Universidad del Valle de Guatemala
Facultad de ingeniería

Proyecto Final: Battleship

Cayetano Molina 20211
Estefanía Elvira 20725
Priscilla Gonzalez 20689

Guatemala 12 de noviembre del 2024

1. **Descripción del problema:**

   El juego de Battleship es un juego de 2 jugadores que consiste en derribar los barcos del oponente. Cada jugador cuenta con 2 cuadrículas, una para poner sus barcos y otra donde intenta adivinar dónde se encuentran los barcos de su oponente. El juego se divide en 2 fases, la fase de posicionamiento y la fase de ataque.

   Figura 1: Cuadrícula de posicionamiento (abajo) y cuadrícula de ataque (arriba) junto con sus barcos respectivos.

   Para la fase de posicionamiento, cada jugador cuenta con 5 barcos:

   - Carrier: 5 espacios
   - Battleship: 4 espacios
   - Destroyer: 3 espacios
   - Submarine: 3 espacios
   - Patrol Boat: 2 espacios

   En esta fase los jugadores posicionan todos sus barcos en sus cuadrículas correspondientes ya sea de forma vertical u horizontal pero no en diagonal. Cabe mencionar que los barcos no se pueden traslapar uno sobre el otro, es decir, cada casilla en la cuadrícula solo puede ser ocupada por una parte de un único barco.

   Una vez puestos los barcos de cada jugador, se puede pasar a la fase de ataque. Cabe remarcar que una vez puestos los barcos no se podrán mover de su posición. En la fase de ataque, cada jugador toma turnos para “lanzar” un torpedo hacia la cuadrícula de barcos enemiga. Si al lanzar a ese punto en la cuadrícula fue un fallo, se marca como tal en la

cuadrícula de adivinanzas. Asimismo, si al lanzar hay un acierto, se debe marcar con otra etiqueta en la cuadrícula de adivinanzas. Si al lanzar un torpedo se derriba un barco, el dueño del barco está obligado a decir que el barco ha sido hundido.

2. **Análisis:**

Para plantear el problema en el contexto de RL se necesitan hacer unos cambios a las reglas del juego y tener ciertas consideraciones. Si se imagina a un juego, puede que este dure más de lo necesario o bien que dure menos de lo necesario debido a que alguno de los dos jugadores derrotó al otro. Por lo tanto, se considerará que los 2 jugadores tendrán un rol establecido, el agente como atacante, y el otro que solo pondrá sus barcos en su cuadrícula para que sean atacados.

Dentro de las consideraciones a tomar en cuenta hay que aclarar que el propósito del agente es que aprenda a reconocer patrones en cuanto a sus tiros. Que entienda que si tiene un acierto, lo más posible es que haya un cuadro cercano que también sea un acierto. Incluso, se debe tomar en cuenta que dado que no hay otro oponente como tal, cuando el atacante hunda algún barco no va a existir una respuesta de que en verdad hundió un barco. Es decir, cada vez que el agente tire un torpedo, lo único que podrá saber es si fue un acierto o bien si fue un fallo.

Tomando estas consideraciones en cuenta sobre el juego como tal, se puede establecer un paradigma acerca de lo que debería incluir un ambiente para el agente. Este ambiente no sólo debe establecer cómo se llevará a cabo el juego, es decir, reward y estados siguientes, sino que también debe establecer parámetros como tamaño de cuadrícula, botes, entre otras cosas. Para un juego normal de Battleship se tiene una cuadrícula de 10x10 y eso sería lo normativo para este ambiente. Esto quiere decir que el espacio de observación del juego sería del tamaño de cuadrícula escogida. Asimismo, el espacio de acciones disponibles también sería del mismo tamaño de la cuadrícula escogida.

Ahora bien, para llevar a cabo el entrenamiento de algún agente también se debe tomar en cuenta el espacio estado-acción que se puede observar. Para esto se deben considerar que para cada celda de la cuadrícula pueden existir 3 estados posibles. El primero que es cuando nada se ha lanzado, el segundo cuando se hizo un lanzamiento pero hubo fallo y el tercero que indica que hubo un acierto. Esto quiere decir que para una cuadrícula de $N \times N$, existen 3^{nxn} estados. Para una cuadrícula de 10x10, esta cantidad de estados sería demasiada como para llevar una tabla completa de estado acciones, se acabaría la memoria antes, por lo tanto, se debe implementar un solución que no deba saber todos los estados acciones completos para poder entrenar a un agente.

3. Propuesta de Solución:

Dadas las restricciones que se tienen en cuanto a memoria para guardar todos los estados posibles, se pueden utilizar modelos o algoritmos que tomen eso en cuenta como lo son los algoritmos de aproximación de funciones. Entre estos se encuentran algoritmos que implementan redes neuronales a su uso como lo pueden ser: Deep Q-Networks (DQN), Proximal Policy Optimization (PPO) y Advantage Actor-Critic (A2C). Estos algoritmos son útiles para realizar una estimación acerca de lo buena que sería una acción en un estado específico sin importar si se ha visto ese estado o no. Por esto mismo, esos fueron los algoritmos que fueron propuestos para la solución al problema del Battleship.

Sin embargo, antes de poder utilizar los algoritmos para entrenar a un agente, antes es necesario crear un ambiente. El ambiente sería un ambiente nuevo que heredaría de la clase Env de la librería Gymnasium en Python. En este ambiente se podrían definir tanto el tamaño de la cuadrícula de juego como el tamaño y cantidad de los barcos utilizados. Para definir un estándar para juegos, se utilizaría una cuadrícula de 10x10 con los barcos antes mencionados. Además, se estableció el espacio de acciones disponibles al igual que el espacio de observación. El estado de observación en este caso sería parecido a un array de 10x10 para poder representar la cuadrícula. Mientras que el estado de acción sería una lista de todos los números entre 0 y 100, cada uno representando una casilla en la cuadrícula de ataque. Esto se hizo por términos de simplicidad al implementar otras herramientas que se explicarán más adelante.

Para el ambiente también se tiene que establecer una función que interprete la acción realizada por el agente. En esta situación cada acción representará que se lanzó un torpedo a la cuadrícula de ataque. En esta cuadrícula se tendrán 3 valores, -1 para representar espacios vacíos, 0 para representar fallos y 1 para representar aciertos. Se sabrá si se tiene un acierto o un fallo al comparar la casilla contra una cuadrícula que contiene barcos previamente puestos y seleccionados y revisar si acertó o no.

Además por cada acción también se devolverá el estado actual respectivo al igual que la recompensa que se tiene por esta acción. Las recompensas están divididas de la siguiente manera:

- Aciertos = 1 punto
- Primer fallo = 0 puntos
- Volver a disparar a una casilla ya antes disparada = -0.2 puntos
- Fallo pero tiene aciertos anteriores cerca = 0.2 puntos por cada otro acierto a menos de 2 casillas.
- Ganar la partida = 10 puntos

Estas recompensas servirán para saber que tan bien fue la acción realizada para un estado específico. Además, para comprobar si se ganó el juego, se llevará un conteo de los aciertos a cada barco y se comparará con el tamaño total que ocupan los barcos. Cabe mencionar que para el fallo con aciertos cerca se utilizará la distancia Manhattan que indica la distancia entre 2 puntos si solo se puede ir horizontal o verticalmente.

$$D = abs(x_2 - x_1) + abs(y_2 - y_1)$$

Por último, se 3 agentes con diferentes algoritmos bajo este mismo ambiente para poder determinar cual es el que mejor rendimiento tiene y el que mejor logra generalizar el problema de BattleShip

4. Herramientas aplicadas:

Para la solución del problema se utilizaron 2 librerías específicamente para llevar a cabo el entrenamiento del agente y creación del ambiente. Para crear el ambiente se utilizó Gymnasium, la cual se vio durante clase, mientras que para el entrenamiento se utilizó stable baseline 3. Esta última es compatible con los ambientes de Gymnasium y todos los que heredan de estos. Además, ofrece una variedad de algoritmos como: PPO, DQN, A2C, los cuales son útiles para la solución de este proyecto.

La librería stable baseline 3 además, ofrece la opción de entrenamiento tanto con tarjeta gráfica como con CPU, e incluso de ser posible puede paralelizar tareas para que sea más rápido el entrenamiento. Para poder llevar un control de lo que ocurre durante el entrenamiento, esta librería ofrece callbacks para poder observar el reward que obtuvo el agente por cada episodio y también la cantidad de pasos que le tomó terminar con el episodio (o en otras palabras hundir todos los barcos).

Por último, utilizando estas métricas se pudo comparar los 3 algoritmos mencionados anteriormente, específicamente se compararon utilizando gráficas realizadas con matplotlib. Con esto no solo se pudo observar la tendencia de cada agente al ser entrenado sino que además, se pudo observar la distribución de sus métricas al ser evaluados.

5. Resultados:

Las figuras 2 y 3 muestran las métricas vistas durante el entrenamiento del agente que fue entrenado utilizando el algoritmo PPO. Como se puede observar, tanto la recompensa como la cantidad de pasos tienden a un valor en particular. En la figura 2, se puede

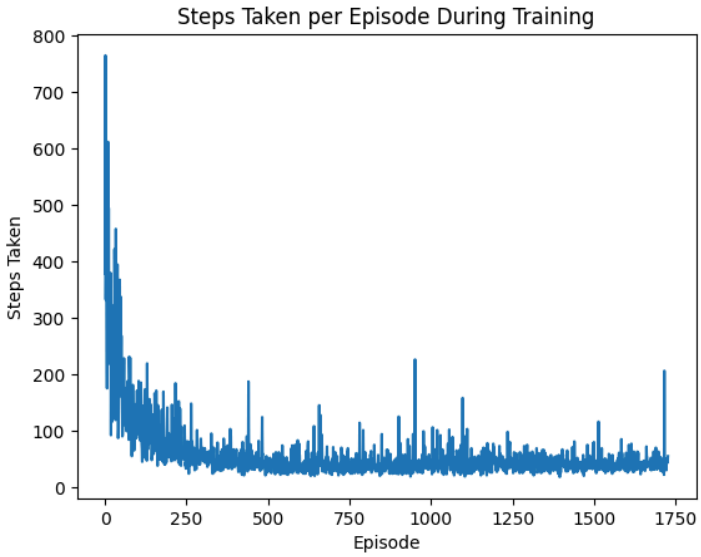
observar que por lo general, la recompensa total de un episodio apunta a ser 40 puntos o un poco más.

Sin embargo, lo que más puede darnos un indicio de que tan bien le fue al agente es la métrica de cuántos pasos le tomó hundir todos los barcos. Como se puede observar, en la figura 3, existe una curva de aprendizaje evidente donde los pasos para terminar un episodio cada vez son menos. Se puede ver una tendencia a números menores de 100. Esto indica que se está observando o una mejora significativa e incluso podría decirse que es algo esperado. Debido a que la cuadrícula tiene 100 cuadros disponibles donde apuntar, el hecho de que tome más de 100 pasos terminar, significa que se está lanzando a cuadros ya antes lanzados.

Por lo tanto, al tener una recompensa que se estabiliza y una cantidad de pasos menor a 100, se puede decir que el agente está aprendiendo a reconocer patrones importantes a la hora de lanzar torpedos.

Figura 2: Recompensa obtenida por cada episodio para algoritmo PPO

Figura 3: Pasos tomados por cada episodio para algoritmo PPO



La misma tendencia vista para el algoritmo PPO se puede observar para el algoritmo DQN. Sin embargo, aquí existe un cambio más abrupto en cuanto a una estabilización. Como se puede observar, al principio el algoritmo tiene un rendimiento increíblemente malo, lo cual puede indicar que está realizando una exploración a fondo del ambiente. Poco a poco se puede observar que tanto para recompensas como para pasos el agente logra encontrar una política que hace sentido y con ella se mantiene sin mucho cambio durante los siguientes episodios.

Debido a que los agentes fueron entrenados en una cantidad de pasos y no una cantidad de episodios, el algoritmo DQN, al llegar a una estabilización antes, pasa por más episodios que el PPO. Esto significa que, los pasos que necesita este algoritmo para terminar cada episodio son menos que los que fueron necesarios para el PPO.

Figura 4: Recompensa obtenida por cada episodio para algoritmo DQN

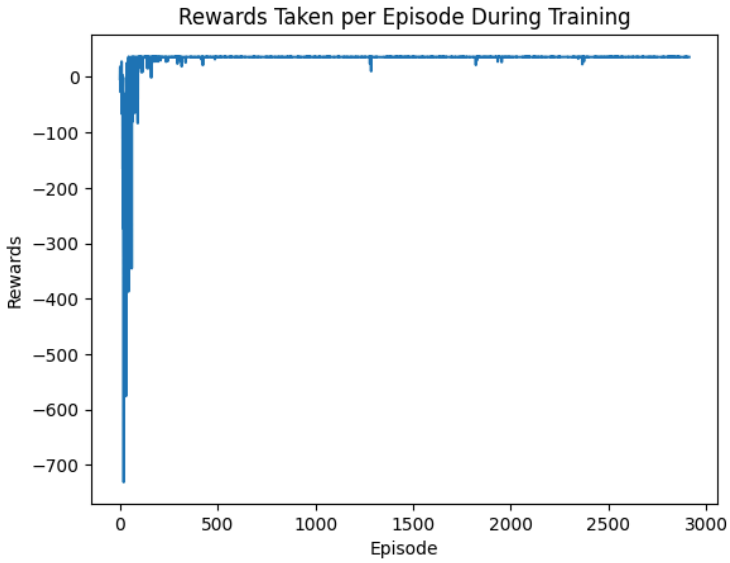
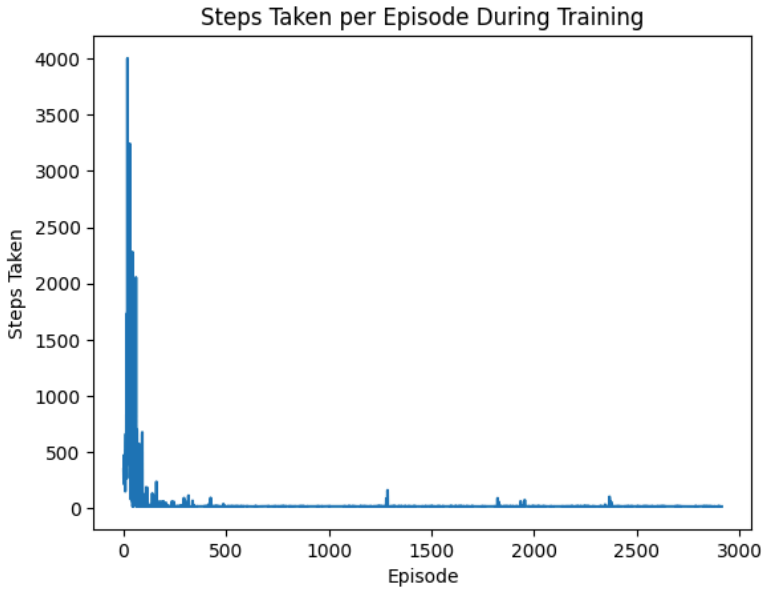


Figura 5: Pasos tomados por cada episodio para algoritmo DQN



Por último se tiene el algoritmo A2C, este algoritmo por mucho es el que más se tarda en converger a un valor óptimo en la póliza. Esto es evidente por las figuras 6 y 7 donde se puede observar que le toma varios episodios poder estabilizarse. Esto puede indicar que el algoritmo está logrando aprender lentamente y puede reconocer ciertos patrones que son necesarios para completar un episodio. Al igual que con el DQN, al tardar más en promedio en completar un episodio, este algoritmo no logra pasar tantos episodios como los otros. De lo visto por las gráficas, este algoritmo presenta los resultados más bajos en cuanto al entrenamiento.

Figura 6: Recompensa obtenida por cada episodio para algoritmo A2C

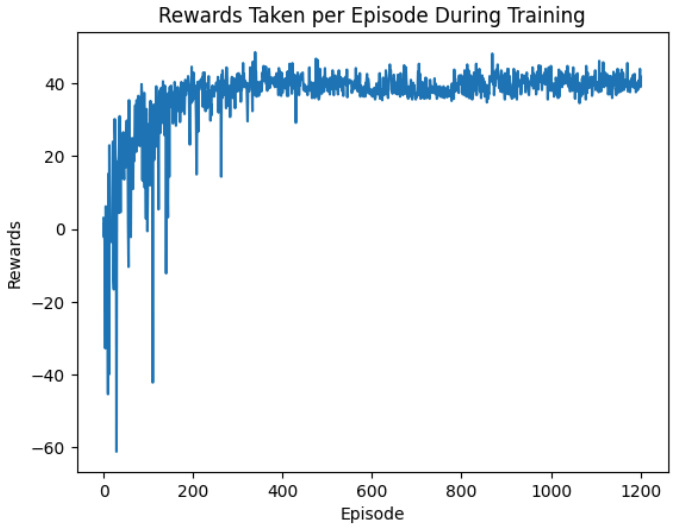
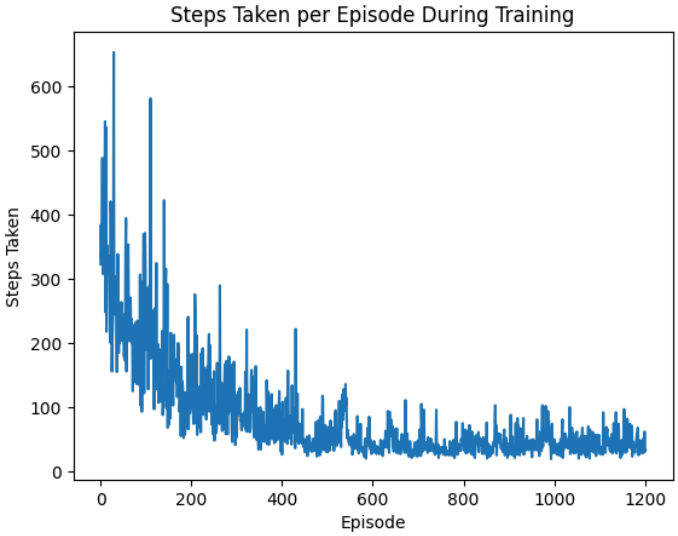


Figura 7: Pasos tomados por cada episodio para algoritmo A2C



Las figuras 8, 9 y 10 muestran las métricas para cada algoritmo al ser utilizados en un ambiente con el que no interactuaron. Este nuevo ambiente tenía las mismas características del de entrenamiento con la diferencia que este tenía los barcos en posiciones diferentes. De forma general, todos los algoritmos tuvieron un rendimiento mucho peor que el que tuvieron durante su entrenamiento. Esto indica, que los patrones que descubrieron o que pudieron observar estaban adaptados a la configuración de los barcos de ese ambiente. En otras palabras, los agentes estaban sobreajustados.

Sin embargo, si se observa más a fondo se puede saber que los resultados son completamente lo contrario a lo visto en la parte de entrenamiento. En esta evaluación el peor agente fue el entrenado usando DQN, lo cual indica que su rápida estabilización en el entrenamiento fue causada por un sobreajuste a la configuración de barcos. Por otro lado, el algoritmo A2C fue el que mejores resultados dió, pudiendo establecerse en un rango más cercano a lo ideal en cuanto a recompensas y pasos tomados. Aún así, como se mencionó anteriormente, debido a que no llega a tener un rendimiento de pasos tomados mejor a 100, esto indica que no sería útil utilizarlo en un ambiente que no conozca.

Figura 8: Métricas del algoritmo PPO para ambiente de evaluación

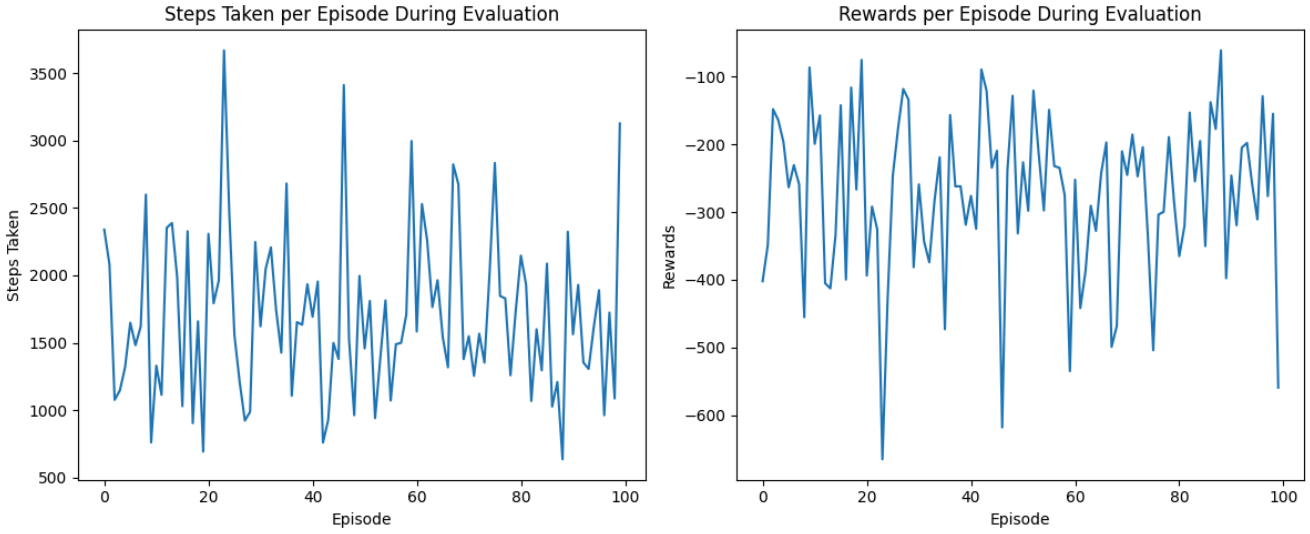


Figura 9: Métricas del algoritmo DQN para ambiente de evaluación

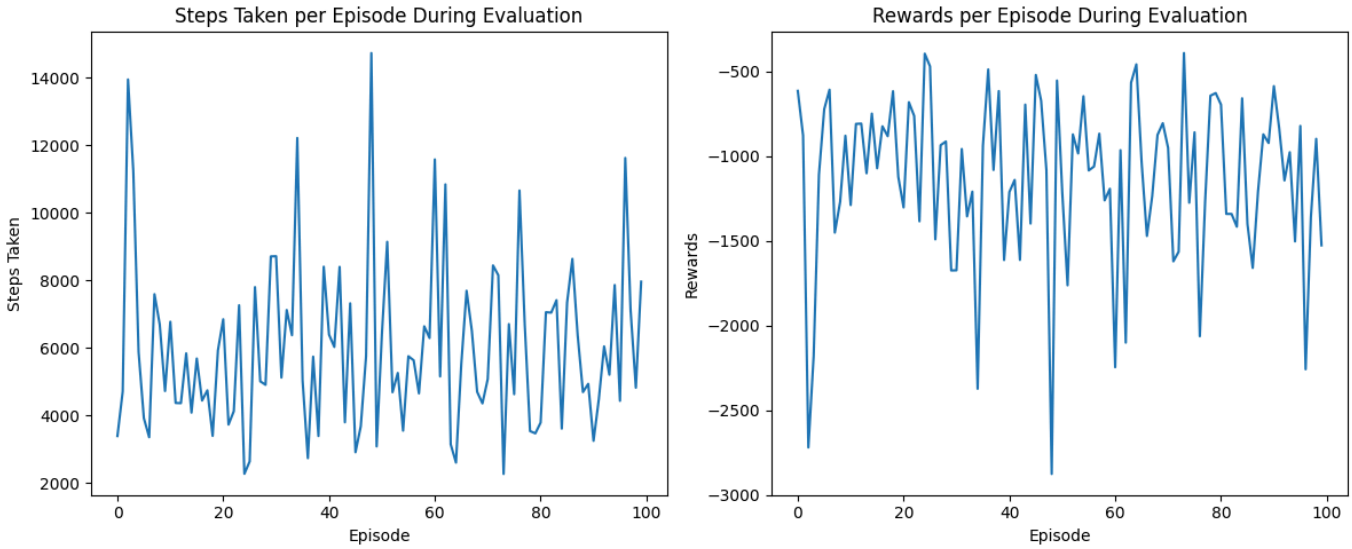
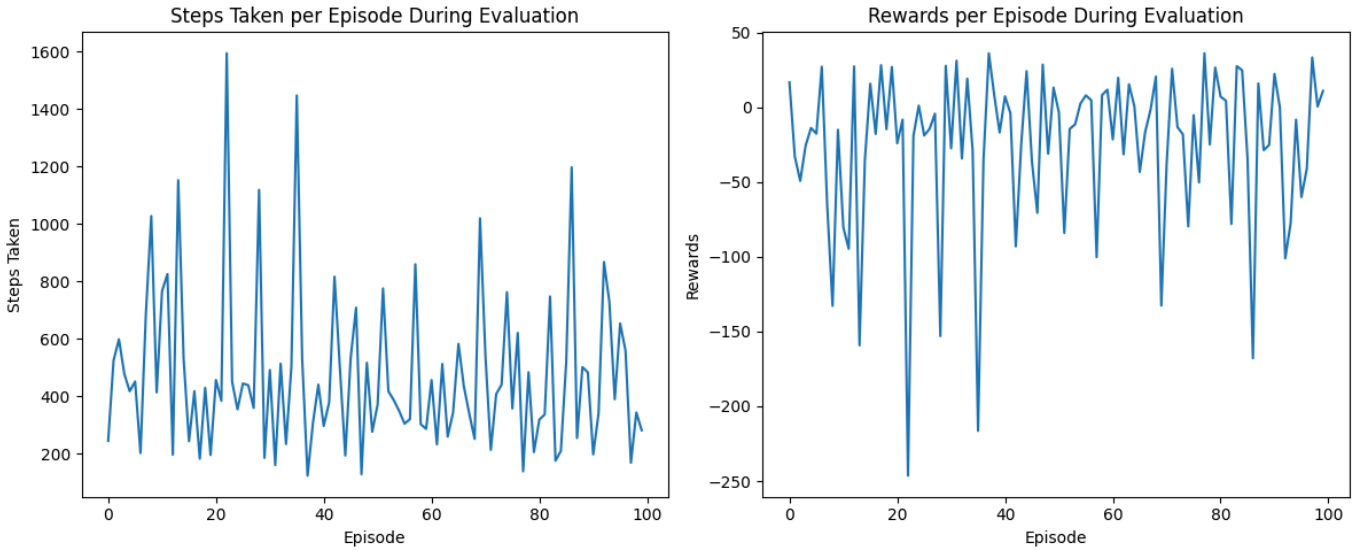


Figura 10: Métricas del algoritmo A2C para ambiente de evaluación



6. Conclusión:

1. El algoritmo de A2C fue el que mejor logró reconocer los patrones para una solución de BattleShip.
2. El algoritmo de DQN ofrece una estabilización rápida durante el entrenamiento (la más rápida de todos los modelos) pero puede tender a sobreajustarse fácilmente.
3. Es necesario implementar alguna medida que evite el sobreajuste y que pueda adquirir más información de otras configuraciones de barcos no antes vistas.

7. Bibliografía:

1. Kumar, D. (2024). PPO Algorithm. https://medium.com/@danushidk507/ppo-algorithm-3b33195de14a
2. Stable Baselines3 (2024). Stable-Baselines3 Docs - Reliable Reinforcement Learning Implementations. https://stable-baselines3.readthedocs.io/en/master/index.html
3. qwerty_gfg. (2024). Actor-Critic Algorithm in Reinforcement Learning. https://www.geeksforgeeks.org/actor-critic-algorithm-in-reinforcement-learning/
4. Clementis, L. (2013). Supervised and Reinforcement Learning in Neural Network Based Approach to the Battleship Game Strategy. DOI: 10.1007/978-3-319-00542-3_20
5. Kancko, T. (2020). Reinforcement Learning for the Game of Battleship. MASARYK UNIVERSITY. FACULTY OF INFORMATICS.
6. Patel, R. (2021). battleship. Github repo: https://github.com/rshnn/battleship
7. Hirtz, T. (2023). gym-battleship. Github repo: https://github.com/thomashirtz/gym-battleship