ESCUELA POLITÉCNICA NACIONAL

FACULTAD DE CIENCIAS

APLICACIÓN DE LAS 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

8 DE JUNIO 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

RESUMEN

(Máximo 250 palabras)

Palabras clave: palabra1, palabra2, palabra3,. . ., palabra6.

ABSTRACT

(Máximo 250 palabras)

Keywords: palabra1, palabra2, palabra3,. . ., palabra6.

Índice general

1. Descripción del componente desarrollado **1**

1.1. Descripción del Proyecto 1

1.2. Objetivo general 1

1.3. Objetivos específicos 2

1.4. Alcance 2

2. Marco teórico **4**

2.1. Conceptos Fundamentales 4

2.1.1. Agente 4

2.1.2. Entorno 4

2.1.3. Estado 4

2.1.4. Acción 6

2.1.5. Recompensa 6

2.1.6. Política 7

2.2. Rainforcement Learning 7

2.2.1. Ecuacion de Bellman 8

2.2.2. Ecuación de Bellman para procesos estocásticos . . . 10

2.3. Factor de penalización 11

2.4. Q-Learning 12

2.5. Diferencial Temporal 13

2.5.1. Tasa de aprendizaje α 13
2.6. Deep Learning 14
2.6.1. Red Neuronal 15
2.6.2. Perceptrón o Neurona 16
2.6.3. Función de activación 17
2.6.4. Función de costos 19
2.6.5. Método del Gradiente Descendiente 20
2.6.6. Método del Gardiente Descendiente Estocástico 21
2.6.7. Propagación hacia atras 22
2.6.8. Deep Q-Learning 23
2.7. Proceso de Digitalización de imágenes 25
2.7.1. Pixel 25
2.7.2. Imagen Digital 25
2.7.3. Niveles de Gris 26
2.7.4. Cuantificación de una imagen 27
2.8. Procesamiento de imágenes 28
2.8.1. Redes Neuronales Convolucionales (CNN) 28
2.8.2. Convolución 29
2.8.3. Capa Relu 31
2.8.4. Max Pooling 32
2.8.5. Flattening 34
2.8.6. Full Conection 35
2.8.7. Capa de Salida 37
2.9. Modelo del Mundo 37

3. Metodología **38**

3.1. Entorno de desarrollo y herramientas utilizadas 38
3.1.1. Lenguaje de programación 38

3.1.2. Librerias y Frameworks 39
3.1.3. Entorno de Desarrollo Integrado (IDE) 39
3.2. Agente QLearner en entornos simples 39
3.2.1. Algoritmo RL básico 39
3.2.2. Clase QLearner 41
3.2.3. Entrenamiento QLearner 42
3.2.4. Aprendizaje 44
3.3. Agente QLearner Optimizado 44
3.3.1. Perceptro 45
3.3.2. Clase SwallowQLearner 45
3.3.3. Memoria de Repetición 47
3.3.4. Entrenamiento SLP con Memoria de Repetición . . . 48
3.3.5. Aprendizaje 48

4. Resultados **49**
4.1. Rendimiento del Agente Q-Learner 49
4.1.1. Evaluación en un entorno simple 50
4.1.2. Evaluación de la SLP 51
4.1.3. Evaluación de la Memoria Cíclica 52
4.2. Rendimiento de la CNN 53

5. Conclusiones y recomendaciones **54**
5.1. Conclusiones 54
5.2. Recomendacione 54

A. Anexos **55**
A.1. Código del algoritmo RL básico 55

Bibliografía **55**

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 q- leearning del cual 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 así tomar elecciones que le permita tener un aprendizaje con cada decisión tomada en el entorno en el que se encuentra.

1.2. Objetivo general

Investigar y desarrollar un agente Q-learning que incorpore redes neuronales convolucionales para la realización de tareas en entornos gráficos, con el fin de demostrar cómo los matemáticos pueden contribuir al mundo del arte gráfico y la computación gráfica. Además, se resaltará el papel crucial de la optimización estadística proporcionada por los mate-

máticos en el desarrollo de inteligencias artificiales, especialmente en el campo del deep learning, con el objetivo de demostrar su relevancia en la mejora de la eficiencia y precisión en la resolución de problemas.

1.3. Objetivos específicos

- Desarrollar un agente q-learning que integre redes neuronales convolucionales.
- Crear un entorno virtual que permita al agente realizar diversas tareas de las cuales aprende sucesivamente.
- Recopilar datos mediante la observación y registro de las acciones y decisiones tomadas por el agente durante su proceso de aprendizaje en el entorno gráfico.
- 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

Este estudio se enfocará en el desarrollo e implementación de un agente de aprendizaje basado en el algoritmo Q-Learning, integrado con redes neuronales convolucionales, para la realización de tareas en entornos gráficos. 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.

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. Además, se analizará cómo la integración de redes neuronales convolucionales y técnicas de optimización estadística influye en la capacidad de aprendizaje y ejecución de tareas del agente, identificando mejoras significativas en términos de eficiencia y precisión.

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."(16)

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.(25) 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(4):

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.(4)

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.(18) 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.(26)

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.(25) 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.(31)

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.(9) 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.(3) 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.(14)

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.(27) 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.(19) 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. (15)

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.(13) 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.(13)

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. (17)

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.(23) 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.(5)

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.(21) 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.(7)

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.(29) 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.(12) 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. (30) 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.(10)

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.(20) 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.(8)

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.(11) 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 (33).

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. (2) 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(35). 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 (22). 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. (32)

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 de píxel en las dimensiones vertical y horizontal, respectivamente) 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 (24). 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). (32)

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

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

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` (6)

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**

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.(AnexoA.1)

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.(6) 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. (Anexo A.1)

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.(28)

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.(34) 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). (6) 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 entrenamiento 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 (28). 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. (24)

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.(6) 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.(Anexo A.1)

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 comple-

tamente 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.

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 (34). 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. (Anexo A.1)

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 (32).

- *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

Función de Softmax

Función de Entropía

2.9. Modelo del Mundo

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.1)

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.

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. (Anexo A.1)

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.

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.1)

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.1)

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.1)

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.1)

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.1)

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. (Anexo A.1)

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 nece-

sarios 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 vertical 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. (Anexo A.1)

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.

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

Capítulo 5

Conclusiones y recomendaciones

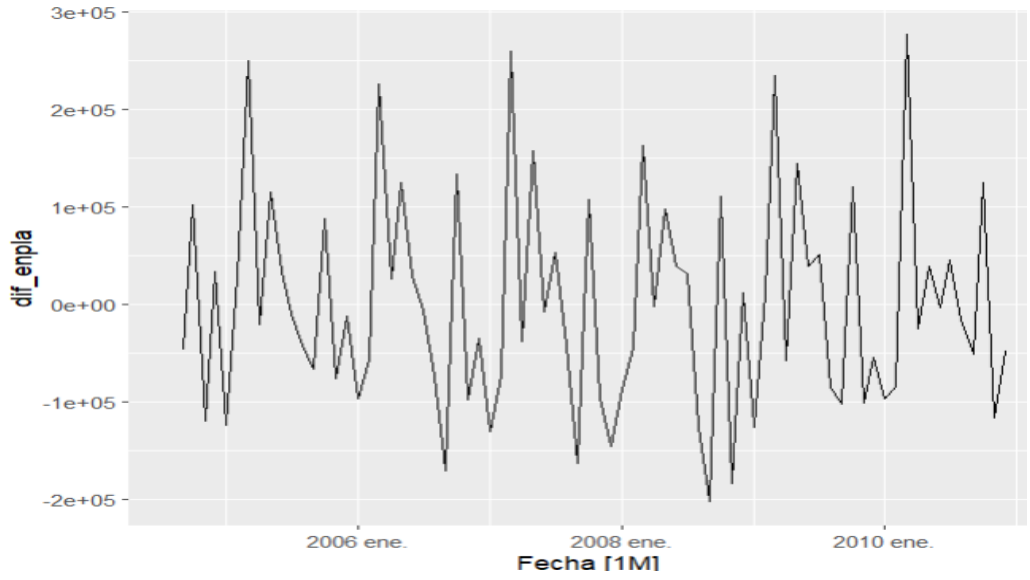
5.1. Conclusiones

5.2. Recomendacione

Capítulo A

Anexos

A.1. Código del algoritmo RL básico



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

[3] 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.

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

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

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

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

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

[9] 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.

[10] 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

[11] 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

[12] 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/.

[13] 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/.

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

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

[16] 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-.

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

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

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

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

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

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

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

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

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

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

[27] 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.

[28] 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.

[29] 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.

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

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

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

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

[34] 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.

[35] 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.