ESCUELA TÉCNICA SUPERIOR DE INGENERÍA INFORMÁTICA/ UNIVERSIDAD DE SEVILLA

Trabajo fin de Grado

Grado en Ingeniería Informática del Software

Aprendizaje por Refuerzo con Rainbow DQN

Realizado por
Rubén Casal Ferrero

Dirigido por
José Luis Ruíz Reina

Departamento
Ciencias de la Computación
e Inteligencia Artificial

Sevilla, Junio de 2024

Abstract

This project focuses on enhancing the capabilities of reinforcement learning algorithms through the application of the Rainbow DQN algorithm within the Ms Pacman game environment. The goal was to explore the integration of advanced reinforcement learning strategies to improve decision-making processes in complex and dynamic environments. By experimenting with various configurations and components, the project identifies optimal settings that maximize learning efficiency and algorithm performance.

The findings suggest that careful tuning of the learning architecture and strategy components significantly boosts the effectiveness of the model. These enhancements help in achieving superior performance compared to traditional reinforcement learning models, providing insights into the development of more intelligent and adaptable AI systems. The project demonstrates the potential of advanced reinforcement learning techniques in solving complex problems in gaming and other dynamic scenarios.

Índice general

Índice de figuras

CAPÍTULO 1

Introducción

En los últimos años, hemos sidos testigos de una transformación sin precedentes en el campo de la informática y la tecnología. Concretamente la inteligencia artificial, ha experimentado un crecimiento exponencial, impulsada por los avances en el procesamiento de datos y la capacidad de cómputo.

Esta revolución de la inteligencia artificial se inspira en los principios de la biología y el funcionamiento del ser humano y de los animales. Sin ir más lejos, en las redes neuronales artificiales, podemos observar cierto paralelismo en su funcionamiento con el cerebro, en el cual las neuronas interactúan entre sí a traves de conexiones llamadas sinapsis para procesar la información y transmitir una señal de respuesta.

El campo de estudio de este TFG, los algoritmos de aprendizaje por refuerzo en el entorno de juego clásicos de Atari, se inspira en el aprendizaje por prueba y error, donde tenemos un `entorno` que es el mundo que nos rodea con el que podemos interactuar, un `agente` el cual podría ser cualquier ser vivo o nosotros mismos y una `recompensa` que nos incentiva a aprender una nueva tarea o habilidad. Todo el código relacionado con este trabajo estara disponible en el siguiente repositorio de GitHub `https://github.com/RubenCasal/Pacman_Rainbow`

1.1– Motivación

La elección de dedicar la realización de mi Trabajo de Fin de Grado en el estudio y la aplicación del algoritmo `Rainbow DQN` en el juego `Ms Pacman`, proviene del interés personal sobre el campo del aprendizaje por refuerzo. Con anterioridad había realizado algunos proyectos más sencillos para aprender sobre la materia, y el TFG me pareció una gran oportunidad para enfrentarme a un desafio técnico más exigente que permitiera profundizar en el tema, ya que como he mencionado anteriormente me parece asombroso cómo el software puede emular el proceso del aprendizaje humano. La decisión de aplicar `Rainbow DQN`, se debe a que este algoritmo se nutre de muchas de las mejoras que habían demostrado tener un rendimiento mayor por separado y las une, resultando en un rendimiento aún mayor que por separado, permitiendo aprender ampliamente el campo del reinforcement learning con este algoritmo. Se ha escogido el juego de Atari de `Ms Pacman` ya que es un juego conocido por la gran parte de la población y es uno de los juegos con mayor complejidad para los algoritmo del machine learning dentro del repertorio de los juegos de

Atari.

1.2– Objetivos

El objetivo de este TFG es estudiar, desarrollar e implementar el algoritmo de `Rainbow DQN` en el entorno del videojuego clásico de `Ms Pacman` de Atari, pasando por todas las mejoras que incluye este algoritmo.

- DQN
- Double DQN
- Prioritized DDQN
- Dueling Networks
- Distributional DQN
- Noisy Networks
- N-step Learning

En este trabajo se realizará una explicación del fundamento teórico y matemático sobre el que se basa el algoritmo Rainbow DQN, pasando por la explicación del aprendizaje por refuerzo, el Proceso de Decisión de Markov, Q-Learning... . Después se definirán las mejoras que componen el algoritmo, explicando el marco teórico de cada una de ellas y después se mostrará el código necesario para poder realizar la implementación en Python, la implementación seguirá las directrices de los artículos publicados sobre las respectivas mejoras y otras implementaciones de Rainbow DQN, pero adaptadas al entorno de Ms-Pacman, los cuales se pueden encontrar en la bibliografía. Finalmente se realizará una experimentación del algoritmo donde se probarán distintos valores de los hiperparámetros, distintas funciones de recompensa y otras modificaciones, para entender cómo estas afectan al resultado final y maximizar el rendimiento del modelo.

1.3– Herramientas

En este apartado se definen las herramientas utilizadas en la implementación del algoritmo de aprendizaje por refuerzo Rainbow DQN en el entorno de Ms Pacman, se elebora una descripción detallada de cada una de las tecnologías y librerías seleccionadas, así cómo su papel dentro del proyecto.

Entorno de Desarrollo Integrado (IDE): Visual Studio Code

Visual Studio Code (VS Code) es un IDE ligero pero potente, diseñado para el desarrollo de software moderno. Ofrece funcionalidades como autocompletado de código, depuración integrada, control de versiones y una amplia variedad de extensiones para personalizar el entorno de desarrollo. Su compatibilidad con Python y facilidad de integración con sistemas de control de versiones como Git lo convierten en una elección ideal para este proyecto.

Lenguaje de Programación: Python

Python es un lenguaje de programación de alto nivel, es particularmente popular en la comunidad científica y de investigación por su extenso ecosistema de librerías para ciencia de datos, inteligencia artificial y aprendizaje automático. Python facilita la implementación de algoritmos complejos y manipulación de grandes volúmenes de datos, siendo esencial para el desarrollo de aplicaciones de IA, como la que se propone en este proyecto.

Framework de Inteligencia Artificial: PyTorch

PyTorch es un framework de aprendizaje profundo `open source` desarrollado por Facebook's AI Research lab. Se caracteriza por su flexibilidad, eficiencia y facilidad de uso, ofreciendo una rica API para la construcción y entrenamiento de redes neuronales. PyTorch es especialmente apreciado por su capacidad de realizar cálculos tensoriales con aceleración GPU y su diferenciación automática, facilitando la implementación de algoritmos de aprendizaje por refuerzo como el Rainbow DQN.

Entorno de Simulación: Gymnasium de OpenAI

Gymnasium es una herramienta desarrollada por OpenAI que proporciona una amplia variedad de entornos de simulación para el entrenamiento y evaluación de algoritmos de inteligencia artificial. El entorno de `Ms Pacman` dentro de Gymnasium ofrece un campo de prueba ideal para el algoritmo `Rainbow DQN`, permitiendo simular interacciones complejas dentro del juego.

Librerías Auxiliares

NumPy: Una librería fundamental para Python. Ofrece soporte para arrays y matrices de gran tamaño, junto con una colección de funciones matemáticas para operar sobre estos datos. NumPy es crucial para la gestión de estados y la manipulación de datos numéricos en el proyecto.

Matplotlib: Una librería de gráficas para Python que ofrece funcionalidades para la creación de una amplia gama de gráficos y visualizaciones. Matplotlib se utilizará para realizar gráficas sobre el rendimiento y la evolución del aprendizaje del modelo a lo largo del tiempo.

OpenCV: Una librería de visión por ordenador. OpenCV permite el procesamiento y análisis de imágenes en tiempo real, lo cual es fundamental para el tratamiento de las imágenes que recibe el agente como input en el entorno de Ms Pacman.

Control de Versiones: GitHub

GitHub es una plataforma de desarrollo colaborativo para alojar proyectos utilizando el sistema de control de versiones Git. El uso de GitHub en este proyecto garantiza una gestión eficiente del código fuente y facilita la documentación y el intercambio de conocimientos con la comunidad.

La combinación de estas herramientas y tecnologías proporciona una base sólida para el desarrollo eficiente del proyecto. Cada componente ha sido seleccionado por su facilidad de uso, compatibilidad y conocimiento de la tecnología por un uso en proyecto anteriores.

CAPÍTULO 2

Estado del Arte

El aprendizaje por refuerzo (Reinforcement Learning, RL) ha experimentado un desarrollo significativo en las últimas décadas, especialmente en su aplicación a juegos, donde ha servido para probar teorías computacionales del aprendizaje a un coste menor para avanzar en el campo de la inteligencia artificial. Este capítulo presenta un panorama de las aplicaciones actuales y pasadas del RL, particularmente en juegos de Atari, destacando las funcionalidades que comparten con el proyecto de Trabajo de Fin de Grado (TFG) propuesto, que se enfoca en la implementación y optimización del algoritmo Rainbow DQN en el entorno de Ms Pacman.

2.1– Aplicaciones Históricas y Actuales

El proyecto Arcade Learning Environment (ALE) proporcionó uno de los primeros bancos de pruebas y entornos significativos para el aprendizaje por refuerzo, ofreciendo una interfaz común para desarrollar algoritmos sobre juegos de Atari. Con el desarrollo de DQN (Deep Q-Networks) demostraron la capacidad para superar el nivel de juego humano en varios juegos de Atari, estableciendo un antes y un después en este campo.

Los avances no solo se han producido en el ámbito de los juegos de Atari, sino que también hemos podido ver avances en el campo del Reinsforcement Learning, los cuales se han hecho bastantes famosos en el sector de la Inteligencia Artifical, como AlphaGo de mano de DeepMind y su sucesor AlphaZero, que dominaron el juego del Go, superando a los mejores del mundo, pero además, demostraron, que se puede extrapolar su conocimiento para mostrar un rendimiento excelente en otros juegos de estrategia distintos con el que inicialmente fueron entrenados como el ajedrez.

Posteriormente, la evolución de DQN dio lugar a variantes más sofisticadas como Double DQN, Dueling DQN y Prioritized Experience Replay, que buscaron abordar las limitaciones encontradas en el DQN original. Estos avances han demostrado mejoras significativas en términos de estabilidad y rendimiento del aprendizaje. El algoritmo Rainbow DQN integra ocho mejoras clave del DQN en una sola arquitectura, marcando un hito en la eficiencia y efectividad del aprendizaje por refuerzo en entornos complejos. Este avance ha sido crucial para demostrar la capacidad de generalización y adaptabilidad de los algoritmos de RL.

Aunque el foco inicial de RL en juegos de Atari fue demostrar la capacidad de aprendizaje de máquinas en entornos cerrados y controlados, las técnicas han encontrado aplicaciones en áreas tan diversas como la robótica, la optimización de sistemas de energía y la medicina personalizada, mostrando la versatilidad y el potencial de esta tecnología.

2.2– Componentes de Rainbow DQN

Rainbow DQN es un algoritmo de aprendizaje por refuerzo que integra varias mejoras sobre el algoritmo DQN original para mejorar su rendimiento y estabilidad. La intuición detrás de las mejoras que componen el algoritmo de Rainbow DQN es la siguiente:

- **DDQN (Double Deep Q-Network):** Considera DDQN como el proceso de tomar una segunda opinión antes de tomar una decisión. En lugar de evaluar y elegir una acción usando la misma estimación, que puede ser excesivamente optimista, DDQN separa estas tareas para obtener una valoración más realista y menos sesgada.

- **Prioritized Experience Replay:** Es similar a estudiar para un examen centrándose en los temas que no sabes bien, en lugar de repasar todo con la misma frecuencia. Rainbow prioriza la revisión de experiencias pasadas que son más sorprendentes o menos conocidas, optimizando el aprendizaje.

- **Dueling Networks:** Esta estructura divide la red en dos caminos: uno para estimar el valor general de estar en un estado particular y otro para calcular el beneficio adicional de cada acción. Es como tener una visión más detallada para entender mejor si es la situación la que es buena o es realmente la acción específica la que marca la diferencia.

- **Distributional DQN:** A diferencia de un enfoque que solo predice una recompensa promedio por acción, Distributional DQN considera toda la gama de posibles resultados y sus probabilidades. Es como planificar el futuro pensando en diferentes escenarios posibles en lugar de contar solo con una sola expectativa.

- **Noisy Networks:** En lugar de explorar al azar, Noisy Networks introduce ruido directamente en los parámetros de la red para impulsar la exploración de una manera más coherente y efectiva. Es una especie de impulso interno que motiva al agente a probar cosas nuevas.

- **N-step Learning:** Extiende la mirada hacia adelante, actualizando el valor de una acción basándose en múltiples pasos futuros, en lugar de solo el siguiente. Esto ayuda a aprender secuencias de acciones en lugar de solo mirar a corto plazo, un poco como planear una estrategia en lugar de solo una táctica inmediata.

2.3– Comparación con las aquitecturas por separado

El aprendizaje profundo por refuerzo ha llevado al desarrollo de múltiples variantes de DQN (Deep Q-Network), cada una con la intención de superar limitaciones particulares del algoritmo original. En la Figura 2.1, se muestra una comparación de desempeño de distintas variantes de DQN a lo largo del entrenamiento en un conjunto de juegos de Atari 2600, concretamente en 57 juegos distintos.

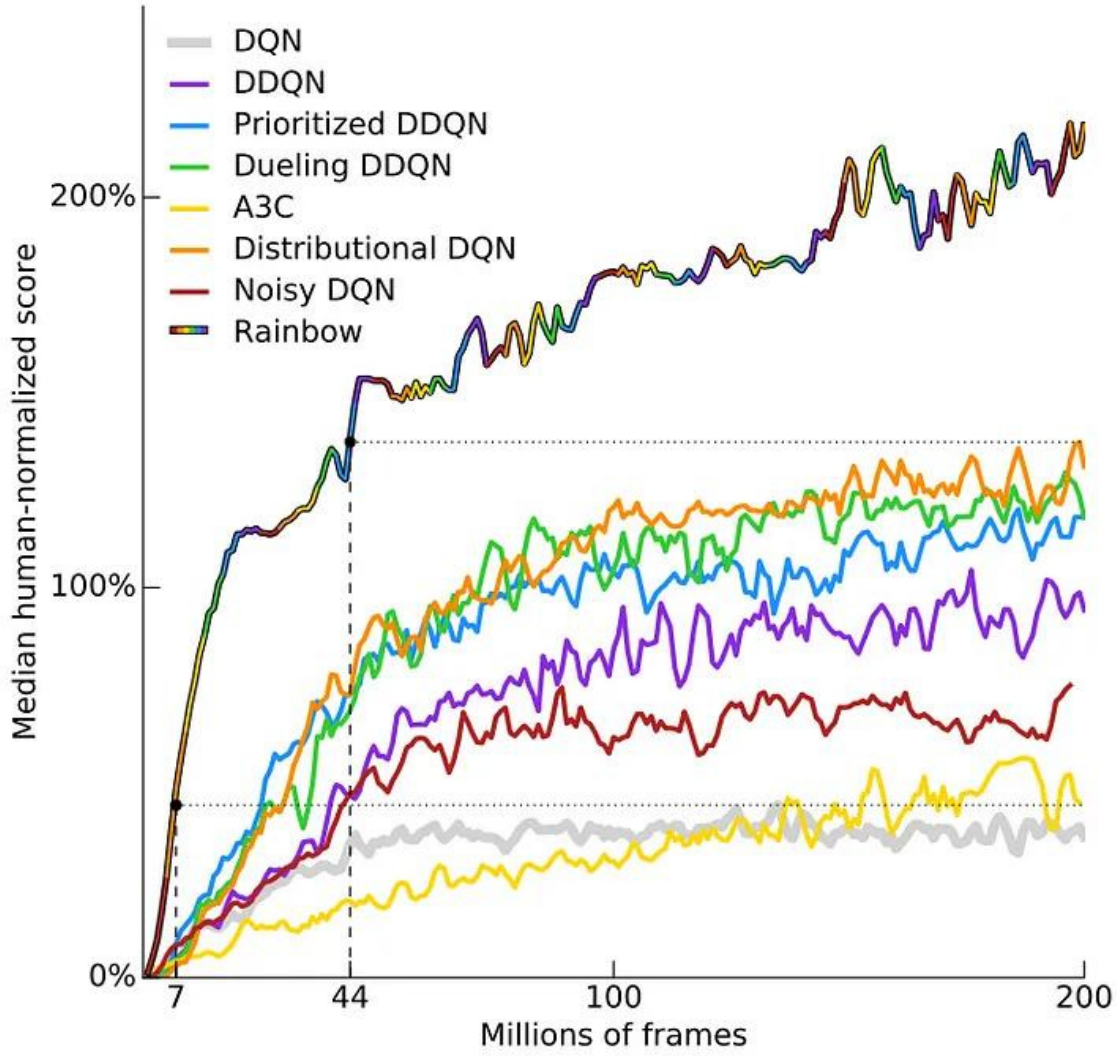

Figura 2.1: Comparación de Rainbow DQN en juego Atari, de "Combining Improvements in Deep Reinforcement Learning"[15]

Lo que se observa en la imagen, es la mediana del rendimiento de las distintas arquitecturas en los distintos juegos de Atari, dicho rendimiento se mide en base a la media del rendimiento de un ser humano experto en dicho juego, es decir, un 0 significa un modelo que tiene un comportamiento aleatorio y un 100 significa que tiene un comportamiento similar a un ser humano, por lo que es muy bueno jugando. Se compara el algoritmo DQN original con sus variantes: DDQN (Double DQN), Prioritized DDQN, Dueling DDQN, Distributional DQN, Noisy DQN, y Rainbow, que es una integración de estas mejoras en una sola arquitectura. En la gráfica no solo podemos ver que la integración conjunta de todas las mejoras obtiene un modelo con un rendimiento mucho mayor, si no que converge en una solución óptima mucho más rápido, permitiendo reducir el coste computacional, debido que hacen falta menos tiempo de entrenamiento para llegar un rendimiento similar a las arquitecturas de las mejoras por separado.

2.4– Desafíos y Limitaciones Actuales en el Aprendizaje por Refuerzo

A pesar de los avances significativos en el aprendizaje por refuerzo, existen varios desafíos y limitaciones que persisten y requieren atención continua. A continuación, veremos algunos de las principales problemas en el desarrollo de RL.

2.4.1. Dependencia de Grandes Cantidad de Datos

Uno de los principales retos en el RL es la gran cantidad de interacciones con el entorno que se necesitan para aprender políticas efectivas. Esto es particularmente problemático en entornos reales donde cada interacción puede tener un coste significativo o requerir de mucho tiempo de entrenamiento.

2.4.2. Transferencia de Políticas entre Tareas Similares

La habilidad de transferir conocimiento entre tareas podría acelerar significativamente el aprendizaje en nuevas tareas. Sin embargo, la transferencia de políticas efectiva entre tareas sigue siendo un desafío debido a la diferencia en la distribución de estados o las dinámicas entre las tareas.

2.4.3. Interpretación de Políticas de Agentes

A medida que los modelos de RL se vuelven más complejos, su interpretabilidad disminuye. Entender y explicar las decisiones tomadas por políticas de RL es crucial, especialmente en aplicaciones donde se requiere confianza y transparencia. Se estan haciendo grandes esfuerzos para mejorar la interpretación de modelos de inteligencia artificial, pero queda mucho por hacer para lograr sistemas de RL que sean completamente transparentes y comprensibles.

2.4.4. Desafíos en Entornos No Estacionarios o con Información Parcial

En muchos entornos del mundo real, la distribución de los estados puede cambiar con el tiempo o la información completa sobre el estado puede no estar disponible, lo que hace que el entorno sea no estacionario o parcialmente observable. Estos problemas requieren enfoques de RL que puedan adaptarse dinámicamente a los cambios y manejar la incertidumbre inherente a la observación de estados parciales.

CAPÍTULO 3

Aprendizaje por Refuerzo

El aprendizaje por refuerzo (Reinforcement Learning o RL) es un área de la inteligencia artificial que se centra en cómo los agentes deberían tomar decisiones en un entorno para maximizar algun tipo de recompensa acumulativa, la maximización de esta recompensa, lleva a cabo el aprendizaje de un tarea o habilidad. Este capítulo proporciona una descripción detallada de los fundamentos teóricos del RL y sus componentes clave.

3.1– Agente, Entorno y Recompensa

El concepto de agente y entorno es un pilar fundamental en el aprendizaje por refuerzo, la interacción dinámica entre un automata y un mundo con el que interactúa, es el fundamento del concepto de aprendizaje por refuerzo. A continuación se desarrollará todos sus componentes para llegar una comprensión de su funcionamiento.

3.1.1. Agente

En el contexto del RL, un agente se refiere a cualquier entidad capaz de recibir inputs sobre el estado de sus entorno y actuar en consecuencia de ello. Los agentes pueden ser desde programas software hasta robots equipados con sensores para percibir su entorno. La principal característica de un agente es la capacidad de tomar decisiones y ejecutar acciones basadas en el estado de su entorno y lo aprendido mediante la interacción con este, con el objetivo de maximizar una señal de recompensa.

3.1.2. Entorno

El entorno, por otro lado, es el contexto o el mundo externo con el que el agente interactúa. Pueder ser de un tablero de juego, pasando por un entorno físico real o llegando incluso a simulaciones del mundo real. El entorno se define mediante un conjunto de estados y la transiciones entre estos estados, por los cuales va transcurriendo el agente al cabo del tiempo. A cada acción ejecutada por el agente, el entorno responde con un nuevo estado y una nueva recompensa, a esta información otorgada por el entorno dentro del marco del aprendizaje por refuerzo se le conoce como observaciones, y es de lo que se nutre el agente para llevar a cabo su aprendizaje.

3.1.3. Recompensa

El concepto de recompensa es fundamental en el aprendizaje por refuerzo, actuando como la señal que guía el aprendizaje y la adaptación del agente al entorno en el que se encuentra. La recompensa es un feedback cuantitativo que el entorno proporciona al agente tras la ejecución de una acción, reflejando el éxito o fracaso de dicha acción respecto a los objetivos del agente. La estructuración de las recompensas entregadas por el entorno es crucial ya que tiene una influencia determinante en el aprendizaje y comportamiento del agente. Una función de recompensa bien diseñada puede acelerar significativamente el aprendizaje del agente, mientras que una función mal especificada puede llevar a aprendizajes ineficientes o incluso contraproducentes.

3.1.4. Interacción entre Agente y Entorno

La interacción entre el agente y el entorno es un ciclo continuo de observaciones, acciones y recompensas. En cada paso de tiempo, el cual vamos a denominar t, el agente observa el estado S_t del entorno, y basándose en esta observación ejecuta la acción A_t. El entorno, a su vez, transiciona a un nuevo estado S_{t+1} y emite una recompensa R_{t+1} (esta indica el éxito o el fracaso de la acción tomada). Esta secuencia de acciones conforma un episodio, que puede continuar indefinidamente o hasta que se alcance un estado terminal S_T, que podría significar el final de un juego, la finalización de una tarea o el fracaso de la misma. La secuencia completa de todos los episodios constituye la experiencia del agente, de la cual se nutrirá para el aprendizaje y la mejora de su comportamiento.

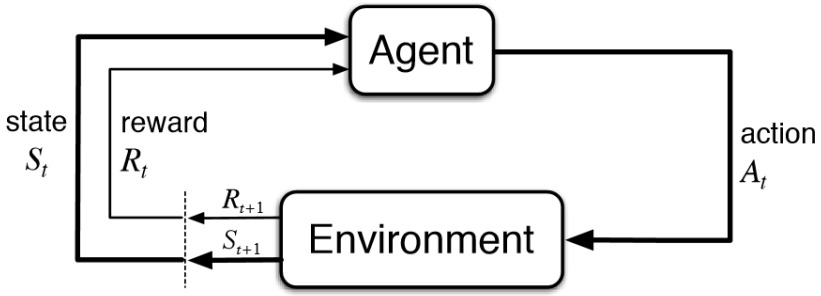

Figura 3.1: Interacción Agente/Entorno [16]

Aprendizaje Off-policy y On-policy

El aprendizaje por refuerzo puede clasificarse en dos enfoques principales basados en cómo las políticas son evaluadas y mejoradas:

Aprendizaje Online El aprendizaje online en el contexto de aprendizaje por refuerzo implica que el agente aprende de manera continua a medida que interactúa con el entorno. En este enfoque, cada decisión o acción tomada por el agente influye de inmediato en su desarrollo y actualización de la política. El aprendizaje online es dinámico y permite que el agente se adapte rápidamente a los cambios o nuevas informaciones que puedan surgir en su entorno. Este método es ideal en entornos que cambian constantemente o cuando el agente tiene que empezar a actuar sin tener mucho conocimiento previo.

Aprendizaje Offline A diferencia del aprendizaje online, el aprendizaje offline se basa en un conjunto predefinido de datos o experiencias para entrenar o mejorar las políticas del agente. En este modelo, el agente no aprende en tiempo real, sino que utiliza una gran cantidad de datos históricos para entrenar o perfeccionar su modelo de decisión antes de ser desplegado para actuar en el mundo real. El aprendizaje offline es útil cuando los entornos son estables o cuando se pueden simular situaciones diversas para mejorar la robustez del agente antes de su implementación. Este método permite una exploración exhaustiva y controlada de las posibles situaciones, minimizando los riesgos antes de la ejecución en vivo.

Estas modalidades de aprendizaje representan dos enfoques distintos en la manera de entrenar y optimizar los algoritmos de aprendizaje por refuerzo. Mientras que el aprendizaje online promueve la adaptabilidad y el aprendizaje continuo, el aprendizaje offline permite un análisis profundo y una preparación cuidadosa utilizando grandes conjuntos de datos. Cada uno tiene sus ventajas dependiendo del contexto de aplicación y los requisitos específicos del entorno o de la tarea que el agente debe realizar.

3.2– Proceso de Decisión de Markov

Un Proceso de Decisión de Markov (MDP) es un marco matemático para modelar la toma de decisiones en entornos donde los resultados son parcialmente aleatorios y las acciones que se pueden tomar están bajo el control de un agente. Los MDP son cruciales en el aprendizaje por refuerzo, ya que proporcionan una formulación formal para los problemas de decisión secuencial que un agente debe resolver para tener un comportamiento óptimo para llevar a cabo una determinada tarea. Formalmente, un MDP es una tupla (S, A, P, R, γ) donde:

- **Conjunto de Estados (S)**: Todos los posibles estados en los que puede encontrarse el entorno. Cada estado proporciona la información necesaria para decisiones futuras.
- **Conjunto de Acciones (A)**: Todas las acciones disponibles que el agente puede elegir en cada estado. La selección de una acción lleva al agente a un nuevo estado según la dinámica del entorno.
- **Función de Transición de Estado (P)**: Define la probabilidad $P(s'|s, a)$ de transicionar a un nuevo estado s', dado un estado actual s y una acción a.
- **Función de Recompensa (R)**: Asigna un valor numérico a la transición de un estado a otro debido a una acción, indicando la utilidad de esa transición para el agente $R(s, a, s')$, siendo s el estado actual y s' el estado al que transiciona.
- **Factor de Descuento (γ)**: Pondera la importancia de las recompensas futuras en comparación con las inmediatas.

3.2.1. Recompensa

La recompensa es una señal numérica que el agente recibe del entorno después de realizar una acción. Esta señal le indica al agente como de buena fue su acción en términos de alcanzar su objetivo. Por ejemplo, si el objetivo es moverse hacia adelante, una acción que efectivamente mueva al agente hacia adelante podría recibir una recompensa positiva, mientras que moverse en la dirección opuesta podría resultar en una recompensa menor

o incluso una penalización. Aunque cada acción produce una recompensa inmediata, el verdadero objetivo del agente no es solo maximizar la recompensa en un solo paso, sino acumular la mayor cantidad de recompensa posible a lo largo del tiempo. Esto significa que a veces el agente podría optar por recibir una recompensa más pequeña ahora si eso conduce a recompensas mayores en el futuro. Podemos reflejar la intuición detrás de la formulación de la recompensa con la siguiente frase:

> "Todo lo que entendemos por metas y propósitos puede ser bien concebido como la maximización del valor esperado de la suma acumulativa de una señal escalar recibida (llamada recompensa).[16]

3.2.2. Función de Retorno de Recompensas

El aprendizaje por refuerzo establece como objetivo principal maximizar la suma acumulativa de recompensas que un agente recibe en el tiempo. Esta suma se formaliza a través del concepto de *retorno*, definido como la suma total de recompensas obtenidas desde un paso temporal específico hasta el final de un episodio.

Matemáticamente, el retorno G_t a partir de un tiempo t se define como:

$$G_t = R_{t+1} + R_{t+2} + R_{t+3} + \cdots + R_T, \tag{3.1}$$

donde T denota el último paso de tiempo dentro de un episodio. Esta configuración se aplica cuando las interacciones entre el agente y el entorno están claramente segmentadas en episodios discretos, como partidas de un juego o recorridos por un laberinto, donde cada episodio termina en un estado terminal.

Factor Descuento

Para abordar efectivamente la acumulación de recompensas en todas las tareas, se introduce el concepto de *descuento*. El retorno descontado se expresa como:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots, \tag{3.2}$$

donde γ representa el factor de descuento, satisfaciendo $0 \leq \gamma \leq 1$. Este factor reduce el valor presente de las recompensas futuras, permitiendo al agente valorar más las recompensas inmediatas en comparación con las más distantes, y es crucial para la planificación a largo plazo.

Esta estructura facilita el diseño de algoritmos que optimizan las políticas de acción basadas en la maximización del retorno esperado descontado, proporcionando un enfoque efectivo y eficiente en diversas configuraciones de tareas en aprendizaje por refuerzo.

3.2.3. Políticas

Una política define la manera en que un agente se comporta en un determinado momento. Formalmente, es una estrategia que asigna a cada estado s, del entorno una acción a, que el agente ejecutará cuando se encuentre en ese estado. Las políticas pueden ser deterministas o estocásticas:

Políticas Deterministas: En una política determinista, para cada estado hay exactamente una acción que será seleccionada por el agente. Esto significa que la política define

una función directa de estados a acciones, $\pi(s) \rightarrow A$, donde $\pi(s) \in A(s)$ es la acción seleccionada en el estado s.

Políticas Estocásticas: A diferencia de las deterministas, una política estocástica asigna una probabilidad a cada acción en un estado dado, reflejando la incertidumbre o la exploración en la toma de decisiones del agente. Así, $\pi(a|s)$ representa la probabilidad de tomar la acción a estando en el estado s. Las políticas estocásticas son particularmente útiles en entornos con dinámicas inciertas o para fomentar la exploración.

3.2.4. Función Valor

En el contexto del aprendizaje por refuerzo, una función de valor asigna a cada estado (o par estado-acción) un valor numérico que representa el retorno esperado, es decir, las recompensas futuras acumuladas que el agente puede esperar obtener, empezando desde ese estado o estado y acción, y siguiendo una política específica. Esta noción de valor es fundamental para que el agente pueda tomar decisiones informadas sobre qué acciones realizar en función de sus objetivos a largo plazo. Existen dos tipos principales de función valor:

La **función valor de estado (v)**, denotada como $v^{\pi}(s)$, mide el retorno esperado para un agente que comienza en un estado s y sigue una política π de ahí en adelante. Matemáticamente, se define como la expectativa del retorno total G_t, que es la suma descontada de todas las recompensas futuras, partiendo desde el estado s bajo la política π. La fórmula para $v^{\pi}(s)$ es la siguiente:

$$v_{\pi}(s) := E_{\pi}[G_t|S_t = s] = E_{\pi}\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s\right]$$

donde γ es el factor de descuento que pondera la importancia de las recompensas futuras. Un valor más bajo de γ da más peso a las recompensas inmediatas en comparación con las futuras, ayudando al agente a enfocarse en beneficios a corto plazo. Refleja el valor a largo plazo de estar en un estado determinado.

La **función valor de acción (q)**, o $q^{\pi}(s, a)$, evalúa el retorno esperado de tomar una acción específica a en un estado s, y luego continuar siguiendo la política π. Esta función es especialmente útil para políticas que requieren una evaluación explícita de las acciones individuales en cada estado. La definición formal de $q^{\pi}(s, a)$ es:

$$q_{\pi}(s, a) := E_{\pi}[G_t|S_t = s, A_t = a] = E_{\pi}\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s, A_t = a\right]$$

donde el término $\mathbb{E}^{\pi}[\cdot]$ representa el valor esperado cuando el agente toma la acción a en el estado s, siguiendo la política π . Esta función es crucial para la toma de decisiones, ya que ayuda al agente a evaluar la efectividad de las acciones individuales para cada estado.

3.2.5. Ecuación de Bellman

La ecuación de Bellman es fundamental en la teoría del aprendizaje por refuerzo ya que se nutre tanto de las funciones de valor de estado como de las de acción. Expresa la relación recursiva entre el valor de un estado (o de una acción en un estado) y el valor de los estados siguientes. Consideramos un proceso de decisión de Markov (MDP) con un conjunto de estados S, un conjunto de acciones A, una función de transición de estados

$p(s', r \mid s, a)$ que determina la probabilidad de transición al estado s' con recompensa r dado que la acción a fue tomada en el estado s, y un factor de descuento γ. La política $\pi(a \mid s)$ representa la probabilidad de tomar la acción a en el estado s bajo la política π.

$$v_\pi(s) = \sum_a \pi(a \mid s) \sum_{s',r} p(s', r \mid s, a) \left[r + \gamma v_\pi(s') \right], \text{ para todo } s \in S,$$

- $\sum_a \pi(a \mid s)$ es la suma sobre todas las acciones posibles desde el estado s bajo la política π. Esto refleja la expectativa con respecto a la política.
- $\sum_{s',r} p(s', r \mid s, a)$ es la suma sobre todos los posibles estados siguientes s' y recompensas r recibidas al tomar la acción a en el estado s. Esto representa la expectativa con respecto a las transiciones de estado y las recompensas.
- $[r + \gamma v_\pi(s')]$ es el retorno esperado al recibir la recompensa inmediata r y el valor descontado del estado siguiente $v_\pi(s')$, ponderado por el factor de descuento γ.

La Ecuación de Bellman establece que el valor de un estado bajo una política es igual a la suma ponderada de los valores esperados de las acciones posibles en ese estado, donde cada valor esperado es la suma de las recompensas esperadas inmediatas más los valores futuros descontados de los estados siguientes según esa política.

Este marco no solo es central para evaluar políticas en MDPs, sino que también es la base para algoritmos de mejora de políticas y métodos de aprendizaje por refuerzo como Q-learning.

3.2.6. Políticas y Funciones de Valor Óptimas

Una **política óptima** π^* es aquella que garantiza el máximo retorno esperado desde cualquier estado inicial en comparación con todas las otras políticas posibles. En términos formales, una política π es óptima si satisface:

$$v_{\pi^*}(s) \geq v_\pi(s), \quad \forall \pi, \forall s \in S,$$

donde $v_\pi(s)$ es la función de valor de estado bajo la política π. Esto implica que la política óptima π^* maximiza la función de valor en cada estado del espacio de estados.

Las **funciones de valor óptimas** en MDPs se presentan en dos formas principales:

1. **Función de Valor de Estado Óptima** $v^*(s)$: Representa el valor máximo que puede ser obtenido desde el estado s bajo cualquier política. Se define como:

   $$v^*(s) = \max_\pi v_\pi(s),$$

   que corresponde al máximo retorno esperado desde el estado s bajo la mejor política posible.

2. **Función de Valor de Acción Óptima** $q^*(s, a)$: Es el valor esperado de tomar una acción a en un estado s y luego seguir la mejor política posible. Se define matemáticamente por:

   $$q^*(s, a) = \mathbb{E}[R_{t+1} + \gamma v^*(S_{t+1}) \mid S_t = s, A_t = a],$$

   donde R_{t+1} es la recompensa recibida después de tomar la acción a en el estado s, y $v^*(S_{t+1})$ es el valor óptimo del nuevo estado.

Las políticas óptimas son teoricamente ideales pero son practicamente imposibles de calcular debido al alto coste computacional debido a la alta dimensionalidad que tienen algunos entornos, ya que el agente tendría que recorrer todas las combinaciones posibles entre estado y acción. En realidad, los agentes frecuentemente deben conformarse con aproximaciones de las políticas óptimas.

3.2.7. Exploración vs. Explotación

Uno de los problemas centrales en el aprendizaje por refuerzo es el equilibrio entre la exploración de nuevas acciones para descubrir recompensas potencialmente mayores y la explotación de las acciones conocidas que ya han dado buenos resultados. Esta sección discute el dilema de exploración vs. explotación y examina diversas estrategias para equilibrar estos dos aspectos cruciales.

El aprendizaje por refuerzo busca optimizar las políticas de decisión de un agente en un entorno incierto. Sin embargo, para maximizar las recompensas a largo plazo, el agente debe equilibrar entre explorar acciones desconocidas y explotar su conocimiento actual del entorno.

Exploración

Se refiere al acto de un agente de probar acciones que no ha seleccionado frecuentemente en el pasado o de las cuales tiene poca información. El propósito de la exploración es descubrir nuevas estrategias que podrían resultar en recompensas más altas. La exploración es crucial especialmente en las etapas iniciales del aprendizaje o en entornos dinámicos donde las recompensas asociadas con las acciones pueden cambiar. Sin una exploración adecuada, un agente puede no descubrir las políticas óptimas o puede quedarse atrapado en una política óptima local.

Explotación

Implica usar el conocimiento acumulado para tomar decisiones que maximicen la recompensa basada en la información actual. Cuando un agente actúa de esta forma, elige las acciones que cree que le ofrecerán la mayor recompensa, basándose en los datos recolectados de experiencias anteriores. La explotación es efectiva para maximizar la recompensa a corto plazo.

El dilema entre exploración y explotación surge porque un agente tiene que decidir entre probar nuevas acciones que podrían llevar a mejores recompensas a futuro (exploración) o utilizar su conocimiento actual para obtener la máxima recompensa inmediata (explotación). La elección entre estas dos opciones es crucial porque una inclinación excesiva hacia la explotación puede hacer que el agente ignore mejores opciones que no ha explorado suficientemente. Por otro lado, demasiada exploración puede resultar en la pérdida de recompensas obtenibles al no aprovechar el conocimiento adquirido sobre las acciones más recompensantes. Diversas estrategias han sido propuestas para equilibrar la exploración y la explotación, cada una con sus ventajas y limitaciones.

Políticas ϵ-greedy La política ϵ-greedy es una estrategia fundamental en la que el agente selecciona la acción que maximiza la recompensa esperada con probabilidad $1 - \epsilon$

y una acción aleatoria con probabilidad ϵ. Matemáticamente, la selección de acción a en el estado s puede representarse como:

$$a = \begin{cases} \arg\max_a Q(s,a) & \text{con probabilidad } 1-\epsilon, \\ \text{acción aleatoria} & \text{con probabilidad } \epsilon. \end{cases}$$

Donde $Q(s,a)$ es la función que estima el valor de tomar la acción a en el estado s. Este enfoque fomenta un balance efectivo entre explorar el entorno y explotar el conocimiento adquirido.

Decaimiento de ϵ Para mejorar la política ϵ-greedy y adaptarla a lo largo del aprendizaje, se puede implementar un decaimiento de ϵ. Este método reduce gradualmente la tasa de exploración con una función como $\epsilon(t) = \epsilon_0 e^{-\lambda t}$, donde ϵ_0 es el valor inicial de ϵ, λ es la tasa de decaimiento, y t es el número de episodios o pasos de tiempo. Esto permite que el agente se enfoque más en explotar las acciones de mayor valor a medida que su conocimiento del entorno se profundiza.

Optimismo ante la Incertidumbre Esta técnica involucra inicializar las estimaciones de recompensa de manera optimista, para motivar la exploración de acciones desconocidas. Si $Q(s,a)$ es la función de valor inicializada como:

$$Q(s,a) = \text{valor alto inicial},$$

el agente explorará diversas acciones hasta que los datos empíricos ajusten las estimaciones hacia sus verdaderos valores. Esta estrategia es particularmente útil para evitar mínimos locales y fomentar un conocimiento más completo del espacio de acciones.

Métodos Basados en Valor En los métodos basados en valor, como el Upper Confidence Bound (UCB), las acciones se seleccionan según:

$$a = \arg\max_a \left(Q(s,a) + c\sqrt{\frac{\log N(s)}{N(s,a)}} \right),$$

donde $Q(s,a)$ es el valor estimado de la acción, $N(s)$ es el número total de veces que se ha visitado el estado s, $N(s,a)$ es el número de veces que se ha tomado la acción a en el estado s, y c es un coeficiente que equilibra la exploración y la explotación. Esta técnica enfoca la exploración hacia acciones cuyo valor es aún incierto.

3.2.8. Ventajas de los Procesos de Decisión de Markov

Modelado Completo de la Incertidumbre: Los MDPs permiten modelar explícitamente la incertidumbre en las transiciones entre estados y las recompensas, lo que es crucial en muchos entornos reales donde el resultado de una acción no puede ser determinado con certeza.

Optimización a Largo Plazo: A través del uso del factor de descuento, los MDPs facilitan la toma de decisiones que equilibran las recompensas a corto y largo plazo, permitiendo al agente aprender estrategias que maximizan las recompensas acumuladas a lo largo del tiempo.

Políticas Óptimas: Los MDPs brindan un marco para encontrar buenas aproximaciones a las políticas óptimas, es decir, estrategias de decisión que maximizan la recompensa esperada para el agente, dada su percepción del entorno.

Flexibilidad y Generalidad: Los MDPs pueden aplicarse a una amplia variedad de problemas de toma de decisiones en campos tan diversos como la robótica, economía, control automático, y más, gracias a su naturaleza general y flexible.

Fundamentación Teórica Sólida: Los MDPs cuentan con una base teórica sólida que proporciona garantías sobre la convergencia y la calidad de las soluciones obtenidas mediante diferentes algoritmos de resolución.

3.2.9. Problemas y Desafíos de los Procesos de Decisión de Markov

El problema de la Dimensionalidad: A medida que el número de estados y acciones aumenta, el espacio de estados se vuelve exponencialmente grande, lo que hace que la resolución de MDPs sea computacionalmente intratable con los métodos tradicionales.

Modelado del Entorno: En muchos casos prácticos, no se conoce completamente el modelo del entorno, es decir, las probabilidades de transición y las funciones de recompensa. Esto requiere el uso de técnicas de aprendizaje por refuerzo que pueden aprender políticas óptimas sin conocimiento previo del modelo, pero estas técnicas enfrentan sus propios desafíos, como la convergencia y la eficiencia del aprendizaje.

Sensibilidad al Factor de Descuento: La elección del factor de descuento γ tiene un impacto significativo en la política óptima aprendida. Un valor inapropiado puede llevar al agente a preferir recompensas inmediatas sobre recompensas futuras más significativas o viceversa.

Presupuestos de Tiempo y Recursos: La resolución de MDPs, especialmente en entornos complejos o de alta dimensión, puede requerir una cantidad significativa de tiempo de computación y memoria, lo que puede ser un obstáculo para aplicaciones en tiempo real o dispositivos con recursos limitados.

3.3– Métodos de Solución en Aprendizaje por Refuerzo

Esta sección se centra en los métodos de solución comúnmente empleados en el aprendizaje por refuerzo para resolver problemas modelados mediante los Procesos de Decisión Markov (MDPs) y optimizar las políticas y funciones de valor de un agente. Se exploran desde tres enfoques principales: la programación dinámica, los métodos de Monte Carlo y el aprendizaje temporal diferencial (TD).

3.3.1. Aprendizaje On-line vs Off-line

El aprendizaje On-line y el aprendizaje Off-line son dos modalidades fundamentales en el campo del machine learning y el aprendizaje por refuerzo. Cada modalidad tiene sus propias características, ventajas y desventajas, que son cruciales para entender su aplicabilidad en diferentes escenarios.

Aprendizaje On-line: El aprendizaje On-line la actualización continua de un modelo o agente a medida que se recibe nueva información. Este enfoque es especialmente útil en entornos que cambian dinámicamente o cuando los datos se generan continuamente. Estas son las características que tiene el aprendizaje On-line:

- **Actualizaciones Continuas:** El modelo se ajusta después de cada nueva interacción o entrada de datos.
- **Adaptabilidad:** Permite una alta adaptabilidad a cambios en el entorno o en los patrones de datos.
- **Requerimientos de Memoria:** Generalmente requiere menos memoria ya que no necesita almacenar grandes volúmenes de datos históricos.

Aprendizaje Off-line: El aprendizaje Off-line se basa en el entrenamiento de un modelo sobre un conjunto de datos predefinido y fijo antes de su implementación en la aplicación real. Estas son las características del aprendizaje Off-line:

- **Uso de Datos Históricos:** Utiliza un conjunto de datos acumulados para entrenar el modelo.
- **Estabilidad:** Los modelos suelen ser más estables ya que se entrenan en un entorno controlado.
- **Riesgo de Sobreajuste:** Existe un mayor riesgo de sobreajuste si el modelo está excesivamente afinado para los datos de entrenamiento. El sobreajuste o también conocido como overfitting es el efecto que se produce cuando se sobreentrena al modelo o no hay suficiente datos pruebas el cual produce que el modelo no sea capaz de generalizar y su rendimiento empeora considerablemente cuando se le pasan datos de entrada distintos a los del entrenamiento.

El aprendizaje On-line y Off-line ofrecen enfoques distintos para el entrenamiento de modelos de machine learning y aprendizaje por refuerzo. La elección entre uno u otro dependerá de las necesidades específicas del proyecto, la naturaleza del entorno de datos y los objetivos de aprendizaje.

3.3.2. On-policy vs Off-policy

En el aprendizaje por refuerzo, los enfoques on-policy y off-policy son fundamentales para la actualización de políticas y la estimación de funciones de valor. Cada enfoque tiene sus propias características, ventajas y desventajas, que son cruciales para su aplicabilidad en diferentes escenarios de aprendizaje.

On-policy: El enfoque on-policy se centra en mejorar la misma política que se utiliza para interactuar con el entorno. Este enfoque es útil cuando se desea optimizar la política actual de manera directa.

Estas son las características del aprendizaje on-policy:

- **Mejora Directa de la Política Actual:** El modelo mejora la política que está siendo seguida actualmente para generar acciones.
- **Consistencia de Datos:** Los datos utilizados para actualizar la política provienen de la política misma, asegurando consistencia.
- **Ejemplos Clásicos:** Métodos como SARSA (State-Action-Reward-State-Action) son ejemplos de algoritmos on-policy.

Off-policy: El enfoque off-policy se basa en aprender una política diferente a la que se utiliza para generar los datos. Esto permite mayor flexibilidad y reutilización de datos históricos o generados por políticas distintas.

Estas son las características del aprendizaje off-policy:

- **Uso de Políticas Diferentes:** El modelo aprende y mejora una política distinta a la utilizada para recolectar datos.
- **Flexibilidad en el Uso de Datos:** Permite utilizar datos generados por diferentes políticas, incluyendo políticas exploratorias o históricas.
- **Ejemplos Clásicos:** Métodos como Q-learning y el algoritmo DQN (Deep Q-Network) son ejemplos de algoritmos off-policy.

La elección entre enfoques on-policy y off-policy depende de las necesidades específicas del proyecto, la naturaleza del entorno de datos y los objetivos de aprendizaje. Ambos enfoques son cruciales en el campo del aprendizaje por refuerzo y ofrecen distintas ventajas que pueden ser explotadas según el contexto del problema a resolver.

3.3.3. Programación Dinámica

La programación dinámica (PD) es una técnica matemática utilizada para resolver problemas de optimización. En el contexto de los MDPs, la PD se emplea cuando se conoce el modelo completo del entorno, es decir, cuando se dispone de la función de transición de estados y la función de recompensa.

La PD se basa en descomponer un problema de decisión complejo en subproblemas más simples, resolviéndolos de manera secuencial. Esta estrategia se alinea con el principio de optimalidad de Bellman, el cual afirma que las políticas óptimas contienen subpolíticas que también son óptimas.

En el aprendizaje por refuerzo, la Programación Dinámica se aplica a través de dos algoritmos fundamentales: la iteración de valor y la iteración de política.

Iteración de Valor: Este algoritmo busca encontrar la función de valor óptima $V^*(s)$ aplicando iterativamente la ecuación de Bellman para el valor hasta que la función converge a un punto fijo.

Iteración de Política: Consiste en alternar entre la evaluación de una política (calcular la función de valor de una política dada) y la mejora de la política (actualizar la política basándose en la función de valor calculada), hasta encontrar una política que sea óptima.

3.3.4. Métodos de Monte Carlo

Los Métodos de Monte Carlo en el aprendizaje por refuerzo se utilizan para estimar funciones de valor y desarrollar políticas sin necesidad de un modelo explícito del entorno. Estos métodos están basados en la experiencia: aprenden directamente de secuencias completas de estados, acciones y recompensas.

Los métodos de Monte Carlo estiman las funciones de valor a partir de los retornos obtenidos al final de los episodios, un episodios es la secuencia de acciones y estados que recorre el agente desde que empieza a interactuar con el entorno hasta que llega a un estado terminal.

En los Métodos de Monte Carlo la función de valor de un estado se estima promediando los retornos obtenidos después de visitas a ese estado. El retorno G_t de un episodio iniciado en el tiempo t se calcula como:

$$G_t = R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{T-t-1} R_T$$

donde R_{t+k} es la recompensa recibida en el paso de tiempo $t+k$, γ es el factor de descuento y T es el paso de tiempo en el que termina el episodio.

Los métodos de Monte Carlo cuentan con las siguientes ventajas y limitaciones:

- **Ventajas:** Los métodos de Monte Carlo son conceptuales y computacionalmente simples, no requieren un modelo del entorno y son efectivos en tareas con episodios bien definidos.
- **Limitaciones:** No son adecuados para problemas continuos sin episodios definidos y pueden ser ineficientes si hay una gran varianza en los valores de las recompensas.

Los métodos de Monte Carlo proporcionan una herramienta poderosa para el aprendizaje por refuerzo, especialmente adecuada para entornos con episodios con una finalización clara. Ofrecen un enfoque directo y fundamentado para estimar funciones de valor y desarrollar políticas sin un modelo previo del entorno.

3.3.5. Aprendizaje por Diferencia Temporal (TD)

El aprendizaje por diferencia temporal es un enfoque en el aprendizaje por refuerzo que permite al agente aprender de la experiencia directa sin requerir un modelo del entorno. Los métodos TD son especialmente conocidos por su capacidad de aprender secuencialmente y de manera incremental.

El aprendizaje TD se basa en la idea de actualizar las estimaciones de valor no solo a partir de las recompensas reales recibidas, sino también a partir de las predicciones futuras (o estimaciones) que hace el modelo, este enfoque es conocido como *bootstrapping*. Esta característica los distingue significativamente de los métodos de Monte Carlo, que esperan hasta el final de un episodio para actualizar las estimaciones basadas en el retorno total real observado.

El algoritmo más simple de TD, conocido como TD(0) o one-step TD, utiliza la siguiente regla de actualización para la función de valor $V(s)$ del estado s:

$$V(s) \leftarrow V(s) + \alpha[r + \gamma V(s') - V(s)],$$

donde r es la recompensa recibida tras la transición del estado s al estado s', γ es el factor de descuento y α es la tasa de aprendizaje. En el aprendizaje TD, las actualizaciones de los valores estimados de los estados o de las acciones se realizan en cada transición o step (una transición es cuando el agente realiza un acción y transiciona al nuevo estado), después de cada transición del entorno.

El aprendizaje por diferencia por temporal ofrece las siguientes ventajas:

- Los métodos TD pueden aprender antes de conocer el resultado final de un episodio.
- No necesitan el modelo completo del entorno y son particularmente útiles en entornos no deterministas o donde el modelo es difícil de obtener.

3.3.6. Q-learning

Q-learning es un método de aprendizaje por refuerzo que busca determinar la función de valor óptima para un agente que actúa en un entorno determinístico o estocástico. Uno de los aspectos más destacados de Q-learning es su capacidad para aprender a partir de las transiciones de estado-acción sin requerir un modelo del entorno y su política independiente, lo que le permite evaluar la política óptima mientras sigue una política de exploración.

Tabla Q

La función de valor de acción en Q-Learning, denotada como $Q(s, a)$, refleja el valor esperado de tomar una acción a en un estado s y seguir la política actual. Q-Learning mantiene una tabla conocida como tabla Q, que almacena y actualiza las estimaciones para cada par estado-acción, $Q(S, A)$.

Función Valor

La actualización de Q-learning se realiza según la siguiente fórmula:

$$Q(s, a) \leftarrow Q(s, a) + \alpha \left[R + \gamma \max_{a'} Q(s', a') - Q(s, a) \right]$$

donde:

- s es el estado actual.
- a es la acción tomada en el estado s.
- s' es el estado siguiente después de tomar la acción a.
- R es la recompensa recibida después de moverse a s'.
- α es la tasa de aprendizaje.
- γ es el factor de descuento.
- $\max_{a'} Q(s', a')$ representa el valor máximo de Q para el próximo estado

Dinámica Aprendizaje

- **Inicialización**: Todos los valores $Q(s, a)$ se inicializan, comúnmente a cero o a pequeñas variaciones aleatorias.
- **Interacción con el Entorno**: El agente toma acciones en el entorno basadas en la política derivada de Q, típicamente utilizando una estrategia ϵ-greedy para equilibrar entre exploración y explotación.
- **Actualización de** Q: Después de cada acción tomada y la recompensa recibida, $Q(s, a)$ se actualiza usando la fórmula de actualización.
- **Repetición**: Este proceso se repite para múltiples episodios hasta que Q converge hacia Q^*.

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

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

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

Figura 3.2: Pseudocódigo Q-learning [16]

Desafíos

- **Espacios de Estado Grandes**: En entornos con espacios de estado grandes o acciones numerosas, Q-learning puede enfrentarse a dificultades prácticas debido a la necesidad de estimar valores para cada par estado-acción, lo que puede llevar a una demanda de memoria y tiempo computacionalmente prohibitiva.
- **Riesgo de Divergencia**: En algunos casos, especialmente si las condiciones para la tasa de aprendizaje o la política de exploración no son adecuadas, Q-learning puede no converger o incluso diverger.
- **Dependencia de ϵ-greedy**: La eficacia de Q-learning puede depender en gran medida de cómo se maneja la exploración, lo que puede no ser óptimo en todos los entornos.

3.3.7. Double Q-learning

Double Q-learning es una variante del algoritmo Q-learning que busca mitigar el problema de la sobreestimación de los valores Q, un problema común en el Q-learning estándar. Este método divide la estimación de los valores Q en dos funciones de valor, lo que permite una evaluación más precisa y estable.

Tabla Q Doble

Double Q-learning mantiene dos tablas de valores de acción, $Q_A(s, a)$ y $Q_B(s, a)$, en lugar de una sola tabla Q. Cada tabla se actualiza de manera alternada, lo que ayuda a reducir el sesgo de sobreestimación.

Función Valor

La actualización en Double Q-learning se realiza usando dos tablas de Q de la siguiente manera:

$$\begin{cases} Q_A(s,a) \leftarrow Q_A(s,a) + \alpha\left[R + \gamma Q_B(s', \arg\max_{a'} Q_A(s',a')) - Q_A(s,a)\right] & \text{con probabilidad } 0.5 \\ Q_B(s,a) \leftarrow Q_B(s,a) + \alpha\left[R + \gamma Q_A(s', \arg\max_{a'} Q_B(s',a')) - Q_B(s,a)\right] & \text{con probabilidad } 0.5 \end{cases}$$

donde:

Algorithm 1 Double Q-learning

Initialize Q^A,Q^B,s
repeat
  Choose a, based on $Q^A(s,\cdot)$ and $Q^B(s,\cdot)$, observe r, s'
  Choose (e.g. random) either UPDATE(A) or UPDATE(B)
  if UPDATE(A) **then**
    Define $a^* = \arg\max_a Q^A(s',a)$
    $Q^A(s,a) \leftarrow Q^A(s,a) + \alpha(s,a)\left(r + \gamma Q^B(s',a^*) - Q^A(s,a)\right)$
  else if UPDATE(B) **then**
    Define $b^* = \arg\max_a Q^B(s',a)$
    $Q^B(s,a) \leftarrow Q^B(s,a) + \alpha(s,a)(r + \gamma Q^A(s',b^*) - Q^B(s,a))$
  end if
  $s \leftarrow s'$
until end

Figura 3.3: Pseudocódigo Double Q-learning [13]

- s es el estado actual.
- a es la acción tomada en el estado s.
- s' es el estado siguiente después de tomar la acción a.
- R es la recompensa recibida después de moverse a s'.
- α es la tasa de aprendizaje.
- γ es el factor de descuento.
- $\arg\max_{a'} Q_A(s',a')$ representa la acción que maximiza el valor de Q_A en el próximo estado.
- $\arg\max_{a'} Q_B(s',a')$ representa la acción que maximiza el valor de Q_B en el próximo estado.

Dinámica del Aprendizaje

- **Inicialización**: Todos los valores $Q_A(s,a)$ y $Q_B(s,a)$ se inicializan, comúnmente a cero o a pequeñas variaciones aleatorias.
- **Interacción con el Entorno**: El agente toma acciones en el entorno basadas en la política derivada de la suma de Q_A y Q_B, típicamente utilizando una estrategia ϵ-greedy para equilibrar entre exploración y explotación.
- **Actualización de Q_A y Q_B**: Después de cada acción tomada y la recompensa recibida, se selecciona una de las tablas Q_A o Q_B para actualizarse de acuerdo con la fórmula de actualización.
- **Repetición**: Este proceso se repite para múltiples episodios hasta que Q_A y Q_B convergen hacia una estimación precisa de Q^*.

Desafíos

- **Complejidad Computacional**: El mantenimiento de dos tablas de valores Q aumenta los requerimientos de memoria y puede incrementar la complejidad computacional.
- **Equilibrio en la Exploración**: Al igual que en Q-learning, la efectividad de Double Q-learning puede depender en gran medida de cómo se maneja la exploración, lo que puede no ser óptimo en todos los entornos.
- **Implementación**: La implementación es más compleja en comparación con Q-learning estándar debido a la necesidad de gestionar dos tablas de valores Q y alternar su actualización.

3.3.8. SARSA

SARSA, que significa State-Action-Reward-State-Action, es un algoritmo de aprendizaje por refuerzo que actualiza las estimaciones de los valores de las acciones directamente con base en las acciones que toma el agente. A diferencia de Q-learning, SARSA aprende la función de valor de la política que sigue actualmente el agente, incluyendo las decisiones tomadas durante la exploración.

Función Valor

La función de valor de acción, denotada como $Q(s, a)$, representa el valor esperado de iniciar en el estado s, tomar la acción a, y luego seguir la política actual del agente, el algoritmo de SARSA también cuenta con Q-table como Q-Learning. La actualización de SARSA se realiza según la siguiente fórmula de aprendizaje:

$$Q(S, A) \leftarrow Q(S, A) + \alpha \left[R + \gamma Q(S', A') - Q(S, A)\right]$$

donde:

- S es el estado actual.
- A es la acción tomada en el estado s.
- S' es el estado siguiente tras la acción a.
- A' es la próxima acción elegida según la política actual.
- R es la recompensa recibida tras moverse a s'.
- α es la tasa de aprendizaje.
- γ es el factor de descuento.

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

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

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

Figura 3.4: Pseudocódigo SARSA [16]

Ciclo de Aprendizaje

- **Inicialización**: Todos los valores $Q(s, a)$ se inicializan, comúnmente a cero o a pequeñas variaciones aleatorias.
- **Interacción con el Entorno**: El agente toma acciones en el entorno basadas en la política derivada de Q, típicamente utilizando una estrategia ϵ-greedy para equilibrar entre exploración y explotación.
- **Actualización de Q**: Después de cada acción y la recompensa recibida, junto con la elección de la próxima acción a' según la política actual, $Q(s, a)$ se actualiza usando la fórmula de actualización.
- **Repetición**: Este proceso se repite para múltiples episodios hasta que Q converge hacia la política óptima bajo la política evaluada.

Desafíos

- **Dependencia de la Política de Exploración**: SARSA puede ser más susceptible a los problemas derivados de una política de exploración inadecuada, ya que aprende una política que incluye las decisiones de exploración.
- **Riesgo de Converger a una Política Subóptima**: Si la exploración no es suficiente, existe un riesgo mayor de converger a una política subóptima.

3.4– Desafíos y Tendencias en Aprendizaje por Refuerzo

3.4.1. Desafíos en el Aprendizaje por Refuerzo

El aprendizaje por refuerzo ha demostrado ser una herramienta poderosa para enseñar a los agentes cómo tomar decisiones. Sin embargo, la implementación de algoritmos de RL en la práctica enfrenta varios desafíos que pueden afectar significativamente su efectividad.

Elección de Parámetros La configuración de los parámetros, el factor de descuento, la política de exploración y demás hiperparámetros, es crucial para el rendimiento del agente de RL. Una elección inapropiada puede llevar a un aprendizaje lento o incluso a la convergencia a políticas subóptimas.

Representación del Estado La forma en que se representa el estado del entorno puede tener un impacto significativo en la capacidad del agente para aprender. Una representación inadecuada puede ocultar relaciones importantes o hacer que el espacio de estados sea innecesariamente complejo, dificultando el aprendizaje.

Escalabilidad A medida que aumenta la complejidad del entorno, también lo hace el espacio de estados y acciones, lo que puede hacer que los algoritmos de aprendizaje por refuerzo se vuelvan impracticables debido a los requisitos computacionales. La escalabilidad sigue siendo un desafío importante en la implementación de RL en problemas del mundo real.

3.4.2. Tendencias en el Aprendizaje por Refuerzo

La investigación en el aprendizaje por refuerzo continúa avanzando rápidamente, impulsando nuevas aplicaciones y mejorando las técnicas existentes.

Integración con Aprendizaje Profundo

El Aprendizaje por Refuerzo Profundo (Deep RL) está en la vanguardia de la investigación, permitiendo a los agentes aprender de entradas de alta dimensionalidad como imágenes y resolver tareas que antes eran inaccesibles para los algoritmos de RL tradicionales. Se utilizan redes neuronales para estimar las funciones valor y políticas que ayudan a controlar los espacios con una gran cantidad de estados.

RL en Entornos Complejos del Mundo Real

La aplicación de RL en entornos complejos del mundo real, como la robótica, los vehículos autónomos y la gestión de recursos, es una tendencia creciente. Estas aplicaciones plantean nuevos desafíos, incluida la necesidad de seguridad y robustez en las decisiones del agente.

Exploración Eficiente

Desarrollar métodos para una exploración más eficiente del espacio de acciones sigue siendo un área de investigación activa. Esto incluye técnicas para equilibrar mejor la exploración y la explotación, así como métodos para dirigir la exploración hacia áreas más prometedoras del espacio de estado.

Generalización y Aprendizaje por Transferencia

Mejorar la capacidad de los agentes de RL para generalizar su aprendizaje a nuevas tareas o entornos y transferir conocimientos entre tareas está ganando atención. Esto podría permitir que los sistemas de RL se adapten más rápidamente a nuevas situaciones con menos datos de entrenamiento. Transferir conocimientos preciamente aprendidos por

los modelos de aprendizaje por refuerzo puede reducir significamentente el tiempo y los datos necesarios para el aprendizaje.

Interacción Humano-Agente

La interacción entre humanos y agentes de RL es un área emergente que explora cómo los agentes pueden aprender de las interacciones con los humanos, ya sea a través de la demostración, el refuerzo proporcionado por los humanos o la colaboración directa.

CAPÍTULO 4

Preparación del entorno

Este capítulo describe el funcionamiento de PyTorch y varios ejemplos prácticos para ayudar a tener un mejor entendimiento de como construir una red neuronal y entrenar un modelo, una pequeña descripción de las principales funciones que ofrece de gymnasium y cómo funciona y por último, las técnicas de preprocesamiento y preparación del entorno de Ms. Pac-Man implementadas para mejorar el proceso de aprendizaje por refuerzo utilizando el algoritmo Rainbow DQN.

4.1– PyTorch

PyTorch es una biblioteca de aprendizaje profundo que proporciona flexibilidad y velocidad durante el desarrollo de modelos de inteligencia artificial. Es popular entre los investigadores por su interfaz intuitiva y eficiencia en la ejecución de cálculos tensoriales complejos, especialmente en la GPU.

4.1.1. Módulos y Funciones Clave de PyTorch

torch.Tensor En PyTorch, un **tensor** es una estructura de datos similar a los arrays multidimensionales. Es el bloque fundamental para todas las operaciones de computación en PyTorch, diseñado para operar de manera eficiente tanto en CPU como en GPU. Los tensores soportan una amplia gama de operaciones, incluyendo aritméticas y lógicas, que son esenciales para la construcción y el entrenamiento de modelos de aprendizaje profundo. Esta estructura de datos no solo facilita el almacenamiento y la manipulación de datos de alta dimensionalidad sino que también permite la diferenciación automática, crucial para optimizar redes neuronales.

Autograd PyTorch utiliza un módulo llamado `autograd` para automatizar el calculo del gradiente. Este módulo rastrea todas las operaciones realizadas en los tensores y construye un gráfico de cálculo que facilita la diferenciación automática.

torch.nn y torch.nn.functional El módulo `torch.nn` contiene clases que facilitan la construcción de redes, como `nn.Conv2d` para capas convolucionales y `nn.ReLU` para fun-

ciones de activación no lineales. `torch.nn.functional` proporciona versiones funcionales de estas mismas operaciones.

torch.optim Este módulo gestiona varios algoritmos de optimización, como SGD y Adam, fundamentales para actualizar los parámetros de las redes neuronales durante el entrenamiento.

torch.no_grad Se usa para indicar que las operaciones dentro de su bloque no requieren cálculos de gradientes, lo cual es útil durante la evaluación del modelo para mejorar la eficiencia.

torch.save/torch.load Estas funciones se utilizan para guardar y cargar modelos de PyTorch, respectivamente, facilitando la reutilización de modelos entrenados y la interrupción y reanudación del entrenamiento.

4.1.2. Ejemplo: Configuración de una Red Neuronal Simple

Para ilustrar cómo se configura una red neuronal en PyTorch, consideremos una red convolucional simple denominada `SimpleCNN`, diseñada para clasificar imágenes. Esta red incluye una combinación de capas convolucionales y lineales intercaladas con funciones de activación ReLU.

```
import torch
import torch.nn as nn
import torch.optim as optim

class SimpleCNN(nn.Module):
    def __init__(self):
        super(SimpleCNN, self).__init__()
        # Definición de la primera capa convolucional
        self.conv1 = nn.Conv2d(1, 20, 5, stride=1, padding=0)
        #  (1 canal de entrada, 20 canales de salida, kernel de tamao 5)
        self.relu1 = nn.ReLU()

        # Definición de la segunda capa convolucional
        self.conv2 = nn.Conv2d(20, 50, 5, stride=1, padding=0)
        # (20 canales de entrada, 50 canales de salida, kernel de tamao 5)
        self.relu2 = nn.ReLU()

        # Capa de aplanamiento para conectar la salida convolucional a las capas
            densas
        # Capa lineal que conecta las características aplanadas a 500 nodos
        self.fc1 = nn.Linear(50 * 4 * 4, 500)
        self.relu3 = nn.ReLU()

        # Capa lineal de salida que mapea los 500 nodos a 10 categorías de clase
        self.fc2 = nn.Linear(500, 10)

    def forward(self, x):
        # Aplicar la primera capa convolucional seguida de ReLU
        x = self.relu1(self.conv1(x))
```

```
29
30          # Aplicar la segunda capa convolucional seguida de ReLU
31          x = self.relu2(self.conv2(x))
32
33          # Aplanar la salida de las capas convolucionales para alimentar las capas
                lineales
34          x = x.view(-1, 50 * 4 * 4)
35
36          # Aplicar la primera capa lineal seguida de ReLU
37          x = self.relu3(self.fc1(x))
38
39          # Aplicar la capa lineal de salida
40          x = self.fc2(x)
41
42          return x
```

Capas Conv1 y Conv2 Estas capas son fundamentales para procesar las imágenes de entrada. Utilizan filtros para transformar estas imágenes en mapas de características que capturan información espacial crucial. Los parámetros clave de estas capas son:

- `in_channels`: Número de canales en la imagen de entrada.
- `out_channels`: Número de filtros aplicados, que determina la profundidad del mapa de características de salida.
- `kernel_size`: Tamaño de cada filtro, que en este caso es 5x5.
- `stride`: Número de píxeles que el filtro se mueve a través de la imagen. Un paso de 1 significa que el filtro se mueve un píxel a la vez.
- `padding`: Cantidad de píxeles añadidos al borde de la imagen para permitir que el filtro cubra completamente los bordes de la imagen.

Función ReLU Esta función de activación se utiliza para introducir no linealidades en el modelo, permitiendo a la red aprender patrones más complejos. ReLU convierte todos los valores negativos a cero y mantiene los valores positivos.

Capas Lineales (Fully Connected Layers) Estas capas transforman las características aprendidas por las capas convolucionales en predicciones finales. Detalles importantes incluyen:

- La primera capa lineal toma la salida aplanada de las capas convolucionales y la conecta a 500 nodos.
- La capa final lineal mapea estos 500 nodos a las 10 clases de salida, correspondientes a las categorías que la red está diseñada para clasificar.

Función `forward` Esta función es crucial ya que define el flujo de datos a través de la red. Especifica cómo las entradas pasan secuencialmente a través de las capas convolucionales, ReLU y lineales para producir la salida de la red. La secuencia de operaciones es vital para asegurar que los datos se procesen correctamente en cada etapa del modelo.

Resumen de la Arquitectura

- Dos capas convolucionales, cada una seguida por una función de activación ReLU.
- Dos capas lineales, con una función ReLU entre ellas para introducir no linealidades.
- La función `forward` define explícitamente el flujo de datos a través de la red, esencial para la correcta propagación y ajuste de los parámetros durante el entrenamiento.

4.1.3. Ejemplo: Entrenamiento del Modelo

Una vez que la red está definida, el siguiente paso es entrenarla utilizando un conjunto de datos. Este proceso implica la ejecución de un ciclo de entrenamiento que incluye la evaluación de predicciones, el cálculo de errores y la actualización de parámetros.

Configuración del Modelo y Optimización Inicialmente, se crea una instancia del modelo y se configuran la función de pérdida y el optimizador. Utilizamos la función de pérdida de entropía cruzada, que es común para las tareas de clasificación, y el optimizador Adam, conocido por su eficiencia en diversos escenarios de entrenamiento.

```
model = SimpleCNN()
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

Optimizador Adam Adam, cuyo nombre significa *Adaptive Moment Estimation*, es un algoritmo de optimización avanzado que combina las fortalezas de dos otros optimizadores extendidos del gradiente descendente estocástico: AdaGrad y RMSProp. Adam ajusta la tasa de aprendizaje de cada parámetro del modelo individualmente. Esto se logra mediante la estimación adaptativa de las medias y las varianzas de los gradientes.

La actualización de cada parámetro en Adam se realiza según la siguiente fórmula matemática:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon}\hat{m}_t$$

donde:

- θ_t es el parámetro en el tiempo t.
- η es la tasa de aprendizaje.
- $\hat{m}_t$ y $\hat{v}_t$ son estimaciones de las media y la varianza de los gradientes en el momento t respectivamente.
- ϵ es un pequeño número para evitar la división por cero, típicamente 10^{-8}.

El término η, o tasa de aprendizaje, controla cuánto se modifican los parámetros del modelo en respuesta al gradiente observado en un paso de tiempo. Por ejemplo, una tasa de aprendizaje de 0.001, como se indica en `lr=0.001`, implica que los parámetros del modelo se ajustan a un ritmo moderado, equilibrando la velocidad de aprendizaje y la estabilidad del entrenamiento. Este valor debe ser lo suficientemente pequeño para permitir un aprendizaje gradual y evitar saltos excesivos que puedan llevar a la divergencia del entrenamiento.

Cálculo de la Pérdida y Optimización de Parámetros

Batch En el aprendizaje profundo, un *batch* o lote se refiere a un conjunto de ejemplos de entrenamiento utilizados en una única iteración del proceso de entrenamiento para actualizar los parámetros del modelo. Un batch es una parte del conjunto de datos total, y su tamaño, conocido como *batch size*, es un hiperparámetro crucial que influye en la estabilidad y velocidad del entrenamiento. Un tamaño de batch más grande proporciona una estimación más precisa del gradiente, pero también requiere más memoria y puede llevar a una convergencia más lenta. Por el contrario, un tamaño de batch pequeño puede hacer que el entrenamiento sea más ruidoso, con actualizaciones de parámetros más erráticas, pero a menudo puede acelerar la convergencia y ayudar a escapar de mínimos locales en la superficie de error.

Para cada lote de imágenes y etiquetas:

- Se calcula la salida del modelo (predicciones).
- Se evalúa la pérdida comparando las predicciones con las etiquetas reales.
- Se reestablecen los gradientes acumulados de pasos anteriores.
- Se utiliza la retropropagación para calcular los gradientes de la pérdida respecto a los parámetros del modelo.
- Finalmente, se actualizan los parámetros del modelo utilizando los gradientes calculados.

```
outputs = model(images)
loss = criterion(outputs, labels)
optimizer.zero_grad() # Limpiar los gradientes antiguos
loss.backward()      # Calcular nuevos gradientes
optimizer.step()     # Actualizar los pesos
```

Descripción del Proceso

- `outputs = model(images)`: El modelo genera predicciones para el lote actual de imágenes.
- `loss = criterion(outputs, labels)`: Se calcula la pérdida comparando las predicciones con las etiquetas verdaderas.
- `optimizer.zero_grad()`: Se eliminan todos los gradientes acumulados del optimizador antes de iniciar un nuevo cálculo de gradiente, lo que es crucial para evitar la acumulación de gradientes de múltiples pasos de retropropagación.
- `loss.backward()`: Este método calcula los gradientes de la pérdida con respecto a todos los parámetros del modelo que son diferenciables.
- `optimizer.step()`: Actualiza los parámetros del modelo según los gradientes calculados y la tasa de aprendizaje configurada en el optimizador.

Este ciclo se repite para cada lote de datos durante el entrenamiento, lo que se conoce como un `epoch`. Este proceso iterativo permite que el modelo aprenda gradualmente ajustando sus pesos para minimizar la función de pérdida.

4.2– Gymnasium

Gymnasium, anteriormente conocido como Gym, es una biblioteca de código abierto desarrollada por OpenAI que proporciona una colección estándar de entornos de simulación para experimentar y desarrollar algoritmos de aprendizaje por refuerzo. El objetivo principal de Gymnasium es ofrecer un conjunto de interfaces comunes para una variedad de entornos, desde simples problemas de control hasta juegos complejos, facilitando así la comparación y desarrollo de nuevos algoritmos de inteligencia artificial.

4.2.1. Entornos

Gymnasium define los entornos como espacios donde los agentes pueden interactuar. Cada entorno en Gymnasium tiene un conjunto específico de acciones posibles que un agente puede ejecutar y estados que puede observar. Estos entornos están organizados en diversas categorías, que incluyen:

- **Clásicos:** Ejemplos típicos incluyen CartPole, MountainCar, entre otros. Estos son problemas de control simples diseñados para enseñar los fundamentos del aprendizaje por refuerzo.
- **Juegos:** Incluye una variedad de juegos de Atari, así como juegos de mesa como ajedrez y go. Estos entornos son útiles para desarrollar y probar algoritmos que pueden manejar complejidades tácticas y estratégicas.
- **Simulaciones de Robots:** Utilizan plataformas como MuJoCo para simular entornos físicos detallados donde los algoritmos pueden aprender a controlar movimientos y tareas de robots con gran precisión.

Estos entornos se utilizan como bancos de pruebas para evaluar y comparar el desempeño de diferentes técnicas y algoritmos de aprendizaje por refuerzo. Al proporcionar un conjunto estandarizado de desafíos, Gymnasium ayuda a los investigadores a identificar las fortalezas y debilidades de sus enfoques en una variedad de situaciones complejas y controladas.

4.2.2. Espacio de Acciones y Observaciones

Espacio de Acciones

Los espacios de acción en Gymnasium definen el conjunto de todas las acciones posibles que un agente puede tomar en un entorno. Este espacio puede configurarse de dos maneras principales, reflejando las necesidades del entorno y la naturaleza de las tareas que se van a realizar:

- **Discreto:** Un espacio de acción discreto incluye un número fijo de acciones no relacionadas entre sí. Este tipo es común en entornos donde las acciones representan decisiones claramente definidas, como moverse en una dirección específica o seleccionar un objeto. Ejemplos de entornos con espacios de acción discreta incluyen juegos de Atari donde las acciones podrían ser moverse a la izquierda, a la derecha o disparar.

- **Continuo:** En un espacio de acción continuo, las acciones implican la selección de valores numéricos, que pueden representar cantidades como fuerzas, velocidades o cambios angulares. Este tipo es típico en simulaciones de robots o en entornos donde es necesario un control preciso y gradual, como en la simulación de vehículos o brazos robóticos.

Espacio de Observaciones

Los espacios de observación describen el formato y rango de todos los estados posibles que un agente puede observar en respuesta a sus acciones dentro del entorno. Estos espacios son críticos porque definen cómo el agente percibe su mundo y pueden variar significativamente en dimensiones y tipos:

- **Vectores Simples:** En muchos casos, las observaciones son vectores de números reales que representan medidas cuantitativas del entorno, como la posición, la velocidad o el ángulo de elementos dentro del entorno.
- **Imágenes en 2D:** Para tareas más complejas, especialmente en entornos que involucran la navegación o el reconocimiento visual, las observaciones pueden ser imágenes en 2D. Estas proporcionan una representación visual rica y detallada del entorno, permitiendo algoritmos que emplean técnicas de visión por computadora para interpretar y actuar en consecuencia.

La configuración de estos espacios es fundamental, ya que influyen directamente en el diseño del agente de aprendizaje. Deben ser elegidos cuidadosamente para alinear las capacidades del agente con los desafíos del entorno, facilitando un aprendizaje efectivo y eficiente.

Métodos Fundamentales

En Gymnasium, los métodos fundamentales proporcionan la interfaz básica entre un agente y el entorno. Estos métodos permiten al agente interactuar y recibir retroalimentación del entorno de manera estructurada:

- **reset():** Este método reinicia el estado del entorno a una configuración inicial, devolviendo la primera observación del nuevo episodio. Es esencial para comenzar nuevos episodios y para garantizar que el entorno esté en un estado conocido y controlable antes de que el agente comience a tomar decisiones.
- **step(action):** Aplica una acción seleccionada por el agente al entorno y retorna una tupla de cinco elementos:
    1. **Observación:** La nueva observación del estado del entorno después de que la acción ha sido ejecutada.
    2. **Recompensa:** Un valor numérico que evalúa la efectividad de la acción tomada, guiando al agente en el aprendizaje de qué acciones maximizan el rendimiento.
    3. **Done:** Un indicador booleano de si el episodio ha terminado, ya sea por alcanzar un estado terminal del entorno o por cumplir con un criterio de finalización específico. item

4. **Truncated:** Un indicador booleano que refleja si el episodio fue terminado debido a un truncamiento, como un límite de tiempo, sin alcanzar un estado terminal.
5. **Info:** Un diccionario con información adicional útil para la depuración o para proporcionar detalles adicionales que no afectan directamente la dinámica del juego.

- **render():** Este método genera una representación visual del entorno para la observación humana, que puede ser útil tanto para la depuración como para la presentación del comportamiento del entorno y del agente en tiempo real.

Uno de los mayores beneficios de usar Gymnasium es su flexibilidad. Los investigadores pueden modificar entornos existentes o crear nuevos para satisfacer necesidades específicas. Esto se facilita mediante el uso de `wrappers`, que permiten modificar comportamientos de entornos, como la percepción de las observaciones o la escala de recompensas, sin alterar el código base.

4.3– Wrappers de Gymnasium

Los *wrappers* de Gymnasium son herramientas de extensión para los entornos de Gymnasium que permiten modificar y personalizar el comportamiento del entorno de formas específicas sin alterar el código base del entorno. Estas modificaciones pueden influir en las observaciones, las recompensas, las acciones o la configuración general del entorno, con el objetivo de adaptarlo mejor a las necesidades de un algoritmo de aprendizaje por refuerzo específico o para mejorar la eficiencia del entrenamiento. Los wrapper que se verán mas adelante han sido desarrollados para ajustarlos a las necesidades especificas del entorno de Ms Pacman, siguiendo la documentación que ofrece OpenAI.

4.3.1. Funcionalidad de los Wrappers

Los *wrappers* permiten una amplia gama de modificaciones, tales como:

- **Preprocesamiento de las observaciones:** Esto puede incluir la redimensión de imágenes, la conversión a escala de grises, la normalización de píxeles, o el apilamiento de múltiples frames para dar contexto temporal al agente.
- **Modificación de las recompensas:** Para hacer las recompensas más consistentes o ajustar su escala, facilitando así la convergencia durante el aprendizaje.
- **Alteración de las acciones:** Como limitar el conjunto de acciones disponibles o cambiar la frecuencia con la que se toman las acciones, para simplificar el espacio de acción o introducir una estrategia de decisión temporal.
- **Gestión del ciclo de vida del episodio:** Añadir lógica adicional al inicio o al final de cada episodio, por ejemplo, para restablecer el estado del entorno de manera más controlada.

Implementación

La implementación de un *wrapper* implica extender una de las clases base proporcionadas por Gymnasium y sobrescribir los métodos correspondientes al aspecto del entorno que se desea modificar. Por ejemplo, para cambiar cómo se procesan las observaciones, se extendería la clase `ObservationWrapper` y se sobrescribiría el método `observation(obs)`.

Aplicación en el Aprendizaje por Refuerzo

En la práctica, el uso de *wrappers* permite a los investigadores y desarrolladores experimentar con diferentes configuraciones de entornos de manera rápida y flexible, probar nuevas ideas o técnicas de preprocesamiento, y ajustar los entornos a las necesidades específicas de sus algoritmos de aprendizaje por refuerzo, como en el caso de aplicar Rainbow DQN al juego de Ms. Pac-Man.

Entorno Ms. Pac-Man de Gymnasium El entorno Ms. Pac-Man de Gymnasium es una simulación del popular juego de arcade Ms. Pac-Man, que se utiliza ampliamente como un desafío de prueba para algoritmos de aprendizaje por refuerzo. En este entorno, el agente controla a Ms. Pac-Man, con el objetivo de comer todas las píldoras del laberinto mientras evita ser capturado por los fantasmas. Cada nivel del juego se vuelve progresivamente más difícil a medida que los fantasmas aumentan su velocidad y su inteligencia. Este entorno proporciona un espacio de estado complejo y dinámico, ideal para probar y desarrollar algoritmos avanzados de aprendizaje por refuerzo que requieren toma de decisiones en tiempo real y estrategias adaptativas.

4.3.2. SkipFramesEnv

La implementación de `SkipFramesEnv` permite al agente experimentar una versión del entorno donde las decisiones se toman en intervalos de tiempo más espaciados, reduciendo el número total de decisiones necesarias a lo largo de un episodio y, por lo tanto, el tiempo de entrenamiento. En el entorno de Ms. Pac-Man de Gymnasium, un *episodio* comienza cuando el juego se inicia con Ms. Pac-Man en su posición de partida y termina cuando se cumplen ciertas condiciones, tales como perder todas las vidas o completar el nivel al comer todas las píldoras y frutas presentes en el laberinto.

Constructor

El constructor inicializa la envoltura con el entorno base y el número de frames a saltar.

```
1 class SkipFramesEnv(gym.Wrapper):
2     def __init__(self,env,skip_frames):
3         super().__init__(env)
4         # Buffer para almacenar las observaciones de los últimos frames
5         self.frame_buffer = deque(maxlen=skip_frames)
6         # Número de frames a omitir
```

Método step

El método `step` en este `wrapper` juega un papel crucial en la manipulación del entorno de aprendizaje por refuerzo para adaptarse a las necesidades específicas del algoritmo de

entrenamiento. Este método toma una única acción como entrada y la aplica repetidamente al entorno subyacente, un número de veces especificado por el atributo `self._skip`. En cada aplicación de la acción, se observan y acumulan varios resultados:

- **Observaciones:** Cada paso devuelve una observación del estado actual del entorno, que se almacena en `self.frame_buffer`.
- **Recompensas:** Las recompensas obtenidas en cada paso se acumulan en la variable `accumulated_reward`. Esta acumulación permite que el método devuelva una recompensa total que refleja el resultado de múltiples acciones aplicadas, en lugar de solo el resultado de una.
- **Estado de Finalización:** La variable `done` indica si el entorno ha llegado a un estado terminal. Si en cualquier paso el entorno indica que se ha terminado (`is_done` es verdadero), el bucle se rompe anticipadamente para evitar acciones adicionales en un entorno ya concluido.
- **Maximización de frames:** Después de aplicar todas las acciones, se selecciona el frame máximo de los almacenados en `self.frame_buffer` usando `np.max` sobre el eje especificado. Este paso es crucial porque puede ayudar a evitar problemas visuales causados por el parpadeo de elementos en juegos clásicos, consolidando las características visuales más relevantes de los cuadros observados en un solo cuadro representativo.

Finalmente, el método `step` devuelve el frame máximo, la recompensa total acumulada, y el estado de terminación junto con cualquier información adicional proporcionada por el entorno en `info`.

```
1   def step (self,action):
2       accumulated_reward = 0.0
3       is_done = None
4       # Ejecuta la acción y acumula las recompensas y observaciones por el nú
            mero de frames especificado
5       for _ in range(self.skip_frames ):
6           obs,reward,done,truncated,info = self.env.step(action)
7           self.frame_buffer .append(obs)
8           accumulated_reward += reward
9           if done:
10              is_done = True
11              break
12      # Toma el máximo valor de los frames almacenados para la observación
            actual
13      max_frame = np.max(np.stack(self.frame_buffer ),axis=0)
14      return max_frame, accumulated_reward, is_done ,truncated,info
```

Propósito

Este enfoque tiene múltiples beneficios:

- Reduce la carga computacional al disminuir la frecuencia de las decisiones tomadas por el agente.

- Ayuda a evitar que el agente se atasque en estados de alta frecuencia pero de baja relevancia, al enfocar el aprendizaje en cambios más significativos en el entorno.
- Puede mejorar la estabilidad del aprendizaje al reducir la variabilidad de las observaciones consecutivas.

4.3.3. ResizeFrame

El método `ResizeFrame` es un *wrapper* de Gymnasium diseñado para preprocesar las observaciones del entorno de Ms. Pac-Man redimensionando las imágenes a un tamaño estándar. Esta normalización es crucial para el entrenamiento de redes neuronales, ya que proporciona consistencia en las dimensiones de entrada al modelo.

Implementación

La clase `ResizeFrame` permite modificar las observaciones provenientes del entorno. Su principal funcionalidad se centra en ajustar el tamaño de los frames, en nuestro caso particular a 84×84 píxeles, un tamaño comúnmente utilizado en el procesamiento de imágenes para aprendizaje profundo en tareas de visión por ordenador aplicadas al aprendizaje por refuerzo.

Proceso de Redimensionamiento

- **Espacio de Observación:** Se redefine el espacio de observación para reflejar el nuevo tamaño de las imágenes, utilizando `gym.spaces.Box` para especificar los límites de los valores de píxeles y las dimensiones de la imagen.
- **Método de Proceso:** Utiliza OpenCV para redimensionar efectivamente los frames del entorno. Este método estático toma un frame como entrada y realiza los siguientes pasos:
  1. Utiliza `cv2.resize` con interpolación bilineal para cambiar el tamaño del frame a 84×84.
  2. Reestructura el frame redimensionado a las dimensiones deseadas y asegura que el tipo de dato sea `np.float32`.
- **Redimensionamiento en la Observación:** El método `observation(obs)` invoca `ResizeFrame.process` para aplicar el proceso de redimensionamiento a cada observación recibida del entorno, asegurando que todas las observaciones pasen por este preprocesamiento antes de ser utilizadas por el algoritmo de aprendizaje.

```
1 class ResizeFrame(gym.ObservationWrapper):
2     def __init__(self,env=None):
3         super().__init__(env)
4         self._observation_space = gym.spaces.Box(low=0,high=255, shape=(84,84,1),
              dtype=np.uint8)
5 
6     @staticmethod
7     def resize_and_normalize(frame):
8         # Redimensionar la imagen con interpolación bilineal
```

```
9         resized_frame = cv2.resize(frame,(84,84),interpolation=cv2.INTER_LINEAR)
10        # Reorganizar la forma del array
11        reshaped_frame = resized_frame.reshape((84, 84, 1))
12         # Convertir el tipo de dato a float32 para normalización
13        normalized_frame = np.array(reshaped_frame).astype(np.float32)
14        return normalized_frame
15
16    def observation(self,obs):
17        return ResizeFrame.process(obs)
```

Importancia

Este preprocesamiento simplifica significativamente el entorno visual de Ms. Pac-Man, reduciendo la complejidad y el costo computacional asociados al manejo de imágenes de mayor resolución. Además, la estandarización del tamaño de las observaciones facilita el diseño de la arquitectura de la red neuronal, ya que se puede contar con una entrada de dimensiones fijas.

Este enfoque de redimensionamiento es una práctica estándar en el aprendizaje por refuerzo aplicado a problemas que involucran datos visuales, permitiendo un entrenamiento más rápido y eficiente del modelo.

4.3.4. FrameReshape

La clase `FrameReshape` es un *wrapper* de Gymnasium diseñado para adaptar las observaciones del entorno de manera que cumplan con los requisitos específicos de entrada de modelos de redes neuronales, especialmente las redes neuronales convolucionales (CNN). Este *wrapper* modifica la estructura de las observaciones, especialmente útil cuando se trabajan con datos visuales en tareas de aprendizaje profundo.

Constructor

El constructor de `FrameReshape` realiza las siguientes operaciones esenciales:

- Toma el entorno original `env` como argumento y lo inicializa.
- Ajusta el orden de las dimensiones de las observaciones originales. Las imágenes deben ser reformateadas para que el canal de color se posicione como la primera dimensión, en lugar de la última. Esto es crucial porque las redes convolucionales en PyTorch están diseñadas para trabajar con tensores donde la primera dimensión representa el canal de color.
- Calcula `new_shape` para reorganizar las dimensiones de las observaciones, moviendo el canal de color (que generalmente es la última dimensión en el formato de imagen estándar) al frente.
- Establece un nuevo espacio de observación que refleje este nuevo formato, con los valores de cada píxel entre 0.0 y 1.0.

```
class FrameReshape(gym.ObservationWrapper):
    def __init__(self,env):
        super().__init__(env)
        old_shape = self.observation_space.shape
        new_shape = (old_shape[-1], old_shape[0], old_shape[1])
        self.observation_space = gym.spaces.Box(
            low = 0.0,
            high=1.0,
            shape = new_shape,
            dtype=np.float32
        )
```

Función `observation`

La función `observation` es responsable de aplicar la transformación de las dimensiones cada vez que el entorno devuelve una observación:

- Utiliza `np.moveaxis` para reorganizar las dimensiones de la observación. Esta función mueve la última dimensión de la matriz al frente, transformando así el formato de la imagen para que se alinee con las expectativas de las capas convolucionales en PyTorch.
- Este reordenamiento es fundamental para asegurar que las operaciones de convolución aplicadas por las CNN (redes convolucionales) procesen los datos de la forma esperada, considerando el canal de color en su dimensión correcta.

```
    def observation(self,observation):
        return np.moveaxis(observation, 2, 0)
```

Este *wrapper* facilita la integración de entornos de Gymnasium con arquitecturas de aprendizaje profundo que requieren un formato específico de entrada, mejorando la compatibilidad y reduciendo la necesidad de preprocesamiento manual de datos antes del entrenamiento.

Impacto en el Proceso de Aprendizaje

- Esta reorganización permite una integración más directa y eficiente con las CNN de PyTorch, facilitando el aprendizaje automático a partir de observaciones visuales.
- Aplica una normalización de los frames del entorno que recibirá la red neuronal como entrada para mejorar el aprendizaje

4.3.5. FrameStack

La clase `FrameStack` es un *wrapper* de Gymnasium que está diseñado para apilar múltiples frames de observaciones, lo que permite al agente obtener una visión más comprensiva de la dinámica temporal del entorno. Este enfoque es particularmente valioso en entornos de aprendizaje por refuerzo donde las acciones y sus consecuencias no son instantáneas, sino que dependen de secuencias de eventos.

Implementación

`FrameStack` modifica el espacio de observación del entorno para incluir un número específico de frames anteriores en la observación actual, permitiendo así que el agente observe no solo el estado presente sino también estados anteriores, lo que es crucial para tareas que requieren comprensión del movimiento y cambios temporales.

Constructor

El constructor de `FrameStack` configura el nuevo espacio de observación para acumular varios frames, como se muestra en el siguiente código:

```
class FrameStack(gym.ObservationWrapper, gym.utils.RecordConstructorArgs):
    def __init__(self, env, num_stack):
        super().__init__(env)
        self.num_stack = num_stack
        self.frames = deque(maxlen=num_stack)
        # Ajustar el espacio de observación para acomodar el apilamiento de frames
        old_space = self.observation_space
        new_shape = (old_space.shape[0]*num_stack, old_space.shape[1], old_space.
            shape[2])
        self.observation_space = Box(
            low=old_space.low.repeat(num_stack, axis=0),
            high=old_space.high.repeat(num_stack, axis=0),
            shape=new_shape,
            dtype=old_space.dtype
        )
```

Funcionalidad

- **Espacio de Observación Modificado:** La modificación del espacio de observación asegura que cada frame dentro del stack sea tratado como una parte integral de la observación del entorno. Al extender el espacio de observación para contener múltiples frames consecutivos, el agente puede extraer patrones temporales cruciales que son necesarios para la toma de decisiones efectiva.
- **Gestión de Frames:** Utiliza una estructura de datos `deque`, que es ideal para este propósito debido a su eficiencia en operaciones de inserción y eliminación de elementos. Cada nuevo frame observado se añade al final del deque, mientras que el frame más antiguo se elimina automáticamente cuando el tamaño del deque excede el número `num_stack`.

Esta implementación no solo facilita una percepción enriquecida de la situación actual y pasada del entorno sino que también es fundamental para algoritmos que requieren un análisis detallado de la evolución del entorno a lo largo del tiempo, como es típico en muchas aplicaciones de control dinámico y juegos.

4.3.6. NormalizeFrame

La clase `NormalizeFrame` es un *wrapper* diseñado para normalizar las observaciones del entorno en Gymnasium, un paso de preprocesamiento crucial en el aprendizaje por refuerzo, especialmente en tareas que involucran datos visuales.

Propósito

Normalizar las observaciones tiene como objetivo ajustar los valores de los píxeles de las imágenes, que originalmente están en el rango de 0 a 255, a un rango de 0.0 a 1.0. Aplicar una normalización de los pixeles, asegura que todos los valores de entrada estén en una escala común, facilitando el proceso de aprendizaje del modelo al hacer que el entrenamiento sea menos sensible a la escala de características originales de la entrada.

Implementación

La implementación de `NormalizeFrame` es sencilla pero efectiva:

- Extiende `gym.ObservationWrapper`, permitiendo así modificar las observaciones que se pasan al agente.
- La función `observation` recibe una observación (un frame del entorno), la convierte a un `numpy array` de tipo `np.float32` y luego divide cada valor de píxel por 255. Este proceso convierte los valores de los píxeles a un rango entre 0.0 y 1.0.

```
1 class NormalizeFrame(gym.ObservationWrapper):
2     def observation(self,observation):
3         return np.array(observation).astype(np.float32) / 255.0
```

Impacto en el Aprendizaje

- **Mejora de la Convergencia:** Al escalar los valores de entrada a un rango más manejable, se facilita el ajuste de los pesos de la red neuronal durante el entrenamiento, lo que puede resultar en una convergencia más rápida y estable.
- **Uniformidad de los Datos:** La normalización asegura que todos los datos de entrada al modelo tengan una escala uniforme, lo cual es importante para evitar que ciertas características dominen el proceso de aprendizaje debido a su escala.

4.3.7. ReduceActionSpaceEnv

La clase `ReduceActionSpaceEnv` es un *wrapper* diseñado para simplificar el espacio de acciones de un entorno. Este *wrapper* es útil en situaciones donde el espacio de acciones original es demasiado grande o complejo, y se desea limitar las acciones disponibles para el agente a un conjunto más manejable. Esto puede facilitar el proceso de aprendizaje y mejorar la eficiencia del entrenamiento del agente.

Implementación

La implementación de `ReduceActionSpaceEnv` implica extender la clase `ActionWrapper` de Gymnasium y sobrescribir el método `action` para mapear las acciones del nuevo espacio reducido al espacio de acciones original del entorno.

Constructor El constructor realiza las siguientes operaciones esenciales:

- **Inicialización del entorno base:** Toma el entorno original `env` como argumento y llama al constructor de la clase base.
- **Redefinición del espacio de acciones:** Define un nuevo espacio de acciones discreto con solo 4 acciones posibles.
- **Mapeo de acciones:** Establece un diccionario de mapeo `action_map` que traduce cada acción del espacio reducido a una acción en el espacio original.

```
class ReduceActionSpaceEnv(gym.ActionWrapper):
    def __init__(self, env):
        super().__init__(env)
        # Define un nuevo espacio de acciones con 4 posibles acciones
        self.action_space = gym.spaces.Discrete(4)
        # Mapea las acciones del espacio reducido al espacio de acciones original
        self.action_map = {0: 1, 1: 4, 2: 2, 3: 3}
```

Método `action` El método `action` es responsable de mapear las acciones del espacio reducido al espacio de acciones original del entorno:

- **Mapeo de acciones:** Utiliza el diccionario `action_map` para traducir la acción del espacio reducido (`action`) a la acción correspondiente en el espacio original del entorno.

```
    def action(self, action):
        # Retorna la acción mapeada del espacio reducido al espacio original
        return self.action_map[action]
```

Propósito

Este *wrapper* simplifica la toma de decisiones del agente al reducir la cantidad de acciones posibles. Esto puede ser beneficioso en varios aspectos:

- **Facilita el aprendizaje:** Con un espacio de acciones más pequeño, el agente puede aprender más rápidamente y de manera más eficiente.
- **Reducción de la complejidad:** Disminuye la complejidad del problema, haciendo que sea más manejable tanto para el agente como para el algoritmo de aprendizaje.

4.4– Métodos Auxiliares

4.4.1. Transformación de Recompensas

La función `transform_reward` se utiliza para modificar las recompensas obtenidas del entorno con el fin de mejorar el proceso de aprendizaje. La adaptación de las recompensas es una práctica común en el aprendizaje por refuerzo para hacer que el entrenamiento sea más estable o para alinear mejor las recompensas con los objetivos a largo plazo del agente. En nuestro caso particular las recompesas que recibe el agente en el entorno Ms Pacman son las siguientes:

1. Comer una pastilla pequeña: +10 puntos.
2. Comer una pastilla grande (poder): +50 puntos.
3. Comer un fantasma azul:
   - Primer fantasma: +200 puntos.
   - Segundo fantasma (en la misma secuencia): +400 puntos.
   - Tercer fantasma (en la misma secuencia): +800 puntos.
   - Cuarto fantasma (en la misma secuencia): +1600 puntos.
4. Comer una fruta:
   - Cereza: +100 puntos.
   - Fresa: +300 puntos.
   - Naranja: +500 puntos.
   - Manzana: +700 puntos.
   - Melón: +1000 puntos.
   - Galaxian Boss: +2000 puntos.
   - Campana: +3000 puntos.
   - Llave: +5000 puntos.

Las recompensas de realizar algunas acciones difieren en una gran cantidad con la recopensa comer las pastillas pequeñas y de esta forma completar el nivel, si no aplicamos una normalización de las recompensas el agente podría desarrollar una política de acciones que se centrará únicamente y exclusivamente en por ejemplo comer fantasmas y dejará de lado los demás objetivos.

Descripción de la Función

La función `transform_reward` recibe dos argumentos: `reward`, que es la recompensa original obtenida del entorno, y `dead`, un booleano que indica si el agente ha `muerto` o terminado el episodio de manera no exitosa.

```
def transform_reward(reward,dead):
    if dead:
        reward = -log(20,1000)
    return log(reward, 1000) if reward > 0 else reward
```

Lógica de Transformación

1. Si el agente está muerto (`dead` es `True`), se ajusta la recompensa a un valor negativo predefinido, utilizando la función logarítmica para escalar este valor. Esto penaliza al agente de manera significativa por terminar el episodio de manera no exitosa.
2. Si el agente no está muerto y la recompensa original es mayor que cero, se aplica una transformación logarítmica a la recompensa para escalar los valores positivos, lo que puede ayudar a manejar recompensas de gran magnitud y asegurar una actualización de pesos más estable en el entrenamiento del modelo.

3. Si la recompensa es cero o negativa, se devuelve sin cambios. Esto mantiene las penalizaciones o la ausencia de recompensa intactas.

Impacto en el Aprendizaje

- **Penalización de Terminaciones Negativas:** Al ajustar negativamente las recompensas cuando el agente termina el episodio de manera no exitosa, se refuerza el aprendizaje hacia estrategias que evitan tales terminaciones.
- **Normalización de Recompensas Positivas:** La transformación logarítmica de recompensas positivas ayuda a normalizar la escala de las recompensas, facilitando el ajuste de los pesos del modelo y mejorando la estabilidad general del entrenamiento.

4.4.2. Skip Initial Frames

La función `skip_initial_frames` tiene como objetivo inicializar el entorno de Ms. Pac-Man de una manera estandarizada antes de comenzar cada episodio de entrenamiento o evaluación. Esta técnica ayuda a mitigar el impacto de las condiciones iniciales predeterminadas y introduce variabilidad en el inicio del entorno.

Descripción de la Función

La función ejecuta una acción nula (generalmente, la acción 0 que puede representar no moverse o un comando similar) un número fijo de veces (en este caso, 16 veces) al comienzo de cada episodio. Esto permite que el entorno avance varios frames desde su estado inicial antes de que el agente comience a tomar decisiones.

Implementación

```
1 def skip_initial_frames(env):
2     for i in range(16):
3         next_state,reward,done,truncated,info, = env.step(0)
```

Impacto en el Aprendizaje

- **Coste Computacional:** Reduce las predicciones del modelo cuando se reinicia el entorno, y este no permite al agente realizar ninguna acción.

CAPÍTULO 5

DQN

5.1– Introducción

La innovación de Deep Q-Networks (DQN) original constituye la base sobre la cual se construye Rainbow DQN, integrando avances significativos en el aprendizaje por refuerzo con las capacidades de generalización de las redes neuronales profundas. Esta sección explora con más detalle el algoritmo DQN, su funcionamiento y las innovaciones que introdujo en el campo del aprendizaje por refuerzo profundo [14].

5.2– DQN: Integración de Redes Neuronales

La técnica de Deep Q-Network (DQN) marcó un avance significativo en el campo del aprendizaje por refuerzo al introducir redes neuronales profundas para aproximar la función Q, la cual asigna pares de estado-acción a valores que estiman la calidad de tomar cierta acción en un estado dado. Antes de la aparición de DQN, los enfoques tradicionales de aprendizaje por refuerzo se apoyaban en tablas de valores Q para representar estos pares, lo que funcionaba bien en entornos con espacios de estado y acción discretos y de baja dimensión. Sin embargo, esta técnica no escalaba eficazmente para entornos de alta dimensionalidad, como aquellos donde el agente percibe su entorno a través de imágenes de píxeles, debido a la explosión combinatoria de posibles estados y acciones.

5.2.1. Red Neuronal como Aproximador de Función Q

DQN soluciona los desafíos de dimensionalidad utilizando una red neuronal profunda, denominada red Q, que puede procesar directamente entradas de alta dimensión y generar estimaciones de valores Q para todas las acciones posibles desde un estado dado. Esta red, por lo tanto, actúa como un aproximador de función valor, capturando la relación entre el estado del entorno y la acción óptima a tomar.

5.2.2. Adaptación de la Ecuación de Bellman en DQN

La Ecuación de Bellman es fundamental en el aprendizaje por refuerzo, ya que proporciona una forma recursiva de actualizar los valores Q basada en las recompensas obtenidas

y las estimaciones de los valores futuros. En DQN, esta ecuación se adapta para entrenar la red Q y se expresa de la siguiente manera:

$$Q(s,a) = r + \gamma \max_{a'} Q(s',a')$$

donde:

- $Q(s,a)$ representa el valor Q estimado por la red para el estado actual s y la acción a.
- r es la recompensa inmediata recibida después de tomar la acción a en el estado s.
- $\gamma \max_{a'} Q(s',a')$ es el valor máximo del valor Q para el próximo estado s', descontado por el factor de descuento γ. Este término representa el mejor valor Q proyectado para el siguiente estado, ponderado por la probabilidad de que ocurra ese estado.

5.2.3. Función de Pérdida en DQN

En el entrenamiento de la red Q dentro del marco de Deep Q-Network (DQN), se emplea una función de pérdida que evalúa la precisión de las aproximaciones de los valores Q realizadas por la red. Esta evaluación se basa en la Ecuación de Bellman, utilizando el error cuadrático medio (Mean Squared Error, MSE) para medir la discrepancia entre los valores Q predichos y los valores Q objetivos. La expresión matemática de la función de pérdida es:

$$L = \left[Q(s,a) - \left(r + \gamma \max_{a'} Q(s',a') \right) \right]^2$$

Donde:

- L es la pérdida o coste calculado, indicando cuánto se desvían las predicciones actuales de los valores teóricamente óptimos.
- $Q(s,a)$ es el valor Q predecido por la red para el estado actual s y la acción a.
- $r + \gamma \max_{a'} Q(s',a')$ representa el objetivo o target, que es la suma de la recompensa inmediata r y el valor Q máximo estimado para el próximo estado s', descontada por el factor de descuento γ.

El ajuste de los parámetros de la red Q, o sea, sus pesos, para minimizar la función de pérdida, es el núcleo del proceso de entrenamiento. Al reducir la función de pérdida, la red aprende gradualmente a predecir valores Q que concuerdan más estrechamente con las recompensas obtenidas y con las proyecciones de los valores futuros, aproximándose a la función Q óptima.

Este proceso de optimización se lleva a cabo comúnmente mediante algoritmos como el Descenso del Gradiente Estocástico (SGD) o Adam. Estos métodos ajustan los pesos de la red en dirección a una menor pérdida, facilitando un aprendizaje eficaz y robusto.

La capacidad de DQN para manejar entradas complejas y de alta dimensión, como imágenes, y su uso de una función de pérdida que incorpora la Ecuación de Bellman, hacen de este enfoque una herramienta extremadamente poderosa para una amplia gama de aplicaciones de aprendizaje por refuerzo, extendiéndose desde juegos hasta aplicaciones en robótica y más allá.

5.2.4. Técnicas para la Estabilización del Aprendizaje

Dentro de las técnicas implementadas por DQN para estabilizar y mejorar la eficiencia del aprendizaje, una de las más significativas es el Experience Replay. Aunque existen otras técnicas, como los Fixed Q-targets, nos centraremos en cómo el Experience Replay contribuye a un aprendizaje más robusto y estable.

Experience Replay La técnica de Experience Replay se basa en el almacenamiento y reutilización de experiencias pasadas del agente, cada una compuesta por tuplas (s, a, r, s'), donde s es el estado actual, a es la acción tomada, r es la recompensa recibida y s' es el estado siguiente. Estas experiencias se almacenan en un búfer de memoria que puede contener miles o incluso millones de estas tuplas.

Propósito y Beneficios

El propósito principal del Experience Replay es descorrelacionar las experiencias de entrenamiento del agente, que de otro modo serían secuenciales y por lo tanto altamente correlacionadas. Al muestrear aleatoriamente de este búfer para obtener los datos de entrenamiento, DQN reduce las correlaciones temporales entre muestras consecutivas. Esto tiene varios beneficios clave:

- **Mejora la Eficiencia del Aprendizaje:** Al reutilizar cada experiencia múltiples veces, el agente aprende más de cada secuencia de acción-estado, mejorando la eficiencia del aprendizaje sin la necesidad de obtener más datos a través de la interacción con el entorno.
- **Estabilización de la Convergencia:** La reducción de correlaciones entre las experiencias consecutivas ayuda a evitar ciclos de retroalimentación que pueden llevar a inestabilidades o divergencias en el aprendizaje, es decir, esta técnica disminuye el riesgo del que el agente siempre se quede atascado en el mismo punto, repitiendo una y otra vez la misma secuencias de acciones. Esto contribuye a una convergencia más suave y estable de la red neuronal.
- **Diversificación de Datos:** El muestreo aleatorio promueve una rica variedad de situaciones de entrenamiento, evitando el sobreajuste o overfitting a secuencias específicas y mejorando la capacidad del modelo para generalizar a partir de su experiencia.

Funcionamiento en DQN

En la práctica, el búfer de Experience Replay opera como un buffer de tamaño fijo: las nuevas experiencias se añaden continuamente mientras que las más antiguas se descartan una vez que se alcanza la capacidad máxima del buffer. Durante el proceso de entrenamiento, se extrae un mini-lote aleatorio de experiencias del buffer para calcular la pérdida y actualizar los pesos de la red.

5.3– Implemetación

Antes de empezar con la implementación en python, vamos a revisar el pseudocódigo de DQN con Experience Replay para tener una mayor entendimiento del algoritmo. El

pseudocódigo es el mostrado en la imagen siguiente y estos son los pasos que sigue:

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$
  end for
end for

Figura 5.1: Pseudocódigo de DQN [3]

1. Inicializar la memoria de repetición (replay memory) D con capacidad N:
   Se crea un buffer de memoria donde se almacenan las transiciones experimentadas por el agente. Esta memoria tiene un tamaño limitado N.

2. Inicializar la función de valor-acción Q con pesos aleatorios:
   Se configura la red neuronal que aproximará la función Q, inicializando sus pesos de forma aleatoria.

3. Para cada episodio de 1 a M:
   Se comienza un nuevo episodio hasta un máximo de M episodios de entrenamiento.

4. Inicializar la secuencia $s_1 = \{x_1\}$ y preprocesar la secuencia para obtener el estado $\phi_1 = \phi(s_1)$:
   Se toma el estado inicial del entorno, se preprocesa y se obtiene el estado inicial ϕ_1 que será utilizado por la red neuronal.

5. Para cada paso de tiempo t de 1 a T:
   Se ejecuta el algoritmo paso a paso para un número T de pasos de tiempo dentro del episodio.

6. Con probabilidad ϵ seleccionar una acción aleatoria a_t, de lo contrario seleccionar $a_t = \text{máx}_a\, Q^*(\phi(s_t), a; \theta)$:
   Se implementa una política ϵ-greedy para explorar el espacio de acción: con probabilidad ϵ se escoge una acción al azar para explorar el entorno, y con probabilidad $1 - \epsilon$, se elige la acción que maximiza la función Q para el estado actual $\phi(s_t)$.

7. Ejecutar la acción a_t y observar la recompensa r_t y la imagen x_{t+1}:
   Se lleva a cabo la acción seleccionada en el emulador (el entorno del problema), y se observa la recompensa y el nuevo estado como resultado de esa acción.

8. Preprocesar la siguiente secuencia $\phi_{t+1} = \phi(s_{t+1})$:
   El estado siguiente se preprocesa para ser compatible con las entradas de la red neuronal.

9. Almacenar la transición $(\phi_t, a_t, r_t, \phi_{t+1})$ en D:
   La experiencia obtenida (estado actual, acción tomada, recompensa recibida y siguiente estado) se guarda en la memoria de repetición.

10. Tomar una muestra aleatoria de mini-lotes de transiciones $(\phi_j, a_j, r_j, \phi_{j+1})$ de D:
    Se extrae un subconjunto aleatorio de experiencias previas almacenadas en la memoria para entrenar la red neuronal, lo que ayuda a romper las correlaciones en la secuencia de observaciones y estabilizar el aprendizaje.

11. Para cada muestra, establecer y_j:

    - Para estados no terminales, $y_j = r_j + \gamma \, \text{máx}_{a'} \, Q(\phi_{j+1}, a'; \theta')$.
    - Para estados terminales, $y_j = r_j$.

12. Aplicar el descenso del gradiente en $(y_j - Q(\phi_{j+1}, a'; \theta)^2)$.

5.3.1. Red Neuronal

La implementación de DQN al entorno de Atari `Ms Pacman`, requiere de la construcción de un red neuronal, para que como hemos visto anteriormente, devuelva las predicciones de los valores de Q, La red neuronal recibe una entrada de 4x84x84, que corresponde a los 4 fotogramas que el agente recibe como observación del entorno, que tienen un tamaño 84 pixeles de alto y de ancho, inicialmente el entorno que proporciona una observación de 210X160 pero se ha reducido el tamaño para reducir el costo computacional y se almacenan de 4 en 4, porque con un solo fotograma el agente no podría conocer la dirección del movimiento de los objetos, se podría decir que se le pasa como observación un video muy corto(se hablará más adelante más detalladamente sobre los cambios en el procesamiento del entorno). Como salida tiene 9 neuronas, una por cada acción posible que puede realizar el agente (Pacman) en el entorno, las cuales son:

1. **Ninguna acción:** Ms. Pac-Man se queda en su posición actual sin moverse.

2. **Mover hacia arriba:** Ms. Pac-Man se mueve hacia arriba en el laberinto.

3. **Mover hacia abajo:** Ms. Pac-Man se mueve hacia abajo en el laberinto.

4. **Mover hacia la izquierda:** Ms. Pac-Man se mueve hacia la izquierda en el laberinto.

5. **Mover hacia la derecha:** Ms. Pac-Man se mueve hacia la derecha en el laberinto.

6. **Mover hacia arriba a la izquierda:** Una combinación de mover hacia arriba y hacia la izquierda simultáneamente.

7. **Mover hacia arriba a la derecha:** Una combinación de mover hacia arriba y hacia la derecha simultáneamente.

8. **Mover hacia abajo a la izquierda:** Una combinación de mover hacia abajo y hacia la izquierda simultáneamente.

9. **Mover hacia abajo a la derecha:** Una combinación de mover hacia abajo y hacia la derecha simultáneamente.

La estructura red neuronal deberá contar con 3 fases diferenciables, un conjunto de capas de redes convolucionales para procesar el frame pasado como observación al agente, un aplanamiento de la salida devuelta por las redes convolucionales y un capas lineales para hacer la predicción final de la acción a elegir por el agente.

1ª parte: capas convolucionales

En nuestro caso particular se ha seleccionado un conjuto de 3 capas convolucionales (en Pytorch definimos capas convulucionales con nn.Conv2d) para extraer los detalles de la observaciones que proporciona el entorno de `Ms Pacman` de OpenAI. Se hace una disminución progresiva de los filtros, desde un tamaño de 8x8 pixeles a 3x3 (kernel-size es lo que indica este valor), permitiendo de esta forma una extracción de las características más refinada a medida que los datos de entrada son procesados por las capas convolucionales.

```
self.convolutional_layers = nn.Sequential(
          nn.Conv2d(input_size[0], 32, kernel_size=8, stride=4),
          nn.ReLU(),
          nn.Conv2d(32, 64, kernel_size=4, stride=2),
          nn.Conv2d(64, 64, kernel_size=3, stride=2),
          nn.ReLU(),
          )
```

2ª Parte: Aplanamiento de las capas convolucionales

Después de procesar la entrada a través de las capas convolucionales, el siguiente paso es preparar esta salida para las capas lineales. Esto involucra dos procesos clave: el aplanamiento de la salida convolucional y el cálculo del tamaño de esta salida para adecuar las dimensiones de las capas lineales.

El aplanamiento transforma los arrays de características bidimensionales resultantes de la última capa convolucional en un vector unidimensional. Este proceso es crucial porque las capas lineales requieren entradas en formato de vector para su procesamiento. En PyTorch, esto se realiza utilizando nn.Flatten(), que prepara los datos para la transición de la representación espacial a una estructura lineal adecuada para la decisión de acciones.

El método `get_convolutional_output` juega un papel fundamental en la adaptación de la arquitectura de la red a cualquier tamaño de entrada. Este método calcula dinámicamente el tamaño de la salida de las capas convolucionales, lo que es esencial para configurar correctamente la primera capa lineal que sigue al aplanamiento. La razón detrás de este cálculo es que el tamaño de salida de las capas convolucionales depende de varias variables, como el tamaño de entrada, el tamaño del kernel, el stride y el padding.

El proceso implica pasar un tensor de ceros con las mismas dimensiones que la entrada de la red a través de las capas convolucionales sin aplicar ninguna operación de aplanamiento o lineal posteriormente. Esto resulta en un tensor de salida cuyo tamaño total es representativo del número de características que serán alimentadas a la primera capa lineal.

```
def _get_convolutional_output(self, input_size):
      # Función para calcular la salida de las capas convolucionales
```

```
        with torch.no_grad():
            sample = torch.zeros(1, *input_size)
            sample = self.convolutional_layers(sample)
        return int(np.prod(sample.size()))
```

Este valor calculado se utiliza entonces para definir el tamaño de entrada de la primera capa lineal (nn.Linear), asegurando que la red pueda adaptarse a diferentes tamaños de entrada sin requerir ajustes manuales en la arquitectura de las capas lineales.

3ª Parte: Capas lineales

La última fase de nuestra red involucra la capa de salida, que está directamente vinculada con el número de acciones posibles que el agente puede elegir en el entorno de `Ms Pacman`. La función de esta capa es mapear la representación densa obtenida de las capas lineales anteriores a un vector de salida cuya dimensión corresponde al número de acciones disponibles.

En este paso, `output_size` representa el número total de acciones posibles. La capa de salida no aplica una función de activación como softmax o sigmoide directamente, ya que en el contexto de DQN, el agente a la hora de seleccionar la acción óptima, eligirá la de mayor valor de Q, como ocurre en la selección de acciones basada en ε-greedy.

Esta estructura permite que la red neuronal, a partir de una imagen de entrada que representa el estado actual del juego, prediga los valores Q para cada posible acción. Estos valores Q estiman la utilidad de tomar una determinada acción dada la situación actual, facilitando así la toma de decisiones del agente de manera informada y basada en el aprendizaje adquirido durante el entrenamiento.

```
self.linear_layers = nn.Sequential(
            nn.Flatten(),
            nn.Linear(convolutional_output, 512),
            )
self.output_layer == nn.Linear(512,output_size)
```

Funciones auxiliares

Además de la arquitectura principal de la red, es crucial contar con funciones auxiliares que permitan manipular y utilizar la red de manera efectiva en diferentes contextos, como durante el entrenamiento y la evaluación del agente. A continuación, detallamos las funciones auxiliares clave de nuestra red neuronal diseñada para el entorno de `Ms Pacman`.

Función forward La función forward define cómo se procesa la entrada a través de la red para generar una salida. En el contexto de nuestra red neuronal para DQN, esta función toma un tensor de entrada x, que representa el estado actual del entorno (por ejemplo, un frame de `Ms Pacman`), y lo procesa para producir los valores Q asociados a cada acción posible desde ese estado. Esta función es fundamental para el proceso de decisión del agente, ya que los valores Q calculados se utilizan para seleccionar la acción a realizar según la política de elección (por ejemplo, ε-greedy).

```
def forward(self, x):
        x = self.convolutional_layers(x)
```

```
        x = x.view(x.size(0), -1)
        #Procesamiento a través de las capas lineales
        x = self.linear_layers(x)
        #Obtener los Q-values como salida
        q_values = self.output_layer(x)
        return q_values
```

Función save/load model La capacidad de guardar el modelo es esencial para poder reanudar el entrenamiento en otro momento o utilizar el modelo entrenado sin necesidad de volver a entrenarlo desde cero. La función `save_model` facilita esta tarea guardando los parámetros de la red en un archivo. Al proporcionar un nombre de archivo, esta función guarda el estado actual de la red, incluyendo todos los pesos y sesgos de las capas, lo que permite una fácil recuperación y reutilización del modelo. Complementaria a `save_model`, la función `load_model` permite cargar un modelo previamente guardado. Esto es especialmente útil para la evaluación de modelos o la continuación del entrenamiento. Al cargar el estado de la red desde un archivo, esta función reinstaura los pesos y sesgos del modelo, permitiendo que el agente retome su comportamiento aprendido sin necesidad de un nuevo entrenamiento desde cero.

Estas funciones auxiliares añaden flexibilidad y eficiencia al proceso de entrenamiento y evaluación de la red, permitiendo un manejo más sencillo del modelo a lo largo de diferentes fases del proyecto de aprendizaje por refuerzo.

```
def save_model(self, filename):
        torch.save(self.state_dict(), filename)

def load_model(self, filename):
    self.load_state_dict(torch.load(filename))
```

Implementación Q-learning

Una vez que ya tenemos montada la estructura de la red neuronal para predecir los valores de Q, vamos a ver cómo implementaremos la función de entrenamiento para que dicha red sea capaz de mejorar su política con cada iteración. Nuestro agente contará con una función train, esta función se ocupará de aplicar el algoritmo de Q-learning y de calcular los errores para entrenar a nuestro modelo. Vamos a ver paso por paso como funciona esta función.

```
 def train(self,step_counter):
        if self.memory.size() < self.train_start:
            return
        #Get samples
        idxs,states_tensor,actions_tensor,rewards_tensor,next_states_tensor,
            dones_tensor = self.memory.sample(self.batch_size,self.device)
        current_q_values = self.model(states_tensor).gather(1,actions_tensor.
            unsqueeze(-1)).squeeze(-1)
        with torch.no_grad():
            next_q_values = self.model(next_states_tensor).max(1)[0]
            next_q_values[dones_tensor] = 0.0
            expected_q_values = rewards_tensor + self.discount_factor *
                next_q_values
```

```
        loss = nn.MSELoss()(current_q_values, expected_q_values.detach())
        self.optimizer.zero_grad()
        loss.backward()
        self.optimizer.step()
```

1. **Recopilación de la experiencia**: La función espera a que se llene la memoria del buffer del experience replay, una vez que esta llena la memoria, recoge 128 transiciones (el tamaño de batch-size), estas transiciones van a ser utilizadas para calcular el error y entrenar al modelo. El experience replay se verá detalladamente más adelante.

```
if self.memory.size() < self.train_start:
      return
   states_tensor, actions_tensor, rewards_tensor, next_states_tensor,
      dones_tensor = self.memory.sample(self.batch_size, self.device)
```

2. **Cálculo de los valores Q objetivo**: Para cada muestra en el lote, calcula los valores Q objetivo para las acciones tomadas. Estos se estiman como la recompensa observada por tomar la acción más el valor Q del mejor acción posible en el siguiente estado, descontado por un factor gamma (γ = discount-factor), el valor de q se pone a 0 para no actualizar en estados terminales.

```
current_q_values = self.model(states_tensor).gather(1, actions_tensor.
    unsqueeze(-1)).squeeze(-1)
    with torch.no_grad():
        next_q_values = self.model(next_states_tensor).max(1)[0]
        next_q_values[dones_tensor] = 0.0
        expected_q_values = rewards_tensor + self.discount_factor *
            next_q_values
```

3. **Función de pérdida y retropropagación**: Calcula la pérdida entre los valores de Q actuales y los esperados usando el error cuadrático medio (MSE). Luego realiza el proceso de retropropagación del error para actualizar los pesos del modelo: primero, reinicia los gradientes del optimizador; después, ejecuta la retropropagación; finalmente, realiza un paso de optimización para ajustar los pesos.

```
    loss = nn.MSELoss()(current_q_values, expected_q_values.detach())
    self.optimizer.zero_grad()
    loss.backward()
    self.optimizer.step()
```

Exploración vs Explotación: ϵ-greedy

El método ϵ-greedy es una estrategia fundamental para equilibrar la exploración y la explotación en el aprendizaje por refuerzo. La idea detrás de esta estrategia es simple pero efectiva: con una probabilidad ϵ, el agente toma una acción aleatoria (exploración), y con una probabilidad 1-ϵ, el agente elige la acción que maximiza los Q-values según su conocimiento actual (explotación). Este enfoque permite al agente descubrir nuevas estrategias explorando el espacio de acción mientras aprovecha el conocimiento adquirido para mejorar su desempeño. La elección de ϵ es crucial; un valor muy alto favorece la

exploración pero puede impedir que el agente aprenda estrategias efectivas, mientras que un valor muy bajo puede llevar a una explotación prematura y posiblemente a converger a políticas subóptimas.

```
def get_action(self, state):
       if np.random.rand() <= self.epsilon:
           return random.randrange(self.action_size)
       else:

           state_tensor = torch.tensor(state).unsqueeze(0).to(self.device)
           q_value = self.model(state_tensor)
       return np.argmax(q_value[0].cpu().detach().numpy())
```

Como podemos observar en la imagen, el agente del entorno elige entre hacer una acción aleatoria y la acción predecida por el modelo que es más probable que devuelva una recompensa mayor, dicha elección tiene una probabilidad de 1-ϵ, el factor ϵ se va reduciendo a medida que el agente va iterando, gracias a lo que se conoce como un "ϵ-decay", que marca el ritmo con el que el factor ϵ se reduce, este paso se repite cada vez que guarda una experiencia nuevo, lo veremos en el capitulo de `Prioritized Experience Replay`. Al principio la mayoría de las acciones tomadas por el agente serán aleatorias y progresivamente irá tomando las mejores acciones. La técnica ϵ-greedy para controlar la exploración y la explotación es facil de implementar permitiendo que el agente tenga más variedad de observaciones, ya que sin esta exploración inicial, el agente siempre tomaría el mismo camino, (en nuesto caso de `Ms Pacman`) y este camino sería subóptimo.

CAPÍTULO 6

Double DQN

6.1– Introducción

Double Deep Q-Networks (Double DQN) se posiciona como una mejora crucial sobre el algoritmo DQN tradicional. Al abordar el problema de la sobreestimación de los valores Q, Double DQN refina la precisión de las estimaciones de valor y, por ende, la calidad de la política de aprendizaje. Esta sección profundiza en el desarrollo, mecanismos y el impacto de Double DQN dentro del ecosistema de Rainbow DQN [2].

6.1.1. Sobreestimación de Valores Q en DQN

La implementación estándar de DQN tiende a sobreestimar los valores Q debido a su mecanismo de selección y evaluación de la mejor acción, basado únicamente en la maximización del valor Q estimado por una única red. Este sesgo de sobreestimación puede desviar al agente hacia políticas subóptimas, afectando negativamente su capacidad de aprender de manera eficaz, para ello, lo que propone Double DQN es una segunda red neuronal para contrastar las estimaciones de los valores de Q.

6.2– Fundamentos de Double Deep Q-Network (Double DQN)

Double DQN es una variante del algoritmo Deep Q-Network (DQN) diseñada para mitigar el problema de sobreestimación que surge al estimar valores Q. Esto se logra mediante la utilización de dos conjuntos de parámetros separados: θ para la red que selecciona la acción y θ' para la red que evalúa el valor de la acción seleccionada.

6.2.1. Mecanismo de Actualización en Double DQN

El proceso de actualización en Double DQN se define matemáticamente como:

$$Q(s,a;\theta) \leftarrow Q(s,a;\theta) + \alpha \left(r + \gamma Q \left(s', \underset{a'}{\operatorname{argmax}}\, Q(s',a';\theta);\theta' \right) - Q(s,a;\theta) \right),$$

donde:

- s y s' representan el estado actual y el siguiente estado, respectivamente.

- a y a' representan la acción actual y la acción que maximiza la función de valor Q en el estado s', respectivamente.
- α es la tasa de aprendizaje, que controla cuánto se actualiza la función de valor Q en cada paso.
- r es la recompensa recibida después de tomar la acción a en el estado s.
- γ es el factor de descuento, que modula la importancia de las recompensas futuras.
- θ y θ' son los parámetros de las redes neuronal, que seleccionan acciones y evalúan valores respectivamente.

6.2.2. Función de Pérdida

La función de pérdida en Double DQN se expresa como el error cuadrático medio (MSE) entre el valor Q actualizado y el valor Q objetivo:

$$L(\theta) = \mathbb{E}\left[\left(Q(s,a;\theta) - \left(r + \gamma Q\left(s', \underset{a'}{\operatorname{argmax}}\, Q(s',a';\theta);\theta'\right)\right)\right)^2\right].$$

Esta función de pérdida ayuda a ajustar los parámetros θ de la red de comportamiento para alinear mejor las estimaciones de valor Q con los retornos reales obtenidos, mejorando así la calidad del aprendizaje y la estabilidad del entrenamiento.

6.2.3. Desafíos y Soluciones

Aunque Double DQN ofrece una solución significativa al problema de la sobreestimación de los valores Q, su implementación no está exenta de desafíos. Uno de los principales retos es el mantenimiento de dos conjuntos de parámetros y el ajuste de la frecuencia con la que se actualiza la red de evaluación (θ'). La práctica común es actualizar los parámetros de la red de evaluación (θ') con los parámetros de la red de selección (θ) cada cierto número de pasos. Este enfoque equilibra entre la estabilidad y la actualidad de la evaluación, crucial para el aprendizaje efectivo.

6.2.4. Conclusión

Double DQN representa un avance significativo en la búsqueda de algoritmos de aprendizaje por refuerzo más estables y eficaces. Al abordar el problema de la sobreestimación de los valores Q, esta técnica mejora la calidad del aprendizaje y la política de acción del agente. Su integración en Rainbow DQN ejemplifica como la combinación de múltiples mejoras puede conducir a avances significativos en la capacidad de los agentes para aprender de manera eficiente y efectiva en entornos complejos. La continua exploración y mejora de estas técnicas subraya el potencial creciente del aprendizaje por refuerzo profundo para aplicaciones prácticas y desafíos computacionales.

6.3– Implementación

En esta sección vamos a explicar la implementación del algoritmo de Double-DQN, para añadir una nueva mejora para cada vez acercarnos más al algoritmo de Rainbow-DQN, primero vamos a ver el pseudocógo antes de pasar a la implementación real.

Algorithm 1 : Double Q-learning (Hasselt et al., 2015)

Initialize primary network Q_θ, target network $Q_{\theta'}$, replay buffer $\mathcal{D}$, $\tau << 1$
for each iteration **do**
  for each environment step **do**
    Observe state s_t and select $a_t \sim \pi(a_t, s_t)$
    Execute a_t and observe next state s_{t+1} and reward $r_t = R(s_t, a_t)$
    Store (s_t, a_t, r_t, s_{t+1}) in replay buffer $\mathcal{D}$
  for each update step **do**
    sample $e_t = (s_t, a_t, r_t, s_{t+1}) \sim \mathcal{D}$
    Compute target Q value:
      $Q^*(s_t, a_t) \approx r_t + \gamma\ Q_\theta(s_{t+1}, argmax_{a'} Q_{\theta'}(s_{t+1}, a'))$
    Perform gradient descent step on $(Q^*(s_t, a_t) - Q_\theta(s_t, a_t))^2$
    Update target network parameters:
      $\theta' \leftarrow \tau * \theta + (1 - \tau) * \theta'$

Figura 6.1: Pseudocódigo Double DQN[2]

Inicialización:

- Q_θ: Inicializa la red neuronal principal que se usa para seleccionar acciones.
- $Q_{\theta'}$: Inicializa la red neuronal objetivo que se usa para evaluar acciones.
- D: Inicializa el búfer de experiencia donde se almacenarán las transiciones (estado, acción, recompensa, estado siguiente).
- τ: Establece un factor de actualización (τ) para la red objetivo, con un valor muy cercano a 0 pero menor que 1.

Para cada iteración:

1. **Para cada paso del entorno:**
   - Observa el estado actual s_t y selecciona una acción a_t basada en la política π. La política π suele ser una estrategia ϵ-greedy, donde con probabilidad ϵ se selecciona una acción aleatoria para fomentar la exploración, y con probabilidad $1 - \epsilon$ se selecciona la acción que maximiza el valor Q en el estado actual s_t, es decir, $a_t = \arg\max_a Q_\theta(s_t, a)$.
   - Ejecuta la acción a_t y observa el nuevo estado s_{t+1} y la recompensa r_t.
   - Almacena la transición en el búfer de experiencia D.

2. **Para cada paso de actualización:**

   - Muestrea una mini-batch e_t de transiciones del búfer de experiencia D.
   - Calcula el valor Q objetivo Q^* para cada muestra en la mini-batch, usando la recompensa observada r_t y el valor Q de la mejor acción en el siguiente estado s_{t+1}, obtenido de la red principal Q_θ, pero evaluado usando los pesos de la red objetivo $Q_{\theta'}$. Esta es la parte crucial que separa la selección de la acción de su evaluación para evitar la sobreestimación.
   - Realiza un paso de descenso de gradiente para minimizar la diferencia al cuadrado entre el Q objetivo Q^* y el valor Q predicho por la red principal Q_θ, ajustando así los parámetros de la red principal.
   - Actualiza los parámetros de la red objetivo θ' mezclando gradualmente los parámetros de la red principal θ con los de la red objetivo, utilizando el factor τ.

6.3.1. Implementación Práctica

El primer paso a seguir para realizar la implementación es inicializar dos redes neuronales, para ello, utilizaremos la definida en el capitulo anterior de DQN, necesitamos dos modelos (uno para seleccionar las acciones y otro para evaluar las acciones evitando el problema de sobreestimación) con los mismos hiperparámetros y estructura, también necesitaremos un parámetro que indique cada cuanto actualizaremos la red que evalua las acciones `model_target` (la clase `NeuralNetwork` es la definida en el capitulo de DQN):

```
1 self.update_rate = 1000
2           self.model = Neural_Network(self.state_size,self.action_size,self.
              learning_rate).to(device)
3           self.model_target = Neural_Network(self.state_size,self.action_size,
              self.learning_rate).to(device)
```

Entrenamiento de las redes

El código para la implementación no difiere mucho del código para DQN. Los principales cambios son:

Actualización con retraso del modelo target. La red que evalúa las acciones del modelo principal, se actualiza con los pesos de la red neuronal principal con un retraso o índice de actualización para que el aprendizaje sea más estable y efectivo. En nuestro caso particular, hemos tenido que añadir a la función de entrenamiento un valor de entrada que indica el número de la iteración (step-counter) y junto con el hiperparámetro de la imagen anterior `self.update_rate`, hacemos que se actualice cada 1000 iteraciones.

```
1 if step_counter % self.update_rate==0:
2    self.model_target.load_state_dict(self.model.state_dict())
```

Cálculo de los valores Q en Double DQN. A diferencia de DQN, donde los valores Q para las acciones futuras se estiman con la misma red que se usa para seleccionar la acción, Double DQN introduce una red objetivo separada para evaluar estas acciones futuras. Durante el proceso de entrenamiento, la acción futura se selecciona utilizando la red principal pero se evalúa con la red objetivo, cuyos parámetros están desfasados respecto a la red principal y solo se actualizan periódicamente, como hemos visto anteriormente.

```
    q_state_values = self.model(states_tensor).gather(1, actions_tensor.unsqueeze
        (-1)).squeeze(-1)

    with torch.no_grad():
        q_next_states_values = self.model_target(next_states_tensor).max(1)[0]
        q_next_states_values[dones_tensor] = 0.0
        q_next_states_values = q_next_states_values.detach()
    target_q_val = q_next_states_values * self.discount_factor + rewards_tensor
```

Finalmente se calcula el error cuadratico medio entre los valores de q actuales y los futuros y se hace el descenso del gradiente en el modelos principal como en DQN, en el target no hace falta, ya que se actualizará cuando sea necesario con los pesos de la red principal. El código de la función de entrenamiento de los modelos finalmente queda tal que asi:

```
def train(self,step_counter):
      if self.memory.size() < self.train_start:
          return
      self.optimizer.zero_grad()

      if step_counter % self.update_rate == 0:
          self.model_target.load_state_dict(self.model.state_dict())

      states_tensor,actions_tensor,rewards_tensor,next_states_tensor,
          dones_tensor = self.memory.sample(self.batch_size,self.device)

      q_state_values = self.model(states_tensor).gather(1, actions_tensor.
          unsqueeze(-1)).squeeze(-1)
      with torch.no_grad():
          q_next_states_values = self.model_target(next_states_tensor).max(1)[0]
          q_next_states_values[dones_tensor] = 0.0
          q_next_states_values = q_next_states_values.detach()

      target_q_val = q_next_states_values * self.discount_factor +
          rewards_tensor
      loss = nn.MSELoss()(q_state_values,target_q_val)
      loss.backward()
      self.optimizer.step()
```

CAPÍTULO 7

Distributional RL

7.1– Introducción

El aprendizaje por refuerzo distribucional (Distributional Reinforcement Learning, DRL) es un enfoque que se centra en la distribución completa de las recompensas futuras, en lugar de solo esperar un valor medio. Esta técnica proporciona una comprensión más rica y completa del entorno, ya que captura la naturaleza estocástica del retorno de las recompensas [7].

7.2– Fundamentos DRL

En los métodos de aprendizaje por refuerzo clásicos, la función de valor se define como la expectativa de la suma acumulada de recompensas descontadas. En contraste, DRL considera la distribución de la suma acumulada de recompensas. Esta sección explorará los fundamentos teóricos detrás de este cambio de paradigma y cómo puede llevar a una mejor representación de la incertidumbre.

7.2.1. El Problema con la Expectativa

Comenzamos hablando de cómo el enfoque en la expectativa puede ser limitante en entornos con alta varianza en las recompensas. Tradicionalmente, los métodos de aprendizaje por refuerzo buscan maximizar la expectativa de la suma de recompensas futuras. Aunque este enfoque ha demostrado ser efectivo en muchos escenarios, tiene limitaciones significativas, especialmente en entornos donde la varianza de las recompensas es alta.

Limitaciones de la Esperanza

El valor máximo de una distribución, si bien proporciona una medida central, no captura la dispersión o la forma de la distribución de las recompensas. Esto significa que dos distribuciones muy diferentes pueden tener la misma esperanza, pero con implicaciones muy distintas para la toma de decisiones.

Impacto en la Toma de Decisiones

En situaciones donde las recompensas tienen alta varianza, un enfoque basado únicamente en la esperanza puede llevar al agente a tomar decisiones subóptimas que no reflejan adecuadamente la naturaleza de los riesgos involucrados. Esto se debe a que maximizar la esperanza no necesariamente maximiza la utilidad del agente.

7.2.2. Distribuciones de Valor en DRL

DRL plantea una pregunta fundamental: ¿Qué pasaría si, en lugar de estimar simplemente el valor esperado de la suma de recompensas futuras, tratáramos de predecir la distribución completa de dicha suma? Esto se logra mediante la modelización de la función de valor como una distribución de probabilidad sobre los retornos posibles.

Para formalizar este enfoque, se introduce el concepto de la función de distribución de valor, $Z(s, a)$, que para un estado s y una acción a, describe la distribución de la suma descontada de recompensas futuras. En lugar de actualizar un solo número que representa el valor esperado, se actualizan parámetros de una distribución de probabilidad.

El Operador de Bellman Distributivo

El operador de Bellman distributivo actualiza la distribución completa de los retornos esperados y no solo su media. Matemáticamente, si Z es la función de distribución del valor, el operador de Bellman distribucional aplicado a Z para una política π se define como [7]:

$$Z(s, a) = R(s, a) + \gamma Z(S', A'),$$

donde los términos son definidos de la siguiente manera:

- $Z(s, a)$: Representa la función de distribución de valor para el estado s y la acción a. Esta función de distribución encapsula todos los retornos esperados cuando se está en el estado s y se toma la acción a.
- $R(s, a)$: Denota la recompensa recibida por tomar la acción a en el estado s. Es un componente esencial que refleja el beneficio inmediato de una acción en un estado dado.
- γ: Es el factor de descuento, utilizado para moderar la importancia de las recompensas futuras en comparación con las inmediatas. Un valor más bajo de γ da más peso a las recompensas recibidas en el corto plazo.
- $Z(S', A')$: Es la función de distribución de valor para el estado y acción subsiguientes, S' y A', donde S' y A' son variables aleatorias que representan la transición estocástica a un nuevo estado y la elección de una nueva acción según la política π.

7.2.3. Beneficios del Aprendizaje por Refuerzo Distribucional

La adopción de un enfoque distribucional en el aprendizaje por refuerzo ofrece múltiples ventajas sobre los métodos tradicionales. Estos beneficios incluyen:

Captura de la Incertidumbre DRL no solo predice el valor esperado, sino que también captura la incertidumbre inherente a los retornos de las recompensas. Esta característica proporciona una visión más completa de los posibles resultados y sus probabilidades, permitiendo que el agente tome decisiones más informadas en entornos estocásticos.

Estabilidad en la Aprendizaje La modelización de la distribución completa de los retornos puede conducir a una mayor estabilidad durante el entrenamiento. Al considerar la varianza y la forma completa de la distribución, los algoritmos DRL pueden ser más resistentes a las variaciones extremas de las recompensas, lo que resulta en una convergencia más suave.

Mejora de la Exploración DRL puede mejorar la estrategia de exploración del agente. Al comprender la distribución de los retornos, el agente puede explorar de manera más efectiva, buscando acciones que puedan no tener el valor esperado más alto, pero que ofrezcan oportunidades de recompensas altas con cierta probabilidad.

Políticas basadas en Riesgo El enfoque distribucional permite al agente formular políticas que manejan el riesgo de manera explícita. Por ejemplo, un agente puede optar por evitar acciones con alta varianza en las recompensas, incluso si ofrecen un valor esperado atractivo, lo cual es crítico en aplicaciones donde la minimización de riesgos es fundamental.

7.2.4. Distributional DQN

Dentro del marco de Distributional Reinforcement Learning (DRL), Distributional DQN se posiciona como una adaptación del algoritmo Deep Q-Network (DQN) que incorpora los principios de DRL. Mientras que DQN tradicional estima el valor esperado de la acción-valor $Q(s, a)$ para un estado s y una acción a, Distributional DQN busca modelar la distribución completa de los posibles retornos asociados a cada pareja estado-acción. Estas son las características clave de Distributional DQN:

- **Modelado de la Distribución de Retornos:** A diferencia de DQN, que aprende una estimación del valor esperado de $Q(s, a)$, Distributional DQN aprende una distribución de probabilidad sobre todos los posibles valores de retorno de $Q(s, a)$. Esto se logra generalmente utilizando una red neuronal para parametrizar la distribución de retorno.
- **Actualización del Operador de Bellman:** En Distributional DQN, el operador de Bellman se adapta para trabajar con distribuciones. La actualización de la distribución objetivo se realiza de manera que tenga en cuenta la naturaleza probabilística de los retornos futuros, lo que requiere métodos especializados para comparar y actualizar distribuciones.

7.2.5. Conclusión

El Aprendizaje por Refuerzo Distribucional (DRL) representa un avance significativo en el campo del aprendizaje por refuerzo, al abordar de manera más completa y matizada la incertidumbre y la variabilidad inherentes a muchos entornos de decisión. Al alejarse

de la simple maximización de una expectativa de recompensas futuras para abrazar la modelización de toda la distribución de estas recompensas, DRL ofrece un enfoque más rico y potencialmente más robusto para el aprendizaje de políticas óptimas.

Sin embargo, es importante reconocer que, si bien DRL ofrece muchas ventajas, también introduce desafíos adicionales, como la necesidad de estimar y actualizar distribuciones completas de retorno, lo que puede aumentar la complejidad computacional y algorítmica del proceso de aprendizaje. A pesar de estos retos, los beneficios que ofrece DRL, especialmente en términos de una representación más rica del entorno y una mejor gestión del riesgo, lo posicionan como un enfoque prometedor para el avance del aprendizaje por refuerzo.

En conclusión, el Aprendizaje por Refuerzo Distribucional se destaca como un paso evolutivo crucial en la búsqueda de sistemas de IA que puedan navegar y tomar decisiones en entornos que cuentan con una gran incentidumbre y mucha dimensinalidad.

7.3– Implementación

Como hemos visto anteriormente, Distributional DQN cambia la forma en que se modelan las recompensas. En lugar de estimar el valor esperado de la recompensa (como en DQN estándar), se modela la distribución completa de los posibles valores de retorno. Esto permite que el agente aprenda no solo el promedio de los posibles retornos sino también cómo se distribuyen, lo que puede ayudar a capturar la incertidumbre y variabilidad en los retornos. Es importante recalcar que se va a implementar Distributional DQN junto con Advantage Actor Critic (A2C) que será explicado en el próximo capítulo.

```
    actor = self.actor_layer(x)
        value = self.critic_layer(x)
        distribution = Categorical(F.softmax(actor, dim=-1))
        return distribution, value
```

Se crea una distribución categórica basada en las salidas de la capa de actor. Aquí, la función softmax se utiliza para obtener una distribución de probabilidad sobre las acciones. La distribución categórica, (se utiliza una distribución categorica, ya que la salidas de nuestro modelo son discretas) se utiliza para muestrear acciones durante la fase de entrenamiento, y para seleccionar la acción más probable durante la fase de inferencia. La función forward al completo queda tal que así:

```
def forward(self, x):
        x = self.convolutional_layers(x).view(x.size()[0],-1)
        x = self.linear_layers(x)

        actor = self.actor_layer(x)
        value = self.critic_layer(x)
        distribution = Categorical(F.softmax(actor,dim=-1))
        return distribution,value
```

7.3.1. Elección de acción

El agente de nuestro entorno seguirá utilizando una política de exploración vs explotación ϵ-greedy, como se explicó en el capítulo de DQN, pero esta vez no debemos de

escoger la acción de entre las 9 posibles que tenga una mayor estimación de Q, eligiremos una aletoriamente teniendo en cuenta la probabilidad muestrada por nuestro modelo, es decir, dichas acciones que cuenten con un porcentaje mayor de la distribución serán más propensas a ser elegidas por el agente de nuestro entorno. La variable `prob` contiene la distribución categórica que devuelve el modelo y con la función `sample`, escogemos una acción al azar teniendo en cuenta dicha distribución.

```
def get_action(self, state):
      if np.random.rand() <= self.epsilon:
         return random.randrange(self.action_size)
      else:
         state_tensor = torch.tensor(state).unsqueeze(0).to(self.device)
         prob,_ = self.model(state_tensor)
      action = prob.sample().item()
      return action
```

7.3.2. Entrenamiento

El único cambio que hay que realizar en la función de entrenamiento de nuesto modelo, es tener en cuentra que ahora trabajamos con probabilidades estocásticas y para ello hacemos uso de los logaritmos para representar dichas probabilidades para evitar problemas numéricos con las acciones con probabilidades muy pequeñas.

```
      dist,values = self.model(states_tensor)
      values = values.squeeze()
      log_probs = dist.log_prob(actions_tensor)
```

Finalmente la función de entrenamiento con la nueva mejora de Distributional DQN queda tal que así:

```
def train(self,step_counter):
      if self.memory.size() < self.train_start:
         return
      states_tensor,actions_tensor,rewards_tensor,next_states_tensor,
         dones_tensor = self.memory.sample(self.batch_size,self.device)
      dist,values = self.model(states_tensor)
      values = values.squeeze()
      log_probs = dist.log_prob(actions_tensor)

      with torch.no_grad():
         _,next_values = self.model(next_states_tensor)
         next_values[dones_tensor] = 0.0
         next_values = next_values.detach().squeeze()
         error = rewards_tensor + self.discount_factor *next_values
      advantage = error-values
      actor_loss = -(log_probs * advantage.detach()).mean()
      critic_loss = F.mse_loss(values, error)
      total_loss = (actor_loss + critic_loss).mean()

      self.optimizer.zero_grad()
      total_loss.backward()
      self.optimizer.step()
```

CAPÍTULO 8

Dueling Networks

8.1– Introducción

Rainbow DQN combina varias mejoras significativas sobre el DQN básico, y las Dueling Networks representan una de estas innovaciones clave. Estas redes se centran en descomponer la función Q en dos componentes distintos: la función de valor y la función de ventaja, lo que permite al agente distinguir entre el valor de estar en un estado particular y el valor de realizar una acción en ese estado, la intuición básica detrás de esta mejora, es que la función valor estima cual debería ser la proxima acción del agente y la función de ventaja valora como de buena ha sido la estimación realizada por la función valor [6].

8.2– Fundamentos Dueling Networks

8.2.1. Definición de Funciones de Valor y Ventaja

En el contexto del Dueling DQN, se descompone la función de valor-acción $Q(s, a)$ en dos componentes: la función de valor $V(s)$ y la función de ventaja $A(s, a)$. La función de valor estima la cantidad de recompensa esperada al estar en un estado s, mientras que la función de ventaja mide cuánto mejor es tomar una acción específica a en comparación con otras posibles en el mismo estado.

Ecuaciones Básicas

En un modelo Dueling DQN, descomponemos la función de valor-acción $Q(s, a)$, que estima el valor esperado en términos de recompensa futura, al tomar una acción a en un estado s. Esta función se divide en dos componentes:

- **Función de Valor** $V(s)$**:** Estima cuán bueno es estar en el estado s, sin considerar qué acción se tomará.
- **Función de Ventaja** $A(s, a)$**:** Mide cuánto mejor es tomar la acción a comparada con otras acciones posibles en el mismo estado.

Matemáticamente, la función de valor-acción se representa inicialmente como:

$$Q(s, a) = V(s) + A(s, a)$$

Problema de Identificabilidad

Esta representación inicial no permite identificar de manera única las funciones $V(s)$ y $A(s,a)$ a partir de $Q(s,a)$. Esto se debe a que modificar $V(s)$ agregando o restando una constante, y haciendo el ajuste opuesto en $A(s,a)$, no altera el resultado de $Q(s,a)$. Esto conduce a múltiples combinaciones posibles de V y A que podrían explicar el mismo Q.

Solución: Ajuste para la Identificabilidad

Para hacer el modelo identificable y mejorar la precisión en la estimación de $V(s)$ y $A(s,a)$, se modifica la forma en que se calcula Q [6]:

$$Q(s,a) = V(s) + \left(A(s,a) - \frac{1}{|\mathcal{A}|} \sum_{a'} A(s,a') \right)$$

donde $|\mathcal{A}|$ es el número total de acciones posibles. Esta modificación garantiza que el promedio de las ventajas de todas las acciones en un estado sea cero, lo que hace que $V(s)$ refleje verdaderamente el valor de estar en el estado s, independientemente de la acción tomada. Esto permite determinar de manera única $V(s)$ y $A(s,a)$ desde $Q(s,a)$.

Esta modificación no solo soluciona el problema de identificabilidad sino que también mejora la estabilidad del entrenamiento del modelo, permitiendo un aprendizaje más efectivo del valor inherente de los estados y cómo las acciones influyen en esos valores.

8.2.2. Descomposición de la Red Neuronal

En la arquitectura Dueling DQN, la red se bifurca en dos caminos al final: uno para la estimación del valor del estado y otro para la estimación de la ventaja de cada acción. Esta bifurcación permite que la red aprenda más eficientemente cuándo las acciones no tienen un impacto significativo en el entorno.

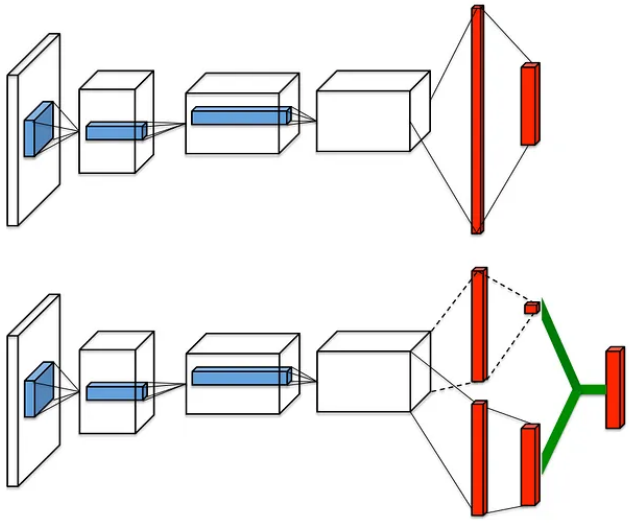

Figura 8.1: Arriba: arquitectura para DQN. Abajo: arquitectura para Dueling DQN [6]

Las Dueling Networks introducen una mejora significativa en la forma en que se estiman los valores Q, proporcionando varios beneficios:

- **Eficiencia en la Estimación:** Al separar la estimación del valor del estado de la del valor de las acciones, Dueling Networks permite una asignación de recursos más eficiente, especialmente en estados donde la elección de la acción es menos crítica.
- **Mejora en la Política de Selección de Acciones:** Esta arquitectura facilita el aprendizaje de políticas de acción más sofisticadas, al poder distinguir claramente entre el valor de los estados y las acciones.
- **Estabilidad en el Entrenamiento:** La solución al problema de identificabilidad contribuye a un entrenamiento más estable y a una convergencia más rápida, ya que reduce la ambigüedad en la asignación de valor a estados y acciones.

8.3– Conclusión

Las Dueling Networks representan un paso adelante en el diseño de arquitecturas de redes neuronales para el aprendizaje por refuerzo. Al descomponer la función Q en componentes de valor y ventaja , esta innovación permite un aprendizaje más eficiente y políticas de selección de acciones más precisas. Su integración en el marco de Rainbow DQN destaca la importancia de las mejoras en la evolución de los algoritmos de aprendizaje por refuerzo, prometiendo aplicaciones aún más robustas y efectivas en el futuro.

8.4– Implementación

En esta sección vamos a explicar la implementación del algoritmo de Advantage Actor Critic (A2C), el cual es uno de los distintos algoritmos que hay con una arquitectura de Dueling Networks, para añadir una nueva mejora para cada vez acercarnos más al algoritmo de Rainbow-DQN, A2C es una metodología de aprendizaje por refuerzo que utiliza dos componentes principales: un *actor*, que sugiere acciones a tomar, y un *crítico*, que evalúa las acciones tomadas por el actor. A continuación se describe el flujo del algoritmo presentado en el pseudocódigo:

1. Se inicializan los parámetros de los modelos del actor θ y del crítico θ^-, así como un contador global $T = 0$. Cada hilo de ejecución mantiene su propio contador de pasos t, inicializado en 0, y un acumulador de gradientes $d\theta$, también inicializado en 0. Se obtiene el estado inicial s.
2. Se entra en un bucle que se ejecutará hasta que el contador global T exceda un máximo predefinido $T_{\max}$.
3. Dentro del bucle, se toma una acción a utilizando una política ϵ-greedy basada en la función de valor acción $Q(s, a; \theta)$. Esto implica una exploración del espacio de acciones, donde ϵ define la probabilidad de tomar una acción aleatoria.
4. Tras realizar la acción, se recibe un nuevo estado s' y una recompensa r. Se utiliza una actualización basada en TD-error, donde y se calcula de una forma para estados terminales, y de otra para estados no terminales, utilizando el valor acción estimado para el siguiente estado y la acción bajo los parámetros del crítico θ^-.
5. Se actualiza el contador global T.

Algorithm 1 Advantage Actor-Critic (A2C)

1: //Assume global shared $\theta, \theta^-, and\ counter\ T = 0$.
2: Initialize thread step counter $t \leftarrow 0, \theta^- \leftarrow \theta, d\theta \leftarrow 0$.
Get initial state s.
3: **repeat**
Take action a with ϵ-greedy policy base on Q$(s, a;\ \theta)$
4: Receive new state s' and reward r

$$y = \begin{cases} r & for\ terminal\ s' \\ r + \gamma max_{a'} Q(s', a'; \theta^-) & for\ non-terminal\ s' \end{cases}$$

$s = s'$
$T \leftarrow T + 1\ and\ t \leftarrow t + 1$
5: **If** $T\ mod\ I_{target} == 0$ **then**
Update the target network $\theta^- \leftarrow \theta$
end if
if t $mod\ I_{AsyncUpdate} ==$ 0 or s is terminal **then**
Perform asynchronous update of θ using $d\theta$.
Clear gradients $d\theta \leftarrow 0$.
end if
until $T > T_{max}$

Figura 8.2: A2C Pseudocódigo A2C[5]

6. Si se alcanza un número determinado de pasos I_{target}, se actualiza la red objetivo, sincronizando los parámetros del crítico con los del actor.

7. De manera similar, cada $I_{\text{AsyncUpdate}}$ pasos, o si el estado s es terminal, se realiza una actualización asíncrona de los parámetros del actor utilizando los gradientes acumulados $d\theta$, y se reinician los gradientes.

Este proceso permite que el actor aprenda a tomar mejores decisiones a través de las evaluaciones del crítico, y que el crítico mejore sus evaluaciones a medida que recibe retroalimentación de las acciones tomadas por el actor.

8.4.1. Modificación Red Neuronal

Como hemos visto anteriormente, para la implementación de Advantage Actor Critic (A2C), necesitamos la estimación de dos valores, la estimación de las acciones del agente (actor) y la estimación de la calidad de las acciones tomadas por el agente (crítico), para ello, debemos modificar nuestra red neuronal haciendo una bifurcación de como pasa la información a traves de nuestro red creando dos nuevas capas.

```
1    self.actor_layer = nn.Sequential(
2        nn.ReLU(),
3        nn.Linear(512, output_size),
4        )
5    self.critic_layer = nn.Sequential(
6        nn.ReLU(),
7        nn.Linear(512, 1),
8        )
```

Como se muestra en la imagen la capa del actor tendra una dimensión igual a la cantidad de acciones que puede realizar el agente, que en nuestro caso particular de `Ms Pacman` son 9 y la la capa del crítico tiene una dimensión de 1 que determina que tan buena o que tan mala es la predicción realizada por la capa del actor. Al tener una bifurcación también tendremos que modificar nuestra función `forward`, ya que nuestro modelo devolvera 2 valores.

```
def forward(self, x):
        x = self.convolutional_layers(x)
        x = x.view(x.size(0), -1)
        x = self.linear_layers(x)
        actor = self.actor_layer(x)
        value = self.critic_layer(x)
        distribution = Categorical(F.softmax(actor, dim=-1))
        return distribution, value
```

Estos son los dos únicos cambios que habría que realizar en nuesto modelo para implementar A2C, en la siguiente sección veremos como modificar la función train para entrenar a este modelo. El código de la red neuronal al completo queda tal que así:

```
class Neural_Network(nn.Module):
    def __init__(self, input_size, output_size, learning_rate):
        super(Neural_Network, self).__init__()

        # Definición de las capas
        self.convolutional_layers = nn.Sequential(
            nn.Conv2d(input_size[0], 32, kernel_size=8, stride=4),
            nn.ReLU(),
            nn.Conv2d(32, 64, kernel_size=4, stride=1),
            nn.ReLU(),
            nn.Conv2d(64, 64, kernel_size=3,stride=2),
            nn.ReLU()
        )
        convolutional_output = self._get_convolutional_output(input_size)

        self.linear_layers = nn.Sequential(
            nn.Flatten(),
            nn.Linear(convolutional_output, 512),

        )
        self.actor_layer = nn.Sequential(
            nn.ReLU(),
            nn.Linear(512, output_size)
        )
        self.critic_layer = nn.Sequential(
            nn.ReLU(),
            nn.Linear(512,1)
        )

    def _get_convolutional_output(self,input_size):
        o = self.convolutional_layers(torch.zeros(1,*input_size))
        return int(np.prod(o.size()))


```

```
    def forward(self, x):
        x = self.convolutional_layers(x).view(x.size()[0],-1)
        x = self.linear_layers(x)
        actor = self.actor_layer(x)
        value = self.critic_layer(x)
        distribution = Categorical(F.softmax(actor,dim=-1))
        return distribution,value

    def save_model(self, filename):
        torch.save(self.state_dict(), filename)

    def load_model(self, filename):
        self.load_state_dict(torch.load(filename))
```

8.4.2. Modificación función train

La principal diferencia a la hora de entrenar el modelo, es que ahora nuestra red neuronal devuelve 2 valores y tenemos que calcular la ventaja (advantage). Primero se calcula el error temporal, que es la estimación del retorno del estado actual. Este error se calcula tomando la recompensa actual o inmediata (`rewards_tensor`) y sumándole el valor del siguiente estado (`next_values`) y multiplicandolo por el factor de descuento (`self.discount_factor`). La ventaja (`advantage`) se calcula como la diferencia entre el error calculado y el valor que la capa crítica ha asignado al estado actual.

```
    dist,value = self.model(state_tensor)
    values = values.squeeze()
    log_probs = dist.log_prob(actions_tensor)
        with torch.no_grad():
            _,next_values = self.model(next_state_tensor)
            next_values[done] = 0.0
            next_values = next_values.detach()
            error = reward + self.discount_factor *next_values
        advantage = error-value
```

El error para poder realizar el descenso del gradiente y entrenar el modelo se realiza de la siguiente forma, se calcula el error del actor y el error del crítico, una vez que se han obtenido estos 2 valores gracias al valor de la ventaja, se hace la media de estos 2 valores, y es el error que se le proorciona al modelo para poder entrenarlo.

```
    actor_loss = -(log_probs * advantage.detach())
    critic_loss = F.mse_loss(values,error)
    total_loss = (actor_loss + critic_loss).mean()
```

La ventaja es una señal crucial porque informa al actor cuánto mejor o peor fue la acción tomada en comparación con lo que se esperaba. Si la ventaja es positiva, se incentivará al actor a tomar esa acción más a menudo en el futuro; si es negativa, se desincentivará. Finalmente así queda el código de la implementación Advantage Actor Critic (A2C).

```
    def train(self,step_counter):

        if self.memory.size() < self.train_start:
            return
```

```
5
6       states_tensor,actions_tensor,rewards_tensor,next_states_tensor,
            dones_tensor = self.memory.sample(self.batch_size,self.device)
7
8       dist,values = self.model(states_tensor)
9       values = values.squeeze()
10      log_probs = dist.log_prob(actions_tensor)
11
12      with torch.no_grad():
13          _,next_values = self.model(next_states_tensor)
14          next_values[dones_tensor] = 0.0
15          next_values = next_values.detach().squeeze()
16          error = rewards_tensor + self.discount_factor *next_values
17
18      advantage = error-values
19
20      actor_loss = -(log_probs * advantage.detach()).mean()
21      critic_loss = F.mse_loss(values, error)
22      total_loss = actor_loss + critic_loss
23
24
25
26      self.optimizer.zero_grad()
27      total_loss.backward()
28      self.optimizer.step()
```

CAPÍTULO 9

Noisy Networks

9.1– Introducción

Las Noisy Networks (Redes Ruidosas) presentan un nuevo enfoque en la exploración en el aprendizaje por refuerzo, integrando el ruido directamente en los pesos de la red neuronal para facilitar la exploración eficiente. El ruido, en el contexto de aprendizaje por refuerzo y redes neuronales, se refiere a la adición de valores aleatorios a los pesos de la red durante el entrenamiento. Esta adición induce variabilidad en las acciones seleccionadas por la red. Esta metodología difiere de las técnicas de exploración convencionales, como epsilon-greedy y la regularización de entropía, al permitir una adaptación dinámica del nivel de exploración durante el entrenamiento [8].

9.2– Fundamentos Noisy Networks

La innovación clave en las Noisy Networks es la introducción de la capa lineal ruidosa, que incorpora ruido paramétrico directamente en los pesos y sesgos de la red. El ruido paramétrico es un tipo específico de ruido que se introduce en los parámetros de un modelo, a diferencia del ruido aditivo que simplemente se suma a la salida o a las entradas de una red, esto significa que los valores de los pesos se modifican aleatoriamente según una distribución específica, como una distribución gaussiana. Esta sección describe la construcción y función de una capa lineal ruidosa dentro de la arquitectura de NoisyNet, proporcionando una nueva herramienta para mejorar la exploración en el aprendizaje por refuerzo.

9.2.1. Definición

Tradicionalmente, una capa lineal en redes neuronales se define por la ecuación $y = Wx + b$, donde W y b son, respectivamente, la matriz de pesos y el vector de sesgos, y x es el vector de entrada. En contraste, la capa lineal ruidosa modifica esta relación añadiendo términos de ruido paramétrico a los pesos y sesgos, Noisy Networks for Exploration [8]:

$$y = (\mu^W + \sigma^W \odot \epsilon^W)x + \mu^b + \sigma^b \odot \epsilon^b$$

Aquí, μ^W y μ^b representan los componentes deterministas (pesos y sesgos medios), mientras que σ^W y σ^b son sus correspondientes desviaciones estándar que multiplican el ruido ϵ^W y ϵ^b. Este ruido se muestrea de distribuciones probables, generalmente normales, permitiendo variaciones estocásticas en la función de la capa.

9.2.2. Metodologías para la Introducción de Ruido en Noisy Networks

Una característica distintiva de las Noisy Networks es la metodología utilizada para introducir ruido en los parámetros de la red. Se experimentan principalmente con dos enfoques: Ruido Gaussiano Independiente y Ruido Gaussiano Factorizado. Ambos métodos tienen como objetivo mejorar la exploración al inducir variabilidad en la toma de decisiones del agente, pero difieren en su implementación y efectos potenciales sobre el aprendizaje.

Impacto en la Exploración

La introducción de ruido en los parámetros de la red permite una forma única de exploración. A diferencia de las técnicas convencionales de exploración, como el método epsilon-greedy que introduce aleatoriedad en la selección de acciones, el ruido en los parámetros de la red induce una política de exploración que es coherente durante múltiples decisiones y se adapta a lo largo del tiempo. Este método promueve la exploración de estados y acciones de una manera que puede ser más sutil y efectiva, ya que el agente aprende a navegar el entorno con una política que inherentemente incluye exploración.

Ventajas y Desafíos

La capacidad de las NoisyNets para aprender la magnitud óptima del ruido presenta claras ventajas en términos de eficiencia exploratoria. Sin embargo, este enfoque también introduce nuevos desafíos, como la necesidad de balancear adecuadamente el aprendizaje de los parámetros medios y de ruido para evitar una exploración excesiva o insuficiente. La calibración de estos parámetros es crucial para maximizar el rendimiento del aprendizaje.

Ruido Gaussiano Independiente

El Ruido Gaussiano Independiente introduce variabilidad en cada peso de la capa ruidosa de forma independiente. En este enfoque, cada peso w en la red tiene asociado su propio μ y σ, que son aprendidos por el modelo durante el entrenamiento. Esto permite que cada conexión en la red tenga su propio nivel de ruido, adaptándose dinámicamente según lo que el modelo aprenda que es óptimo.

$$W' = W + \epsilon \cdot \sigma,$$

donde W' son los pesos ajustados por ruido, W son los pesos originales, ϵ es una muestra de una distribución gaussiana y σ es la desviación estándar aprendida por el modelo.

Ruido Gaussiano Factorizado

A diferencia del enfoque independiente, el Ruido Gaussiano Factorizado reduce la dimensionalidad del ruido mediante la utilización de dos vectores: uno correspondiente a la longitud de entrada (f_{in}) y otro a la longitud de salida (f_{out}) de la capa. Estos vectores

son transformados mediante una función especial y su producto matricial resulta en una matriz de ruido que se suma a los pesos originales de la capa.

$$W' = W + \sigma \cdot (f_{in} \odot f_{out}),$$

donde W' son los pesos ajustados por ruido, W son los pesos originales, σ es la desviación estándar global aprendida, y $\odot$ denota el producto matricial de los vectores ruidosos transformados correspondientes a la entrada y salida.

9.3– Conclusión

La capa lineal ruidosa representa un avance significativo en el diseño de redes neuronales para el aprendizaje por refuerzo. Al integrar la exploración directamente en la arquitectura de la red, las NoisyNets ofrecen un enfoque poderoso y flexible para aprender en entornos complejos y no deterministas. Esta técnica no solo mejora la capacidad del modelo para explorar el espacio de estados y acciones sino que también contribuye a una convergencia más rápida hacia políticas óptimas, destacando el potencial de las Noisy Networks en la mejora del aprendizaje por refuerzo.

9.4– Implementación Práctica de Noisy Networks

La implementación de las Noisy Networks o capas ruidosas en español para mejorar la exploración en nuestro modelo es muy sencillo de implementar ya que Pytorch ofrece soporte para este tipos de capas para formar nuestra red neuronal. En nuestro modelo específico, la clase `NoisyLinear` juega un papel crucial en la implementación de Noisy Networks, permitiéndonos introducir ruido paramétrico en las capas finales de la red neuronal. Esta clase redefine la tradicional capa lineal para incluir ruido en los parámetros de los pesos y los sesgos, lo que induce una exploración intrínseca durante el entrenamiento del modelo.

9.4.1. Clase NoisyLinear

La clase `NoisyLinear`, derivada de `nn.Module` de PyTorch, introduce dos conjuntos de parámetros: μ (mu) y σ (sigma), para cada peso y sesgo en la capa. Estos parámetros representan, respectivamente, el valor central y la desviación estándar del ruido que se aplicará a los pesos y sesgos. Además, se generan valores ϵ (epsilon) para modelar el ruido aplicado durante cada paso de forward.

9.4.2. Inicialización y Generación de Ruido

Durante la inicialización (`reset_parameters`), los parámetros μ se inicializan uniformemente dentro de un rango determinado por el número de características de entrada, y los parámetros σ se inicializan con un valor estándar definido por el usuario, ajustado por la raíz cuadrada del número de características de entrada. Este proceso asegura que el ruido inicial tenga una magnitud adecuada en relación con los valores de los pesos.

La generación de ruido se realiza a través del método `reset_noise`, que calcula ϵ para los pesos y sesgos. Este cálculo utiliza una función especial para escalar el ruido, asegurando que su distribución sea adecuada para la operación de multiplicación matricial que se realiza durante el forward.

9.4.3. Función Forward NoisyLinear

La función de forward decide si utilizar los parámetros ruidosos o los parámetros centrales, dependiendo de si el modelo está en modo de entrenamiento o evaluación, aunque se ha demostrado que no hay una diferencias significativa entre el rendimiento de estas dos modalidades. En el modo de entrenamiento, se aplica el ruido a los pesos y sesgos, utilizando los valores ϵ generados, para promover la exploración. En el modo de evaluación, se utilizan los parámetros μ sin ruido, permitiendo que el modelo explote lo aprendido.

Este es el código de Pytorch de las capas lineales con exploración con ruido, aunque no hace falta y se puede importar directamente, se ha descargado y añadido en el proyecto para entender su funcionamiento y poder realizar la explicación anterior.

```
class NoisyLinear(nn.Module):
 def __init__(self, in_features, out_features, std_init=0.5):
   super(NoisyLinear, self).__init__()
   self.in_features = in_features
   self.out_features = out_features
   self.std_init = std_init

   self.weight_mu = nn.Parameter(torch.empty(out_features, in_features))
   self.weight_sigma = nn.Parameter(torch.empty(out_features, in_features))
   self.register_buffer('weight_epsilon', torch.empty(out_features, in_features)
       )
   self.bias_mu = nn.Parameter(torch.empty(out_features))
   self.bias_sigma = nn.Parameter(torch.empty(out_features))
   self.register_buffer('bias_epsilon', torch.empty(out_features))
   self.reset_parameters()
   self.reset_noise()


 def reset_parameters(self):
   mu_range = 1 / math.sqrt(self.in_features)
   self.weight_mu.data.uniform_(-mu_range, mu_range)
   self.weight_sigma.data.fill_(self.std_init / math.sqrt(self.in_features))
   self.bias_mu.data.uniform_(-mu_range, mu_range)
   self.bias_sigma.data.fill_(self.std_init / math.sqrt(self.out_features))

 def _scale_noise(self, size):
   x = torch.randn(size, device=self.weight_mu.device)
   return x.sign().mul_(x.abs().sqrt_())

 def reset_noise(self):
   epsilon_in = self._scale_noise(self.in_features)
   epsilon_out = self._scale_noise(self.out_features)
   self.weight_epsilon.copy_(epsilon_out.ger(epsilon_in))
   self.bias_epsilon.copy_(epsilon_out)

 def forward(self, input):
   if self.training:

     return F.linear(input, self.weight_mu + self.weight_sigma * self.
         weight_epsilon, self.bias_mu + self.bias_sigma * self.bias_epsilon)
   else:
```

```
40
41      return F.linear(input, self.weight_mu, self.bias_mu)
```

9.4.4. Aplicación en la Arquitectura del Modelo

La incorporación de la clase `NoisyLinear` en nuestra red neuronal se realiza reemplazando las capas lineales estándar en las últimas capas de la red, específicamente en las capas que forman los componentes actor y crítico del modelo.

```
1     self.linear_layers = nn.Sequential(
2         nn.Flatten(),
3         NoisyLinear(convolutional_output,512),
4         )
5     self.actor_layer = nn.Sequential(
6         nn.ReLU(),
7         NoisyLinear(512,output_size)
8         )
9     self.critic_layer = nn.Sequential(
10        nn.ReLU(),
11        NoisyLinear(512,1)
12        )
```

CAPÍTULO 10

Prioritized Experience Replay

10.1– Introducción

El Prioritized Experience Replay (PER) representa una mejora significativa en la eficiencia del aprendizaje por refuerzo, al introducir una metodología para priorizar qué experiencias se repasan durante el entrenamiento. A diferencia del enfoque tradicional de muestreo uniforme de un búfer de replay, PER selecciona experiencias basándose en la magnitud de su error de predicción, permitiendo que el agente se concentre en aquellas experiencias de las que puede aprender más [9].

10.2– Fundamentos PER

10.2.1. Experience Replay

Antes de profundizar en el Prioritized Experience Replay, es crucial entender el concepto de Experience Replay y su papel fundamental en el aprendizaje por refuerzo. Experience Replay es una técnica que rompe las correlaciones secuenciales en las observaciones del agente, al almacenar experiencias pasadas en un búfer de replay y muestrear aleatoriamente de este conjunto para el entrenamiento. Esto permite que el agente aprenda de experiencias anteriores, potencialmente múltiples veces, mejorando la eficiencia del aprendizaje y la estabilidad del entrenamiento.

Búfer de Replay

El búfer de replay, también conocido como replay memory, almacena una colección de experiencias que ha realizado el agente anteriormente. Cada experiencia está compuesta típicamente por una tupla que incluye el estado actual, la acción tomada, la recompensa recibida, y el siguiente estado (s, a, r, s'). El tamaño del búfer puede variar, pero debe ser suficientemente grande como para almacenar una amplia gama de experiencias.

Muestreo Uniforme

En el enfoque tradicional de Experience Replay, las experiencias se muestrean uniformemente del búfer de replay. Esto significa que cada experiencia tiene la misma probabilidad

de ser seleccionada para el entrenamiento, independientemente de la relevancia o el impacto potencial de la experiencia en el aprendizaje del agente.

Ventajas de Experience Replay

- **Eficiencia de Datos:** Permite al agente reutilizar experiencias previas para el aprendizaje, maximizando la utilidad de cada experiencia.
- **Estabilidad de Aprendizaje:** Reduce la varianza en los datos de entrenamiento y ayuda a prevenir oscilaciones o divergencias en los parámetros del modelo debido a la correlación de las experiencias secuenciales.

Limitaciones

A pesar de sus ventajas, el enfoque de muestreo uniforme en el Experience Replay tradicional no considera la relevancia o importancia de cada experiencia, lo que puede llevar a una eficiencia subóptima en el aprendizaje. Algunas experiencias pueden ser más informativas que otras, y muestrearlas con igual probabilidad podría resultar en un proceso de aprendizaje más lento.

La limitación principal del muestreo uniforme en Experience Replay motivó el desarrollo de técnicas más avanzadas, como el Prioritized Experience Replay. PER aborda esta limitación al introducir un método para priorizar experiencias basándose en su importancia para el aprendizaje, lo que permite al agente enfocarse en aprender de las experiencias más significativas.

10.2.2. Fundamentos de Prioritized Experience Replay

Priorización de Experiencias

La clave de PER radica en cómo se priorizan las experiencias para el replay. Las experiencias se ponderan según la magnitud de su error de predicción (Temporal Difference), con la idea de que experiencias con errores mayores indican una discrepancia significativa entre las expectativas del modelo y la realidad, representando así oportunidades valiosas de aprendizaje, se puede definir la prioridad con la que se muestrea cada experiencia se puede definir de la siguiente forma [9]:

$$P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha}$$

donde $P(i)$ es la probabilidad de muestrear la i-ésima experiencia, p_i es la prioridad de la experiencia, y α es un parámetro que determina qué tan fuerte es la priorización.

Beneficios de Prioritized Experience Replay

- **Aprendizaje Acelerado:** Al centrarse en experiencias significativas, PER puede acelerar el aprendizaje.
- **Eficiencia de Muestreo:** Mejora la eficiencia del uso de experiencias almacenadas, aprendiendo más de menos datos.
- **Mejora en la Estabilidad:** Aunque introduce sesgo en el proceso de muestreo, este puede ser compensado adecuadamente, mejorando la estabilidad del aprendizaje.

Desafíos

A pesar de sus ventajas, la implementación de PER no está exenta de desafíos, como la computación adicional requerida para mantener y muestrear según las prioridades. Sin embargo, las mejoras en la eficiencia del aprendizaje suelen compensar estos costes adicionales.

10.3– Conclusión

Prioritized Experience Replay es una técnica poderosa que, al permitir que el agente de aprendizaje por refuerzo se enfoque en las experiencias más informativas, representa un avance significativo en la eficiencia y efectividad del aprendizaje. Su integración en algoritmos modernos de aprendizaje por refuerzo, como parte del conjunto de técnicas en Rainbow DQN, destaca la importancia de estrategias sofisticadas de muestreo en el avance del campo.

10.4– Implementación

En esta parte del capítulo se explicará la realización de la implementación práctica de una de las mejoras que conforman el algoritmo de Rainbow DQN, Prioritized Experience Replay (PER). Primero explicaremos este pseudocódigo que aplica PER al algoritmo visto anteriormente Double DQN:

Algorithm 1 Double DQN with proportional prioritization

Input: minibatch k, step-size η, replay period K and size N, exponents α and β, budget T.
Initialize replay memory $\mathcal{H} = \emptyset$, $\Delta = 0$, $p_1 = 1$
Observe S_0 and choose $A_0 \sim \pi_\theta(S_0)$
for $t = 1$ **to** T **do**
  Observe S_t, R_t, γ_t
  Store transition $(S_{t-1}, A_{t-1}, R_t, \gamma_t, S_t)$ in $\mathcal{H}$ with maximal priority $p_t = \max_{i<t} p_i$
  if $t \equiv 0 \mod K$ **then**
    for $j = 1$ **to** k **do**
      Sample transition $j \sim P(j) = p_j^\alpha / \sum_i p_i^\alpha$
      Compute importance-sampling weight $w_j = (N \cdot P(j))^{-\beta} / \max_i w_i$
      Compute TD-error $\delta_j = R_j + \gamma_j Q_{\text{target}}(S_j, \arg\max_a Q(S_j, a)) - Q(S_{j-1}, A_{j-1})$
      Update transition priority $p_j \leftarrow |\delta_j|$
      Accumulate weight-change $\Delta \leftarrow \Delta + w_j \cdot \delta_j \cdot \nabla_\theta Q(S_{j-1}, A_{j-1})$
    end for
    Update weights $\theta \leftarrow \theta + \eta \cdot \Delta$, reset $\Delta = 0$
    From time to time copy weights into target network $\theta_{\text{target}} \leftarrow \theta$
  end if
  Choose action $A_t \sim \pi_\theta(S_t)$
end for

Figura 10.1: Pseudocódigo PER [9]

- **Entradas:** Recibe varios parámetros incluyendo el tamaño del minibatch k, tamaño del paso η, período de replay K, tamaño del buffer de replay N, los exponentes α y β, y el presupuesto o número de pasos T.

- **Inicialización:** Configura la memoria de replay $\mathcal{H}$ como vacía, el cambio acumulado Δ como 0, y la prioridad inicial p_1 como 1. Además, inicializa el valor Q y la política π.

- **Observar el estado inicial y elegir la primera acción.**

- **Bucle de tiempo t de 1 a T:** Ejecuta los pasos por cada momento en un episodio.

    - Observar la transición: Incluye el estado S_t, la recompensa R_t, y el factor de descuento γ_t.
    - Almacenar la transición: Guarda la experiencia en la memoria de replay con la prioridad máxima observada hasta el momento.
    - Si t es múltiplo de K: Ejecuta la actualización basada en la memoria de replay.
        * Para cada muestra del minibatch: Muestrea transiciones de la memoria de replay.
        * Muestreo de transiciones: Usa priorización proporcional basada en α para determinar la probabilidad de muestreo de cada transición.
        * Calcular el peso de muestreo de importancia: Se usa para ajustar la actualización del valor Q y se escala por β para manejar el sesgo introducido por la priorización.
        * Calcular el error TD: La diferencia entre la recompensa observada y la estimación del valor Q actualizada por la recompensa futura esperada usando la red objetivo.
        * Actualizar la prioridad de la transición: La prioridad se basa en el valor absoluto del error TD.
        * Acumular el cambio de peso: Se guarda el producto del peso de muestreo, el error TD y el gradiente del valor Q.
    - Actualizar los pesos de la red: Aplica el cambio acumulado a los pesos de la red Q.
    - De vez en cuando, copiar los pesos a la red objetivo: Actualiza los pesos de la red objetivo con los de la red Q principal para estabilizar el aprendizaje.
    - Seleccionar la siguiente acción: Usando la política π basada en el estado actual.

10.5– Implementación (PER)

La implementación del Prioritized Experience Replay (PER) permite almacenar y muestrear experiencias con una prioridad asignada, mejorando la eficiencia del aprendizaje en algoritmos de refuerzo.

La clase `Prioritized_Replay` gestiona un buffer de experiencias y permite agregar nuevas experiencias, muestrearlas y actualizar sus prioridades.

10.5.1. Inicialización

El constructor inicializa el buffer con una capacidad específica y define parámetros necesarios para gestionar las prioridades y el muestreo.

```
from collections import deque
import random
import numpy as np
import torch
import collections

Experience = collections.namedtuple("Experience", field_names=['state', 'action',
    'reward', 'done', 'next_state', 'priority'])

class Prioritized_Replay:
    def __init__(self, capacity):
        self.capacity = capacity
        self.len_buffer = 0
        self.buffer = np.empty(self.capacity, dtype=[("priority", np.float32), ("
            experience", Experience)])
        self.alpha = 0.01
```

- **capacity**: Capacidad máxima del buffer de experiencias.
- **len_buffer**: Longitud actual del buffer.
- **buffer**: Arreglo `numpy` para almacenar experiencias y sus prioridades.
- **alpha**: Parámetro que determina la influencia de las prioridades.

10.5.2. Añadir Experiencia

La función `add` añade una nueva experiencia al buffer, calculando su prioridad basada en la ventaja proporcionada.

```
    def add(self, exp, advantage):
        if isinstance(advantage, torch.Tensor):
            priority = advantage.detach().cpu().numpy()
            priority = np.abs(priority)
        if self.size() == self.capacity:
            if priority > self.buffer["priority"].min():
                idx = self.buffer["priority"].argmin()
                self.buffer[idx] = (priority, exp)
        else:
            self.buffer[self.size()] = (priority, exp)
            self.len_buffer += 1
```

- **advantage**: Ventaja utilizada para calcular la prioridad.
- La prioridad se calcula como el valor absoluto de la ventaja.
- Si el buffer está lleno, se reemplaza la experiencia con la prioridad más baja.
- Si el buffer no está lleno, se añade la nueva experiencia.

10.5.3. Tamaño del Buffer

La función `size` devuelve el número de experiencias actualmente almacenadas en el buffer.

```
    def size(self):
        return self.len_buffer
```

10.5.4. Limpiar el Buffer

La función `clear` vacía el buffer de experiencias.

```
    def clear(self):
        self.buffer.clear()
```

10.5.5. Muestreo de Experiencias

La función `sample` selecciona un conjunto de experiencias del buffer, basándose en sus prioridades.

```
    def sample(self, batch_size, device):
        batch_size = min(batch_size, self.size())
        priorities = self.buffer[:self.size()]["priority"]
        n_priorities = priorities ** self.alpha / np.sum(priorities ** self.alpha)

        idxs = np.random.choice(np.arange(priorities.size), size=batch_size,
            replace=True, p=n_priorities)
        experiences = self.buffer["experience"][idxs]

        states = torch.tensor(np.array([e.state for e in experiences if e is not
            None])).to(device)
        next_states = torch.tensor(np.array([e.next_state for e in experiences if
            e is not None])).to(device)
        actions = torch.tensor(np.array([e.action for e in experiences if e is not
             None], dtype=np.int64)).to(device)
        rewards = torch.tensor([e.reward for e in experiences if e is not None]).
            to(device)
        dones = torch.BoolTensor([e.done for e in experiences if e is not None]).
            to(device)

        return (idxs, states, actions, rewards, next_states, dones)
```

- **batch_size**: Número de experiencias a muestrear.
- **device**: Dispositivo de PyTorch (`cuda` o `cpu`) donde se realizarán los cálculos.
- Se normalizan las prioridades elevándolas a la potencia de `alpha`.
- Se seleccionan índices basados en las prioridades normalizadas.
- Se extraen las experiencias correspondientes a los índices seleccionados.
- Se convierten las experiencias a tensores de PyTorch y se devuelven.

10.5.6. Actualización de Prioridades

La función `update_priorities` actualiza las prioridades de las experiencias seleccionadas.

```
    def update_priorities(self, idxs, priorities):
        priorities = np.abs(priorities) ** self.alpha
        total = np.sum(priorities) ** self.alpha
        if total > 0:
            n_priorities = priorities / total
        else:
            n_priorities = np.ones_like(priorities) / len(priorities)

        self.buffer["priority"][idxs] = n_priorities
```

- **idxs**: Índices de las experiencias a actualizar.
- **priorities**: Nuevas prioridades para las experiencias.
- Las prioridades se actualizan basándose en el valor absoluto de las ventajas.

10.5.7. Función `append_sample` en el Agente

La función `append_sample` añade nuevas experiencias al buffer, calculando su error de diferencia temporal (TD) y determinando su prioridad antes de almacenarla.

```
def append_sample(self, state, action, reward, next_state, done):
    state_tensor = torch.tensor(state).unsqueeze(0).to(self.device)
    next_state_tensor = torch.tensor(next_state).unsqueeze(0).to(self.device)
    dist, value = self.model(state_tensor)

    with torch.no_grad():
        _, next_values = self.model(next_state_tensor)
        next_values[done] = 0.0
        next_values = next_values.detach()
        error = reward + self.discount_factor * next_values

    advantage = error - value
    advantage.detach().cpu().numpy()
    exp = Experience(state, action, reward, done, next_state)
    self.memory.add(exp, advantage)
    if self.epsilon > self.epsilon_min and self.memory.size() == self.train_start
        :
        self.epsilon *= self.epsilon_decay
```

- **state, action, reward, next_state, done**: Componentes de la nueva experiencia.
- **state_tensor, next_state_tensor**: Conversión del estado y próximo estado a tensores de PyTorch.
- **dist, value**: Distribución y valor predecido por el modelo para el estado actual.
- **next_values**: Valor predecido para el próximo estado, ajustado para terminales.

- **error**: Error de diferencia temporal (TD).
- **advantage**: Ventaja calculada como la diferencia entre el error y el valor actual.
- La experiencia se añade al buffer con la prioridad basada en la ventaja.
- **epsilon**: Parámetro de exploración que se actualiza si el buffer ha alcanzado el tamaño de entrenamiento.

10.5.8. Función `train` en el Agente

Durante el entrenamiento, se seleccionan experiencias basadas en sus prioridades y se actualizan las prioridades después de calcular el error de diferencia temporal (TD).

```
def train(self, step_counter):
    if self.memory.size() < self.train_start or step_counter % 10 == 0:
        return
    idxs, states_tensor, actions_tensor, rewards_tensor, next_states_tensor,
        dones_tensor = self.memory.sample(self.batch_size, self.device)

    dist, values = self.model(states_tensor)
    values = values.squeeze()
    log_probs = dist.log_prob(actions_tensor)

    with torch.no_grad():
        _, next_values = self.model(next_states_tensor)
        next_values[dones_tensor] = 0.0
        next_values = next_values.detach().squeeze()
        error = rewards_tensor + self.discount_factor * next_values

    advantage = error - values
    actor_loss = -(log_probs * advantage.detach())
    actor_loss_mean = actor_loss.mean()
    critic_loss = advantage ** 2

    total_loss = (actor_loss_mean + critic_loss)
    total_loss_mean = total_loss.mean()

    probabilities = total_loss.detach().cpu().numpy()
    for i in range(self.batch_size):
        idx = idxs[i]
        self.memory.update_priorities(idx, probabilities[i])

    self.optimizer.zero_grad()
    total_loss_mean.backward()
    self.optimizer.step()
```

- **step_counter**: Contador de pasos de entrenamiento.
- **idxs, states_tensor, actions_tensor, rewards_tensor, next_states_tensor, dones_tensor**: Experiencias muestreadas.
- **dist, values**: Distribución y valores predichos por el modelo para los estados actuales.

- **next_values**: Valores predichos para los próximos estados, ajustados para terminales.
- **error**: Error de diferencia temporal (TD).
- **advantage**: Ventaja calculada como la diferencia entre el error y los valores actuales.
- **actor_loss, critic_loss**: Pérdidas del actor y crítico calculadas a partir de las ventajas.
- **total_loss**: Pérdida total combinada.
- **probabilities**: Probabilidades utilizadas para actualizar las prioridades de las experiencias muestreadas.
- Las prioridades se actualizan y el optimizador se ajusta según la pérdida total.

CAPÍTULO 11

N-Step Learning

11.1– Introducción

El aprendizaje de n-step es una técnica en el aprendizaje por refuerzo que extiende la idea del aprendizaje de un paso, considerando las recompensas de varios pasos en el futuro para actualizar las estimaciones del valor de un estado o acción. Esta metodología permite una aproximación más precisa del retorno esperado, equilibrando la vista inmediata y a largo plazo de las consecuencias de las acciones [18].

11.2– Fundamentos del Aprendizaje de N-Step

Aprendizaje de Un Paso

Tradicionalmente, en el aprendizaje por refuerzo, las actualizaciones del valor estimado se basan en la recompensa inmediata más el valor del siguiente estado, descontado por un factor γ. Esta aproximación, conocida como aprendizaje de un paso, puede ser limitante al no considerar adecuadamente el efecto de las acciones futuras más allá del siguiente paso.

Extensión a N-Step

El aprendizaje de n-step extiende esta idea al calcular la recompensa total esperada considerando las próximas n recompensas. Esto no solo incorpora más información sobre el futuro en la actualización del valor, sino que también puede acelerar el aprendizaje al propagar las recompensas de regreso al estado actual más rápidamente.

Retorno de N-Step

El retorno de n-step se define como la suma de las recompensas futuras descontadas hasta n pasos adelante, más el valor estimado del estado después de n pasos, también descontado adecuadamente. Matemáticamente, se expresa como [16]:

$$G_t^{(n)} = R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{n-1} R_{t+n} + \gamma^n V(S_{t+n})$$

donde $G_t^{(n)}$ es el retorno de n-step, R_{t+k} es la recompensa recibida después de k pasos, y $V(S_{t+n})$ es el valor estimado del estado después de n pasos.

Balance entre Sesgo y Varianza

Una de las ventajas clave del aprendizaje de n-step es su capacidad para balancear el sesgo y la varianza en las estimaciones de valor. Mientras que el aprendizaje de un paso puede tener un sesgo bajo pero una alta varianza, y el aprendizaje de monte carlo (considerando retornos hasta el final del episodio) tiene alta varianza pero bajo sesgo, el aprendizaje de n-step permite encontrar un punto medio ajustando n.

Beneficios del Aprendizaje de N-Step

- **Aprendizaje Acelerado:** Al considerar las recompensas futuras más inmediatamente, el aprendizaje de n-step puede acelerar la convergencia hacia la política óptima.

- **Mejora en la Estabilidad:** Proporciona un mejor balance entre sesgo y varianza, lo que puede resultar en una mayor estabilidad durante el entrenamiento.

Desafíos y Consideraciones

La elección del valor de n es crítica, ya que diferentes entornos y tareas pueden beneficiarse de diferentes horizontes de planeación. Además, la implementación de n-step requiere una gestión cuidadosa de la memoria y la computación, especialmente en entornos con episodios largos.

11.3– Conclusión

El aprendizaje de n-step representa una extensión poderosa y flexible del aprendizaje de un paso en el aprendizaje por refuerzo, ofreciendo mejoras significativas en la eficiencia del aprendizaje y la calidad de las políticas aprendidas. Su capacidad para equilibrar el sesgo y la varianza lo hace especialmente valioso en la práctica del aprendizaje por refuerzo moderno.

11.4– Implementación

En esta sección, describiremos el algoritmo de n-step Temporal Difference (TD) learning. Este enfoque extiende la actualización de valores de un solo paso a una secuencia de n pasos, lo que permite una estimación más precisa del valor futuro y un aprendizaje más eficiente. A continuación, se muestra el pseudocódigo para la estimación de la función de valor V utilizando n-step TD learning.

Inicialización:

- Inicializar los valores $V(s)$ para todos los estados s de manera arbitraria.

- Las operaciones de almacenamiento y acceso para S_t y R_t pueden tomar su índice módulo $n + 1$.

n-step TD for estimating $V \approx v_\pi$

Input: a policy π
Algorithm parameters: step size $\alpha \in (0, 1]$, a positive integer n
Initialize $V(s)$ arbitrarily, for all $s \in \mathcal{S}$
All store and access operations (for S_t and R_t) can take their index mod $n + 1$

Loop for each episode:
  Initialize and store $S_0 \neq$ terminal
  $T \leftarrow \infty$
  Loop for $t = 0, 1, 2, \ldots$:
  | If $t < T$, then:
  | Take an action according to $\pi(\cdot|S_t)$
  | Observe and store the next reward as R_{t+1} and the next state as S_{t+1}
  | If S_{t+1} is terminal, then $T \leftarrow t + 1$
  | $\tau \leftarrow t - n + 1$ (τ is the time whose state's estimate is being updated)
  | If $\tau \geq 0$:
  | $G \leftarrow \sum_{i=\tau+1}^{\min(\tau+n,T)} \gamma^{i-\tau-1} R_i$
  | If $\tau + n < T$, then: $G \leftarrow G + \gamma^n V(S_{\tau+n})$ $(G_{\tau:\tau+n})$
  | $V(S_\tau) \leftarrow V(S_\tau) + \alpha\,[G - V(S_\tau)]$
  Until $\tau = T - 1$

Figura 11.1: n-step TD para estimar $V \approx v_\pi$ [16]

Para cada episodio:

1. Inicializar y almacenar el estado inicial S_0 que no es terminal.
2. Establecer T a infinito.
3. **Para** cada paso temporal $t = 0, 1, 2, \ldots$ **hacer**:
   - **Si** $t < T$, entonces:
     - Tomar una acción de acuerdo a la política $\pi(\cdot|S_t)$.
     - Observar y almacenar la recompensa resultante R_{t+1} y el siguiente estado S_{t+1}.
     - **Si** S_{t+1} es terminal, entonces $T \leftarrow t + 1$.
   - Establecer $\tau \leftarrow t - n + 1$. ($\tau$ es el tiempo en el que se actualizará el valor del estado.)
   - **Si** $\tau \geq 0$, entonces:
     - Calcular el retorno G para el paso n: $G \leftarrow \sum_{i=\tau+1}^{\text{mín}(\tau+n,T)} \gamma^{i-\tau-1} R_i$.
     - **Si** $\tau + n < T$, entonces $G \leftarrow G + \gamma^n V(S_{\tau+n})$.
     - Actualizar el valor $V(S_\tau) \leftarrow V(S_\tau) + \alpha[G - V(S_\tau)]$.
4. **Hasta que** $\tau = T - 1$.

Notas:

- El algoritmo utiliza un enfoque iterativo, donde el valor de cada estado se actualiza considerando una secuencia de n recompensas futuras.

- La actualización de valores se realiza solo después de que se ha procesado una secuencia completa de n pasos, o cuando se alcanza un estado terminal.

- El factor de descuento γ y el tamaño del paso α son hiperparámetros que pueden ajustarse según el entorno y la tarea específica.

11.5– Implementación Práctica

Para la implementación N-step learning necesitaremos un buffer para almacenar la información de las n-experiencias y poder calcular las recompensas acumuladas de estas mismas, para ellos definimos dos nuevos hiperparámetros, unos para indicar el tamaño de n y otro para inicializar el buffer.

```
    self.n_step = 2
    self.n_step_buffer = deque(maxlen=self.n_step)
```

11.5.1. Función `calculate_n_step_info`

La función `calculate_n_step_info` calcula el retorno descontado acumulado de una serie de pasos (n pasos) adelante en el entorno. La función itera a través del búfer `self.n_step_buffer` que contiene una serie de tuplas con la forma (estado, acción, recompensa, siguiente estado, terminado). Para cada elemento en el búfer, se acumula la recompensa descontada utilizando un factor de descuento γ elevado a la potencia del índice del paso, comenzando en 0 hasta $n-1$. El índice representa el número de pasos adelante desde el estado actual. Se puede definir de la siguiente forma [16]:

$$R_{t:t+n} = \sum_{i=0}^{n-1} \gamma^i \cdot r_{t+i}$$

La función finalmente devuelve el retorno acumulado y la información del último estado y si es un estado terminal, que son cruciales para la actualización de la función de valor.

La razón para utilizar el aprendizaje de n-pasos es que puede ofrecer una mejor estimación del retorno futuro que simplemente mirar la recompensa inmediata o esperar hasta el final del episodio. Al utilizar una vista de n-pasos adelante, se puede obtener una compensación entre la varianza y el sesgo de la estimación del retorno, potencialmente conduciendo a una convergencia más rápida y estable en el entrenamiento del modelo.

```
  def calculate_n_step_info(self):
      reward_n_step = 0
      for idx, (_, _, reward, _, _) in enumerate(self.n_step_buffer):
          reward_n_step += (self.discount_factor ** idx) * reward
      _, _, _, next_state_n_step, done_n_step = self.n_step_buffer[-1]

      return reward_n_step, next_state_n_step, done_n_step
```

11.5.2. Función `append_sample`

La función `append_sample` se encarga de administrar el proceso de acumulación de experiencias dentro de un entorno de aprendizaje por refuerzo. Este enfoque se beneficia del balance entre el aprendizaje a corto plazo y el aprendizaje que considera consecuencias a largo plazo de las acciones tomadas por el agente.

Al invocarse, la función añade una nueva experiencia, consistente en el estado actual, la acción tomada, la recompensa obtenida, el siguiente estado y una señal que indica si se ha llegado al final del episodio, al búfer de n-step. Si el búfer no ha alcanzado el tamaño requerido para un aprendizaje de n-step y el episodio aún no ha terminado, la función se interrumpe prematuramente. En caso contrario, procede a calcular la recompensa descontada acumulada y la información del estado resultante al cabo de los n pasos. Esta información se utiliza para actualizar la política o la función de valor del agente, ayudándole a optimizar su comportamiento en el entorno.

El aprendizaje de n-step ayuda al agente a mirar más allá de las recompensas inmediatas y considerar una estrategia más amplia que pueda llevar a mejores resultados a largo plazo. Este enfoque intenta equilibrar la estimación de la función valor.

```
1  def append_sample(self,state,action,reward,next_state,done):
2        self.n_step_buffer.append((state,action,reward,next_state,done))
3
4        if len(self.n_step_buffer)< self.n_step and not done:
5            return
6
7        reward_n_step, next_state_n_step, done_n_step = self.calculate_n_step_info
             ()
8        state_n_step, action_n_step = self.n_step_buffer[0][:2]
9        exp = Experience(state_n_step,action_n_step,reward_n_step,done_n_step,
             next_state_n_step)
10       state_tensor = torch.tensor(state_n_step).unsqueeze(0).to(self.device)
11       next_state_tensor = torch.tensor(next_state_n_step).unsqueeze(0).to(self.
             device)
12       dist,value = self.model(state_tensor)
13
14
15
16       with torch.no_grad():
17           _,next_values = self.model(next_state_tensor)
18           next_values[done] = 0.0
19           next_values = next_values.detach()
20           error = reward + self.discount_factor *next_values
21
22
23       advantage = error-value
24       advantage.detach().cpu().numpy()
25       exp = Experience(state_n_step,action_n_step,reward_n_step,done_n_step,
             next_state_n_step)
26       self.memory.add(exp,advantage)
27       if self.epsilon>self.epsilon_min and self.memory.size()==self.train_start:
28
29           self.epsilon*=self.epsilon_decay
```

CAPÍTULO 12

Experimentación

12.1– Introducción

En los capítulos anteriores, hemos desarrollado y detallado la arquitectura del modelo Rainbow DQN, así como su implementación práctica en el entorno del juego Ms Pacman.

Este capítulo está dedicado a la exploración de los hiperparámetros y variaciones estructurales dentro del modelo Rainbow DQN. A través de una serie de experimentos, buscaremos optimizar el rendimiento del modelo ajustando sus componentes individuales y su configuración general. La metodología se centrará en la alteración de los hiperparámetros clave y en la variación de las mejoras específicas que conforman Rainbow DQN.

Para evaluar y comparar el rendimiento de las diferentes configuraciones del modelo, utilizaremos visualizaciones gráficas detalladas. Estas gráficas mostrarán la evolución de las recompensas por episodio, las líneas azules representan las puntuaciones obtenidas en cada episodio y una línea roja que reflejará la media móvil de las recompensas de los últimos 100 episodios. Este enfoque no solo facilitará la interpretación visual de los resultados, sino que también permitirá identificar tendencias, patrones de mejora o deterioro en el rendimiento del modelo a lo largo del tiempo.

Además, implementaremos un mecanismo de guardado automático para el modelo. Este sistema almacenará la configuración del modelo cada vez que se alcance una nueva media máxima de recompensas, permitiendo así preservar los estados del modelo que demuestren un rendimiento superior. Este proceso no solo garantiza que no perdamos configuraciones óptimas durante los experimentos, sino que también nos proporciona un punto de referencia fiable para futuras comparaciones y análisis.

A lo largo de este capítulo, no solo describiremos los cambios realizados y los resultados obtenidos, sino que también formularemos hipótesis sobre por qué ciertas modificaciones en los parámetros influyen en el rendimiento del modelo. Este análisis detallado ayudará a entender mejor las dinámicas del aprendizaje por refuerzo y la influencia de cada componente del modelo Rainbow DQN en el proceso de aprendizaje y optimización.

En conclusión, este capítulo no solo busca alcanzar un rendimiento óptimo a través de la afinación precisa del modelo, sino que también pretende aportar a la comprensión teórica y práctica del aprendizaje por refuerzo en entornos controlados, aportando conocimientos que podrían ser aplicados a otros proyectos y escenarios dentro del campo de la inteligencia artificial.

12.2– Capas Modelo

En la búsqueda de optimizar el rendimiento del modelo Rainbow DQN aplicado al entorno de Ms. Pacman, hemos explorado la adición de varias configuraciones de capas convolucionales y lineales. Cada configuración aporta un aumento en la profundidad y complejidad del modelo podría traducirse en una mejora significativa en la capacidad de aprendizaje del agente, justificando así el incremento en el costo computacional. Las siguientes configuraciones fueron evaluadas durante nuestros experimentos:

- **1. Configuración Básica**. Compuesta por 3 capas convolucionales seguidas de 1 capa lineal, además de una capa actor y una capa crítico. Esta configuración sirve como nuestra línea base.

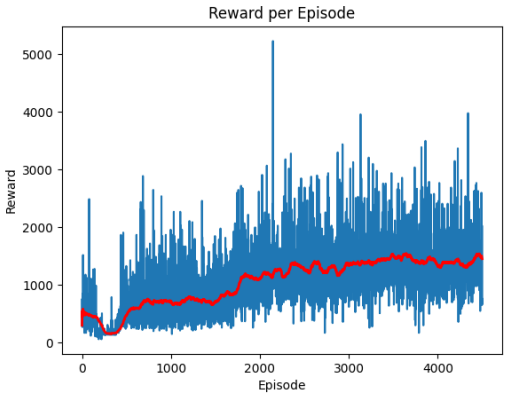

Figura 12.1: 3 capas convolucionales 1 capa linear, 1 capa actor y 1 capa crítico.

- **2. Configuración Intermedia**. Incorpora 4 capas convolucionales y 2 capas lineales, manteniendo la capa actor y la capa crítico. El objetivo es evaluar el impacto de una capa lineal adicional en la capacidad de generalización del modelo.

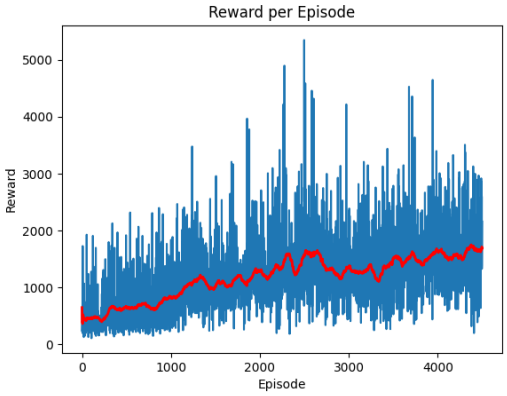

Figura 12.2: 4 capas convolucionales, 2 capas lineares, 1 capa actor y 1 capa crítico

- **3. Configuración Avanzada**. Se caracteriza por tener 4 capas convolucionales y 4 capas lineales, con las correspondientes capas actor y crítico. Esta configuración es

la más compleja y busca maximizar el rendimiento del aprendizaje.

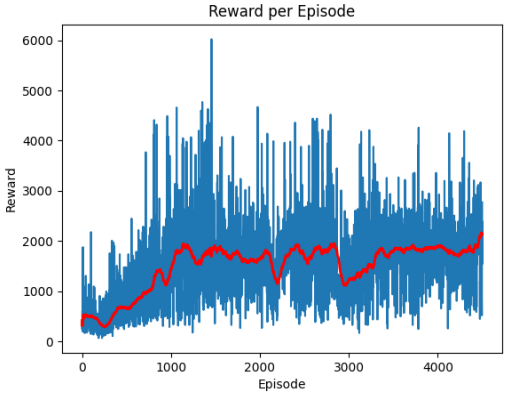

Figura 12.3: 4 capas convolucionales, 4 capas lineares, 1 capa actor y 1 capa crítico

12.2.1. Resultados y Análisis

La evaluación de cada configuración se llevó a cabo observando las métricas de rendimiento representadas en las figuras 12.1, 12.2 y 12.3. Los resultados indican que la configuración avanzada mostró un rendimiento superior, como se refleja en un aumento notable en la línea roja de la media móvil de las recompensas en los últimos 100 episodios.

La configuración avanzada, con un mayor número de capas, ha demostrado ser la más efectiva. Esto sugiere que el modelo se beneficia de una representación más rica y detallada del entorno, lo que permite al agente realizar predicciones más precisas y tomar decisiones más informadas durante el juego. La mejora en el rendimiento, justificada por un aumento en la complejidad, confirma nuestra hipótesis inicial de que una mayor profundidad puede ser ventajosa en este entorno específico.

Con más capas, una red neuronal puede aprender un espectro más amplio de representaciones útiles que podrían no ser capturadas por una red más superficial. Esto permite que el modelo generalice mejor a diferentes situaciones dentro del juego, como esquivar a los fantasmas o planificar rutas óptimas para maximizar la puntuación. La generalización es fundamental en entornos de aprendizaje por refuerzo, donde las situaciones pueden variar significativamente de un episodio a otro.

12.3– N-step learning

El n-step learning es una técnica dentro del aprendizaje por refuerzo que permite al agente evaluar las acciones basándose en una secuencia de `n` pasos en el futuro en lugar de un solo paso. Este enfoque puede ayudar a balancear entre la evaluación inmediata y la evaluación a largo plazo de las acciones, lo que es particularmente útil en entornos complejos como el de Ms. Pacman, donde las consecuencias de las acciones pueden desarrollarse a lo largo de múltiples pasos temporales. Para explorar el impacto de diferentes valores de `n` en el rendimiento del modelo, realizamos experimentos con los siguientes ajustes de n-step learning, hasta ahora habíamos hecho todas las pruebas con n=2:

- **4. n=8**. Evaluación de recompensas acumuladas a lo largo de ocho pasos.

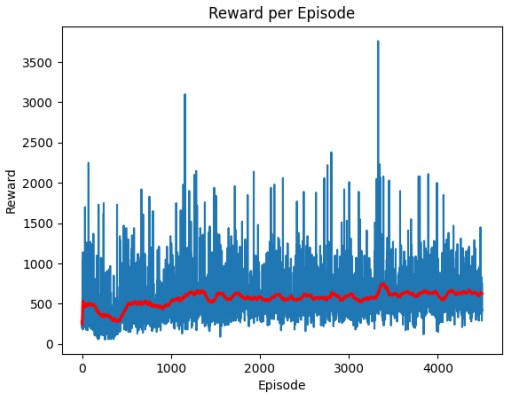

Figura 12.4: n=8

- **5. n=4**. Evaluación de recompensas acumuladas a lo largo de cuatro pasos.

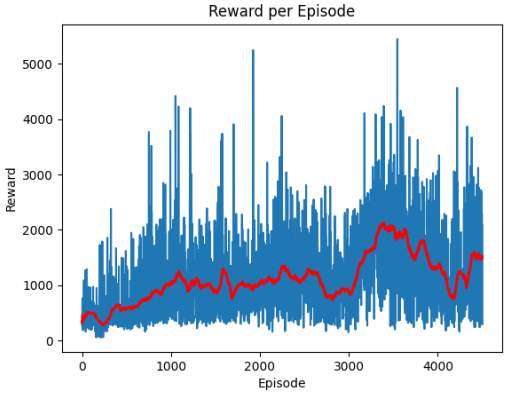

Figura 12.5: n=4

- **6. n=1**. Evaluación de la recompensa inmediata, sin acumulación.

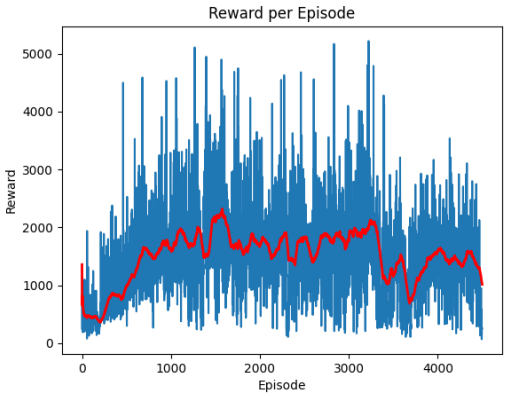

Figura 12.6: n=1

12.3.1. Resultados y Análisis

Los resultados de estos experimentos se visualizan en las figuras 12.4, 12.5, y 12.6. A partir de estos datos, observamos que los ajustes de n=4, n=8 y n=1 no mejoran significativamente el rendimiento del modelo en comparación con nuestra configuración base de n=2. De hecho, se percibe una tendencia donde un valor mayor de n no necesariamente conduce a un mejor rendimiento en este contexto específico. A continuación, se discuten algunas posibles razones y lecciones aprendidas de los experimentos:

Mayor valor de n y su efecto: Un valor más alto de n permite una mejor visión a largo plazo, pero también puede introducir ruido y variabilidad en las estimaciones de recompensa debido a la propagación de errores en las predicciones de recompensas futuras. En el caso de n=8, es posible que la acumulación de error haya contrarrestado los beneficios de una vista más amplia.

Optimización sobre corto plazo: El modelo con n=1 enfoca su aprendizaje en recompensas inmediatas, lo que puede ser menos efectivo en un entorno donde las estrategias a largo plazo son cruciales para el éxito. Esto puede explicar por qué n=1 y valores más altos no superaron el desempeño de n=2.

Equilibrio en n-step learning: La elección de n=2 parece proporcionar un equilibrio adecuado entre la evaluación a corto y largo plazo en este entorno, permitiendo al modelo tomar decisiones más informadas sin sufrir de manera excesiva la propagación del error.

A través de estos experimentos con diferentes configuraciones de n-step learning, hemos identificado que n=2 ofrece el equilibrio óptimo para nuestro modelo en el entorno de Ms. Pacman. Este resultado resalta la importancia de ajustar los parámetros de n-step learning según las características específicas del entorno y los objetivos del aprendizaje.

12.4– Priotized Replay

La técnica de Prioritized Replay Memory juega un papel importante en la eficiencia del aprendizaje en el modelo Rainbow DQN, al permitir que el agente se enfoque más en las transiciones de las que tiene más que aprender. Observamos en la figura 12.7 un descenso en el rendimiento cuando se desactiva esta característica, evidenciando su importancia en el proceso de aprendizaje por refuerzo. Este enfoque prioriza las experiencias basadas en la magnitud de su error de predicción, optimizando el proceso de aprendizaje al enfocarse en las áreas donde el modelo todavía puede mejorar sustancialmente.

- **7. Sin Prioritized Replay Memory**

Como observamos en la figura 12.7, el rendimiento del modelo merma al no aplicarle la priorización en las experiencias, por lo que nos damos cuenta de la influencia de esta mejora de Rainbow DQN aunque se vea afectado el coste computacional al tener que calcular el error, cada vez que se almacena una experiencia.

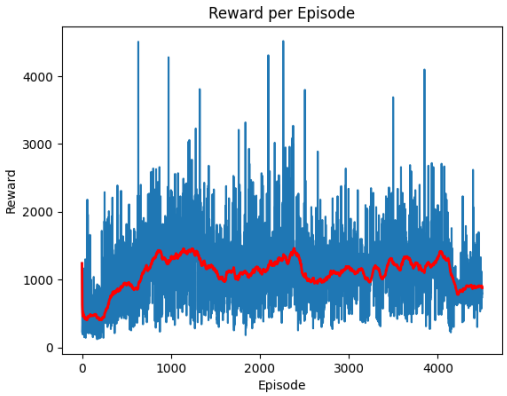

Figura 12.7: Replay Memory (sin PER)

12.4.1. Variación del Tamaño Buffer Replay

Para comprender aún más el impacto de la memoria de replay en el aprendizaje, hemos probado la implementación de Prioritized Replay con diferentes tamaños de buffer. El objetivo es observar cómo la capacidad de almacenamiento afecta la habilidad del modelo para generalizar y aprender de un conjunto diversificado de experiencias acumuladas. Las siguientes capacidades del buffer de replay fueron probadas, con el fin de evaluar su impacto en el rendimiento del modelo:

- **8. Buffer=10.000**
- **9. Buffer=20.000**

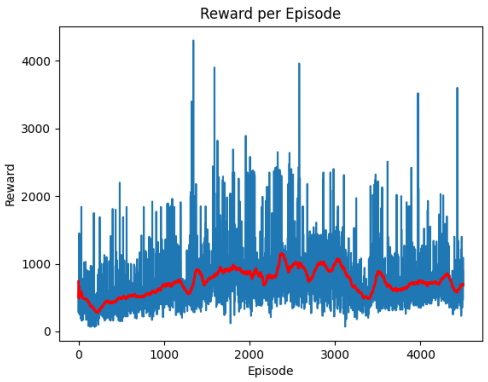

Figura 12.8: Prioritized Experience Replay 10.000

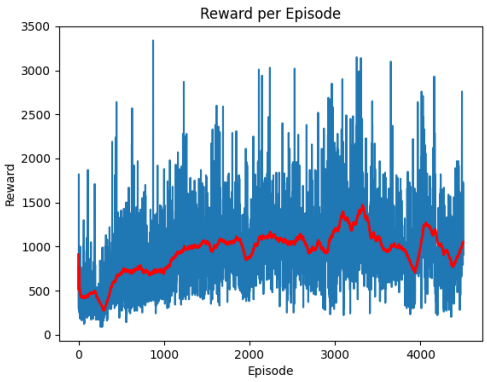

Figura 12.9: Prioritized Experience Replay 20.000

- **10. Buffer=40.000**
- **11. Buffer=80.000**

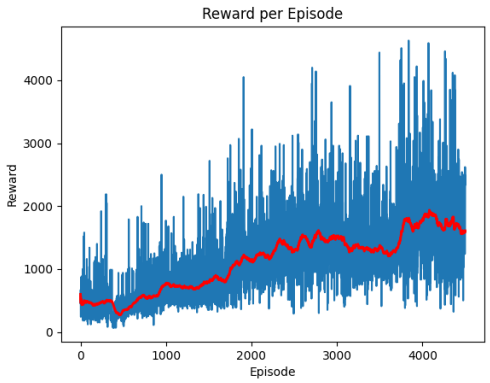

Figura 12.10: Prioritized Experience Replay 40.000

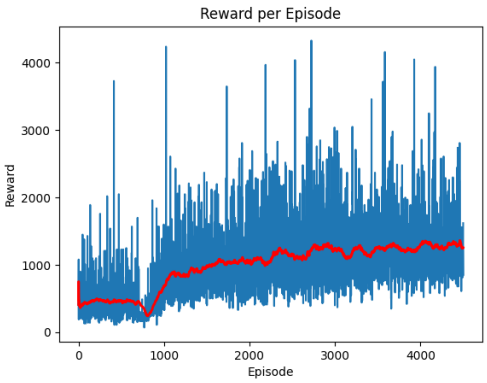

Figura 12.11: Prioritized Experience Replay 80.000

12.4.2. Resultados y Análisis

Cada configuración fue evaluada en términos de la recompensa total por episodio, como se puede observar en las figuras asociadas a cada configuración. Los gráficos muestran cómo varía la recompensa promedio a medida que se incrementa el tamaño del buffer, reflejando las dinámicas de aprendizaje bajo diferentes volúmenes de memoria.

Los resultados indican una tendencia interesante: mientras que los buffers más pequeños limitan la cantidad de experiencias diversas que el modelo puede revisitar y aprender, los buffers más grandes proporcionan una rica base de datos de experiencias, lo que puede ayudar a mejorar la estabilidad y consistencia del aprendizaje. Sin embargo, un buffer demasiado grande también podría diluir la importancia de las experiencias críticas, potencialmente ralentizando el aprendizaje si las experiencias menos útiles se revisan con demasiada frecuencia.

12.5– Estrategia de Recompensas

Este capítulo se enfoca en la evaluación y modificación de la estrategia de recompensas dentro del modelo Rainbow DQN para Ms. Pacman, dependiendo de la estategia que se siga para recompensar al agente y el valor que se le a cada acción, el comportamiento del agente puede cambiar drásticamente. Originalmente, todas las recompensas recibidas del entorno por medio del agente estaban normalizadas usando $\log(\text{reward}, 1000)$ y se asignaba una penalización de -100 si el agente perdía una vida durante el episodio. Se realizaron tres modificaciones principales en el sistema de recompensas, con el objetivo de explorar diferentes comportamientos del agente y su impacto en el aprendizaje:

- **12.Eliminación de la Normalización de Recompensas:** Las recompensas no se normalizan siguiente el sistema original de Ms Pacman pero manteniendo la penalización por pérdida de vida.

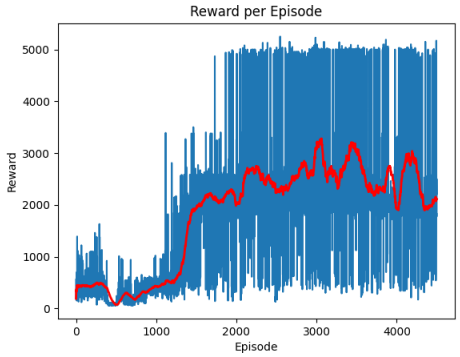

Figura 12.12: Eliminación de la Normalización de Recompensas

- **13. Recompensas Solo por Comer Píldoras:** Las recompensas no se normalizan y se eliminan las recompensas por comer fantasmas o frutas, centrandose únicamente en las píldoras del mapa.

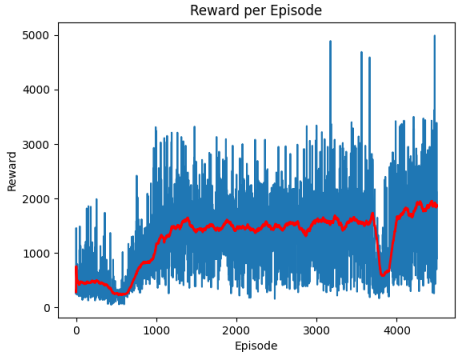

Figura 12.13: Recompensas Solo por Comer Píldoras

- **14. Recompensas Normalizadas con Límite Máximo:** Las recompensas están normalizadas, pero se establece un límite máximo de recompensa de 20 por acción.

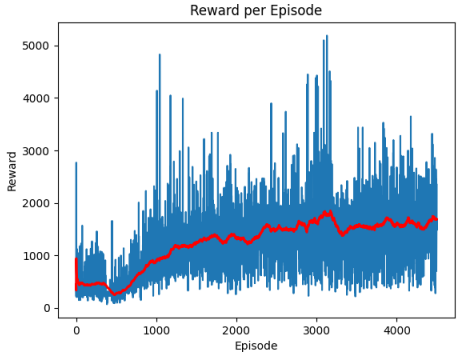

Figura 12.14: Recompensas Normalizadas con Límite Máximo

- **15. Recompensas por Supervivencia Prolongada:** Las recompensas se normalizan y se asigna una penalización negativa por perder vidas a lo largo del episodios. Adicionalmente, si el agente sobrevive más de 150 pasos o steps en el episodio, se le recompensa continuamente mientras siga vivo.

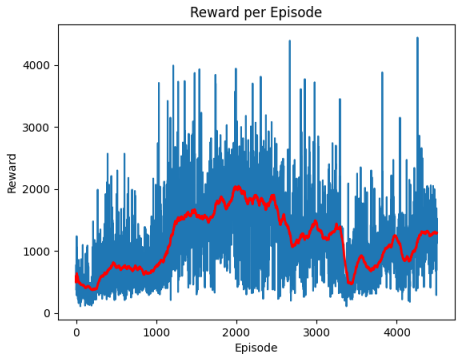

Figura 12.15: Recompensas por Supervivencia Prolongada

- **16. Recompensas normalizadas:** Las recompensas se normalizan mediante un logaritmo en base 1000 y se asigna una penalización negativa por perder vidas a lo largo del episodios.

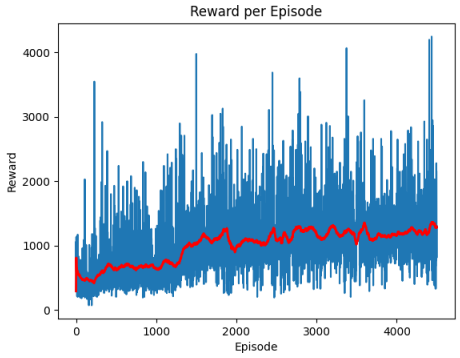

Figura 12.16: Recompensas normalizadas

12.5.1. Resultados y Análisis

Para poder sacar conclusiones de esta experimentación únicamente no ha servido con observar la gráfica, si no que se ha tenido que observar el rendimiento y el tipo de comportamiento del modelo jugando a Ms Pacman una vez terminado el entrenamiento. Se han observado los siguientes comportamientos:

12. Eliminación de la Normalización de Recompensas , La media de las recompensas sube drásticamente, pero el agente tiende a cazar los fantasmas cuando están en azul repetidamente, pero una vez que ha conseguido comer tres o cuatro fantasmas, entra en un bucle de comportamiento sin ningún tipo de sentido y no intenta conseguir más recompensas durante el episodios.

13. Recompensas Solo por Comer Píldoras: El agente mejora notablemente su rendimiento, manteniéndose cerca de comer todas las píldoras necesarias para avanzar de nivel. Este enfoque resulta en un comportamiento más estratégico y de menos riesgo, al no priorizar los `power-ups` para comer a los fantasmas.

14. Recompensas Normalizadas con Límite Máximo: El comportamiento del agente mejora ligeramente en comparación con la configuración anterior, demostrando un equilibrio entre el intento de conseguir puntos comiendo fantasmas e intentar completar el nivel comiendo todas las píldoras del mapa.

15. Recompensas por Supervivencia Prolongada El agente desarrolla un comportamiento inactivo tras alcanzar los 150 pasos en el episodio, dirigiéndose a las esquinas y permaneciendo allí para beneficiarse de las recompensas continuas por supervivencia. Esto resulta en un modelo no óptimo ya que evita interactuar significativamente con el entorno después de cumplir con el requisito de pasos mínimos.

16. Recompensas normalizadas El agente desarrolla un buen rendimiento similar al del experimento 14 (Recompensas Normalizada con Límite Máximo), teniendo un equilibrio entre la búsqueda de completar el nível comiendo píldoras y la máximización de los puntos búscando comer fantasmas en serie.

Cada una de estas modificaciones ofrecen comportamientos variados sobre el agente demostrando cómo las recompensas influencian las estrategias y el aprendizaje del agente. La eliminación de la normalización de recompensas parece incentivar comportamientos extremos, mientras que la limitación y focalización de las recompensas ayudan a guiar al agente hacia comportamientos más deseables y eficientes. La elección de la estrategia de recompensas dependerá de los objetivos que se propongan para el agente, si se pretende maximizar la puntuación a toda costa, si se pretender terminar los niveles o un equilibrio entre ambos

12.6– Noisy Nets

En esta sección probaremos a modificar el valor de std en la clase Noisy Linear que representa la variabilidad de la cantidad de ruido introducida a los pesos de la capa de la red neuronal. Después probaremos a varios el numero de capas con ruido en nuestro modelo para encontrar la configuración con mayor rendimiento. Hasta ahora habiamos hecho todas las pruebas con std=0.5 y con capas ruidosas en todas las de tipo linear.

- **17.Capa final como única capa ruidosa:** Se utilizó std=0.8 (no std=0.5 como antes para compensar el hecho de que solo hay una capa de exploración) pero solo con la última capa lineal añadiendo ruido, para investigar el efecto de concentrar la exploración en una etapa más avanzada del procesamiento.

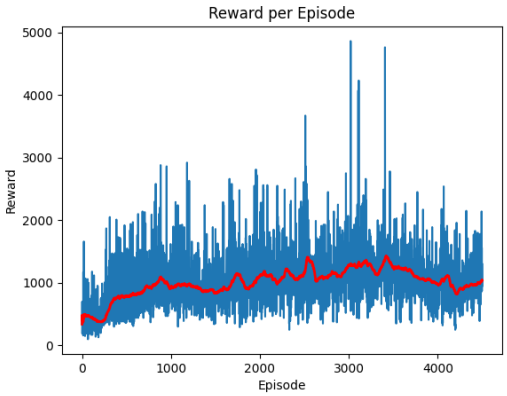

Figura 12.17: Una única capa ruidosa

12.6.1. Análisis del Impacto del Ruido

La técnica utilizada para fomentar la exploración que hemos utilizado es la de introducir ruido en las capas de procesamiento del modelo, el ruido que utilizamos en esta capa lo podemos modificar mediante la variación del para metro std. Esta sección analiza cómo diferentes niveles de ruido, controlados por el parámetro de desviación estándar (std), afectan el rendimiento del modelo en términos de la recompensa acumulada por episodio.

- **18. std=0.3** Un nivel bajo de ruido, que favorece la explotación sobre la exploración.
- **19. std=0.5** Un nivel moderado que busca un equilibrio entre explorar nuevas políticas y explotar las ya conocidas.

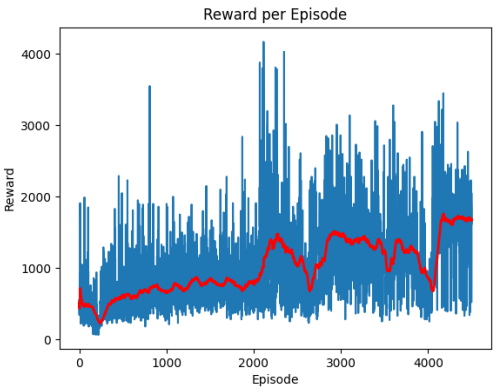

Figura 12.18: std=0.3

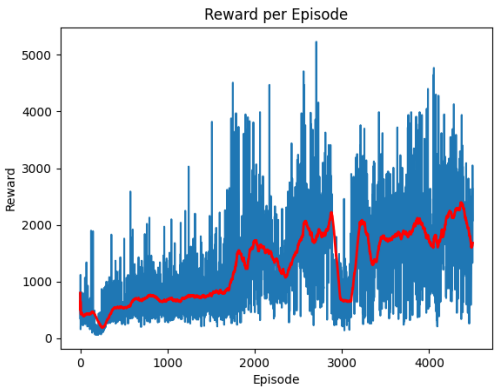

Figura 12.19: std=0.5

- **21. std=0.7** Un nivel elevado de ruido, aumentando significativamente la exploración a costa de posiblemente incrementar la varianza en las recompensas.
- **22. std=0.9** Un nivel muy alto de ruido, que maximiza la exploración pero puede comprometer la estabilidad del aprendizaje.

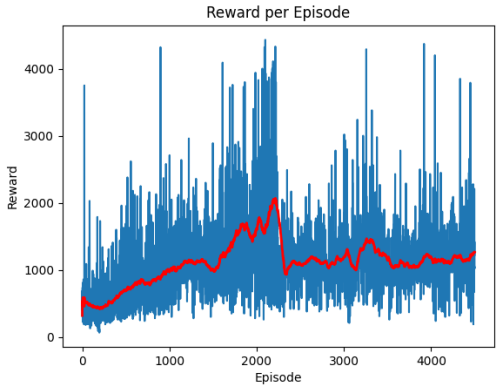

Figura 12.20: std=0.7

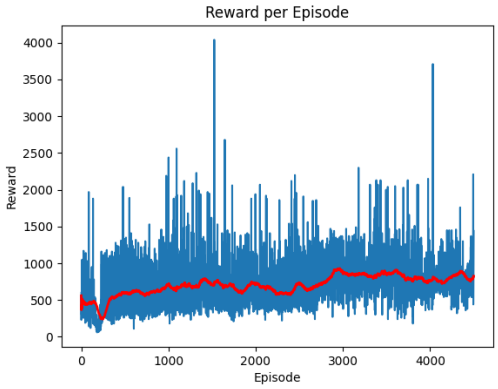

Figura 12.21: std=0.9

12.6.2. Resultados y Análisis

En los experimentos realizados, el análisis de los diferentes niveles de ruido estocástico (std) en modelos de aprendizaje por refuerzo demuestra que el std=0.5 ofrece el mejor rendimiento al equilibrar efectivamente la exploración y explotación. Este nivel medio de ruido facilita la identificación de estrategias efectivas, reflejada en las recompensas más altas y consistentes en comparación con otros niveles de std. Los niveles más bajos y más altos de std resultan en una exploración insuficiente o excesiva, respectivamente, lo que impacta negativamente en la estabilidad y eficacia del aprendizaje del modelo. Dado este hallazgo, se decidió continuar utilizando std=0.5 para futuros experimentos con el fin de optimizar el balance entre exploración y explotación.

12.7– Variación del Learning Rate

En esta sección, exploramos el impacto del hiperparámetro learning rate (tasa de aprendizaje) en el rendimiento del modelo Rainbow DQN en el entorno de Ms. Pacman. La tasa de aprendizaje es un parámetro crítico en el proceso de optimización, ya que determina el tamaño de los pasos que el modelo da al actualizar sus parámetros. Un learning rate demasiado alto puede provocar oscilaciones y divergencia en el entrenamiento, mientras que uno demasiado bajo puede resultar en un aprendizaje extremadamente lento y posible estancamiento en mínimos locales subóptimos.

Hasta ahora, todos los experimentos anteriores se realizaron con una tasa de aprendizaje de lr = 0.0001. Para investigar el efecto de este hiperparámetro, realizamos experimentos adicionales con dos valores alternativos de learning rate: lr = 0.001 y lr = 0.00001.

Los resultados de estos experimentos demostraron un rendimiento deficiente en comparación con la configuración original. A continuación, se detallan los hallazgos específicos para cada valor de tasa de aprendizaje probado:

- **lr = 0.001**. Este valor resultó en un rendimiento inestable, con oscilaciones significativas en las recompensas por episodio. El modelo parecía incapaz de converger adecuadamente, probablemente debido a que los pasos de actualización eran demasiado grandes, causando que el aprendizaje se volviera errático.

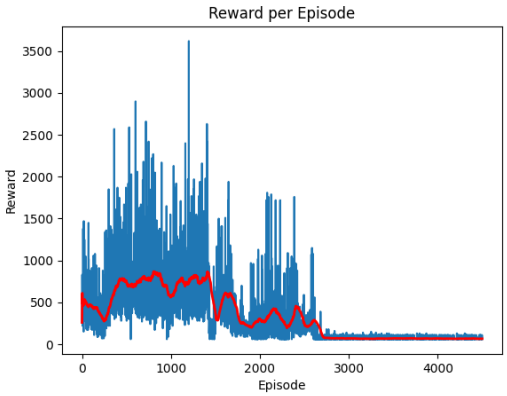

Figura 12.22: Recompensa por episodio con lr = 0.001

- **lr = 0.00001**. En contraste, este valor de learning rate fue demasiado bajo, resultando en un proceso de aprendizaje extremadamente lento. El modelo mostró una mejora mínima en las recompensas acumuladas a lo largo de los episodios, indicando que el pequeño tamaño de los pasos de actualización no permitía un aprendizaje efectivo en un tiempo razonable.

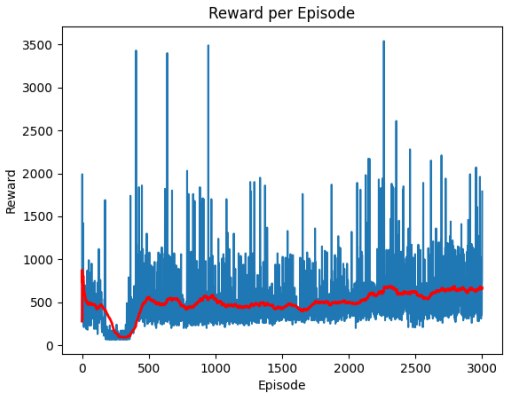

Figura 12.23: Recompensa por episodio con lr = 0.00001

12.7.1. Resultados y Análisis

El análisis de los resultados obtenidos con las diferentes tasas de aprendizaje revela la importancia de seleccionar un valor adecuado para este hiperparámetro. Un learning rate demasiado alto como lr = 0.001 lleva a un comportamiento inestable y divergente, mientras que un valor demasiado bajo como lr = 0.00001 no proporciona la suficiente inercia para un aprendizaje efectivo y rápido.

El valor de lr = 0.0001 se mantiene como el más adecuado para este modelo en el entorno de Ms. Pacman, proporcionando un equilibrio óptimo entre la velocidad de aprendizaje y la estabilidad del proceso de optimización.

12.8– Evaluación de Diferentes Optimizadores

En esta sección, exploramos el impacto de varios optimizadores en el rendimiento del modelo Rainbow DQN en el entorno de Ms. Pacman. Los optimizadores determinan cómo se actualizan los parámetros del modelo durante el entrenamiento. En este experimento, evaluamos cuatro optimizadores populares: Adam, SGD, RMSprop y AdamW. Cada uno de estos optimizadores tiene características únicas que pueden influir en la eficacia y eficiencia del entrenamiento.

12.8.1. Optimizador SGD

El optimizador *Stochastic Gradient Descent* (SGD) es uno de los métodos más simples y ampliamente utilizados para la optimización de redes neuronales. Actualiza los parámetros en la dirección negativa del gradiente de la función de pérdida con respecto a los parámetros del modelo. La fórmula de actualización es:

$$\theta_t = \theta_{t-1} - \alpha g_t$$

donde θ_t son los parámetros del modelo en el paso t, α es el learning rate, y g_t es el gradiente de la función de pérdida con respecto a los parámetros en el paso t.

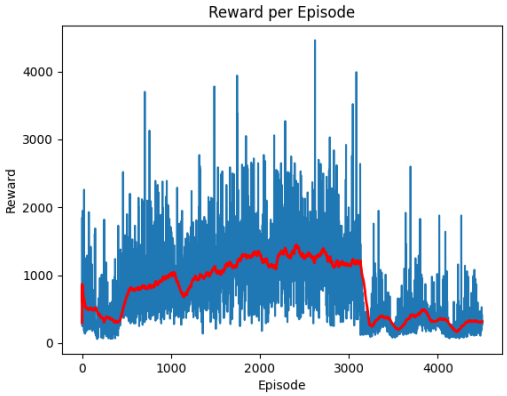

Figura 12.24: Rendimiento del modelo con optimizador SGD

12.8.2. Optimizador RMSprop

RMSprop (*Root Mean Square Propagation*) es un optimizador que ajusta el learning rate de cada parámetro de manera adaptativa, lo que es particularmente útil en problemas con gradientes que pueden cambiar de manera drástica. RMSprop mantiene una media móvil de los cuadrados de los gradientes recientes para normalizar los gradientes, estabilizando el proceso de entrenamiento.

Las actualizaciones de los parámetros en RMSprop se calculan como:

$$v_t = \beta v_{t-1} + (1 - \beta) g_t^2$$

$$\theta_t = \theta_{t-1} - \frac{\alpha g_t}{\sqrt{v_t} + \epsilon}$$

donde v_t es la media móvil de los cuadrados de los gradientes, α es el learning rate, β es el factor de decaimiento y ϵ es un término de suavizado.

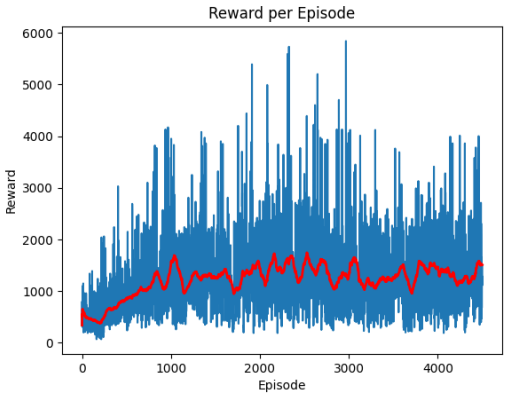

Figura 12.25: Rendimiento del modelo con optimizador RMSprop

12.8.3. Optimizador AdamW

AdamW es una variante del optimizador Adam que introduce una corrección explícita para el decay de los pesos. A diferencia de Adam, que ajusta el decay de los pesos a través del learning rate, AdamW realiza esta operación de manera explícita, lo que puede resultar en una mejor regularización y menor riesgo de sobreajuste.

Las actualizaciones de los parámetros en AdamW se calculan como en Adam, pero con un término adicional para el decay de los pesos:

$$\theta_t = \theta_{t-1} - \alpha \left(\frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} + \lambda \theta_{t-1} \right)$$

donde λ es el factor de decay de los pesos.

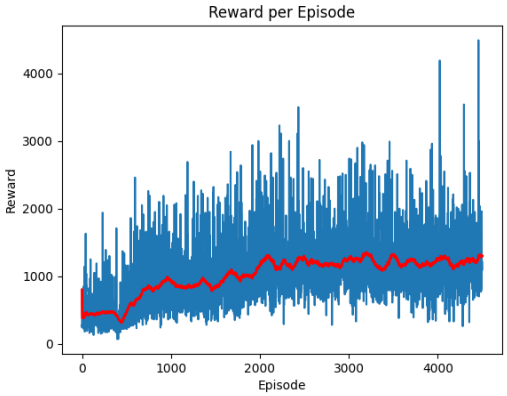

Figura 12.26: Rendimiento del modelo con optimizador AdamW

12.8.4. Resultados y Análisis

Los resultados de los experimentos con los diferentes optimizadores se muestran en las figuras anteriores. A continuación, se resumen los hallazgos clave:

- **SGD:** Como se puede observar en la Figura 12.24, el optimizador SGD mostró un rendimiento aceptable al principio,pero al tener una mayor variabilidad resulto en un modelo no óptimo.
- **RMSprop:** En la Figura 12.25, RMSprop muestra un rendimiento más estable que SGD. La línea roja de la media móvil indica una mejora constante y más uniforme en las recompensas por episodio. Este optimizador ofrece una adaptación eficiente del learning rate, lo que es beneficioso en problemas con gradientes variables.
- **AdamW:** La Figura 12.26 muestra que AdamW proporciona una rápida convergencia y una línea roja de la media móvil más estable y más alta en comparación con SGD y RMSprop. AdamW combina la rápida convergencia de Adam con una mejor regularización de los pesos, resultando en una mayor estabilidad y menor riesgo de sobreajuste.

En conclusión, la elección del optimizador puede tener un impacto significativo en el rendimiento del modelo Rainbow DQN. AdamW demostró ser particularmente efectivo, ofreciendo una buena combinación de rapidez y estabilidad en el entrenamiento. Sin embargo, la selección del optimizador debe ser realizada considerando las características específicas del problema y los recursos disponibles para el entrenamiento.

CAPÍTULO 13

Conclusión

Este trabajo ha estudiado profundamente las mejoras que ofrece el algoritmo Rainbow DQN frente al algoritmo básico de Q-learning en un entorno complejo y dinámico como es el juego Ms. Pac-Man. En los distintos capítulos hemos estado recorriendo una a una la integración de las distintas técnicas avanzadas de aprendizaje por refuerzo, tales como Double DQN, Prioritized Experience Replay, y Dueling Networks, dando en primer lugar el fundamento teórico de estas mejoras y posteriormente una implementación practica gracias al entorno proporcionado por OpenAI llamado Gymnassium, con python como lenguaje de programación y PyTorch como framework de aprendizaje automatico, estos capítulos anteriores permiten no solo entender el algoritmo Rainbow DQN en su conjunto, sino también profundizar en la comprensión teórica y práctica de cada componente.

A lo largo de la experimentación, se ha ido variando los distintos hiperparmétros y estrutura de las mejoras incorporadas, para verificar que valores generan un modelo capaz de generar partidas con un mayor puntuación, y experimento tras experimento se ha ido configurando el modelo hasta acabar en el modelo final.

Finalmente, este Trabajo de Fin de Grado no solo ha servido para validar la efectividad del Rainbow DQN en el entorno específico de Ms. Pac-Man, sino que también ha establecido una base sólida de conocimiento de distintas técnicas que podría ser utilizadas en otros ámbitos de la inteligencia artificial o otros algoritmos de aprendizaje por refuerzo, promoviendo una comprensión más rica de la interacción entre componentes algorítmicos y la dinámica de entornos complejos.

En resumen, los resultados obtenidos promueven un mayor desarrollo y refinamiento del Rainbow DQN y de algoritmos de aprendizaje por refuerzo en general, con el objetivo de alcanzar soluciones cada vez más eficientes y adaptativas para problemas de decisión complejos en el mundo real.

Bibliografía

[1] Z. Wang, T. Schaul, M. Hessel, H. Van Hasselt, M. Lanctot, N. de Freitas *Dueling Network Architectures for Deep Reinforcement Learning* Proceedings of The 33rd International Conference on Machine Learning, 1995–2003, 2016. https://arxiv.org/abs/1511.06581

[2] H. Van Hasselt, A. Guez, D. Silver *Deep Reinforcement Learning with Double Q-learning* Proceedings of the AAAI Conference on Artificial Intelligence, 2094–2100, 2016. https://ojs.aaai.org/index.php/AAAI/article/view/10295

[3] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, Martin Riedmiller *Playing Atari with Deep Reinforcement Learning* NIPS Workshop, 2013. https://arxiv.org/abs/1312.5602

[4] T. Schaul, J. Quan, I. Antonoglou, D. Silver *Prioritized Experience Replay* International Conference on Learning Representations, 2016. https://arxiv.org/abs/1511.05952

[5] V. Mnih, A. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, K. Kavukcuoglu *Asynchronous Methods for Deep Reinforcement Learning* International Conference on Machine Learning, 2016. https://arxiv.org/abs/1602.01783

[6] V. Mnih, A. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, K. Kavukcuoglu *Dueling Network Architectures for Deep Reinforcement Learning* Google DeepMind https://arxiv.org/pdf/1511.06581

[7] M. Bellemare, W. Dabney, R. Munos *A Distributional Perspective on Reinforcement Learning* International Conference on Machine Learning, 2017. https://arxiv.org/pdf/1707.06887

[8] M. Fortunato, M. G. Azar, B. Piot, J. Menick, I. Osband, A. Graves, V. Mnih, R. Munos, D. Hassabis, O. Pietquin, C. Blundell, S. Petersen *Noisy Networks for Exploration* International Conference on Learning Representations, 2018. https://arxiv.org/abs/1706.10295

[9] Tom Schaul, John Quan, Ioannis Antonoglou and David Silver *Prioritized Experience Replay* 2020-04-14 https://arxiv.org/pdf/1511.05952

[10] *IntroRL* https://introrl.readthedocs.io/en/latest/quickstart.html

[11] Dilith Jayasinghe *Actor-Critic Methods* 2023 https://dilithjay.com/blog/actor-critic-methods

[12] Towards Data Science *TD3: Learning to Run with AI* 2019 https://towardsdatascience.com/td3-learning-to-run-with-ai-40dfc512f93

[13] Hado van Hasselt *Double Q-learning* https://proceedings.neurips.cc/paper$_f$*iles/paper*/2010/*file*/091*d*5 *Paper.pdf*

[14] Avandekleut *Deep Q-Networks* 2020 https://avandekleut.github.io/dqn/

[15] Matteo Hessel *Rainbow: Combining Improvements in Deep Reinforcement Learning 2017 https://arxiv.org/pdf/1710.02298*

[16] Richard S. Sutton and Andrew G. Barto Reinforcement Learning 2nd Edition. MIT Press, 2nd edition, 2018.

[17] García-Ferreira, E. Explotación o exploración: el gran dilema 2024 https://www.garcia-ferreira.es/explotacion-o-exploracion-el-gran-dilema/

[18] Sagi Shaier N-step Learning httpshttps://towardsdatascience.com/introduction-to-reinforcement-learning-rl-part-7-n-step-bootstrapping-6c3006a13265

[19] Gymnasium Documentation. Gymnasium Documentation 2024 https://gymnasium.farama.org/index.html

[20] Sanz, M. Introducción al aprendizaje por refuerzo. Parte 4: Double DQN y Dueling DQN 2024 https://markelsanz14.medium.com/introducci

[21] Pugh, D. Prioritized Experience Replay 2020 https://davidrpugh.github.io/stochastic-expatriate-descent/pytorch/deep-reinforcement-learning/deep-q-networks/2020/04/14/prioritized-experience-replay.html

[22] Latinx in AI. Optimizadores https://medium.com/latinxinai/optimizadores-5123992d344c