMap)

#Initialize some variables
observation = env.reset()
actions = env.action_space
numActions = actions.n  #Possible actions: 4
memory = [] #state, action, reward, next_state
discountRate = 0.8
trainSteps = 12000   #25000 for linear regression, 12000 for tree.␣
 ↪Uncomment lines at preprocessObs and initRegressors

# Plot variables
iterRewards, qVals = [], []  #Usefull to plot
totalWinRate = []
games, step = 0, 0
iterWins, iterGames, trainIter = 0, 0, 0

# The first time we train we have to initialize ramdon weights␣
 ↪because sklearn forces to train before fit
firstTime = True
```

```
# Saver for checkpoints
fileName = 'saver_v7.7_dt.sav' #Save trained model in this file
if os.path.exists(fileName):
    bestRegressors = pickle.load(open(fileName, 'rb'))
    regressorsStar = copy.deepcopy(bestRegressors) #Load optimum␣
 ↪regressors.
    regressorsHat = copy.deepcopy(bestRegressors) #Load optimum␣
 ↪regressors.
    firstTime = False

# Avoid oscilations variables
maxTrainItersWithLowerWinrate = 4
maxWinRate, counter = -1*float('Inf'), maxTrainItersWithLowerWinrate
state = preprocessObservation(env, observation)
bestRegressors = initRegressors(state, numActions) #For optimizing␣
 ↪gradient
```

Por último en esta sección, tenemos el bucle principal del algoritmo. Es importante tener en cuenta la restricción a la hora de inicializar los regresores. Como hemos estudiado anteriormente, el algoritmo se puede dividir en dos etapas: una primera donde el agente juega varias partidas y una segunda en la que se entrena el modelo. En la primera etapa, extraeremos información del estado procesándolo, obtendremos la acción y ejecutaremos un `step` y procesaremos la recompensa y el siguiente estado. Toda esta información será guardada en la memoria. Cabe puntualizar, que la condición para pasar de una fase a otra se tiene que medir en steps. Esto es debido a que si fuerzas a jugar un cierto número de partidas, es posible que llegue a no completarse esa partida. En la segunda etapa, lo que haremos será ajustar la ecuación de Bellman. Recordemos que esta ecuación se formula así:

$$y^{(i)} = r^{(i)} + \gamma \max_{a'} Q(s'^{(i)}, a', \theta\hat{}) \tag{4.1}$$

En definitiva, nuestra meta es que el modelo acabe dando como salidas los Q-Values óptimos, por lo que tendremos que entrenarlo (`fit`) para utilizando la función de Bellman como labels de nuestro regresor. Para eso construimos las matrices `X` e `y`, que tendrán una columna por cada acción y una fila por cada experiencia de la memoria que analicemos. La matriz `X` sólo guarda los estados como entradas, mientras que la matriz `y` almacena las etiquetas. Para ello, de cada experiencia obtenemos el siguiente estado y hacemos una predicción para ver sus Q-Values, utilizando el regresor que está jugando (el modelo*), no el que está entrenando. Obtenida esa predicción, lo que haremos será simplemente entrenar el modelo$\hat{}$ que almacenará los Q-Values óptimos. Una vez comprendida esta parte, el resto es tangencial al algoritmo primario. Se utiliza un contador para evitar oscilaciones, que almacenará el regresor que mejor winrate tenga. En caso de acumular 5 iteraciones con un modelo con peor winrate, se restaurará el modelo con mejores resultados. Finalmente, se almacena este regresor en un fichero y se resetean las variables utilizadas para representar la eficiencia del algoritmo.

[23]:
```
print("games; winRate; maxWinRate | rewardMed; qValMed; expRate; counter") #View progress
while(True):
    step += 1
    state = preprocessObservation(env, observation)

    # Mandatory train before fit. First time we train we force to predict randomly
    if firstTime:
        regressorsStar = initRegressors(state, numActions) #Q*
        regressorsHat = initRegressors(state, numActions)  #Q^
        firstTime = False

    # Compute Qvals, choose action and apply that action to the enviroment
    qvaluesCurrentState = getAllQValues(state, regressorsStar)
    action, expRate = epsilonGreedy(trainIter, qvaluesCurrentState)
    qVals.append(qvaluesCurrentState[action]) #Plot
    observation, reward, done, info = env.step(action)
    nextState = preprocessObservation(env, observation)

    # Process reward
    if done:
        iterGames += 1
        if reward==1.0:
            processedReward = 1000.0 #If win: reward = 1000
            iterWins += 1
        else:
            processedReward = -1000.0  #If lost: reward = -1000
    else:
        processedReward = -1.0 #If neither win or lost: reward = -1

    # Store experience in replay memory
    memory.append([state, action, processedReward, nextState])

    #Each time the agent ends a game generate a random map
    if done:
        randomMap = getRandomMap(env)
        env = gym.make('FrozenLake-v0', desc = randomMap)
        observation = env.reset()
        games  += 1 #Plot
        iterRewards.append(processedReward) #Plot

    #Train each trainSteps
    # Ojo, esto no se puede hacer por "games", porque puede ser que un game no se termine y se pase varias veces por aqui!!!
    if step%trainSteps !=0: continue
```

```
    batchSize = len(memory)*1.0    #study 100% of steps in memory
    trainStates, trainActions, trainRewards, trainNextStates =␣
↪sampleMemory(memory, batchSize)

    X = [[],[],[],[]]
    y = [[],[],[],[]]
    for state, action, reward, nextState in zip(trainStates,␣
↪trainActions, trainRewards, trainNextStates):
        qvaluesNextState = getAllQValues(nextState, regressorsStar)
        maxQvaluesNextState = np.max(qvaluesNextState)
        X[action].append(state)
        y[action].append(reward +␣
↪discountRate*maxQvaluesNextState)#Puede ser que falte el continue

    # Train Q^
    for action in range(numActions):
        if len(X[action]) == 0: continue  #No sé por qué hay veces que␣
↪una acción no se coge
        regressorsHat[action].fit(X[action], y[action])

    regressorsStar = copy.deepcopy(regressorsHat)

    # Plot variables
    winRate = iterWins/iterGames
    rewardMed = np.mean(iterRewards)
    qValMed = np.mean(qVals)

    #Checkpoints avoiding oscilate for more than 5 iters
    if winRate > maxWinRate: #If better winrate store new regressors
        maxWinRate = winRate
        counter = maxTrainItersWithLowerWinrate  #max num of training␣
↪iters with lower winrate
        bestRegressors = copy.deepcopy(regressorsHat)
    else:
        if counter == 0: #If lower winrate 5 iters restore best␣
↪regressors
            counter = maxTrainItersWithLowerWinrate
            regressorsStar = copy.deepcopy(bestRegressors) #Load␣
↪optimum regressors.
            regressorsHat = copy.deepcopy(bestRegressors) #Load␣
↪optimum regressors.
        else:
            counter -=1

    #Show progress
```

```
    line = "Checkpoint {:d}: {}; {:.3f}; {:.3f} | {:.3f}; {:.3f}; {:.
↪3f}; {}".format(trainIter, games, winRate, maxWinRate,
                                                                    ␣
↪            rewardMed, qValMed, expRate, counter)
    print(line)

    totalWinRate.append(winRate)  #Plot

    #Save trained models
    pickle.dump(bestRegressors, open(fileName, 'wb'))

    #Reset training and plot variables
    memory = []
    iterRewards, qVals = [], [] #Usefull to plot
    iterWins, iterGames = 0, 0
    trainIter += 1 #Need for greedy


env.close()
```

4.2. Implementación de Qbert

Primero importamos las librerías necesarias.

- gym será la principal ya que proporcionará el entorno en el que testearemos nuestro algoritmo de reinforcement learning.
- tensorflow y concretamente keras proporcionarán los métodos para crear y entrenar el modelo.
- numpy para trabajar con las operaciones matemáticas.
- matplotlib la usaremos para representar nuestros resultados. Matplotlib suele abrir las representaciones en una ventana aparte. Con %matplotlib inline le decimos que los muestre en la misma "superficie" que muestra los prints. Esto es necesario al estar ejecutando el notebook en un servidor headless. Más adelante se profundiza un poco en esta idea.
- time nos ayudará a ver con más claridad los pasos que va dando nuestro algoritmo, pudiendo introducir un delay en el código para observar bien el estado de la pantalla.
- display para poder limpiar la "pantalla" de salida.
- os para poder acceder a los ficheros del equipo y guardar parámetros de nuestro modelo.
- copy para copiar los elementos en lugar de utilizar referencias. Necesario para guardar los coeficientes del mejor modelo encontrado.
- pickle para serializar y guardar los modelos entrenados.

[1]:
```
import gym

import tensorflow as tf
from tensorflow import keras

import numpy as np
%matplotlib inline
import matplotlib.pyplot as plt
import time
from IPython import display

import os
import copy
import pickle

#Make this notebook's output stable across runs
seed = 6844
np.random.seed(seed)
tf.random.set_seed(seed)
np.set_printoptions(threshold=np.inf)  # For seeing the whole arrays
```

El primer paso es investigar un poco el entorno para conocer cuál es el objetivo del juego, ver cuántos enemigos hay, las vidas que tienes, cómo se mueve el agente, etc.

Viendo gameplays del juego encontramos que consiste en saltar de base en base para recorrer toda la pirámide. En la primera pantalla vemos que sólo hay 2 enemigos, pero conforme avanzan los niveles aparecen más tipos de enemigos. Algunos te cambian el color de las bases por las que has pasado y para pasar la pantalla hay que volver a marcarlas. Podríamos utilizar el color de cada base en cada momento para modelar el estado. El problema es que en cada nivel los colores cambian, pero más adelante veremos cómo afrontamos esto. Además podemos ver que en cada episodio se pueden perder 3 vidas como máximo. Si mueres una cuarta vez ya se termina el juego. Puede perderse una vida tanto cuando un enemigo te toca como cuando te caes de la pirámide. También, podemos ver que hay unas barras laterales que se activan desde las últimas plataformas desde el penúltimo nivel. Estas barras solo pueden ser usadas una vez por pantalla y te llevan al principio de la pirámide.

Así que vamos a ponernos manos a la obra. Creamos el entorno más sencillo posible para ver cómo va avanzando el juego. En la librería OpenAI Gym, cada juego tiene varias versiones y en la versión por defecto (Qbert-v0), el simulador coge una acción y la repite entre 1 y 3 steps. Utilizaremos QbertDeterministic que evita esto para que todo sea menos aleatorio, es decir, más fácil. En esta web se muestran los entornos disponibles hasta la fecha: https://github.com/openai/gym/wiki/Table-of-environments

[2]:
```
env = gym.make("QbertDeterministic-v0")
rawObs = env.reset()
while True:
    rawObs, reward, done, info = env.step(env.action_space.sample())
    env.render()
```

```
    if done:
        break
```

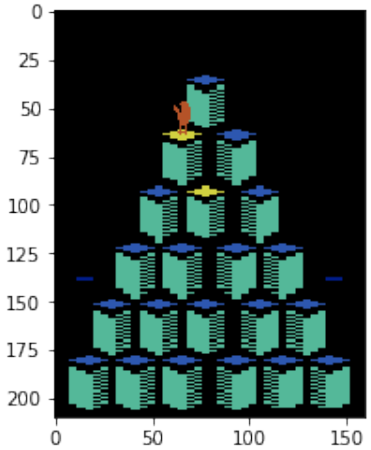
```
.
.
.
    NoSuchDisplayException: Cannot connect to "None"
```

Lo primero que obtenemos es un error, buen comienzo. Investigando vemos que cuando trabajamos con jupyter notebook en un servidor que sin pantalla (headless) no podemos llamar a env.render(). Esto se debe a que Gym por defecto elige que se abra una nueva ventana para mostrar el juego. Sin embargo, la pantalla que busca Gym para que muestre el resultado no existe, y en su lugar queremos que la muestre en el navegador del cliente. Vamos a implementar una pequeña función que solucione esto.

[3]:
```
def render(env, clearOutput = True):
    img = plt.imshow(env.render(mode='rgb_array'))
    img.set_data(env.render(mode='rgb_array'))
    display.display(plt.gcf())
    if clearOutput:
        display.clear_output(wait=True)
```

Ahora sí que ya estamos listos para ver cómo avanza el videojuego. Para no ocupar demasiado, en la versión en papel donde no se pueden ver vídeos solo pondremos los fotogramas que nos resulten interesantes para describir el código.

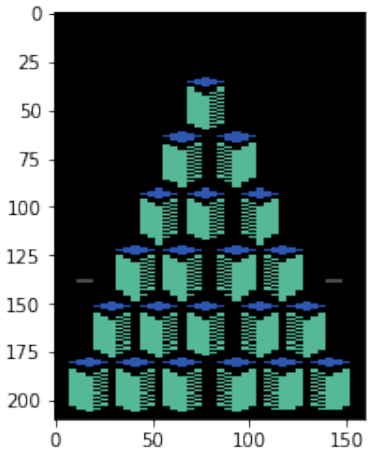
[4]:
```
env = gym.make("QbertDeterministic-v0")
rawObs = env.reset()
while True:
    rawObs, reward, done, info = env.step(env.action_space.sample())
    render(env)
    if done:
        rawObs = env.reset()
        break
```



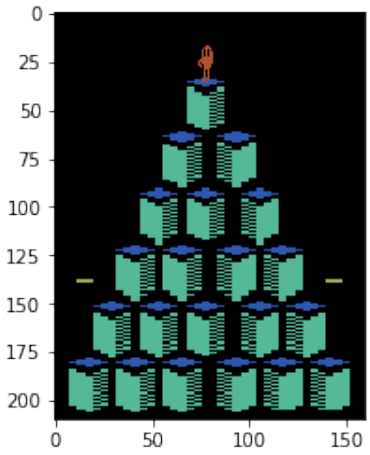
Vemos que al principio de cada episodio hay un tiempo donde no aparece el personaje y ninguna acción que hagamos parece tener repercusión. Para eso lo que hacemos es ver los frames de cada episodio y localizar cuándo aparece el personaje. Ese será el número de frames que tenemos que saltarnos. El motivo de saltarnos estos pasos es que, de cara al tiempo de entrenamiento, introducen un retraso innecesario, lo que hace el algoritmo menos eficiente

[5]:
```
env = gym.make("QbertDeterministic-v0")
env.seed(seed)
rawObs = env.reset()
maxSteps = 60

for step in range(maxSteps):
    rawObs, reward, done, info = env.step(env.action_space.sample())
    print(step)
    render(env, False)
    if done: break
```

```
    .
    .
    .
36
```




```
37
```



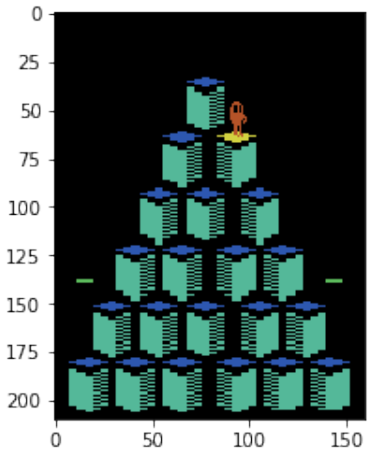
.
.
.

Vemos que hasta el frame 37 no aparece el personaje y que ninguna acción que hagamos tiene repercusión, así que nos saltaremos los primeros 37 steps de cada episodio.

Ahora debemos mapear cada acción del juego con la notación de la librería gym. Para ello gym te da la posibilidad de saber el número total de acciones para este entorno mediante la función `env.action_space.n = 6` . Ahora falta saber cuál es cada una para lo que vamos probando con cada acción.

[6]:
```
env = gym.make("QbertDeterministic-v0")
env.seed(seed)
rawObs = env.reset()
maxSteps = 50
step = 0
skipFrames = 37
action = 0

while action < 6:
    step += 1
    if step < skipFrames:
        rawObs, reward, done, info = env.step(env.action_space.sample())
        continue
    render(env)
    print(action)
    rawObs, reward, done, info = env.step(action)   #Try this from 0␣
 ↪to env.action_space.n

    if done or step == maxSteps-1:
        env.reset()
        action += 1
        step = 0
```

```
3
```



Relacionando el sentido de cada acción con el simulador web de retrogames, vemos que el significado de cada acción es el siguiente:

- 0 = no hace nada (NOOP)
- 1 = no hace nada (FIRE)
- 2 = up
- 3 = right
- 4 = left
- 5 = down

Ahora procederemos a tratar de obtener un estado que poder pasarle como input de la red neuronal al algoritmo. Este estado tiene que ser una representación de la pantalla que dé información suficiente para describir el estado actual del juego. La información que almacenaremos en el estado, como mínimo, será:

- Color de cada plataforma
- Plataforma donde se encuentra el personaje principal
- Para cada enemigo: color del enemigo, posición del enemigo
- Posición de cada barra lateral

También, debemos llevar cuidado con una cosa. Por ejemplo, si estamos jugando a Pong, un estado que represente la pantalla en un momento concreto no aportará mucha información, porque no se sabe la dirección de la bola. Más adelante veremos cómo tratamos este aspecto en QBert.

Lo primero que tenemos que hacer es preprocesar la imagen para obtener una más fácil de tratar. Originalmente la pantalla utiliza tres canales de colores pero a nosotros nos vale con uno: `rawObs[: , :, 0]`. Una vez hecho esto obtenemos los diferentes colores que puede tener una pantalla con `np.unique(obs)` e identificaremos a qué color corresponde cada elemento de la pantalla. Junto a cada color (`fullColours`) vamos a ir anotando el grupo de píxeles que están de ese color (`pixelColours`).

[7]:
```
env = gym.make("QbertDeterministic-v0")
env.seed(seed)
rawObs = env.reset()
maxSteps = 322
maxIterations = 50
skipFrames = 37

fullColours, pixelColours = [], []
for iteration in range(maxIterations):
    for step in range(maxSteps):
        obs = rawObs[:,:,0]
        if step < skipFrames or 181 not in obs:
            rawObs, reward, done, info = env.step(0)
            continue

        colours = np.unique(obs)
        for colour in colours:
            if colour not in fullColours:
                fullColours.append(colour)
                pixels = np.argwhere(obs == colour)
                pixelColours.append(pixels)  #pixel where each colour␣
 ↪appears

        rawObs, reward, done, info = env.step(env.action_space.sample())
        if done:
            env.reset()

for colour, pixels in zip(fullColours, pixelColours):
    print("COLOR: ", colour, "\t | \t PIXEL: ", pixels[0])
```

```
COLOR:  0       |       PIXEL:  [0 0]
COLOR:  45      |       PIXEL:  [34 76]
COLOR:  84      |       PIXEL:  [39 68]
COLOR:  160     |       PIXEL:  [138  12]
COLOR:  181     |       PIXEL:  [17 78]
COLOR:  109     |       PIXEL:  [138  12]
COLOR:  48      |       PIXEL:  [138  12]
COLOR:  180     |       PIXEL:  [138  12]
COLOR:  92      |       PIXEL:  [138  12]
COLOR:  50      |       PIXEL:  [138  12]
COLOR:  117     |       PIXEL:  [138  12]
COLOR:  210     |       PIXEL:  [ 6 35]
COLOR:  169     |       PIXEL:  [138  12]
COLOR:  127     |       PIXEL:  [138  12]
COLOR:  78      |       PIXEL:  [138  12]
COLOR:  20      |       PIXEL:  [138  12]
COLOR:  66      |       PIXEL:  [138  12]
COLOR:  146     |       PIXEL:  [40 93]
```

```
COLOR:  24      |       PIXEL:  [138  12]
COLOR:  132     |       PIXEL:  [138  12]
COLOR:  101     |       PIXEL:  [138  12]
COLOR:  188     |       PIXEL:  [138  12]
COLOR:  149     |       PIXEL:  [138  12]
COLOR:  104     |       PIXEL:  [138  12]
COLOR:  51      |       PIXEL:  [138  12]
COLOR:  212     |       PIXEL:  [138  12]
COLOR:  236     |       PIXEL:  [138  12]
COLOR:  184     |       PIXEL:  [138  12]
COLOR:  252     |       PIXEL:  [67 89]
COLOR:  151     |       PIXEL:  [138  12]
COLOR:  228     |       PIXEL:  [138  12]

.
.
.

COLOR:  223     |       PIXEL:  [138  12]
COLOR:  195     |       PIXEL:  [138  12]
```

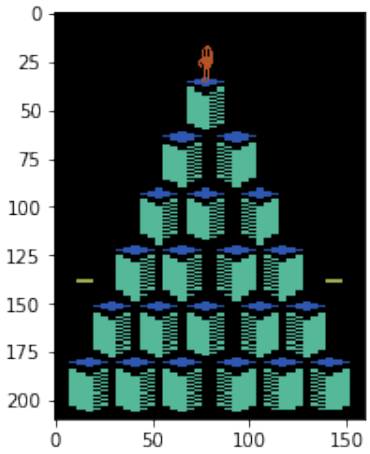
A priori, observamos que hay muchos más colores que elementos vemos en la pantalla, algo extraño. Vemos que el píxel [138 140] adopta muchos colores a lo largo de la partida, por lo que el elemento de la pantalla situado en ese píxel cambia constantemente de color. Contemplando las ejecuciones anteriores vemos que las barras laterales van cambiando de color dentro de la partida. Para asegurarnos vamos a ver la pantalla justo en el primer momento en el que aparece un color que no estaba antes, para estudiarlos casi todos. Vamos a evitar todos los colores de las barras laterales, que son los que aparecen en el píxel [138 140]. Esto nos dará una idea de a qué corresponde cada color.

Para evitar información redundante, hemos puesto que evite nombrar los colores de las barras laterales con `if not np.array_equal(pixels, pixelColours[3])`. El problema es que cuando hay un píxel de una barra lateral que no es de ese color, aparecerá en la salida. Esto pasa por ejemplo, cuando el agente toca cualquier píxel de una barra lateral.

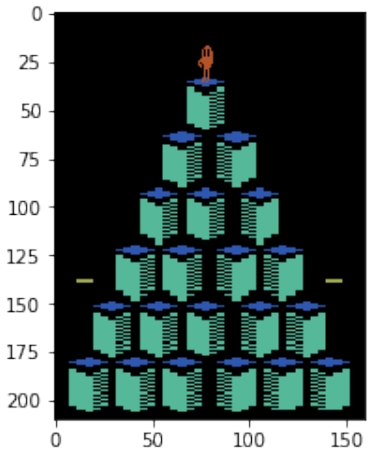
[8]:
```
env = gym.make("QbertDeterministic-v0")
env.seed(seed)
rawObs = env.reset()
maxSteps = 322
maxIterations = 50
skipFrames = 37

fullColours, pixelColours2, imgs = [], [], []
for iteration in range(maxIterations):
    for step in range(maxSteps):
        obs = rawObs[:,:,0]
        if step < skipFrames or 181 not in obs:
            rawObs, reward, done, info = env.step(0)
            continue
```

```
        colours = np.unique(obs)
        for colour in colours:
            if colour not in fullColours:
                pixels = np.argwhere(obs == colour)
                if not np.array_equal(pixels, pixelColours[3]): ␣
↪#Avoid lateral bars
                    fullColours.append(colour)
                    pixelColours2.append(pixels)  #pixel where each␣
↪colour appears
                    imgs.append(rawObs)

        rawObs, reward, done, info = env.step(env.action_space.sample())
        if done:
            env.reset()
cont = 0
for colour, img in zip(fullColours, imgs):
    print("COLOR: ", colour, "CONT: ", cont)
    plt.imshow(img)
    plt.show()
    cont += 1
```

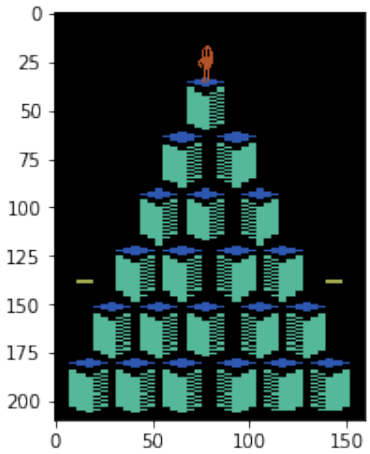
```
COLOR:  0 CONT:  0
```



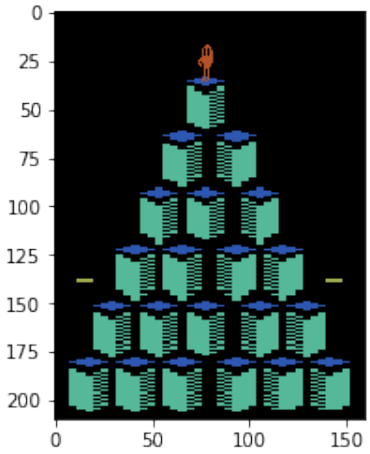

```
COLOR:  45 CONT:  1
```



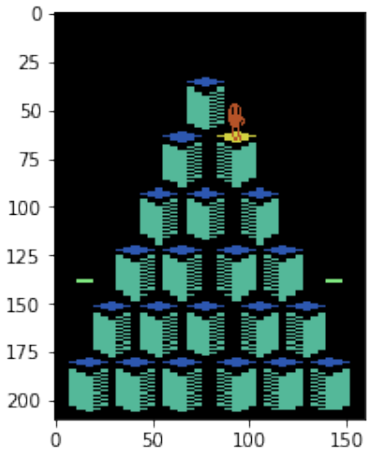

```
COLOR:  84 CONT:  2
```



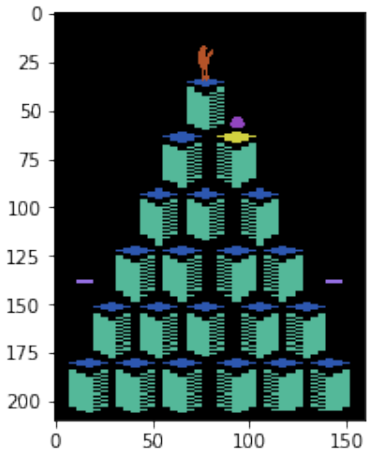

```
COLOR:  181 CONT:  3
```



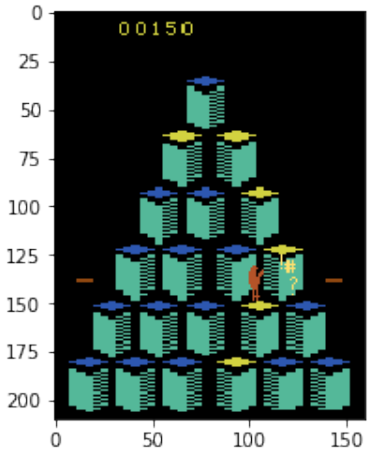

```
COLOR:  210 CONT:  4
```



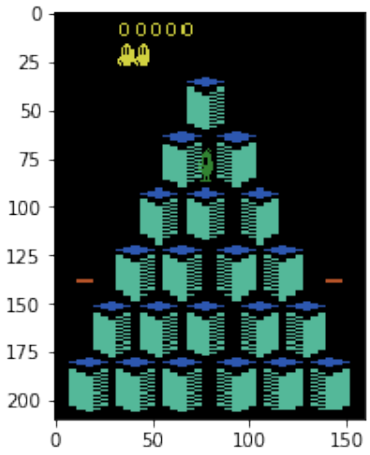

```
COLOR:  146 CONT:  5
```




```
COLOR:  252 CONT:  6
```




```
COLOR:  50 CONT:  7
```

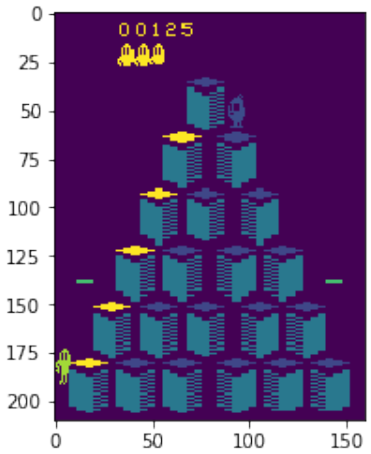


Observando las últimas capturas, aparece el color de una plataforma marcada (210), un enemigo redondo (color 146), la exclamación de cuando el personaje muere (color 252) y un enemigo verde (color 50). Sin embargo, las primeras 4 capturas, los colores que aparecen son 0, 45, 84 y 181, que corresponderán cada uno a elementos como el fondo, las bases, las plataformas o el agente. Para identificar cada color vamos a coger una imagen en la que haya el máximo de elementos posibles e iremos suprimiendo color a color e iremos viendo qué elementos van desapareciendo.

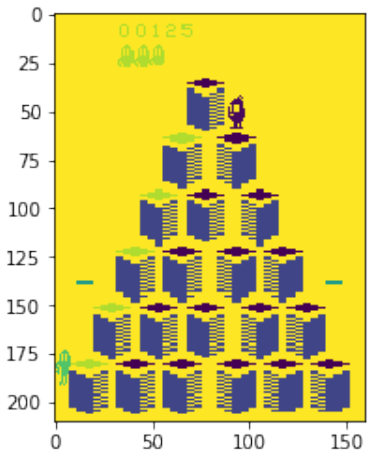
[9]:
```
env = gym.make("QbertDeterministic-v0")
env.seed(seed)
rawObs = env.reset()
maxSteps = 80  #Con deterministic (skip=37) lo mejor es 80
skipFrames = 37

for step in range(maxSteps):
    rawObs, reward, done, info = env.step(5)
    if done: break


lastObs = rawObs.copy()[:,:,0]   #list=old_list is a reference. Change
 ↪something in list will change old_list
colours = np.unique(lastObs)
print("Diferents colours in the screen: ", colours)

print("Screen with default colours")
plt.imshow(lastObs)
plt.show()

pixelsGroupByColour = [[] for i in range(len(lastObs))]
for index, colour in enumerate(np.unique(colours)):
    pixelsGroupByColour[index].append(np.argwhere(lastObs == colour))

count = 0
for ps in pixelsGroupByColour:
```

```
    for pp in ps:
        for p in pp:
            lastObs[p[0],p[1]] = 232
        print("Screen without colour" ,colours[count])
        count += 1
        plt.imshow(lastObs)
        plt.show()
```

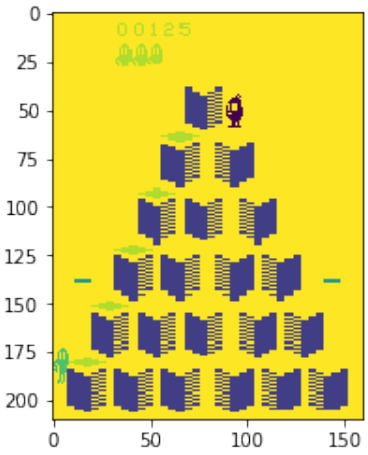
```
Diferents colours in the screen:  [  0  45  50  84 148 181 210]
Screen with default colours
```



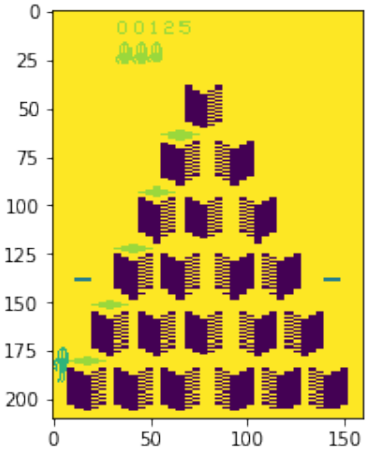

```
Screen without colour 0
```



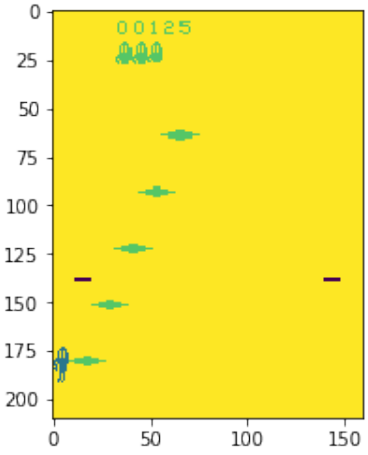

```
Screen without colour 45
```



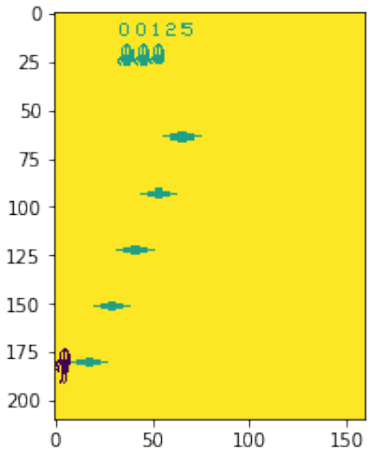
Screen without colour 50



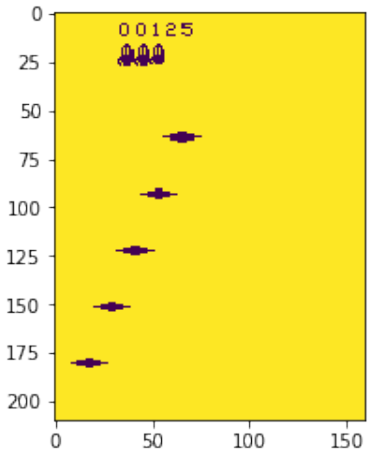
Screen without colour 84



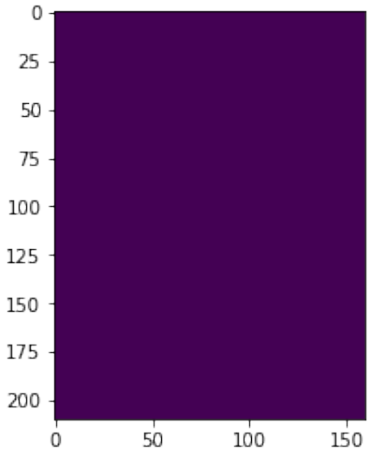
Screen without colour 148




```
Screen without colour 181
```




```
Screen without colour 210
```



Vemos que los diferentes colores significan lo siguiente:

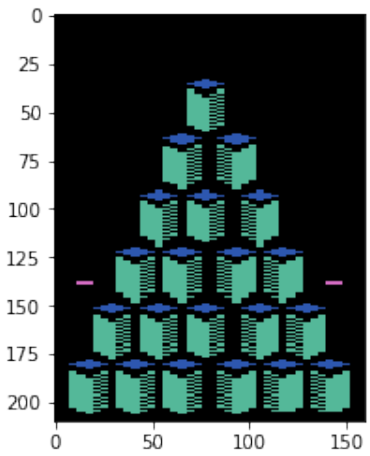
- 0 = Fondo de pantalla
- 45 = Base de la plataforma por defecto (sin pasar por ella)

- 50 = Enemigo 1
- 84 = Cilindro de la plataforma
- 146 = Enemigo 2
- 148 = Barra lateral
- 181 = PJ
- 210 = Base de la plataforma visitada y marcador de puntos y vidas
- 252 = Exclamación cuando el agente muere

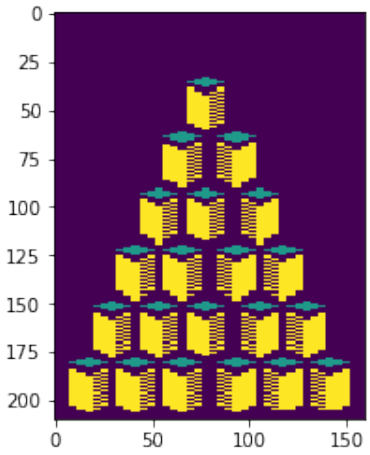
Antes, nos hemos dado cuenta de que las barras laterales van cambiando de color. Lo que vamos a hacer es fijarlas a un color constante. Primero necesitamos saber el color de las barras laterales cuando reiniciamos la pantalla para poder sacar los píxeles que componen exáctamente las barras laterales.

[10]:
```
env = gym.make("QbertDeterministic-v0")
rawObs = env.reset()
obs = rawObs[:,:,0]
colours = np.unique(obs)
print(colours)
render(env, False)
for px, py in np.argwhere(obs == 212):
    obs[px, py] = 0
plt.imshow(obs)
```

```
[  0  45  84 212]
```



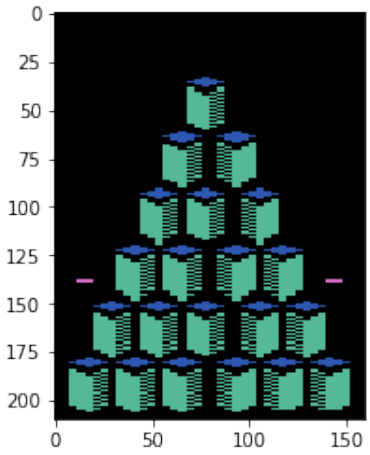
[10]:
```
<matplotlib.image.AxesImage at 0x7ff0567ce390>
```



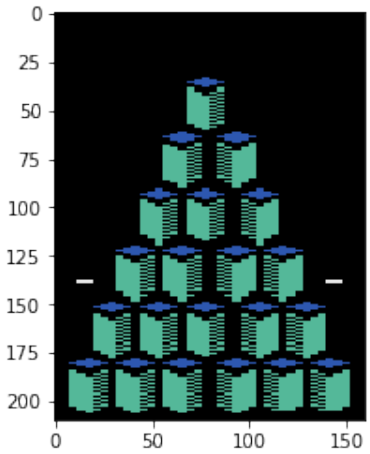
Vemos que el primer color que adoptan las barras laterales, justo después de hacer `env.reset()`, es 212. Definimos una función que preprocese la pantalla para quedarnos con un solo canal y cambiar los colores de las barras laterales. Para eso, obtendremos los píxeles de las barras laterales y los ponemos a color 232 siempre y cuando no estén del color del fondo.

[11]:
```
rawObs = env.reset()    #Cant reset because then lateral bars colour !=
 ↪148
lateralBarsPixels = np.argwhere(rawObs[:,:,0] == 212)

def preprocessImg(rawObs, debug=False):
    obs = rawObs.copy()
    for p in lateralBarsPixels:
        if not np.array_equal(obs[p[0], p[1]], np.array([0, 0, 0])):
            obs[p[0], p[1]] = np.array([232, 232, 232])

    if debug:
        print("Screen with changed lateral bars: ")
        plt.imshow(obs)
        plt.show()

    return obs[:,:,0]

print("Screen without preprocessing: ")
plt.imshow(rawObs)
plt.show()

obs = preprocessImg(rawObs, debug = True)
print("Preprocessed screen: ")
plt.imshow(obs)
plt.show()
```

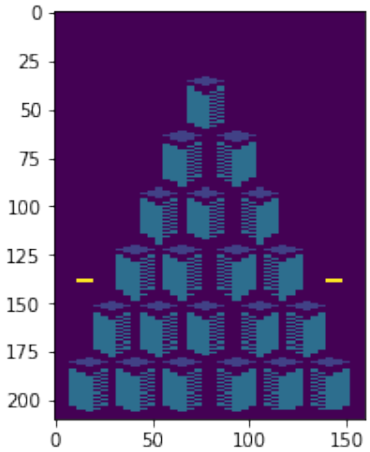
```
Screen without preprocessing:
```



`Screen with changed lateral bars:`



`Preprocessed screen:`



En definitiva, la pantalla queda con esta gama de colores:

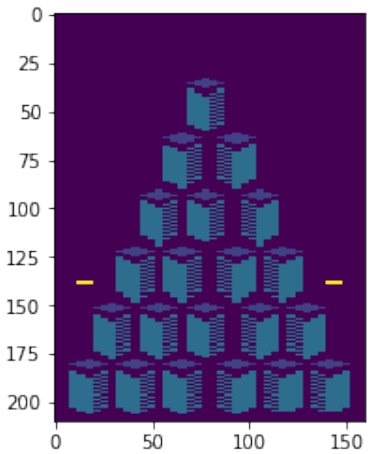
- 0 = Fondo de pantalla
- 45 = Base de la plataforma por defecto (sin pasar por ella)

- 50 = Enemigo 1
- 84 = Cilindro de la plataforma
- 146 = Enemigo 2
- **232** = Barra lateral
- 181 = PJ
- 210 = Base de la plataforma visitada y marcador de puntos y vidas
- 252 = Exclamación cuando el agente muere

Resulta obvio, pero hay que tener en cuenta que este cambio no se verá cuando llamemos a render(env) porque ahí coge la pantalla tal cual del entorno, no la preprocesada. Entonces, las barras latelares seguirán cambiando de color cuando pintemos la pantalla, pero para nuestro algoritmo serán constantes a 232.

Ahora necesitamos saber qué píxeles componen cada base. Para ello reseteamos el entorno (para que todas las bases tengan el mismo color) y vemos qué píxeles están pintados del color 45.

[12]:
```
rawObs = env.reset()
obs = preprocessImg(rawObs)
plt.imshow(obs)
plt.show()

np.argwhere(obs == 45.0)
```



[12]:
```
array([[ 34,  76],
       [ 34,  77],
.
.
.
       [182, 142],
       [182, 143]])
```

Atendiendo a la separación entre los píxeles que están coloreados del color 45, podemos localizar el primer píxel de cada plataforma. Lo que queremos es hacer una función que nos dé la base más cercana de un píxel dado. Para ello, nos quedaremos con el último píxel de cada base. Cabe destacar que la primera dimensión representa la altura y la segunda el ancho.

- primera fila: [38, 79]
- segunda fila: [66, 65], [66, 95]
- tercera fila: [95, 55], [95, 79], [95, 107]
- cuarta fila: [124, 42], [124, 65], [124, 93], [124, 117]
- quinta fila: [153, 29], [153, 53], [153, 77], [153, 105], [153, 129]
- sexta fila: [182, 17], [182, 42], [182, 65], [182, 93], [182, 117], [182, 141]

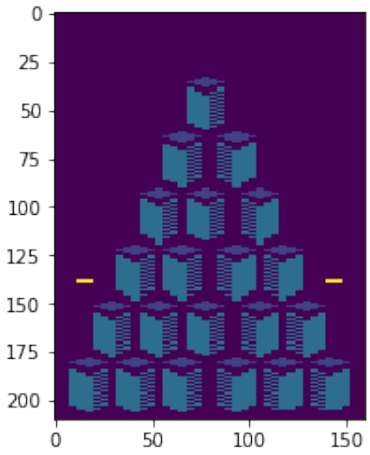
Ahora necesitaremos una función que dado un píxel nos devuelva cuál es la base más cercana. Para ello, ponemos en un array un píxel de cada base `basePosition` y nos damos cuenta de que tenemos 21 bases por lo que las bases las numeraremos en función del índice dentro del array `basePosition`. Para eso comparamos la altura y vemos qué base es más cercana. La variable `nearestLineBaseIndex` nos dice en qué fila se halla el agente. Ahora tenemos que encontrar en qué base concreta estamos. La primera línea empieza en la posición 0 y tiene 1 base. La segunda línea empieza en la posición 1 y tiene 2 bases. La tercera empieza en 3 y tiene 3 bases. La cuarta línea empieza en la posición 6 y tiene 4 bases. Si observamos la sucesión vemos que sigue la fórmula $\frac{n^2+n}{2}$. Con eso extraemos las bases pertenecientes a la línea que marca `nearestLineBaseIndex`. Ahora obtendremos `nearestBaseIndex`, que nos dice en qué base de todo el mapa se encuentra el agente.

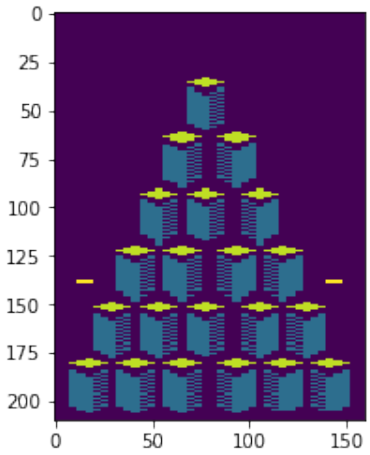
[13]:
```
basePosition = np.array([
            [ 38,  79],
            [ 66,  65], [ 66,  95],
            [ 95,  55], [ 95,  79], [ 95, 107],
            [124,  42], [124,  65], [124,  93], [124, 117],
            [153,  29], [153,  53], [153,  77], [153, 105], [153, 129],
            [182,  17], [182,  42], [182,  65], [182,  93], [182, 117],␣
 ↪[182, 141]
])

numBases = len(basePosition)

def getBaseIndex(pixel):
    #Check which line we are in
    pixelDistance = np.unique(basePosition[:,0]) - pixel[0]
    #Keep distance if its positive. If positive base will be behind␣
 ↪pixel. Get minimum of positive distances
    nearestLineBaseIndex = np.argmin([np.inf if (distance < 0) else␣
 ↪distance for distance in pixelDistance])
    # Line base index [0, 1, 3, 6, 10, 15] --> sequence = (n^2 + n)/2
    firstLineBase = int( (nearestLineBaseIndex * nearestLineBaseIndex +␣
 ↪nearestLineBaseIndex)/2 )
```

```
    lastLineBase = int( ( ((nearestLineBaseIndex+1) *␣
↪(nearestLineBaseIndex+1) + nearestLineBaseIndex+1)/2) )
    nearestBaseIndex = np.argmin(np.absolute(pixel[1] -␣
↪basePosition[firstLineBase:lastLineBase][:,1]))
    baseIndex = int( ( (nearestLineBaseIndex * nearestLineBaseIndex +␣
↪nearestLineBaseIndex)/2) + nearestBaseIndex)

    return baseIndex
```

Ahora reutilizaremos esta función para obtener qué píxeles están asociados a cada base. Crearemos otra función que agrupe los píxeles en arrays distribuidos según su base de esta forma: `pixelsGroupByBase` = [[píxeles que componen la base 0], [píxeles que componen la base 1], [píxeles que componen la base 2]...]. Para evitar que una base pueda haber sido pisada y haber cambiado de color, necesitamos que todas estén con el color por defecto (45). Por lo tanto reiniciaremos el entorno.

[14]:
```
env = gym.make("QbertDeterministic-v0")
rawObs = env.reset()
obs = preprocessImg(rawObs)
plt.imshow(obs)
plt.show()

def getBasePixels(obs):
    basePixelsFull = np.argwhere(obs == 45.0)

    pixelsGroupByBase = [[] for _ in range(numBases)]
    for basePixel in basePixelsFull:
        pixelPos = getBaseIndex(basePixel)
        pixelsGroupByBase[pixelPos].append(list(basePixel))
    return pixelsGroupByBase

for ps in getBasePixels(obs):
    for p in ps:
        obs[p[0],p[1]] = 210
plt.imshow(obs)
plt.show()
```

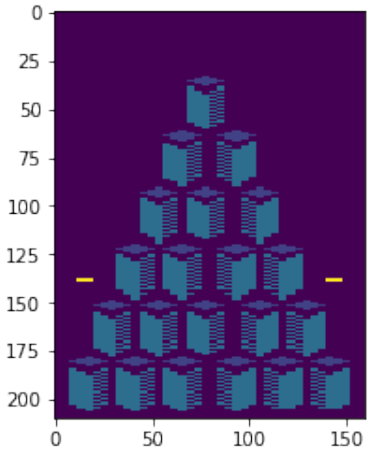




Una vez que tenemos todos los píxeles que forman cada base en `pixelsGroupByBase`, procedemos a obtener el color de cada base. Sería mucho más sencillo obtener el color de un píxel representativo de cada base, pero hay veces (por ejemplo cuando morimos) en las que ese píxel está dibujando el agente y esto puede falsear el resultado. Por eso vamos a observar toda la base y coger el color que más se repita.

[17]:
```
env = gym.make("QbertDeterministic-v0")
rawObs = env.reset()
obs = preprocessImg(rawObs)
plt.imshow(obs)
plt.show()

pixelsGroupByBase = getBasePixels(obs)

def getBaseColours(obs):
    baseColours = []
    currentBaseColour = []
    for basePixels in pixelsGroupByBase:
        for pixel in basePixels:
            currentBaseColour.append(obs[pixel[0], pixel[1]])
        colurs, freq = np.unique(currentBaseColour, return_counts=True)
```

```
        currentBaseColour = []
        baseColours.append(colurs[np.argmax(freq)])

    return baseColours
baseColours = getBaseColours(obs)
np.array(baseColours).reshape(1,-1)
```



[17]:
```
array([[45, 45, 45, 45, 45, 45, 45, 45, 45, 45, 45, 45, 45, 45, 45, 45,
        45, 45, 45, 45, 45]], dtype=uint8)
```

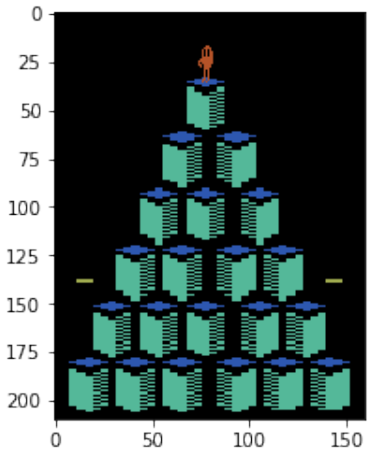
Una vez que tenemos el color de cada plataforma guardaremos la plataforma en la que está el personaje principal. Obtenemos en qué píxeles está el agente (color 181) y nos quedamos con el último píxel, que será el más cercano a la base sobre la que está el agente: `np.argwhere(obs == 181)[-1]`.

Una vez tenemos estas funciones, la primera parte del estado está casi completada, sabemos dónde está el agente y qué bases están coloreadas. Uno de los problemas que hemos anticipado es que en los siguientes niveles los colores cambian, entonces la representación del estado podría fallar. Para evitar eso, lo que vamos a hacer es reiniciar la primera pantalla siempre que se termine el episodio.

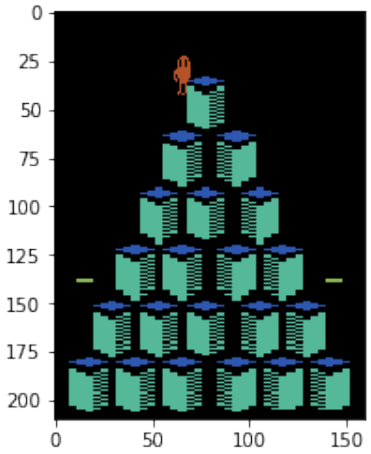
[18]:
```
env = gym.make("QbertDeterministic-v0")
env.seed(seed)
rawObs = env.reset()
maxSteps = 100
skipFrames = 37

for step in range(maxSteps):
    if step < skipFrames:
        rawObs, reward, done, info = env.step(env.action_space.sample())
        continue
    rawObs, reward, done, info = env.step(5)
    obs = preprocessImg(rawObs)
    if 181 in obs:
        pjBottom = np.argwhere(obs == 181)[-1] #PJ bottom
```

```
        print(getBaseIndex(pjBottom))
    render(env, False)
    if done: break
```

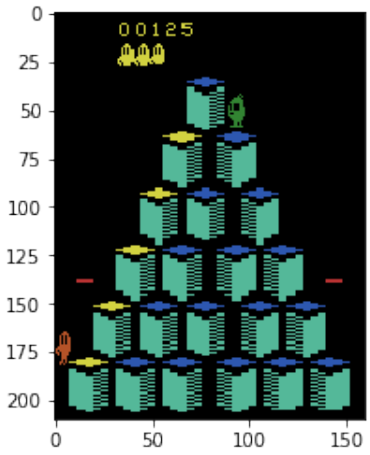
0



.
.
.
1



.
.
.
15



Con esta información vamos a tratar de hacer un pequeño heurístico que se pase el primer nivel sin tener en cuenta los enemigos, para así ver recompensas obtenidas al pisar una plataforma, al perder una vida y al pasar de nivel. Con esto más adelante podremos reiniciar cada vez que se pase el nivel para que empiece desde el primero.

El heurístico consistirá en recorrer primero las bases de la derecha (acción 3). Una vez estemos en la última base (número 21) pasaremos por las dos filas de abajo, intercalando subir y bajar hacia la izquierda (acciones 4 y 5). Una vez en la primera base de la última fila (16) lo que haremos será subir por ese lateral (acción 2) hasta llegar a la primera base. Ahí bajaremos una vez hacia la derecha a la base 3 (acción 3). Terminaremos el nivel recorriendo el rombo central, es decir, las bases 5, 8, 13 y 9.

Nota: Ver la figura 1.3 para entender la numeración de las bases

[19]:
```
def getActionHeuristic(currentBase, firstRound):
    action = 0
    if currentBase in [0, 5, 9, 7]:
        action = 3
    elif currentBase in [13, 11, 4]:
        action = 5
    elif currentBase in [20, 19, 18, 17, 16]:
        action = 4
    elif currentBase in [15, 6, 3, 1]:
        action = 2
    if firstRound[currentBase]:
        if currentBase in [2, 14]:
            action = 3
        elif currentBase in [12, 10]:
            action = 5
        firstRound[currentBase] = 0
    else:
        if currentBase in [14, 2]:
            action = 5
        elif currentBase in [10, 12]:
            action = 2
```

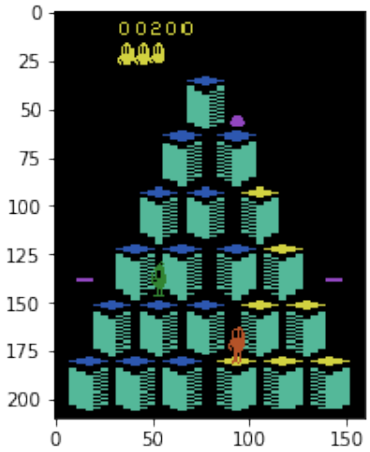
```
    return action, firstRound
```

Y ahora, ¿qué pasa con lo que comentábamos del movimiento anteriormente? Bien pues en este entorno, el movimiento puede crear problemas cuando estamos en medio de un salto de una base a otra. Cuando pulsamos el botón de saltar a otra base, el personaje inicia la acción, pero durante el salto hay varios steps en los que pulsar una acción no tiene repercusión. En teoría, sólo se puede aplicar una acción cuando el personaje llega a una base. El problema es que, suponemos que para mejorar la fluidez, el entorno admite acciones antes de acabar el salto. Esto genera que, en algunas situaciones, se tome una decisión sin tener la información correcta porque estaremos estudiando el estado antes de completar la acción anterior. Esto puede afectar al rendimiento del sistema y para evitar estos contratiempos, forzaremos a que el agente sólo tome acciones cuando esté parado encima de una base.

Para ello, lo que haremos será guardar la posición del agente en la última iteración para luego compararla con la actual. Si no está parado, por defecto aplicaremos la acción 0, que no hace nada. Si el agente permanece igual durante dos frames, entonces supondremos que está parado en una base y estaremos en posición de aplicar otra acción.

[20]:
```
env = gym.make("QbertDeterministic-v0")
env.seed(seed)
rawObs = env.reset()
maxSteps = 340
skipFrames = 37

currentBase = 0
firstRound = np.ones(21)

pjLastStep = np.arange(76*2).reshape(76,2)  #random init to compare␣
 ↪current PJ possition with PJ possition at last frame
count = 0

imgs, debug = [], []
for step in range(maxSteps):
    action = 0
    obs = preprocessImg(rawObs)
    if step < skipFrames or 181 not in obs:
        rawObs, reward, done, info = env.step(0)
        continue

    pj = np.argwhere(obs == 181)

    render(env)
    debug.append(("step", step, "action", action, "count", count))
    imgs.append(rawObs)
```

```
    if np.array_equal(pj, pjLastStep):  #Check if pj in the same␣
↪position as before
        count += 1
        if count == 2:  #Check if pj stopeed two consecutive steps
            count = 0
            currentBase = getBaseIndex(pj[-1])
            action, firstRound = getActionHeuristic(currentBase,␣
↪firstRound)
    else:
        count = 0

    pjLastStep = pj



    rawObs, reward, done, info = env.step(action)
    if done: break
```



Analizando las recompensas, vemos que se otorga una de +25.0 cuando pisas una plataforma que estaba desactivada. Si un enemigo desactiva una plataforma por la que ya has pasado y vuelves a pasar por ella ganas 25 puntos más. Por eso el heurístico acaba con 550 puntos, 25 de recompensa por cada base multiplicado por las 21 bases del mapa + la que desactiva el enemigo. $25 \times 22 = 550$. Cuando el enemigo alcanza al agente y lo matan, no se obtiene recompensa negativa.

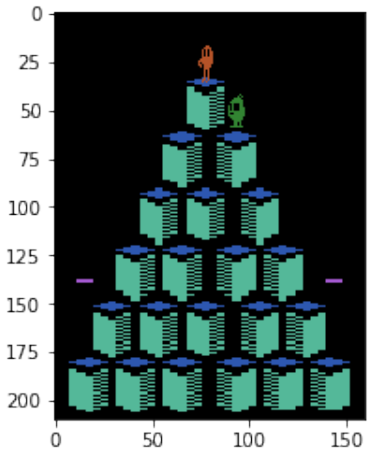
En el siguiente paso, el objetivo será modelizar a los enemigos. Para eso lo primero que haremos será fijarnos en los colores que hay en la pantalla y ver si hay enemigos (colores 50 y 146). Si está el enemigo ponemos el índice de la base (entero del 0 al 20) y si no existe ese enemigo pondremos -1.

[21]:
```
enemiesColours = [50, 146]
def getEnemiesBase(obs):
    currentScreen = np.sort(np.unique(obs))
    enemiesBase = []
```

```
    for enemyID, enemyColour in enumerate(enemiesColours):
        if enemyColour in currentScreen:
            enemy = np.argwhere(obs == enemyColour)
            enemiesBase.append(getBaseIndex(enemy[-1]))
        else:
            enemiesBase.append(-1)
    return enemiesBase
```

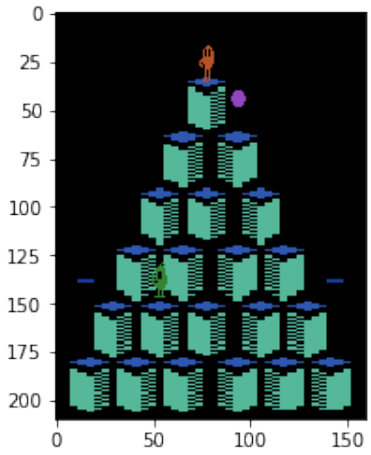
[22]:
```
env = gym.make("QbertDeterministic-v0")
env.seed(seed)
rawObs = env.reset()
maxSteps = 322
skipFrames = 37

for step in range(maxSteps):
    obs = preprocessImg(rawObs)
    if step < skipFrames or 181 not in obs:
        rawObs, reward, done, info = env.step(0)
        continue
    print(getEnemiesBase(obs))
    render(env)
    rawObs, reward, done, info = env.step(0)
    if done: break
```

```
[2, -1]
```




```
------------------------------------------------------------
```

```
[11, 2]
```



Ahora ya estamos listos para modelizar el estado. Inicialmente iba a ser un array al que ir añadiéndole valores, pero el problema es que tanto el número de enemigos como el de barras laterales es variable por lo que tendríamos que variar la arquitectura de la red neuronal en función de cada escenario. Como eso no se puede hacer hay que buscar un sistema de codificación en el que tengamos un array de x posiciones fijas. Una alternativa a la descrita sería almacenar un número que representará el estado de cada base. Por ejemplo, podría ser: 0 si la base no tiene ningún personaje encima. 1 si está el agente principal. 2 si está un enemigo. 3 si está el personaje principal y un enemigo, etc. También tendría que contemplar el color de la base así como si tiene barra lateral o no.

Pese a que parezca más intuitiva, esa codificación puede ser menos eficiente que una binaria. Simplemente nuestro estado tendrá un total de 21×4=84 posiciones, donde incluiremos:

- Para cada plataforma: un array un array de 21 posiciones, una por cada plataforma. Contendrá un 1 si esa plataforma ya la hemos pisado y un 0 si no.
- Plataforma donde se encuentra el personaje principal: un array de 21 posiciones, una por cada plataforma. Tendrá un 1 en la plataforma en la que se encuentre el personaje.
- Enemigos: Hay 2 enemigos y para cada enemigo guardaremos la plataforma donde se encuentra: un array de 21 posiciones, una por cada plataforma. Tendrá un 1 en la plataforma en la que se encuentre el enemigo.
- Barras laterales: 2 variables binarias: 1 si existe barra lateral y 0 si no. En un principio no iban a estar incluidas, pero en la versión final deberían figurar porque parecen alcanzar buenos resultados.

[23]:
```
def getState(obs, debug = False):
    state = []

    #Base colours:
    for baseColour in getBaseColours(obs):
        state.append(int(baseColour != 45))

    if debug:
```

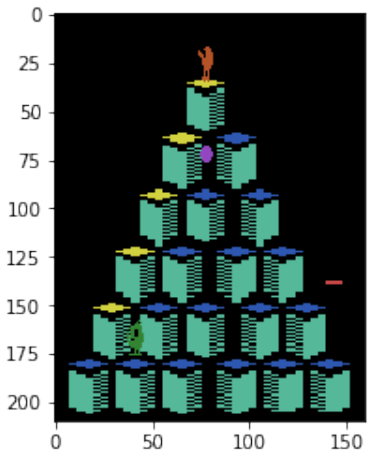
```
        print("Base colours", state)

    #For binary (K-Sparse) codification
    eye = np.eye(numBases, dtype = int)

    #PJ position:
    if 181 not in obs:
        for position in np.zeros(numBases, dtype = int):
            state.append(position)
    else:
        pjBottom = np.argwhere(obs == 181)[-1]
        pjPosition = getBaseIndex(pjBottom)
        pjPositionBinary = eye[pjPosition]
        for position in pjPositionBinary:
            state.append(position)

    if debug:
        print("PJ position ", list(pjPositionBinary))

    #Enemies:
    for enemyID, enemyPosition in enumerate(getEnemiesBase(obs)):
        if enemyPosition == -1:
            enemyPositionBinary = np.zeros(numBases, dtype = int)
        else:
            enemyPositionBinary = eye[enemyPosition]

        for position in enemyPositionBinary:
            state.append(position)

    if debug:
        print("Enemy position", enemyID, list(enemyPositionBinary))

    #Lateral bars:
    lateralBar = 1 if obs[138,12] else 0
    state.append(lateralBar)
    lateralBar = 1 if obs[138,140] else 0
    state.append(lateralBar)
    if debug:
        print("Left lateral bar", state[-2])
        print("Right lateral bar", state[-1])

    return np.array(state)
```

[24]:
```
def getActionLateralBar(currentBase, rightLateralBar):
    if rightLateralBar:  # If we want to go right
        action = 0
        if currentBase in [0, 2, 5, 9]:
```

```
            action = 3
        if currentBase == 14:
            action = 2
    else: # If we want to go left
        action = 0
        if currentBase in [0, 1, 3, 6]:
            action = 5
        if currentBase == 10:
            action = 4

    return action
```

[25]:
```
env = gym.make("QbertDeterministic-v0")
env.seed(seed)
rawObs = env.reset()
step = 0
maxSteps = 340
skipFrames = 37

currentBase = 0
firstRound = np.ones(21)

pjLastStep = np.arange(76*2).reshape(76,2)  #random init to compare␣
 ↪current PJ position with PJ position at last frame
count = 0
state = np.zeros(84)

for step in range(maxSteps):
    #Default action is stay stopped
    action = 0

    #Preprocess screen
    obs = preprocessImg(rawObs)

    #Skip initial part and if pj is not in screen
    if step < skipFrames or 181 not in obs:
        rawObs, reward, done, info = env.step(0)
        continue

    #Choose action. Only when we are stoped (not in the middle of a␣
 ↪jump)
    pj = np.argwhere(obs == 181)
    if np.array_equal(pj, pjLastStep): #Check if pj in the same␣
 ↪position as before
        count += 1
        if count == 2:  #Check if pj stopped two consecutive steps
            count = 0
```

```
            currentBase = getBaseIndex(pj[-1])

            #Process state
            state = getState(obs, debug = True)
            render(env)
            input(state)

#             action, firstRound = getActionHeuristic(currentBase,␣
 ↪firstRound)
            action = getActionLateralBar(currentBase, False)

    else:
        count = 0

    pjLastStep = pj

    rawObs, reward, done, info = env.step(action)


    if done or np.array_equal(state[0:numBases], np.ones(numBases)):␣
 ↪break
```



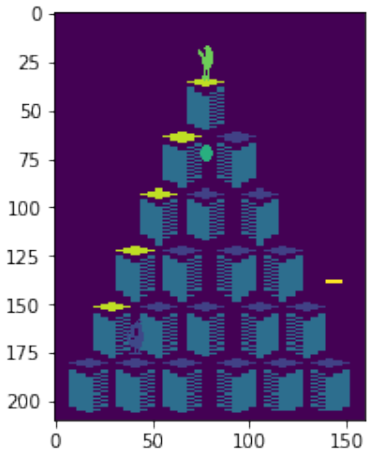
Nos damos cuenta de que las recompensas son: 25 si pisas una plataforma que no estaba coloreada, 100 si te pasas el nivel. Por defecto las muertes no están penalizadas y puede desarrollar un algoritmo menos eficiente. Por eso, vamos a ver cuándo perdemos una vida y cuando eso pase otorgaremos una recompensa de -250 si nos han quitado una vida y de -1000 si las hemos perdido todas.

Para ello, primero crearemos una función que observe las vidas y en función de ellas iremos otorgando las recompensas pertinentes.

[26]:
```
rewardPixels = [6, 35]
lifePixels = [[25, 33], [25, 44], [25, 54]]

def getLives(obs):
```

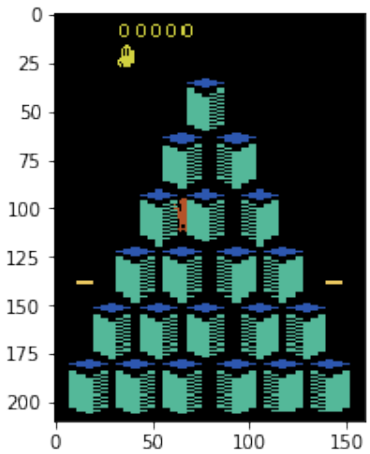
```
    lifes = 0
    for pixel in lifePixels:
        lifes += int(obs[pixel[0], pixel[1]] == 210)
    return lifes

plt.imshow(obs)
getLives(obs)
```

[26]: 0



[27]:
```
env = gym.make("QbertDeterministic-v0")
env.seed(seed)
rawObs = env.reset()
step = 0
maxSteps = 340
skipFrames = 37

currentBase = 0
firstRound = np.ones(21)

pjLastStep = np.arange(76*2).reshape(76,2)  #random init to compare␣
 ↪current PJ position with PJ position at last frame
count = 0

for step in range(maxSteps):
    #Default action is stay stopped
    action = 0

    #Preprocess screen
    obs = preprocessImg(rawObs)

    #Skip initial part and if pj is not in screen
    if step < skipFrames or 181 not in obs:
        rawObs, reward, done, info = env.step(0)
```

```
        continue

    #Choose action. Only when we are stoped (not in the middle of a␣
↪jump)
    pj = np.argwhere(obs == 181)
    if np.array_equal(pj, pjLastStep): #Check if pj in the same␣
↪position as before
        count += 1
        if count == 2:  #Check if pj stopped two consecutive steps
            count = 0

            action = 4
    else:
        count = 0

    pjLastStep = pj

    rawObs, reward, done, info = env.step(action)
    if obs[rewardPixels[0], rewardPixels[1]]==210:
        render(env)
        input(("lives: ", getLives(obs)))

    if done or np.array_equal(state[0:numBases], np.ones(numBases)):␣
↪break
```



Una vez tenemos el modelo del estado bien representado, inicializaremos el optimizador que usaremos, que será una variación del descenso del gradiente tradicional: Adam. Lo fijaremos con un learning rate de 0.01. Además declararemos la función de pérdidas que utilizaremos, que será la categorical cross entropy.

Una vez fijados estos parámetros, pasamos a crear el modelo de la ANN. Este modelo tendrá como entradas tantas neuronas como parámetros tenga nuestro estado. En este caso serán 86 contando con las barras laterales, serían 84 si no estuviesen. También tendrá esa misma cantidad de neuronas intermedias. Como función de activación para la capa

oculta, elegiremos elu. Para la capa de salida, tendremos tantas neuronas de salida como acciones posibles. En este caso, las dos primeras acciones hacen lo mismo (nada), por lo que pondremos 5 neuronas de salida. Como función de activación de esta capa utilizaremos softmax, para obtener las probabilidades de cada acción en lugar de la acción en sí.

[29]:
```
optimizer = keras.optimizers.Adam(lr=0.01)
loss_fn = keras.losses.categorical_crossentropy

numActions = env.action_space.n  #gym accepts any number from 0 to 5␣
 ↪in actios but action 1 and 0 are equals
usefulActions = numActions -1  #There is one action unused
numInputs = len(state)

model = keras.models.Sequential([
    keras.layers.Dense(numInputs, activation="elu",␣
 ↪input_shape=[numInputs]),   #elu is a derivable function, relu not
    keras.layers.Dense(usefulActions, activation='softmax')
])
```

Una vez definido el modelo, pasaremos al sistema de elección de acción. En este caso utilizaremos un ϵ-greedy con ratio de descenso 0.01, una probabilidad de explorar máxima de 0.9 y mínima de 0.1.

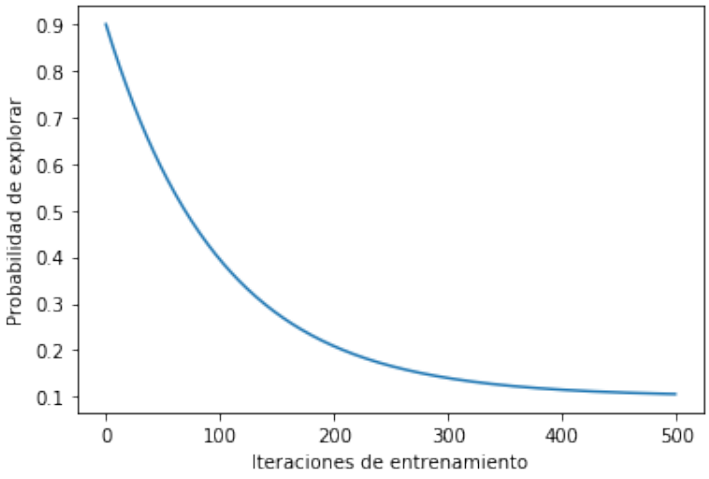
Como sólo tenemos 5 neuronas de salida, pero las acciones están numeradas del 0 al 6, supondremos que las probabilidades de salida de la red neuronal son para las acciones de la 1 a la 6 (nada, up, right, left y down). De esta forma, obviamos la acción 0, ya que hace lo mismo que la acción 1. Es por ello que al explotar, elegimos la acción `np.argmax(probs) + 1`.

[33]:
```
#Epsilon greedy: Exploration vs Exploitation
# Note that shape(probs)=(1,5) while env.anction_space.n=6. Action 0␣
 ↪and action 1 are equals so the model avoid using action 0.

expDecayRate = 0.01
maxExpRate = 0.9
minExpRate = 0.1
def epsilonGreedy(iteration, probs):
    expRate = minExpRate + (maxExpRate-minExpRate) * np.
 ↪exp(-expDecayRate * iteration)
    if np.random.random() > expRate:
        action = np.argmax(probs) + 1 #Avoid action 0 because its same␣
 ↪than 1. Useful actions=from 1 to 5
    else:
        action = np.random.randint(len(probs[0]))+1 #exploration
    return action, expRate
```

[34]:
```
i = 0
rates = []
while i<500:
```

```
    probs = [[1, 2, 3, 5, 6]]
    a, expRate = epsilonGreedy(i, probs)
    rates.append(expRate)
    i+=1
plt.ylabel("Probabilidad de explorar")
plt.xlabel("Iteraciones de entrenamiento")
plt.plot(rates)
```

[34]: `[<matplotlib.lines.Line2D at 0x7ff056c116d0>]`



Cuando queramos elegir la acción que tomaremos en cada estado, primero obtendremos las probabilidades de la red neuronal. Esas probabilidades junto con la iteración del algoritmo en el que estamos se lo pasaremos a la función ϵ-greedy. Para este algoritmo, es necesario obtener los gradientes de los pesos del modelo cada vez que ejecutemos una acción. Se declarará una variable `y_target` que contendrá la salida deseada de la red neuronal. Por ejemplo, si la red neuronal quiere que vayamos hacia abajo (acción 5), lo ideal sería que nos diese unas probabilidades así [0 0 0 0 1]. Sin embargo, la salida de la red neuronal será, por ejemplo [0.15, 0.20, 0.5, 0.10, 0.50]. Los gradientes los calcularemos respecto a la función de pérdidas que se definió junto con el modelo de la red neuronal.

[39]:
```
def getAction(state, model, loss_fn, iteration):
    with tf.GradientTape() as tape:
        probs = model(state[np.newaxis])  #shape(state)=(84,) -->␣
↪state[np.newaxis] --> shape(state)=(1,84) => transpose
        action, expRate = epsilonGreedy(iteration, probs)
        y_target = np.eye(usefulActions)[action-1]   #If action=3 we␣
↪want probs=[0 0 1 0 0]. UsefulActions = from 1 to 5
        loss = tf.reduce_mean(loss_fn(y_target, probs))  #gets mean.␣
↪If inputs==ints returns = np.floor(mean)
    grads = tape.gradient(loss, model.trainable_variables)
    return action, grads
```

Y ya casi estamos al final del algoritmo. Generaremos una función que vaya gestionando las partidas de las iteraciones de entrenamiento. Hemos implementado dos bucles,

uno se dedicará a recorrer los juegos de cada iteración de entrenamiento y otro a recorrer los pasos de cada juego. Se jugarán tantas partidas como marque la variable `maxGamesPerTrainingIteration`. También, se limita el máximo de pasos por cada juego por si la partida llegase a estancarse de alguna forma. Para cada juego, por defecto, se inicializa la acción 0 (no hacer nada) para que si el agente está parado continúe el salto. También, se pasan los primeros frames y cuando no esté el agente en la pantalla, para hacer el algoritmo más eficiente.

El siguiente paso consiste en comprobar si el agente está en un salto, observando si la posición del personaje permanece igual durante dos frames seguidos. Siempre que el agente esté parado, se analizará el estado y se llamará a la función `getAction()`, que nos dará la acción y sus gradientes. Luego, sumaremos las recompensas obtenidas desde la última vez que el agente estuvo parado hasta ahora, almacenadas en `allActionRewards`. El resultado de esta suma será la recompensa de cada acción, y se guardará en `actionReward`. A continuación, añadiremos la recompensa de cada acción a la lista de recompensas de cada partida (`gameRewards`).

Haya estado o no parado el agente, se aplicará la acción que corresponda, es decir, no hacer nada si el agente estaba en movimiento, o la acción que determine la red neuronal si estaba parado. Después, se procesará la recompensa para otorgar -250 si te han matado. La recompensa obtenida, se añadirá a la variable `allActionRewards`. De esta forma, la siguiente vez que el personaje esté parado, se podrán sumar todas las recompensas obtenidas a lo largo del salto. Para comprobar si ha terminado un episodio chequearemos la variable done. En caso de que haya finalizado, estudiaremos si el agente ha ganado la partida (todas las bases activadas) para añadir una recompensa de +2000 a las recompensas del juego.

Finalmente, una vez concluida una partida, almacenaremos los gradientes de cada partida (`gameGrads`) en la variable `iterGrads`, que contendrá los gradientes de todas las partidas. De la misma forma, guardaremos también las recompensas de cada partida (`gameRewards`) en la variable `iterRewards`. Sobre esta última parte, hay que añadir que las recompensas van con un paso de retraso, por lo que la primera no la tendremos en cuenta. Esto es debido a que justo cuando se inicia la partida y el agente está parado en la primer base, se guarda una recompensa que es 0.

[40]:
```
def playMultipleGames(env, maxGamesPerTrainingIteration,␣
↪maxStepsPerGame, skipFrames, iteration, model, loss_fn):
    iterRewards, iterGrads = [], []

    for game in range(maxGamesPerTrainingIteration):
        pjLastStep = np.arange(76*2).reshape(76,2)  #random init to␣
↪compare current PJ position with PJ position at last frame
        countFramesStopped = 0
        state = np.zeros(84)
        lastStoppedStep = 0
        lives = 3

        allActionRewards, gameRewards, gameGrads = [], [], []
        rawObs = env.reset()
```

```
        for step in range(maxStepsPerGame):
            action = 0  #Default action is stay stopped
            obs = preprocessImg(rawObs)  #Preprocess screen

            #Skip initial part and if pj is not in screen
            if step < skipFrames or 181 not in obs:
                rawObs, reward, done, info = env.step(0)
                continue

            #Choose action. Only when we are stoped (never in the␣
 ↪middle of a jump)
            pj = np.argwhere(obs == 181)
            if np.array_equal(pj, pjLastStep): #Check if pj is in the␣
 ↪same position as before
                countFramesStopped += 1
                if countFramesStopped == 2:  #Check if pj stay stopped␣
 ↪for two consecutive steps
                    countFramesStopped = 0
                    #Process state and get action
                    state = getState(obs)

                    #Check if agent explored bottom
#                       if np.any(state[15:21]):
#                           print(iteration, "Ha pasado por debajo")
#                           imgsBottom.append(rawObs)

                    action, grads = getAction(state, model, loss_fn,␣
 ↪iteration)
                    gameGrads.append(grads)

                    actionReward = np.sum(allActionRewards)
                    gameRewards.append(actionReward)
                    allActionRewards = []

            else:
                countFramesStopped = 0

            rawObs, reward, done, info = env.step(action)
            if obs[rewardPixels[0], rewardPixels[1]]==210:  #If life␣
 ↪switch store new value
                if lives > getLives(obs):
                    reward = -250.0
                    lives = getLives(obs)

            allActionRewards.append(reward)

            pjLastStep = pj
```

```
            #If PJ is dead or all bases Ok -> reset
            if done:
                if np.array_equal(state[0:numBases], np.
↪ones(numBases)):
                    gameRewards.append(2000)
                break
        iterRewards.append(gameRewards[1:])   #Ignore first reward
        iterGrads.append(gameGrads)

    return iterRewards, iterGrads
```

Una vez jugadas varias partidas, lo que haremos será descontar las recompensas de esa iteración de entrenamiento. Para ello, establecemos estas dos funciones que primero aplicarán una tasa de descuento definida por discountRate y luego normalizarán las recompensas según su media y su desviación estándar.

[41]:
```
def discountRewards(rewards, discountRate, debug=False):
    discountedRewards = np.array(rewards)   #ie rewards=[10,0,-50],␣
↪discountRate=0.8
    for index in range(len(rewards) - 2, -1, -1):    #step in [1, 0]
        discountedRewards[index] += discountedRewards[index + 1] *␣
↪discountRate #rs[1]=0+(-50)*0.8=-40; rs[0]=10+(-40)*0.8=-22
        if debug: print("index", index, "discounted",␣
↪discountedRewards)
    return discountedRewards

def discountAndNormalizeRewards(iterRewards, discountRate):
    allDiscountedRewards = [discountRewards(rewards, discountRate) for␣
↪rewards in iterRewards]
    flatRewards = np.concatenate(allDiscountedRewards)
    rewardMean = flatRewards.mean()
    rewardStd = flatRewards.std()
    return [(discountedRewards - rewardMean) / rewardStd for␣
↪discountedRewards in allDiscountedRewards]
```

En vista de que tenemos todas las piezas bien definidas, ya solo queda montar el puzle. Antes de pasar al bucle principal, hay que inicializar ciertos parámetros, como el factor de descuento a 0.95 o las partidas que se jugarán por cada iteración de entrenamiento a 500. Hay que llevar cuidado con este parámetro porque si se pone demasiado alto, el algoritmo ocupa una cantidad desmesurada de memoria. También, como comentamos en Frozen Lake, hemos incluido un fichero que guardará los pesos de la red, la arquitectura y un fichero de recompensas que nos permitirán continuar por donde íbamos.

Por último el bucle principal, que recogerá las experiencias de las 500 partidas en forma de gradiente y recompensas. Una vez obtenidos estos datos, descontaremos las recompensas y las normalizaremos utilizando las dos funciones del chunk anterior. Estas recompensas, se multiplicarán a los gradientes, para que se aplique el signo y la proporción de las

recompensas a los gradientes. Finalmente, se aplicará el optimizador Adam, declarado anteriormente, para optimizar los gradientes. De esta forma, si unos pesos han tenido una recompensa positiva, al estar multiplicando al gradiente, se seguirá la dirección que marquen estos gradientes. Sin embargo, ante una recompensa muy negativa, se tomarán los gradientes opuestos. En último lugar, simplemente se guardará el progreso del modelo cada 10 iteraciones de entrenamiento.

[ ]:
```
env = gym.make("QbertDeterministic-v0")
env.seed(seed)
rawObs = env.reset()
maxStepsPerGame = 10000
skipFrames = 37
iteration = 0

maxGamesPerTrainingIteration = 500  #episodes per training update
discountRate = 0.95
meanRewards = []

fileName = 'saver_8.9' #Save trained model in this file

if os.path.exists(fileName+".json"):# load json and create model
    json_file = open(fileName+".json", 'r')
    loaded_model_json = json_file.read()
    json_file.close()
    model = keras.models.model_from_json(loaded_model_json)
    model.load_weights(fileName+".h5")  # load weights into new model

endGreedyCounter = 0
while True: #Puesto el 23-4
    iterRewards, iterGrads = playMultipleGames(env,␣
↪maxGamesPerTrainingIteration, maxStepsPerGame,
                                               skipFrames, iteration,␣
↪model, loss_fn)

    totalRewards = sum(map(sum, iterRewards))      #rewards this␣
↪training iteration
    currentMeanReward = totalRewards / maxGamesPerTrainingIteration
    meanRewards.append(currentMeanReward)
    allFinalRewards = discountAndNormalizeRewards(iterRewards,␣
↪discountRate)
    allMeanGrads = []
    for varIndex in range(len(model.trainable_variables)):
        #Each gradient of each step of each episode multiplied by the␣
↪mean reward of that iteration (10 episodes)
        meanGrads = tf.reduce_mean(
            [finalReward * iterGrads[episodeIndex][step][varIndex]
             for episodeIndex, finalRewards in␣
↪enumerate(allFinalRewards)
```

```
                for step, finalReward in enumerate(finalRewards)],␣
↪axis=0)
        allMeanGrads.append(meanGrads)


    optimizer.apply_gradients(zip(allMeanGrads, model.
↪trainable_variables))


    checkString = "Iteration: {}, mean rewards: {:.1f}".
↪format(iteration, currentMeanReward)
    print(checkString, end="\n")

    with open(fileName+".rw", "a+") as reward_file:  #save rewards
        reward_file.write(str(currentMeanReward)+"\n")


    if iteration % 10:  #Save trained model
        model_json = model.to_json()   #serialize model to JSON
        with open(fileName+".json", "w") as json_file:
            json_file.write(model_json)
        model.save_weights(fileName+".h5")   #serialize weights to HDF5


    iteration += 1



env.close()
```

```
Iteration: 0, mean rewards: -523.9
Iteration: 1, mean rewards: -535.0
Iteration: 2, mean rewards: -525.4
Iteration: 3, mean rewards: -519.1
Iteration: 4, mean rewards: -535.2
Iteration: 5, mean rewards: -517.1
Iteration: 6, mean rewards: -534.0
Iteration: 7, mean rewards: -531.5
Iteration: 8, mean rewards: -527.8
Iteration: 9, mean rewards: -520.2
```

CAPÍTULO 5

Análisis de resultados

En este apartado analizaremos los resultados obtenidos con ambos algoritmos. Primero expondremos los resultados mediante el algoritmo de Q-Learning aproximado utilizado para resolver el entorno de Frozen Lake. En segundo lugar, comentaremos los resultados del algoritmo policy-gradient que hemos implementado para resolver QBert. Sobre Frozen Lake no hemos recogido demasiados datos, puesto que inicialmente el objetivo del proyecto se limitaba a un juego de Atari. Sin embargo, de QBert si que vamos a estar en posición de aportar estadísticas y vídeos sobre lo aprendido por nuestro sistema.

5.1. Resultados de Frozen Lake

Como viene siendo habitual en este proyecto, antes de comentar los resultados debemos explicar el parámetro utilizado para medir las prestaciones de nuestro algoritmo. Para medir el rendimiento, hemos definido la variable `winRate` como las victorias que obtiene el algoritmo dividido entre el número de partidas que juega el algoritmo:

$$\text{winRate} = \frac{\text{iterWins}}{\text{iterGames}} \tag{5.1}$$

Como se ha comentado en los capítulos anteriores, este problema lo hemos abordado utilizando dos aproximaciones diferentes. En primer lugar, comenzamos utilizando regresores lineales, y aunque no eran malos resultados, se podían mejorar. Por ello, decidimos utilizar árboles de decisión en lugar de regresores lineales, y este cambio produjo mejoras significativas. En la versión final del algoritmo, utilizando regresores lineales se ha obtenido un `winRate` máximo de 0.22, lo que implica que el algoritmo gana una de cada cinco partidas. Sin embargo, mediante el uso de árboles de decisión, este ratio aumenta hasta un 0.33, lo que nos viene a decir que el algoritmo gana una de cada 3 partidas. Este resultado es bastante más aceptable, aunque dista de los mejores resultados obtenidos en el mundo académico. También es cierto que, al ser este algoritmo un paso intermedio de nuestro proyecto, no profundizamos en su entrenamiento. Estos resultados se han conseguido en 245 épocas de entrenamiento para regresores lineales y en 348 para árboles de decisión. Es posible que dejándolo más tiempo los resultados mejoren. Es importante destacar que nos percatamos *a posteriori* de que había un error en el código que generaba

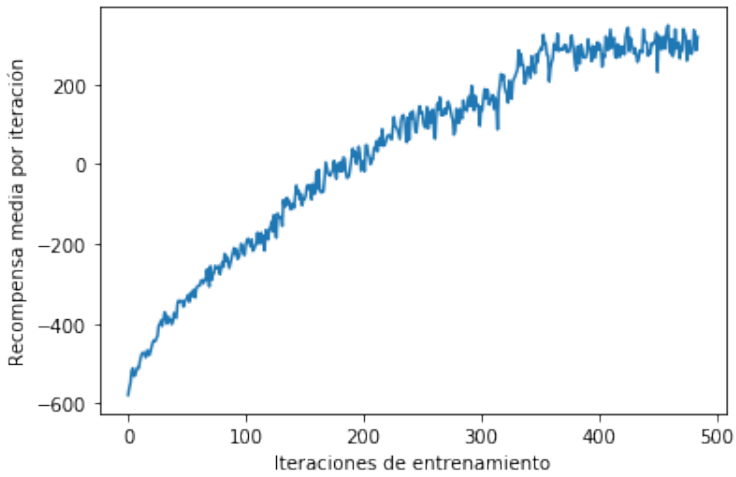


Figura 5.1: Evolución de la recompensa frente a las iteraciones de entrenamiento

mapas aleatorios. Esto provocaba que hubiese mapas que no pueden ganarse puesto que no había ruta posible hacia la meta. Por ello, los resultados no son demasiado justos. No obstante, el objetivo de este proyecto es demostrar que un algoritmo puede aprender y eso queda patente contemplando la evolución del algoritmo. Por ejemplo, en la versión sin mapas aleatorios parte de un `winRate` de 0.069 y, conforme avanza el algoritmo, se consigue un `winRate` de 0.354.

5.2. Resultados de QBert

Para Qbert, el progreso de aprendizaje del algoritmo se aprecia bastante mejor. Observemos la gráfica de recompensas de la primera versión del algoritmo, donde no se contemplan las barras laterales.

Como se puede apreciar en la figura 5.1, vemos que es claramente ascendente hasta que llega a su nivel máximo. Si nos fijamos en su nivel inicial, el algoritmo comienza con una recompensa de -578. Trascurrido el tiempo de entrenamiento necesario, se alcanzan recompensas de +320. Esto denota una progresión evidente y se puede observar en cómo juega el agente las partidas. Hemos obtenido vídeos en diferentes puntos del entrenamiento y, como se muestra en estas figuras, el agente cada vez va completando más bases. Vamos a analizar extractos de esos vídeos.

Cuando examinamos la figura 5.2, podemos ver los primeros pasos del agente, cuando sólo llevaba 82 iteraciones de entrenamiento. Lo primero que hace es completar tres casillas (A), pero pronto aprende que puede matar al primer enemigo (B). Esto, que ni nosotros sabíamos que se podía hacer, le otorga una recompensa de +300. Tras esto, el agente no sabe muy bien qué hacer y simplemente espera en una base hasta que lo matan.

Es a las 164 iteraciones, en la figura 5.3, cuando el algoritmo aprende la interacción con las barras laterales. Al principio (A), el algoritmo sigue el mismo procedimiento: bajar dos posiciones, volver para matar al enemigo y quedarse un tiempo parado en el penúltimo nivel. De hecho, ahí incluso lo matan un par de veces. Sin embargo, luego decide volver para recorrer el lateral derecho de la pirámide. Una vez que el enemigo está cerca, salta

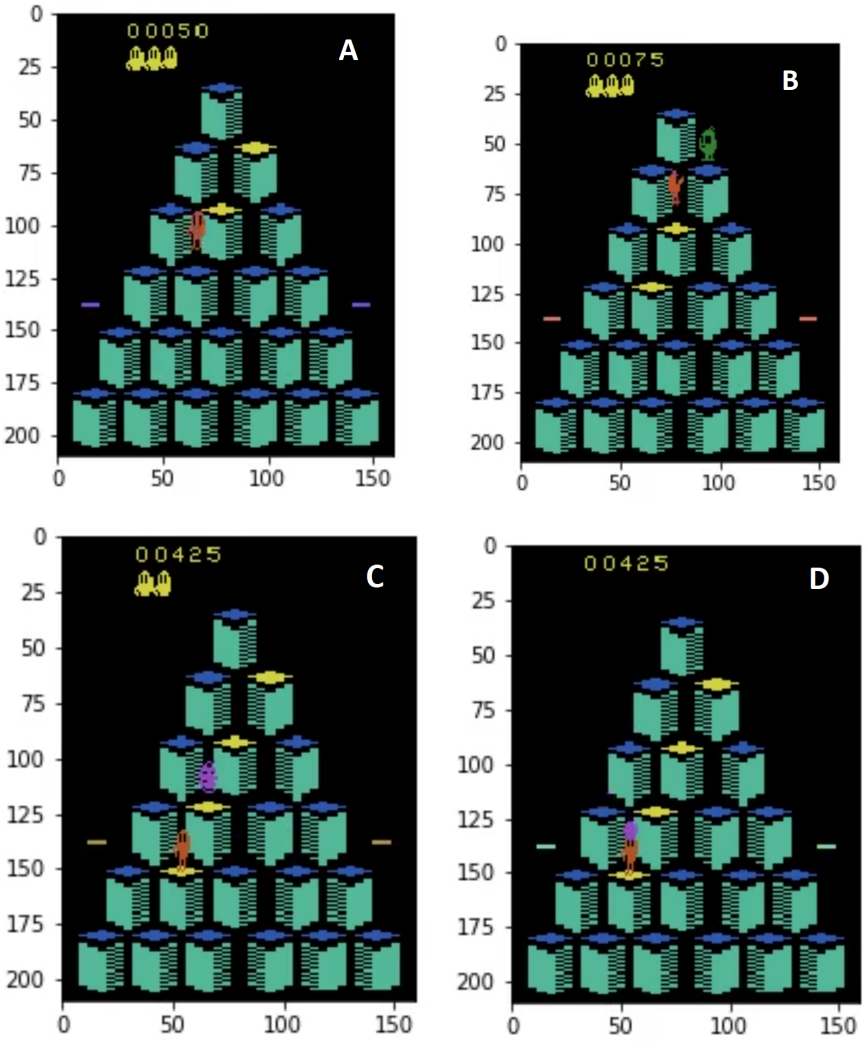


Figura 5.2: Capturas con 82 iteraciones de entrenamiento: (A) Completando sus primeras bases, (B) Eliminando a un enemigo, (C) Se queda parado, (D) Muerte del agente

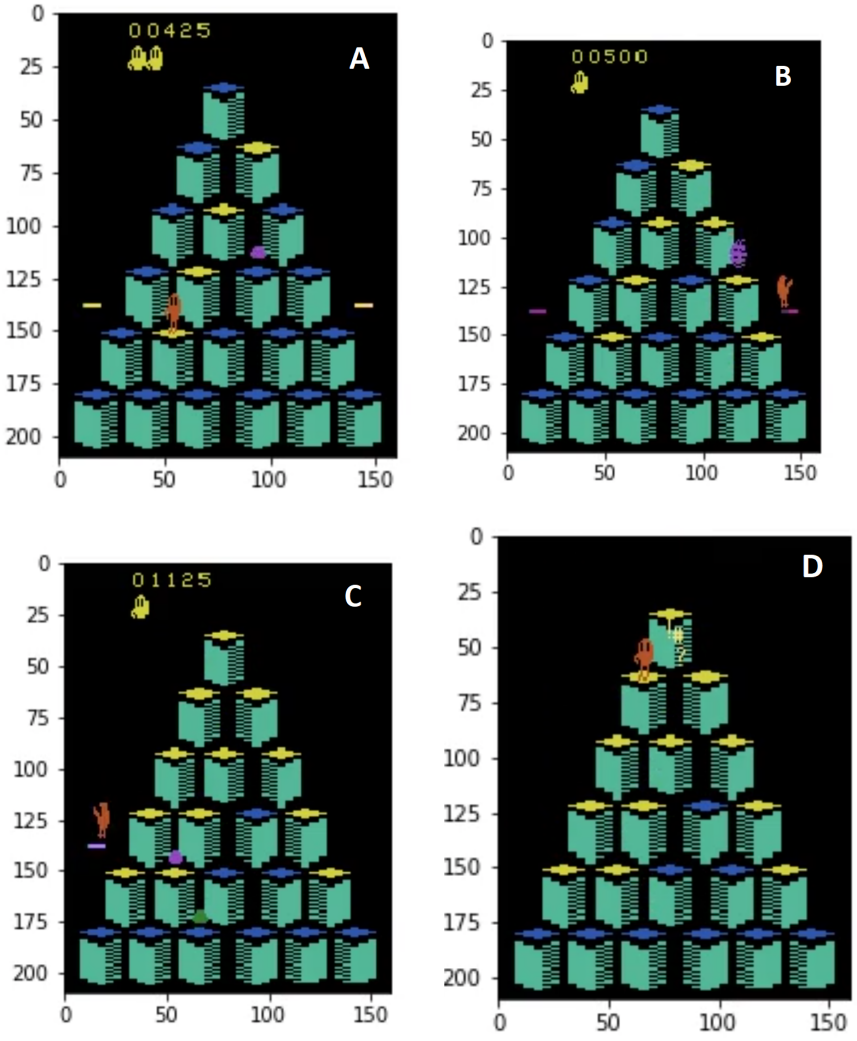


Figura 5.3: Capturas con 164 iteraciones de entrenamiento: (A) Mismo comienzo que en la figura 5.2, (B) Saltando a la barra lateral de la derecha, (C) Saltando a la barra lateral izquierda, (D) Muerte del agente

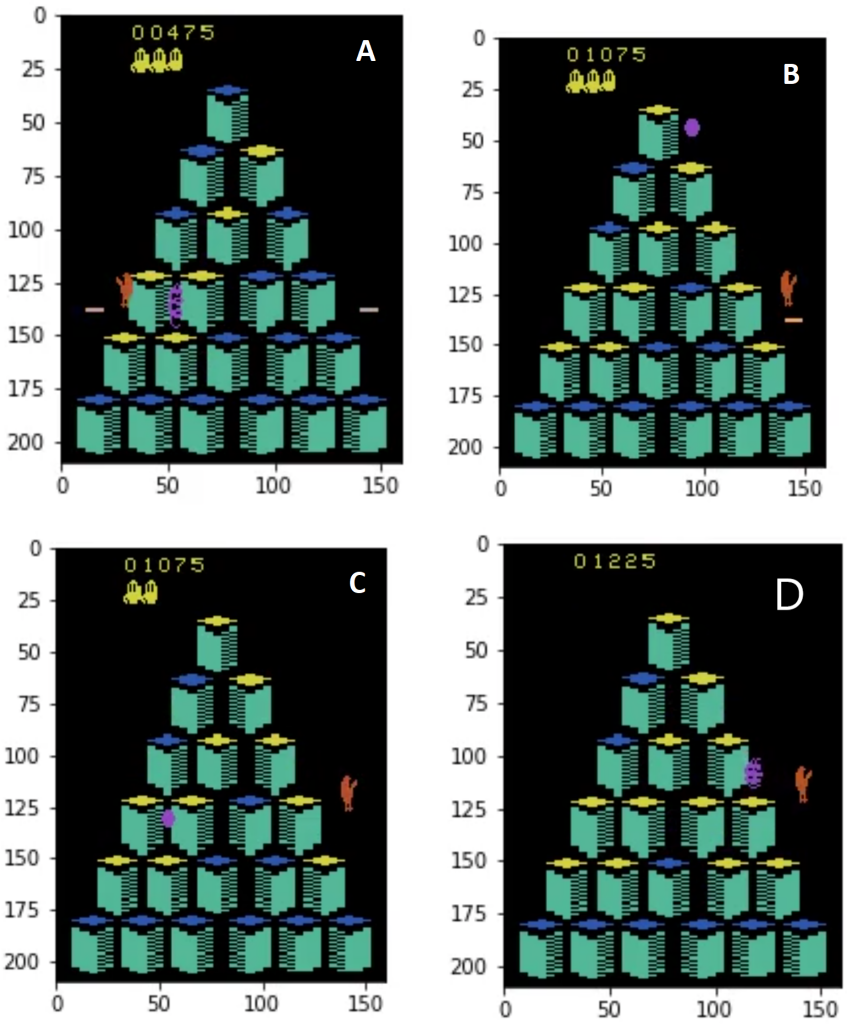


Figura 5.4: Capturas con 164 iteraciones de entrenamiento: (A) Mismo comienzo que en la figura 5.2, (B) Saltando a la barra lateral de la derecha, (C) Saltando a la barra lateral izquierda, (D) Muerte del agente

hacia la barra lateral (B). Estas barras, sólo son accesibles desde las bases a los extremos de la penúltima fila y al saltar en ellas el agente es transportado a la primera base. Lo que resulta sorprendente, es que el agente provoca que el enemigo que lo está persiguiendo se caiga, lo que le otorga una recompensa de +500. Una vez aparece en la punta de la pirámide, el algoritmo decide recorrer el lateral izquierdo para usar la otra barra lateral (C). Y finalmente, el agente termina muriendo por otro enemigo tras reaparecer de la segunda barra lateral (D).

Y para concluir, resta por comentar el último vídeo de las partidas de test de QBert, que se aprecia en la figura 5.4. En él el principio comienza relativamente parecido a los dos casos anteriores. En primer lugar, sigue los mismos pasos de la figura 5.2, para matar el primer enemigo. Luego, va hacia la barra lateral de la izquierda y hace que el segundo enemigo se caiga (A). A continuación, avanza por la barra lateral de la derecha, y la utiliza aunque no mate a ningún enemigo (B). Un detalle importante es que cuando utilizas estas barras laterales desaparecen. Entonces, las siguientes veces que las utiliza, cae al vacío y muere. Entonces, el problema viene cuando el agente se ofusca en saltar a la barra lateral, porque

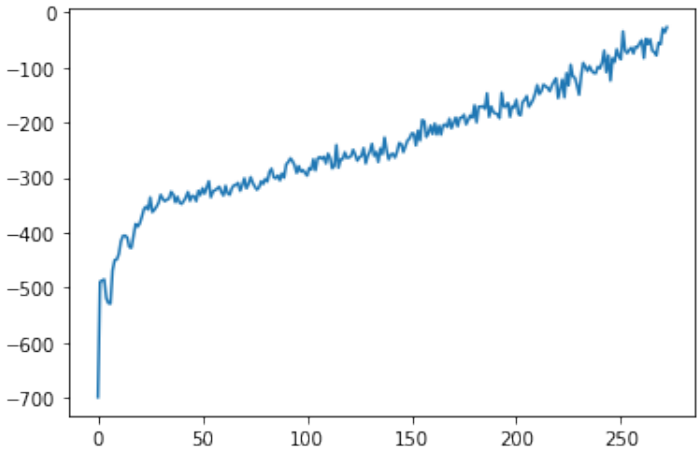


Figura 5.5: Evolución de la recompensa, habiendo modelado las barras laterales en el estado

no deja de morir. Así es como nos dimos cuenta de que había un fallo en la modelización del estado: no contemplaba las barras laterales. De hecho, si analizamos la gráfica de la figura 5.1, vemos que hay un momento en el que la recompensas no mejoran, una asíntota. Esto es debido a que el agente intenta utilizar esta estrategia porque ha aprendido que le da muy buenas recompensas. Pero claro, el sistema no contempla cuándo están operativas las barras y cuando no, por lo que creemos que sería muy complicado que el algoritmo termine aprendiendo cuándo utilizarlas y cuándo o no. Es por este motivo, por el que decidimos cambiar la modelación del estado. Como muestra la figura 5.5, este cambio ha tenido una evolución bastante buena y creemos que podría llegar incluso a tener mejores resultados.

CAPÍTULO 6

Conclusión

Para ultimar este proyecto señalaremos las ideas y sensaciones principales que hemos obtenido. Consideramos que los objetivos están cumplidos, puesto que se ha demostrado con creces la capacidad de un algoritmo para imitar el aprendizaje humano. Se han explicado diferentes técnicas de aprendizaje por refuerzo como son Policy gradient y Q-learning aproximado. Se han puesto en práctica ambas y, sobretodo, en el entorno de QBert, los resultados demuestran de forma clara la evolución del aprendizaje. Tras este proyecto, queda patente que una máquina puede aprender, adaptándose al entorno y consiguiendo las metas establecidas.

6.1. Facilidades y dificultades

Realmente este proyecto no sólo nos ha acercado al mundo del machine learning, sino que ha aumentado nuestros conocimientos sobre herramientas básicas. Entre ellas, se pueden destacar la adquisición de nociones de Python o la capacidad de operar a través un servidor remoto, utilizando jupyter-notebook. La verdad es que, pese a ser un lenguaje con el que nunca había trabajado, ha resultado bastante intuitivo aprender Python. Gracias a librerías como sklearn o keras, trabajar con regresores y redes neuronales se hace sencillo, ya que simplifican sobremanera unas tecnologías muy potentes.

Sin embargo, la verdad es que hemos encontrado dificultades derivadas de trabajar en remoto. Inicialmente, los problemas se reducían a la compatibilidad de los paquetes necesarios para hacer funcionar OpenAI Gym. Siendo sinceros me costó entenderme con los entornos virtuales de conda y con el administrador de paquetes pip. Así mismo, hemos dado con problemas inesperados asociados a características de la vida real que a veces obviamos durante la etapa universitaria. Por ejemplo, un disco duro dejó de funcionar y nos recordó la importancia de las copias de seguridad. Otro contratiempo fue cuando advertimos que el código tenía una fuga de memoria por la que el proceso acababa sobrecargando al servidor, y tuvimos que reducir el número de partidas por éopca de entrenamiento. También, derivado de la restricción de trabajar con jupyter-notebook, tuvimos dificultades a la hora de utilizar la función que incorpora Gym para mostrar la pantalla del videojuego. Por defecto, Gym abre una ventana nueva para mostrar el videojuego. Sin embargo, en un servidor headless, donde te conectas a través de una aplicación web, el intérprete de Python no tiene opción a abrir una nueva ventana. Tuvimos que

buscar una forma de representarlo y la verdad que fue algo no trivial, en primer lugar por tener que saber de dónde venía el error y en segundo lugar solucionarlo. Pese a estos problemas, al final operar en un servidor de buenas prestaciones termina siendo rentable por la capcidad de computación que aporta, que se traduce en una reducción considerable del tiempo de entrenamiento.

Centrándonos en los entornos propuestos, dentro de Frozen Lake creemos que se podrían mejorar varias cosas de nuestra solución. En primer lugar, sería importante mejorar el funcionamiento de los mapas aleatorios para que tuviesen siempre una alternativa a la victoria. También, en QBert descubrimos que poner recompensas negativas a cada paso no parece ayudar a mejorar la recompensa. Quizá, sería conveniente poner esa recompensa a partir de que la red neuronal estuviese ya entrenada, para que aprendiese gradualmente. Respecto a QBert, casi al final del proyecto nos percatamos de que el agente intentaba utilizar las barras laterales incluso cuando no estaban activas. Por ello, añadimos al estado variables binarias que representan si están las barras laterales o no. Este cambio, como hemos anticipado anteriormente, suponía cambiar la estructura de la red neuronal, por lo que había que volverla a entrenar. Lamentablemente, no se ha podido dejar entrenar demasiado tiempo, por lo que sería interesante ver a dónde nos lleva esta modelización del estado.

6.2. Líneas de ampliación

Tras este proyecto, hay una gran variedad de opciones abiertas para continuar explorando. Una de ellas, podría consistir en desarrollar otros algoritmos para resolver QBert y comparar sus resultados con los del algoritmo que aquí presentamos. Por ejemplo, no resultaría demasiado complicado y sería interesante implementar en QBert el algoritmo de Q-learning aproximado que utilizamos para Frozen Lake. De manera análoga, se podría implementar Policy-gradient para Frozen Lake y comparar los resultados de ambos algoritmos para diferentes plataformas de juego. Otra línea posible sería modificar el algoritmo de QBert para que funcionase analizando la pantalla del entorno. En primer lugar, se debería preprocesar la pantalla para facilitar la extracción de características del juego. En segundo lugar, se tendría que adaptar la red neuronal para utilizar como entradas cada píxel de la pantalla, además de añadir capas convolucionales que procesasen imágenes. También, con el fin de estudiar el rendimiento del algoritmo utilizado, se podría llevar la implementación a diferentes entornos de Gym, cambiando sólo la representación del estado. Además, como hemos comentado en la sección en la que analizamos el código de QBert, únicamente hemos entrenado al agente para que juegue la primera pantalla. Sería muy interesante adaptar el código de la modelación del estado para que pudiese jugar a todo el videojuego para ver qué niveles alcanza.

6.3. Reflexión final

Observando los resultados, parece evidente que estas tecnologías tienen un enorme potencial y que van a marcar el día a día de las generaciones futuras. Sin embargo, este campo todavía está prácticamente virgen y tiene muchísimas líneas a tratar. Para fi-

nalizar, parece bueno replantear la primera pregunta, ¿qué significa que las máquinas aprendan? Si nos fijamos en las bases, al sistema se le asigna una meta y va ejecutando acciones, explorando el entorno para ver cuáles le acercan a su meta y cuáles le alejan de ella. Al final, el algoritmo se adapta al entorno y va tomando las acciones que lo guían hacia el objetivo impuesto. La última pregunta que nos gustaría plantear es ¿Qué metas querremos los humanos que resuelvan estos agentes inteligentes?

Bibliografía

[1] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *CoRR*, abs/1606.01540, 2016.

[2] Richard S Sutton, Andrew G Barto, et al. *Introduction to reinforcement learning*, volume 135. MIT press Cambridge, 1998.

[3] Marc G. Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. The arcade learning environment: An evaluation platform for general agents. *CoRR*, abs/1207.4708, 2012.

[4] S. Tseng, Z. Liu, Y. Chou, and C. Huang. Radio resource scheduling for 5g nr via deep deterministic policy gradient. In *2019 IEEE International Conference on Communications Workshops (ICC Workshops)*, pages 1–6, 2019.

[5] Ioan Sorin Comsa, Antonio De-Domenico, and Dimitri Ktenas. Qos-driven scheduling in 5g radio access networks - a reinforcement learning approach. pages 1–7, 12 2017.

[6] Aurélien Géron. *Hands-on machine learning with Scikit-Learn and TensorFlow : concepts, tools, and techniques to build intelligent systems*. O'Reilly Media, Sebastopol, CA, 2017.