Universidad del Valle de Guatemala

Facultad de Ingeniería

Proyecto Final - Implementación y Entrenamiento de un Agente de Aprendizaje por Refuerzo para el Juego de Minesweeper

Reinforcement Learning

Jun Woo Lee Hong, 20358

Roberto Francisco Rios Morales, 20979

11 de noviembre de 2024

Descripción del problema

El objetivo de este proyecto es diseñar y entrenar un modelo de Reinforced Learning que pueda jugar al juego “Minesweeper” de manera autónoma y efectiva. Buscaminas es un juego clásico de lógica y deducción que representa un desafío significativo para los algoritmos de inteligencia artificial, debido a las características random de su dinámica de juego y la naturaleza del entorno. Donde en este juego, el jugador interactúa con un tablero que contiene celdas ocultas, algunas de las cuales esconden minas. El objetivo principal es identificar todas las celdas seguras del tablero sin activar ninguna mina.

Cada vez que el jugador revela una celda que no contiene una mina, la celda muestra un número que indica la cantidad de minas que se encuentran en las celdas adyacentes. A partir de esta información, el jugador debe deducir la ubicación de las minas, utilizando el razonamiento lógico para decidir qué celdas abrir. Por lo cual este juego es uno de un problema de deducción compleja, donde el jugador tiene que encontrar las celdas seguras.

Análisis

El problema de este proyecto es que no siempre se puede deducir perfectamente, y hay que adivinar aunque sea un modelo perfecto. Por lo que es un problema interesante para un modelo de RL ya que hay que tener buena estrategia de exploración y explotación al igual que buen sistema de recompensas.

En el juego usualmente hay 3 estados de celdas ocultas, reveladas, o con bandera. Dónde va cambiando a medida que el agente toma acciones. Donde el agente tiene acceso a la información de las celdas relevadas y con eso tiene que ir pensando cuales son seguras y cuáles no. Y el agente tendría acceso a 3 acciones, una de poner una bandera, quitar una bandera, y la de revelar una celda. Por lo que se tendría que buscar cómo implementar un algoritmo que le ayude al agente desarrollar una estrategia de decisión que le permita identificar patrones en el tablero.

Propuesta de solución

Para resolver este problema, se propone crear un ambiente que funcionaria como el juego de minesweeper donde estarían los posibles estados, acciones, y algoritmos de lógicas sobre el juego. Adicionalmente, se tendría que crear un agente que utiliza el ambiente para entrenarse para que pueda jugar el juego. Se tomará un tablero de 5x5 que resulta en 25 celdas posibles para simplificar el juego en vez del original que el modo principiante es de 8x8. Finalmente se tiene que realizar un código de entrenamiento que le indica al agente como comunicarse con el ambiente. Finalmente, para evaluar el modelo se va a realizar un algoritmo donde va a hacer

que el modelo juegue una n cantidad de juegos y mostrar su promedio porcentaje de victoria.

Descripción de la solución

Para este proyecto se realizaron dos soluciones:

1. Para desarrollar un agente que juegue Minesweeper, primero diseñamos el entorno en *"minesweeper_env.py"* Donde implementa la lógica del juego, en un tablero con 5x5. Donde además de las funciones que manejan la lógica del juego, incluye funciones necesarias que el agente necesita para poder aprender cómo:
    - reset()
        - Reinicia el juego para una nueva partida, asegurando que se generen las minas después del primer click para que nunca pueda perder en el primer movimiento.
    - step(action)
        - Como el entorno responde a una acción del agente. Manejando por ejemplo qué tipo de recompensa, o estado tiene que cambiar a después de un movimiento del agente. La forma que se calcularon las recompensas serían:
            - Seleccionar una mina = -1
            - Seleccionar una celda ya revelada = -0.5
            - Seleccionar una celda no = +0.1
    - render()
        - Esta función no se necesita para el entrenamiento, sino para visualizar a los usuarios humanos de cómo ver los estados del juego.

Luego se desarrolló el agente, con un modelo de Deep-Q Network, para aproximar el valor-Q para los pares de estado-acciones. Por lo que el modelo va sacando valores Q para todas las acciones en estados y el agente va tomando sus acciones basado en esos valores Q. Esto lo hace con las funciones de:

- forward()
    - La función que calcula los valores Q para las posibles acciones y el agente selecciona la acción con el valor Q más alto.
- act()
    - La lógica de cómo el agente selecciona una acción en base a un balance entre exploración y explotación, donde en caso de explorar, es opción al azar y si es explotada, toma la acción basado en los valores Q.

Después tenemos la implementación del agente en el entorno de minesweeper tomando en cuenta la cantidad de episodios a entrenar, y la lógica de cómo balancear la exploración vs explotación a través de un epsilon-greedy. Con las funciones de:

- train_agent
    - La función principal que configura un entorno para entrenar el agente. En bucles de cantidad de episodios, va eligiendo entre exploración o explotación usando la estrategia de epson-greedy. Y Mientras va entrenando, va actualizando el escalón para reducir la exploración y que realice más explotación a fines del entrenamiento.

Finalmente se realizó la evaluación donde se carga el modelo entrenado, y se realizan simulaciones de 100 juegos y se imprime su promedio de victoria.

2. La segunda implementación se utilizó Gymnasium de OpenAI para facilitar el entrenamiento. Donde igualmente se requería crear el entorno en *“minesweeper_env.py”* que utiliza las mismas funciones de reset y step. Estas dos funciones son obligatorias para utilizar el Gymnasium para el entrenamiento. Para este entorno, las recompensas se manejaron de la siguiente manera:
    - Seleccionar una mina = -20
    - Seleccionar una celda ya revelada = -5
    - Seleccionar una celda no previamente revelada = + 2.5
    - Ganar = +20

Para esta parte, la recompensa de revelar una celda ya revelada -5 es muy importante ya que sin esa, el agente aprende a que si revela celdas ya reveladas puede prolongar el juego infinitamente, mientras acumulando recompensas.

Luego tenemos el código del agente, que está utilizando el modelo PPO de “stable_baseliens3” con la política “MlpPolicy” lo cual hace que este sea un modelo de una red Actor-Crítico. Lo cual con el uso de Gymnasium y con esta política, se entrena el modelo.

Herramientas aplicadas

- **Torch:**

  PyTorch es un framework poderoso y flexible que en este caso fue fundamental para implementar desde cero un agente de DQN en el proyecto de Minesweeper. Con PyTorch, se construyó una red neuronal para estimar los valores Q, utilizando sus capacidades de diferenciación automática y

optimización, que simplifican el entrenamiento de estas redes de aprendizaje profundo. La compatibilidad de PyTorch con GPU también permitió acelerar significativamente el proceso de entrenamiento, esencial para realizar múltiples iteraciones de aprendizaje y mejorar el rendimiento del agente en el entorno de juego.

- **Gymnasium:**

  Gymnasium es una biblioteca desarrollada por OpenAI para facilitar la creación y el manejo de entornos de aprendizaje por refuerzo. En tu proyecto, fue clave para estructurar un entorno personalizado de Minesweeper y, en combinación con Stable-Baselines3, permitir el uso de un modelo PPO (Proximal Policy Optimization), un método Actor-Crítico. Este enfoque facilita el equilibrio entre exploración y explotación, maximizando la recompensa esperada mientras permite ajustes de política más estables y eficientes que los métodos de gradiente de política puros. Gymnasium simplificó la integración y evaluación del modelo PPO en el entorno de juego, permitiendo al agente aprender estrategias óptimas a través de la interacción continua.

Resultados

- **Implementación from scratch:**

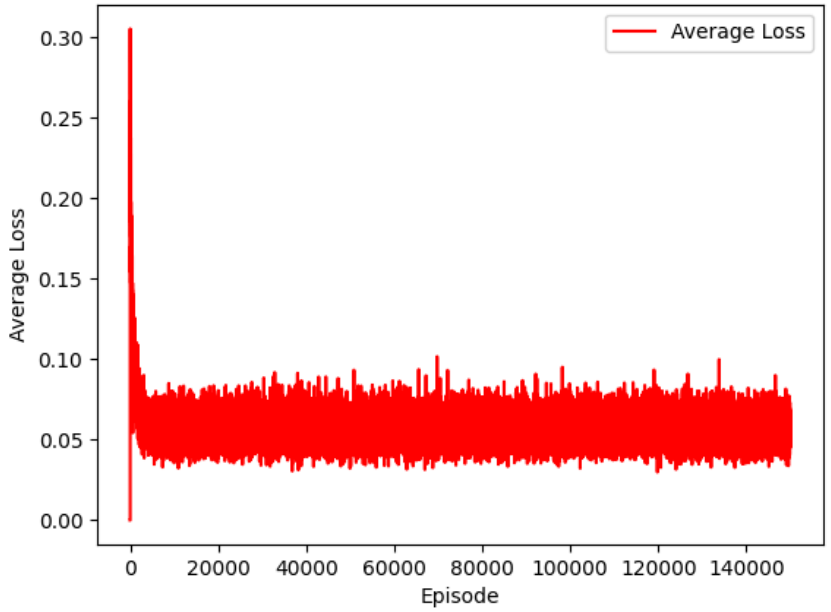

Imagen 1: Pérdida del agente from Scratch a través de los episodios.

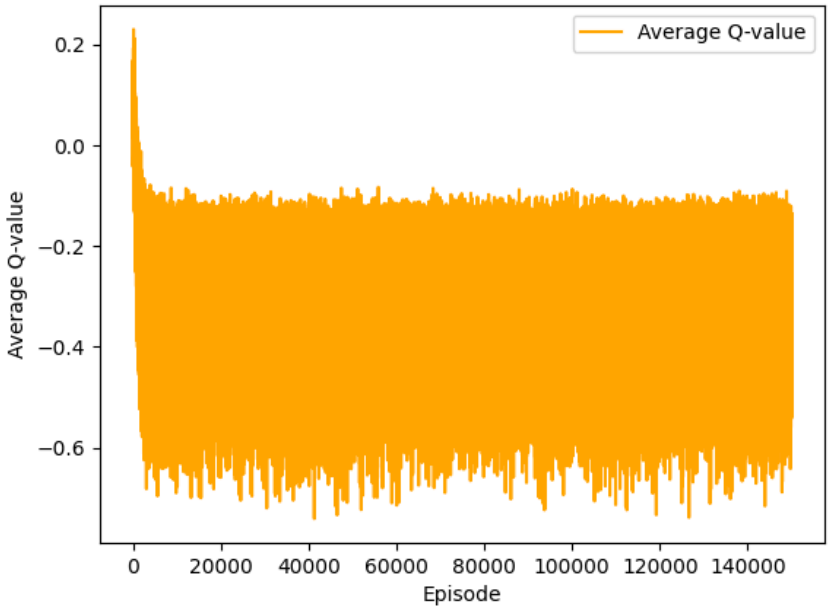

Imagen 2: Valores Q del agente from Scratch a través de los episodios.

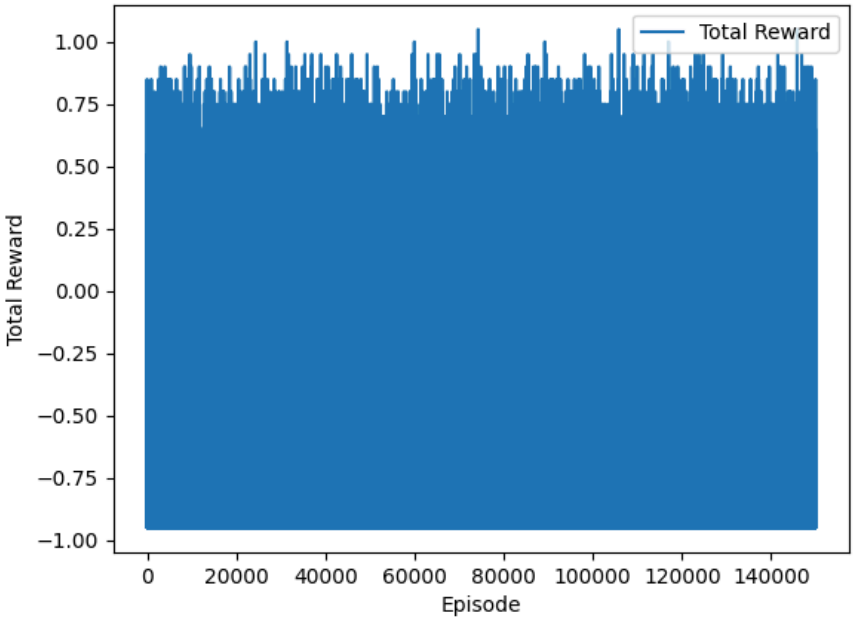

Imagen 3: Recompensas del agente from Scratch a través de los episodios.

Estas imágenes muestran el rendimiento del modelo en su entrenamiento, donde se puede ver que el la Imagen 1, bien rápido se llega a un rango de pérdida que se queda igual para todos los episodios futuros, lo cual no está logrando aprender mucho después de los episodios iniciales, al igual que los valores Q. Adicionalmente se puede ver que las recompensas del modelo se quedan igual entre todo el entrenamiento, indicando que el modelo no logró aprender una estrategia. Esto puede ser debido a que el tablero es muy pequeño con solo valores Q, sin

aplicación de políticas lo cual el modelo no logra encontrar tantos estados diferentes para crear su estrategia. Adicionalmente, no se podía incrementar el tamaño del tablero ya que eso incrementa el tiempo de entrenamiento del modelo mucho más, y los recursos necesarios que no se tendrían. Adicionalmente, al tratar de evaluarlo, simuló 100 partidas en un tablero de 5x5 con 3 minas, y nunca logró ganar. Lo cual se cambió a 2 minas, algo que es muy fácil, debido a la baja densidad de minas. Pero solo logró un 20% de victoria. Lo cual se decidió hacer una segunda implementación utilizando Gymnasium

- **Implementación con Gym:**

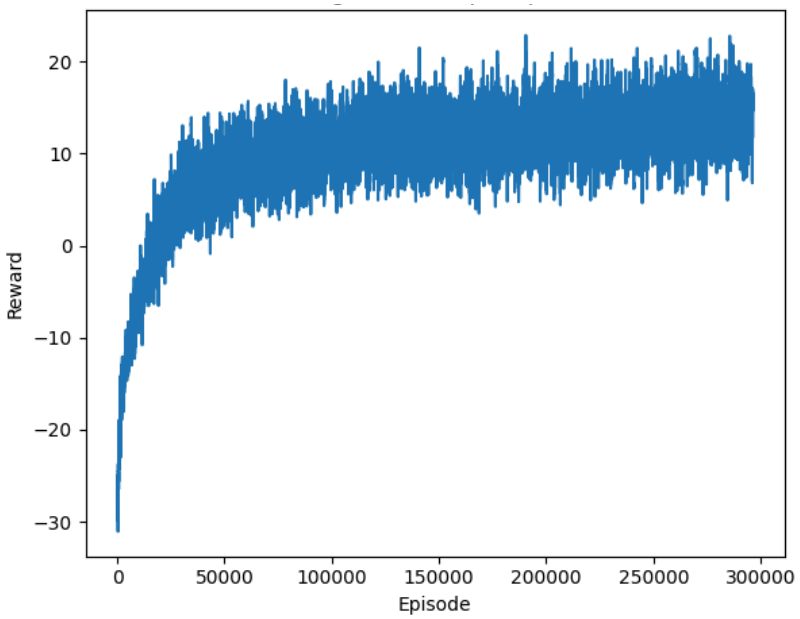

Imagen 4: Recompensas del agente entrenado con Gymnasium a través de los episodios.

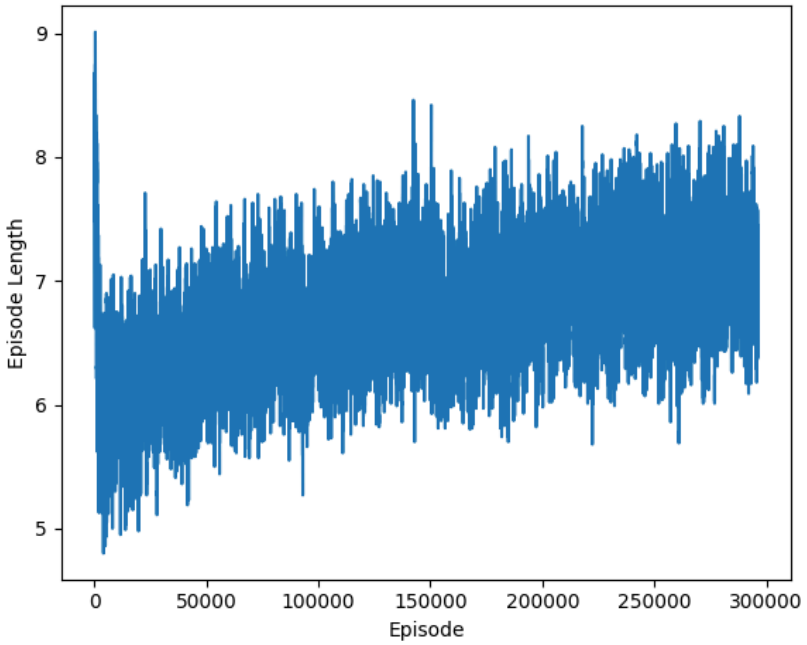

Imagen 5: Duración de episodios del agente entrenado con Gymnasium

Con esta nueva, se puede observar que va aprendiendo por la recompensa, pero también se ve que para de aprender, pero ya que está recibiendo una recompensa alta de un rango de 0 a 10, significa que si logro aprender algo y está ganando. Lo cual al realizar las simulaciones con tablero de 5x5 y 3 minas, en 100 juegos, logra tener un 50% a 60% de victoria. Lo cual es altamente satisfactoria ya que ese tablero, contiene una densidad de minas de 12%. Una densidad aceptable. Adicionalmente, al realizar la simulación con 4 minas, lo cual lleva a una densidad de 16%, la misma densidad de dificultad intermedia del minesweeper de windows, pero ya que es un tablero menor con más dificultad nos dio un 20% a 25% de victoria.

Conclusión

La implementación desde cero con DQN demostró ciertas limitaciones en el aprendizaje del agente. A pesar de un proceso de entrenamiento extenso, los resultados mostraron que tanto la pérdida como los valores Q se estabilizaron rápidamente sin una mejora continua, indicando que el agente no estaba logrando aprender una estrategia efectiva. Las recompensas permanecieron constantes a lo largo del entrenamiento, y las pruebas de simulación reflejaron una tasa de éxito baja incluso en configuraciones más simples, lo cual puede atribuirse a la falta de variación en los estados observados en un tablero pequeño y la imposibilidad de aumentar su tamaño debido a limitaciones de recursos.

En contraste, la implementación con Gymnasium y el modelo PPO mostró una mejora significativa en la capacidad del agente para aprender y adaptarse. El agente fue capaz de alcanzar una tasa de victoria del 50% al 60% en un tablero 5x5 con 3 minas (densidad de minas de 12%), lo cual es satisfactorio para un nivel de dificultad moderado. Al incrementar la densidad de minas a un 16% (4 minas), el agente aún mantuvo una tasa de éxito del 20% al 25%, lo cual es comparable a los niveles de dificultad intermedios en Minesweeper y demuestra que el modelo logró aprender estrategias de juego efectivas en un entorno más desafiante.

En resumen, el uso de Gymnasium y un método Actor-Crítico como PPO mejoraron la eficiencia y el desempeño del agente, permitiéndole aprender de manera más efectiva y enfrentar densidades de minas mayores, mientras que el DQN desde cero demostró ser menos adecuado para entornos complejos debido a su limitada capacidad de adaptación y exploración en un tablero pequeño.

Bibliografía

Gym Documentation. (2022). *Spaces*. Gym Library. https://www.gymlibrary.dev/api/spaces/#discrete

Paszke, A., & Towers, M. (2023). *Reinforcement Learning (DQN) Tutorial*. PyTorch. https://pytorch.org/tutorials/intermediate/reinforcement_q_learning.html

PyTorch. (n.d.). *Reinforcement Learning (PPO) with TorchRL Tutorial*. https://pytorch.org/tutorials/intermediate/reinforcement_ppo.html

Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., & Zaremba, W. (2016). *OpenAI Gym*. arXiv preprint arXiv:1606.01540. https://arxiv.org/abs/1606.01540

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017). *Proximal Policy Optimization Algorithms*. arXiv preprint arXiv:1707.06347. https://arxiv.org/abs/1707.06347