UNIVERSIDAD POLITÉCNICA DE CARTAGENA (UPCT)

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA DE TELECOMUNICACIONES (ETSIT)

Máster Universitario en Ingeniería Telemática

EVALUACIÓN DE ALGORITMOS DE APRENDIZAJE POR REFUERZO EN UN ENTORNO MULTI-AGENTE

Autor
Andrés Zapata García

Director
Juan José Alcaraz Espín

Cartagena
Septiembre 2022

Ficha del proyecto

Tipo de proyecto	Específico
Curso académico	2021-2022
Autor del proyecto	Andrés Zapata García
Director del proyecto	Juan José Alcaraz Espín
Departamento	Tecnologías de la Información y las Comunicaciones
Área de conocimiento	Ingeniería Telemática
Título en español	Evaluación de algoritmos de aprendizaje por refuerzo en un entorno multi-agente
Título en inglés	Evaluation of reinforcement learning algorithms in a multi-agent environment
Objetivos	1. Manejar las librerías de aprendizaje por refuerzo y entorno multi-agente. 2. Entrenar algoritmos de aprendizaje por refuerzo profundo (deep reinforcement learning) en el entorno multi-agente. 3. Implementar y entrenar un agente con aproximación lineal de función valor basada funciones base radiales. 4. Enfrentar entre si los agentes entrenados con distintos algoritmos.
Fases	1. Instalar Open AI gym, instalar entorno e instalar stable-baselines3. 2. Familiarizarse con el interfaz de Open AI Gym. 3. Leer documentación de stable-baselines3 y experimentar con sus funciones. 4. Familiarizarse con el entorno slimevolley. 5. Replicar uno de los experimentos de la documentación: entrenar PPO. 6. Entrenar otros algoritmos (validos para entornos de controles discretos) en slimevolley. 7. Enfrentarlos unos contra otros. 8. Implementar un algoritmo RL que use VFA con funciones base radiales (RBF). 9. Entrenar y evaluar un agente con el algoritmo desarrollado

Resumen

Una de las claves del aprendizaje por refuerzo es su capacidad para trabajar en escenarios donde el objetivo final depende de la toma de múltiples decisiones a lo largo del tiempo en un entorno concreto. El entorno que utilizamos en nuestro proyecto, SlimeVolleyGym, está diseñado para poder probar y evaluar distintos tipos de algoritmos RL y permite trabajar con un único agente que se enfrenta con un modelo de referencia, o con dos que compiten entre sí.

Nuestro trabajo se centra en exponer los fundamentos teóricos sobre los que basan distintos tipos de algoritmos y evaluarlos en el entorno SlimeVolleyGym. Tras detallar la metodología seguida para cada uno de nuestros experimentos, buscamos reflejar los resultados obtenidos de forma clara e intuitiva.

Los algoritmos evaluados cuentan con distintos niveles de complejidad. Principalmente, los clasificamos según la aproximación de la función valor y/o política que se utiliza. Aquellos que realizan una aproximación lineal son los que denominamos *clásicos*, y su implementación se detalla en el proyecto. Por otra parte, los que hacen uso de una aproximación no lineal los consideramos *avanzados*, y utilizaremos la implementación dada por la librería *stable-baselines3*. Además, para este último tipo de algoritmos, emplearemos la técnica denominada *self-play*, que se basa en la idea de aprender enfrentando el agente contra una versión anterior de sí mismo.

Abstract

One of the keys of reinforcement learning is its ability to work in scenarios where the ultimate goal depends on making multiple decisions over time in a given environment. The environment that we use in our project, SlimeVolleyGym, is designed to be able to test and evaluate different types of RL algorithms and allows us to work with a single agent that faces a reference model, or with two that compete against each other.

Our work focuses on exposing the theoretical foundations on which different types of algorithms are based and evaluate them in the SlimeVolleyGym environment. After detailing the methodology followed for each of our experiments, we seek to reflect the results obtained in a clear and intuitive way.

The evaluated algorithms have different levels of complexity. Mainly, we classify them according to the value function or policy approximation that is used. Those that make a linear approximation are what we name *classic*, and their implementation is detailed in the project. On the other hand, those that make use of a non-linear approximation are considered *advanced*, and we will make use of the implementation given by the *stable-baselines3* library. Furthermore, for this last type of algorithms, we will use the technique called *self-play*, which is based on the idea of learning by facing the agent against a previous version of itself.

Índice general

Índice de figuras

Índice de cuadros

Capítulo 1

Introducción y objetivos

1.1. Visión general del aprendizaje por refuerzo

Hacer que una máquina pueda realizar tareas históricamente realizadas por personas ha sido, sin ninguna duda, uno de los mayores avances en los últimos 100 años, desde la fabricación de los primeros computadores. Ciertas tareas que normalmente pueden ser tediosas o directamente irrealizables para una persona, han demostrado ser indicadas para el poder de computación de una máquina. A día de hoy nadie concibe una fábrica de coches sin máquinas de ensamblaje o tener que realizar cálculos complejos sin una calculadora.

No obstante, con el resto de labores que no han sido tan mecánicas, el progreso ha sido distinto. Históricamente, hay tareas que nadie pensaría que acabarían siendo realizadas por una máquina. Si a una persona de los años 70 que acaba de comprarse un vinilo de su grupo favorito le contásemos que hoy podría estar escuchando la música que le recomienda Spotify en función de sus gustos musicales, seguramente pensaría que nos estamos riendo de él.

Pero lo cierto es que, en estos últimos años, ha existido un auge en investigar cómo se puede hacer para que una máquina *aprenda* a realizar tareas no tan triviales desde el punto de vista de una persona.

Actualmente, existen distintas maneras de poder llegar a este objetivo: que una máquina consiga de cierta forma *aprender* qué es lo que buscamos en cada caso, asemejándose a la capacidad humana de realizar determinadas operaciones propias de nuestra inteligencia. Esta es una de las ideas del *machine learning* —en español, aprendizaje máquina—. Dentro del amplio espectro que ocupa el ámbito del machine learning, el alcance de este trabajo se centra en el aprendizaje por refuerzo —*reinforcement learning*, en inglés—.

1.1.1. Definición

El aprendizaje por refuerzo define un tipo de técnicas de machine learning que permite que un agente aprenda en un entorno interactivo mediante prueba y error, retroalimentándose de sus propias acciones y experiencias [1].

En esencia, el concepto es sencillo, y se acerca mucho a lo que los seres humanos entendemos por aprender: al buscar un objetivo concreto, se recompensa con un estímulo positivo cuando la acción que tomamos es lo que buscamos —o nos acerca a llegar a nuestro objetivo—, y se penaliza con un estímulo negativo cuando la acción no es la deseada —o nos aleja del objetivo deseado.—. En este escenario, lo que se buscará será buscar maximizar el número de estímulos positivos, así como intentar reducir al máximo los estímulos negativos.

Esto no es muy diferente de la manera en la que aprendemos las personas: una persona que juegue por primera vez al baloncesto sin ayuda de nadie no sabrá cómo realizar un tiro a canasta para anotar en sus primeros intentos. Pero si prueba a hacer un lanzamiento, sí que será capaz de darse cuenta que, si el balón se queda corto, necesitará hacerlo con más fuerza, y si se escapa por la derecha, necesitará ajustarlo más a la izquierda. Cuando enceste su primera canasta, podrá darse cuenta de cómo ha sido su técnica para acertar. Esto es el fundamento del aprendizaje por refuerzo: anotar una canasta sería nuestro estímulo positivo, mientras que fallar —ya sea por quedarnos cortos o ajustar el lanzamiento demasiado a la derecha— será nuestro estímulo negativo. En el primer caso, sabemos que nuestra técnica ha sido buena, por lo que intentaremos replicar el lanzamiento, mientras que en el segundo caso tendremos que corregir las acciones que nos han llevado a ello.

1.1.2. Contexto histórico

Históricamente, el aprendizaje por refuerzo ha tenido dos orígenes principales [2] [3].

Por un lado, tenemos el aprendizaje mediante prueba y error, que comienza en la psicología del aprendizaje de los animales. Este origen pasa por las investigaciones más antiguas sobre inteligencia artificial y lleva al resurgimiento del aprendizaje por refuerzo a principios de los años 80.

Edward Thorndike describió la esencia del aprendizaje mediante prueba y error con su publicación *Law of Effect* [4], en 1911, que expone que un animal perseguirá la repetición de acciones si esto refuerza su satisfacción, y será desalentado de las acciones que le producen malestar. Además, cuanto mayor sea el nivel de placer o dolor, mayor será su intención de perseguir esta acción o de ser desalentado.

El término *reinforcement learning* fue formalmente usado en el contexto de aprendi-

zaje animal en 1927 por Pavlov, que describió este tipo de aprendizaje como el fortalecimiento de un patrón de comportamiento debido a un animal que recibe un estímulo —un refuerzo— en una relación que depende del tiempo con otro estímulo o con una respuesta.

El otro origen tiene que ver con el problema de control óptimo y solución utilizando funciones valor y programación dinámica. Por lo general, en este caso no se involucraba ningún tipo de aprendizaje.

El término *control óptimo* empezó a utilizarse a finales de los años 50 para describir el problema de diseñar un controlador que minimice una medida del comportamiento dinámico de un sistema a lo largo del tiempo. Uno de los enfoques empleados en este problema fue desarrollado a mitad de los años 50 por Richard Bellman, entre otros, extendiendo una teoría del siglo XIX de Hamilton and Jacobi. Este enfoque usa los conceptos de estado y función valor (o función valor óptima) de un sistema dinámico para definir lo que se conoce como la ecuación de Bellman.

El tipo de métodos para resolver problemas de control óptimo utilizando esta ecuación se conocieron como programación dinámica. Bellman (1957) también introdujo la versión estocástica discreta del problema de control óptimo, lo que conocemos como problemas de decisión de Markov —en inglés, Markov Decision Problem, MDP— y Ronald Howard (1960) ideó el método de policy iteration para MDPs.

1.1.3. Marco de trabajo para problemas RL

A la hora de trabajar con problemas de aprendizaje por refuerzo, necesitamos definir un marco que permita que la gran variedad de entornos que existen puedan ser probados con cualquier tipo de algoritmo, ya que cada entorno es desarrollado de manera diferente.

En la actualidad, Gym es la librería que se ha convertido en el estándar de facto para poder llegar a este objetivo [5]. Gym se define como una librería de Python open source para desarrollar y comparar algoritmos RL.

También utilizaremos stable-baselines3 [6], otra librería de Python que contiene algunos de los algoritmos RL que emplearemos en el trabajo. Concretamente, serán aquellos que definimos como *avanzados*, ya que su implementación es compleja.

En la sección 2.1 describiremos en mayor detalle ambos paquetes.

1.2. Motivación

Este proyecto surge motivado por la idea de probar distintos tipos de algoritmos de aprendizaje por refuerzo en un entorno relativamente complejo. El entorno sobre el que

trabajaremos será SlimeVolleyGym [7], el cual describiremos en la sección 3.1.1.

Además, el repositorio donde se aloja SlimeVolleyGym contiene distintos resultados de evaluación de varios algoritmos en este entorno. Concretamente, sus pruebas se basan en los algoritmos Proximal Policy Optimization (PPO), Evolution Strategy with Covariance Matrix Adaptation (CMA-ES) y Genetic Algorithms (GA). De esta lista, CMA-ES y GA forman parte de lo que se denominan algoritmos evolutivos —métodos heurísticos eficaces que se basan en la evolución darwiniana con características de robustez y flexibilidad para lograr soluciones globales en problemas de optimización complejos [8]—. Por tanto, el único algoritmo de aprendizaje por refuerzo que se utiliza es PPO. En concreto, se utiliza su implementación en la librería stable-baselines en su segunda versión.

Por tanto, el margen de pruebas que tenemos en este entorno es bastante amplio. Por un lado, podemos probar distintos tipos de algoritmos —desde más sencillos a más avanzados—. Por otro lado, podemos ver cómo es el desempeño de los algoritmos en la tercera versión de stable-baselines, en comparación con la segunda. Además, el uso de la técnica self-play, que detallaremos en siguientes capítulos, nos puede dar lugar a nuevas pruebas con algoritmos que no se habían usado antes.

1.3. Objetivos

Tras haber analizado cuál es la motivación de este trabajo, definimos los siguientes objetivos a perseguir:

- Entender y analizar las librerías Gym y stable-baselines3 para el uso de algoritmos RL, así como utilizar sus principales funcionalidades.
- Comprender y experimentar con el entorno SlimeVolleyGym.
- Probar el uso de algoritmos basados en aproximación de la función valor basada en funciones base radiales.
- Experiementar y comparar las métricas de distintos algoritmos ya implementados en stable-baselines3.
- Comprobar si el enfrentamiento contra el propio agente es una técnica válida para el entrenamiento del mismo.

1.4. Contenidos de la memoria

Además de este primer capítulo, donde hemos hecho una introducción al proyecto, nuestra memoria se divide en los siguientes capítulos:

- **Capítulo 2. Entorno de desarrollo y herramientas**: Se describirán cuales han sido los paquetes y tecnologías empleadas para realizar el proyecto, así como la organización del mismo.
- **Capítulo 3. Fundamentos de aprendizaje por refuerzo**: Se comentarán los elementos básicos dentro del aprendizaje por refuerzo, así como la taxonomía y conceptos básicos dentro de los algoritmos. También se describirá en detalle el entorno *slimevolleygym*.
- **Capítulo 4. Algoritmos clásicos**: Se describirán los fundamentos de los algoritmos que denominamos *clásicos* —que hacen uso de la aproximación lineal de la función valor—, se comentará la implementación y metodología utilizada y se presentarán los resultados obtenidos de nuestros experimentos.
- **Capítulo 5. Algoritmos avanzados**: Se expondrá la teoría relacionada con los algoritmos considerados *avanzados* —que hacen uso de redes neuronales profundas para representar la función valor y/o la política—, se mostrará el proceso que se ha seguido para el entrenamiento y evaluación de estos algoritmos y finalmente se mostrarán los resultados conseguidos.
- **Capítulo 6. Técnicas self-play**: Se explicará en qué consisten este tipo de técnicas y por qué pueden ser útiles para problemas RL. También se mostrará el desarrollo realizado, así como los resultados obtenidos tras las pruebas.
- **Capítulo 7. Conclusiones y líneas futuras**: Se terminará la memoria con un repaso a las conclusiones que hemos podido extraer del trabajo, así como los posibles puntos de mejora que hayan podido surgir.

Capítulo 2

Entorno de desarrollo y herramientas

2.1. Librerías empleadas

Como ya mencionamos en el capítulo 1, el lenguaje de programación utilizado para este proyecto ha sido Python. El motivo de esta elección es su relativa sencillez en el uso y la cantidad de paquetes de calidad destinados al machine learning que podemos encontrar. Concretamente, existe una gran variedad de librerías de algoritmos RL, además de Gym, nuestra librería principal para poder trabajar con los entornos.

El desarrollo del código principal se ha realizado empleando Jupyter Notebooks. Esta decisión se debe a la flexibilidad que nos aporta poder ejecutar el código por partes —en celdas—, sobre todo teniendo en cuenta la duración de ciertas ejecuciones, pudiendo llegar a varias horas. Esto nos permite detectar problemas de forma más cómoda y clara.

Además, para este proyecto utilizamos un entorno virtual de Anaconda, lo que nos permite gestionar los paquetes instalados de forma sencilla, así como utilizar versiones compatibles entre paquetes.

A continuación, describimos los principales paquetes que hemos utilizado para la realización de este trabajo y el uso que le hemos dado:

- **gym (0.21.0)** [5]: Gym es una librería de Python open source para desarrollar y comparar algoritmos de aprendizaje por refuerzo. Para ello proporciona un API estándar para la comunicación entre algoritmos y entornos, además de un conjunto de entornos que se ajustan a este API. En nuestro caso, la utilizaremos para poder integrar el desarrollo de nuestros algoritmos, así como los que ya están implementados, con el entorno utilizado.

- **stable-baselines3 (1.5.0)** [6]: Stable Baselines3 es un conjunto de implementaciones fiables de algoritmos RL en PyTorch. Es la siguiente versión de stable-baselines. stable-baselines es un conjunto de implementaciones de algoritmos RL basados en OpenAI Baseline [9]. El mayor cambio de la tercera versión de la librería es el uso de PyTorch en sus implementaciones. El uso de esta librería hace que la velocidad de ejecución de los algoritmos sea muy superior a la que podemos alcanzar usando las librerías estándar de Python. Sus principales características y motivos por los que elegimos esta librería son los siguientes:

  - Estructura unificada para todos los algoritmos.
  - Estilo de código PEP8.
  - Funciones y clases documentadas.
  - Código testeado con alta cobertura (coverage).
  - Código limpio.
  - Soporte a Tensorflow.
  - El rendimiento de cada algoritmo ha sido testeado y puede ser consultado online.

  Haremos uso de esta librería para experimentar con algoritmos complejos, tomando como referencia los experimentos del autor del entorno, intentando aportar novedades.

- **slimevolleygym** (0.1.0) [7]: SlimeVolleyGym es un entorno gym para probar algoritmos RL *single-agent* y *multi-agent*. Esta librería será el entorno principal en el que haremos nuestras pruebas. Aunque en la sección 3.1.1 entramos más en detalle, este entorno se basa en un juego de la década de los 2000s creado por un autor desconocido.

- **numpy (1.22.4)** [10]: Numpy es el paquete fundamental para realizar operaciones con arrays en Python. En este proyecto, se usará principalmente en la implementación de los algoritmos clásicos, donde realizar operaciones con vectores y matrices es necesario.

- **pandas (1.4.2)** [11]: Pandas es un paquete de Python que proporciona estructuras de datos rápidas y flexibles, diseñadas para trabajar con datos relacionales o etiquetados de forma fácil e intuitiva. De manera práctica, nos servirá principalmente para almacenar la información que recogemos de nuestros entrenamientos y evaluaciones, pudiendo ser almacenada en ficheros .csv para su posterior representación y análisis.

- **scikit-learn (1.1.1)** [12]: scikit-learn es un módulo de Python que se utiliza para machine learning, construido como una capa superior a SciPy. Su uso en el proyecto será limitado, ya que únicamente lo necesitaremos a la hora de la aplicación de algoritmos de *clustering* en las pruebas de los algoritmos clásicos.
- **matplotlib (3.5.2)** [13]: Matplotlib es una librería destinada para crear visualizaciones estáticas, animadas e interactivas en Python. En nuestro caso, nos servirá para visualizar estadísticas del entrenamiento y las evaluaciones de nuestros agentes.

2.2. Organización del proyecto

Para gestionar este proyecto, organizamos todo nuestro código, estadísticas, figuras y modelos en la siguiente estructura de directorios:

- */data*: contiene toda la información del muestreo y clustering para los algoritmos clásicos.
- */figures*: contiene las gráficas de aprendizajes y evaluaciones de nuestros modelos.
- */logs*: contiene la información generada tras los aprendizajes y evaluaciones de nuestros modelos, que será la que se utilizará para generar las gráficas.
- */models*: contiene los ficheros .zip resultantes de guardar los modelos entrenados en stable-baselines3.
- */src*: contiene el código principal que hace uso de los algoritmos y experimentos que se desarrollan en este trabajo. Principlamente:
  - *classic_ algorithms.ipynb*: Desarrollo de algoritmos clásicos con implementación propia, tal como se detalla en la sección 4.2.
  - *advanced_ algorithms.ipynb*: Desarrollo de algoritmos avanzados, implementados por stable-baselines3, detallado en la sección 5.2.
  - *selfplay.ipynb*: Desarrollo de algoritmos avanzados utilizando técnicas de aprendizaje self-play, descrito en la sección 6.1.

Debemos mencionar también que todo el desarrollo de este código se ha llevado a cabo utilizando un repositorio de GitHub, al que se puede acceder de manera pública mediante el siguiente enlace: `https://github.com/andreszpt/rl-slimevolley`. Los motivos de esta decisión son poder llevar un control de las versiones de los ficheros, así como ver los cambios introducidos en cada momento.

2.3. Equipo utilizado

Dado que el tiempo de cómputo es un factor clave a la hora de la ejecución de algoritmos RL, es importante mencionar el hardware utilizado en este proyecto. El dispositivo que se ha usado ha sido un ordenador portátil Lenovo Ideapad 700, que cuenta con una CPU Intel Core i7-6700HQ y 8GB de RAM, corriendo Windows 10. Aunque el dispositivo cuente con varios años de antigüedad, sus especificaciones nos permiten correr todos los algoritmos sin problema. El único punto a tener en cuenta es su limitación en términos de velocidad, ya que un procesador más actual será capaz de realizar ejecuciones en un tiempo más reducido.

En el apartado de resultados numéricos de cada uno de los capítulos de desarrollo expondremos en mayor detalle los resultados del tiempo de cómputo de cada uno de los algoritmos, haciendo una comparativa entre ellos.

Capítulo 3

Fundamentos de aprendizaje por refuerzo

3.1. Elementos en RL

Dentro del marco de aprendizaje por refuerzo, hay ciertos conceptos que es necesario comentar para comprender el funcionamiento de las técnicas empleadas.

Básicamente, existen 6 términos esenciales a la hora de trabajar con RL: agente, entorno, modelo, estado, acción y recompensa [14].

En primer lugar, tenemos el agente. El agente es la entidad que aprende a tomar decisiones actuando en un entorno. Por tanto, el entorno es la entidad que se observa y se intenta controlar. El cómo este entorno reacciona a determinadas acciones realizadas por el agente se define mediante un modelo. Este modelo puede ser conocido o no.

El agente observa el estado del entorno. A partir de dicha observación, el agente selecciona una de las acciones posibles y la aplica al entorno. La aplicación de dicha acción hace que el entorno devuelva una recompensa como señal de feedback y que cambie de un estado a otro.

La toma de acciones y la observación de estados y recompensas se realiza en en etapas temporales. Por lo tanto, en una etapa t, el agente recibirá por parte del entorno un estado S_t y una recompensa R_t. En base a estas variables, el agente tomará una acción A_t. Cuando el agente tome A_t, el entorno le devolverá una nueva recompensa R_{t+1} y un nuevo estado S_{t+1}.

Estos conceptos se ilustran en la figura 3.1.

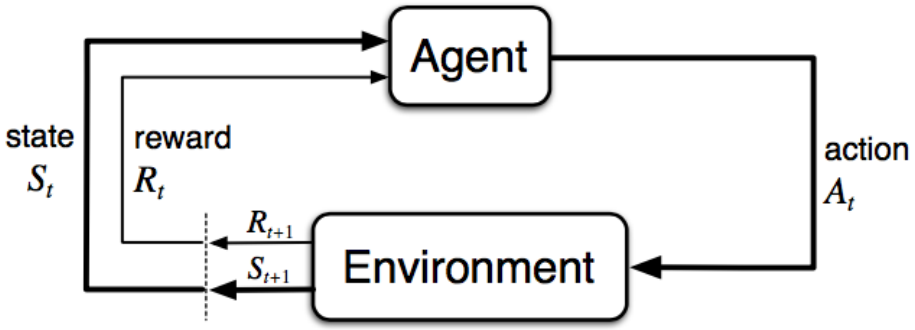

Figura 3.1: Interacción entre un agente y el entorno [2].

3.1.1. Ejemplo: SlimeVolleyGym

Tal como comentamos anteriormente, el entorno que utilizaremos será SlimeVolleyGym. En la propia descripción del entorno se explica lo siguiente:

> El juego es muy sencillo: el objetivo del agente es conseguir que la bola toque el suelo del lado del oponente, lo que hace que el oponente pierda una vida. Cada agente empieza con 5 vidas. El episodio acaba cuando el agente pierde las 5 vidas o al cabo de 3000 etapas. Un agente recibe una recompensa de +1 cuando el oponente pierde una vida o de -1 cuando es el propio agente quien pierde la vida.

En la figura 3.2 se puede ver nuestro entorno gráficamente. El agente será el personaje de la derecha, mientras que el oponente será el personaje de la izquierda.

Respecto al espacio de acciones, tenemos 6 acciones disponibles para tomar:

- *NOOP*: El agente no se mueve.
- *LEFT*: Movimiento a la izquierda.
- *UPLEFT*: Movimiento a la izquierda y salto.
- *UP*: Salto.
- *UPRIGHT*: Movimiento a la derecha y salto.
- *RIGHT*: Movimiento a la derecha.

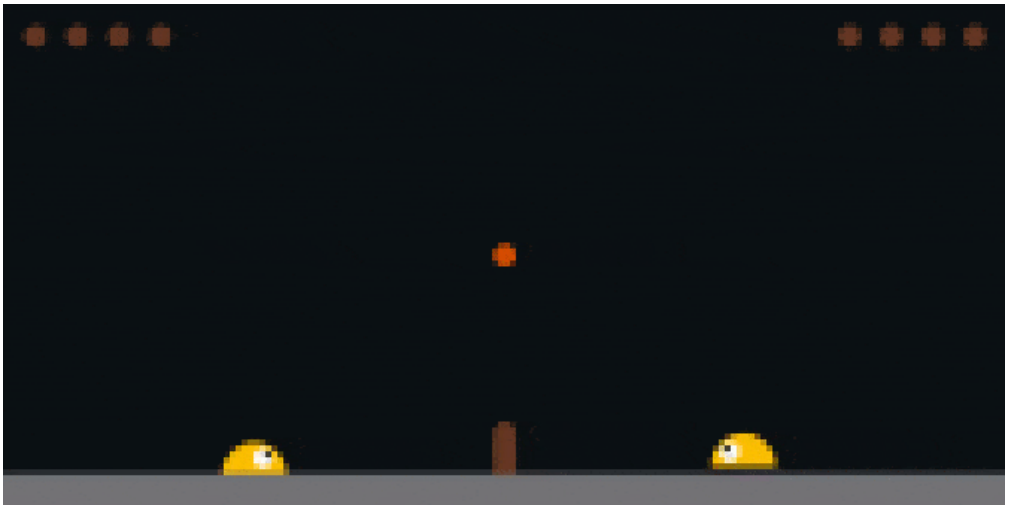

Figura 3.2: Entorno slimevolleygym de manera gráfica [7].

Una de las características de los entornos que emplean el API de Gym es su atributo *action_ space*. Este atributo especifica qué formato tienen las acciones válidas. Entendemos por formato el tipo de datos de Python que se utiliza para definir cada una de las acciones posibles [15].

Por defecto, nuestro entorno usa un espacio del tipo *MultiBinary(3)*. Esto significa que tenemos un espacio binario de tres dimensiones —array de tres dimensiones con valores binarios—, las cuales serían ir/no ir a la izquierda, ir/no ir a la derecha o ir/no ir hacia arriba. A nivel de código del entorno, las acciones posibles se definen de la siguiente manera:

```
action_table = [[0, 0, 0], # NOOP
                [1, 0, 0], # LEFT (forward)
                [1, 0, 1], # UPLEFT (forward jump)
                [0, 0, 1], # UP (jump)
                [0, 1, 1], # UPRIGHT (backward jump)
                [0, 1, 0]] # RIGHT (backward)
```

Desde el punto de vista de desarrollo, para poder aplicar nuestros algoritmos —especialmente los que implementa stable-baselines3, cuya implementación es fija—, nos interesa que el espacio de acciones sea de tipo *Discrete(6)*, siendo este un espacio que contiene un conjunto finito de números enteros. De esta manera, el tipo de datos de nuestras acciones ya no será un array tridimensional con valores binarios, sino un valor entero en el rango [0, 5].

El motivo de esta decisión es que, entre los algoritmos disponibles en stable-baselines3, existe un mayor número de algoritmos que soportan un espacio de acciones *Discrete*, en

Name	Box	Discrete	MultiDiscrete	MultiBinary	Multi Processing
ARS [1]	✔	✔	✘	✘	✔
A2C	✔	✔	✔	✔	✔
DDPG	✔	✘	✘	✘	✔
DQN	✘	✔	✘	✘	✔
HER	✔	✔	✘	✘	✘
PPO	✔	✔	✔	✔	✔
QR-DQN [1]	✘	✔	✘	✘	✔
RecurrentPPO [1]	✔	✔	✔	✔	✔
SAC	✔	✘	✘	✘	✔
TD3	✔	✘	✘	✘	✔
TQC [1]	✔	✘	✘	✘	✔
TRPO [1]	✔	✔	✔	✔	✔
Maskable PPO [1]	✘	✔	✔	✔	✔

[1] (1,2,3,4,5,6): Implemented in SB3 Contrib

Figura 3.3: Tipo de acciones soportadas por algoritmos SB3 [16]

comparación con *MultiBinary*. La figura 3.3 muestra el estado actual de la compatibilidad de los algoritmos con cada uno de los tipos de espacios de acciones.

Si entramos dentro del código del entorno, podemos ver que existe una variable *atari_mode* que modifica el espacio de estado acorde a lo que queremos. Si se inicializa a *True* el espacio de acciones pasa a ser *Discrete(6)*. La modificamos. Este cambio es crítico, ya que de otra manera nuestros algoritmos no serán capaces de funcionar en el entorno:

```
atari_mode = True
...
if self.atari_mode:
      self.action_space = spaces.Discrete(6)
else:
      self.action_space = spaces.MultiBinary(3)
```

Respecto al espacio de estados, el entorno define un vector de valores continuos de 12 dimensiones. Este vector contiene tanto la posición como la velocidad del agente, la pelota y el oponente:

$$(x_{agent},\ y_{agent},\ \dot{x}_{agent},\ \dot{y}_{agent},\ x_{ball},\ y_{ball},\ \dot{x}_{ball},\ \dot{y}_{ball},\ x_{opp},\ y_{opp},\ \dot{x}_{opp},\ \dot{y}_{opp})$$

Estas posiciones y velocidades están definidos en un intervalo determinado.

3.2. Taxonomía de algoritmos RL

Los algoritmos de aprendizaje por refuerzo se pueden clasificar en función de distintos criterios [17].

- **Model-free**: Son algoritmos que no necesitan que el entorno tenga un modelo definido. Es decir, no se sabe de antemano cómo el entorno cambia ante una acción determinada.
  - **Basados en función valor**: Almacenan y actualizan una función valor o una aproximación de ella, sin que haya una política que esté representada explícitamente.
    - **Tabulares**: Se almacena el valor de cada uno de los estados o pares estado-acción del sistema.
    - **Aproximadores de la función valor**: Se utiliza una función que aproxima el valor de cada estado o pares estado-acción del sistema. En vez de actualizar directamente el valor de los estados o pares estado-acción, se actualizan los parámetros de las funciones que aproximan. En estos casos, el número de parámetros siempre será menor al número de estados posibles.
  - **Basados en política**: Almacenan y actualizan una política sin que haya una función valor que esté representada explícitamente.
  - **Actor-Critic**: Se trata de una combinación de los algoritmos basados en función valor y en política, ya que almacenan ambas.
- **Model-based**: Algoritmos que sí que necesitan que el entorno donde se aplican tenga un modelo definido. En este tipo, existe una función de recompensa y unas probabilidades de transición dadas.

En este proyecto, para la parte de aplicación de algoritmos clásicos, utilizaremos algoritmos model-free. Concretamente, basados en aproximación de la función valor (SARSA(λ)) y actor-critic (actor-critic con trazas de elegibilidad).

Para la parte de aplicación de algoritmos avanzados, los algoritmos también serán model-free. En concreto, basados en aproximación de la función valor (DQN), basados en política (PPO) y actor-critic (SAC).

3.3. Funcionamiento básico de algoritmos RL

De manera general, dentro del aprendizaje por refuerzo, tenemos dos problemas relacionados, pero diferentes, para resolver: predicción y control [18].

En predicción, el objetivo es estimar nuestra función valor utilizando una política determinada a partir de la experiencia en los episodios. Podemos definir la función valor en un estado s dada una política π como el retorno esperado empezando en s y usando π a partir de ese momento, tal como muestra la ecuación 3.1.

$$v_\pi(s) = \mathbb{E}_\pi \left[G_t \mid S_t = s \right] \tag{3.1}$$

Esta estimación se puede hacer de diferentes formas. Una de las maneras más básicas es la predicción Monte-Carlo, donde este estimador se calcula utilizando la media empírica, en vez del retorno esperado. Se muestra en la ecuación 3.2.

$$V(s) = \frac{\sum_{n=1}^{N(s)} G_{t(s)}^{(n)}}{N(s)} \tag{3.2}$$

Por otra parte, tenemos el problema de control. El objetivo aquí ahora es obtener la mejor política posible para maximizar la recompensa. El problema es que mejorar la política a partir de v_π —cuya estimación se calcula en predicción— requiere un modelo MDP:

$$\pi'(s) = \underset{a \in \mathcal{A}}{\operatorname{argmax}} \left[r(s,a) + \gamma \sum_{s'} p_{ss'}(a) v_\pi \left(s' \right) \right] \tag{3.3}$$

Como los algoritmos con los que trabajaremos son model-free, esta información es inaccesible para nosotros. Sin embargo, podemos considerar la mejora de la política a partir de la función q_π. De esta manera, nuestro estimador pasaría a ser $Q(s,a)$, tal como aparece en la ecuación 3.4.

$$Q(s,a) = \frac{\sum_{n=1}^{N(s,a)} G_{t(s,a)}^{(n)}}{N(s,a)} \tag{3.4}$$

El problema que surge a raíz de un estimador que utiliza la media empírica es que pueden existir pares estado-acción que no se visiten. Si empezamos en un estado determinado s, la política π que se usa en ese momento puede que haga que ciertas acciones puedan no ser tomadas.

La alternativa más popular ante este problema son las políticas *soft*, que hacen uso del compromiso exploración-explotación. De esta manera, a la hora de seleccionar las acciones

en un estado determinado, con una alta probabilidad elegiremos aquella que maximice la función Q, y con una probabilidad baja seleccionaremos una acción diferente, como puede ser una acción aleatoria. De esta manera estaríamos explorando nuevas acciones.

Como hemos visto en estos casos, las estimaciones $V(s)$ y $Q(s, a)$ que se realizan en predicción y control, requieren un valor para cada estado o para cada par estado-acción. Esto es propio de los métodos tabulares, que comentamos en la sección 3.2. Como veremos más adelante, esto no es una técnica que pueda ser aplicable a entornos de grandes dimensiones, debido al espacio y la cantidad de cómputo necesaria.

Además, en la práctica, para predicción por ejemplo, la estimación de la función valor en los algoritmos no se realiza mediante un único cálculo, sino que se va actualizando esta estimación a lo largo de la experiencia del agente.

Según hagamos esta actualización, estaremos siguiendo un tipo de predicción —o control— u otro. La actualización más sencilla es Monte-Carlo, que emplea retornos completos:

1. Observa: $G_t = R_{t+1} + \gamma R_{t+2} + \ldots + \gamma^{T-t-1} R_T$
2. Actualiza: $V(S_t) \leftarrow V(S_t) + \alpha [G_t - V(S_t)]$

Por otro lado, en los métodos Temporal Difference se realiza la actualización de forma diferente [19]. En vez de usar retornos completos, se actualiza esta estimación con cada recompensa:

1. Observa R_{t+1}
2. Estima retorno: $G_t \approx R_{t+1} + \gamma v_\pi (S_{t+1}) \approx R_{t+1} + \gamma V(S_{t+1})$
3. Actualiza: $V(S_t) \leftarrow V(S_t) + \alpha [R_{t+1} + \gamma V(S_{t+1}) - V(S_t)]$

Esta diferencia en la actualización presenta ventajas y desventajas tanto para MC como para TD. Principalmente, TD reduce la varianza a costa de tener algo de sesgo, mientras que MC tiene alta varianza, pero su sesgo es nulo.

Una forma adicional de actualizar estas estimaciones es la que propone TD(λ) [20]. Esta técnica genera un target a partir de una suma ponderada de los targets para cada valor de n. Esto quiere decir que podemos realizar una actualización teniendo en cuenta todas las etapas sucesivas (como pasaba en MC), pero ponderando qué etapas queremos tener más en cuenta.

Analíticamente se define como:

$$G_t^{\lambda} = (1-\lambda) \sum_{n=1}^{T-t-1} \lambda^{n-1} G_{t:t+n} + \lambda^{T-t-1} G_t \tag{3.5}$$

Donde si $\lambda = 1$ se realiza la actualización MC y si $\lambda = 0$ se realiza la actualización TD comentada anteriormente.

No obstante, para su implementación veremos su uso junto con trazas de elegibilidad, tal como se describe en el siguiente capítulo.

Capítulo 4

Algoritmos clásicos

Dentro de cada una de las categorías que comentamos en la sección 3.2, tenemos una gran variedad de métodos disponibles. Estos métodos varían según los distintos niveles de sofisticación, en función de la complejidad de los algoritmos y de los conceptos que usan para llevarse a cabo.

En esta parte del trabajo, queremos poner a prueba algoritmos que utilizan conceptos más clásicos dentro del ámbito del RL, y que por tanto dan lugar a algoritmos más sencillos, que típicamente se han usado en entornos más simples, con menor número de variables.

En nuestro caso particular, desarrollaremos y probaremos dos algoritmos diferentes. El primero de ellos será SARSA(λ), que forma parte de los métodos basados en aproximación de la función valor. El segundo es un método de los denominados Actor-Critic, en concreto aquel basado en trazas de elegibilidad.

4.1. Fundamentos teóricos

En esta sección detallaremos de forma teórica cuáles son los fundamentos de cada uno de los algoritmos desarrollados y en qué se basan para poder funcionar correctamente.

4.1.1. SARSA(λ)

Dentro de los métodos basados en función valor, dadas las características de nuestro problema —un entorno complejo, con una alta dimensionalidad en el espacio de observación y varias acciones—, los métodos tabulares no son computacionalmente viables debido al gran número de estados y pares estado-acción. Además, el espacio que se necesita en memoria es inasumible.

Por este motivo, la alternativa que surge son los métodos basados en aproximación de la función valor —en inglés, Value Function Approximation, VFA—. La idea en la que se apoyan estos métodos es la siguiente: la experiencia dentro de un subconjunto de estados se puede generalizar para producir una buena aproximación en un conjunto mayor [2].

Para realizar esta generalización se utilizan los llamados aproximadores de función. Estos se pueden emplear para estimar la función valor o la función q. De esta manera, en lugar de aprender valores individuales para estados y acciones, pasaríamos a aprender valores de un vector de pesos que hacen que la aproximación esté lo más cerca posible, generalizando de los estados observados a los no observados.

Para llegar a comprender SARSA(λ), es necesario aplicar los conceptos básicos de predicción dentro de RL y las variantes que tenemos —comentados en la sección 3.3— según el target que se utiliza en métodos VFA, así como el concepto de trazas de elegibilidad. Esto nos llevará a poder desarrollar el algoritmo de control SARSA(λ) con trazas de elegibilidad utilizado en el trabajo.

4.1.1.1. Predicción en métodos VFA

Para predicción, el concepto es el mismo que vimos en el capítulo 1: estimar nuestra función valor utilizando una política determinada a partir de la experiencia en los episodios. La diferencia se encuentra en que la función valor ya no se encuentra representada por una tabla, sino por el vector de parámetros w [20]:

$$\hat{v}(s, w) \approx v_\pi(s) \tag{4.1}$$

Ajustando estos pesos, podemos conseguir una variedad de funciones que se ajusten y aproximen nuestra función valor original. Normalmente, la dimensión del vector w es mucho menor que el número de estados, lo que permite que esta solución sea mucho más escalable que al trabajar con métodos tabulares.

En este escenario, necesitamos una métrica que nos permita evaluar nuestros métodos aproximadores de función. Una de las métricas que podemos utilizar es el error cuadrático:

$$\overline{VE}(w) = \sum_s \mu(s) \left[v_\pi(s) - \hat{v}(s, w)\right]^2 \tag{4.2}$$

Siendo $\mu(s)$ un valor entre [0, 1], tal que la suma para todo $s \in S$ es 1, representando una distribución de probabilidad que especifica la importancia relativa de los errores en los diferentes estados. Se emplea esta distribución porque normalmente es imposible reducir el error a cero en todos los estados, puesto que el número de estados es mayor al número de componentes de w.

En este escenario, la idea sería tratar de encontrar un vector de pesos óptimo w^*, de manera que minimice el error cuadrático para todo w.

Este problema de optimización puede tener distintos niveles de complejidad dependiendo de los aproximadores de funciones que utilicemos. En nuestro caso, utilizaremos funciones lineales, lo que hará que la complejidad se reduzca, tal como veremos más adelante.

Abordaremos este problema como un problema de optimización estocástica, utilizando el descenso del gradiente:

$$\min_{w} \overline{VE}(w) \tag{4.3}$$

En los métodos del descenso del gradiente, el vector de pesos es un vector columna con un número fijo de parámetros, cuyos componentes pertenecen a los números reales. La función valor aproximada es una función diferenciable de w para todos los estados pertenecientes a S.

En este caso, iremos actualizando el valor de w en cada etapa, intentando buscar un mínimo local, avanzando en la dirección del gradiente del error cuadrático respecto a w, siendo esta la dirección que reduce en mayor cantidad el error.

$$w \leftarrow w - \alpha \nabla_w \overline{VE}(w) \tag{4.4}$$

Si asumimos que todos los estados aparecen con la misma distribución $\mu(s)$ y que el ajuste lo realizamos para la estimación del valor de un estado S_t:

$$\begin{aligned} w_{t+1} &= w_t - \frac{1}{2}\alpha \nabla \left[v_\pi\left(S_t\right) - \hat{v}\left(S_t, w_t\right)\right]^2 \\ &= w_t + \alpha \left[v_\pi\left(S_t\right) - \hat{v}\left(S_t, w_t\right)\right] \nabla \hat{v}\left(S_t, w_t\right) \end{aligned} \tag{4.5}$$

Donde α es un parámetro que controla el *step-size* —cómo de grandes son los saltos que hago para llegar al mínimo— y el gradiente de la función valor aproximada se refiere al vector de derivadas parciales respecto a los componentes del vector de pesos:

$$\left(\frac{\partial f\left(w_t\right)}{\partial w_{t,1}}, \frac{\partial f\left(w_t\right)}{\partial w_{t,2}}, \ldots, \frac{\partial f\left(w_t\right)}{\partial w_{t,n}}\right)^\top \tag{4.6}$$

Normalmente, el parámetro α suele ser un valor pequeño —incluso se puede ir reduciendo a medida que pasan las etapas—, ya que idealmente las actualizaciones deberían realizarse en pequeños pasos para poder tener un balance entre los errores en los diferentes estados.

Como vemos en la ecuación 4.5, $v_\pi(S_t)$ se trata de un valor desconocido, ya que precisamente la función valor utilizando una política concreta es lo que buscamos obtener.

Por tanto, para resolver este problema, es necesario utilizar un estimador insesgado V_t. Que el estimador sea insesgado quiere decir que el valor esperado de V_t será igual a la función que buscamos estimar:

$$w_{t+1} = w_t + \alpha \left[V_t - \hat{v} \left(S_t, w_t \right) \right] \nabla \hat{v} \left(S_t, w_t \right) \tag{4.7}$$

Este estimador es lo que se denomina *target*. Según el target que utilicemos, como ya comentamos, estaremos ante un tipo de predicción diferente. Por ejemplo, la predicción Monte-Carlo utiliza $V_t = G_t$, ya que el valor real de un estado es el valor esperado del retorno que conlleva.

Del mismo modo, podríamos usar una predicción Temporal Difference, donde el target pasaría a ser $V_t = R_{t+1} + \gamma \hat{v} \left(S_{t+1}, w_t \right)$ para el caso de 1-step TD.

Tenemos que tener en cuenta que estos métodos que utilizan bootstraping son sesgados, ya que el target contiene el vector de pesos w, por lo que se denominan entonces métodos semi-gradiente.

Siguiendo esta línea, y utilizando los conceptos definidos en la sección 3.3 sobre TD(λ) podríamos usar también el retorno lambda como target, siendo $V_t = G_t^\lambda$. De esta manera:

$$w_{t+1} = w_t + \alpha \left[G_t^\lambda - \hat{v} \left(S_t, w_t \right) \right] \nabla \hat{v} \left(S_t, w_t \right) \tag{4.8}$$

Este concepto será utilizado en el desarrollo de nuestro algoritmo de control SARSA(λ), añadiendo el uso de trazas de elegibilidad y empleando el estimador de la función, el cuál veremos en la siguiente sección.

Antes de introducir el concepto de trazas de elegibilidad, es importante definir el término que nos queda en la ecuación 4.5: la función valor aproximada.

Existen múltiples formas de construir un aproximador: redes neuronales, kernels, decision trees, etc. En este caso optaremos por la solución más simple: métodos lineales. La razón por la que usamos estos métodos es que realizar la operación de gradiente se convierte en una tarea muy sencilla.

Los métodos lineales aproximan la función valor por el producto escalar de $x(s)$ y w:

$$\hat{v}(s, w) = w^\top x(s) = \sum_{i=1}^{n} w_i x_i(s) \tag{4.9}$$

donde $x(s)$ es un vector de características que sustituye a los estados del sistema. Este vector se puede construir según distintos enfoques. En nuestro caso lo haremos usando funciones base radiales.

Las funciones base radiales (en inglés, Radial Basis Functions, RBF) funcionan como una generalización de la cuadrícula para espacios de estados continuos, donde las características presentan una respuesta gaussiana. La idea es subdividir el espacio de acciones

fijando unos centros de estados, donde la cercanía o lejanía a ellos define el vector de características. Analíticamente:

$$x_i(s) \doteq \exp\left(-\frac{\|s - c_i\|^2}{2\sigma_i^2}\right) \tag{4.10}$$

Este concepto se extiende en el apartado 4.2.1.2, pero como vemos en la ecuación, el valor del vector de características para un estado depende de la distancia a unos centros determinados y del denominado ancho de banda, σ_i.

Dada la ecuación 4.9, realizar el cálculo de su gradiente nos da el siguiente resultado:

$$\nabla \hat{v}(s, w) = x(s) \tag{4.11}$$

Esto simplifica el problema considerablemente. Además, en el caso lineal sólo hay un óptimo w^*, por lo que cualquier método que garantiza convergencia —o cerca de convergencia— a un óptimo local, también garantiza convergencia —o cerca de convergencia— a un óptimo global.

4.1.1.2. Trazas de elegibilidad

Se puede definir una traza de elegibilidad como un vector z_t que posee las mismas dimensiones que el vector de pesos w_t y que modula la influencia del error TD en la actualización del valor almacenado en el estado s [20]. El error TD es la diferencia entre el target TD y la estimación anterior:

$$\delta_t = R_{t+1} + \gamma V\left(S_{t+1}\right) - V\left(S_t\right) \tag{4.12}$$

De esta manera,

- w_t es una memoria a largo plazo, que se mantiene en episodios sucesivos.
- z_t es una memoria a corto plazo, que se renueva en menos de T etapas, ayudando al proceso de aprendizaje, ya que controla cómo se propaga el efecto de los errores TD a lo largo del tiempo.

El uso de estas trazas permite hacer que nuestros algoritmos trabajen de forma incremental, implementando la *visión retrospectiva* en el sistema. Esto contrasta con el uso del retorno lambda $V_t = G_t^{\lambda}$, que implementa la *forward view* del descenso del gradiente TD(λ).

Estas dos visiones —views, en inglés— son formas de entender el concepto de trazas de elegibilidad.

En primer lugar, la *forward view* nos ofrece una interpretación más directa, ya que calculan directamente el retorno lambda. Según esta visión, las trazas de elegibilidad son un puente entre métodos Temporal Difference y métodos Monte Carlo.

Si nos centramos en el problema de predicción, sabemos que Monte-Carlo emplea retornos completos para actualizar la función valor, mientras que TD(0) la actualiza a una etapa vista. En TD(n), podemos seleccionar el número de etapas que consideramos para actualizar (cómo de lejos queremos llegar en las futuras recompensas). Sin embargo, en los algoritmos no siempre está claro cómo ajustar n, es decir, cuántas etapas debemos considerar a la hora de actualizar la función valor.

Por otro lado, la *backward view* no obtiene explícitamente el retorno lambda, sino que lo considera indirectamente mediante el uso de las trazas de elegibilidad. Con este enfoque, una traza de elegibilidad sería como un registro temporal de la ocurrencia de un evento, tal como la visita a un estado o la toma de una acción.

Por tanto, ahora la idea sería actualizar nuestro vector de pesos según la siguiente ecuación:

$$w_{t+1} = w_t + \alpha \delta_t z_t \tag{4.13}$$

donde δ_t sería el error TD definido anteriormente, que ahora usa la función valor aproximada:

$$\delta_t = R_{t+1} + \gamma \hat{v}\left(S_{t+1}, w_t\right) - \hat{v}\left(S_t, w_t\right) \tag{4.14}$$

y donde z_t sería un vector columna de trazas de elegibilidad, una para cada componente de w_t, que se actualiza de la siguiente manera en cada etapa del episodio, siendo $z_{-1} = 0$:

$$z_t = \gamma \lambda z_{t-1} + \nabla \hat{v}\left(S_t, w_t\right) \tag{4.15}$$

De esta forma, z_t llevaría la cuenta de qué componentes de w_t han contribuido a las valoraciones recientes de estados, indicando qué elementos de w_t son elegibles para ser actualizados. Si en $t-1$, un componente del vector de pesos anterior fue importante, esto significa que en la t también tendrá alguna importancia, por lo que será necesario actualizarlo.

4.1.1.3. Control con SARSA(λ)

En el algoritmo 1 se muestra un pseudocódigo de la implementación del control para SARSA(λ) [2].

Algorithm 1: Semi-gradient SARSA(λ) for estimating $\hat{q} \approx q_*$

Input: the policy π to be evaluated;
Input: a differentiable function $\hat{v} : \mathcal{S}^+ \times \mathbb{R}^d \to \mathbb{R}$ such that $\hat{v}(terminal, \cdot) = 0$;
Algorithm parameters: step size $\alpha > 0$, trace decay rate $\lambda \in [0, 1]$;
Initialize value function weights w arbitrarily (e.g. $w = 0$);
foreach *episode* **do**
  Initialize S;
  $z \leftarrow 0$;
  foreach *step of episode* **do**
    Choose $A \sim \pi(\cdot \mid S)$;
    Take action A, observe R, S';
    $z \leftarrow \gamma\lambda z + \nabla\hat{q}(S, A, w)$;
    $\delta \leftarrow R + \gamma\hat{q}\left(S', A', w\right) - \hat{q}(S, A, w)$;
    $w \leftarrow w + \alpha\delta z$;
    $S \leftarrow S'$;
  end foreach
end foreach

4.1.2. Actor-Critic con trazas de elegibilidad

Actor-Critic es la tercera rama dentro de los algoritmos RL model-free. Al no trabajar exclusivamente con la función valor ni tampoco con la política, forman una rama propia dentro del aprendizaje por refuerzo.

Por tanto, debido al enfoque distinto que presentan, es necesario primero ver de qué tratan los métodos actor-critic en general, además de hacer un repaso a conceptos de *policy gradient* para tratar el concepto de actor. Una vez desarrollados estos puntos, podemos entrar a describir cómo funciona el algoritmo actor-critic con el uso de trazas de elegibilidad.

4.1.2.1. Métodos Actor-Critic

La idea básica de estos métodos es que, además de aprender una función valor, aprenden también una política, siendo esta política la que usan para seleccionar las acciones [2] [21].

En realidad, son métodos temporal difference que tienen una estructura de memoria separada para representar la política de manera independiente de la función valor. La estructura que aprende la función valor aproximada actualizando el vector de pesos w se conoce como *crítico*. La estructura que contiene la política, que se ha de optimizar

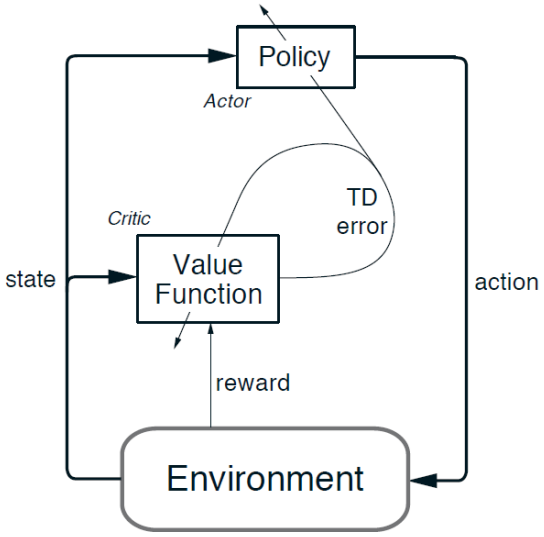

Figura 4.1: Arquitectura actor-critic [2]

actualizando θ y siguiendo la dirección marcada por el crítico se conoce como *actor*. De esta manera, el actor intenta mejorar la política actual, mientras que el crítico trata de ayudar al actor evaluando la política actual.

En la figura 4.1 se puede ver una ilustración de esta arquitectura.

En lo que se refiere al actor, sigue el modelo *policy gradient*: trabajamos con un vector de parámetros θ, a partir del cual obtenemos una política parametrizada π_θ, tal que

$$\pi_\theta(a \mid s) = \mathbb{P}\left[A_t = a \mid S_t = s, \theta_t = \theta\right] \tag{4.16}$$

nos da la probabilidad de seleccionar la acción a en la etapa t dado el estado s y el vector de parámetros θ.

Por tanto, dado π_θ, el objetivo será encontrar el θ óptimo: dar mayor probabilidad a las mejores acciones según el estado en el que nos encontremos, esto es, las acciones que nos hagan conseguir un valor esperado más alto.

Para medir la calidad de π_θ utilizamos el retorno esperado asociado a π_θ. Como nuestro entorno es episódico:

$$\mathcal{J}(\theta) = v_{\pi_\theta}(s_0) \tag{4.17}$$

Los métodos de optimización de políticas buscan un máximo local ascendiendo el gradiente de la política respecto a θ:

$$\theta \leftarrow \theta + \alpha \nabla_\theta \mathcal{J}(\theta) \tag{4.18}$$

El gradiente de $J(\theta)$ es desconocido, por lo que el objetivo es que las actualizaciones de los métodos policy gradient aproximen el ascenso del gradiente. El estimador será una variable aleatoria cuyo valor esperado sea gradiente de $J(\theta)$.

Computar el gradiente de $J(\theta)$ es una tarea compleja porque depende de la acción seleccionada —dada por π_θ— y la distribución estacionaria de los estados $d_{\pi_\theta}(s)$ —dada por las probabilidades de transición, que dependen del entorno y π_θ—.

Es el teorema del gradiente de la política el que nos proporciona una formulación del gradiente de $J(\theta)$ en la que no intervienen los gradientes de $q_{\pi_\theta}(s,a)$ ni de $d_{\pi_\theta}(s)$.

Aunque no entremos en la demostración de dicho teorema, su resultado es el siguiente:

$$\nabla_\theta \mathcal{J}(\theta) \propto \sum_s d_{\pi_\theta}(s) \sum_a q_{\pi_\theta}(s,a) \nabla \pi_\theta(a \mid s) \tag{4.19}$$

Manipulando esta expresión, podemos llegar a la siguiente:

$$\nabla_\theta \mathcal{J}(\theta) = \mathbb{E}_{\pi_\theta} \left[q_{\pi_\theta}(S_t, A_t) \frac{\nabla \pi_\theta(A_t \mid S_t)}{\pi_\theta(A_t \mid S_t)} \right] \tag{4.20}$$

Según la definición de función q, podemos sustituir lo siguiente:

$$\nabla_\theta \mathcal{J}(\theta) = \mathbb{E}_{\pi_\theta} \left[G_t \frac{\nabla \pi_\theta(A_t \mid S_t)}{\pi_\theta(A_t \mid S_t)} \right] \tag{4.21}$$

Finalmente, teniendo en cuenta la actualización policy gradient, definida en la ecuación 4.18, aplicando los conceptos explicados, la actualización REINFORCE se define de la siguiente manera:

$$\theta_{t+1} = \theta_t + \alpha G_t \frac{\nabla \pi_\theta(A_t \mid S_t)}{\pi_\theta(A_t \mid S_t)} \tag{4.22}$$

El teorema del gradiente de la política puede generalizarse para incluir una comparación de $q_{\pi_\theta}(s,a)$ respecto a una referencia o *baseline*:

$$\nabla_\theta \mathcal{J}(\theta) \propto \sum_s d_{\pi_\theta}(s) \sum_a \left(q_{\pi_\theta}(s,a) - b(s) \right) \nabla \pi_\theta(a \mid s) \tag{4.23}$$

El baseline puede ser cualquier función o variable aleatoria, siempre que no varíe con la acción. En su lugar, este baseline debe variar con los estados: estados cuyas acciones tienen todas valores altos tendrán un baseline alto, mientras que estados con acciones con valores bajos tendrán baseline bajo.

En este caso, la elección natural es el estimador del valor del estado, dado por $\hat{v}(s,w) \approx v_{\pi_\theta}(s)$.

La dimensión del vector w será distinta a la dimensión del vector θ, de manera que w se aprende en paralelo a θ. Si realizamos la sustitución, tendríamos otro tipo de actualización: actualización REINFORCEMENT con baseline:

$$\theta_{t+1} = \theta_t + \alpha \left(G_t - \hat{v}(S_t, w)\right) \nabla \ln \pi_\theta \left(A_t \mid S_t\right) \tag{4.24}$$

Si queremos ir un paso más al realizar la actualización de θ, llegamos a actor-critic, donde la función valor aproximada no sólo se aprende como un baseline, sino también para *bootstrapping*, utilizando el target TD en lugar de Monte Carlo para actualizar la estimación del valor del estado a partir de los valores estimados de estados posteriores.

$$\theta_{t+1} = \theta_t + \alpha \left(R_{t+1} + \gamma \hat{v} \left(S_{t+1}, w_t\right) - \hat{v} \left(S_t, w_t\right)\right) \nabla \ln \pi_\theta \left(A_t \mid S_t\right) \tag{4.25}$$

Al igual que en la comparativa TD vs Monte Carlo, utilizar bootstrapping introduce sesgo pero reduce varianza.

Por otro lado, en lo que se refiere al crítico, el problema es más similar a lo que vimos en los métodos basados en aproximación de la función valor, ya que se trata de un problema de predicción.

Después de seleccionar cada acción por parte del actor, el crítico evalúa el nuevo estado para determinar si el resultado ha sido mejor o peor que el esperado. Esta evaluación se realiza con el error TD:

$$\delta_t = R_{t+1} + \gamma V_t \left(S_{t+1}\right) - V \left(S_t\right) \tag{4.26}$$

siendo V_t la función valor implementada por el crítico en la etapa t. Este error TD puede usarse para evaluar la acción A_t en el estado S_t. Si el error TD es positivo, significa que la tendencia a seleccionar A_t debe reforzarse en etapas posteriores, mientras que si este error es negativo, la tendencia debe disminuir.

Teniendo en cuenta lo comentado anteriormente, un algoritmo actor-critic realizaría 2 actualizaciones básicas.

Por un lado, es necesario, actualizar el vector de pesos w, que forma parte del crítico. Por otro lado, debemos actualizar el vector de la política parametrizada θ, que forma parte del actor:

$$\begin{aligned} w_{t+1} &= w_t + \alpha \left(R_{t+1} + \gamma \hat{v} \left(S_{t+1}, w_t\right) - \hat{v} \left(S_t, w_t\right)\right) \nabla \hat{v} \left(S_t, w_t\right) \\ \theta_{t+1} &= \theta_t + \alpha \left(R_{t+1} + \gamma \hat{v} \left(S_{t+1}, w_t\right) - \hat{v} \left(S_t, w_t\right)\right) \nabla \ln \pi_\theta \left(A_t \mid S_t\right) \end{aligned} \tag{4.27}$$

4.1.2.2. Control con AC + trazas de elegibilidad

Como vemos en las actualizaciones anteriores, en ambas empleamos el retorno TD(0). El hecho diferencial del método que usa trazas de elegibilidad es que ahora utilizaremos el retorno λ, tal como vimos en la sección 4.1.1:

$$
\begin{aligned}
w_{t+1} &= w_t + \alpha \left(G_t^{\lambda} - \hat{v}\left(S_t, w_t\right) \right) \nabla \hat{v}\left(S_t, w_t\right) \\
\theta_{t+1} &= \theta_t + \alpha \left(G_t^{\lambda} - \hat{v}\left(S_t, w_t\right) \right) \nabla \ln \pi_\theta \left(A_t \mid S_t\right)
\end{aligned}
\tag{4.28}
$$

Si lo expresamos con visión retrospectiva, nos encontramos con dos parámetros λ, así como dos vectores de trazas de elegibilidad —para el crítico y para el actor—. Las actualizaciones de los vectores w y θ se realizan empleando estos vectores de trazas de elegibilidad junto con sus respectivos λ:

$$
\begin{aligned}
z_{t+1}^{w} &= \gamma \lambda^{w} z_t^{w} + \nabla_\theta \hat{v}\left(S_t, w_t\right) \\
z_{t+1}^{\theta} &= \gamma \lambda^{\theta} z_t^{\theta} + \gamma^{t} \nabla \ln \pi_\theta \left(A_t \mid S_t\right) \\
\delta_t &= R_{t+1} + \gamma \hat{v}\left(S_{t+1}, w_t\right) - \hat{v}\left(S_t, w_t\right) \\
w_{t+1} &= w_t + \alpha^{w} \delta_t z_{t+1}^{w} \\
\theta_{t+1} &= \theta_t + \alpha^{\theta} \delta_t z_{t+1}^{\theta}
\end{aligned}
\tag{4.29}
$$

De esta manera, el algoritmo definitivo a implementar es el que se muestra en el pseudocódigo 2.

4.2. Metodología

En esta sección, describiremos en detalle cómo ha sido el desarrollo y los experimentos que realizamos utilizando los algoritmos descritos en la sección anterior.

4.2.1. Consideraciones generales

A la hora de desarrollar ambos algoritmos, hay algunos puntos comunes a tener en cuenta.

4.2.1.1. Uso de wrappers

En la implementación, haremos uso de lo que se denominan *wrappers*. En Gym, un *wrapper* es una interfaz que utilizamos entre nuestro código y el código del entorno, que sirve para realizar cambios en el funcionamiento del entorno, sin tener que modificar el entorno directamente [22]. Son los siguientes:

- *ReducedDimension*: Es un wrapper del tipo *ObservationWrapper*, el cual se utiliza para modificar la observación del estado que se devuelve al tomar un paso en la

Algorithm 2: Actor-Critic with Elegibility Traces (episodic) for estimating $\pi_\theta \approx \pi*$

Input: a differentiable policy parameterization $\pi(a \mid s, \theta)$;
Input: a differentiable state-value function parameterization $\hat{v}(s, w)$;
Parameters: trace-decay rates $\lambda^\theta \in [0,1], \lambda^w \in [0,1]$; step sizes $\alpha^\theta > 0, \alpha^w > 0$;
Initialize policy parameter $\theta \in \mathbb{R}^{d'}$ and state-value weights $w \in \mathbb{R}^d$ (e.g., to 0);
foreach *episode* **do**
  Initialize S (first state of episode);
  $z^\theta \leftarrow 0$ (d'-component eligibility trace vector);
  $z^w \leftarrow 0$(d-component eligibility trace vector);
  $I \leftarrow 1$;
  foreach *step of episode while S is not terminal* **do**
    $A \sim \pi(\cdot \mid S, \theta)$;
    Take action A, observe S', R;
    $\delta \leftarrow R + \gamma \hat{v}\left(S', w\right) - \hat{v}(S, w)$ (if S' is terminal, then $\hat{v}\left(S', w\right) \doteq 0$);
    $z^w \leftarrow \gamma \lambda^w z^w + \nabla \hat{v}(S, w)$;
    $z^\theta \leftarrow \gamma \lambda^\theta z^\theta + I \nabla \ln \pi(A \mid S, \theta)$;
    $w \leftarrow w + \alpha^w \delta z^w$;
    $\theta \leftarrow \theta + \alpha^\theta \delta z^\theta$;
    $I \leftarrow \gamma I$;
    $S \leftarrow S'$;
  end foreach
end foreach

simulación del entorno. Concretamente, buscamos reducir el número de dimensiones. Como comentamos, por defecto, el entorno nos da información de posición y velocidad acerca del oponente, el agente y la pelota. A la hora de entrenar un agente, considereamos que la información sobre nuestro oponente no es lo suficiente relevante, así que descartamos esas 4 dimensiones. Este wrapper está implementado por nosotros, y su implementación se muestra a continuación:

```
class ReducedDimension(gym.ObservationWrapper):
    def __init__(self, env):
        super().__init__(env)
        self._observation_space = gym.spaces.Box(shape=(8,), low=-2.0,
        ↪  high=2.0)
    def observation(self, obs):
        return obs[0:8]
```

- *SurvivalRewardEnv*: Es un wrapper del tipo *RewardWrapper*, que se utiliza para modificar las recompensas que nos devuelve el entorno. Concretamente, esta clase busca que la recompensa dependa, no solo de las vidas que mantenga nuestro agente, sino también en función del número de etapas que sobreviva, añadiendo un bonus de 0.01 por cada etapa. Elegimos esta implementación para el entrenamiento, ya que tiene sentido recompensar al jugador el hecho de aguantar durante un episodio, aunque no gane el punto necesariamente. Este wrapper viene implementado en el entorno SlimeVolley, por lo que solamente es necesario usarlo. Su implementación es la siguiente:

```
class SurvivalRewardEnv(gym.RewardWrapper):
    def __init__(self, env):
        """
        adds 0.01 to the reward for every timestep agent survives

        :param env: (Gym Environment) the environment
        """
        gym.RewardWrapper.__init__(self, env)
    def reward(self, reward):
        """
        adds that extra survival bonus for living a bit longer!

        :param reward: (float)
        """
        return reward + 0.01
```

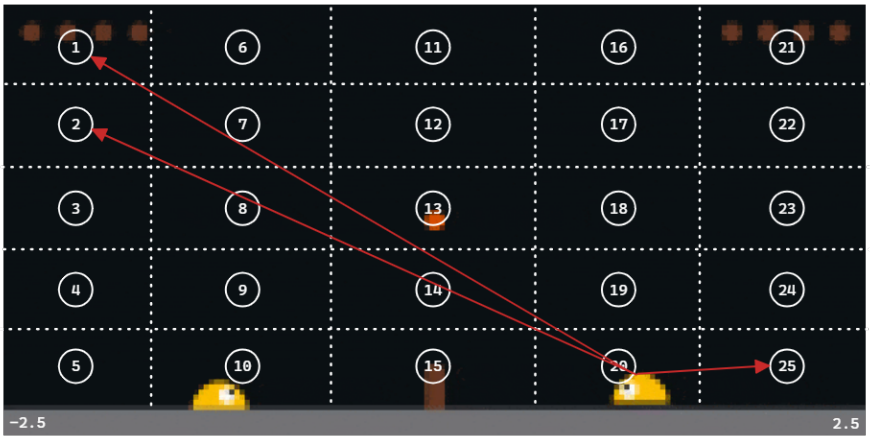

Figura 4.2: Ejemplo simplificado *BoundFeaturizer*

4.2.1.2. Obtención del vector de características

Para obtener nuestro vector de características, descrito teóricamente en la sección 4.1.1.1, utilizaremos dos alternativas para poder comparar el rendimiento de cada una de ellas. Ambas se basan en el uso de métodos lineales y funciones base radiales, que comentamos en la sección 4.1.1. La diferencia se encuentra en cuáles son los centros que se utilizan a la hora de realizar los cálculos.

Por un lado, tenemos la clase *BoundFeaturizer*. En este caso, los centros que utilizamos para calcular los elementos de nuestro vector de características, se calculan haciendo una división en partes iguales del espacio de observaciones. Para ello, además del ancho de banda —que controla qué influencia tiene cada uno de los centros—, necesitamos especificar el número de particiones que queremos, así como los límites del espacio de observaciones. Sabiendo que el vector del estado que obtenemos es de 8 dimensiones y el número de particiones elegido para cada dimensión es 3, tendremos un total de 3^8 dimensiones para nuestro vector de características.

En la figura 4.2 podemos ver un ejemplo muy simplificado. Este ejemplo usa un espacio de estados de dos dimensiones —las coordenadas (x, y) de nuestro agente—. Elegimos realizar 5 particiones, dividiendo así el espacio en 2^5 secciones, cada una de ellas con un centro que aparece numerado en la figura. Nuestro vector de características en este caso sería la gaussiana de la posición del agente en un t determinado a cada uno de estos centros. En el ejemplo del estado que se presenta en la imagen, el valor de las componentes 1 y 2 del vector $x(s)$ sería bajo —la influencia de los centros es baja, dado que la distancia estos centros es alta—, mientras que el valor del mismo vector para las componentes 15, 19, 20 o 25 será alto —estos centros tienen mayor influencia, ya que la distancia a los respectivos centros es baja—.

Su implementación es la siguiente:

```
class BoundFeaturizer:
    def __init__(self, observation_space, partitions = 10, sigma = 0.1, bound =
    ↪  5.0):
        self.n_dim = len(observation_space.low)
        self.sigma = sigma
        offset, scale_factor = [], []
        for i in range(self.n_dim):
            lower_bound = max(-1*bound, observation_space.low[i])
            upper_bound = min(bound, observation_space.high[i])
            offset.append(-1* lower_bound)
            scale_factor.append(1/(upper_bound - lower_bound))
        self.offset = np.array(offset, dtype=np.float_)
        self.scale_factor = np.array(scale_factor ,dtype=np.float_)
        scale = np.linspace(1.0/partitions,1.0,partitions) - 0.5/partitions
        self.landmarks = np.array(list(product(scale, repeat=self.n_dim)))
        self.n_parameters = partitions**self.n_dim
    def feature_vector(self, s):
        s = (s + self.offset) * self.scale_factor
        dist = (self.landmarks - s)**2
        f = np.exp(-dist.sum(axis=1)/(2*self.sigma**2))
        return f
```

Por otro lado, tenemos la clase *ClusteringFeaturizer*. Aquí, los centros que empleamos vendrán dados por los centroides de los clusters resultantes tras aplicar un algoritmo de clustering a un conjunto de estados muestreados en una simulación anterior. El 90 % de los estados muestreados vendrán de una simulación donde el agente emplea la política baseline (igual que la del oponente), mientras que el 10 % restante se obtiene en una simulación donde el agente utiliza una política aleatoria.

Para el muestreo y el guardado de información realizamos la siguiente implementación:

```
class Slime:
    ...
    def sample_states(self, n_samples, mode):
        obs = self.env.reset()
        sampled_states = np.empty((n_samples, 12))
        for i in range(n_samples):
            if (mode == 'RANDOM'):
                action = self.env.action_space.sample()
            elif (mode == 'MODEL'):
                action = self.model.predict(obs)
```

```
            obs, reward, done, info = self.env.step(action)
            if (i % 1000 == 0):
                print(f'Sample n = {i}')
            sampled_states[i] = info["state"]
        return sampled_states

slime = Slime('BSLN')
random_ss = slime.sample_states(100_000, 'RANDOM')
bsln_ss = slime.sample_states(900_000, 'MODEL')
head = 'x_agent,y_agent,xdot_agent,ydot_agent,' \
    'x_ball,y_ball,xdot_ball,ydot_ball,' \
    'x_opponent,y_opponent,xdot_opponent,ydot_opponent'
def save_ss_file(ss, name):
    np.savetxt(fname=join(DATADIR, name),
            X=ss,
            fmt='%.5f',
            delimiter=',',
            header=head,
            comments='')
save_ss_file(random_ss, "ss_random_100k.csv")
save_ss_file(bsln_ss, "ss_bsln_900k.csv")
```

Para la aplicación de algoritmos de clustering —concretamente *MiniBatchKMeans*— y el guardado de datos realizamos la implementación que aparece a continuación:

```
ss_bsln = pd.read_csv(join(DATADIR, 'ss_bsln_900k.csv')).iloc[:, :8]
ss_random = pd.read_csv(join(DATADIR, 'ss_random_100k.csv')).iloc[:, :8]
ss = pd.concat([ss_bsln, ss_random],
↪  ignore_index=True).sample(frac=1).to_numpy()
mbkm_model_5 = MiniBatchKMeans(n_clusters=5_000, random_state=0,
↪  batch_size=2048, verbose=True)
mbkm_model_5.fit(ss)      # 2min
mbkm_model_10 = MiniBatchKMeans(n_clusters=10_000, random_state=0,
↪  batch_size=2048, verbose=True)
mbkm_model_10.fit(ss)      # 4min

def save_centroids_file(model, name):
    head = 'x_agent,y_agent,xdot_agent,ydot_agent,' \
        'x_ball,y_ball,xdot_ball,ydot_ball'
    np.savetxt(fname=join(DATADIR, name),
        X=model.cluster_centers_,
        fmt='%.5f',
        delimiter=',',
        header=head,
```

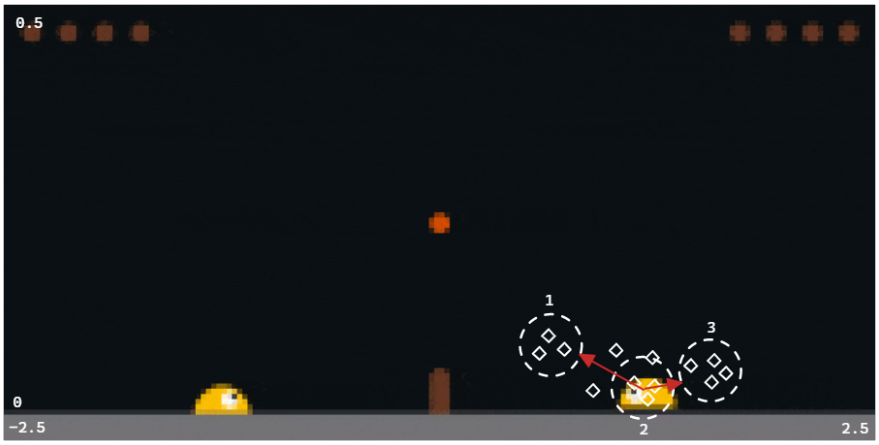

Figura 4.3: Ejemplo simplificado *ClusteringFeaturizer*

```
        comments='')

save_centroids_file(mbkm_model_5, "c_5_mix.csv")
save_centroids_file(mbkm_model_10, "c_10_mix.csv")
```

En la figura 4.3 vemos otro ejemplo simplificado en dos dimensiones —coordenadas (x, y) del agente—. Imaginamos que los símbolos de rombos son estados muestreados en una simulación previa. Una vez que tenemos un número definido de muestras —en este ejemplo 13, en nuestro código 1 millón—, aplicamos un algoritmo de clustering para buscar clusters que se ajusten a nuestras muestras. Suponemos que para la ejecución del algoritmo en este ejemplo especificamos 3 clusters, pero para nuestras muestras utilizamos 5000 y 10000. Al igual que antes, nuestro vector de características será la gaussiana de cada uno de los centros, y en este caso este vector será tridimensional. Para el estado de la imagen, la segunda característica del vector tendrá el mayor valor —mayor cercanía a la posición del agente—, mientras que la primera y tercera característica tendrán un valor menor —los centros están más lejanos—.

La implementación de *ClusteringFeaturizer* se detalla a continuación:

```
class ClusteringFeaturizer:
    def __init__(self, observation_space, centroids, sigma=0.5):
        self.n_dim = len(observation_space.low)
        self.centroids = np.array(centroids)
        self.sigma = sigma
        self.n_parameters = len(self.centroids)
    def feature_vector(self, s):
        dist = (self.centroids - np.array(s))**2
```

```
        x = np.exp(-dist.sum(axis=1)/(2*self.sigma**2))
        return x
```

En ambos casos, el parámetro σ determina el ancho de banda de nuestras funciones base radiales, por lo que un ancho de banda alto hará que un estado se vea afectado por centros cercanos y lejanos, mientras que un ancho de banda bajo hará que un estado solo se vea afectado por los centros más cercanos. La idea es elegir un valor que esté equilibrado, pudiendo tener un vector de características con valores suficientemente significativos, pero sin llegar a saturar.

Para elegir σ empíricamente, desarrollamos unas líneas de código que nos muestran gráficamente los valores del vector de características resultante de un estado cualquiera de nuestros estados muestreados, usando un objeto *BoundFeaturizer*:

```
ss_bsln = pd.read_csv(join(DATADIR, 'ss_bsln_900k.csv')).iloc[:, :8]
ss_random = pd.read_csv(join(DATADIR, 'ss_random_100k.csv')).iloc[:, :8]
ss = pd.concat([ss_bsln, ss_random],
↪  ignore_index=True).sample(frac=1).to_numpy()
featurizer_test = BoundFeaturizer(slime_env.observation_space, 3, sigma = 0.2,
↪  bound = 2.0)
x = featurizer_test.feature_vector(ss[0, :])
fix, ax = plt.subplots()
ax.plot(x, linewidth=1)
ax.set_title('Example of feature vector values')
ax.set_ylabel('$x_i(s)$')
ax.set_xlabel('$i$');
```

Realizamos el experimento para σ=0.1, σ=0.2 y σ=0.8. Como vemos, σ=0.2 sería el valor a elegir en este *BoundFeaturizer*. Este proceso se realizaría de forma equivalente para *ClusteringFeaturizer*. Presentamos estos resultados en esta sección, en la figura 4.4, ya que serán claves a la hora de realizar los experimentos con los algoritmos.

4.2.2. Experimentos con SARSA(λ)

A la hora de realizar nuestros experimentos con estos algoritmos, observamos que en problemas de RL no hay un proceso concreto a seguir ni unos hiperparámetros determinados que nos aseguren la obtención de buenos resultados. Por tanto, debemos realizar un gran número de pruebas para abordar gran parte de los casos.

En el caso de SARSA(λ), seguimos un proceso de 3 pasos:

1. Ejecutamos una batería de 9 tests según el ajuste de los hiperparámetros λ, ϵ y α.

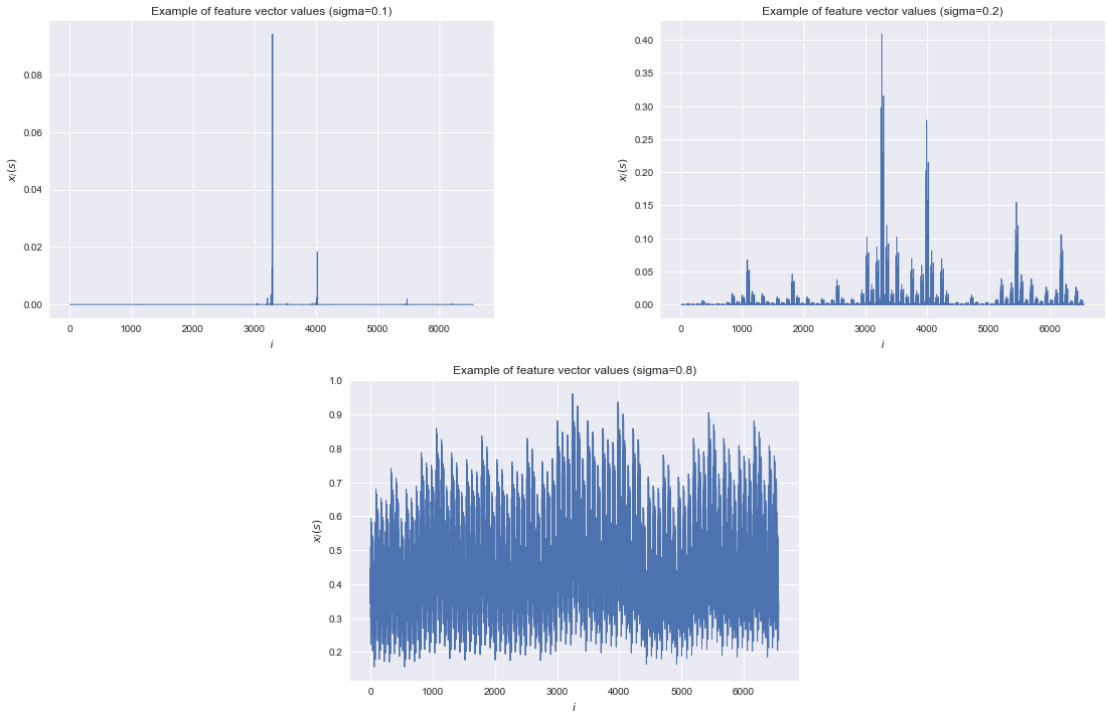

Figura 4.4: Comparativa de valores de σ para *BoundFeaturizer*

Las configuraciones se detallan en el cuadro 4.1. Todos los experimentos tendrán las siguientes características en común:

- Vector de características empleando *BoundFeaturizer* con 3 particiones, límites de 2.0 y $\sigma = 0.2$.
- 2 millones de etapas de entrenamiento, lo que equivale alrededor de 3000 episodios, dependiendo de la duración de los mismos.
- Factor de descuento $\gamma = 1$, dado que es un entorno episódico.

2. Según los resultados de la primera batería de pruebas, añadimos cuatro experimentos más, utilizando configuraciones parecidas a aquel que mejores resultados nos reporta. En concreto, probamos a modificar el *step-size* y aumentar ligeramente el valor de λ. Las configuraciones se detallan en el cuadro 4.2.

3. Al igual que antes, partiendo de la primera batería de pruebas, volvemos a ejecutar los tests que nos reportaron mejores resultados, pero esta vez empleando los vectores de características dados por *ClusteringFeaturizer*, especificando 10000 centroides a la hora de ejecutar el algoritmo de clustering. Las configuraciones se detallan en el cuadro 4.3.

	λ	ϵ	α
TEST #1	0.99	0.1	0.1
TEST #2	0.99	0.3	0.3
TEST #3	0.99	0.5	0.5
TEST #4	0.9	0.1	0.1
TEST #5	0.9	0.3	0.3
TEST #6	0.9	0.5	0.5
TEST #7	0.7	0.1	0.1
TEST #8	0.7	0.3	0.3
TEST #9	0.7	0.5	0.5

Cuadro 4.1: Primer conjunto de tests para SARSA(λ)

	λ	ϵ	α
TEST #10	0.9	0.1	0.03
TEST #11	0.9	0.1	0.01
TEST #12	0.95	0.1	0.1
TEST #13	0.9	0.1	0.2

Cuadro 4.2: Segundo conjunto de tests para SARSA(λ)

	λ	ϵ	α	**Featurizer**
TEST #14	0.9	0.1	0.1	ClusteringFeaturizer (10K)
TEST #15	0.9	0.3	0.3	ClusteringFeaturizer (10K)
TEST #16	0.9	0.5	0.5	ClusteringFeaturizer (10K)

Cuadro 4.3: Tercer conjunto de tests para SARSA(λ)

	α	β	λ_{critic}	λ_{actor}
TEST #17	0.005	0.0001	0.5	0.5
TEST #18	0.001	0.005	0.5	0.5
TEST #19	0.01	0.05	0.5	0.5
TEST #20	0.005	0.0001	0.6	0.6
TEST #21	0.001	0.005	0.6	0.6
TEST #22	0.01	0.05	0.6	0.6
TEST #23	0.005	0.0001	0.8	0.8
TEST #24	0.001	0.005	0.8	0.8
TEST #25	0.01	0.05	0.8	0.8

Cuadro 4.4: Primer conjunto de tests para AC+trazas

	α	β	λ_{critic}	λ_{actor}
TEST #26	0.001	0.005	0.9	0.9
TEST #27	0.001	0.005	0.95	0.95

Cuadro 4.5: Segundo conjunto de tests para AC+trazas

4.2.3. Experimentos con AC y trazas de elegibilidad

En el caso de Actor-Critic con trazas de elegibilidad, seguimos un proceso similar al caso anterior:

1. Ejecutamos una batería de 9 tests con las configuraciones de hiperparámetros que se detallan en el cuadro 4.4. Al igual que antes, se usa *BoundFeaturizer* (σ=0.2, bound=2.0, 3 particiones), 2 millones de etapas y factor de descuento $\gamma = 1$.

2. Según los resultados de estas primeras pruebas, añadimos dos más, en las que usamos valores de λ más altos. Las configuraciones se detallan en el cuadro 4.5.

3. De la misma manera, volvemos a ejecutar el test que nos da mejores resultados, pero en este caso empleando los vectores de características dados por *Clustering-Featurizer*, especificando 5000 y 10000 clusters a la hora de ejecutar el algoritmo de clustering. Las configuraciones se detallan en el cuadro 4.6.

	α	β	λ_{critic}	λ_{actor}	**Featurizer**
TEST #28	0.001	0.005	0.8	0.8	ClusteringFeaturizer (5K)
TEST #29	0.001	0.005	0.8	0.8	ClusteringFeaturizer (10K)

Cuadro 4.6: Tercer conjunto de tests para AC+trazas

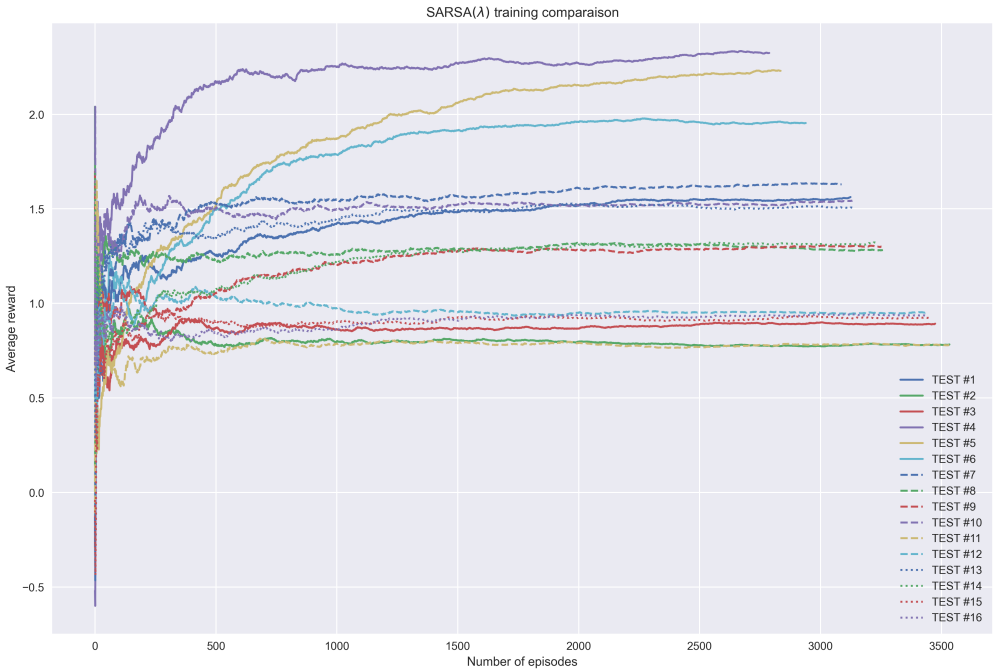

Figura 4.5: Resultados de los entrenamientos usando SARSA(λ)

4.3. Resultados numéricos

Una vez hemos detallado cómo se ha realizado el desarrollo de los algoritmos, pasamos a ver qué resultados hemos podido obtener.

En las figuras 4.5 y 4.6 se pueden ver gráficamente los resultados tras la ejecución de los distintos experimentos.

La métrica que utilizamos es la recompensa media acumulada en cada episodio. Como ya vimos, para cada uno de los episodios tenemos en cuenta las vidas que pierden el agente y el oponente —siendo el peor resultado -5, y el mejor, +5—, así como las etapas que sobrevive —sumando 0.01 por cada etapa—.

Teniendo esto en cuenta, podemos observar que los resultados obtenidos no son lo buenos que esperábamos. Usando actor-critic y trazas de elegibilidad, no llegamos a ver ninguna muestra de aprendizaje, independientemente de la configuración de hiperparámetros elegida.

Por otra parte, usando SARSA(λ) sí que llegamos a ver muestras de que nuestro agente llega a aprender, ya que la forma que tiene la gráfica en alguna de las pruebas sigue una tendencia creciente. Sin embargo, estos resultados son muy limitados, teniendo en cuenta que apenas logramos una recompensa media de 2 en algunas de las simulaciones.

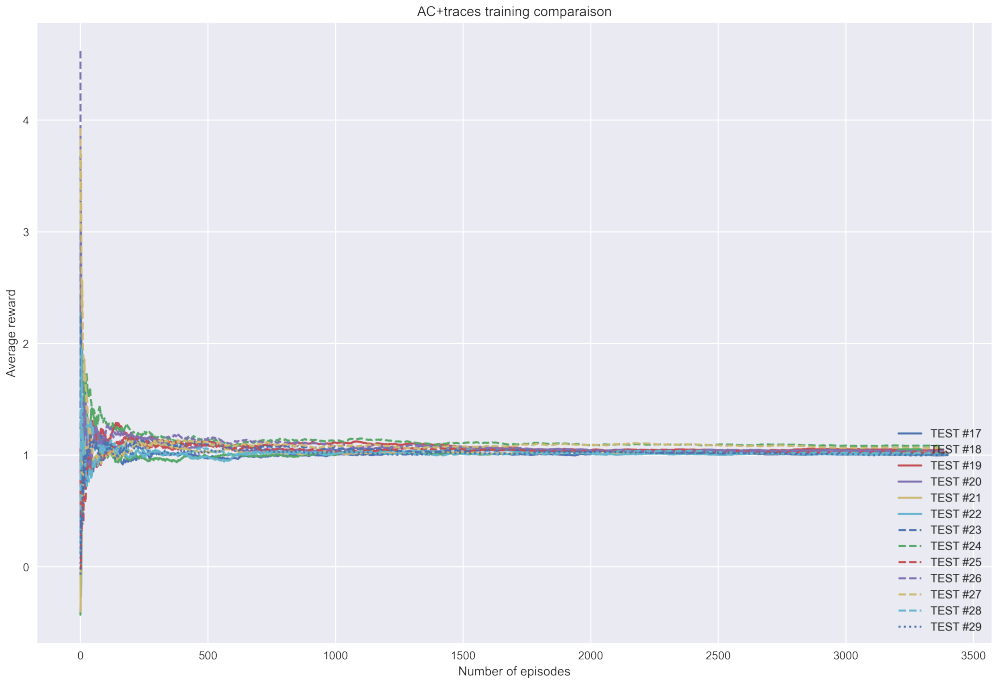

Figura 4.6: Resultados de los entrenamientos usando AC y trazas de elegibilidad

Podemos confirmar el rendimiento haciendo una evaluación del agente con la configuración de hiperparámetros que obtiene los mejores resultados en nuestras pruebas, el TEST #4. Como se muestra en la figura 4.7, tras evaluar nuestro agente durante 1000 episodios, la media ronda en torno a -4.85, resultado similar a lo que podemos conseguir con una política aleatoria.

Debemos mencionar que, a pesar que los resultados no sean los esperados, estas pruebas no confirman del todo que el uso de métodos VFA no sea una alternativa válida. La principal limitación en nuestras pruebas es el tiempo necesario de computación. Dado que la implementación de estos algoritmos se ha realizado directamente en Python, el tiempo para ejecutar cada uno de ellos es demasiado alto, llegando a necesitar varias horas para la ejecución de ciertos tests. Sin ir más lejos, el entrenamiento del test #5 muestra una tendencia claramente creciente, por lo que teniendo un mayor número de recursos podríamos llegar a hacer pruebas más robustas y fiables.

En lo que se refiere a los tiempos de ejecución, la figura 4.8 muestra una comparativa entre las distintas variantes que hemos utilizado en las pruebas de nuestros algoritmos. Los resultados que se muestran son los tiempos de cómputo necesarios para ejecutar 100000 etapas.

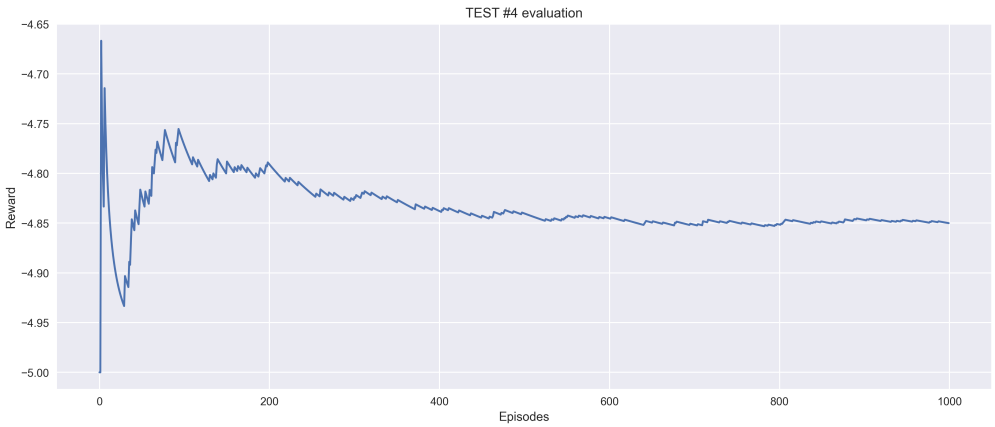

Figura 4.7: Evaluación del TEST #4

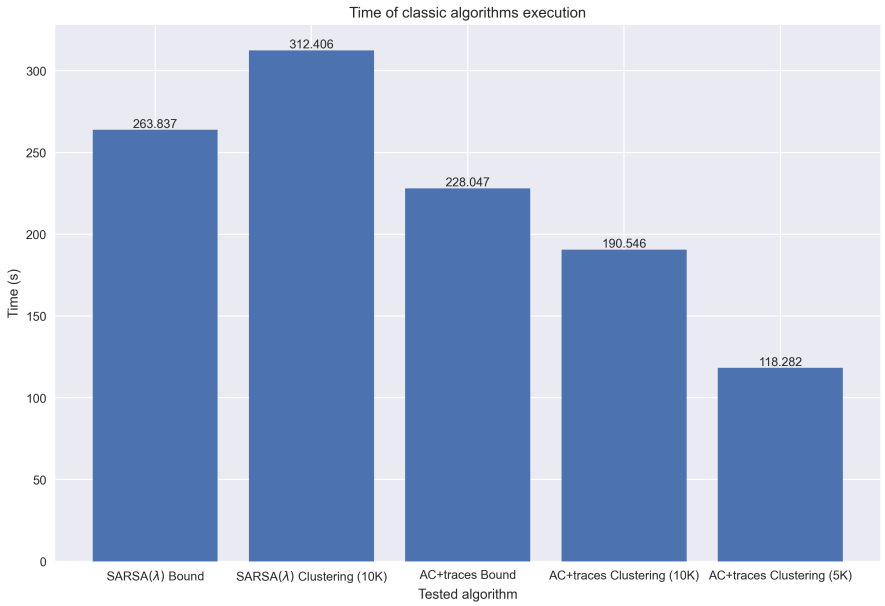

Figura 4.8: Tiempos de cómputo para algoritmos clásicos en 100000 etapas

Capítulo 5

Algoritmos avanzados

Una vez hemos explorado el uso de algoritmos clásicos en nuestro entorno de aprendizaje por refuerzo, pasamos a ver algoritmos que utilizan conceptos más sofisticados dentro de este ámbito. Como veremos, son algoritmos más complejos, que sirven, por tanto, para abordar entornos que presentan gran número de dimensiones en el espacio de acciones y estados.

5.1. Fundamentos teóricos

Los algoritmos que utilizaremos en el proyecto serán PPO, DQN y SAC. En esta sección hacemos un repaso a los fundamentos teóricos de cada uno de ellos.

5.1.1. DQN

En la actualidad, el aprendizaje profundo ha supuesto grandes avances en ámbitos como la visión por computador o el reconocimiento del habla, gracias al uso de todo tipo de redes neuronales dentro del aprendizaje supervisado y no supervisado. Por tanto, tiene sentido pensar en aplicar este tipo de técnicas dentro del aprendizaje por refuerzo [23].

No obstante, dadas las características de RL, existen ciertas dificultades técnicas relacionadas con el aprendizaje profundo. Por ejemplo, la mayoría de aplicaciones que utilizan aprendizaje profundo necesitan grandes cantidades de datos etiquetados, mientras que los algoritmos RL deben aprender mediante una señal de recompensa, que puede ser escasa, ruidosa y con retardo. En aprendizaje supervisado, la interacción entre acción y recompensa —input y output en este caso— es directa. Otro problema surge de le suposición por parte de los algoritmos de aprendizaje por refuerzo de que nuestras muestras de datos son independientes entre ellas, algo que no es lo normal en RL, donde el esta-

do actual del agente estará relacionado con los estados anteriores. Por último, debemos tener en cuenta que en RL la distribución de los datos cambia a medida que el agente aprende, algo que no pasa en aprendizaje supervisado, por ejemplo, donde nuestros datos de entrada son siempre fijos.

Es aquí donde Deep Q Network toma protagonismo, buscando solucionar estas dificultades. Deep Q Network (DQN) se basa en el algoritmo de control model-free off-policy conocido como *Q-learning*. El enfoque que plantea Q-learning es seleccionar las acciones con una política distinta a la que se usa para hacer las actualizaciones de la función Q. Como vemos en su formulación, la política que selecciona las acciones (target) es ϵ-greedy respecto a Q, mientras que la de las acciones sucesoras (prediction) es greedy respecto a Q:

$$Q(a,s) \leftarrow Q(a,s) + \alpha \cdot \left(r_s + \gamma \max_{a'} Q\left(a', s'\right) - Q(a,s) \right) \tag{5.1}$$

La idea básica de este algoritmo es estimar la función Q utilizando la ecuación de Bellman para realizar actualizaciones iterativas. Estos algoritmos de convergen a la función Q óptima cuando el número de iteraciones tiende a infinito, pero se ha demostrado que este enfoque no es práctico, porque la función Q se aprende para cada secuencia de forma separada, sin ninguna generalización.

Por tanto DQN utiliza un aproximador de función para estimar la función Q, tal como hacía SARSA(λ) y describimos en el capítulo 4. La diferencia clave de DQN respecto a estos algoritmos es que, en lugar de utilizar un aproximador lineal, utiliza un aproximador no lineal, basado en redes neuronales. Un aproximador de funciones basado en redes neuronales con pesos θ se conoce como Q-network. Una Q-network puede ser entrenada buscando minimizar una secuencia de funciones de error $L_i(\theta_i)$ que cambia a cada iteración i:

$$L_i\left(\theta_i\right) = \mathbb{E}_{s,a\sim\rho(\cdot)}\left[\left(y_i - Q\left(s,a;\theta_i\right)\right)^2\right] \tag{5.2}$$

siendo

$$y_i = \mathbb{E}_{s'\sim\mathcal{E}}\left[r + \gamma \max_{a'} Q\left(s', a'; \theta_{i-1}\right) \mid s, a \right] \tag{5.3}$$

el target para la iteración i y $\rho(s,a)$ una distribución de probabilidad de secuencias s y acciones a, denominada distribución de comportamiento.

Como vemos, el enfoque es muy parecido al objetivo de predicción que desarrollábamos en el capítulo anterior, donde intentamos minimizar el error cuadrático, en este caso utilizando la función Q en lugar de la función valor, y el vector de pesos θ en vez de w.

Volviendo a la comparación con el aprendizaje supervisado, vemos que en este caso el target depende de los pesos de la red, en lugar de estar fijos antes de empezar el entrenamiento.

Si diferenciamos la función de pérdidas respecto a los pesos, llegamos al siguiente gradiente:

$$\nabla_{\theta_i} L_i(\theta_i) = \mathbb{E}_{s,a\sim\rho(\cdot);s'\sim\mathcal{E}}\left[\left(r + \gamma \max_{a'} Q\left(s', a'; \theta_{i-1}\right) - Q\left(s, a; \theta_i\right)\right)\nabla_{\theta_i} Q\left(s, a; \theta_i\right)\right] \tag{5.4}$$

En lugar de computar las esperanzas que aparecen en la expresión, normalmente es conveniente optimizar dicha función utilizando el descenso estocástico del gradiente, tal como veíamos en los métodos VFA lineales.

Las últimas tendencias dentro de aprendizaje profundo se basan en entrenamientos eficaces utilizando grandes cantidades de datos. Los enfoques que consiguen los mejores resultados realizan su entrenamiento empleando directamente los datos de entrada sin manipular, usando actualizaciones ligeras basadas en descenso estocástico del gradiente. Estos sugieren que aportando suficiente cantidad de información a las redes neuronales profundas, es posible a menudo conseguir mejores representaciones que características elaboradas.

DQN se adapta a este enfoque mediante la técnica conocida como *experience replay*. Esta técnica se basa en almacenar las experiencias del agente en cada etapa, $e_t = (s_t, a_t, r_t, s_{t+1})$, en un dataset $D = e_1 \dots e_N$, acumulados a lo largo de los episodios en lo que se conoce como *replay memory*. Durante el bucle interior del algoritmo, se realizan actualizaciones Q-learning —o minibatch— a muestras de la experiencia $e \sim D$, elegidas de manera aleatoria del conjunto de muestras almacenadas.

Después de realizar *experience replay*, el agente selecciona y ejecuta una acción siguiendo una política ϵ-greedy. Dado que usar trayectorias de longitudes arbitrarias como entrada de una red neuronal puede ser complicado, la función Q trabaja con representaciones de trayectorias de longitud fija dadas por una función ϕ.

DQN presenta varias ventajas respecto al uso de Q-learning. En primer lugar, cada etapa de la simulación se usa potencialmente en muchas actualizaciones de los pesos θ, lo que permite una mayor eficiencia de datos. Después, sabemos que aprender directamente de muestras consecutivas no es eficiente, ya que existe una gran correlación entre muestras, por lo que tomando muestras aleatorias eliminamos estas correlaciones, reduciendo la varianza de las actualizaciones. También, al realizar un aprendizaje on-policy, los parámetros actuales determinan la siguiente muestra de datos con la que se entrenan los parámetros: por ejemplo si en un momento dado la acción que maximiza es hacer un movimiento hacia la izquierda, nuestras muestras estarán más representadas por muestras en el lado izquierdo. Igual pasaría si en cierto momento son los movimientos a la derecha los que maximizan la recompensa. Esto puede llevar a bucles inesperados o parámetros estancados en mínimos locales que no son lo suficientemente buenos, incluso divergencias en la optimización. Usar experience replay, aprendiendo off-policy suaviza el aprendizaje

y evita oscilaciones y divergencias en nuestros parámetros.

En lo que se refiere a la arquitectura del modelo de la red neuronal, originalmente DQN fue diseñado para trabajar con juegos Atari, tomando una representación de píxeles. Concretamente, la idea es trabajar con una región de 84x84 píxeles en RGB partiendo de una escala original de 210x160 píxeles y una paleta de 128 colores. A partir de esta entrada, se desarrolla una red neuronal convolucional de un determinado número de capas.

Sin embargo, dado que en nuestro entorno no trabajamos con los píxeles de la imagen directamente, stable-baselines3 ofrece una política alternativa a la que usa CNN. Esta política recibe el nombre de MlpPolicy —MLP, Multilayer Perceptron, alias de DQNPolicy—, y utiliza dos capas completamente conectadas de 64 nodos cada una [24]. Esta será la política que utilizaremos para nuestro entorno, como veremos en detalle en la sección 5.2.

5.1.2. PPO

Proximal Policy Optimization (PPO) es un algoritmo del tipo *policy gradient* que surge motivado por la siguiente pregunta: ¿cómo podemos obtener el mayor salto de mejora posible en una política, utilizando los datos que tenemos actualmente, sin tomar un salto que haga que el rendimiento colapse? Conseguir buenos resultados con estos métodos no siempre es sencillo, ya que son sensibles a la elección del step-size: si es muy bajo, el progreso es demasiado lento, mientras que si es demasiado alto, podemos encontrarnos con caídas catastróficas del rendimiento [25].

PPO es una familia de métodos de primer orden que utilizan una serie de recursos para mantener las nuevas políticas cerca de las antiguas. Ser un algoritmo de primer orden significa que utiliza únicamente información del gradiente para construir la siguiente iteración del entrenamiento. Se alterna entre el muestreo de información mediante interacción con el entorno y optimización de una función objetivo "sustituta" utilizando el ascenso de gradiente estocástico.

Mientras que típicamente los métodos policy gradient realizan una actualización del gradiente por cada muestra, la función objetivo que se utiliza en PPO permite la agrupación de actualizaciones en minilotes.

Este algoritmo presenta distintas ventajas respecto a sus rivales. Si lo comparamos con DQN, descrito en el apartado anterior, sabemos que DQN funciona bien con entornos Atari con espacio de acciones discretas, pero no está concebido para problemas de control continuo. Respecto a TRPO (Trust Region Policy Optimization), aunque no se describe en este trabajo, se trata de un algoritmo de segundo orden, y por tanto más complejo que PPO, que además es compatible con arquitecturas que introducen ruido

o que comparten parámetros —entre política y función valor, o con otras tareas—. Por último, otros métodos policy gradient presentan una eficiencia de datos y robustez menor [26].

PPO presenta dos principales variantes: *PPO-Penalty* y *PPO-Clip*. En este trabajo detallaremos la versión *Clip*, en la cual se basa la implementación que utilizamos en nuestro desarrollo. PPO-Clip no utiliza el término de la divergencia KL en la función objetivo y no tiene ninguna restricción. En su lugar, hace uso de la técnica de clipping —en español, truncamiento— en la función objetivo para evitar que la nueva política se aleje demasiado de la antigua.

Debemos mencionar que la implementación que utilizamos (stable-baselines3) presenta varias modificaciones respecto al algoritmo original que no están documentadas por OpenAI: las ventajas se normalizan y también se puede aplicar el clip a la función valor [27].

Como ya sabemos, al tratarse de un algoritmo policy gradient, tenemos una política θ que se actualiza a lo largo del tiempo. En este caso concreto, PPO-clip actualiza la política según la siguiente expresión:

$$\theta_{k+1} = \arg\max_{\theta} \mathrm{E}_{s,a\sim\pi_{\theta_k}} \left[L\left(s, a, \theta_k, \theta\right) \right] \tag{5.5}$$

donde se toman múltiples etapas —normalmente minibatch— del descenso estocástico del gradiente para maximizar el objetivo. L viene dada por la siguiente expresión:

$$L\left(s, a, \theta_k, \theta\right) = \min\left(\frac{\pi_\theta(a \mid s)}{\pi_{\theta_k}(a \mid s)} A^{\pi_{\theta_k}}(s,a), \quad \operatorname{clip}\left(\frac{\pi_\theta(a \mid s)}{\pi_{\theta_k}(a \mid s)}, 1-\epsilon, 1+\epsilon \right) A^{\pi_{\theta_k}}(s,a) \right) \tag{5.6}$$

donde ϵ es un hiperparámetro que indica cómo de lejos podrá estar la nueva política respecto a la antigua. Dado que se trata de una expresión compleja, podemos utilizar una simplificación matemática, en la que no entraremos en detalle [28].

La siguiente expresión resulta más fácil de interpretar y es la que implementa en OpenAI:

$$L\left(s, a, \theta_k, \theta\right) = \min\left(\frac{\pi_\theta(a \mid s)}{\pi_{\theta_k}(a \mid s)} A^{\pi_{\theta_k}}(s,a), g\left(\epsilon, A^{\pi_{\theta_k}}(s,a)\right) \right) \tag{5.7}$$

donde

$$g(\epsilon, A) = \begin{cases} (1+\epsilon)A & A \geq 0 \\ (1-\epsilon)A & A < 0 \end{cases} \tag{5.8}$$

siendo A el estimador de la función de ventaja, que describe cómo de mejor o de peor es la acción a en comparación con las demás acciones en el estado s (siguiendo la política π):

$$\mathcal{A}_\pi(s,a) = q_\pi(s,a) - v_\pi(s) \tag{5.9}$$

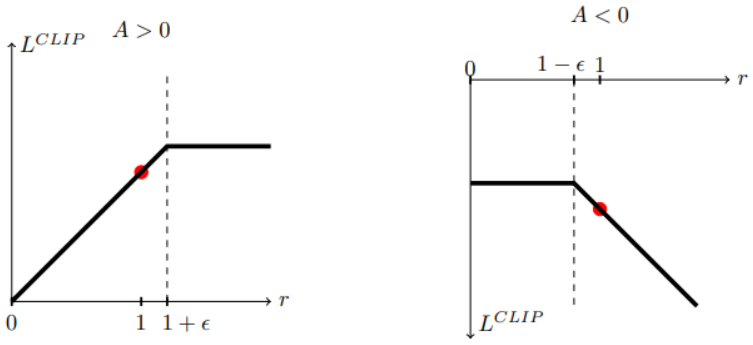

Figura 5.1: PPO-clipping gráficamente [26]

Por tanto, asumiendo un determinado par (s, a), si la función de ventaja es positiva —la acción elegida ha sido buena—, el objetivo aumentará si la acción se vuelve más probable ($\pi_\theta(a|s)$ aumenta). Pero el mínimo en este término indica el límite de cuánto puede aumentar el objetivo.

Cuando $\pi_\theta(a \mid s) > (1+\epsilon)\pi_{\theta_k}(a \mid s)$, el mínimo hace efecto y este término tiene un valor máximo de $(1+\epsilon)A^{\pi_{\theta_k}}(s, a)$. Por tanto la nueva política no se beneficia de alejarse de la política antigua.

Por el otro lado, si la función de ventaja es negativa —la acción elegida ha sido mala—, el objetivo aumentará si la acción se vuelve menos probable ($\pi_\theta(a|s)$ disminuye), teniendo un máximo que limita cuánto el objetivo puede aumentar.

Cuando $\pi_\theta(a \mid s) < (1-\epsilon)\pi_{\theta_k}(a \mid s)$, el máximo hace efecto y este término tiene un valor máximo de $(1-\epsilon)A^{\pi_{\theta_k}}(s, a)$. Por tanto, de nuevo, la nueva política no se beneficia de alejarse de la política antigua.

Estos casos se muestran gráficamente en la figura 5.1, que ilustra la función sustituta L^{CLIP} en una etapa determinada como una función del ratio de probabilidad r para las ventajas positivas y negativas.

La técnica de clipping sirve para regularizar eliminando incentivos a la política de cambiar de forma dramática, y el hiperparámetro ϵ indica cuánto puede alejarse la nueva política de la antigua, mientras que todavía se beneficia del objetivo.

5.1.3. SAC

Soft Actor Critic (SAC) es un algoritmo que optimiza una política estocástica de manera off-policy, utilizando un enfoque a medio camino entre la optimización estocástica de la política y la aproximación que utiliza DDPG [29].

Aunque no entra en el alcance de este trabajo, DDPG (Deep Deterministic Policy

Gradient) es un algoritmo que aprende de manera concurrente una función Q y una política. Utiliza información off-policy y la ecuación de Bellman para aprender la función Q, y emplea esta función Q para aprender la política. Este enfoque está estrechamente relacionado con Q-learning, el cual vimos en el apartado 5.1.1 [30].

Un concepto clave dentro de SAC es lo que se conoce como *regularización de la entropía*. La política se entrena para maximizar una suma ponderada del retorno esperado y entropía, siendo la entropía una medida de aleatoreidad en la política. Esto está relacionado con el compromiso de exploración y explotación, propio de los algoritmos de RL. Incrementar los resultados de entropía conlleva mayor exploración, lo que puede acelerar el aprendizaje en etapas posteriores. También puede conseguir evitar que la política converja de manera prematura en un óptimo local malo.

Por tanto, para poder entender SAC, es necesario introducir el concepto de *aprendizaje por refuerzo regularizado por entropía*.

Como hemos comentado, la entropía es la magnitud que indica cómo de aleatoria es una variable. Si trucamos una moneda para que siempre salga cara, su entropía será baja. Si la moneda está equilibrada y la probabilidad de cara o cruz es la misma, la entropía será alta.

Si x es una variable aleatoria con una función de densidad P, la entropía H de x se calcula a partir de su distribución P según la siguiente expresión:

$$H(P) = \mathrm{E}_{x\sim P}[-\log P(x)] \tag{5.10}$$

En RL regularizado por entropía, en cada etapa, el agente consigue una recompensa adicional proporcional a la entropía de la política en dicha etapa. Por tanto, la política óptima que buscamos ahora es la siguiente:

$$\pi^* = \arg\max_{\pi} \mathrm{E}_{\tau\sim\pi}\left[\sum_{t=0}^{\infty} \gamma^t \left(R\left(s_t, a_t, s_{t+1}\right) + \alpha H\left(\pi\left(\cdot \mid s_t\right)\right)\right)\right] \tag{5.11}$$

donde α es el coeficiente de trade-off —indica la importancia relativa que le damos a la entropía: mayor α se corresponde con una mayor exploración y menor α conlleva mayor explotación—.

SAC aprende de forma concurrente una política π_θ y dos funciones, Q_{ϕ_1} y Q_{ϕ_2}. Existen dos variantes de este algoritmo. La primera usa un coeficiente fijo de regularización α, mientras que la otra utiliza un coeficiente variable a lo largo del entrenamiento. Para explicar este algoritmo asumiremos α fijo, aunque a veces es preferible que este parámetro sea variable.

Respecto al aprendizaje de Q, entre sus características, podemos destacar que las

funciones Q se aprenden usando el error cuadrático medio de Bellman (MSBE), que nos indica cómo de cerca está nuestra función Q de satisfacer la ecuación de Bellman. También, como se muestra en la ecuación 5.12, el target incluye un término que viene del uso de la regularización de entropía, y las acciones del estado siguiente usadas en el objetivo vienen de la política actual, en vez de venir de la política objetivo.

Es importante saber cómo se integra la contribución de la regularización de entropía. Partiendo de la ecuación recursiva de Bellman para la regularización de entropía:

$$\begin{aligned} Q^{\pi}(s,a) &= \underset{\substack{s'\sim P\\ a'\sim\pi}}{\mathrm{E}}\left[R\left(s,a,s'\right)+\gamma\left(Q^{\pi}\left(s',a'\right)+\alpha H\left(\pi\left(\cdot \mid s'\right)\right)\right)\right] \\ &= \underset{\substack{s'\sim P\\ a'\sim\pi}}{\mathrm{E}}\left[R\left(s,a,s'\right)+\gamma\left(Q^{\pi}\left(s',a'\right)-\alpha\log\pi\left(a' \mid s'\right)\right)\right] \end{aligned} \tag{5.12}$$

La parte derecha de la ecuación es una esperanza de los estados sucesivos —que vendrán dados por el *replay buffer*— y las acciones sucesivas —que vendrán dadas por la política actual—. Dado que es una esperanza, lo podemos aproximar con muestras:

$$Q^{\pi}(s,a) \approx r+\gamma\left(Q^{\pi}\left(s',\tilde{a}'\right)-\alpha\log\pi\left(\tilde{a}' \mid s'\right)\right), \quad \tilde{a}'\sim\pi\left(\cdot \mid s'\right) \tag{5.13}$$

SAC fija el error MSBE para cada función Q utilizando este tipo de aproximación para el target. La única incógnita que queda sin resolver es qué función Q se utiliza para computar el backup de muestras. Utiliza el truco *clipped double-Q*, tomando el mínimo valor entre los dos aproximadores Q.

Teniendo todo esto en cuenta, la función de pérdidas para las Q-network es la siguiente:

$$L\left(\phi_i,\mathcal{D}\right)=\underset{(s,a,r,s',d)\sim\mathcal{D}}{\mathrm{E}}\left[\left(Q_{\phi_i}(s,a)-y\left(r,s',d\right)\right)^2\right] \tag{5.14}$$

donde el target $y(r,s',d)$ viene dado por la siguiente expresión:

$$y\left(r,s',d\right)=r+\gamma(1-d)\left(\min_{j=1,2}Q_{\phi_{\mathrm{targ},j}}\left(s',\tilde{a}'\right)-\alpha\log\pi_{\theta}\left(\tilde{a}' \mid s'\right)\right), \quad \tilde{a}'\sim\pi_{\theta}\left(\cdot \mid s'\right) \tag{5.15}$$

Respecto a la política, para cada estado, actúa para maximizar el retorno futuro esperado junto con la entropía futura esperada. Analíticamente, la expresión a actualizar es $V_{\pi}(s)$:

$$\begin{aligned} V^{\pi}(s) &= \underset{a\sim\pi}{\mathrm{E}}\left[Q^{\pi}(s,a)\right] + \alpha H(\pi(\cdot \mid s)) \\ &= \underset{a\sim\pi}{\mathrm{E}}\left[Q^{\pi}(s,a) - \alpha \log \pi(a \mid s)\right] \end{aligned} \tag{5.16}$$

Para realizar esta optimización, hacemos uso del que se conoce como *reparameterization trick*. Mediante esta técnica, una muestra de $\pi_\theta(\cdot \mid s)$ se aproxima calculando una función determinista de estado, parámetros de la política y ruido independiente.

El reparameterization trick nos permite reescribir la esperanza sobre las acciones —que contiene un punto débil, ya que la distribución depende de los parámetros de la política— en una esperanza sobre ruido —que elimina este punto débil, ya que la distribución ya no depende de parámetros—:

$$\underset{a\sim\pi_\theta}{\mathrm{E}}\left[Q^{\pi_\theta}(s,a) - \alpha \log \pi_\theta(a \mid s)\right] = \underset{\xi\sim\mathcal{N}}{\mathrm{E}}\left[Q^{\pi_\theta}\left(s, \tilde{a}_\theta(s,\xi)\right) - \alpha \log \pi_\theta\left(\tilde{a}_\theta(s,\xi) \mid s\right)\right] \tag{5.17}$$

Para obtener la pérdida de la política, el último paso es sustituir Q^{π_θ} por uno de nuestros aproximadores de función:

$$\underset{\theta}{\mathrm{máx}} \underset{\substack{s\sim\mathcal{D}\\ \xi\sim\mathcal{N}}}{\mathrm{E}}\left[\underset{j=1,2}{\mathrm{mín}} Q_{\phi_j}\left(s, \tilde{a}_\theta(s,\xi)\right) - \alpha \log \pi_\theta\left(\tilde{a}_\theta(s,\xi) \mid s\right)\right] \tag{5.18}$$

5.2. Metodología

En lo que se refiere al desarrollo de algoritmos avanzados, en primer lugar, implementaremos y probaremos los 3 algoritmos que hemos descrito en la sección anterior en el entorno SlimeVolleyGym. Concretamente, será el agente quien será entrenado con estos algoritmos mientras se enfrenta al oponente. Este oponente utiliza lo que se conoce en el entorno como la política *baseline*. Esta política se ha conseguido mediante el uso de una pequeña red neuronal de 120 parámetros, ya entrenada, y que sirve como referencia para nuestros primeros experimentos [31].

Como hemos comentado, estos algoritmos ya vienen implementados por stable-baselines3, pero en este capítulo exploraremos distintas técnicas que empleamos para hacer que nuestro entrenamiento sea efectivo, según las opciones que nos ofrece gym y la propia librería. Además, utilizaremos la evaluación de los modelos para así poder comparar el rendimiento de cada uno de ellos.

Una de las principales ventajas de estos algoritmos, es su rendimiento. Al estar implementados en PyTorch, las velocidades que podemos llegar a conseguir son muy superiores a las que se utilizan empleando las librerías estándar de Python.

5.2.1. Fases del desarrollo

A lo largo de esta sección, para cada uno de los algoritmos tendremos en cuenta dos partes: entrenamiento y evaluación.

Para el entrenamiento, es necesario instanciar el modelo que deseamos con el constructor que proporciona stable-baselines3 y ejecutar su método *learn*. Esto lo veremos a continuación.

La evaluación la haremos de forma manual. Para ello desarrollamos una función que toma el entorno en el que queremos realizar la evaluación, la política del modelo entrenado y el número de episodios que queremos ejecutar. Esta función nos devuelve una array de recompensas y longitudes de la evaluación en cada uno de los episodios. También definimos una función que nos permite guardar los resultados en un .csv para manipular estos datos a posteriori. Su código es el siguiente:

```
def evaluate_algorithm(env, policy, episodes):
    rewards = np.zeros(episodes)
    lengths = np.zeros(episodes)
    ep = 0
    for i in range(episodes):
        S = env.reset()
        G = 0
        t = 0
        ep += 1
        done = False
        while not done:
            A, _states = policy.predict(S, deterministic=True)
            S, R, done, _ = env.step(A)
            G += R
            t += 1
        rewards[i] = G
        lengths[i] = t
    return rewards, lengths

def save_results(rewards, lengths, file_name):
    training_data = np.vstack((rewards, lengths)).T
    np.savetxt(join(LOGDIR, file_name), training_data, delimiter=',',
               header='eval_rewards,eval_lengths', comments='')
```

5.2.2. Pasos previos al aprendizaje

Antes de explorar el uso y el rendimiento de los distintos algoritmos, queremos ver cuál es el efecto de dos variantes que se presentan a la hora de realizar el aprendizaje.

La primera de ellas ya la vimos en el desarrollo de algoritmos clásicos. Se trata del uso de los wrappers *SurvivalRewardEnv* y *ReducedDimension* ya descritos. Aunque nuestra intuición ya nos decía que usar recompensas más progresivas y unas dimensiones más ajustadas a lo relevante era beneficioso para el entrenamiento, aprovechamos el uso de estos algoritmos (cuya ejecución es mucho más rápida) para demostrarlo empíricamente.

La otra variante que se presenta es el uso del callback *EvalCallback*. Un *callback* en stable-baselines3 es un conjunto de funciones que se llaman en un momento concreto del aprendizaje [32]. Sus aplicaciones son muy amplias, pero en nuestro caso emplearemos EvalCallback para realizar una evaluación de de nuestro modelo durante un número de episodios determinado en el intervalo de etapas que deseemos. Este callback también nos permite la opción de ir guardando el mejor modelo hasta el momento.

Esta última opción toma un tiempo de cómputo considerable —en nuestro caso realizamos evaluaciones de 100 episodios cada 100 000 etapas—, por lo que no es algo despreciable, tal como se observa en el apartado de resultados, en la figura 5.7. Por tanto, queremos asegurarnos que su uso suponga una mejora del entrenamiento significativa.

Estas pruebas las realizaremos utilizando el algoritmo PPO, ya que sabemos que puede llegar a converger, según los resultados de las pruebas del autor del entorno.

Teniendo estas dos variantes en cuenta, fijamos el siguiente marco de pruebas:

- Dos entornos SlimeVolley-v0:

  1. Entorno que no usa ningún wrapper además de Monitor.

     ```
     slime_env = Monitor(env=gym.make('SlimeVolley-v0'),
                         filename=join(LOGDIR, 'ppo3m'))
     ppo_model = PPO('MlpPolicy', slime_env, verbose=0)
     ppo_model.learn(TIMESTEPS, callback=eval_callback);
     ```

  2. Entorno que usa SurvivalRewardEnv y ReducedDimension además de Monitor.

     ```
     slime_env = Monitor(env=SurvivalRewardEnv(ReducedDimension(
     ↪  gym.make('SlimeVolley-v0'))),
                         filename=join(LOGDIR, 'ppo3m_wra'))
     ppo_model = PPO('MlpPolicy', slime_env, verbose=0)
     ppo_model.learn(TIMESTEPS, callback=eval_callback);
     ```

- El aprendizaje en ambos entornos utiliza el siguiente EvalCallback:

  ```
  eval_callback = EvalCallback(slime_env,
                               best_model_save_path=join(MODELDIR,
                               ↪  'PPO'),
                               eval_freq=100_000,
                               n_eval_episodes=100)
  ```

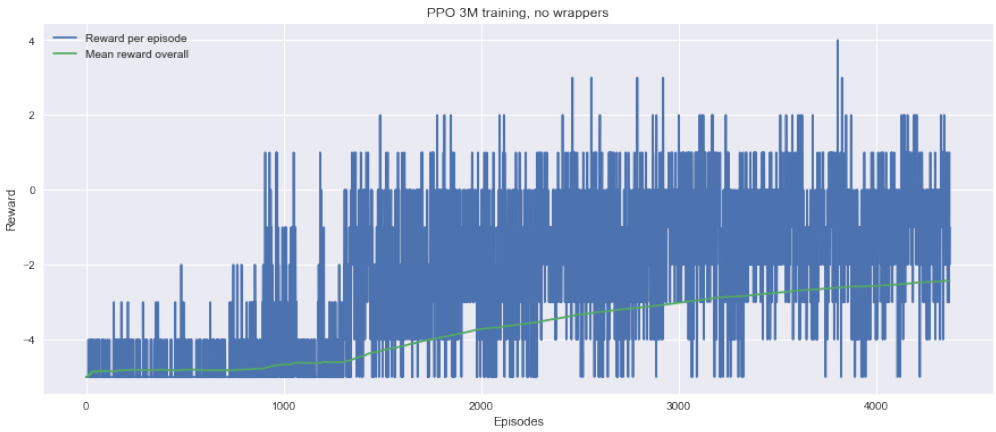

Figura 5.2: Entrenamiento PPO sin wrappers

- El número de etapas en nuestras variantes será de 3000000 etapas.

Además del entrenamiento, queremos realizar una evaluación de estos dos entornos que utilizan (o no) los wrappers mencionados, tal como comentamos. Esta evaluación se realizará para 1000 episodios. Esta evaluación deberíamos realizarla de la misma manera para cada una de nuestras variantes. Para ello, ejecutamos el siguiente código, utilizando las funciones definidas en el código de la sección 5.2.1:

```
ppo_policy = PPO.load(join(MODELDIR, 'PPO', 'PPO_3M'), env=slime_env_gym)
eval_rew, eval_len = evaluate_algorithm(slime_env_gym, ppo_policy, 1000)
save_results(eval_rew, eval_len, 'ppo3m_eval.csv')
```

Dado que los resultados de estos experimentos nos muestran cómo realizar el resto de pruebas, discutiremos estos resultados en esta misma sección.

En las figuras 5.2 y 5.3 podemos ver la diferencia de entrenamiento entre el entorno que no utiliza los wrappers descritos y el que sí lo hace.

Aunque las magnitudes de las recompensas no son representativas —al usar SurvivalRewardEnv también recompensamos la supervivencia en el episodio, por lo que las recompensas en el entorno con este wrapper siempre serán mayores—, sí que se observa una mejora en el proceso de aprendizaje por parte del entorno que emplea wrappers, observando una menor varianza y un crecimiento más exponencial en su curva.

Por otro lado, la gráfica que aparece en la figura 5.4 muestra la evaluación a lo largo de 1000 episodios de los entornos ya entrenados, así como los seleccionados por el EvalCallback como “mejores modelos”.

Observando estas imágenes podemos llegar a dos conclusiones:

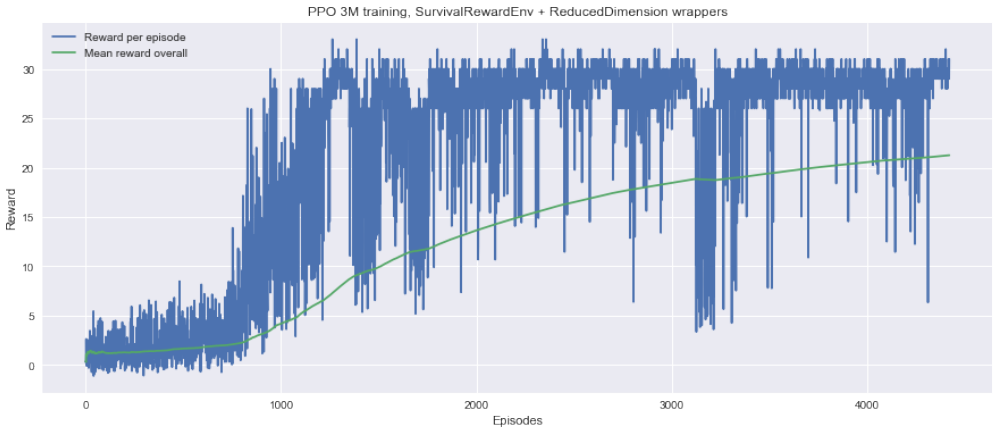

Figura 5.3: Entrenamiento PPO con wrappers

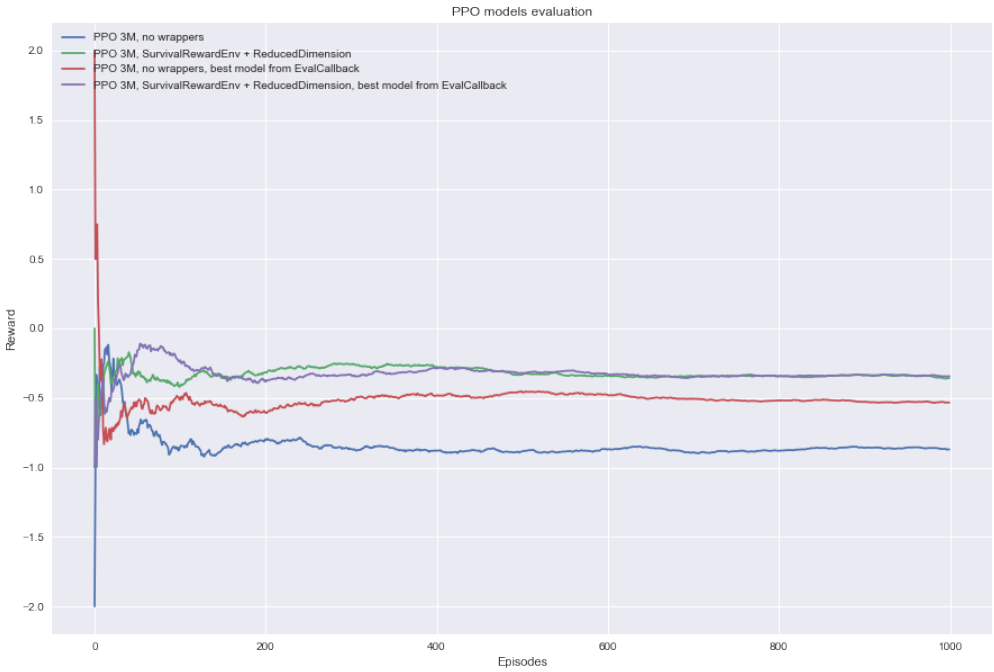

Figura 5.4: Evaluación de modelos PPO

- Los modelos que utilizan los wrappers *SurvivalRewardEnv* y *ReducedDimension* obtienen mejores resultados.
- Los mejores modelos resultantes del uso de EvalCallback producen mejores resultados en general, pero esta mejoría no es tan significativa para compensar el gasto computacional que suponen.

Por tanto, viendo estos resultados intermedios, para pruebas sucesivas utilizaremos los wrappers *SurvivalRewardEnv* y *ReducedDimension* sin utilizar el callback *EvalCallback*.

Llegados a este punto, un punto importante que discutir es cuál va a ser la manera de comparar entre algoritmos. Hay dos alternativas:

- Fijar un número de etapas máximo y comparar los algoritmos en base al rendimiento que han podido llegar al ser entrenados durante ese tiempo.
- Fijar un umbral de recompensa al que deben llegar y comparar los algoritmos en base a las etapas que necesitan para llegar a este umbral.

Nos decidimos por la primera opción, ya que definir este umbral en un entorno con este modelo de recompensas no es algo trivial. Además, el número de etapas que necesitan los algoritmos para llegar a un umbral dado puede variar según el tipo de algoritmo.

Decidimos fijar este número de etapas máximo a 3000000, excepto para el caso de SAC, que este número lo fijamos en 1000000, debido a que la ejecución de este algoritmo consume muchos más recursos.

5.2.3. Proceso de entrenamiento y evaluación

La forma de entrenar y evaluar el resto de algoritmos es similar a lo que ya hemos explicado para PPO. En esta sección profundizamos en algunos detalles más concretos en DQN y SAC.

Como hemos podido ver antes y vemos en el código siguiente, a la hora de realizar el aprendizaje utilizamos un wrapper adicional, que no hemos comentado, llamado *Monitor*.

```
slime_env =
↪  Monitor(env=SurvivalRewardEnv(ReducedDimension(gym.make('SlimeVolley-v0'))),
                    filename=join(LOGDIR, 'dqn3m'))
dqn_model = DQN('MlpPolicy', slime_env, verbose=1)
dqn_model.learn(TIMESTEPS);
```

El wrapper *Monitor* sirve para poder generar un archivo .csv que contiene información del entrenamiento: por cada episodio, la recompensa, el número de etapas y una

marca temporal. Vemos un ejemplo de las primeras 5 filas del monitor resultante del entrenamiento de uno de los algoritmos:

```
r,l,t
2.05,705,0.212
4.1,910,0.392052
0.18,518,0.483047
2.72,672,0.599999
0.08,508,0.712029
```

Esto nos resulta de gran utilidad, ya que de esta manera se puede acceder de forma fácil a esta información, que será valiosa a la hora de hacer análisis y comparar algoritmos más adelante.

Una vez el entrenamiento termina, guardamos el modelo en el directorio destinado para ello. Guardar el modelo nos permite poder cargarlo a posteriori y poder evaluarlo en cualquier momento. Esta operación se realiza mediante el método *save* que ofrece stable-baselines3. Además, mostramos la información de cómo han ido progresando las recompensas a lo largo del entrenamiento. En la misma gráfica mostramos las recompensas inmediatas de cada episodio, así como la media acumulada a lo largo del entrenamiento. El código utilizado es el siguiente:

```
def plot_sb_results(filename, title):
    metrics = pd.read_csv(join(LOGDIR, filename), skiprows=1)
    episode_rew = metrics.loc[:, 'r']
    episode_rew_average = episode_rew.expanding().mean()
    fig, ax = plt.subplots(figsize=(15, 6))
    ax.plot(episode_rew, label='Reward per episode')
    ax.plot(episode_rew_average, label='Mean reward overall')
    ax.set_xlabel('Episodes')
    ax.set_ylabel('Reward')
    ax.set_title(title)
    ax.legend(loc='upper left');

dqn_model.save(join(MODELDIR, 'DQN', 'DQN_3M'))
plot_sb_results('dqn3m.monitor.csv')
```

La evaluación se realiza de la misma manera que hemos explicado en el caso de PPO: evaluamos en 1000 episodios y guardamos los datos resultantes de la evaluación. Utilizando esta información, podemos dibujar la gráfica que muestra cómo se han mantenido las recompensas a lo largo de los 1000 episodios de evaluación:

```
dqn_eval = pd.read_csv(join(LOGDIR, 'dqn3m_eval.csv')).loc[:,
↪  'eval_rewards'].expanding().mean()
```

```
fig, ax = plt.subplots(figsize=(15, 10))
ax.plot(dqn_eval)
```

Respecto a SAC, el proceso es muy similar, teniendo únicamente en cuenta dos diferencias.

SAC es un algoritmo que está diseñado para un espacio de acciones continuo, lo que se conoce en gym como tipo *Box*. Como comentamos al principio de la memoria, nuestro entorno por defecto utiliza acciones *Discrete* (o *MultiBinary*), por lo que necesitamos un wrapper de tipo *ActionWrapper* que nos permita transformar nuestras acciones discretas a continuas.

Definimos nuestro nuevo espacio continuo de dos dimensiones, donde la primera dimensión es de tipo *activación* y toma un valor entre 0 y 1 —NOOP si es menor que 0.5, movimiento en dirección marcada por la segunda dimensión en caso contrario— y la segunda dimensión toma un valor en el intervalo [1, 6) e indica la dimensión del movimiento que tomar —LEFT, UPLEFT, UP, UPRIGHT, RIGHT—. Su implementación es la siguiente:

```
class DiscreteToBoxWrapper(gym.ActionWrapper):
    def __init__(self, env, new_action_set):
        super().__init__(env)
        self.new_action_set = new_action_set
        self.action_space = gym.spaces.Box(low=np.array([0, 1]),
        ↪  high=np.array([1, 5.999]))
    def action(self, box_act):
        if box_act[0] < 0.5:
            action = self.new_action_set[0]
        else:
            action = self.new_action_set[int(np.floor(box_act[1]))]
        return action

action_table = [[0, 0, 0], # NOOP
                [1, 0, 0], # LEFT (forward)
                [1, 0, 1], # UPLEFT (forward jump)
                [0, 0, 1], # UP (jump)
                [0, 1, 1], # UPRIGHT (backward jump)
                [0, 1, 0]] # RIGHT (backward)
slime_env =
↪  Monitor(env=SurvivalRewardEnv(DiscreteToBoxWrapper(ReducedDimension(
↪  gym.make('SlimeVolley-v0')), action_table)),
                    filename=join(LOGDIR, 'sac1m'))
```

La otra diferencia a tener en cuenta es que SAC, como mencionamos, es un algoritmo

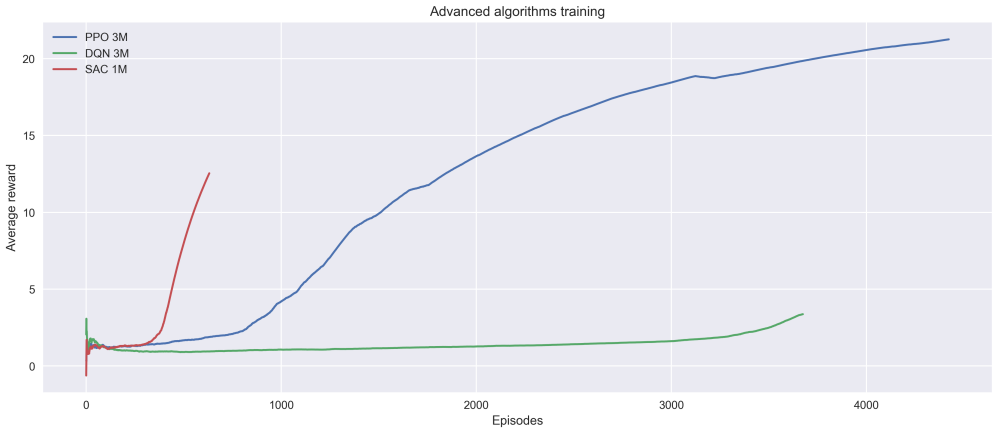

Figura 5.5: Entrenamiento de algoritmos avanzados

que consume muchos recursos en comparación con el resto de algoritmos que probamos —con nuestros recursos, 1 millón de etapas requiere alrededor de 8 horas de cómputo—. Por tanto, fijamos el número de etapas de entrenamiento a 1000000.

5.3. Resultados numéricos

Habiendo visto el desarrollo de los algoritmos avanzados, pasamos a analizar los resultados obtenidos tras nuestros experimentos.

En primer lugar, en la figura 5.5 podemos ver el entrenamiento de los 3 algoritmos descritos. Se puede ver claramente la diferencia entre ellos.

El que peor rendimiento ofrece es DQN, dado que necesita un número muy alto de episodios para comenzar a mostrar aprendizaje por parte del agente. El caso contrario ocurre con SAC. A pesar de ser un entrenamiento corto en comparación —la tercera parte en comparación a PPO y DQN—, se puede observar un crecimiento muy rápido de las recompensas obtenidas alrededor de los 500 episodios. PPO se encuentra en un punto medio, ya que nos ofrece buenos resultados, pero su eficiencia es menor que SAC.

Por otro lado, la figura 5.6 muestra los resultados tras la evaluación en 1000 episodios de nuestros agentes con los distintos algoritmos. Las gráficas demuestran los resultados obtenidos en el entrenamiento. El mejor resultado lo ofrece SAC, que con un entrenamiento de únicamente 1 millón de etapas, consigue una puntuación media de 0 al enfrentarse al agente con la política baseline. Esto significaría un empate contra el que consideramos *campeón*, lo cual es un resultado muy bueno, teniendo en cuenta también los resultados descritos en [7]. En el caso de SAC concretamente, podríamos llegar a vencer al agente ya entrenado con un número relativamente pequeño de etapas.

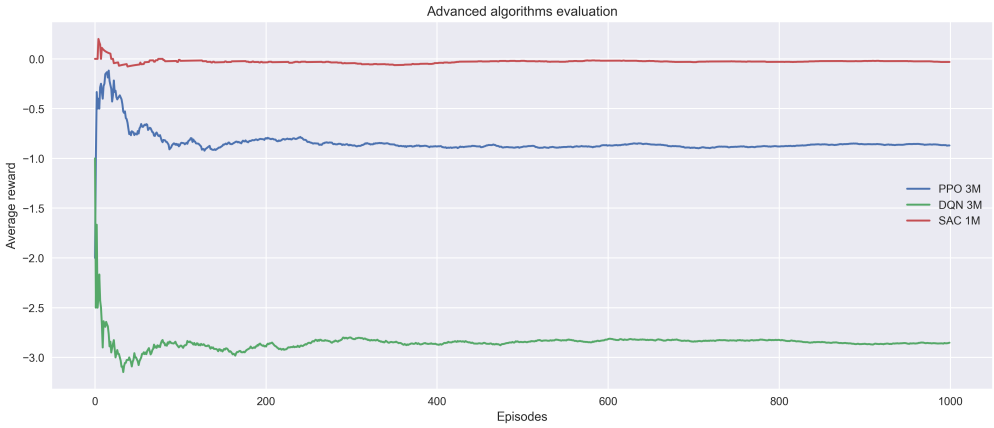

Figura 5.6: Evaluación de algoritmos avanzados

Aunque en [7], las recompensas en las evaluaciones que se describen llegan a valores de 1.377 ± 1.133 para PPO, y 1.148 ± 1.071 para CMA-ES, las recompensas obtenidas en nuestros experimentos están cerca de estos resultados, teniendo en cuenta que no estamos realizando múltiples entrenamientos con distintas semillas ni cambiando las configuraciones de hiperparámetros.

Respecto a los tiempos de ejecución, la figura 5.7 muestra una comparativa, al igual que en el capítulo anterior, entre los algoritmos empleados y algunas de sus variantes. Las ejecuciones se han realizado para 100000 etapas, excepto para el caso de SAC, que empleamos 10000 etapas debido a su mayor tiempo de cómputo. No obstante, para poder comparar de forma justa, el valor que muestra la figura 5.7 para SAC ha sido multiplicado por 10, mostrando el valor que tomaría la ejecución de 100000 etapas, al igual que el resto de algoritmos.

Salvo este último caso, se pueden observar velocidades de ejecución significativamente más altas que los algoritmos que utilizan aproximación lineal de la función valor.

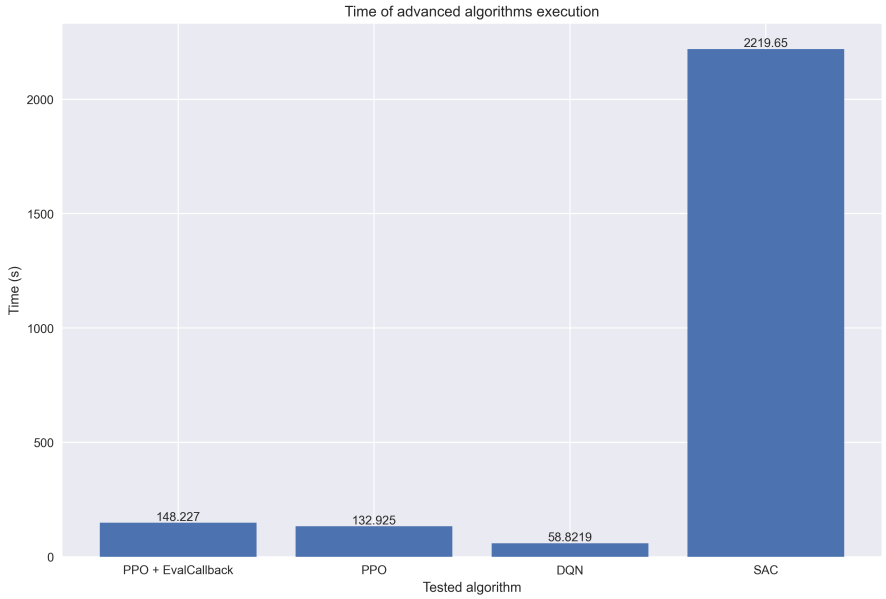

Figura 5.7: Tiempos de cómputo para algoritmos avanzados en 100000 etapas

Capítulo 6

Técnicas self-play

En este capítulo, tomaremos un enfoque distinto al anterior. Inspirado por los experimentos aportados en el repositorio de SlimeVolleyGym [7], utilizaremos una técnica distinta para entrenar nuestros agentes. En lugar de enfrentarlos con un oponente que ya es experto —utiliza la política baseline—, la idea es poder entrenarlos contra ellos mismos, utilizando una versión pasada de su propio entrenamiento. De esta manera, nuestro agente mejora al enfrentarse con un oponente de menor nivel.

Intuitivamente, este enfoque tiene sentido. Volviendo al ejemplo del baloncesto, comentado en la introducción, a nadie se le ocurriría empezar a aprender este deporte jugando contra un equipo de la NBA. Suena más razonable empezar a practicar con gente de un nivel parecido al nuestro, y mejorar de esta manera.

Finalmente, realizaremos una evaluación con nuestro oponente que utiliza la política *baseline*, pudiendo comprobar si el aprendizaje que ha realizado nuestro agente ha sido valioso o no.

6.1. Metodología

La metodología seguida en esta sección es muy similar a la que se usa para entrenar y evaluar algoritmos de stable-baselines3: se realiza el entrenamiento durante un determinado número de etapas y posteriormente se realiza la evaluación en un número de episodios. Nuestra implementación está basada en la implementación disponible en [7].

Respecto al aprendizaje, debemos tener en cuenta cómo manejar la lógica del self-play, es decir, que el agente se entrene enfrentándose a una versión anterior a él.

La idea es tener un directorio, al principio vacío, que irá almacenando los modelos generados por el entrenamiento de nuestro agente. Estos modelos serán las versiones

pasadas de nuestro agente, que se guardan al encontrar un modelo que obtiene mejores recompensas. Para ello definimos un wrapper y un callback.

Por un lado, el wrapper *SlimeVolleySelfPlayEnv* se encarga de cargar el mejor modelo self-play que haya disponible. Para ello redefinimos la función *reset*. Ahora, cuando empecemos un episodio nuevo, nuestro agente usará el modelo guardado previamente con el índice más alto. También redefinimos la función *predict*, que nos devolverá la predicción del mejor modelo si se ha definido, o una predicción aleatoria en caso de que no haya definido un modelo todavía. Habiendo sustituido el mejor modelo en la función *reset*, la función *predict* permite que este modelo tome la acción elegida en función a la política aprendida en el entrenamiento previo. Su implementación es la siguiente:

```
class SlimeVolleySelfPlayEnv(slimevolleygym.SlimeVolleyEnv):
  # wrapper over the normal single player env, but loads the best self play
  ↪  model
  def __init__(self, alg_name):
    super(SlimeVolleySelfPlayEnv, self).__init__()
    self.policy = self
    self.alg_name = alg_name
    self.best_model = None
    self.best_model_filename = None
  def predict(self, obs): # the policy
    if self.best_model is None:
      return self.action_space.sample() # return a random action
    else:
      action, _ = self.best_model.predict(obs)
      return action
  def reset(self):
    # load model if it's there
    modellist = [f for f in listdir(LOGDIR) if f.startswith("history")]
    modellist.sort()
    if len(modellist) > 0:
      filename = join(LOGDIR, self.alg_name, modellist[-1]) # the latest best
      ↪  model
      if filename != self.best_model_filename:
        print("loading model: ", filename)
        self.best_model_filename = filename
        if self.best_model is not None:
          del self.best_model
        if self.alg_name == 'PPO':
          self.best_model = PPO.load(filename, env=self)
        elif self.alg_name == 'DQN':
          self.best_model = DQN.load(filename, env=self)
        elif self.alg_name == 'SAC':
```

```
        self.best_model = SAC.load(filename, env=self)
    return super(SlimeVolleySelfPlayEnv, self).reset()
```

Debemos mencionar también que este wrapper se encarga de asignar la política al oponente. Esto lo hace sobreescribiendo el atributo *self.policy* en la función de inicialización. Originalmente, dicho atributo se inicializa con una instancia de *BaselinePolicy()*. Sin embargo, en nuestro wrapper especificamos que esta política esté definida por nosotros mismos, siendo la misma del agente.

De esta manera, si observamos la función *step* del código fuente, que se muestra a continuación, vemos que la acción del oponente —variable *otherAction*— viene de realizar la predicción usando la política asignada —que antes era *BaselinePolicy*, y ahora estará marcada por uno de los algoritmos probados—. La acción del jugador viene dada por el atributo *action* que toma la función *step*.

```
def step(self, action, otherAction=None):
    """
    baseAction is only used if multiagent mode is True
    note: although the action space is multi-binary, float vectors
    are fine (refer to setAction() to see how they get interpreted)
    """
    done = False
    self.t += 1

    if self.otherAction is not None:
      otherAction = self.otherAction

    if otherAction is None: # override baseline policy
      obs = self.game.agent_left.getObservation()
      otherAction = self.policy.predict(obs)

    if self.atari_mode:
      action = self.discreteToBox(action)
      otherAction = self.discreteToBox(otherAction)

    self.game.agent_left.setAction(otherAction)
    self.game.agent_right.setAction(action) # external agent is agent_right
    ...
```

Las últimas líneas que se muestran de esta función indican cómo gestionar la toma de acciones según si el agente se encuentra a la izquierda —oponente— o a la derecha —jugador—. Esto es importante porque los valores de las coordenadas variarán en función de lado al que se encuentre cada uno de los agentes. Cada objeto *agent_ left* y *agent_ right* trabaja con un rango de posiciones diferentes.

Por otro lado tenemos el callback *SelfPlayCallback*, que hereda de *EvalCallback*, y cuya función es evaluar nuestro modelo cada vez que haya transcurrido un número determinado de etapas. Si en esta evaluación se supera un umbral determinado, guardamos el que sería nuestro nuevo mejor modelo siguiendo la nomenclatura *history_XXXXXXXX*, donde *XXXXXXXX* denota un índice autoincremental. Este modelo .zip será el que toma nuestro agente al ejecutar la función *reset* del entorno y según el cual toma las acciones sucesivas, tal como hemos comentado al describir el wrapper *SlimeVolleySelfPlayEnv* El código se muestra a continuación:

```
class SelfPlayCallback(EvalCallback):
  # hacked it to only save new version of best model if beats prev self by
  ↪  BEST_THRESHOLD score
  # after saving model, resets the best score to be BEST_THRESHOLD
  def __init__(self, alg_name, *args, **kwargs):
    super(SelfPlayCallback, self).__init__(*args, **kwargs)
    self.best_mean_reward = BEST_THRESHOLD
    self.generation = 0
    self.alg_name = alg_name
  def _on_step(self) -> bool:
    result = super(SelfPlayCallback, self)._on_step()
    if result and self.best_mean_reward > BEST_THRESHOLD:
      self.generation += 1
      print("SELFPLAY: mean_reward achieved:", self.best_mean_reward)
      print("SELFPLAY: new best model, bumping up generation to",
      ↪  self.generation)
      source_file = join(LOGDIR, self.alg_name, "best_model.zip")
      backup_file = join(LOGDIR, self.alg_name,
      ↪  "history_"+str(self.generation).zfill(8)+".zip")
      copyfile(source_file, backup_file)
      self.best_mean_reward = BEST_THRESHOLD
    return result
```

Respecto a la evaluación, utilizaremos la misma función que definimos en la sección 5.2.1, que nos permite especificar la política a seguir y el número de episodios.

Por ejemplo, el entrenamiento de un agente PPO mediante self-play se haría de la siguiente manera:

```
env = Monitor(env=SlimeVolleySelfPlayEnv(),
              filename=join(LOGDIR, 'PPO', 'ppo15m_v2_train'))
model = PPO('MlpPolicy', env, n_steps=4096, verbose=1)
eval_callback = SelfPlayCallback(env,
                                 best_model_save_path=join(LOGDIR, 'PPO'),
                                 log_path=join(LOGDIR, 'PPO'),
```

```
                                    eval_freq=EVAL_FREQ,
                                    n_eval_episodes=EVAL_EPISODES,
                                    deterministic=False)
model.learn(total_timesteps=15_000_000, callback=eval_callback)
model.save(join(LOGDIR, 'PPO', 'final_model'))
```

En nuestras pruebas, con el objetivo de observar el efecto del número de etapas en este tipo de aprendizajes, probamos a entrenar nuestro agente con distinto número de etapas, según el algoritmo. En este caso concreto, probamos a entrenarlo con 3, 6 y 15 millones de etapas, siendo este el número de etapas totales donde ocurren los distintos episodios de self-play. Para la prueba de 15 millones de etapas también llegamos a probar a utilizar el atributo *n_steps* por defecto y *n_steps*=4096. *n_steps* indica el número de etapas que se toman por cada entorno por actualización. Esta última prueba la hacemos para constrastar resultados con los experimentos realizados en [7].

Además, en algunos casos probamos a evaluar tanto el mejor modelo tras el entrenamiento, como el último modelo generado, para ver si las diferencias son significativas.

Por tanto, poniendo el ejemplo del último campeón entrenado en 15000000 etapas, la evaluación se realizaría de la siguiente manera:

```
slime_env_gym = gym.make('SlimeVolley-v0')
ppo_policy = PPO.load(join(LOGDIR, 'PPO', 'history_00000148.zip'),
↪  env=slime_env_gym)
eval_rew, eval_len = evaluate_algorithm(slime_env_gym, ppo_policy, 1000)
save_results(eval_rew, eval_len, join('PPO', 'ppo_self_eval_15m_v2.csv'))
ppo_eval = pd.read_csv(join(LOGDIR, 'PPO', 'ppo_self_eval_15m_v2.csv')).loc[:,
↪  'eval_rewards'].expanding().mean()
fig, ax = plt.subplots(figsize=(15, 6))
ax.plot(ppo_eval, label='15M, last champion v2')
ax.set_xlabel('Episodes')
ax.set_ylabel('Reward')
ax.set_title('PPO self-play evaluation')
ax.legend(loc='lower right');
```

Para el caso de DQN y SAC, el proceso es el mismo. Lo único a tener en cuenta es que en SAC, al requerir un espacio de acciones continuo, es necesario añadir el wrapper *DiscreteToBoxWrapper* que comentamos en la sección anterior.

Aunque para PPO y DQN fijamos el número de etapas en 15M —ya que en las primeras pruebas se observa que este número es necesario que sea incrementado—, para SAC debemos quedarnos en 2M debido a el tiempo de cómputo adicional que necesita este algoritmo.

Figura 6.1: Entrenamientos usando PPO y self-play

6.2. Resultados numéricos

En la realización de estos experimentos, podemos ver resultados diferentes.

En primer lugar, comentaremos el entrenamiento y evaluación de nuestros agentes utilizando el algoritmo PPO y la técnica self-play mencionada.

En primer lugar, la figura 6.1 muestra el entrenamiento de nuestro agente para simulaciones de 3, 6 y 15 millones de etapas. En las curvas de aprendizaje se puede observar cómo, a pesar de usar los mismos hiperparámetros —por defecto $n\_steps = 2048$—, el aprendizaje no siempre es igual. Esto se debe al componente aleatorio que tienen este tipo de algoritmos. Por ejemplo, llegando a los 15000 episodios, la simulación de 15 millones de etapas nos ofrece un mejor rendimiento que la simulación de 6 millones de etapas, a pesar de tener la misma configuración.

También podemos ver el efecto que el hiperparámetro n_steps tiene en este entrenamiento. Modificando este valor, conseguimos un entrenamiento más rápido y eficiente en muestras.

Por otro lado, en la figura 6.2 se muestra la evaluación de los algoritmos PPO, y se demuestra esta diferencia que comentamos. El modelo que mejor se comporta en el entrenamiento usando self-play es el que mejor resultados ofrece al realizar la evaluación contra el oponente baseline.

En este algoritmo concreto, es destacable ver cómo, a pesar del gran número de etapas de entrenamiento —15 millones supone un gran costo computacional— y los relativamente buenos resultados en el entrenamiento —llegamos a recompensas medias superiores a 4—, el agente entrenado no llega a tener un buen rendimiento contra el agente experto. Los agentes que utilizan el parámetro n_steps tienen un rendimiento

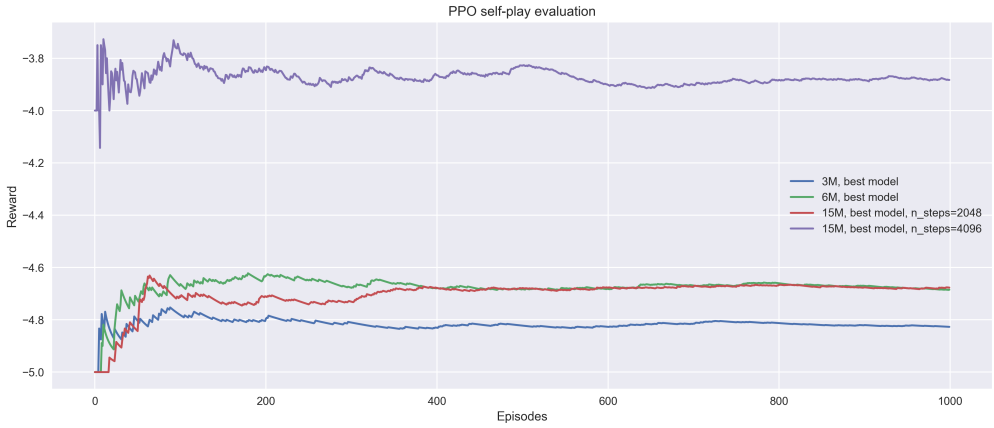

Figura 6.2: Evaluaciones usando PPO y self-play

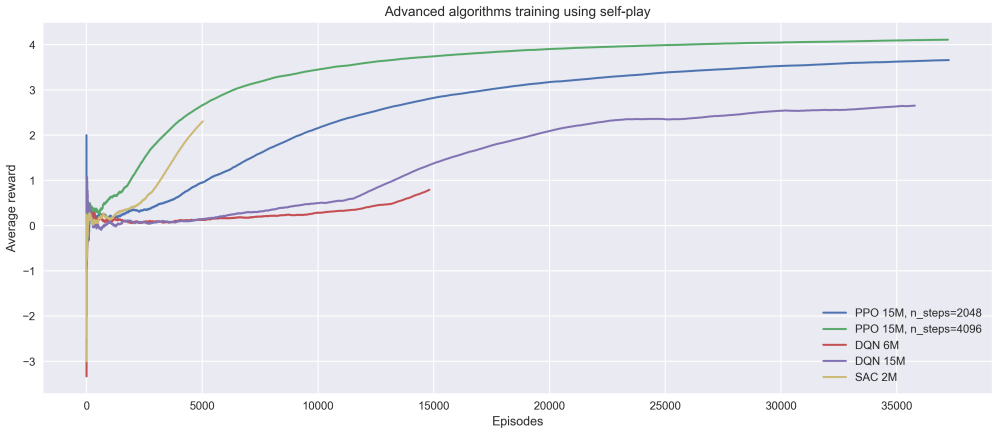

Figura 6.3: Entrenamiento de algoritmos avanzados usando self-play

similar a la política aleatoria, y la mejora que se obtiene el modificar este parámetro es muy leve.

La figura 6.3 muestra los entrenamientos de PPO, DQN y SAC usando self-play. Entre estos, DQN es el que peores resultados ofrece y PPO el que mejores. Sin embargo, debemos destacar el rendimiento de SAC, que a pesar de utilizar un número de etapas relativamente bajo se demuestra muy eficiente en muestras, algo que ya apreciamos cuando se usaba contra la política baseline.

En la evaluación, que se muestra en la figura 6.4, podemos ver resultados acordes. A pesar de que en el entrenamiento el rendimiento de PPO era superior, acaba siendo el que peor actúa al enfrentarse con el oponente baseline. Esto es algo que puede pasar al trabajar con técnicas self-play, ya que el entrenamiento contra una versión anterior del agente no siempre es válido en un entorno real, contra un oponente experto.

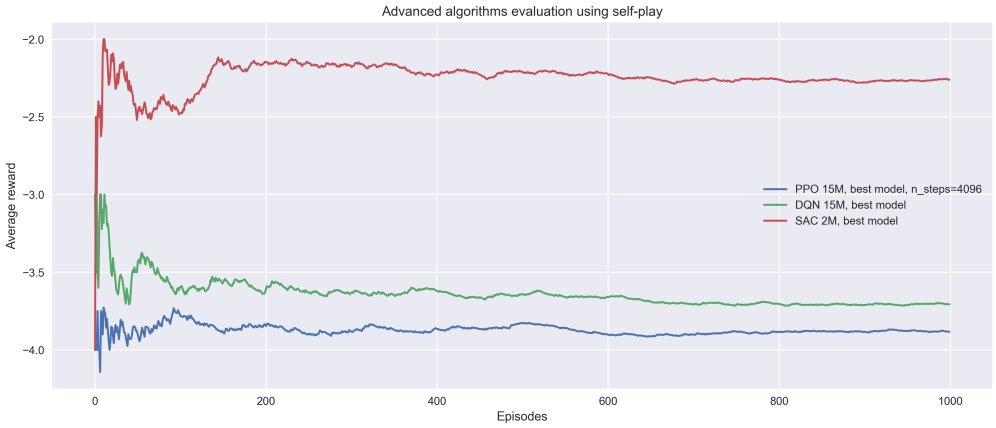

Figura 6.4: Evaluación de algoritmos avanzados usando self-play

Al igual que antes, SAC es el algoritmo que mejores resultados nos devuelve, con una diferencia considerable respecto a PPO y DQN. Este resultado sugiere que este entorno se beneficia de la mayor exploración que se consigue al añadir la entropía a la recompensa tal como hace SAC.

Capítulo 7

Conclusiones y líneas futuras

Tras la realización de este trabajo, podemos destacar varios aprendizajes.

En primer lugar, hemos hecho un repaso a todos los fundamentos teóricos que rodean el aprendizaje por refuerzo, pudiendo así entender los conceptos que se utilizan en los algoritmos empleados en el trabajo. Además, hemos relacionado todos estos conceptos teóricos con el entorno estudiado.

Por otra parte, hemos podido trabajar con la interfaz *gym*, ampliamente utilizada a la hora de tratar con entornos RL, así como de la librería *stable-baselines3*, una de las librerías más consolidadas dentro de la implementación de algoritmos RL.

Además de esto, hemos podido ver distintos tipos de algoritmos, con varios niveles de complejidad, tanto a nivel teórico como práctico, comparando su implementación, su desempeño y sus resultados.

Respecto a las conclusiones que hemos podido obtener tras la realización de este trabajo, podemos mencionar cuatro puntos.

- El uso de algoritmos de aprendizaje por refuerzo puede ser una opción válida para entornos como SlimeVolleyGym, donde el espacio de estados y acciones tiene múltiples dimensiones.
- Dentro de estos algoritmos, aquellos que utilizan aproximación lineal de la función valor no se han podido demostrar del todo válidos, si bien sí que se observan muestras de aprendizaje, concretamente en el caso de SARSA(λ). Con el equipo que utilizamos para realizar las simulaciones y el tiempo que disponemos para la elección de hiperparámetros no podemos garantizar el aprendizaje. En cualquier caso, si pudieran ser válidos, no serían lo suficientemente eficientes en muestras, tal como hemos visto en nuestros experimentos.

- Por otra parte, aquellos algoritmos que utilizan una aproximación no lineal de la función valor o la política sí que ofrecen resultados válidos, comparables a los observados en [7]. Concretamente, PPO y SAC son los que mejores resultados nos devuelven dadas las características del entorno. Debido a la implementación que se utiliza en stable-baselines3, también son más eficientes en términos de tiempo de cómputo, en comparación con la implementación de los algoritmos *clásicos*.
- Finalmente, dentro de los algoritmos considerados *avanzados*, hemos podido comprobar que el uso de la técnica *self-play* nos ofrece resultados válidos, si bien estos necesitan un número de etapas considerablemente mayor. Este resultado puede llegar a ser relevante, ya que en este tipo de problemas no siempre se dispone de un modelo de referencia —en nuestro caso la política baseline—. Es interesante que un agente pueda llegar a aprender de la experiencia, ya no solo obtenida mediante la interacción en un entorno con agentes externos, sino también a partir la propia experiencia anterior del agente.

En lo que se refiere a posibles líneas futuras de desarrollo, podríamos resaltar tres.

- Para tener mayor certeza sobre las pruebas realizadas, todos los experimentos podrían realizarse en lotes, utilizando distintas semillas y tomando una media de los resultados obtenidos. El principal motivo por el que esta línea podría ser interesante es debido al componente aleatoria que presentan los algoritmos RL. Tomando distintas iteraciones tendríamos unos resultados más fieles a la realidad.
- En nuestro proyecto, a la hora de probar algoritmos como PPO, DQN o SAC, únicamente hemos utilizado *stable-baselines3* debido a las ventajas que presenta, comentadas anteriormente. Otra línea podría consistir en probar distintas implementaciones de los mismos algoritmos, con tal de observar si la implementación concreta de un algoritmo influye en los resultados obtenidos en nuestro entorno.
- Por último, dado que los resultados obtenidos en en capítulo 4 no son los esperados, se podrían tomar alternativas para realizar distintos experimentos. Dado que el tiempo de cómputo que necesitan estos algoritmo es mayor, se podría probar a realizar una implementación en un lenguaje más eficiente, como C++. También, los experimentos se podrían realizar con un número de etapas mucho mayor en un equipo más potente, para poder comprobar realmente si este aprendizaje se mantiene con el tiempo.

Personalmente, realizar este Trabajo de Fin de Máster ha sido una experiencia interesante y enriquecedora. Poder investigar acerca del aprendizaje por refuerzo, una disciplina

poco conocida para mí, me ha hecho darme cuenta de las infinitas posibilidades que ofrece. A pesar de que este proyecto haya puesto el foco en un único entorno, los conceptos y herramientas utilizadas son universales para muchos de los problemas que busca resolver el aprendizaje por refuerzo.

Poder haber comprendido estos conceptos y empleado estas herramientas me parece clave en un futuro donde la inteligencia artificial podrá ayudarnos a resolver cada vez más problemas de nuestra vida diaria. Problemas de los que, como comentamos en la introducción, quizás no seamos conscientes aún.

Bibliografía

[1] Shweta Bhatt. Reinforcement learning 101. https://towardsdatascience.com/reinforcement-learning-101-e24b50e1d292.

[2] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[3] Suf. The history of reinforcement learning. https://researchdatapod.com/history-reinforcement-learning/.

[4] Elizabeth G. E. Kyonka. *Law of Effect*, pages 868–870. Springer US, Boston, MA, 2011.

[5] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016.

[6] Antonin Raffin, Ashley Hill, Adam Gleave, Anssi Kanervisto, Maximilian Ernestus, and Noah Dormann. Stable-baselines3: Reliable reinforcement learning implementations. *Journal of Machine Learning Research*, 22(268):1–8, 2021.

[7] David Ha. Slime volleyball gym environment. https://github.com/hardmaru/slimevolleygym, 2020.

[8] Blas Galván, David Greiner, Jacques Periaux, Mourad Sefrioui, and Gabriel Winter. Parallel evolutionary computation for solving complex cfd optimization problems : A review and some nozzle applications. 12 2003.

[9] Prafulla Dhariwal, Christopher Hesse, Oleg Klimov, Alex Nichol, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, Yuhuai Wu, and Peter Zhokhov. Openai baselines. https://github.com/openai/baselines, 2017.

[10] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk,

Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020.

[11] The pandas development team. pandas-dev/pandas: Pandas, February 2020.

[12] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[13] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007.

[14] Lilian Weng. A (long) peek into reinforcement learning. *lilianweng.github.io*, 2018.

[15] OpenAI. Spaces. `https://www.gymlibrary.ml/content/spaces/`.

[16] stable baselines3. Rl algorithms. `https://stable-baselines3.readthedocs.io/en/master/guide/algos.html`.

[17] Juan José Alcaraz Espín. Introducción a los Algoritmos de Aprendizaje por Refuerzo. 2022.

[18] Juan José Alcaraz Espín. Métodos Monte Carlo. 2022.

[19] Juan José Alcaraz Espín. Métodos Temporal Difference. 2022.

[20] Juan José Alcaraz Espín. Métodos VFA (Value Function Approximation). 2022.

[21] Juan José Alcaraz Espín. Métodos Policy Gradient. 2022.

[22] OpenAI. Wrappers. `https://www.gymlibrary.ml/content/wrappers/`.

[23] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning, 2013.

[24] Stable Baselines3. Source code for stable_baselines3.dqn.policies. `https://stable-baselines3.readthedocs.io/en/master/_modules/stable_baselines3/dqn/policies.html#DQNPolicy`.

[25] OpenAI Spinning Up. Proximal policy optimization. `https://spinningup.openai.com/en/latest/algorithms/ppo.html`.

[26] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms, 2017.

[27] Stable Baselines3. Ppo. https://stable-baselines3.readthedocs.io/en/master/modules/ppo.html.

[28] Joshua Achiam. Simplified ppo-clip objective. https://drive.google.com/file/d/1PDzn9RPvaXjJFZkGeapMHbHGiWWW20Ey/view.

[29] OpenAI Spinning Up. Soft actor-critic. https://spinningup.openai.com/en/latest/algorithms/sac.html.

[30] OpenAI Spinning Up. Deep deterministic policy gradient. https://spinningup.openai.com/en/latest/algorithms/ddpg.html.

[31] David Ha. Neural slime volleyball. *blog.otoro.net*, 2015.

[32] Stable Baselines3. Callbacks. https://stable-baselines3.readthedocs.io/en/master/guide/callbacks.html.