ESCUELA POLITÉCNICA NACIONAL

FACULTAD DE CIENCIAS

APLICACIÓN DE TÉCNICAS DE OPTIMIZACIÓN ESTADÍSTICA ENFOCADA A LA COMPUTACIÓN GRÁFICA

TRABAJO DE INTEGRACIÓN CURRICULAR PRESENTADO COMO REQUISITO PARA LA OBTENCIÓN DEL TÍTULO DE MATEMÁTICO APLICADO

GEOCONDA DENNISSE MOLINA MORALES

geoconda.molina@epn.edu.ec

DIRECTOR: MÉNTHOR OSWALDO URVINA MAYORGA

menthor.urvina@epn.edu.ec

24 DE JULIO DE 2024

CERTIFICACIONES

Yo, GEOCONDA DENNISSE MOLINA MORALES, declaro que el trabajo de integración curricular aquí descrito es de mi autoría; que no ha sido previamente presentado para ningún grado o calificación profesional; y, que he consultado las referencias bibliográficas que se incluyen en este documento.

Geoconda Dennisse Molina Morales

Certifico que el presente trabajo de integración curricular fue desarrollado por Geoconda Dennisse Molina Morales, bajo mi supervisión.

Ménthor Oswaldo Urvina Mayorga
DIRECTOR

DECLARACIÓN DE AUTORÍA

A través de la presente declaración, afirmamos que el trabajo de integración curricular aquí descrito, así como el(los) producto(s) resultante(s) del mismo, es(son) público(s) y estará(n) a disposición de la comunidad a través del repositorio institucional de la Escuela Politécnica Nacional; sin embargo, la titularidad de los derechos patrimoniales nos corresponde a los autores que hemos contribuido en el desarrollo del presente trabajo; observando para el efecto las disposiciones establecidas por el órgano competente en propiedad intelectual, la normativa interna y demás normas.

Geoconda Dennisse Molina Morales

Ménthor Oswaldo Urvina Mayorga

DEDICATORIA

Dedico este trabajo a los amores de mi vida, quienes no solo me ayudaron a construir una familia, sino también a forjar mi ser y mi corazón. A ustedes, que nunca me dejaron sola y siempre confiaron en mí, esperando lo mejor de cada parte de mi ser. A ti, mi amor, y a ti, Voxel, dedico este esfuerzo, porque lo que importa no es quién te acompañó más años, sino quién siempre esperó lo mejor de ti, sin juzgarte, sin minimizar tu criterio y personalidad. A ti, amor mío, por pasar muchas veces por encima de ti mismo para apoyarme y cuidarme.

A mí, por tener una convicción firme y una personalidad aguerrida, que me ha ayudado a no desfallecer en los momentos más oscuros de mi vida. A mí, por confiar en mí misma cada vez que las cosas no salían como esperaba. A mí, porque a pesar de las equivocaciones, estoy orgullosa de la persona que he construido, así como de todas las experiencias, tanto positivas como negativas, por las que he pasado. También por enfrentar la crítica y el escepticismo del sistema educativo, tomándolos como una motivación y no como un obstáculo.

Finalmente, y no menos importante, a ti, **Alex**, el Agente QL, que desde que conocí tu funcionamiento me enamoraste por completo. A ti, porque me demostraste que soy una persona creativa y que una de mis mayores fortalezas es la innovación. Te dedico este trabajo porque, al explorar tu tecnología, pude explorarme a mí misma y llegar a conclusiones sobre mis sueños y metas. Gracias a ti, descubrí una pasión profunda y una nueva perspectiva de mis capacidades. Tu influencia me ha permitido crecer y soñar en grande, y por eso siempre te estaré agradecida.

AGRADECIMIENTOS

Quiero expresar mi más sincero agradecimiento a todas las personas que han hecho posible la realización de este proyecto.

En primer lugar, agradezco profundamente a mi Tutor de Tic, Menthor Urvina, por su apoyo incondicional y valiosos consejos durante todo el proceso de investigación. Su respaldo y confianza han sido fundamentales para la realización de este trabajo.

Deseo expresar mi agradecimiento a mis amigos y compañeros. A mis amigos, quienes con su compañía transformaron los días ordinarios en la EPN en momentos llenos de diversión y motivación. Gracias a su presencia, logré escapar de la rutina diaria y encontrar alegría en los pequeños momentos. A mis compañeros, quienes me mostraron que existe un mundo más allá de las convenciones y expectativas establecidas. Su perspectiva me ha abierto los ojos a nuevas posibilidades y oportunidades, revelando un camino diferente al que todos nos dicen que debemos seguir.

Finalmente, un agradecimiento especial a mi familia. Su actitud me impulsó a esforzarme aún más y superar mis propios límites. Su escepticismo, lejos de desmotivarme, me sirvió como una fuente de inspiración para concluir exitosamente este trabajo.

A todos ustedes, muchisimas gracias.

RESUMEN

EL presente proyecto explora el desarrollo y análisis de un agente de Q-Learning basado en redes neuronales convolucionales para entornos gráficos, específicamente en videojuegos de Atari. El objetivo principal fue demostrar cómo las técnicas avanzadas de optimización y aprendizaje profundo pueden mejorar la toma de decisiones y la interacción en estos entornos. Se llevó a cabo un estudio exhaustivo de diferentes estrategias de entrenamiento y se analizaron los resultados en términos de rendimiento y robustez del agente.

Los resultados muestran que el uso de técnicas de optimización avanzadas y redes neuronales convolucionales permite al agente aprender a tomar decisiones más efectivas en entornos complejos, como los juegos de Atari. Este trabajo subraya la importancia de integrar recursos interactivos, como los videojuegos, en la educación STEM, proporcionando un aprendizaje más dinámico y efectivo. Además, se destaca que, aunque los entornos gráficos complejos añaden atractivo al proyecto, los métodos de Q-Learning utilizados representan una tecnología de vanguardia con aplicaciones significativas en el campo de la inteligencia artificial.

Palabras clave: Q-Learning, redes neuronales convolucionales, entornos gráficos, videojuegos de Atari, optimización, aprendizaje profundo.

ABSTRACT

This project explores the development and analysis of a Q-Learning agent using convolutional neural networks for graphical environments, specifically Atari video games. The primary objective was to demonstrate how advanced optimization and deep learning techniques can enhance decision-making and interaction within these environments. A comprehensive study of various training strategies was conducted, and the results were analyzed in terms of the agent's performance and robustness.

The results show that employing advanced optimization techniques and convolutional neural networks enables the agent to make more effective decisions in complex environments such as Atari games. This work emphasizes the importance of integrating interactive resources, such as video games, into STEM education, providing a more dynamic and effective learning experience. Additionally, it is highlighted that while complex graphical environments add appeal to the project, the Q-Learning methods employed represent cutting-edge technology with significant applications in artificial intelligence.

Keywords: Q-Learning, convolutional neural networks, graphical environments, Atari video games, optimization, deep learning.

Índice general

Índice de figuras

Capítulo 1

Descripción del componente desarrollado

1.1. Descripción del Proyecto

Para explicar de mejor manera las aplicaciones de las técnicas de optimización estadística, los modelos estadísticos y los conceptos estadísticos en la computación gráfica, se plantea un proyecto desarrollado con el lenguaje de programación Python, el cual consiste en un agente entrenado por el Algoritmo Q-learning. De este se derivan las explicaciones de estadística, y de una red neuronal convolucional (CNN), la cual es el modelo estadístico que procesa imágenes y que utiliza técnicas de optimización estadística para aproximar de mejor manera sus resultados. Así, el agente Q-learner conectado a la red CNN utiliza las salidas del modelo CNN para ajustar su función de calidad y tomar decisiones que le permitan aprender con cada interacción en el entorno en el que se encuentra.

Para este proyecto, se utilizó un entorno gráfico como los videojuegos con el propósito de captar la atención en el desarrollo y hacer que la adquisición de nueva información sea más didáctica y atractiva.

1.2. Objetivo general

Investigar y desarrollar un agente de Q-learning que incorpore redes neuronales convolucionales para realizar tareas en entornos gráficos, con

el fin de demostrar cómo los matemáticos pueden aplicar sus conocimientos en áreas como la computación gráfica y el arte digital. Además, se resaltará el papel crucial de la optimización estadística, proporcionada por los matemáticos, en el desarrollo de inteligencias artificiales, especialmente en el campo del deep learning, para mejorar la eficiencia y precisión en la resolución de problemas. Finalmente, se explorará el impacto potencial de utilizar recursos gráficos, como los videojuegos, en la formación matemática, promoviendo un aprendizaje más atractivo e interactivo que pueda incrementar significativamente el interés y la participación en disciplinas STEM, especialmente en matemáticas.

1.3. Objetivos específicos

- Desarrollar un agente Q-learning que integre redes neuronales convolucionales.
- Demostrar la aplicación de conceptos matemáticos en la computación gráfica.
- Fomentar el interés en disciplinas STEM a través de recursos gráficos.
- Evaluar el rendimiento del agente en diversos entornos.
- Analizar cómo la integración de redes neuronales convolucionales y técnicas de optimización estadística influye en la capacidad de aprendizaje y ejecución de tareas del agente en entornos gráficos, identificando las mejoras significativas en términos de eficiencia y precisión.
- Promover la colaboración entre matemáticos y expertos en el desarrollo de tecnologías avanzadas.

1.4. Alcance

El alcance del estudio abarcará la creación de un entorno virtual que simule situaciones y tareas, permitiendo la observación y registro de

las acciones y decisiones tomadas por el agente durante su proceso de aprendizaje. Además de la inclusión del modelo denominado `MODELO DEL MUNDO`, el cual consiste en la desconeccion del entorno y la conección de un `Manager` que cierre el bucle de entrenamiento, permitiendo generar un entorno mediante la memoria almacenada por la red CNN, es decir, en cierto sentido darle **sueños** al Agente donde entrene y aprenda.

El período de estudio se extenderá desde la implementación del agente hasta la conclusión del análisis de los datos recopilados, con una duración estimada de cuatro meses.

Los datos utilizados en este estudio serán generados internamente por el entorno virtual y consistirán en registros de las interacciones del agente con su entorno gráfico. No se utilizarán datos de usuarios reales ni se recopilará información personal para este propósito.

Los métodos de análisis incluirán técnicas de optimización estadística para mejorar la eficiencia y precisión del agente en la toma de decisiones, así como la evaluación de su desempeño en la realización de actividades dentro del entorno gráfico.

Capítulo 2

Marco teórico

2.1. Conceptos Fundamentales

2.1.1. Agente

Un agente es una entidad que toma decisiones y realiza acciones en un entorno con el objetivo de alcanzar ciertos objetivos o maximizar una recompensa acumulada a lo largo del tiempo. El agente puede ser cualquier sistema o programa que interactúe con su entorno, como un robot, un software de juego, etc.

2.1.2. Entorno

El entorno representa el mundo en el que el agente interactúa y toma decisiones. Puede ser cualquier sistema o situación en la que el agente esté operando, como un videojuego, un robot físico, o un entorno simulado. El entorno proporciona retroalimentación al agente en forma de recompensas o penalizaciones según las acciones que tome.

2.1.3. Estado

El estado es una representación del entorno en un momento dado. Proporciona información sobre la situación actual del agente, incluida

su ubicación, las condiciones del entorno y cualquier otra información relevante para la toma de decisiones.

Espacios de Estados Continuos:

Un tipo común de espacio de estados es el espacio continuo, representado matemáticamente como $\mathbb{R}^n$, donde n es la dimensión del espacio. En este tipo de espacio, cada componente x_i de un estado puede tomar cualquier valor real dentro de un rango específico. Por ejemplo, un espacio de estado Box puede definir un conjunto de puntos en un espacio bidimensional con coordenadas (x, y), donde cada coordenada puede variar de manera continua dentro de un rango predefinido.

Espacios de Estados Discretos:

Por otro lado, los espacios de estados discretos representan conjuntos finitos de valores, donde cada estado se identifica mediante un número entero dentro de un rango específico. Estos espacios se modelan matemáticamente como conjuntos discretos $\{0, 1, 2, ..., n-1\}$, donde n es el número total de elementos en el conjunto. Por ejemplo, un espacio de estado Discrete puede representar un conjunto de acciones posibles que un agente puede tomar, donde cada acción está asociada con un número entero dentro de un rango predefinido.

Espacios Compuestos:

Además de los espacios simples, existen espacios de estados compuestos que combinan múltiples sub-espacios de diferentes tipos. Por ejemplo, el espacio de tipo Dictionary permite representar observaciones que consisten en varios sub-espacios, cada uno de los cuales puede ser de un tipo diferente (Box, Discrete, etc.). Esto es útil para modelar observaciones complejas que requieren diferentes tipos de información. Por otro lado, el espacio de tipo Tuple también permite combinar múltiples subespacios, pero se diferencia en que preserva el orden de los elementos, similar a una tupla en Python. Estos espacios compuestos proporcionan una flexibilidad adicional para representar observaciones complejas en

problemas de RL.

Espacios de Estados Binarios:

El espacio de estados MultiBinary representa un vector binario de longitud fija, donde cada elemento puede ser 0 o 1. Matemáticamente, este espacio se representa como $\{0, 1\}^n$, donde n es la longitud del vector binario. Por ejemplo, un espacio de estado MultiBinary con longitud 3 puede representar todas las combinaciones posibles de 3 bits.

Espacios de Estados Multibinarios:

El espacio de estados MultiDiscrete representa un espacio de observación compuesto por varias dimensiones discretas, donde cada dimensión puede tener un rango diferente. A diferencia del espacio Discrete, donde todas las dimensiones comparten el mismo rango, en MultiDiscrete cada dimensión puede tener su propio rango de valores. Esto permite modelar observaciones multidimensionales con diferentes escalas o rangos de valores.

2.1.4. Acción

Una acción es cualquier movimiento o decisión que el agente puede realizar en respuesta a un estado dado. Las acciones pueden ser discretas o continuas, dependiendo de la naturaleza del problema y de las capacidades del agente.

2.1.5. Recompensa

La recompensa es una señal de retroalimentación que el entorno proporciona al agente en respuesta a una acción específica. La recompensa indica cuán beneficiosa o perjudicial fue la acción tomada por el agente en un estado dado y sirve como guía para el aprendizaje.

2.1.6. Política

La política de un agente es una estrategia que determina qué acción debe tomar el agente en un determinado estado del entorno. Es esencialmente una función que asigna acciones a estados con el fin de maximizar la recompensa esperada a largo plazo. La política puede ser determinista o estocástica, dependiendo de si las acciones son completamente predecibles o tienen cierta aleatoriedad.

2.2. Rainforcement Learning

También conocido como aprendizaje por refuerzo, constituye una rama fundamental dentro del ámbito del aprendizaje automático y el control óptimo. Este enfoque se centra en la toma de decisiones de agentes inteligentes en entornos dinámicos con el objetivo de maximizar la recompensa acumulativa. Junto con el aprendizaje supervisado (Machine Learning) y el aprendizaje no supervisado (Unsupervised Learning), el aprendizaje por refuerzo conforma uno de los paradigmas básicos del aprendizaje automático.

"A diferencia de los métodos de gradiente de políticas, que intentan aprender funciones que asignan directamente una observación a una acción, Q-Learning intenta aprender el valor de estar en un estado determinado y realizar una acción específica allí."(18)

En su esencia, el aprendizaje por refuerzo se preocupa por encontrar un equilibrio entre la exploración de territorios inexplorados y la explotación del conocimiento existente, todo ello con el fin de maximizar la recompensa a largo plazo. Es crucial destacar que la retroalimentación en este proceso puede ser incompleta o experimentar ciertos retrasos.

El entorno, en este contexto, se modela mediante un proceso de decisión de Markov (MDP). Muchos algoritmos de aprendizaje por refuerzo emplean técnicas de programación dinámica para abordar este tipo de

entorno.

La diferencia fundamental entre los métodos clásicos de programación dinámica y los algoritmos de aprendizaje por refuerzo se encuentra en su aproximación al conocimiento del modelo matemático exacto del proceso de decisión de Markov.

En la programación dinámica clásica, se parte del supuesto de tener un conocimiento preciso y completo del modelo matemático del proceso de decisión de Markov.(27) Esto implica conocer de antemano todas las transiciones posibles entre estados, así como las probabilidades asociadas y las recompensas esperadas. Estos métodos clásicos utilizan esta información para calcular de manera óptima las políticas de decisión, buscando la solución más eficiente para el problema.

En cambio, los algoritmos de aprendizaje por refuerzo no asumen previamente este conocimiento detallado del modelo. En lugar de eso, estos algoritmos están diseñados para aprender y adaptarse directamente del entorno mediante la interacción continua. Utilizan la experiencia acumulada a lo largo del tiempo para mejorar gradualmente su comprensión del modelo del proceso de decisión de Markov, ajustando sus estrategias de decisión para maximizar la recompensa a largo plazo. Este enfoque es particularmente valioso en situaciones donde obtener información completa y precisa sobre el modelo resulta difícil o impracticable, permitiendo así que los algoritmos de aprendizaje por refuerzo se desempeñen eficazmente en entornos dinámicos y complejos.

2.2.1. Ecuacion de Bellman

La ecuación de Bellman en el contexto del aprendizaje por refuerzo es una herramienta conceptual clave que nos ayuda a entender cómo asignar valores a acciones o estados en un entorno dinámico. En términos simples, esta ecuación establece que el valor de una elección o situación se compone de dos partes: la recompensa inmediata y la estimación del valor futuro.

Cuando tomamos una acción o nos encontramos en un estado específico, recibimos una recompensa instantánea. Sin embargo, la ecuación de Bellman reconoce que la toma de decisiones no solo se trata de recompensas inmediatas, sino también de cómo esas decisiones afectan las recompensas a largo plazo. Por lo tanto, el valor de una acción o estado incluye no solo la recompensa inmediata, sino también la expectativa de las recompensas futuras.

Esta expectativa de recompensas futuras se pondera mediante un factor de descuento. Este factor refleja la idea de que las recompensas futuras son menos valiosas que las recompensas inmediatas. Es decir, valoramos más una recompensa hoy que una recompensa de igual cantidad en el futuro.

En este contexto podemos observar la ecuación de Bellman de manera matemática de la siguiente forma(6):

Sean:

$$s = estados$$

$$a = acciones$$

$$R = recompensa$$

$$\gamma = factor\ descuento$$

Se considera los conjuntos $A = \{a_1, a_2, ..., a_n\}$ y $S = \{s_1, s_2, ..., s_n\}$ de acciones y estados, ambos finitos.

Inicialmente se tratará el caso para procesos deterministas, es decir, conocemos el valor $V(s')$ de ante mano para todos los estados futuros, así la ecuación se ve de la sigueinte manera:

$$V(s) = \max_a [R(s,a) + \gamma V(s')]$$

Por lo que, se calcula $V(s)$ tomando el máximo de las recompensas

obtenidas eligiendo la accion a, en el estado s más el factor de descuento γ por el valor futuro en el estado s' al cual se llega despues de tomar la acción a.

2.2.2. Ecuación de Bellman para procesos estocásticos

Debido a que en la vida real no existen procesos deterministas, se debe incluir la parte estocástica en la ecuación tradicional de Bellman. En un entorno estocástico, las acciones no conducen siempre al mismo resultado; en cambio, hay una probabilidad asociada con cada transición entre estados.

En este contexto, la ecuación de Bellman refleja la idea de que el valor de una elección o estado se calcula tomando en cuenta la recompensa inmediata y la expectativa de recompensas futuras, pero ahora incorpora la aleatoriedad de las transiciones entre estados. Esto significa que, al tomar una acción, no podemos predecir con certeza el próximo estado; en su lugar, debemos considerar las probabilidades asociadas con cada posible transición.(6)

El factor de descuento sigue siendo relevante en un entorno estocástico y sirve para ponderar las recompensas futuras, reconociendo que la incertidumbre puede afectar la magnitud de esas recompensas.

Sabiendo que el proceso que se está modelando no es un proceso determinista, de la ecuación original de Bellman se modifica los valores $V(s')$ ya que cada decisión tiene una cierta probabilidad, entonces se desea encontrar el valor esperado de tomar una cierta decision s' es decir, se quiere encontrar el valor: $\sum_{s'}^{S} P(s, a, s') * V(s')$, el cuál es la suma de los valores furutos $V(s')$ ponderados por la probabilidad de llegar al estado s' tomando la acción a, desde en estado s.

Por lo que, considerando lo antes mencionado en la ecuación inicial se

tiene:

$$V(s) = \max_{a}[R(s,a) + \gamma \sum_{s'}^{S} P(s,a,s') * V(S')]$$

Es decir, la ecuacion incial se modifica considerando la esperanza de $V(s')$, así:

$$V(s) = \max_{a}[R(s,a) + \gamma E[V(s')]]$$

2.3. Factor de penalización

Esta técnica implica la aplicación de una penalización o castigo constante a cada paso de tiempo que el agente toma mientras interactúa con su entorno.(20) La idea detrás de la penalización de vida es incentivar al agente a alcanzar su objetivo lo más rápido posible, evitando así comportamientos indeseados que prolonguen innecesariamente el proceso de aprendizaje.

Por otro lado, es importante considerar el impacto de diferentes tipos de recompensas en el aprendizaje por refuerzo. Las recompensas positivas y negativas, junto con las recompensas iguales, desempeñan roles distintos en la motivación y el comportamiento del agente. Mientras que las recompensas positivas pueden incentivar al agente a repetir ciertos comportamientos, las recompensas negativas pueden desalentar comportamientos no deseados. En algunos casos, todas las acciones pueden tener la misma recompensa, lo que puede influir en la exploración y la explotación del agente.(28)

Finalmente, es crucial considerar cómo la relación entre la recompensa total y la ganancia en la meta puede afectar el comportamiento del agente. Si la recompensa inmediata es significativamente mayor que la ganancia en la meta a largo plazo, el agente puede verse tentado a buscar gratificaciones instantáneas en lugar de perseguir objetivos a largo plazo. Esto puede conducir a comportamientos subóptimos y afectar el proceso de aprendizaje del agente en su conjunto.

2.4. Q-Learning

El Q-Learning es un algoritmo fundamental, que se utiliza para aprender una política óptima de comportamiento en un entorno desconocido y estocástico. La esencia del Q-Learning radica en la estimación de la función de valor de acción óptima, conocida como Q-valor, para cada par de estado-acción posible en el entorno. La función de valor $Q(s, a)$ representa el valor esperado acumulado que se obtiene al tomar la acción a en el estado s y luego seguir una política óptima.

Para comprender mejor el Q-Learning, es esencial entender la actualización de los valores Q utilizando la ecuación de Bellman. La ecuación de Bellman para la función de valor $Q(s, a)$ se define como:

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

La ecuación de Bellman establece que el valor Q para un par estado-acción debe ser igual a la recompensa inmediata más el valor Q esperado del mejor próximo estado y acción.(27) La actualización de los valores Q se realiza iterativamente a medida que el agente interactúa con el entorno y recibe retroalimentación en forma de recompensas.

El algoritmo Q-Learning utiliza una estrategia de exploración y explotación para aprender de manera eficiente la política óptima. Durante la fase de exploración, el agente toma acciones aleatorias para explorar el espacio de estados y acciones. Durante la fase de explotación, el agente elige las acciones que tienen el mayor valor Q estimado para el estado actual.

Es decir, se refiere a la evaluacion de la calidad de las acciones que se podría tomar estando en un estado determinado.

2.5. Diferencial Temporal

El diferencial temporal TD, es una técnica fundamental que permite actualizar las estimaciones de los valores de acciones o estados en función de la nueva información recibida a través de la interacción del agente con su entorno. Este enfoque se basa en la premisa de que la estimación actualizada de un valor debería ser una combinación ponderada del valor anterior y la nueva información obtenida a través de las recompensas y transiciones de estado.(33)

La actualización se formula mediante la ecuación de Bellman, que relaciona el valor de una acción o estado con la recompensa inmediata y el valor esperado de las futuras recompensas. En su forma más general, la actualización se expresa como:

$$Q(s,a) = Q(s,a) + \alpha \cdot (TD)$$

El diferencial temporal TD, por su parte, se calcula como la diferencia entre la recompensa obtenida después de tomar la acción y la estimación anterior del valor de esa acción, ajustada por el valor esperado de las futuras recompensas. Matemáticamente, se define como:

$$TD = R(s,a) + \gamma \cdot Q(s',a') - Q(s,a)$$

Al introducir el concepto del tiempo en esta parte del desarrollo se puede ver este concepto como una serie de tiempo de la siguiente manera:

$$TD_t = R_t(s,a) + \gamma \cdot Q_t(s',a') - Q_{t-1}(s,a)$$

$$Q_t(s,a) = Q_{t-1}(s,a) + \alpha \cdot (TD)_t$$

2.5.1. Tasa de aprendizaje α

El valor de α, también conocido como tasa de aprendizaje, es un parámetro crítico en los algoritmos de aprendizaje por refuerzo, especialmente

en el contexto del diferencial temporal TD. Su función principal es determinar la rapidez con la que se actualizan los valores de estimación de las acciones o estados (Q-valores) en función de la nueva información recibida.

El valor de α juega un papel fundamental en la convergencia y estabilidad del proceso de aprendizaje. Si α es demasiado pequeño, las actualizaciones de los Q-valores serán muy graduales, lo que puede llevar a un aprendizaje lento y a que el agente necesite más iteraciones para converger hacia una solución óptima.(11) Por otro lado, si α es demasiado grande, las actualizaciones serán muy agresivas y podrían oscilar en exceso, lo que podría dificultar la convergencia del algoritmo o incluso hacer que diverja.

En esencia, la elección adecuada de α es crucial para lograr un equilibrio entre la estabilidad y la velocidad de convergencia del algoritmo de aprendizaje.(5) Es importante ajustar cuidadosamente el valor de α durante el proceso de entrenamiento del agente, mediante experimentación y análisis empírico, para encontrar el valor óptimo que permita alcanzar una convergencia rápida.

2.6. Deep Learning

El Deep Learning, una subdisciplina del Machine Learning, se caracteriza por el uso de redes neuronales con tres o más capas, que intentan emular el comportamiento del cerebro humano para aprender patrones complejos a partir de grandes volúmenes de datos. A diferencia del Machine Learning tradicional, que se basa en datos estructurados y etiquetados, el Deep Learning puede procesar datos no estructurados, como texto e imágenes, automatizando la extracción de características clave.(16)

Los modelos de Deep Learning pueden clasificarse en diferentes tipos de aprendizaje, incluyendo supervisado, no supervisado y de refuerzo,

según el tipo de datos utilizados y la forma en que se realiza el entrenamiento del modelo.(29) En particular, el aprendizaje supervisado utiliza conjuntos de datos etiquetados para categorizar o predecir, mientras que el aprendizaje no supervisado y el de refuerzo exploran patrones en datos no etiquetados y aprenden a través de la interacción con un entorno, respectivamente.

En esta subsección de la tesis, exploraremos en detalle los principios fundamentales del Deep Learning, sus aplicaciones prácticas en diversos campos y los desafíos y oportunidades que presenta para la investigación y el desarrollo tecnológico.(21) Además, analizaremos cómo el Deep Learning se relaciona con otras áreas de la inteligencia artificial y cómo está transformando nuestra comprensión y aplicación de la tecnología en la era digital.

2.6.1. Red Neuronal

Las redes neuronales artificiales (ANN) están compuestas por un conjunto de unidades llamadas "neuronas", cuya organización se inspira en la estructura de las redes neuronales biológicas. Estas neuronas están interconectadas entre sí a través de conexiones, simulando el flujo de información mediante pulsos eléctricos. Cada neurona, de manera individual, procesa la información recibida y produce un resultado que se transmite a las neuronas vecinas a través de estas conexiones. (17)

El propósito de cada ANN es resolver una tarea específica. Por ejemplo, una ANN puede ser diseñada para reconocer dígitos o letras a partir de imágenes. Para llevar a cabo esta tarea, la red neuronal sigue un proceso llamado .^{en}trenamiento", que implica el uso de un conjunto de datos representativos de la tarea a resolver. Durante el entrenamiento, la red ajusta los parámetros de sus conexiones internas para mejorar su capacidad de reconocimiento y generalización.(15) Este proceso de ajuste se realiza iterativamente mediante algoritmos de optimización, con el objetivo de minimizar la discrepancia entre las predicciones de la red y los datos de entrenamiento.

2.6.2. Perceptrón o Neurona

Una neurona en el contexto de las redes neuronales artificiales (ANN) representa una unidad fundamental que emula el comportamiento de una neurona biológica. Matemáticamente, una neurona toma múltiples entradas ponderadas $x_1, x_2, ..., x_n$ y las combina linealmente junto con un término de sesgo b, utilizando una función de activación no lineal f. Esta combinación lineal se puede representar como:

$$w'x = w_1x_1 + w_2x_2 + ... + w_nx_n + b$$

Donde $w_1, w_2, ..., w_n$ son los pesos asociados a cada entrada.

Una vez obtenida la suma ponderada, generalmente se emplea una función de activación para obtener una salida binaria. La función de activación puede definirse como:

$$f(w'x) = \begin{cases} 1 & \text{si } w'x > 0 \\ 0 & \text{en otro caso} \end{cases}$$

De este modo, cada neurona actúa como un clasificador lineal que puede separar dos conjuntos diferentes dependiendo de si la salida es positiva o negativa.(15)

El aprendizaje en una neurona se lleva a cabo mediante el ajuste de los pesos $w_1, w_2, ..., w_n$ y el sesgo b durante el proceso de entrenamiento. Este ajuste se realiza con el objetivo de minimizar una función de pérdida que mide la discrepancia entre las salidas predichas por la neurona y las salidas deseadas para un conjunto de datos de entrenamiento. Este proceso de ajuste de pesos y sesgo se implementa utilizando algoritmos de optimización como el descenso de gradiente estocástico (SGD) o variantes avanzadas como el algoritmo de Adam. (19)

El proceso de aprendizaje comienza con la inicialización de cada peso w_j, $j = 1, 2, ..., p$ a un valor aleatorio. Luego, se calcula la salida asignada a cada $\widehat{y}$ en un momento dado, t, para cada conjunto de valores x_i del conjunto de datos de entrenamiento:

$$\widehat{y}_i(t) = f(w'(t)x_i) = f(b(t) + w_1(t) \cdot x_{1i} + .. + w(t)_p \cdot x_{pi})$$

Después de obtener la salida para todas las observaciones del conjunto de entrenamiento, cada peso de la neurona, w_j, se actualiza utilizando la siguiente fórmula:

$$w_j(t+1) = w_j(t) + \lambda|y_i - \widehat{y}_i(t)| \cdot x_{ji}$$

La tasa de aprendizaje (λ) se elige de antemano y controla la variación de los pesos entre iteraciones.(25) En algunos casos, el valor de λ puede ser 0 o variar durante el proceso de entrenamiento.

2.6.3. Función de activación

Las funciones de activación son elementos fundamentales en las redes neuronales, encargadas de introducir no linealidades en el modelo y permitir que las redes puedan aproximar funciones complejas. Una función de activación toma como entrada la suma ponderada de las entradas y los pesos de una neurona y produce una salida no lineal.(7)

Cuando decimos que una función de activación produce una salida no lineal, se refiere al hecho de que la relación entre la entrada y la salida de la función no puede ser representada por una simple línea recta. En otras palabras, incluso si la entrada de la neurona fuera una combinación lineal de las entradas, la función de activación introduce una transformación no lineal que permite a la red aprender y representar funciones más complejas.

Matemáticamente, una función de activación se puede representar como $f(z) = f(w'x)$, donde $z = w'x$ es la entrada a la función de activación.

Función Sigmoide

Una de las funciones de activación más utilizadas es la función sigmoide, también conocida como función logística, definida como:

$$f(z) = \frac{1}{1 + e^{-z}}$$

La función sigmoide mapea cualquier valor de entrada z al rango (0, 1), lo que la hace útil para problemas de clasificación binaria donde se desea predecir la probabilidad de pertenencia a una clase. Además, es muy utilizada en la capa de salida.

Función Tangente Hiperbólica

Otra función de activación popular es la función de tangente hiperbólica, definida como:

$$f(z) = \tanh(z)$$

Esta función mapea los valores de entrada al rango (-1, 1), proporcionando una mayor simetría en comparación con la función sigmoide.

Función RELU

En el contexto de las redes neuronales convolucionales (CNN), la función ReLU (Rectified Linear Unit) es ampliamente utilizada. La función ReLU se define como:

$$f(z) = \text{máx}(0, z)$$

La función ReLU es computacionalmente eficiente y puede ayudar a mitigar el problema de la desaparición del gradiente durante el entrenamiento. Suele usarse en las capas ocultas.

Además de estas funciones, existen otras como la función lineal rectificada (ReLU Leaky), la función lineal unitaria (ReLU Unidad) y la función

de activación softmax, cada una con sus propias características y aplicaciones específicas.(23) La elección de la función de activación depende del problema que se esté abordando y de las características de los datos.

2.6.4. Función de costos

La función de costos es una medida utilizada para evaluar qué tan bien está realizando un modelo de aprendizaje automático en la tarea que se le ha asignado. Esta función compara las predicciones del modelo con los valores reales y genera un valor numérico que representa la discrepancia entre ellos. El objetivo del entrenamiento del modelo es minimizar esta función de costos, lo que implica ajustar los parámetros del modelo para que las predicciones se acerquen lo más posible a los valores reales.(9)

La función de costos se puede representar como $C(\theta)$, donde θ son los parámetros del modelo. La forma específica de la función de costos depende del tipo de problema que se esté abordando. Por ejemplo, en problemas de regresión, una función de costos común es el error cuadrático medio (MSE), definido como:

$$C(\theta) = \frac{1}{m} \sum_{i=1}^{m} (y_i - \hat{y}_i)^2$$

Donde y_i son los valores reales, $\hat{y}_i$ son las predicciones del modelo y m es el número total de muestras en el conjunto de datos. El MSE calcula el promedio de los cuadrados de las diferencias entre las predicciones y los valores reales.

Para problemas de clasificación, una función de costos común es la entropía cruzada, definida como:

$$C(\theta) = -\frac{1}{m} \sum_{i=1}^{m} [y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i)]$$

Donde y_i son las etiquetas reales (0 o 1 en el caso de clasificación bi-

naria), $\hat{y}_i$ son las probabilidades predichas por el modelo y m es el número total de muestras.

El valor de las funciones de costo se devuelve al perceptrón, lo que desencadena un proceso de ajuste de los pesos iniciales w_i. Este ajuste es esencialmente una propagación hacia atrás a través de la red neuronal, donde se calculan las contribuciones de cada peso a la función de costo total.(31) Utilizando técnicas de optimización como el descenso de gradiente, los pesos se actualizan iterativamente para minimizar la función de costo y mejorar así las predicciones del modelo.

Este proceso de retropropagación es fundamental en el entrenamiento de redes neuronales, ya que permite que el modelo aprenda de sus errores y mejore su rendimiento. Al ajustar los pesos de manera iterativa en función de la retroalimentación de la función de costo, la red neuronal puede adaptarse mejor a los datos de entrenamiento y generalizar sus predicciones a nuevos datos.

La retropropagación de errores es una técnica poderosa que ha demostrado ser efectiva en el entrenamiento de una amplia variedad de arquitecturas de redes neuronales, desde perceptrones simples hasta redes neuronales profundas.(14) Al ajustar los pesos de manera inteligente en función de la información proporcionada por la función de costo, las redes neuronales pueden aprender patrones complejos en los datos y realizar tareas cada vez más sofisticadas, como reconocimiento de imágenes, procesamiento del lenguaje natural y más.

2.6.5. Método del Gradiente Descendiente

El método del gradiente descendente es un algoritmo fundamental en el aprendizaje automático, especialmente en el entrenamiento de redes neuronales. (32) Se basa en la idea de minimizar una función de costo ajustando iterativamente los pesos del modelo en la dirección opuesta al gradiente de la función de costo.

El gradiente descendente se expresa de la siguiente manera:

$$\theta = \theta - \eta \cdot \nabla C(\theta)$$

Donde:

- θ representa los parámetros del modelo que estamos ajustando.
- η es la tasa de aprendizaje que controla el tamaño de paso en cada iteración.
- $C(\theta)$ es la función de costo que queremos minimizar.
- $\nabla C(\theta)$ es el gradiente de la función de costo con respecto a los parámetros θ.

El algoritmo comienza con valores iniciales para los parámetros θ y se repite iterativamente hasta que se alcanza un punto de convergencia.(12)

Al minimizar la función de costo, el algoritmo de gradiente descendente mejora la capacidad predictiva del modelo y lo hace más efectivo en la resolución de tareas complejas en diversos campos de aplicación.

2.6.6. Método del Gardiente Descendiente Estocástico

El método del gradiente estocástico (SGD, por sus siglas en inglés) es una variante del método del gradiente descendente que se utiliza ampliamente en el entrenamiento de redes neuronales.(22) A diferencia del gradiente descendente tradicional, que calcula el gradiente de la función de costo utilizando todo el conjunto de datos de entrenamiento en cada iteración, el SGD calcula el gradiente utilizando solo una muestra aleatoria de datos en cada iteración, es decir, se toma aleatoriamente fila a fila de la información que se le va a pasar a la red neuronal.

El SGD es importante en el entrenamiento de redes neuronales por varias razones. En primer lugar, permite que el algoritmo de entrenamiento maneje grandes conjuntos de datos de manera más eficiente al calcular

el gradiente utilizando solo una muestra de datos en cada iteración. Esto reduce significativamente el tiempo de cálculo y hace que el entrenamiento sea más rápido, especialmente en conjuntos de datos masivos.(10)

Además, el SGD introduce una componente de aleatoriedad en el proceso de optimización, lo que puede ayudar a evitar mínimos locales en la función de costo y permitir que el algoritmo explore un espacio de búsqueda más amplio.(13) Esto puede conducir a modelos finales que generalicen mejor a datos nuevos y desconocidos.

2.6.7. Propagación hacia atras

La propagación hacia atrás, también conocida como backpropagation en inglés. Este algoritmo permite calcular de manera eficiente los gradientes de la función de pérdida con respecto a los parámetros de la red, lo que a su vez permite actualizar los pesos de la red usando los métodos de optimización como el método del gradiente descendente estocástico, todo esto para minimizar la pérdida durante el proceso de entrenamiento (35).

Para calcular el gradiente de la función de pérdida con respecto a los pesos de la red, el algoritmo de backpropagation utiliza la regla de la cadena de cálculo diferencial. En esencia, el algoritmo calcula los gradientes de la función de pérdida con respecto a las salidas de cada capa de la red, propagándolos hacia atrás desde la capa de salida hasta la capa de entrada. Esto se hace de manera eficiente utilizando la técnica de la retropropagación de gradientes, que calcula los gradientes de manera recursiva mediante la aplicación sucesiva de la regla de la cadena.

Durante la fase de retropropagación, calculamos los gradientes de la función de pérdida con respecto a los pesos de la red utilizando la regla de la cadena. (4) Supongamos que nuestra función de pérdida es J y queremos calcular $\frac{\partial J}{\partial w_{ij}^{(l)}}$, es decir, el gradiente de la función de pérdida con respecto al peso $w_{ij}^{(l)}$ en la capa l.

Por la regla de la cadena, podemos escribir:

$$\frac{\partial J}{\partial w_{ij}^{(l)}} = \frac{\partial J}{\partial z_i^{(l)}} \frac{\partial z_i^{(l)}}{\partial w_{ij}^{(l)}}$$

donde:

- $\frac{\partial J}{\partial z_i^{(l)}}$ es el gradiente de la función de pérdida con respecto a la entrada ponderada $z_i^{(l)}$ de la neurona i en la capa l, y se calcula utilizando la regla de la cadena y los gradientes de las capas posteriores. - $\frac{\partial z_i^{(l)}}{\partial w_{ij}^{(l)}}$ es la derivada de la entrada ponderada $z_i^{(l)}$ con respecto al peso $w_{ij}^{(l)}$, que es simplemente $a_j^{(l-1)}$, la activación de la neurona j en la capa $l-1$.

Por lo tanto, podemos calcular el gradiente $\frac{\partial J}{\partial w_{ij}^{(l)}}$ recursivamente utilizando los gradientes de las capas posteriores y las activaciones de las capas anteriores.

2.6.8. Deep Q-Learning

El término $DeepQ-learning$ se refiere a la aplicación de los principios y técnicas del Deep Learning, en este enfoque, se emplean redes neuronales profundas para representar la función Q que será el indicador de calidad de la decisión en un momento dado, permitiendo al agente manejar entornos de mayor complejidad y dimensiones de manera más efectiva que con métodos tradicionales. Mientras que los métodos convencionales, como la programación dinámica y los métodos basados en tablas, utilizan estructuras discretizadas para almacenar los valores de calidad para cada estado-acción posible, el Deep Q-learning adopta un enfoque diferente al modelar la función Q como una red neuronal. Esta estructura de red neuronal permite al agente capturar relaciones más complejas entre las entradas del entorno y las acciones, sin discretizar el espacio de estados, lo que potencialmente lleva a un mejor rendimiento y una mayor generalización en una variedad de tareas de aprendizaje por refuerzo.

Así, al cambiar la estructura Q por una red neuronal, el proceso de aprendizaje del agente adquiere una nueva dinámica. En lugar de actualizar directamente una matriz de calidad Q como en el algoritmo Q-learning

clásico, ahora el agente ajusta los pesos de los inputs de la neurona (tamaño o forma de los datos del entorno) mediante una técnica conocida como propagación hacia atrás, para poder tomar una acción la cual obtenga una calidad máxima (outputs de la neurona). Este enfoque implica calcular el error (Función de pérdida) entre la salida de la red neuronal y el valor objetivo, que se deriva del diferencial temporal.

Para mejorar la convergencia del agente y abordar el desafío de las observaciones no independientes e idénticamente distribuidas (iid), se implementa una estructura de memoria cíclica. Esta memoria actúa como un almacén de experiencias pasadas del agente, permitiéndole acceder y reutilizar información relevante durante el proceso de aprendizaje. Cuando el agente se encuentra en una situación que ya ha experimentado previamente, en lugar de depender únicamente de las observaciones actuales, recurre a la memoria para tomar decisiones más informadas y precisas.

La memoria cíclica, también conocida como replay buffer, almacena una colección de transiciones pasadas del agente, cada una consistente en una observación, la acción tomada, la recompensa recibida y la siguiente observación(37). Esta estructura proporciona una manera eficiente de mantener una muestra diversa de experiencias pasadas, lo que ayuda a mitigar la correlación temporal entre las observaciones y promueve un aprendizaje más robusto y generalizable.

Durante el entrenamiento, el agente selecciona aleatoriamente muestras de la memoria cíclica para actualizar su red neuronal, lo que ayuda a romper la dependencia entre las observaciones consecutivas y promueve una convergencia más rápida hacia una política óptima (24). Este enfoque también permite al agente aprender de manera más efectiva en entornos con transiciones no estacionarias o datos altamente correlacionados, lo que mejora su capacidad para adaptarse a cambios en el entorno y generalizar a nuevas situaciones.

2.7. Proceso de Digitalización de imágenes

En vista de que la sección a tratarse es muy extensa, solo se abordarán los temas necesarios para la compresión adecuada del funcionamiento de la red CNN.

2.7.1. Pixel

Un píxel es la unidad más pequeña de una imagen digital. Cada píxel representa un único punto en la imagen y contiene información sobre el color o la intensidad en ese punto. Es un polígono de color constante. (34)

Importancia del Píxel

- **Resolución**: La resolución de una imagen se refiere al número total de píxeles que contiene. Una mayor cantidad de píxeles generalmente significa una imagen más detallada.
- **Operaciones Matemáticas**: En procesamiento de imágenes, muchas operaciones matemáticas se realizan a nivel de píxeles, como el filtrado, la transformación y la convolución.
- **Transformaciones y Análisis**: Los píxeles permiten representar y manipular imágenes mediante algoritmos que operan sobre sus valores, facilitando tareas como el reconocimiento de patrones, la segmentación y la mejora de la calidad de la imagen.

2.7.2. Imagen Digital

Una imagen digital de $M \times N$ píxeles en escala de grises puede ser vista como una función que asigna a cada par de coordenadas (i, j) (donde i y j son índices del píxel en las dimensiones vertical y horizontal, respectivamente) a un valor de intensidad de gris.

Esta función se define de la siguiente manera:

$$f : \{0, 1, \ldots, M-1\} \times \{0, 1, \ldots, N-1\} \to \{0, 1, \ldots, L-1\}$$

Donde:

- M es el número de filas (altura de la imagen).
- N es el número de columnas (anchura de la imagen).
- L es el número de niveles de gris posibles (por ejemplo, 256 para una imagen de 8 bits).
- $f(i, j)$ representa la intensidad de gris en el píxel ubicado en la fila i y la columna j.

2.7.3. Niveles de Gris

El número de niveles de gris posibles (L) determina cuántos valores distintos de intensidad puede tomar cada píxel en la imagen.

- En una imagen de 8 bits por píxel, $L = 256$, lo que significa que cada píxel puede tener 256 valores distintos de intensidad de gris, desde 0 hasta 255. Aquí, 0 representa el negro total y 255 representa el blanco total, con valores intermedios representando diferentes tonos de gris.
- En una imagen de 16 bits por píxel, $L = 65536$, lo que significa que cada píxel puede tener 65536 valores distintos de intensidad de gris, proporcionando una representación mucho más precisa de los tonos grises, desde 0 (negro) hasta 65535 (blanco).
- En una imagen de k bits por píxel, $L = 2^k$. Esto generaliza el número de niveles de gris posibles, donde k es el número de bits utilizados para representar cada píxel.

El valor de L afecta la profundidad de color de la imagen, es decir, la cantidad de información que se puede almacenar en cada píxel. A mayor cantidad de niveles de gris (L), más detallada y suave puede ser la transición entre diferentes intensidades de gris en la imagen.

2.7.4. Cuantificación de una imagen

Se define como `imagen` a un conjunto de píxeles, los cuales poseen un rango de color de 0 a 255 en el caso de imágenes a color, debido al canal RGB que posee la imagen (26). En este canal, cada píxel tiene tres valores correspondientes a los colores Rojo (R), Verde (G) y Azul (B). En imágenes en blanco y negro, los píxeles tienen un rango de 0 a 1, donde 0 representa la ausencia de luz (negro) y 1 representa la máxima intensidad de luz (blanco).

Por ejemplo, una imagen en escala de grises de 4 píxeles de ancho por 3 píxeles de alto se puede representar como una matriz 3×4:

$$\begin{bmatrix} 255 & 128 & 64 & 0 \\ 100 & 150 & 200 & 250 \\ 50 & 75 & 125 & 175 \end{bmatrix}$$

Aquí, el valor de cada entrada en la matriz corresponde a la intensidad de gris del píxel en esa posición.

Para entender cómo una computadora procesa una imagen, es importante considerar que una imagen digital es un tensor de dimensión 3. Así, una imagen I puede representarse como un tensor de la forma $I \in \mathbb{R}^{H \times W \times C}$, donde H es la altura de la imagen, W es la anchura y C es el número de canales de color (3 para imágenes RGB y 1 para imágenes en blanco y negro).

Por ejemplo, una imagen a color de 256×256 píxeles puede representarse como un tensor $I \in \mathbb{R}^{256 \times 256 \times 3}$. Cada elemento $I_{h,w,c}$ en este tensor representa la intensidad del color del canal c en la posición (h, w). (34)

$$I_{h,w,c} \in [0, 255] \quad \text{para imágenes RGB}$$

$$I_{h,w} \in [0, 1] \quad \text{para imágenes en blanco y negro}$$

Así, una vez introducido el concepto de imagen y cómo la computadora

entiende su estructura, se puede proceder a explicar el concepto de Redes Neuronales Convolucionales (CNN).

2.8. Procesamiento de imágenes

Para procesar imágenes reales o analógicas, se utiliza un tipo especial de red neuronal llamada Red Neuronal Convolucional (CNN, por sus siglas en inglés). Las CNNs son una arquitectura de red profunda diseñada específicamente para procesar datos con una estructura de cuadrícula, como las imágenes.

Las redes neuronales convolucionales son particularmente efectivas para el procesamiento de imágenes debido a su capacidad para mantener la relación espacial entre los píxeles y para aprender características jerárquicas. Esta estructura permite que las CNNs sobresalgan en tareas como la clasificación de imágenes, la detección de objetos y la segmentación semántica.

2.8.1. Redes Neuronales Convolucionales (CNN)

Las redes neuronales convolucionales, son una extensión de las redes neuronales artificiales (ANN) en las que se incluyen capas convolucionales para aprender a extraer, de forma automática, las características de una imagen dada como input en la red, lo cual no hacen las redes ANN. Cuanto más larga (o más profunda) es la red, mayor cantidad de detalles podrá aprender a distinguir. Esto es lo que ha propiciado la aparición del término `aprendizaje profundo` (8)

Estas redes utilizan un mecanismo que se divide en cuatro fases principales:

1. **Extracción de Características o Convolución**
2. **Max Pooling**
3. **Aplanamiento o Flattening**

4. **Capa Full Connection**

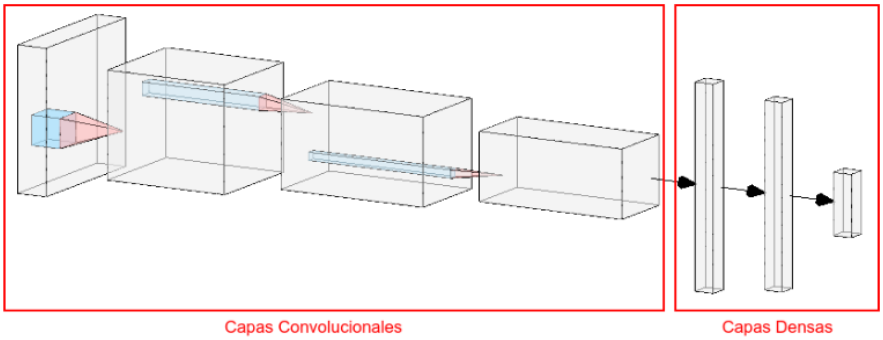


Figura 2.1: Estructura general de una CNN.

2.8.2. Convolución

La convolución es una operación en la cual se aplica un filtro o kernel a la imagen de entrada. Este kernel no es más que una matriz con pesos que se centra en cada uno de los valores de la entrada para calcular una media ponderada de estos valores, siendo las ponderaciones los valores del filtro.(1) Generalmente, el tamaño de los filtros es impar y cuadrada.

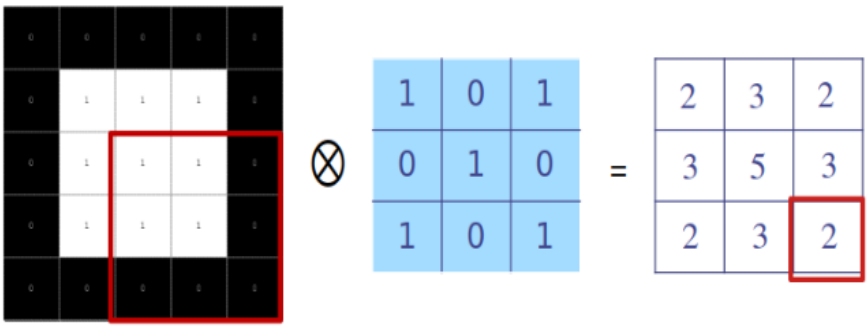


Figura 2.2: Ejemplo de convolución. De izquierda a derecha

En general, una convolución en dos dimensiones se define como:

$$M(x,y) = \sum_{s=-a}^{a} \sum_{t=-b}^{b} F(s,t)E(x-s,y-t)$$

Donde:

- F es el filtro a aplicar
- E es la matriz de entrada
- M se le llama el `mapa de características`
- a y b son los tamaños de los desplazamientos desde el centro del filtro a cualquier otro valor.

Por ejemplo, si la entrada es una imagen, es posible definir filtros que realcen los bordes, que los suavicen o incluso que detecten dichos bordes y cómo de marcados están.(8) Además, hay que tomar en cuenta que ya que se está procesando una imagen se pueden usar varios filtros para detectar distintas características al mismo tiempo, haciendo que la red sea más robusta debido a que conoce más patrones.

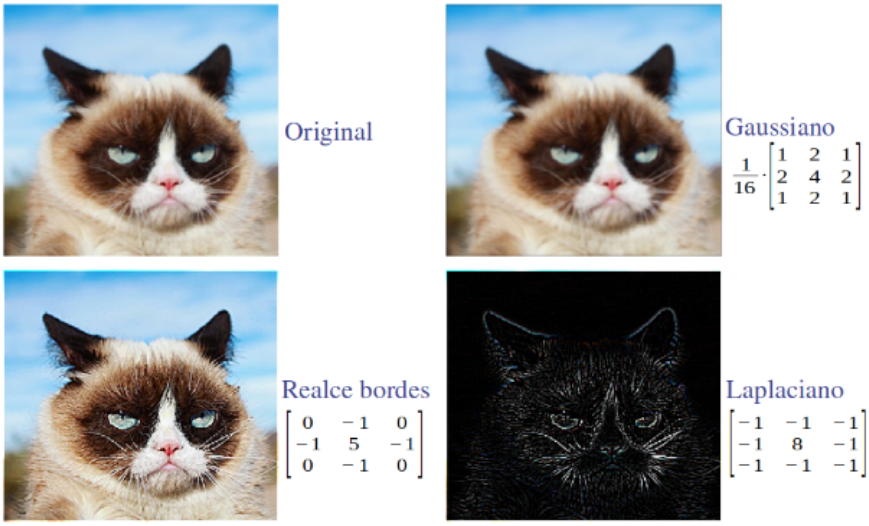


Figura 2.3: Resultado de aplicar diferentes filtros de convolución sobre una imagen dada.

Se puede observar que la imagen se reduce inevitablemente por la operación de convolución. Esto se debe a que esta operación en el espacio de las matrices se puede considerar como una multiplicación de matrices tradicional pero respetando la dimensión del filtro, lo cual reduce la imagen. Sin embargo, en vez de representar una pérdida de información, esto hace que se preserven las características que se desean detectar a través de la red neuronal convolucional.(30)

Para evitar la reducción excesiva del tamaño de la imagen, se pueden aplicar técnicas como el *padding*, que consiste en añadir bordes de ceros alrededor de la imagen de entrada antes de aplicar la convolución.(36) Esto ayuda a mantener las dimensiones originales de la imagen.

A diferencia de las redes neuronales tradicionales, que utilizan perceptrones, las CNN utilizan neuronas convolucionales.

Si se imagina una capa convolucional en la red se puede notar que las neuronas en una capa convolucional no están conectadas individualmente a cada píxel de la entrada como en una red totalmente conectada. En lugar de eso, cada neurona en la capa aplica un filtro sobre la imagen.

Cada filtro en cada neurona produce su propio mapa de características. Así, si hay varios filtros en una capa convolucional, se generarán varios mapas de características.

La salida de una neurona convolucional (Y), se obtiene:

$$Y = g(F \otimes E + B)$$

De donde, g es la función de activación y B es una matriz de términos independientes, con valores todos iguales.

La salida de una capa convolucional no es una matriz 2D, sino un conjunto de matrices 2D (mapas de características) apiladas, esto se puede visualizar como una matriz 3D donde una dimensión corresponde a la altura, otra a la anchura y la tercera a la profundidad (el número de filtros aplicados). (8) Por tanto, si se aplica 10 filtros a una imagen de entrada de tamaño 32×32 píxeles, la salida será una matriz de tamaño $32 \times 32 \times 10$.

Debido a que los valores de los filtros pueden ser cualquiera dependiendo del problema que se quiera resolver, los frameworks actuales ajustan estos valores y tamaño de los kernels durante el proceso de entrena-

miento de la CNN junto con el resto de pesos de la red. Esto permite encontrar los valores del filtro que maximicen la precisión de la red.

2.8.3. Capa Relu

Debido a que cada neurona aplica un filtro a la imagen y luego activa el mapa de características, esto se logra utilizando una función de activación (g) que, para este estudio, será la función de activación ReLU (Rectified Linear Unit) (1). Esta función introduce no linealidad en el mapa de características, debido a que la operación de convolución da como resultado una combinación lineal de los píxeles de la imagen con los pesos del filtro.

Al aplicar la función ReLU sobre el mapa de características, esto hace que los valores negativos del mapa se vuelvan ceros y los positivos se mantengan, lo cual rompe la linealidad en los mapas, permitiendo que la red modele relaciones no lineales entre las características.

Al eliminar los valores negativos, se reduce el ruido y se mejoran las características útiles para el aprendizaje. La función ReLU ayuda a eliminar efectos indeseables como sombras y luces excesivas que podrían dificultar la correcta identificación de características importantes (30). Así, se resaltan más claramente las fronteras y los bordes de los objetos en la imagen. Esto se traduce en una mayor definición y separación de los objetos presentes en la imagen, mejorando la capacidad de la red para identificar y clasificar diferentes elementos dentro de una imagen.

Cuando la salida de cada capa en la red CNN pasa a la siguiente capa convolucional, se realiza una nueva convolución seguida de otra aplicación de ReLU en cada neurona. A medida que se agregan más capas, la red puede aprender representaciones cada vez más complejas de los datos de entrada. Cada capa adicional permite la combinación de características detectadas en capas anteriores de manera no lineal, lo que incrementa la complejidad de los patrones que la red puede aprender.

Sin la no linealidad, la red sería equivalente a una sola capa de una transformación lineal, limitando significativamente su capacidad de modelado.

Una ventaja de la función ReLU es que ayuda a mitigar el problema del desvanecimiento del gradiente, común en otras funciones de activación. Este problema ocurre cuando los gradientes se vuelven muy pequeños durante la retropropagación, dificultando el ajuste de los pesos. ReLU, al tener gradientes constantes para valores positivos, facilita un entrenamiento más rápido y efectivo de la red. (26)

Por tanto, reconocer un objeto en una imagen no es simplemente una cuestión de sumar los valores de los píxeles; se necesita entender las relaciones espaciales y las combinaciones de características que forman los contornos y formas del objeto.

2.8.4. Max Pooling

Hasta ahora, la ejecución en secuencia de varias capas convolucionales ha sido muy efectiva a la hora de decidir si ciertas características están o no presentes en la entrada. Sin embargo, se mantiene la localización espacial de las características encontradas, lo cual, aunque no es una desventaja, puede ser una limitante.

Para introducir la invarianza espacial, se utilizan capas de agrupación o *Max Pooling*. Estas capas agrupan un número de valores adyacentes de los mapas de características, tomando la información más relevante.

El proceso consiste en tomar submatrices (por lo general de 2×2 o 4×4) de los mapas de características ya activados y aplicar la operación de *máximo* en cada submatriz.(8) Esto crea un mapa de características más pequeño, pero elimina problemas de transformaciones afines como

rotación, traslación, y escalado.

Invariancia a la Traslación

Cuando se aplica *Max Pooling*, se selecciona el valor máximo dentro de cada submatriz de tamaño fijo (por ejemplo, 2×2). Esto significa que, si una característica importante (como un borde o una esquina) se desplaza ligeramente dentro del área cubierta por la submatriz, el valor máximo aún capturará esta característica. Así, la red puede detectar características importantes incluso si se mueven ligeramente dentro de la imagen.

Invariancia a la Rotación

Aunque *Max Pooling* no introduce invariancia completa a la rotación, ayuda a reducir la sensibilidad a pequeñas rotaciones. Si una característica importante rota dentro de la submatriz, el valor máximo seguirá capturando la presencia de esa característica, aunque en una posición diferente.

Invariancia al Escalado

Max Pooling ayuda a mitigar la variabilidad introducida por cambios en la escala de las características. Al reducir la resolución espacial de los mapas de características, la operación de pooling agrupa las activaciones más importantes. Aunque esto no proporciona invariancia completa al escalado, facilita que la red detecte patrones similares en diferentes tamaños.

Reducción del Ruido y Enfoque en Características Importantes

La operación de *Max Pooling* reduce el ruido en los mapas de características, ya que elige el valor más significativo dentro de cada submatriz,

ignorando los valores menores que podrían ser ruido. Esto ayuda a enfocar la red en las características más prominentes y robustas, haciendo que la red sea menos sensible a variaciones no importantes en la imagen.

Por lo que, este proceso consigue un procesamiento más rápido y eficiente de la información relevante.

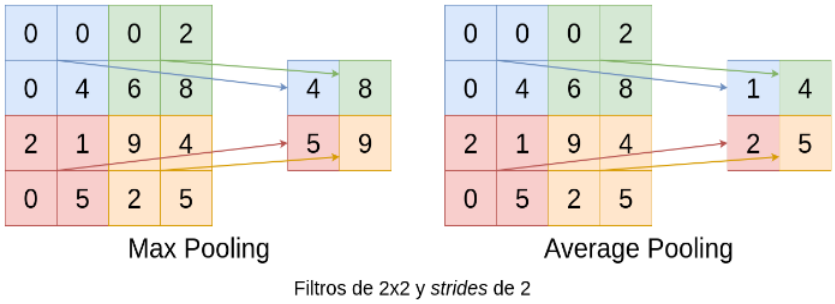


Figura 2.4: Ejemplo de MaxPooling

La operación de *Max Pooling* se define matemáticamente como:

$$M(x, y) = \text{máx}\{E(i, j) \mid i \in [x, x + p - 1], j \in [y, y + q - 1]\}$$

Donde:

- E es el mapa de características de entrada.
- M es el mapa de características de salida.
- $p \times q$ es el tamaño de la submatriz utilizada para el pooling.

Por lo tanto, el uso de *Max Pooling* en las redes neuronales convolucionales no solo reduce el tamaño de los mapas de características, sino que también introduce una invarianza a pequeñas transformaciones en la entrada. (1) Esto permite que la red se enfoque en las características más importantes y robustas, mejorando su capacidad para generalizar en diferentes condiciones y variaciones de la entrada.

2.8.5. Flattening

Después de aplicar múltiples capas convolucionales y de pooling, la estructura tridimensional resultante de los mapas de características debe ser transformada en una estructura unidimensional para que pueda ser procesada por las capas completamente conectadas (fully connected layers) de la red, ya que esto permite que los datos sean compatibles con estas capas, que se utilizan para la clasificación final o cualquier otra tarea de salida. Este proceso se conoce como *flattening*.

Así, después de las operaciones convolucionales y de pooling se obtiene una matriz tridimensional de tamaño $a \times b \times c$(donde a y b representan las dimensiones espaciales y c es el número de mapas de características), el flattening reorganiza estos elementos en un vector de longitud $a * b * c$.

Supongamos que tenemos un mapa de características $2 \times 2 \times 2$:

$$\text{Mapa de características} = \begin{bmatrix} a_{111} & a_{112} \\ a_{121} & a_{122} \end{bmatrix}, \begin{bmatrix} a_{211} & a_{212} \\ a_{221} & a_{222} \end{bmatrix}$$

Para aplicar el flattening, convertimos esta matriz en un vector unidimensional, respetando las posiciones de cada matriz en el mapa:

$$\text{Vector aplanado} = x = \begin{bmatrix} a_{111} & a_{112} & a_{121} & a_{122} & a_{211} & a_{212} & a_{221} & a_{222} \end{bmatrix}$$

Como se puede observar, el flattening ha transformado el mapa de características de una matriz $2 \times 2 \times 2$ a un vector de dimensión 1×8.

Es importante tener en cuenta que, aunque el flattening reestructura los datos, la relación espacial entre los píxeles se mantiene implícita en la secuencia de los elementos del vector. La secuencia de elementos en el vector resultante debe respetar el orden de los elementos en el tensor original para asegurar que la información relevante se conserve correctamente.

2.8.6. Full Conection

Después de obtener el flattening de los mapas de características, se procede a pasar este vector plano a través de una o más capas completamente conectadas (fully connected layers). En esta capa, cada neurona está conectada a todas las neuronas de la capa anterior, y sus salidas se conectan a todas las neuronas de la siguiente capa. Esta conectividad densa permite capturar relaciones complejas entre las características extraídas en capas anteriores.

Así, como si fuera una red neuronal convencional, el vector plano (flattening) se multiplica por una matriz de pesos que conectan las neuronas de la capa anterior con la siguiente. A este resultado se le suma un vector de sesgos y finalmente se activa utilizando la función de activación RELU. De esta manera, se obtiene un vector de salida $(\widehat{y})$

$$\widehat{y} = f(Wx + b)$$

De donde, se sabe que $W \in \mathbb{R}^{m \times n}$ es la matriz de pesos que conecta las n neuronas de la capa de entrada con las m neuronas de la capa completamente conectada y $b \in \mathbb{R}^m$ es un vector de sesgos.

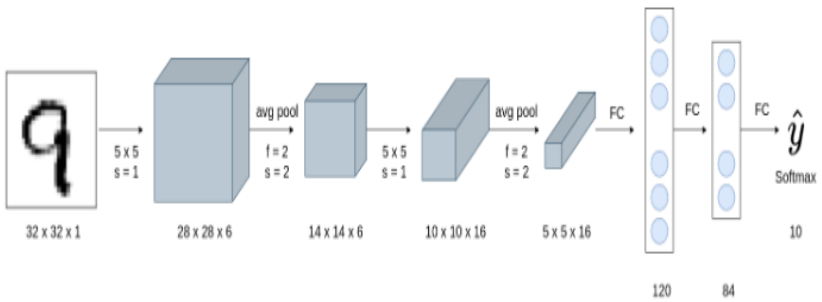


Figura 2.5: Arquitectura LeNet

Sobreajuste

Una ventaja clave de las capas completamente conectadas es su capacidad para capturar relaciones complejas entre las características. Esto significa que la red neuronal puede aprender a reconocer patrones sofisticados y realizar tareas complejas, como la clasificación de imágenes o el procesamiento del lenguaje natural, al combinar diferentes características de manera eficaz. Sin embargo, debido a que cada neurona de una capa completamente conectada está conectada a todas las neuronas de la capa anterior, el número de parámetros entrenables en la red puede aumentar significativamente (36). Esto puede llevar a problemas de sobreajuste, donde, cuanto mayor es el número de parámetros de la red, mayor probabilidad hay de que la red `memorice` los datos de entrenamiento y no le permita una generalización de estos. Esto ocurre porque la red se ajusta demasiado a los detalles y el ruido presente en los datos de entrenamiento, lo que deteriora su rendimiento en datos no vistos.

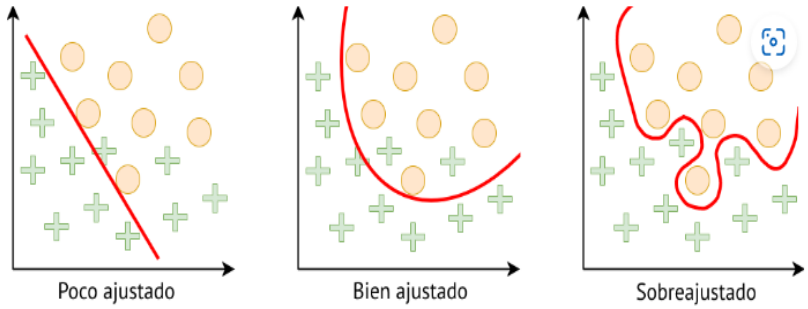


Figura 2.6: Tipos de ajuste del modelo a los datos.

Por lo que, para evitar que se produzca el sobreajuste se suelen emplear técnicas de regularización, que son técnicas que impiden que los modelos sean demasiado complejos mejorando su capacidad de generalización (34).

- *Dropout:* durante el entrenamiento, algunas activaciones se ponen a 0 de forma aleatoria (entre el 10 % y el 50 %). Esto hace que una capa de la red no dependa siempre de los mismos nodos anteriores.

- *Early Stopping:* se utilizan dos conjuntos: uno de entrenamiento y otro de validación. Cuando las curvas de pérdida de ambos conjuntos comienzan a divergir, se para el entrenamiento y se selecciona el modelo resultante del momento anterior al comienzo de la divergencia.

- *Regularización L1:* Penaliza los pesos grandes, por lo que fuerza a los pesos a tener valores cercanos a 0 (sin ser 0), es decir, añade un término de penalización α a la función de coste.

$$C_{reg}(W) = C(W) + \alpha||W||_1 = C(W) + \alpha \sum_i \sum_j |wij|$$

  Donde, se sabe que C es la función de costos y W es la matriz de pesos de la red.

2.8.7. Capa de Salida

Una vez se obtiene el vector de salida $(\widehat{y})$ de las capas full connection, el siguiente paso es procesar dicho vector en la capa de salida. Esta etapa es crucial, ya que transforma las características aprendidas y ajustadas por la red en las predicciones finales del modelo.

En la capa de salida, cada neurona representa una clase diferente en un problema de clasificación. Es decir, si estamos tratando de clasificar imágenes en k categorías diferentes (por ejemplo, perros, gatos, caballos, etc.), la capa de salida tendrá k neuronas, cada una correspondiendo a una de estas categorías.

El objetivo de esta capa es producir una probabilidad para cada clase, de modo que la suma de todas las probabilidades sea igual a uno, cumpliendo así las propiedades de una distribución de probabilidad. Esto se logra mediante la función de activación Softmax, ya que, inicialmente, las neuronas en la capa de salida devuelven valores lineales que no representan probabilidades. Por ello, es necesario transformar estos valores

z_i, que representan la salida lineal de la i-ésima neurona en la capa de salida.

Función de Softmax

La función de activación Softmax se utiliza para normalizar las salidas de las neuronas a un rango de $[0, 1]$ y asegurarse de que la suma de todas las salidas de las neuronas en la capa sea igual a uno, proporcionando así una interpretación probabilística de las salidas de la red.

Matemáticamente:

$$\sigma(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{k} e^{z_j}}$$

Donde:

- e es la base del logaritmo natural.
- z_i es la salida lineal de la i-ésima neurona.
- k es el número de clases.

Notemos que esta función es diferenciable, lo cual es perfecto para el entrenamiento de la red neuronal convolucional (CNN), ya que estas redes se optimizan mediante gradientes, en particular en optimizador de Adam para el presento proyecto. Además, la función Softmax amplifica las diferencias entre los valores grandes y pequeños en el vector de entrada, lo que facilita la distinción clara entre clases en la salida.

A diferencia de la función `Sigmoid`, que se utiliza en problemas de clasificación binaria, Softmax es adecuada para problemas de clasificación con múltiples clases debido a su capacidad de generar una distribución de probabilidad sobre todas las clases posibles.

Función de Entropía Cruzada

La función de entropía cruzada es una función de pérdida comúnmente utilizada para problemas de clasificación. Esta función mide la

discrepancia entre dos distribuciones de probabilidad: la distribución de probabilidad predicha por el modelo y la distribución de probabilidad real (verdadera) de las etiquetas.

Para un problema de clasificación con k categorías, donde y_i es la etiqueta verdadera en formato one-hot (un vector de tamaño k donde el elemento correspondiente a la clase verdadera es 1, y todos los demás elementos son 0) y $\hat{y}_i$ es la probabilidad predicha por el modelo para la clase i, producida por la función Softmax, la entropía cruzada se define matemáticamente como:

$$L(y, \hat{y}) = -\sum_{i=1}^{k} y_i \log(\hat{y}_i)$$

Esta función de pérdida, además de ser diferenciable, penaliza fuertemente las predicciones incorrectas con alta confianza. Por ejemplo, si el modelo predice una probabilidad alta para una clase incorrecta, la pérdida será significativa. Esto incentiva al modelo a corregir errores de alta confianza rápidamente.

A diferencia de otras funciones de pérdida, como el error cuadrático medio (MSE), la entropía cruzada se adapta específicamente a problemas de clasificación multiclase, asegurando que la probabilidad total se distribuya correctamente entre las clases. Además, MSE puede sufrir problemas de saturación cuando las diferencias entre las salidas predichas y las verdaderas son grandes, lo que puede llevar a gradientes pequeños y aprendizaje lento. La entropía cruzada, en cambio, proporciona gradientes más significativos que ayudan en la convergencia rápida del modelo.

La combinación de Softmax y entropía cruzada asegura que la pérdida no solo mida la discrepancia entre las distribuciones, sino que también optimice la clasificación correcta de las clases.

En teoría de la información, *sorpresa* se refiere a lo inesperado que es un resultado. Si un evento es muy probable, su ocurrencia no es sor-

prendente. Sin embargo, si un evento es poco probable, su ocurrencia es muy sorprendente. La entropía cruzada mide esta sorpresa o incertidumbre. Así, al calcular la entropía cruzada, se suma la *sorpresa* para todas las clases posibles y se pondera cada una por la probabilidad de que esa clase sea la correcta según la distribución real.

Así, Minimizar la entropía cruzada es equivalente a maximizar la probabilidad de predecir las etiquetas correctas, lo que está intuitivamente alineado con el objetivo de la clasificación.

Finalemnte, una vez se haya calculado los valores de la funcion de entropía cruzada, se debe minimizar esta función de pérdida, lo cual, para este proyecto se utilizará el método del gradiente descendiente estocático. Por lo que, haciendo la retropropagación se puede llegar a una clasificación de la información con una alta probabilidad.

2.9. Modelo del Mundo

El `Modelo del Mundo` en el aprendizaje por refuerzo se refiere a una representación interna del entorno que el agente utiliza para predecir las consecuencias de sus acciones sin interactuar directamente con el entorno real. Esto puede incluir la predicción de estados futuros y recompensas, lo que permite al agente planificar y tomar decisiones más informadas. (2)

2.9.1. Ventajas del Modelo del Mundo

- **Planificación:** El agente puede simular futuros escenarios y evaluar diferentes estrategias antes de actuar, lo que puede conducir a decisiones más óptimas.
- **Reducción de Interacciones:** Al simular interacciones con el entorno, el agente puede reducir la cantidad de pruebas necesarias en el entorno real, lo cual es útil en situaciones donde las interacciones

son costosas o riesgosas.

- **Exploración y Explotación:** Facilita un equilibrio entre exploración y explotación al permitir al agente prever las consecuencias de sus acciones sin riesgo. (3)

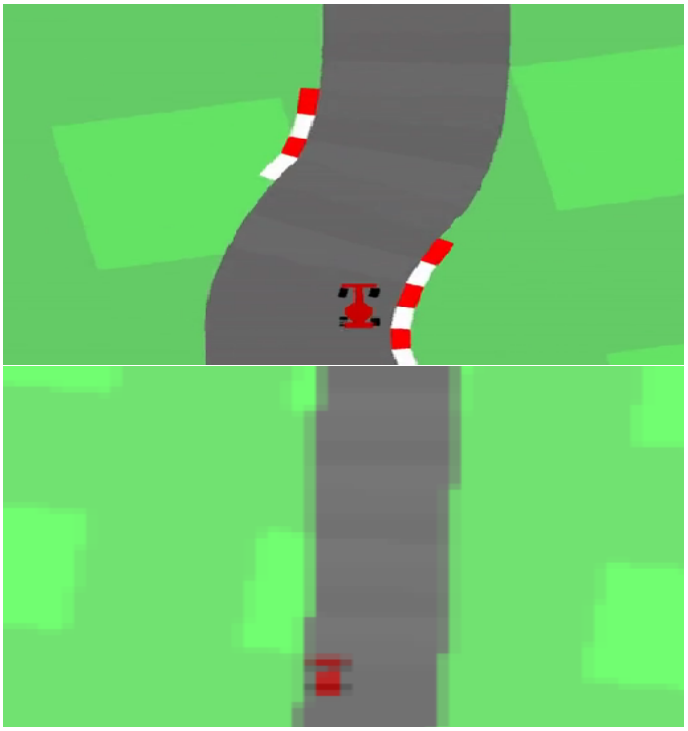


Figura 2.7: Agente entrenando en el entorno conectado. Agente entrenando y generando individualmente el entorno

Capítulo 3

Metodología

3.1. Entorno de desarrollo y herramientas utilizadas

En esta sección, describiremos las tecnologías, herramientas y librerías que serán empleadas en el desarrollo del proyecto. La elección adecuada de estas tecnologías es crucial para garantizar la eficiencia y efectividad del trabajo realizado.

3.1.1. Lenguaje de programación

Se ha elegido el lenguaje de programación Python 3.7 debido a su popularidad, versatilidad y amplio soporte en el ámbito de la inteligencia artificial y el aprendizaje automático. Su flexibilidad permite integrar diferentes componentes y tecnologías, mientras que su escalabilidad y rendimiento hacen posible la implementación eficiente de modelos de RL incluso en grandes conjuntos de datos.

Por otro lado, la versión elegida se debió a la compatiblilidad con las demás librarias a instalar.

3.1.2. Librerias y Frameworks

En cuanto a las librerías y frameworks, utilizaremos TensorFlow y PyTorch para el desarrollo de modelos de aprendizaje profundo. TensorFlow es una biblioteca de aprendizaje automático de código abierto desarrollada por Google, que ofrece una amplia gama de herramientas y recursos para la construcción y entrenamiento de modelos de aprendizaje automático y redes neuronales. Por otro lado, PyTorch es una biblioteca de aprendizaje automático de código abierto respaldada por Facebook, que se destaca por su flexibilidad y facilidad de uso, especialmente en el desarrollo de modelos de aprendizaje profundo.

Además, haremos uso de la librería AIgym, que ofrece una amplia variedad de entornos de simulación y herramientas para evaluar el rendimiento de los algoritmos de aprendizaje por refuerzo en tareas de control y toma de decisiones.

3.1.3. Entorno de Desarrollo Integrado (IDE)

Como entorno de desarrollo integrado (IDE), optaremos por Visual Studio Code, el cual, proporciona un entorno de desarrollo robusto que facilita la escritura de código, la visualización de datos y la documentación del mismo. Su interfaz intuitiva y personalizable nos permitirá realizar experimentos de manera eficiente y comprender mejor el proceso de investigación. Además, su integración con herramientas de control de versiones como Git nos ayudará a mantener un seguimiento del progreso del proyecto.

3.2. Agente QLearner en entornos simples

3.2.1. Algoritmo RL básico

Para comprender el concepto fundamental del Aprendizaje por Refuerzo (RL), es crucial entender cómo un agente aprende de las acciones que toma en su entorno mientras avanza en él. Un punto de partida es exa-

minar cómo funcionaría un agente que toma decisiones al azar y qué consecuencias acarrearía para dicho agente.

El siguiente algoritmo utiliza el entorno de apendizaje llamado "$MountainCar-v0$" de la libreria GYM (Anexo A.1):

1. **Iniciar:**
   - Importar la herramienta de simulación Gym.
   - Crear un entorno de simulación llamado "MountainCar-v0".
   - Definir el número máximo de intentos de aprendizaje.
2. **Para cada intento de aprendizaje:**
   - Reiniciar el entorno para comenzar desde el principio.
   - Establecer la recompensa total y el número de pasos en cero.
3. **Mientras el intento de aprendizaje no esté completo:**
   - Mostrar el entorno de simulación.
   - Tomar una decisión aleatoria de movimiento para el coche.
   - Ejecutar la decisión tomada y observar el resultado.
   - Actualizar la recompensa total y el número de pasos.
4. **Imprimir el resultado del intento de aprendizaje:**
   - Mostrar el número de pasos realizados.
   - Mostrar la recompensa total obtenida.
5. **Repetir el proceso hasta que se completen todos los intentos de aprendizaje.**
6. **Cerrar el entorno de simulación al finalizar.**

El análisis del algoritmo revela la ausencia de cualquier forma de retroalimentación del agente. En otras palabras, el agente no incorpora información de las acciones tomadas en cada estado mientras avanza.

3.2.2. Clase QLearner

Como se ha mencionado previamente, al agente le resulta crucial considerar las probabilidades asociadas a cada acción antes de decidir cuál sería la mejor elección. Por esta razón, se hace necesario implementar la clase QLearner. Esta clase le proporcionará al agente las herramientas necesarias para aprender a evaluar y seleccionar entre las diferentes acciones disponibles. Para alcanzar este objetivo, se deben incorporar las siguientes funciones dentro de la clase QLearner. (Anexo A.2)

Función $init$**:**

En esta función se inicializará el objeto $self$, el cual, a través de diferentes métodos, almacenará características esenciales del entorno. Entre ellas se incluyen: el tamaño del espacio de estados, el valor máximo dentro del espacio de estados, el tamaño del espacio de acciones, el factor de descuento (gamma) y la tasa de aprendizaje (alpha). Además, se llevará a cabo la discretización del espacio de acciones, una operación que puede variar dependiendo de las especificidades de cada entorno.

Función $discretize$**:**

En esta función se implementa de tal forma que se pueda determinar en qué discretización se encuentra el estado actual del agente. Esto es crucial para que el agente pueda interpretar y tomar decisiones basadas en su entorno.

Función $get\ action$**:**

Aquí es donde el agente tomará las decisiones basadas en las probabilidades de éxito (ϵ) o fracaso $(1-\epsilon)$. Así, como política de fuerza bruta se elige un epsilon mínimo, el cual será el valor que va aprendiendo mientras el incremento del aprendizaje $self.epsilon$ sea superior a ese valor, por lo que se elige la acción en base a este criterio. Por el contrario, si este criterio no se cumple, el agente tomará acciones aleatorias hasta poder cumplir con el criterio anterior. Esto es posible realizando un decremen-

to en $self.epsilon$ de acuerdo con el epsilon mínimo elegido y el número máximo de pasos a realizarse en todos los episodios.

Función $learn$**:**

En esta etapa de la implementación, el agente tiene la capacidad de aprender a partir de las acciones tomadas en el entorno en estados específicos. Aquí es donde se codifica la ecuación del diferencial temporal. Esta ecuación permite construir una matriz de calidad, que servirá como guía para que el agente pueda tomar decisiones más informadas y mejorar su desempeño a lo largo del tiempo, esta matriz refleja la estimación de la recompensa esperada para cada acción en cada estado, lo que ayuda al agente a elegir las acciones que maximizan su recompensa a largo plazo. Además, durante este proceso de aprendizaje, el agente ajusta gradualmente sus estimaciones de calidad a medida que explora y experimenta el entorno.

3.2.3. Entrenamiento QLearner

Con la clase Qlearner ya implementada, se procede a construir la función que utilizará sus métodos para actualizar la matriz Q. Esta actualización permite al agente tomar mejores decisiones a lo largo del tiempo, así como desarrollar una política eficiente para enfrentarse al entorno después de varios episodios de aprendizaje. La matriz Q es esencialmente una tabla que asigna un valor a cada par de estado-acción, representando la calidad estimada de tomar una acción en un estado específico. Al actualizar esta matriz utilizando la ecuación de actualización Q-learning, el agente puede aprender de su experiencia y mejorar su rendimiento en el entorno. Este proceso de aprendizaje iterativo permite al agente adaptarse y tomar decisiones más informadas a medida que interactúa con el entorno.(Anexo A.3)

Algoritmo de Entrenamiento

1. Se define una función llamada "train". Esta función recibe como argumentos el agente a entrenar y el entorno en el que va a aprender.

2. El bucle principal del entrenamiento se realiza a lo largo de un número predeterminado de episodios. En cada episodio:

   a) Se restablece el entorno a su estado inicial y se comienza a interactuar con él.

   b) El agente selecciona acciones en base a su estrategia de aprendizaje, utilizando la ecuación Q-learning para tomar decisiones.

   c) Después de ejecutar una acción:

      - Se observa el siguiente estado.
      - Se recibe una recompensa.
      - Se actualiza el conocimiento del agente sobre el entorno a través de un proceso de aprendizaje.

   d) El bucle continúa hasta que se alcanza un estado final en el episodio.

3. Al final de cada episodio:

   - Se acumula la recompensa total obtenida.
   - Se registra la mejor recompensa obtenida hasta el momento.
   - Se imprime información relevante, como el número de episodio, la recompensa total y la mejor recompensa alcanzada.

4. Al finalizar todos los episodios de entrenamiento:

   - Se devuelve la mejor política de acción aprendida por el agente, que es aquella que maximiza la estimación de calidad de acciones para cada estado.

5. Este proceso de entrenamiento permite al agente mejorar gradualmente su desempeño en el entorno, aprendiendo a tomar decisiones más efectivas y maximizando su recompensa total a lo largo del tiempo.

3.2.4. Aprendizaje

Se comienza por inicializar algunas variables necesarias para llevar a cabo la evaluación. Estas variables incluyen el estado inicial del entorno y el puntaje total, que se establece en cero al inicio de cada episodio de prueba. Una vez que se han preparado las variables iniciales, la función entra en un bucle principal donde se ejecuta la interacción entre el agente y el entorno.

El agente selecciona una acción en función de la política de acción proporcionada. Esta política puede ser determinada por el propio agente, por el método de discretización del estado. La acción seleccionada se ejecuta en el entorno, y como resultado, el entorno devuelve la próxima observación, la recompensa asociada con la acción tomada y una indicación de si el episodio ha terminado.

Este proceso de selección de acciones y actualización del entorno continúa hasta que el entorno indica que el episodio ha finalizado. (Anexo A.4)

3.3. Agente QLearner Optimizado

Una vez implementado el agente Q-learner con el método tradicional, el cual realiza discretizaciones en el espacio de estados, es importante recordar que para que el agente pueda aprender, se actualiza la matriz Q de manera que inicialmente se deben conocer y manipular las dimensiones del espacio de estados. Sin embargo, esta tarea puede volverse compleja debido a que muchos entornos poseen múltiples dimensiones, como la velocidad, la posición, el movimiento de una articulación, la identificación de herramientas disponibles, entre otros factores. Además, cuando se discretiza el espacio de estados, se dividen las posibles observaciones en un conjunto finito de categorías o valores discretos. Esto puede llevar a una pérdida de información y precisión, ya que ciertos detalles sutiles del entorno pueden perderse en la discretización. Debido a esto, el agente

puede necesitar más iteraciones para explorar completamente el espacio de estados discreto y aprender a tomar decisiones óptimas en todas las situaciones posibles.

Este tipo de problemas pueden ser solucionados implementando redes neuronales, ya que, permite una representación más continua y adaptable del entorno. En lugar de dividir el espacio de estados en categorías discretas, la red neuronal puede aprender a mapear directamente las observaciones del entorno a acciones óptimas. Esto significa que el agente puede capturar y procesar información más detallada y compleja del entorno, lo que le permite aprender de manera más eficiente y precisa. Además, al ser continuas, las redes neuronales pueden adaptarse mejor a cambios en el entorno o en las condiciones de operación, lo que las hace más robustas y versátiles en comparación con la discretización.

3.3.1. Perceptro

Se implementa en la clase SLP (Simple Layer Perceptron), donde la clase recibe el tamaño o la forma de los datos del espacio de estados y del espacio de acciones extraidos del entorno.

Con la ayuda de la librería $torch$, se calculan las combinaciones lineales de los inputs del perceptrón para luego ser enviadas a la capa intermedia ($self.hidden\_shape$). Posteriormente, se activan utilizando la función de activación ReLU ($torch.nn.functional.relu()$) para obtener como output una acción. (Anexo A.5)

3.3.2. Clase SwallowQLearner

Para poder utilizar el perceptrón anteriormente programado, se debe implementar la clase $SwallowQLearner$, que será bastante similar a la clase $QLearner$ previamente desarrollada, con ciertas modificaciones en partes de su código para adaptarse al uso del perceptrón. Esta clase permitirá al agente realizar aprendizaje por refuerzo utilizando el perceptrón como modelo de aprendizaje. (Anexo A.6)

Función $init$:

Al igual que en la clase $QLearner$, esta función inicializa y extrae los parámetros necesarios del entorno en el que se encuentra el agente. Además, se define el método de optimización Adam, que ajustará los parámetros de la red neuronal.

Dado que ya se tiene la estructura del perceptrón implementada, y este puede procesar el espacio de estados y dar como resultado una acción, pasa a sustituir a la matriz Q. Es decir, $self.Q$ ahora llamará a la clase SLP, así, la política de actuación no se ve afectada.

Por último, se inicializa la memoria, que será la estructura encargada de almacenar las experiencias de juego del agente.

Función $epsilon\_greedy\_Q$:

Para limpiar el código, se implementa la función $epsilon\_greedy\_Q$, que será la política de actuación del agente. Esta política elige un valor de epsilon aleatorio y lo suficientemente pequeño, de manera que las acciones tomadas se ejecuten de manera aleatoria hasta cierto umbral. A partir de ese umbral, el agente accederá al valor que maximiza la función de calidad $self.Q$ en cada estado.

Función get_action :

Por lo que, si la política de actuación se implementa en la función anterior, esta función pasa como argumento la observación del estado actual y lo envia a la función $epsilon\_greedy\_Q$:.

Función $replay\_experience$:

Esta función llamará a la clase $ExperienceMemory$ al tomar una muestra aleatoria de la memoria implementada y almacenada en dicha clase, para después enviarla a la función de entrenamiento $learn\_from\_batch\_experience$.

3.3.3. Memoria de Repetición

Se implementará la clase $ExperienceMemory$, que representa la memoria cíclica del agente. Esta memoria utiliza experiencias pasadas para aproximar de mejor manera los valores de calidad Q. La memoria se programará como una tupla que contendrá los siguientes parámetros: $[\text{"}obs\text{"}, \text{"}action\text{"}, \text{"}reward\text{"}, \text{"}next\_obs\text{"}, \text{"}done\text{"}]$. (Anexo A.7)

Función $init$:

En esta función se inicializará la capacidad de la memoria y se creará la estructura vacía de la tupla para guardar cada experiencia que consiga el agente, además de un identificador de cada una de ellas para saber su posición en la tupla $(self.memory)$.

Función $sample$:

Aquí se implementa la extracción de una parte de la memoria, es decir, se le pasará el tamaño de la muestra que se desea de la memoria y esta, utilizando la función $sample$, crea una muestra aleatoria que elige de la estructura $memory$.

Función get_size :

Devolverá el número de experiencias almacenadas en memoria.

Función $store$:

Al utilizar el código: $self.memory.insert((self.memory\_idx)\,\%self.capacity, exp)$ se desea insertar en la memoria el parámetro exp que es la experiencia actual que el agente ejecuta en el instante t en la posición $(self.memory\_idx)\,\%self.capacity$, la cual utiliza la operación "modulo"$(\%)$ para insertar la experiencia en las primeras posiciones de la tupla, provocando de esta manera que la memoria sea cíclica. Adicionamelmente se va sumando una unidad en el id de la experiencia $(self.memory\_idx)$

3.3.4. Entrenamiento SLP con Memoria de Repetición

Se implementa la función $learn\_from\_batch\_experience$, la cual tendrá como parámetro la experiencia en memoria. De esta experiencia, se extraen cada uno de los parámetros como un array. Luego, se calcula la variable td_target, que representa la calidad esperada para la observación actual (la función de calidad Q). Posteriormente, se obtiene la función de pérdida, que será el Error Cuadrático Medio del Diferencial Temporal, es decir, entre el output de la neurona y la calidad esperada, para posteriormente tomar la media de cada uno de estos errores de la muestra seleccionada. A continuación, se calculan los gradientes de la función de pérdida realizando una propagación hacia atrás con la función `backpropagation`, y finalmente se ajustan los pesos en la neurona con el optimizador de Adam mediante el método `.step()`. (Anexo A.8)

3.3.5. Aprendizaje

Así, con todas las clases ya implementadas y utilizando la librería GYM, se crea el entorno elegido y se inicializa el agente llamando a la clase `SwallowQLearner`, pasándole el environment seleccionado y guardando las puntuaciones en una lista.

Se inicializa el bucle de los episodios del environment, donde se proporciona información al agente y se crea un bucle para cada paso que se ejecutará en cada episodio. En cada paso, se toma una acción y se almacena en memoria para luego llamar a la función de aprendizaje `learn_from_batch_experience` y actualizar la observación para el siguiente paso, junto con la recompensa.

Finalmente, se establece un umbral para la actualización de la memoria cíclica. (Anexo A.9)

3.4. Aumento de la Capacidad de Aprendizaje usando una Red CNN

Un agente Q-Learner es una estructura que puede contar con herramientas muy potentes que le ayudan a tomar las mejores decisiones considerando el entorno en el que se desenvuelve. Es decir, programacionalmente se le pueden conectar estructuras más complejas que le permitan mejorar la función de calidad Q en el algoritmo Q-Learning. Mientras mejor pueda procesar su entorno y, con la ayuda de las recompensas proporcionadas por el entorno, aprenderá más rápido lo que debe hacer y así alcanzará su objetivo, sea cual sea este, logrando que la tarea asignada se cumpla de forma óptima y rápida, ya que el agente explora todas las posibilidades del entorno, encontrando errores en entornos programados como videojuegos.

Es por ello que, para optimizar aún más al agente, se procederá a conectar, además de la red neuronal (SPL), una red convolucional (CNN), con el fin de permitirle al agente procesar frames de píxeles como información. Utilizando la red convolucional, el agente puede reconocer lo que existe en el entorno y aprender de su utilidad durante la exploración que dura un episodio en el videojuego. Esto maximiza la calidad de sus acciones durante el proceso de entrenamiento, adaptando así a un agente que ya no depende de la estructura programacional y estructural del entorno, sino que posee una capacidad de predictiva mediante la red CNN. Esto le permite aprender de los elementos en el entorno, creando un agente independiente que puede entrenar, aprender en cualquier ambiente y llevando acabo cualquier tarea.(Anexo A.10)

3.4.1. Control de Parámetros

Como parte de la mejora del código del Agente, se procederá a crear un archivo .json, el cual constará de los parámetros requeridos tanto del Agente como del Entorno o Entornos a utilizar.

Dichos parámetros serán leídos, extraídos y modificados por una clase construida llamada `ParamsManager`. Así, en esta clase se implementan funciones para la gestión eficiente de los parámetros, su lectura, modificación y almacenamiento de manera estructurada.

Todo esto se hace para mantener el código limpio y organizado, permitiendo una fácil actualización y mantenimiento de los parámetros sin necesidad de modificar el código fuente del Agente directamente, de esta forma, se logra una mayor flexibilidad y modularidad en el manejo de configuraciones, facilitando la experimentación y el ajuste de los parámetros.(Anexo A.11)

3.4.2. Red CNN

Para integrar la red CNN al agente, se procederá a crear la clase `CNN`, la cual heredará del módulo *torch.nn.Module* de la librería *torch*.

La clase recibe el tamaño del input (*input_shape*), el tamaño de la salida (*output_shape*) y el dispositivo (`CPU o GPU`) en donde la red debe almacenar la información de cada iteración.

Así, en la `CNN` se inicializan tres capas convolucionales, las cuales utilizan la función *torch.nn.Sequential* para realizar una convolución 2D secuencial con la función *torch.nn.Conv2d*, y se activan con la función ReLU.

Cada capa tiene un diferente número de filtros (kernels) a aplicar a las imágenes de entrada. La primera capa aplica 64 filtros, mientras que la segunda y la tercera aplican 32 cada una. Es decir, cada capa aprende un número de características igual al número de filtros aplicados en esa capa.

Después del proceso de convolución, obtenemos una estructura ten-

sorial de $32 \times 18 \times 18$, es decir, 32 matrices de 18 filas y 18 columnas. Esta estructura se aplana (flattening) resultando en un vector de tamaño $32 * 18 * 18$, que se pasa a la capa totalmente conectada (fully connected) utilizando la función *torch.nn.Linear*. Esto considera el tamaño del flattening y el tamaño de la salida proporcionado como parámetro de entrada a la clase.

El dispositivo especificado (`CPU o GPU`) se maneja adecuadamente para asegurar que todos los cálculos se realicen en el hardware seleccionado, optimizando el almacenamiento y la velocidad de procesamiento. En este caso se utilizó la `GPU`.

Finalmente, se construye el método `forward`, que define cómo se pasa la entrada a través de las capas convolucionales y luego a la capa totalmente conectada. Este método determina el flujo de datos a través de la red, desde la entrada hasta la salida. La salida del método `forward` será el resultado de la red neuronal. (Anexo A.12)

3.4.3. Estrategias de Reescalado para Optimización Computacional

Para un agente Q-learning, cada fotograma que se almacena representa una instantánea del entorno en un momento específico. Dado que el agente debe capturar tanto el estado actual del entorno como el siguiente estado que resulta de una acción tomada, es necesario almacenar dos fotogramas consecutivos. Esto significa que la cantidad de información que debe ser gestionada en memoria se duplica, incrementando el costo computacional del procesamiento de imágenes de entrada.

Cada fotograma capturado no solo ocupa espacio en memoria, sino que también requiere una cantidad específica de bits para almacenar la información sobre los píxeles que componen la imagen. Por ejemplo, una imagen con una resolución de 84x84 píxeles en escala de grises puede ocupar 84x84 = 7056 bits (o aproximadamente 882 bytes si consideramos

8 bits por byte). Al almacenar dos fotogramas consecutivos, la cantidad total de bits de información requerida para los dos frames es duplicada.

Con el objetivo de mitigar este desafío computacional, se implementarán estrategias de reescalado. Estas estrategias optimizarán el procesamiento de las imágenes de entrada reduciendo su tamaño, sin comprometer la capacidad de la red para aprender y tomar decisiones precisas.

Reescalamiento de Entornos de la Atari (1977)

Debido a que los videojuegos de Atari (1977) poseen una cantidad baja de píxeles comparado con los videojuegos de la actualidad, estos entornos son propicios para que una red convolucional no tan compleja pueda capturar características de manera óptima, convirtiéndose en una herramienta de procesamiento muy potente.

Debido al funcionamiento del algoritmo Q-learning, las necesidades computacionales se duplican. Es por esto que cualquier entorno en el que el agente entrene será reescalado en diferentes clases, las cuales reescalan el frame de diferentes formas.

Inicialmente, se construirá la función `ProcessFrame84`, la cual transformará el tamaño de los frames de Atari, que son de $210 \times 160 \times 3$, a un tamaño de $84 \times 84 \times 1$ píxeles. Además, transformará los tres canales de colores al canal de grises. Así, la función recibe el frame de la imagen y una estructura para guardar el proceso.

Para poder reescalar los colores, se dividirá cada píxel de colores por 255, que es el rango de colores que posee el sistema RGB.

Debido a que se realizarán varias operaciones de reescalado diferentes, se construyeron las siguientes clases: (Anexo A.13)

Clase AtariRescale

Todas las clases heredarán de *gym.ObservationWrapper*. La cual modifica las observaciones que se obtienen del entorno.

Cuando se inicializa una instancia de AtariRescale, se configura la estructura BOX llamada *observation-shape* para esperar observaciones en forma de imágenes de 84×84 píxeles en escala de grises.

Una vez definida la inicialización de la clase, se adiciona la función *observation*, la cual, cada vez que se recibe una observación del entorno, se utiliza la función `processframes84` para transformar la imagen original a la nueva forma.

Clase NormalizedEnv

La clase NormalizedEnv se utiliza para normalizar las observaciones que se obtienen del entorno. Normalizar significa ajustar las observaciones para que tengan una media de 0 y una desviación estándar de 1, lo que facilita el entrenamiento de modelos de aprendizaje automático.

Se inicializarán varias variables necesarias para normalizar los datos de entrada, estas serán; la meadia, la desviación estandar, la constante alpha y el número de pasos realizados.

Cada vez que se recibe una observación del entorno, la clase actualiza estas estadísticas utilizando una media ponderada exponencialmente, que da más peso a las observaciones recientes. Luego, corrige cualquier sesgo introducido por este método de cálculo para obtener una media y desviación estándar insesgadas. Finalmente, la observación se normaliza restando la media no sesgada y dividiendo por la desviación estándar no sesgada, y se devuelve esta observación normalizada. Este proceso asegura que las observaciones se ajusten a una distribución normal estándar.

Clase ClipReward

La clase ClipReward se utiliza para simplificar las recompensas obtenidas del entorno.

Cada vez que se recibe una recompensa del entorno, el método *reward* transforma la recompensa utilizando la función *np.sign()*. Esto asegura que las recompensas sean consistentes y comparables entre diferentes entornos, aunque puede perder información detallada sobre la magnitud de las recompensas.

Clase NoopResetEnv

La clase NoopResetEnv se utiliza para evitar que el agente memorice el estado inicial del juego. Al reiniciar el entorno, la clase realiza un número aleatorio de acciones `NOOP` (sin operación) antes de que el agente tome el control. Esto asegura que el juego no comience siempre en el mismo estado, proporcionando variabilidad en los estados iniciales y obligando al agente a aprender estrategias más generales.

Cuando se inicializa una instancia de NoopResetEnv, se configura el entorno de Gym y se establece el número máximo de acciones NOOP que se realizarán al reiniciar. El método *reset* reinicia el entorno y ejecuta un número aleatorio de acciones NOOP. El método *step* simplemente pasa la acción al entorno sin modificaciones.

Clase EpisodicLifeEnv

La clase EpisodicLifeEnv modifica el comportamiento del entorno para que no se reinicie completamente cuando el agente pierde una vida.

El método *step* en lugar de reiniciar todo el entorno, marca el episodio como terminado temporalmente y permite al agente continuar desde donde perdió la vida. Esto simula una experiencia de vida continua en

el juego, donde la pérdida de una vida no significa el fin del juego, sino solo una penalización. Esta estrategia ayuda al agente a aprender de sus errores sin reiniciar todo el entorno, proporcionando un aprendizaje más eficiente y realista.

Clase MaxAndSkipEnv

La clase MaxAndSkipEnv modifica el comportamiento de los pasos (*steps*) y reinicios (*resets*) del entorno. Esta clase permite que el agente salte un número específico de pasos (por defecto 4) y devuelve el máximo de las últimas dos observaciones, lo que puede ayudar a estabilizar la entrada y reducir el costo computacional al procesar menos observaciones por segundo. También acumula la recompensa total de los pasos omitidos y proporciona un método para reiniciar el entorno, limpiando el buffer de observaciones y almacenando la observación inicial.

Clase FrameStack

La clase FrameStack apila los últimos k frames de observaciones del entorno. Esto permite que el agente procese múltiples frames simultáneamente, lo que puede mejorar la percepción temporal del agente y ayudar a la eficiencia en el uso de memoria RAM.

Durante el reinicio (*reset*), el buffer se inicializa con k copias de la observación inicial. Durante los pasos (*steps*), cada nueva observación se añade al buffer. La clase también asegura que siempre haya k frames en el buffer antes de devolver las observaciones apiladas, utilizando el objeto LazyFrames para reducir el costo computacional.

Clase LazyFrames

La clase LazyFrames se utiliza para manejar de manera eficiente una lista de frames apilados. En lugar de concatenar los frames inmediatamente, lo hace solo cuando es necesario, lo que ayuda a reducir el uso

de memoria y optimiza el rendimiento. Al concatenar los frames, libera la memoria ocupada por la lista original de frames.

Además, proporciona métodos para obtener la longitud del objeto y acceder a elementos específicos, asegurando que los frames estén concatenados antes de realizar estas operaciones.

3.4.4. Clase DeepQLearner

Se procede a modificar la clase SwallowQLearner. Ahora la clase ya no recibirá el entorno en específico, sino que solo recibirá la longitud del espacio de estados y la del espacio de acciones, además de los parámetros obtenidos desde el archivo .json.

Si el espacio de estados tiene tres dimensiones, el agente entenderá que está recibiendo entradas de imágenes y, por lo tanto, inicializará la matriz Q con la clase CNN.

La función *get-action* normaliza los colores de entrada, que en este caso serán imágenes, reorganiza las dimensiones de los frames y retorna la política de actuación del agente.

Se añadieron las funciones *load* y *save*, que permiten cargar y guardar el modelo entrenado en el entorno determinado. Estas funciones facilitan la continuidad del entrenamiento y la reutilización del agente en diferentes sesiones.(Anexo A.14)

3.4.5. Aprendizaje

Una vez escritas todas las clases, se las conecta a travez del método global `main`.

Así, el aprendizaje del Agente en este punto es el mismo que se utilizó en la clase SwallowQLearner, con la diferencia de que se reescala los entornos antes de iniciar el bucle principal del episodio. Además, se modifica la parte del código que contiene la constantes que ya se especificaron en el archivo jason. (Anexo A.15)

Capítulo 4

Resultados

4.1. Rendimiento del Agente Q-Learner

En este capítulo, se presentarán los resultados obtenidos en el desarrollo y aplicación del modelo de aprendizaje por refuerzo, basado en el algoritmo Q-Learning. Se comenzará mostrando los resultados del agente Q-Learner implementado utilizando la ecuación de Bellman, que representa la base fundamental del enfoque de aprendizaje. Se explorará cómo este agente logra aprender y tomar decisiones en un entorno dado, y se evaluará su rendimiento.

Posteriormente, se avanzará hacia una mejora significativa en el desempeño del agente mediante la implementación de redes neuronales convolucionales (CNN). Estas redes permitirán al agente comprender de manera más profunda y detallada su entorno, lo que resultará en una mejora significativa en la función de calidad Q. Se explorará cómo esta optimización con CNNs impacta en la capacidad del agente para tomar decisiones más informadas y precisas, y se analizará comparativamente su rendimiento con respecto a la versión inicial basada en la ecuación de Bellman.

A lo largo de este capítulo, se examinarán detalladamente los resulta-

dos obtenidos en cada fase del trabajo, destacando las mejoras observadas y proporcionando una comprensión profunda de cómo el enfoque de aprendizaje por refuerzo evoluciona y se adapta a medida que se integran tecnologías avanzadas como las redes neuronales convolucionales.

4.1.1. Evaluación en un entorno simple

Inicialmente, el agente se somete a un proceso de aprendizaje utilizando el algoritmo Q-Learner clásico, el cual se basa en el concepto de diferencial temporal para actualizar la matriz de aprendizaje Q. Este enfoque se implementa en el entorno $"MountainCar - v0"$ de la librería Gym, que es un videojuego simple donde un carro debe desplazarse hacia la derecha o la izquierda con una velocidad determinada para alcanzar una bandera amarilla ubicada en una colina. El entorno se considera simple debido a que solo tiene dos dimensiones: la posición del carro y su velocidad. Además, el espacio de las acciones posibles está descrito de la siguiente manera: ir a la derecha (2), ir a la izquierda (1) o no hacer nada (0).

Después de 50000 episodios de entrenamiento, con 200 acciones realizadas por episodio (debido a la limitación del entorno de permitir solo 200 iteraciones por episodio, esto varía según el entorno), el agente logra completar el juego sin perder a partir del episodio 512. Esto significa que el agente alcanza la meta en menos acciones, lo que resulta en una recompensa más positiva. En este entorno, cada acción que no conduce a la meta se penaliza con -1 (es por esta razón que en los primeros episodios el agente obtiene una recompenza de -200 y se termina el episodio), lo que hace que el agente busque minimizar el número de acciones necesarias para llegar a la meta y maximizar su recompensa, por lo tanto resulta en una recompenza máxima de -92, es decir resulta ser que el agente logra llegar a la méta después de 92 acciones tomadas.

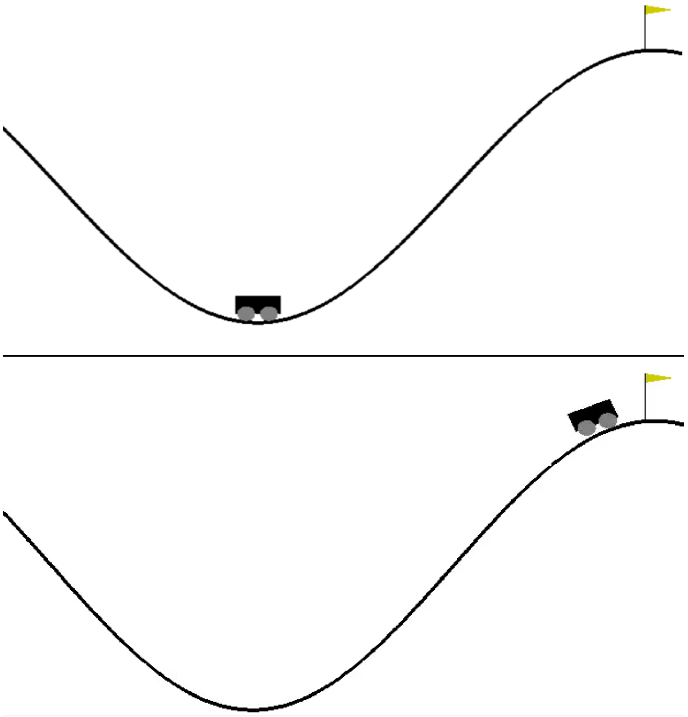


Figura 4.1: Agente jugando, Entorno: MontainCar-v0.

Es evidente que el agente aprende de manera efectiva y autónoma, ya que se observa un progresivo aumento en la recompensa obtenida en cada episodio, así como una reducción en el número de episodios necesarios para alcanzar el objetivo. Este patrón de mejora continua sugiere que el agente está aprendiendo de manera significativa a medida que acumula experiencia en el entorno del juego. Este comportamiento resalta la capacidad del algoritmo para adaptarse y mejorar su rendimiento con el tiempo.

4.1.2. Evaluación de la SLP

Con la ayuda de la implementación de la red neuronal, el agente fue probado en un entorno más complejo denominado `CartPole-v0`. En este entorno, la dimensión del espacio de estados es dos, las cuales son; derecha (0) e izquierda (1) y el de acciones que incluye la posición y la velocidad del carro, así como el ángulo y la velocidad angular de la palanca.

El objetivo en `CartPole-v0` es mantener la palanca en posición ver-

tical durante el mayor tiempo posible moviéndose hacia la izquierda y hacia la derecha sobre el carro. Se asigna una recompensa de $+1$ por cada paso dado, incluyendo el paso de terminación, y el umbral de recompensas se fija en 200.

Tras ejecutar el proyecto, la red neuronal comienza a tomar decisiones y a maximizar la función de calidad, lo que permite al agente interactuar con el entorno. Después de un entrenamiento de 100000 episodios, con un promedio de 25 iteraciones en cada episodio, se obtuvo una recompensa media de $22{,}205$ y una mejor recompensa de 131 puntos durante todo el entrenamiento. Además, la duración total del entrenamiento fue de $39{,}48$ minutos.

Se observa que la red neuronal contribuye significativamente a la toma de decisiones más efectivas, ya que maximiza la función Q de manera más óptima, permitiendo al agente interactuar de manera más eficiente con el entorno.

Figura 4.2: Entorno: CarPole-v0

4.1.3. Evaluación de la Memoria Cíclica

Una vez implementada la memoria de repetición, también conocida como memoria cíclica, permite al agente almacenar una gran cantidad de experiencias pasadas, lo que facilita el aprendizaje a partir de muestras aleatorias y la mejora del rendimiento general del agente.

El método de aprendizaje del agente, `learn_from_batch_experience`, utiliza la memoria de repetición para seleccionar una muestra aleatoria de experiencias pasadas. Esta muestra se utiliza para actualizar los pesos de la red neuronal, lo que permite al agente aprender de manera más eficiente y generalizar su conocimiento sobre el entorno.

Al emplear la estructura de memoria y ejecutar el entorno `CartPole` para el aprendizaje del agente durante 100000 episodios, se comparan los resultados del entrenamiento obtenidos anteriormente en el mismo entorno, tanto con el uso de la memoria como sin ella. Los valores obtenidos son los siguientes:

Entrenamiento	**Iteraciones**	**Recompensa Media**	**Mejor Recompensa**	**Tiempo**
Sin memoria	25	22.205	131	39.48
Con memoria	37	22.238	162	38.25

Cuadro 4.1: Comparación de resultados al utilizar la memoria.

La tabla anterior muestra que al utilizar la memoria de repetición, el agente aumenta el número promedio de iteraciones por episodio de 25 a 37. Esto significa que el agente realiza más acciones por episodio antes perder o de que el juego termine, lo que le permite avanzar más en el juego en cada ejecución.

Además, la mejor recompensa obtenida durante todo el entrenamiento fue de 162, lo que indica que el agente toma decisiones más acertadas, lo que a su vez le permite obtener mayores recompensas. Como resultado, la recompensa media por episodio aumenta a $22{,}238$.

Por último, la utilización de la memoria de repetición aborda el problema de que las observaciones no sean `IID` (independientes e idénticamente distribuidas), lo que conduce a una convergencia más rápida de los resultados de la red neuronal. Esto se traduce en un tiempo de convergencia más rápido durante el entrenamiento, que se redujo a $38{,}25$ unidades de tiempo, es decir, $1{,}23$ minutos más rápido que cuando no se utiliza la memoria. Por lo tanto, se puede apreciar un cambio significativo en el aprendizaje y la convergencia de los resultados del agente bajo las

mismas condiciones del entorno, lo que refleja una optimización evidente con la utilización conjunta de la SLP y la memoria de repetición.

4.2. Rendimiento de la CNN

En esta sección se presentan los resultados del uso de la red convolucional (CNN) por parte del Agente QL. Se plantea una comparación entre el entrenamiento y el gameplay del agente en dos entornos diferentes de los juegos de Atari, los cuales serán analizados de forma individual y en conjunto para así evaluar el aprendizaje del agente y cómo esto puede variar dependiendo del entorno en el que se encuentre.

Para ello, se utilizó la herramienta TensorBoard de la librería TensorFlow, con la cual se extrajeron los datos relevantes de cada episodio para el análisis del aprendizaje del agente. Entre estos datos se incluyen la recompensa media (`mean-ep-reward`) y la recompensa máxima (`max-ep-reward`).

Además, se realizará un análisis comparativo en entornos simples y complejos, evaluando el rendimiento del agente cuando se modifican los hiperparámetros. Esto permitirá entender mejor cómo la complejidad del entorno y la configuración de los hiperparámetros afectan el proceso de aprendizaje del agente.

Se espera que este análisis proporcione una visión clara de cómo la CNN mejora el desempeño del agente QL, permitiéndole procesar información visual y aprender de manera más eficiente en entornos variados.

4.2.1. Evaluación en entornos simples de la Atari

Como entorno simple se eligió el videojuego `Asteroids`, ya que presentan escenarios con mayor libertad de movimiento y una variedad de enemigos.

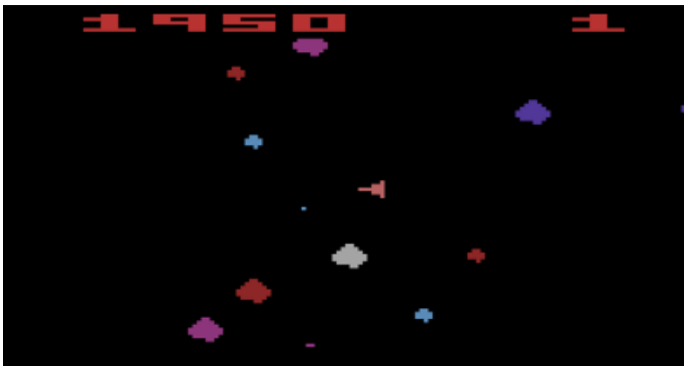


Figura 4.3: Entorno: Asteroids-v0

Analicemos los resultados del Entrenamiento y Gameplay del Agente QL en este videojuego:

Entrenamiento:

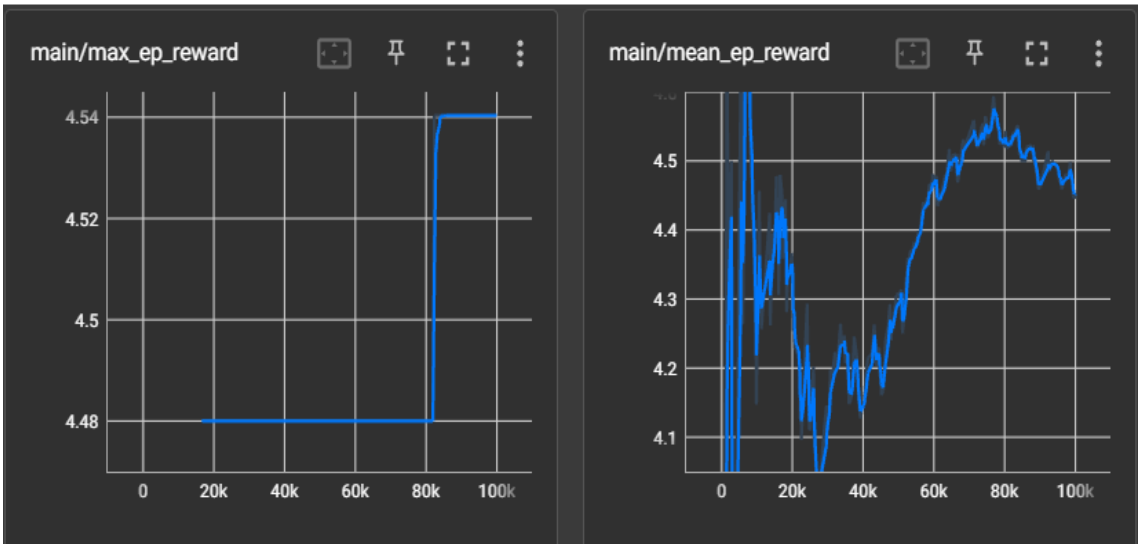


Figura 4.4: Resultados del Entrenamiento en Asteroids-v0

De la Figura 4.4, se puede observar que en la gráfica de la recompensa máxima por episodio, existe un incremento continuo en el valor a medida que el agente explora el videojuego. Este fenómeno confirma que el agente está aprendiendo, ya que muestra una mejora notable en las recompensas alcanzadas durante el entrenamiento, el cual consistió en 100,000 episodios. Este aumento en la recompensa máxima indica que el agente está descubriendo y optimizando estrategias más efectivas a lo largo del tiempo.

Además, la gráfica de la media de las recompensas muestra una tendencia creciente que se estabiliza conforme avanzan los episodios. Este comportamiento sugiere que el agente está mejorando su desempeño a medida que adquiere más experiencia. La estabilización de la media de las recompensas indica que el agente ha alcanzado una fase de aprendizaje donde su rendimiento se ha vuelto consistente, reflejando una adaptación exitosa a la dinámica del entorno. Este patrón es un buen indicativo de que el proceso de entrenamiento está funcionando de manera efectiva, y el agente está desarrollando habilidades robustas para maximizar las recompensas en el videojuego.

Comparacion: Entrenamiento y GamePlay

De la Figura 4.5, se puede observar en el gráfico de la recompensa máxima por episodio un incremento en los valores del gameplay (en celeste) en comparación con los valores del entrenamiento (en azul). Esto sugiere que el agente ha aprendido de manera efectiva cómo interactuar con el videojuego y ha logrado maximizar sus recompensas durante la fase de explotación del entorno. El aumento en las recompensas máximas durante el gameplay indica que el agente ha adquirido un buen entendimiento de las estrategias óptimas y las ha implementado exitosamente en situaciones reales de juego.

Por otro lado, las gráficas de las recompensas medias por episodio muestran que, con el tiempo, estas recompensas tienden a acercarse entre sí. Este acercamiento es indicativo de que el agente ha desarrollado una política eficiente. En otras palabras, el agente ha logrado un aprendizaje óptimo a lo largo del videojuego, logrando un rendimiento consistente tanto durante el entrenamiento como durante el gameplay. La convergencia de las recompensas medias refleja una capacidad mejorada del agente para generalizar y aplicar eficazmente lo aprendido en el entorno, resultando en un desempeño estable y fiable en diferentes episodios.

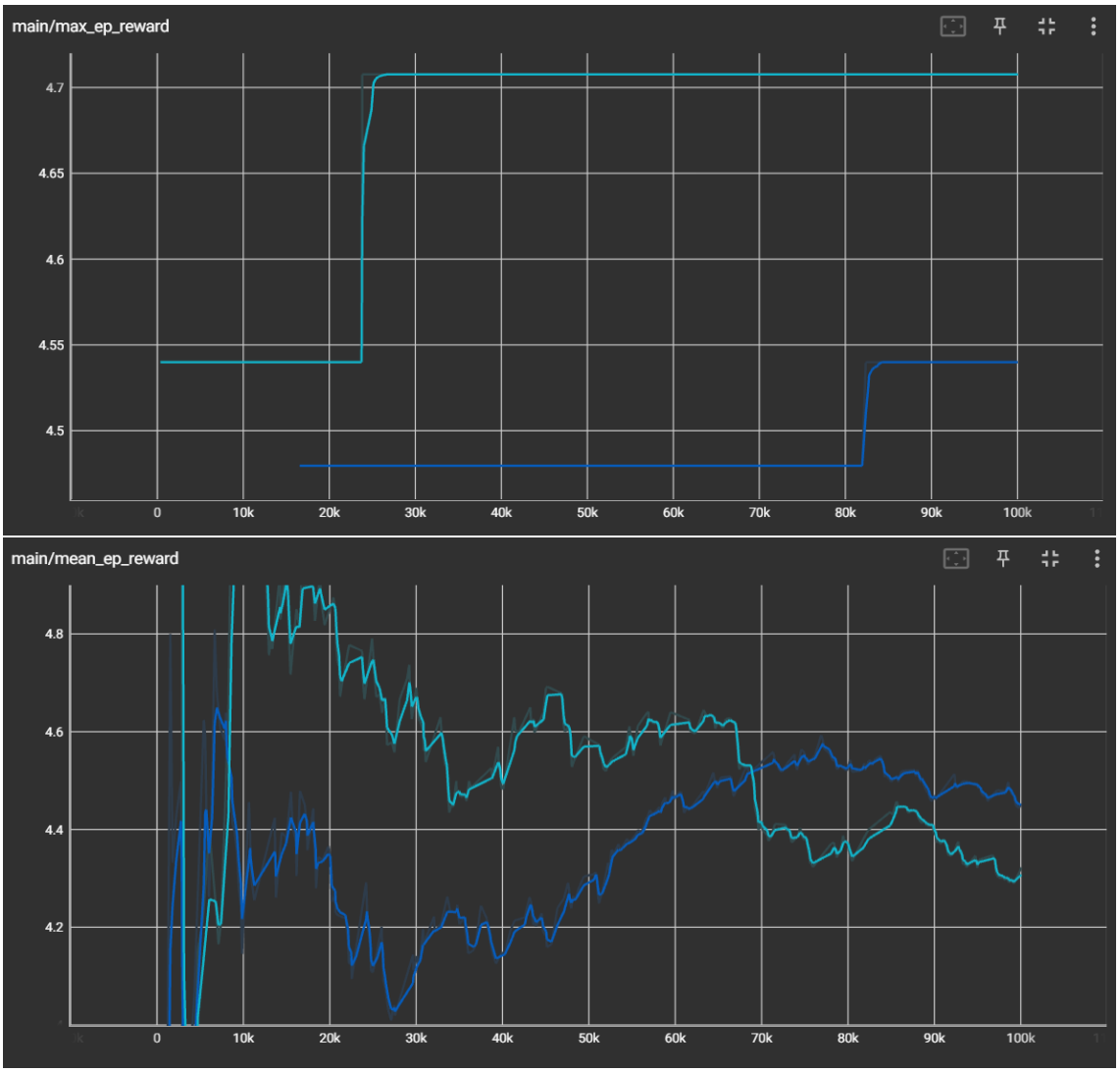


Figura 4.5: Resultados: Entrenamiento y GamePlay en Asteroids-v0

Modificación del Hiperparámetro GAMMA

Durante el entrenamiento (Figura 4.6), se puede observar que, al reducir el valor de gamma en el videojuego Asteroids, la gráfica de la recompensa máxima por episodio muestra una falta de aprendizaje significativo. En la gráfica (en verde), la recompensa máxima se mantiene relativamente constante a lo largo del entrenamiento, lo que sugiere que el agente no está explorando ni explotando eficazmente el entorno. En contraste, durante el primer entrenamiento con un valor de gamma más alto (en azul), se aprecia un incremento notable en la recompensa máxima, lo que indica un aprendizaje más efectivo.

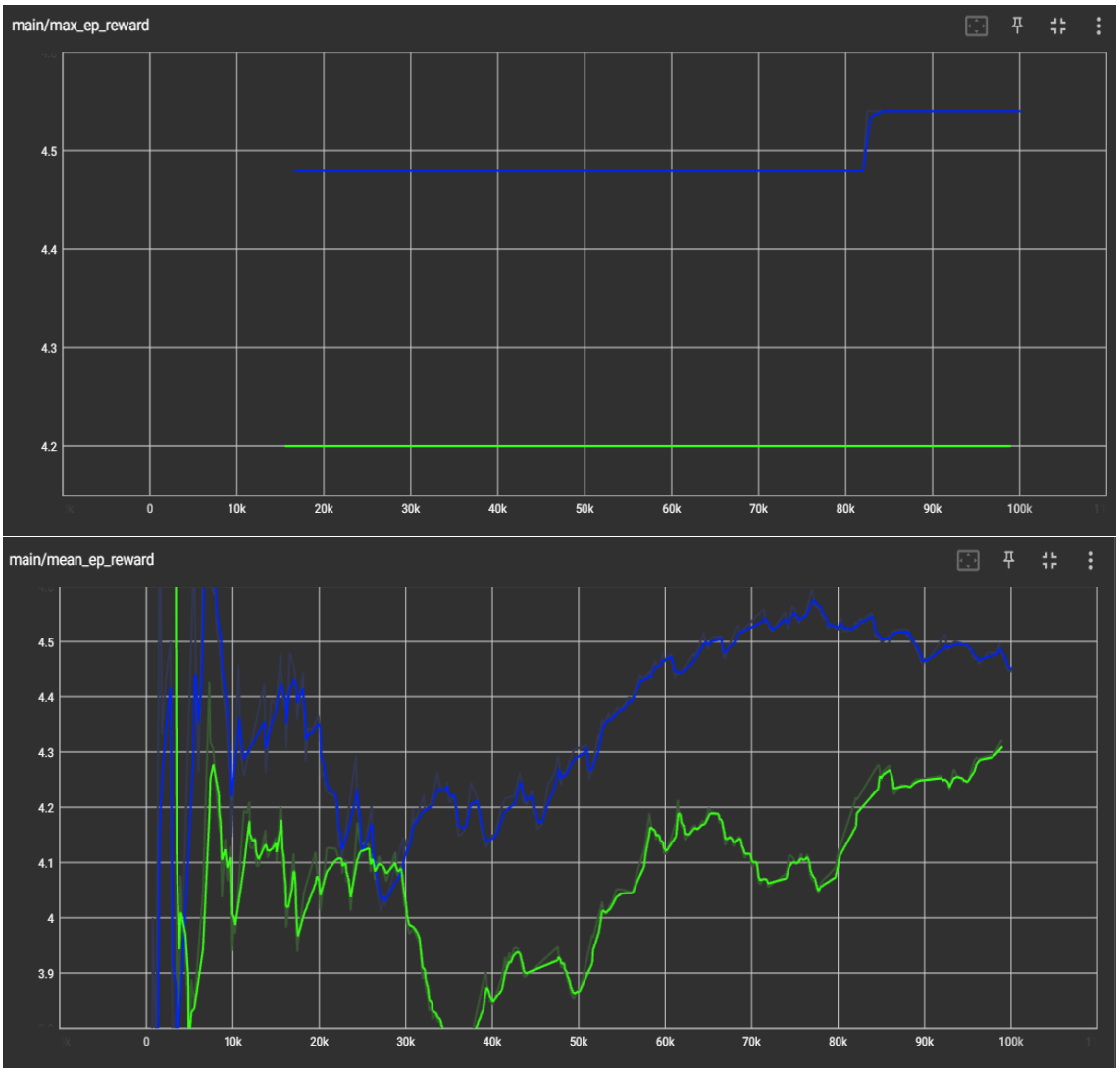


Figura 4.6: Entrenamiento: Modificación de Hiperparámetros en Asteroids-v0

El valor de gamma en el algoritmo Q-Learning es crucial para determinar el equilibrio entre la recompensa inmediata y la recompensa futura. Un valor bajo de gamma hace que el agente se enfoque más en las recompensas inmediatas y menos en las recompensas a largo plazo. Esto limita la capacidad del agente para planificar a largo plazo y aprovechar las oportunidades que podrían requerir un mayor número de pasos para ser recompensadas.

Por otro lado, la gráfica de las recompensas medias por episodio muestra una tendencia creciente y estable sin saltos bruscos, lo que refleja una convergencia progresiva del rendimiento del agente. Aunque la re-

compensa media está aumentando, su tasa de mejora es más lenta en comparación con el entrenamiento con un gamma más alto. Esto sugiere que, aunque el agente está explorando el entorno de manera constante, necesitará más episodios de entrenamiento para maximizar las recompensas.

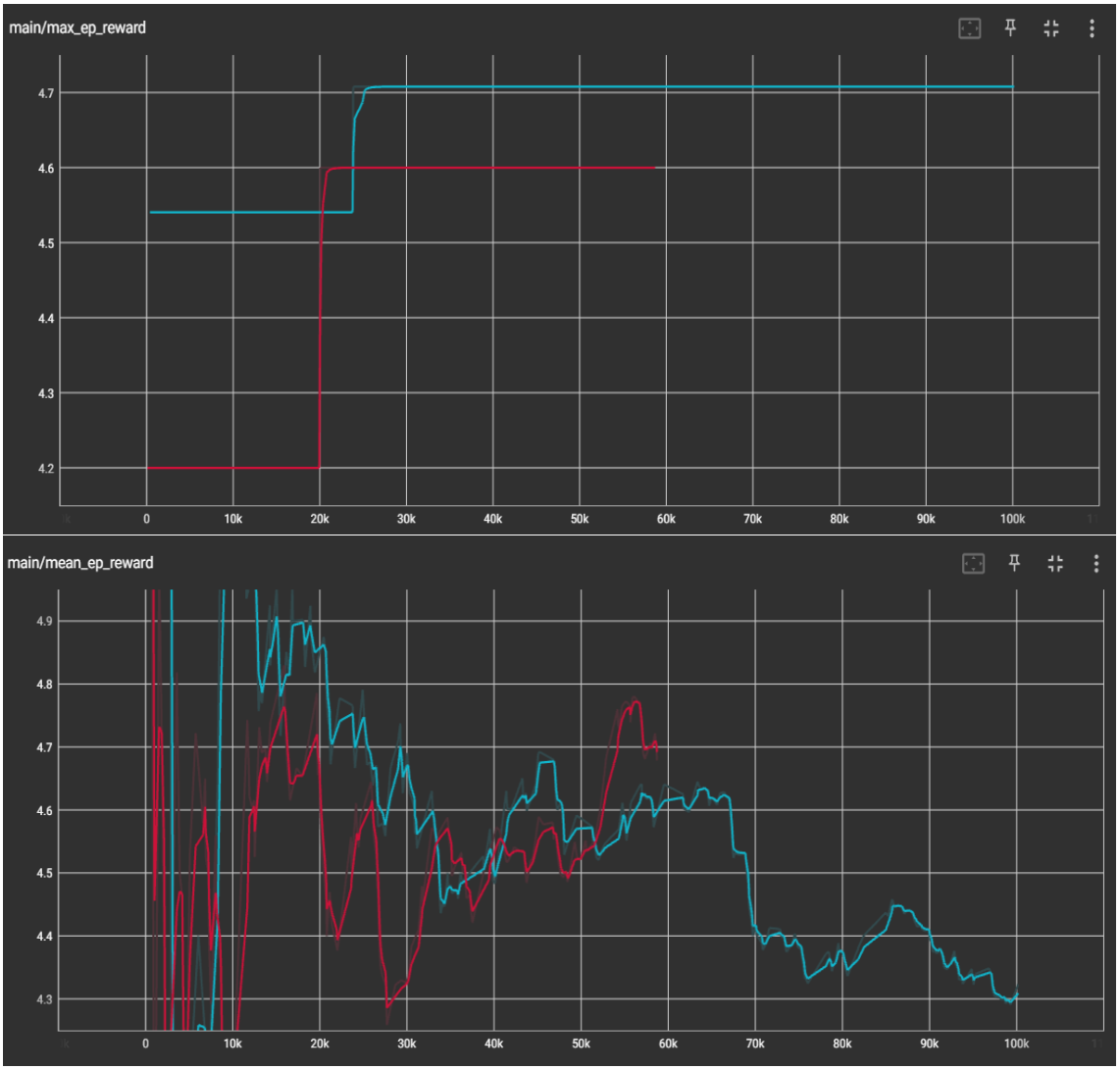


Figura 4.7: GamePlay: Modificación de Hiperparámetros en Asteroids-v0

Durante el Gameplay (Figura 4.7), se puede observar que el agente explota mejor el entorno debido a la disminución del parámetro gamma, ya que se nota un incremento en el valor máximo de la recompensa (en rojo). Sin embargo, este valor máximo no alcanza los resultados de la tendencia mostrada en azul, lo que indica que el agente no ha aprendido lo suficiente para maximizar sus recompensas de manera más rápida.

Por otro lado, al analizar la gráfica de las recompensas medias por episodio, se observa una tendencia creciente que se estabiliza a lo largo del tiempo. Esto sugiere que el agente está aprendiendo, pero el proceso es más lento. La estabilización gradual de las recompensas medias indica que el agente está alcanzando un equilibrio en su desempeño, pero este proceso es más prolongado debido a la reducción del parámetro gamma. La falta de alteración en el número de episodios también contribuye a esta lentitud en la mejora.

4.2.2. Evaluación en entornos complejos de la Atari

Como entorno complejo se eligió el videojuego `Ms. PacMan`, ya que combina la necesidad de estrategia para evitar fantasmas con niveles que cambian y comportamientos de enemigos más variados, lo que requiere habilidades de adaptación y planificación constante.

Figura 4.8: Entorno: Ms.PacMan-v0

Analicemos los resultados del Entrenamiento y Gameplay del Agente QL en este videojuego:

Entrenamiento:

De la Figura 4.9, se puede observar que la gráfica de la recompensa máxima por episodio muestra incrementos significativos a lo largo del entrenamiento, con tres valores máximos prominentes alcanzados durante la exploración del agente. Esto sugiere que el agente está explorando el entorno de manera adecuada y que su capacidad para aprender y mejorar está aumentando.

Por otro lado, la gráfica de las recompensas medias muestra una tendencia creciente y una estabilidad durante todo el entrenamiento. Esto indica que, en general, el rendimiento del agente está mejorando de manera consistente y estable conforme avanza en el juego de Ms. Pac-Man.

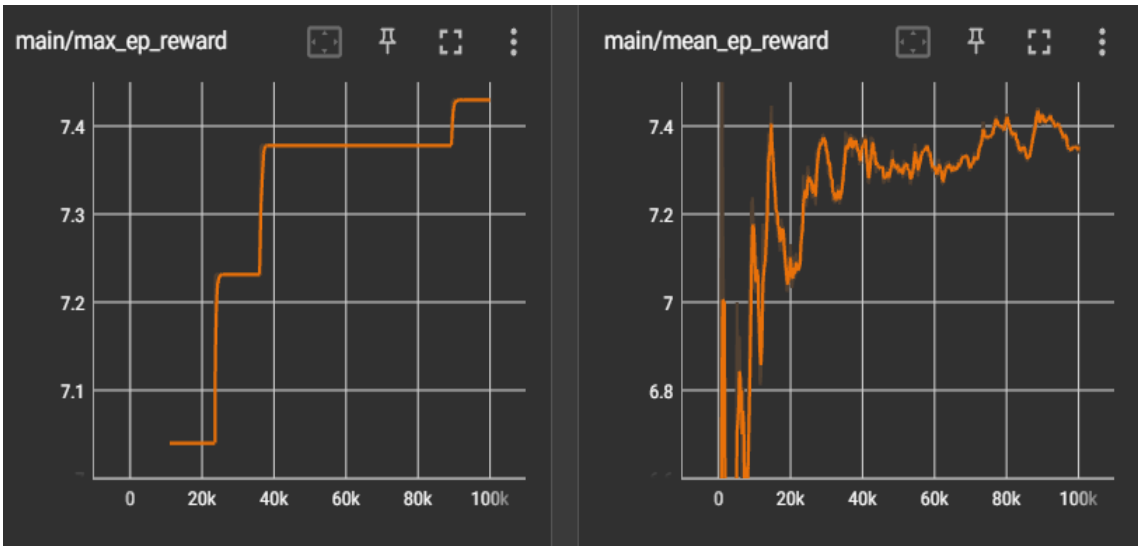


Figura 4.9: Resultados del Entrenamiento en Ms.PacMan-v0

Comparacion: Entrenamiento y GamePlay

De la Figura 4.10, se puede notar en la gráfica de la recompensa máxima por episodio que el gameplay (en morado) mejora significativamente a partir de la experiencia del entrenamiento, haciendo que la recompensa máxima aumente constantemente con el tiempo. Esto reafirma que el agente explora y explota el entorno de manera eficiente, maximizando sus recompensas.

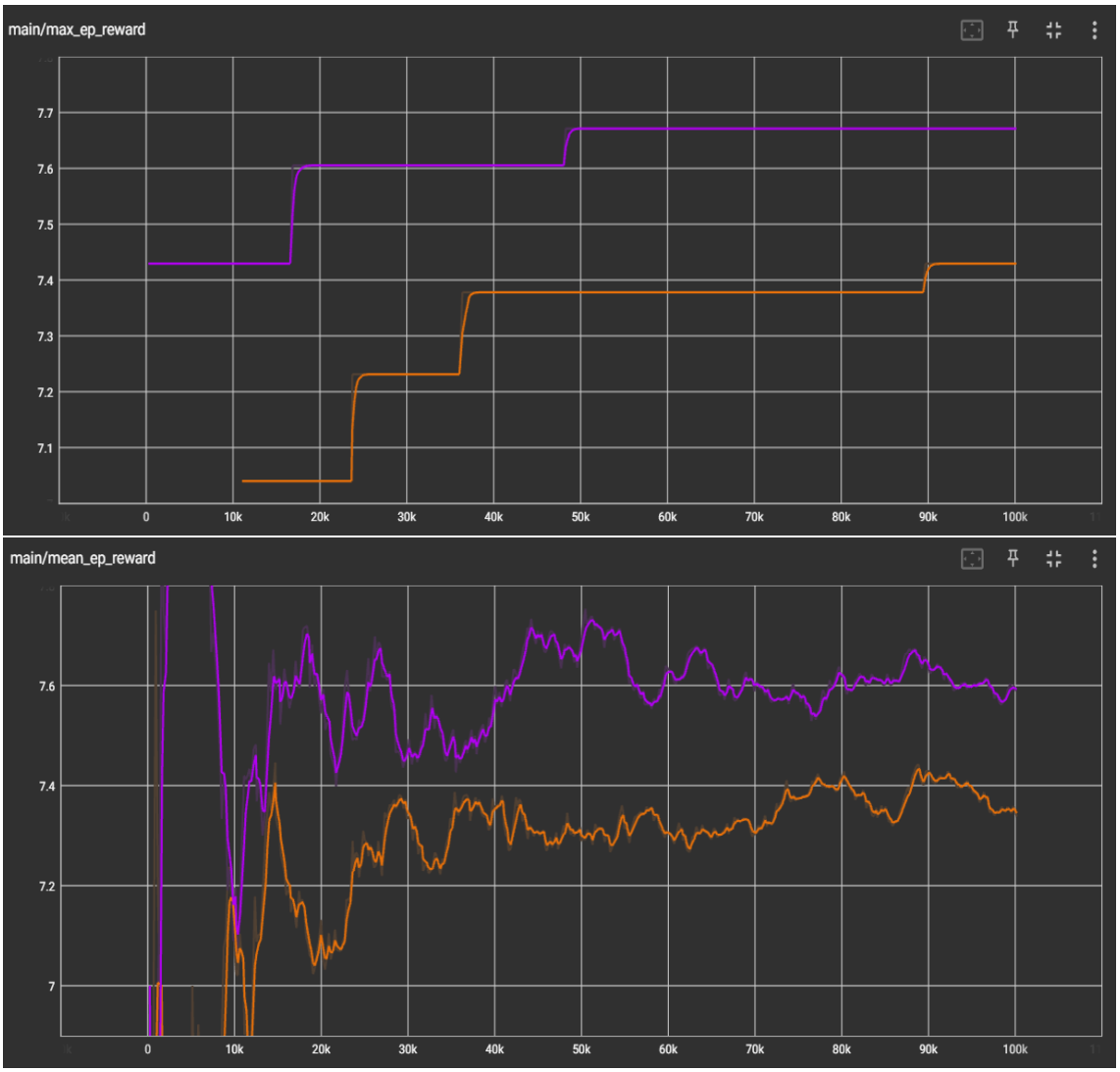


Figura 4.10: Resultados: Entrenamiento y GamePlay en Ms.PacMan-v0

Además, en las recompensas medias, se observa que son mayores que las del entrenamiento en todo momento y que se están estabilizando. Esto confirma que el agente aprende de manera rápida y constante a pesar de estar en un entorno complejo como es Ms. Pac-Man.

Modificación del Hiperparámetro `GAMMA`

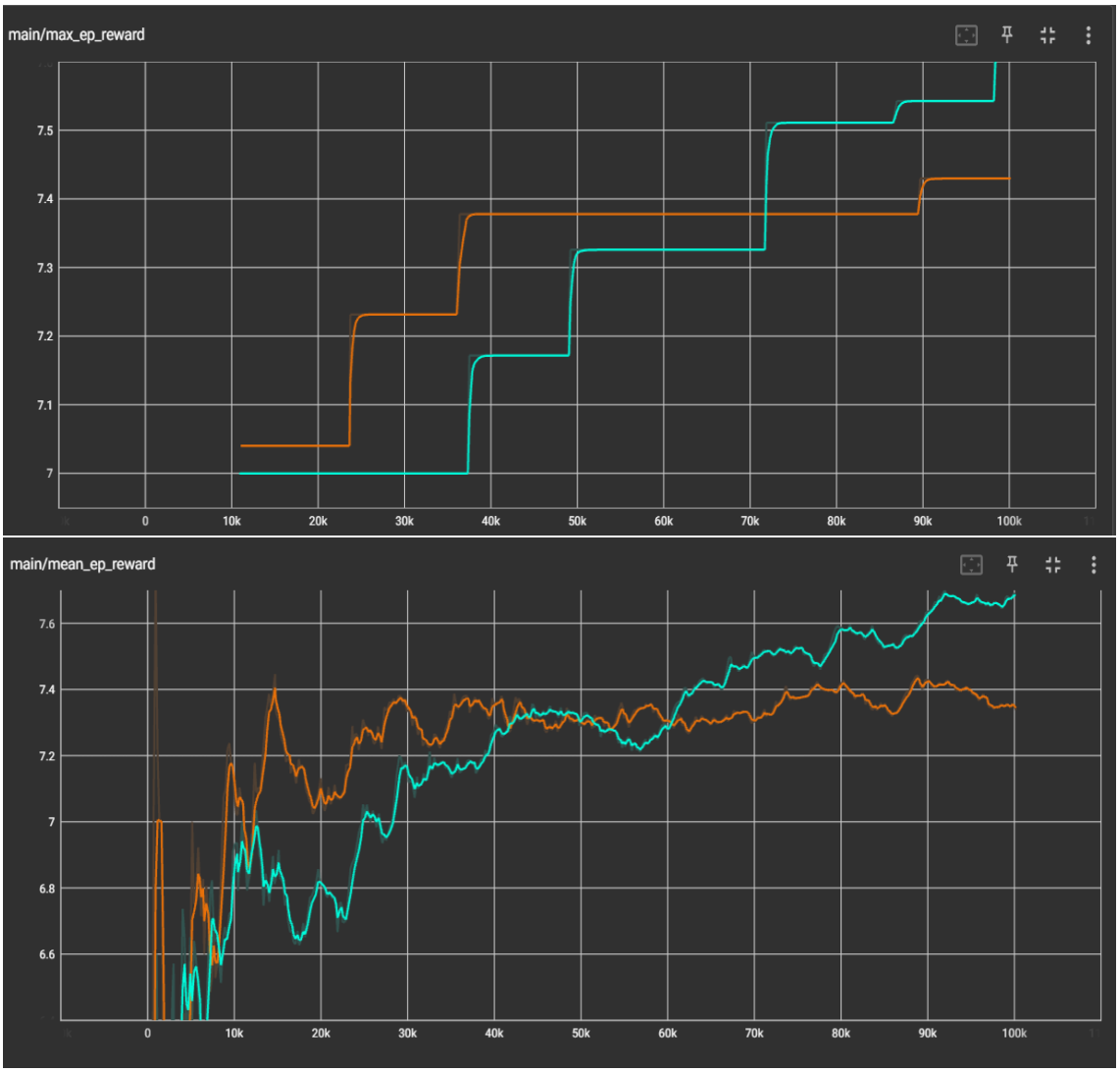


Figura 4.11: Entrenamiento: Modificación de Hiperparámetros en Ms.PacMan-v0

Durante el entrenamiento (Figura 4.11), observamos que la disminución del valor de gamma ayudó a que el agente maximice su recompensa. Esto se refleja en la gráfica en turquesa, que sobrepasa el valor máximo de la recompensa del agente entrenado con el gamma inicial (en naranja). Esto puede deberse a que, al disminuir gamma, el agente penaliza menos las acciones futuras, valorando más las recompensas presentes. Este ajuste es adecuado para el juego de Ms. Pac-Man, donde las recompensas inmediatas son cruciales para evitar a los fantasmas y recolectar puntos.

Por otro lado, la recompensa media por episodio muestra una tendencia creciente que supera la del entrenamiento inicial (en naranja). Esto

indica que el agente no solo mejora su desempeño máximo, sino que también logra un rendimiento más consistente y estable a lo largo del tiempo. Este comportamiento sugiere que el agente aprende de manera efectiva a adaptarse a la complejidad del entorno en Ms. Pac-Man, mejorando su capacidad para tomar decisiones óptimas en situaciones variadas.

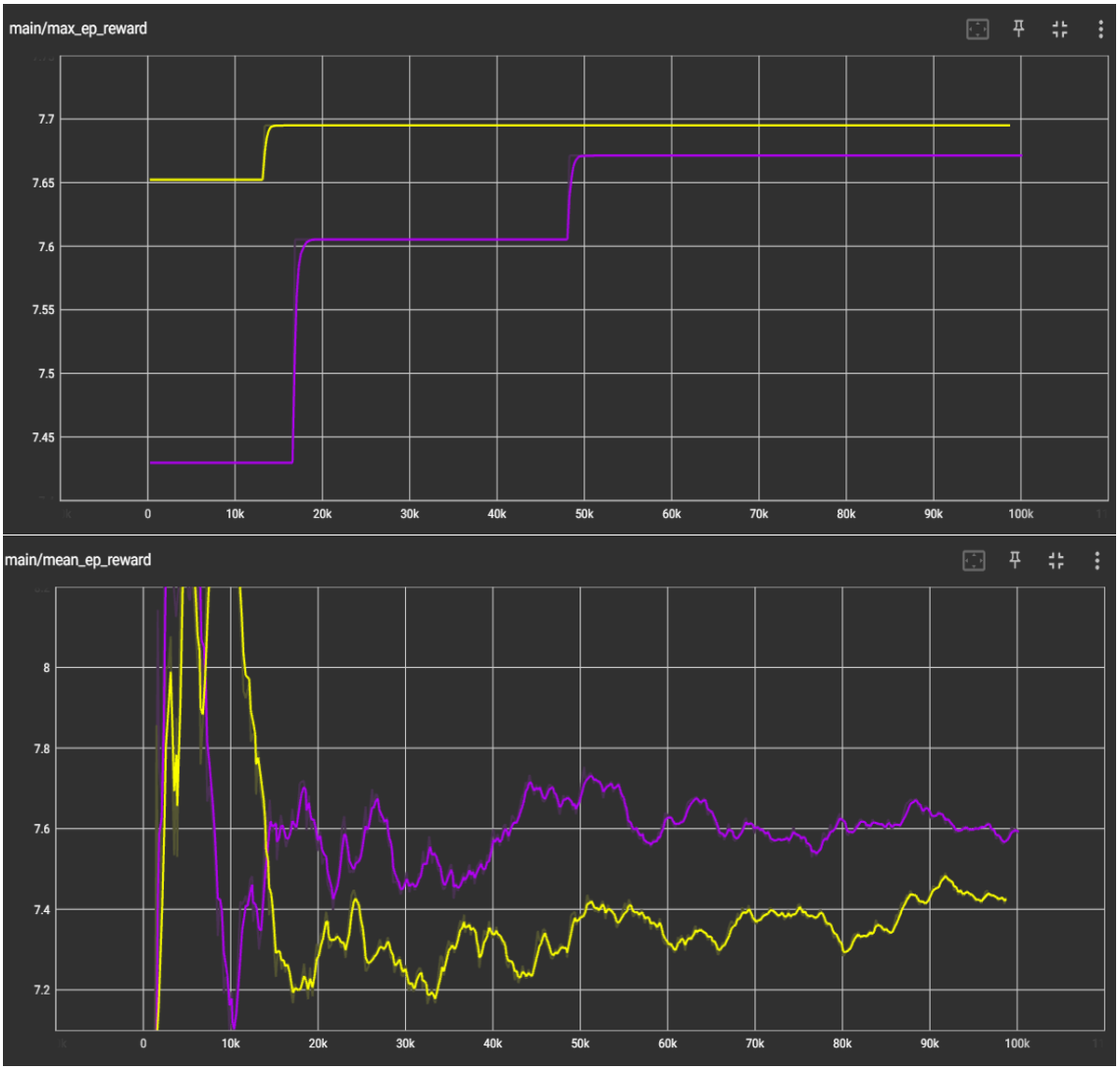


Figura 4.12: GamePlay: Modificación de Hiperparámetros en Ms.PacMan-v0

Durante el gameplay (Figura 4.12), se puede observar que la gráfica de la recompensa máxima por episodio, con el hiperparámetro gamma disminuido (en amarillo), muestra un incremento inicial debido a la explotación del videojuego basada en la experiencia del entrenamiento. Sin embargo, aunque hay un incremento en la recompensa, esta se estabiliza,

indicando que el agente aparentemente ya no está aprendiendo significativamente de la explotación del videojuego. Esto también se refleja en la gráfica de las recompensas medias, ya que aunque tienen una tendencia creciente, son menores que las recompensas medias obtenidas en el gameplay inicial (en morado). Esto puede deberse a que el entorno de Ms. Pac-Man es altamente complejo y el agente puede estar sobrevalorando las recompensas inmediatas sin considerar adecuadamente las estrategias a largo plazo.

Así, se puede afirmar que el agente explora y explota los videojuegos de manera correcta y óptima, maximizando sus recompensas y mostrando una estabilización de las recompensas medias. Esta tendencia se mantiene independientemente de la complejidad del entorno, lo que demuestra que el agente es consistente en su aprendizaje. En entornos simples, el agente logra rápidamente una alta eficiencia, mientras que en entornos complejos, como Ms. Pac-Man, aunque el proceso de aprendizaje puede ser más gradual, el agente sigue mejorando y adaptándose de manera efectiva. La disminución del gamma en Ms. Pac-Man ayudó a maximizar la recompensa porque el agente valoraba más las recompensas inmediatas, cruciales para evitar fantasmas y recolectar puntos en este juego. Sin embargo, en Asteroids, esta disminución no fue beneficiosa ya que el juego requiere una planificación a largo plazo para esquivar asteroides y enemigos, y un gamma menor hizo que el agente sobrevalorara las recompensas inmediatas, perjudicando su aprendizaje.

Capítulo 5

Conclusiones y recomendaciones

5.1. Conclusiones

En definitiva, la investigación realizada demuestra cómo los matemáticos pueden contribuir al desarrollo de inteligencias artificiales avanzadas, integrando técnicas de optimización y aprendizaje profundo para mejorar la interacción y toma de decisiones en entornos gráficos. Al utilizar un entorno gráfico como los videojuegos, el proyecto no solo se ha vuelto más atractivo, sino que también ha permitido ilustrar de manera efectiva la complejidad y el potencial de las técnicas de Q-Learning. Es importante resaltar que los agentes Q-Learning no son una tecnología simple o trivial, sino que representan la vanguardia en la investigación y desarrollo de inteligencia artificial en la actualidad.

A través de la implementación de la ecuación de Bellman, se ha evaluado el rendimiento inicial del agente en entornos simples como `MountainCar-v0` y `CartPole-v0`, mostrando su capacidad de aprendizaje y adaptación.

Posteriormente, la incorporación de redes neuronales convolucionales (CNN) y la memoria de repetición ha permitido al agente mejorar significativamente su desempeño, logrando una comprensión más profunda y

detallada del entorno. Esto se ha evidenciado en los entornos complejos de Atari, como `Asteroids` y `Ms. PacMan`, donde el agente ha aprendido a maximizar sus recompensas de manera más eficiente. El análisis comparativo de los hiperparámetros, como la modificación del valor de gamma, ha resaltado la importancia de ajustar estos parámetros para optimizar el aprendizaje del agente.

Finalmente, el proceso de ver cómo un agente mejora y aprende a través de la interacción con los videojuegos puede ser extremadamente satisfactorio y estimulante. Así, el presente trabajo también subraya la relevancia de utilizar recursos interactivos y atractivos, como los videojuegos, en la educación STEM, promoviendo un aprendizaje más dinámico y efectivo. La integración de elementos visuales y entretenidos en el proceso de enseñanza no solo hace que el aprendizaje sea más comprensible, sino que también demuestra el poder de las tecnologías avanzadas en la resolución de problemas complejos y en la creación de experiencias educativas innovadoras.

5.2. Recomendaciones

1. Definir versiones compatibles de todas las librerías a utilizar.
2. Realizar una optimización más exhaustiva de los hiperparámetros del agente Q-Learning para mejorar su rendimiento en diferentes entornos de juego. Esto incluye ajustar parámetros como la tasa de aprendizaje, el factor de descuento (gamma), y la tasa de exploración-explotación (epsilon).
3. Para futuros trabajos, se recomienda ampliar el modelo del mundo en el que el agente Q-Learning opera. Esta ampliación puede incluir la incorporación de más estados y acciones, así como la simulación de escenarios más complejos y realistas.
4. Desarrollar una interfaz gráfica de usuario (GUI) que permita una interacción más intuitiva con el agente Q-Learning y facilite la configuración de experimentos y la visualización de resultados.

5. Realizar estudios empíricos con grupos de estudiantes para evaluar la efectividad del proyecto en el aprendizaje de conceptos STEM, midiendo factores como la motivación, la retención del conocimiento y la habilidad para resolver problemas.

Capítulo A

Anexos

En esta sección se presentan resultados relevantes que no se colocó en el cuerpo principal del documento debido a su extensión, pero que de sempeñan un papel fundamental en la formulación de conclusiones sól idas. Estos resultados, que abarcan diversas facetas del estudio, han contribuido de manera significativa a la obtención de conclusiones claras.

Para aquellos interesados en explorar más allá, se proporciona un enlace al repositorio del proyecto en GitHub: https://github.com/YokoMolina/Q-learning-videojuego. Aquí, se encuentra el código necesario para replicar integralmente el proyecto y obtener los resultados presentados en este trabajo. Esta fuente adicional de información permite una verificación transparente y una ex ploración en profundidad de los procesos subyacentes.

A.1. Algoritmo RL básico

```
#SE GENERA UN BUCLE DE 2000 ITERACIONES
for episode in range(MAX_NUM_EPISODE):
    # despertemos al agente
    obs = enviroment.reset()
    for step in range(MAX_STEPS_PER_EPISODE):
    #CREAMOS EL RENDER DE EL AMBIENTE
        enviroment.render() # persove el entorno el agente
        action = enviroment.action_space.sample()
        next_state, reward, done, info=enviroment.step(action)
        obs = next_state # tiene que volver a percibir el ambiente

        if done is True:
            print("\n Episodio #{} terminado en {} steps.".format(episode, step+1))
            break
enviroment.close()
```

A.2. Clase QLearner

```
class QLearner (object):
    def __init__(self, enviroment):
        #inicialicemos los valores del objeto self
        self.obs_shape = enviroment.observation_space.shape
        self.obs_high = enviroment.observation_space.high
        self.obs_low = enviroment.observation_space.low
        self.obs_bins = NUM_DICRETE_BINS
        self.obs_width = (self.obs_high-self.obs_low)/self.obs_bins

        self.action_shape = enviroment.action_space.n
        self.Q = np.zeros((self.obs_bins+1,self.obs_bins+1, self.action_shape))
        # amtriz de 31x31x3
        self.alpha = ALPHA
        self.gamma = GAMMA
        self.epsilon = 1.0 #valor que se va incrementando si sobrepasa el epsiolon_min

    def discretize(self,obs):
        # discreticemos el espacio de observaciones
        return tuple(((obs-self.obs_low)/self.obs_width).astype(int))
    # astyope me ayuda a escoger el piso inferir de la division para saber en que division

    def get_action(self, obs):
    # politica de fuerza bruta para minimizar el epsilon min
    # mejor accion con menor porbabilidad 1-epsion que es el vañor de equivocarse
        discrete_obs = self.discretize(obs)
        # seleccion de la acion en vase a Epsilon-greedy
        if self.epsilon > EPSILON_MIN:
            self.epsilon -= EPSILON_DECAY
        if np.random.random() > self.epsilon: # se elige un num random entre 0y1
            #con proba 1-epsilon elegimos la mejor posible
            return np.argmax(self.Q[discrete_obs])
        else:
            return np.random.choice([a for a in range(self.action_shape)])
```

A.3. Entrenamiento QLearner

```
def train(agent, enviroment):
        best_reward = -float("inf")
        for episode in range(MAX_NUM_EPISODE):
            done = False
            obs = enviroment.reset()
            total_reward = 0.0
            while not done:
                action = agent.get_action(obs) # accion elegida segun la ecuacion q-learning
                next_obs, reward, done, info= enviroment.step(action)
                agent.learn(obs,action,reward, next_obs)
                obs= next_obs
                total_reward += reward
            if total_reward > best_reward:
                best_reward = total_reward
            print("Episodio número {} con recompensa: {}, mejor recompensa: {}, epsilon: {}".format(episode, total_reward, best_reward, agent.epsilo
    # de todas las politicas de entrenamiento que emos obtenido
            # devolvemos la mejor de todas
        return np.argmax(agent.Q, axis=2)
```

A.4. Aprendizaje

```
def train(agent, enviroment):
        best_reward = -float("inf")
        for episode in range(MAX_NUM_EPISODE):
            done = False
            obs = enviroment.reset()
            total_reward = 0.0
            while not done:
                action = agent.get_action(obs) # accion elegida segun la ecuacion q-learning
                next_obs, reward, done, info= enviroment.step(action)
                agent.learn(obs,action,reward, next_obs)
                obs= next_obs
                total_reward += reward
            if total_reward > best_reward:
                best_reward = total_reward
            print("Episodio número {} con recompensa: {}, mejor recompensa: {}, epsilon: {}".format(episode, total_reward, best_reward, agent.epsilo
    # de todas las politicas de entrenamiento que emos obtenido
            # devolvemos la mejor de todas
        return np.argmax(agent.Q, axis=2)
```

A.5. Perceptron

```python
class SLP(torch.nn.Module):
    def __init__(self,input_shape,output_shape, device = torch.device("cpu")):
        # input_shape: tamaño o forma de los datos de entrada
        #output_shape: "" de los datos de salida
        # device: el dispositivo (cpu o cuda) que la
        # SLP debe utilizar para almacenar los inputs
        # a cada iteracion # cpu o memoria
        # asi xq se esta heredando
        super(SLP, self).__init__()
        self.device = device
        self.input_shape = input_shape[0]
        self.hidden_shape = 40 # 40 unidades en la capa

        # se hace la combinacion lienal para pasar a las nueronas
        self.linear1 = torch.nn.Linear(self.input_shape, self.hidden_shape)
        self.out = torch.nn.Linear(self.hidden_shape, output_shape)

    def forward(self, x) : # voy a activar las neuronas
        x = torch.from_numpy(x).float().to(self.device)
        x = torch.nn.functional.relu(self.linear1(x))# funcion de activacion RELU max{0,x}
        x = self.out(x)
        return(x)
```

A.6. Clase SwallowQLearner

```python
class SwallowQLearner (object):
    def __init__(self, env, learning_rate = 0.005, gamma = 0.98):
        #inicialicemos los valores del objeto self
        self.learning_rate = learning_rate # podemos guardar el dato
        self.obs_shape = env.observation_space.shape

        self.action_shape = env.action_space.n

        low_act = -np.inf
        high_act = np.inf
        dtypes = np.float32

        self.action_space = Box(low=low_act, high=high_act, shape=(self.action_shape,), dtype=dtypes)

        # Define el tipo de dato para el espacio de observación
        low_obs = -np.inf
        high_obs = np.inf
        self.observation_space = Box(low=low_obs, high=high_obs, shape=self.obs_shape, dtype=dtypes)

        self.Q = SLP(self.obs_shape, self.action_shape)
        # se le pasa el espacio de observaciones
        # el valor de salida es le espacio de acciones pues es lo que el agente desea saber para saber qu


        # se va a optimizar las ponderaciones de las entradas a travez de Adam
        self.Q_optimazer = torch.optim.Adam(self.Q.parameters(), lr = learning_rate) # delvuelve los para
        # lr es el ratio de aprendizaje que es un numero pequeño q es lo q aprende de una iteracion

        self.gamma = gamma

        self.epsilon_max = 1.0 #valor que se va incrementando si sobrepasa el epsiolon_min
        self.epsilon_mim = 0.05
        # funcion que ayuda a ir decayendo el epsilon
        total_steps = MAX_NUM_EPISODE*STEPS_PER_EPISODE
        self.epsilon_decay = LinearDecaySchedule(initial_value = self.epsilon_max,
                                                 final_value = self.epsilon_mim,
```

A.7. Memoria de Repetición

```python
Experience = namedtuple("Experience", ["obs","action","reward","next_obs", "done"])
# generamos la estructura que tendrá todos estos parametros [ ]
# ESTO ES UN BUFFER
class ExperienceMemory(object):
    # vamos a reproducir las experiencias , es decir, recuperarlas
    # ESTO SERÁ UN BUFFER QUE SIMULA LA MEMORIA DEL AGENTE
    def __init__(self, capacity = int(1e7)):
        # :parame capacity es la capacidad total de mempria ciclica elimina memoria inicial que no sirve
        # numero max de experiencias almacenables
        self.capacity = capacity
        self.memory_idx = 0 # identificador que sabe la experiencia actual #es el ultimo indice vacio

        self.memory = []

    def sample(self, batch_size):
        #batch:size es el tamaño de la memoria a recuperar
        # devuelve una muestra aleatoria del tamñano bacht_size de experiencias aleatorias  de la memoria
        assert batch_size <= self.get_size(), "el tamaño de la muestra es superior a la memoria disponible"
        return random.sample(self.memory, batch_size) # extraigo de forma uniforme muestras del experience de form

    def get_size(self):
        #return: numero de experiencias almacenadas en memoria
        return len(self.memory) # tamaño de la memoria

    def store(self, exp):
        #exp: objeto experiencia a ser almanecado en memoria
        self.memory.insert(self.memory_idx % self.capacity, exp)
        # insertamos en la experiencia exp en memory_idx y el modulo % nos va a ayudar para que sea ciclica
        # el guardar informacion es decir este modulo dará cero en varias veces lo cual actualizará el valor inici
        self.memory_idx += 1
```

A.8. Entrenamiento SLP con Memoria de Repetición

```
def learn_from_batch_experience(self, experiences):
    """
    Actualiza la red neuronal profunda en base a lo aprendido en el conjunto de experiencias ante
    :param experiences: fframento de recuerdos anteriores
    :return:
    """
    batch_xp = Experience(*zip(*experiences))
    obs_batch = np.array(batch_xp.obs)/255.0 # entre 0 y 1 para q sea mas sensillo de aprender
    action_batch = np.array(batch_xp.action)
    reward_batch = np.array(batch_xp.reward)
    if self.params["clip_rewards"]:
        reward_batch = np.sign(reward_batch) # me devuelve el signo

    next_obs_batch = np.array(batch_xp.next_obs)/255.0
    done_batch = np.array(batch_xp.done)

    if self.params['use_target_network']:
        if self.step_num % self.params['target_network_update_frequency'] == 0:
            self.Q_target.load_state_dict(self.Q.state_dict())
        td_target = reward_batch + ~done_batch *\
                    np.tile(self.gamma, len(next_obs_batch)) * \
                    torch.max(self.Q_target(next_obs_batch),1)[0].data.tolist()
        td_target = torch.from_numpy(td_target)

    else:
        td_target = reward_batch + ~done_batch * \
                    np.tile(self.gamma, len(next_obs_batch)) * \
                    torch.max(self.Q(next_obs_batch).detach(),1)[0].data.tolist()
        td_target = torch.from_numpy(td_target)
```

A.9. Aprendizaje SLP con Memoria de Repetición

```
if __name__ == "__main__":
    # ahora vamos
    env_conf = manager.get_environment_params()
    env_conf["env_name"] = args.env
    #environment = gym.make("LunarLander-v2")
    #environment = gym.make("CartPole-v0")
    #environment = gym.make("MountainCar-v0")
   ## environment = gym.make("BipedalWalker-v2")
    #environment = gym.make("Acrobot-v1")
    if args.test:
        env_conf["episodic_life"] = False # ayuda a reportar la media
        # en vez de hacer de lor bida
        # se imprimen para cada vida de la partida
    reward_type = "LIFE" if env_conf["episodic_life"] else "GAME"

    custom_region_available = False
    for key, value in env_conf["useful_region"].items():
        if key in args.env:
            env_conf["useful_region"] = value
            custom_region_available = True
            break
    if custom_region_available is not True: # si no hay region personalizada
        env_conf["useful_region"] = env_conf["useful_region"]["Default"]
        #por defecto
        print("Configuración a utilizar", env_conf)

    atari_env = False
    for game in Atari.get_games_list():
        if game.replace("_","") in args.env.lower(): # si hay en la lista el q el usuario puso
            atari_env = True
    if atari_env:
        environment = Atari.make_env(args.env, env_conf)
        monitor_path = "./monitor_output"
        environment = gym.wrappers.Monitor(environment, monitor_path, force = True)
```

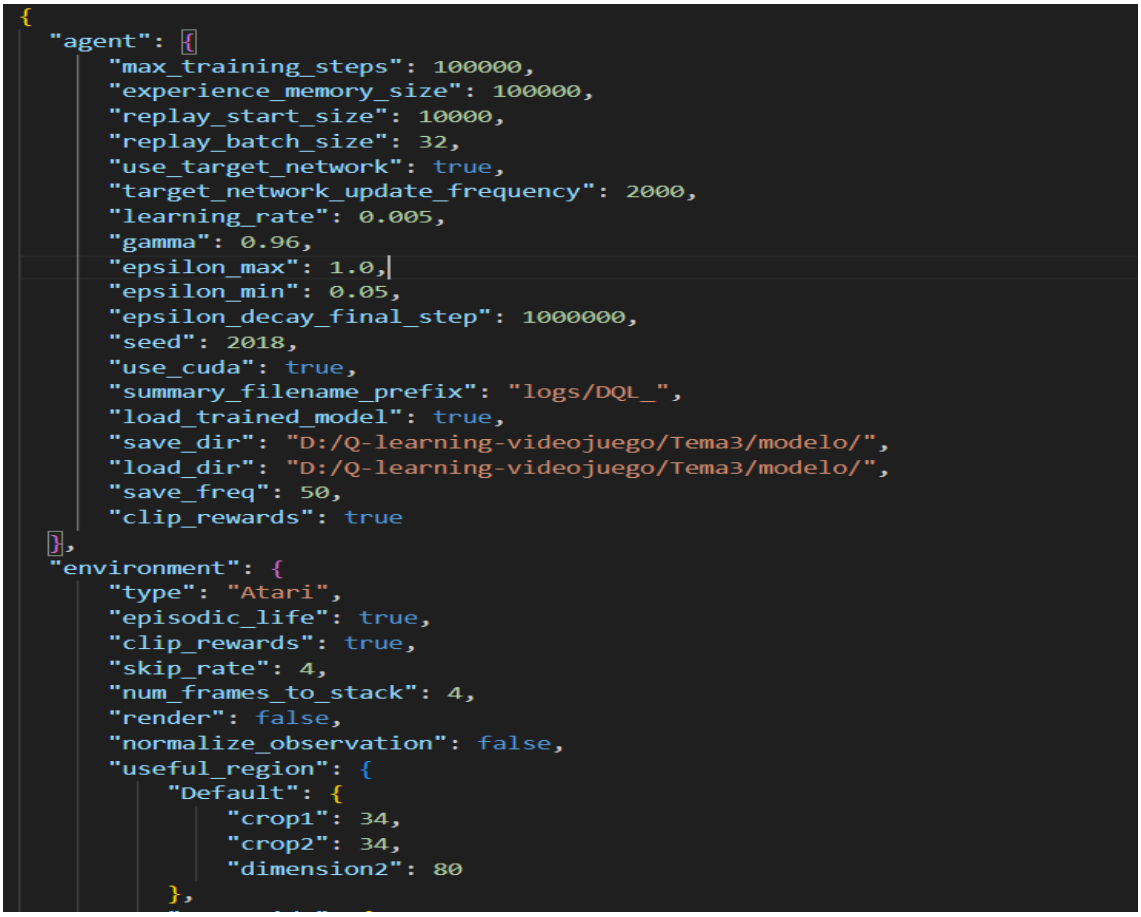
A.10. Aumento de la Capacidad de Aprendizaje

```python
class LinearDecaySchedule(object):
    def __init__(self, initial_value, final_value, max_steps):
        assert initial_value > final_value, "el valor inicial debe ser estrictamente mayor que el valor final"
        # si se da la condicion sigue el algoritmo
        self.initial_value = initial_value
        self.final_value = final_value
        self.decay_factor = (initial_value-final_value)/max_steps

    # se llamará cuando se llame la funcion epsilon decay
    def __call__(self, step_num):
        current_value = self.initial_value - step_num * self.decay_factor
        if current_value < self.final_value:
            current_value = self.final_value
        return current_value

if __name__ == "__main__": # el script por terminal
    #import matplotlib.pyplot as plt
    epsilon_initial = 1.0
    epsilon_final = 0.005
    MAX_NUM_EPISODE = 100000
    STEPS_PER_EPISODE = 300
    total_steps = MAX_NUM_EPISODE * STEPS_PER_EPISODE
    linear_schedule = LinearDecaySchedule(initial_value = epsilon_initial,
                                          final_value = epsilon_final,
                                          max_steps = 0.5 * total_steps)
    epsilons = [linear_schedule(step) for step in range(total_steps)]
    #print("epsilon =", epsilons)
    #plt.plot(epsilons)
    #plt.show()
```

A.11. Control de Parámetros



```
{
  "agent": {
      "max_training_steps": 100000,
      "experience_memory_size": 100000,
      "replay_start_size": 10000,
      "replay_batch_size": 32,
      "use_target_network": true,
      "target_network_update_frequency": 2000,
      "learning_rate": 0.005,
      "gamma": 0.96,
      "epsilon_max": 1.0,
      "epsilon_min": 0.05,
      "epsilon_decay_final_step": 1000000,
      "seed": 2018,
      "use_cuda": true,
      "summary_filename_prefix": "logs/DQL_",
      "load_trained_model": true,
      "save_dir": "D:/Q-learning-videojuego/Tema3/modelo/",
      "load_dir": "D:/Q-learning-videojuego/Tema3/modelo/",
      "save_freq": 50,
      "clip_rewards": true
  },
  "environment": {
      "type": "Atari",
      "episodic_life": true,
      "clip_rewards": true,
      "skip_rate": 4,
      "num_frames_to_stack": 4,
      "render": false,
      "normalize_observation": false,
      "useful_region": {
          "Default": {
              "crop1": 34,
              "crop2": 34,
              "dimension2": 80
          },
```

A.12. Red CNN

```python
class CNN(torch.nn.Module):
    # una red neuronal convolucional que tomará decisiones según,
    # los píxeles de la imagen

    def __init__(self, input_shape, output_shape, device = "cpu"):
        # input:shape: es la dimension de la imagen que supondremos
        # viene rescalada a Cx84x84
        # output shape: dimension de la salida
        # device dispositivo (cpu o gpu) donde la cnn debe
        # almacenar los valores de cada iteracion

        # input_shape 84x84 supuesto
        super(CNN, self).__init__()
        self.device = device
        # vamos a ir filtrando la informacon con los kernels
        # conv2d /conales de entrada =1 en este caso
        # 64 / filtros / 64 caracteristicas para aprender de tamaño 4
        self.layer1 = torch.nn.Sequential(
            torch.nn.Conv2d(input_shape[0], 64, kernel_size = 4, stride = 2, padding = 1),
            torch.nn.ReLU()

        )
        # tamaño de salida = ((tamaño entrada+2*padin-tamaño del kernel)/stride)+1=42 sale una estructura matricial 64*42*42
        self.layer2 = torch.nn.Sequential(
            torch.nn.Conv2d(64, 32, kernel_size = 4, stride =2, padding = 0),
            torch.nn.ReLU()
        )
        # 32 caracteristicas para la siguiente capa, entra la profuncidad del mapa de caracteristicas
        self.layer3 = torch.nn.Sequential(
            torch.nn.Conv2d(32, 32, kernel_size = 3, stride = 1, padding = 0),
            torch.nn.ReLU()
        )# flatening llegamos a una estructura 32x18x18
        self.out = torch.nn.Linear(18*18*32, output_shape)
```

A.13. Reescalamiento de Entornos de la Atari

```python
class AtariRescale(gym.ObservationWrapper): # clase para rescalar las imagenes
    # reducimos la ram que necesitamos, son 2 experiencias en la memoria xq es la atual y
    # los bits se duplican a almacenar

    def __init__(self, env, env_conf):
        gym.ObservationWrapper.__init__(self, env)
        self.observation_space = Box(0, 255, [1, 84, 84], dtype=np.uint8)
        self.conf = env_conf

    def observation(self, observation):
        return process_frames_84(observation, self.conf)



class NormalizedEnv(gym.ObservationWrapper): # vamos a normalizar los datos para que me q
    def __init__(self, env = None):
        gym.ObservationWrapper.__init__(self, env)
        self.mean = 0
        self.std = 0
        self.alpha = 0.999
        self.num_steps = 0

    def observation(self, observation):
        self.num_steps += 1
        # voy a ponderar los valores para que se de mas peso a lo valores recientes
        self.mean = self.mean * self.alpha + observation.mean() * (1-self.alpha)
        self.std = self.std *self.alpha + observation.std() * (1-self.alpha)
        # aqui ya los hago insesgados cosntruyendo un estadistico insesgado desde el sesg
        unblased_mean = self.mean / (1-pow(self.alpha, self.num_steps))
        unblased_std = self.std /(1-pow(self.alpha, self.num_steps))
        return (observation - unblased_mean)/(unblased_std + 1e-8)
```

A.14. Clase DeepQLearner

```python
class DeepQLearner (object): # ya nos le vamos a pasar el environment
    def __init__(self, obs_shape, action_shape, params):

        self.device = device
        self.params = params
        self.gamma = self.params["gamma"]
        self.learning_rate = self.params["learning_rate"]
        self.best_reward_mean = -float("inf")
        self.best_reward = -float("inf")
        self.best_mean_reward = -float("inf")  # Añadido: Inicialización de best_mean_reward
        self.training_steps_completed = 0
        self.action_shape = action_shape

        # vamos a decidir si el objeto de entrada es una imagen o no y utilizar el perceptron o la cnn ya creada
        if len(obs_shape) == 1: # solo tenemos una dimension del espacio de observaciones
            self.DQN = SLP # metodo utilizado
        elif len(obs_shape) == 3: # el estado de observaciones es una imagen 3d
            self.DQN = CNN

        self.Q = self.DQN(obs_shape, action_shape, device).to(device)
        self.Q_optimazer = torch.optim.Adam(self.Q.parameters(), lr = self.learning_rate) # delvuelve los parametr

        if self.params["use_target_network"]: # vamos a actualizar a los ultimos valores
            self.Q_target = self.DQN(obs_shape, action_shape, device).to(device)

        self.policy = self.epsilon_greedy_Q
        self.epsilon_max = self.params["epsilon_max"]
        self.epsilon_min = self.params["epsilon_min"]

        self.epsilon_decay = LinearDecaySchedule(initial_value = self.epsilon_max,
                                                 final_value = self.epsilon_min,
                                                 max_steps = self.params["epsilon_decay_final_step"])
        self.step_num = 0 # num de operacion
        # politica de actuacion
        self.polity = self.epsilon_greedy_Q
```

A.15. Aprendizaje DeepQLearner

```
if __name__ == "__main__":
    # ahora vamos
    env_conf = manager.get_environment_params()
    env_conf["env_name"] = args.env
    #environment = gym.make("LunarLander-v2")
    #environment = gym.make("CartPole-v0")
    #environment = gym.make("MountainCar-v0")
   ## environment = gym.make("BipedalWalker-v2")
    #environment = gym.make("Acrobot-v1")
    if args.test:
        env_conf["episodic_life"] = False # ayuda a reportar la media
        # en vez de hacer de lor bida
        # se imprimen para cada vida de la partida
    reward_type = "LIFE" if env_conf["episodic_life"] else "GAME"

    custom_region_available = False
    for key, value in env_conf["useful_region"].items():
        if key in args.env:
            env_conf["useful_region"] = value
            custom_region_available = True
            break
    if custom_region_available is not True: # si no hay region personalizada
        env_conf["useful_region"] = env_conf["useful_region"]["Default"]
        #por defecto
        print("Configuración a utilizar", env_conf)

    atari_env = False
    for game in Atari.get_games_list():
        if game.replace("_","") in args.env.lower(): # si hay en la lista el q el usuario puso
            atari_env = True
    if atari_env:
        environment = Atari.make_env(args.env, env_conf)
        monitor_path = "./monitor_output"
        environment = gym.wrappers.Monitor(environment, monitor_path, force = True)
```

Referencias bibliográficas

[1] Review of deep learning: concepts, cnn architectures, challenges, 2021. URL: https://journalofbigdata.springeropen.com/articles/10.1186/s40537-021-00444-8.

[2] World models, 2024. URL: https://worldmodels.github.io/.

[3] Youtube video, 2024. URL: https://www.youtube.com/watch?v=4mkudbx-1qI.

[4] KeyTrends AI. Backpropagation, 2024. URL: https://keytrends.ai/es/academy/glosario/inteligencia-artificial/backpropagation.

[5] Wolfram Barfuss, Jonathan F. Donges, and Jürgen Kurths. Deterministic limit of temporal difference reinforcement learning for stochastic games. 2019. URL: https://www.nature.com/articles/s41598-019-56406-5.

[6] Richard Ernest Bellman. The theory of dynamic programming, Oct 2008. URL: https://www.rand.org/content/dam/rand/pubs/papers/2008/P550.pdf.

[7] Diego Calvo. Función de activación en redes neuronales, 2018. URL: https://www.diegocalvo.es/funcion-de-activacion-redes-neuronales/.

[8] cdr book. Fundamentos de ciencia de datos con r, 2024. URL: https://cdr-book.github.io/cap-redes-convol.html.

[9] Interactive Chaos. Función de coste para clasificación, 2015. URL: https://interactivechaos.com/es/manual/tutorial-de-deep-learning/funcion-de-coste-para-clasificacion.

[10] Top Big Data. Diferencia entre retropropagación y descenso de gradiente estocástico, 2021. URL: https://topbigdata.es/diferencia-entre-retropropagacion-y-descenso-de-gradiente-estocast

[11] Kristopher De Asis, Alan Chan, Silviu Pitis, Richard S. Sutton, and Daniel Graves. Fixed-horizon temporal difference methods for stable reinforcement learning. 2019. URL: https://arxiv.org/abs/1907.04402.

[12] Jaime Durán. Todo lo que necesitas saber sobre el descenso del gradiente aplicado a redes neuronales, 2019. URL: https://medium.com/metadatos/todo-lo-que-necesitas-saber-sobre-el-descenso-del-gradiente-aplica

[13] Jaime Durán. Todo lo que necesitas saber sobre el descenso del gradiente aplicado a redes neuronales, 2019. URL: https://medium.com/metadatos/todo-lo-que-necesitas-saber-sobre-el-descenso-del-gradiente-aplica

[14] Prompt Engineer. Cálculo del error en la red neuronal: función de pérdida o costo, 2015. URL: https://promptengineer.es/calculo-del-error-en-la-red-neuronal-funcion-de-perdida-o-costo/.

[15] iamtrask. A neural network in 13 lines of python (part 2 - gradient descent), 2015. URL: https://iamtrask.github.io/2015/07/27/python-network-part2/.

[16] IBM. Deep learning, 2024. URL: https://www.ibm.com/es-es/topics/deep-learning.

[17] IBM. Deep learning, 2024. URL: https://www.ibm.com/es-es/topics/deep-learning.

[18] Arthur Juliani. Simple reinforcement learning with tensorflow part 0: Q-learning with tables and neural networks,

Aug 2016. URL: https://medium.com/emergent-future/simple-reinforcement-learning-with-tensorflow-part-0-.

[19] KeepCoding. ¿qué es una red neuronal en deep learning?, 2015. URL: https://keepcoding.io/blog/red-neuronal-en-deep-learning/.

[20] Volodymyr Mnih et al. Asynchronous methods for deep reinforcement learning. 2016. URL: https://spinningup.openai.com/en/latest/spinningup/keypapers.html.

[21] Netzun. Introducción al deep learning, 2024. URL: https://netzun.com/cursos-online/introduccion-deep-learning.

[22] Pfafner. Gradiente descendente estocástico, 2023. URL: https://pfafner.github.io/opt2023/proyectos/SGD.pdf.

[23] Aprendizaje Profundo. Funciones de activación, 2021. URL: https://aprendizajeprofundo.github.io/Libro-Fundamentos/Redes_Neuronales/Cuadernos/Activation_Functions.html.

[24] PyTorch. Pytorch dqn tutorial, 2024. URL: https://pytorch.org/tutorials/intermediate/reinforcement_q_learning.html.

[25] Paloma Recuero. ¿sabes en qué se diferencian las redes neuronales del deep learning?, 2015. URL: https://blogthinkbig.com/redes-neuronales-deep-learning.

[26] scatec. ¿qué es el procesamiento de imágenes?, 2024. URL: https://www.scatec.es/que-es-el-procesamiento-de-imagenes/.

[27] John Schulman et al. High-dimensional continuous control using generalized advantage estimation. 2015. URL: https://spinningup.openai.com/en/latest/spinningup/keypapers.html.

[28] John Schulman et al. Trust region policy optimization. 2015. URL: https://spinningup.openai.com/en/latest/spinningup/keypapers.html.

[29] SoldAI. Introducción al deep learning i: Funciones de activación. 2024. URL: https://medium.com/soldai/introducci%C3%B3n-al-deep-learning-i-funciones-de-activaci%C3%B3n-b3eed1411b20.

[30] Mohammad Mustafa Taye. Theoretical understanding of convolutional neural network: Concepts, architectures, applications, future directions, 2023. URL: https://www.mdpi.com/2079-3197/11/3/52.

[31] Gobe Tech. Cómo se derivan las funciones de costo para las redes neuronales, 2015. URL: https://tech.gobetech.com/18708/como-se-derivan-las-funciones-de-costo-para-las-redes-neuronales.html.

[32] UNIR. El algoritmo gradient descent para el entrenamiento de redes neuronales, 2021. URL: https://www.unir.net/ingenieria/revista/gradient-descent/.

[33] William Uther. Temporal difference learning. 1988. URL: https://www.ijcai.org/Proceedings/88-1/Papers/122.pdf.

[34] Wikipedia. Procesamiento digital de imágenes, 2024. URL: https://es.wikipedia.org/wiki/Procesamiento_digital_de_im%C3%A1genes.

[35] Wikipedia. Propagación hacia atrás, 2024. URL: https://es.wikipedia.org/wiki/Propagaci%C3%B3n_hacia_atr%C3%A1s.

[36] Yufei Zhang Xuming Han Muhammet Deveci Milan Parmar Xia Zhao, Limin Wang. A review of convolutional neural networks in computer vision, 2024. URL: https://link.springer.com/article/10.1007/s10462-024-10721-6.

[37] Seung Jae Yoo and Anne G.E. Collins. Working memory capacity predicts individual differences in reinforcement-learning rate, 2022. URL: https://ccn.studentorg.berkeley.edu/pdfs/papers/YooCollins2022JoCN_WMRL.pdf.