Universidad de Ingeniería Y Tecnología

Departamento de Ingeniería Mecatrónica

Robótica Avanzada

Proyecto final

Integrantes

Centeno Untiveros, Luis Esteban

Palma Rodríguez, Diego Alonso V.

Vásquez Espinoza, Alexandra Sofia

Profesor: Ph.D Oscar E. Ramos

Lima - Perú

2021-2

1. Introducción

En el siguiente proyecto se realizará la implementación de algoritmos de Reinforcement Learning (RL) en una aplicación sencilla, para luego mostrar cómo podría ser extrapolada a aplicaciones más complejas enfocadas en robótica. En este caso, se escogió el popular juego *Tic Tac Toe* o "Tres en raya". Este se juega por turnos entre dos personas en un tablero de 3x3, tal y como se visualiza en la figura 1. En cada turno un jugador colocar su ficha en una posición del tablero y gana el que logre tener 3 de sus fichas en una sola fila, columna o en diagonal. La simplicidad de este juego nos permite utilizarlo como ejemplo para la aplicación de algoritmos complejos, como los de Inteligencia artificial o RL.

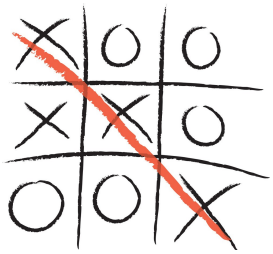


Figura 1: Ejemplo del juego "Tres en raya"

Si bien existen algoritmos que hacen uso de métodos generales de teoría de juegos, como por ejemplo el algoritmo `minimax` (Figura 2), estos siempre asumen un oponente perfecto, lo que significa que el oponente siempre buscará ganar, y cada uno de sus movimientos están enfocados en maximizar su ganancia final (ganar la partida) y en lo posible, perjudicarnos. En cambio, con RL nosotros consideramos el oponente como parte del entorno con el que nuestro agente es capaz de interactuar lo cual permite al agente a aprender a 'planificar' ante cualquier posible movimiento, ya sea bueno o malo para el oponente. La gran ventaja con respecto a utilizar algoritmos de teoría de juegos es que este ahorra un gran esfuerzo computacional, a diferencia de muchos de los algoritmos de teoría de juegos, los cuales suelen recurrir a la recursividad para analizar todos los posibles movimientos y así seleccionar el que da mayor ventaja.

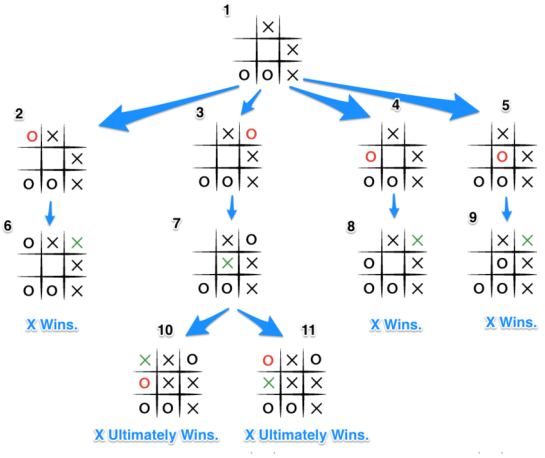


Figura 2: Ejemplo visual del algoritmo `minimax` en Tres en raya

1.1. Estado de arte

La aplicación de RL en el área de los juegos no es un aspecto reciente, la investigación en este rubro ha sido bastante llamativa a través de los años. [1] Uno de los ejemplos que fue más comentado en su tiempo fue el sistema Watson desarrollado por IBM, el cual comenzó su desarrollo en el año 1996. Hizo su debut en la televisión en el 2011, en donde Watson concurso en partidos oficiales del programa Jeopardy!, para los cuales se preparó con anticipación mediante simulacros en donde ganó un 65 por ciento de los partidos. IBM aplicó RL en Watson, un sistema que entiende y responde en lenguaje natural las preguntas que se le dan. Más especificamente, se empleó RL a la estrategia que se utiliza para jugar Jeopardy, como por ejemplo, cuando intentar responder una pregunta según el grado de certeza en sus respuestas, tambien que casilla del tablero seleccionar y cómo apostar en el juego.

Figura 3: Watson desarrollado por IBM

Otro ejemplo se da introduce en el 2019 a AlphaStar desarrollado por Google Deepmind, la primera inteligencia artificial en vencer en condiciones de un partido profesional a un jugador profesional en StarCraft II. Debido a la complejidad del juego, no se encontraban un sistema que resolviera los problemas. En el caso de AlphaStar, este puede jugar de manera satisfactoria un juego completo debido a que utilliza redes neuronales profundas (deep RL) que se entrenan a partir de datos mediante aprendizaje supervizado y *multi-agent reinforcement learning*. En particular, se menciona que usa un nuevo algoritmo para *off-policy reinforcement learning* que permite actualizar de manera eficiente su politica a partir de partidas anteriores jugadas con una politica anterior. Como resultado se obtiene que sistemas de aprendizaje abiertos que utilizan agentes basados en el self-play han logrado resultados impresionantes en dominios que son desafiantes como en el caso de StarCraft II. [2]

Como último ejemplo tenemos el sistema inicialmente nombrado como AlphaGo, también desarrollado por Google Deepmind. En el 2016, Google comenzó a aplicar RL para construir un programa que pueda jugar Go, un juego donde hay 361 movimientos posibles solo para la apertura. Para este programa inicial se utilizaron redes neuronales y árbol de busqueda, asimismo, se dió como información datos humanos, conocimiento acerca del dominio de juego y sus reglas para que el sistema tomara esto de base. Sin embargo, un año despues, el ahora nombrado AlphaGo Zero, aprendio a jugar completamente por su cuenta utilizando RL sin necesidad de información externa. En el 2018, la nueva versión de AlphaZero domino 3 juegos de manera perfecta (Go, ajedrez y Shogi) con un mismo algoritmo para estos. La útima versión publicada, MuZero, se añade entre los juegos a Atari y además, aprende las reglas de juego lo cual permite dominar entornos con dinamicas desconocidas.

2. Desarrollo

En la presenta sección se abordan los conceptos que forman parte del esquema de RL utilizado. Asimismo, se define el método de resolución del problema. Finalmente, se muestran las características del entrenamiento de los algoritmos.

2.1. Componentes de *Reinforcement Learning*

Antes de realizar el entrenamiento es importante definir claramente cada uno de los componentes de nuestro esquema de RL, los cuales se muestran en la figura 3. Esto es de gran iimportancia pues de eso dependerá la eficacia del modelo entrenado. Nuestro esquema es ligeramente distinto al común porque en este caso se entrenarán dos agentes a la vez, donde cada agente tendrá la capacidad de tomar una acción independientemente del otro agente en un mismo entorno. Por esto se puede decir que el Agente 1 considera al Agente 2 como parte de su entorno.

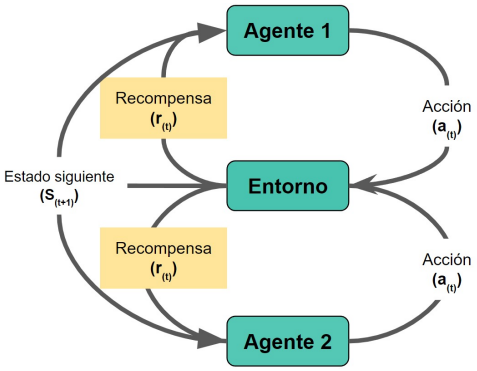


Figura 4: Componentes del esquema de RL empleado

En la Figura 4 se observa que los componentes principales son los agentes, los estados (s_t), las acciones (a_t) y las recompensas (r_t). A continuación, se define y describe cada uno de ellos.

2.1.1. Agentes

El agente es el 'sujeto' quien interactuará con el entorno modificándolo en base a sus acciones, generando así nuevos estados. En esta aplicación el agente será el jugador, quien solo tiene conocimiento del estado actual del tablero (casillas libres y ocupadas) y cual es su ficha ('X' para el agente 1 y 'O' para el agente 2). Esto se se muestra en la figura 5. Cabe mencionar que este no tiene ningún tipo de vínculo o información del oponente.

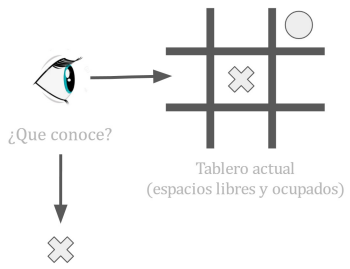
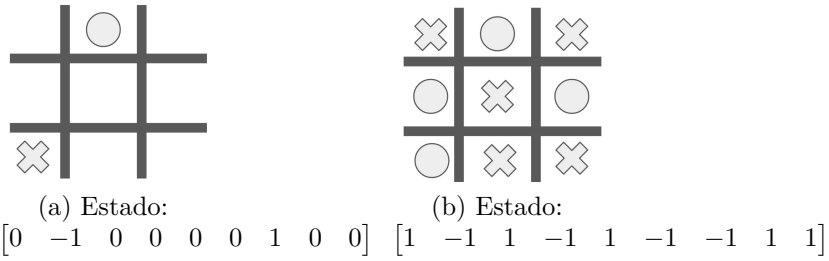


Figura 5: Agente del juego

2.1.2. Estados

Los estados son las distintas configuraciones que se tiene durante la partida. En la Figura. 5 se pueden observar algunos ejemplos de estados que se tendrían en una partida, por facilidad se considerará como 1 y -1 a las fichas 'X' y 'O' respectivamente y 0 a un espacio libre o vacío.



(a) Estado: $[0 \quad -1 \quad 0 \quad 0 \quad 0 \quad 0 \quad 1 \quad 0 \quad 0]$

(b) Estado: $[1 \quad -1 \quad 1 \quad -1 \quad 1 \quad -1 \quad -1 \quad 1 \quad 1]$

Figura 6: Ejemplos de estados en una partida

2.1.3. Acciones

La única acción que el agente puede realizar es decidir una posición donde colocar su ficha en base a la configuración actual del tablero. Un pequeño ejemplo se muestra en la figura 7 La única limitación es que no puede escoger una posición que ya esté ocupada por alguna otra ficha.

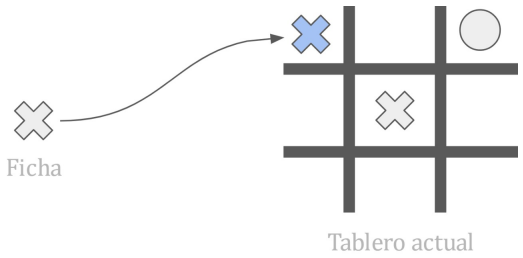


Figura 7: Acción del agente

2.1.4. Recompensas

Una de las principales partes del esquema es definir una correcta recompensa para nuestro agente. Esto es importante porque de esta manera nosotros le 'decimos' al agente que tan bien o mal está tomando las decisiones. Para esta aplicación tendremos 3 posibles recompensas, mostradas en la Tabla 1, las cuales solamente se darán una vez que la partida termine, ya sea en empate o en victoria de alguno de los dos jugadores. En caso de empate se definirá una recompensa de 0.1 para el agente 1 y 0.5 para el agente 2.

Tabla 1: Recompensas

Recompensa	**Valor**
Al ganar	1
Al perder	0
Al empatar	0.1 / 0.5

La recompensa dada al empatar no es un valor alto ya que un valor alto de esta recompensa implicaría que el agente también buscaría empatar con más frecuencia en las partidas. No se

utilizó un valor de 0 por el hecho de que si el agente tendría que decidir entre empatar o perder elija empatar la partida.

2.2. Método de Resolución del problema

Con todos nuestros componentes definidos ahora nos centraremos en el entrenamiento para poder encontrar la mejor política que permita a nuestro agente ser capaz de ganar la mayor cantidad de partidas.

Primero, para poder balancear la exploración de la explotación utilizaremos el método ϵ-*greedy* con un valor constante de 0.3. Lo cual significa que el 30 % de nuestras acciones serán tomadas de manera aleatoria y el porcentaje restante serán acciones tomadas en base a la experiencia del agente, buscando la acción que le permita maximizar su ganancia futura.

Luego de esto definimos la manera en la que actualizaremos los state values, se utilizará el gradient-descent update rule.

$$V(S_t) \leftarrow V(S_t) + \alpha[V(S_{t+1} - V(S_t)] \tag{1}$$

La ecuación (1) nos indica que el valor actualizado del estado t es igual al valor actual del estado t sumando la diferencia entre el valor del siguiente estado y el valor del estado actual, que se multiplica por una tasa de aprendizaje α. Recordemos que no existe ninguna recompensa intermedia, solamente al final.

Finalmente, escogemos un valor de gamma de 0.9 debido a que un valor alto de gamma generará que nuestro agente considere más importante las futuras recompensas y no solamente se enfoque en la acción inmediata.

2.3. Entrenamiento

El entrenamiento de ambos agentes se realizó de manera simultánea, actualizando el *state value* de cada agente independientemente. Un episodio de entrenamiento termina cuando algún agente gana la partida o cuando ya no hay casillas disponibles en el tablero (empate).
Se utilizaron distintas cantidades de entrenamiento, esto para poder evaluar el impacto sobre nuestro modelo. Se utilizaron 500, 5000, 50 000 y 500 000.

Tabla 2: Parámetros utilizados para el entrenamiento

Parámetro	γ	Learning rate	Recompensa al ganar	Recompensa al perder	Recompensa al empatar	Exploration rate	Número de episodios
Valor	0.9	0.2	1	0	0.1/0.5	0.3	500000

3. Resultados

Luego de realizar el entrenamiento se obtiene una tabla similar a la mostrada en la figura 8. En esta tabla se puede observar cada uno de los estados con una valorización, esta valorización nos permitirá eventualmente seleccionar la mejor acción para una configuración. Algo que resaltar es que algunos valores tienden a ser 0.9, estas son las configuraciones donde el agente gana y toma un valor de 0.9 a causa del parámetro γ.

```
'[ 0.  0.  0.  1. -1. -1.  0.  1.  1.]': 0.19974051818820723,
'[ 0.  0.  0.  0. -1.  0.  0.  1.  1.]': 0.17540902307928902,
'[ 1.  1. -1. -1. -1.  1.  1. -1.  1.]': 0.08999999999999998,
'[ 1.  0.  0. -1. -1.  1.  1. -1.  1.]': 0.007111087448083683,
'[-1.  0.  0.  1. -1.  1. -1.  1.  1.]': 0.058827511435155466,
'[-1.  0.  0.  0.  0.  1. -1.  1.  1.]': 0.06484906094682767,
'[-1.  1.  1.  1. -1. -1. -1.  1.  1.]': 0.08999999999999998,
'[ 1.  0.  0.  1.  1. -1. -1.  1. -1.]': 0.00044200942236342536
'[ 1.  0.  0.  1.  0.  0. -1.  1. -1.]': 0.1145562472848929,
'[ 0.  0.  1.  0. -1. -1.  0.  1.  1.]': 0.06984846127606444,
'[ 0.  0. -1.  1. -1. -1.  1.  1.  1.]': 0.899999951472096,
'[ 0.  0. -1.  1. -1.  1.  1. -1.  1.]': 0.004567648943543628,
'[ 0.  0.  0.  0. -1.  1.  1. -1.  1.]': 0.042407495510308826,
'[ 0.  0.  0.  0. -1.  1.  0.  0.  1.]': 0.2509317162028311,
'[ 1.  0.  0.  0. -1.  0.  1. -1.  1.]': 0.00859894956095246,
'[ 1.  0.  0.  0. -1.  0.  0.  0.  1.]': 0.1519468750388638,
'[ 1.  0. -1.  0.  1.  1. -1. -1.  1.]': 0.8999998519045899,
'[ 0.  0. -1.  0.  1.  1.  0. -1.  1.]': 0.8058867636336764,
```

Figura 8: Q table obtenido para el agente 1

Un ejemplo de la selección del 'mejor movimiento' se muestra en la figura 9 donde tenemos como 'entrada' el estado actual del tablero y el agente buscará solamente entre las posibles configuraciones y escogerá el que tiene un mayor valor. Esto es porque el estado con el mayor valor aumentará las posibilidades de ganar la partida.

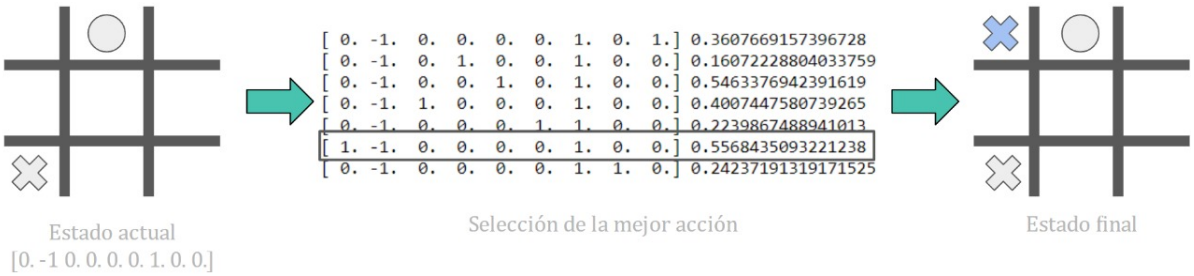


Figura 9: Elección de la mejor acción para el estado $\begin{bmatrix} 0 & -1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix}$

Al brindar solamente una recompensa al final de todo el entrenamiento no es conveniente realizar una gráfica de Recompensas vs Episodios. Esto porque solamente tendríamos valores de 1, -1 y 0.1/0.5, lo cual no brinda información relevante, en cambio se puede graficar el número de victorias de ambos agentes a medida que los episodios aumentan. Lo que esperaríamos de esta gráfica es que la cantidad de victorias aumente significativamente a medida que avanzan los episodios, esto porque se espera que el agente tenga más 'experiencia' jugando.

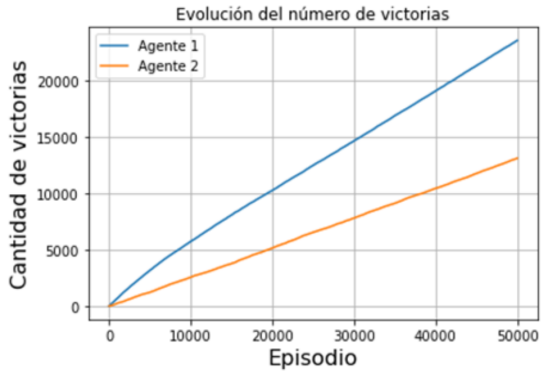
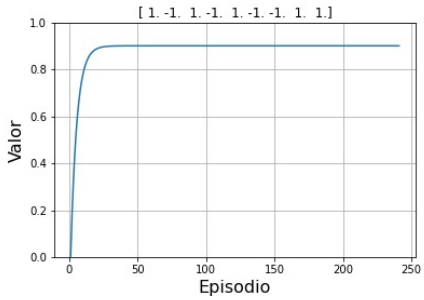


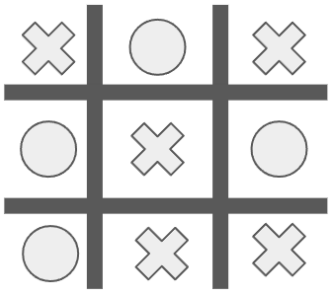
Figura 10: Comportamiento del número de victorias en función del número de episodios

Como se muestra en la figura 10 hay una gran diferencia entre el número de victorias del agente 1 y del agente 2 (aproximadamente 10 000 victorias). Esto se debe a las recompensas que se le dieron a cada uno de los agentes, recordando que al agente 2 se le da una recompensa de 0.5 si es que empata, lo que provoca que de cierta manera también prefiera empatar en varios casos en lugar de buscar la victoria.

Otra información importante que podemos obtener del entrenamiento es la evolución de los valores de los estados en cada uno de los episodios. Estas gráficas nos permitirán notar algún comportamiento de convergencia en los estados para decidir aumentar la cantidad de episodios de entrenamiento o no. Algunas de estas gráficas se muestran en las figuras 11-13.

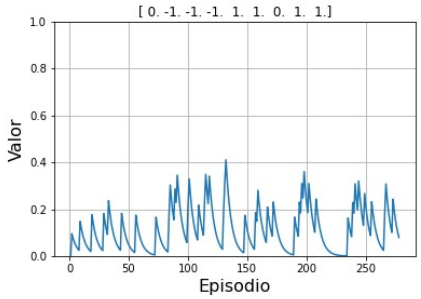


(a) Valor de los estados

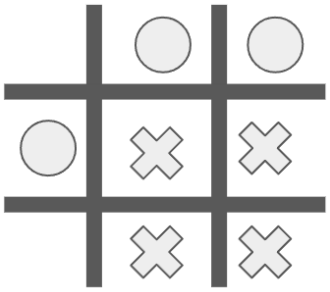


(b) Estado Final

Figura 11: Estado donde el agente 1 ganó

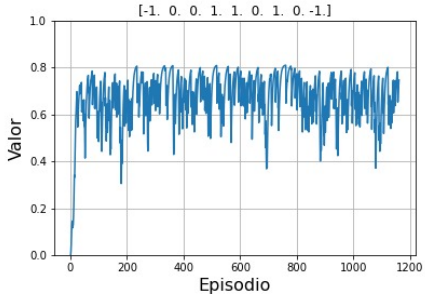


(a) Valor de los estados

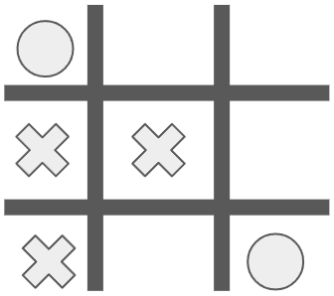


(b) Estado Final

Figura 12: Derrota inminente del agente 1



(a) Valor de los estados



(b) Estado Final

Figura 13: Victoria inminente del agente 1

Las figuras anteriores representan 3 momentos o estados comunes en las partidas de Tres en raya. La primera (Figura 11) muestra el momento donde el agente 1 gana la partida, lo que significaría

que este debe recibir una recompensa de 1, pero al ver la gráfica vemos que el valor logra converger en 0.9, lo cual se debe al factor de Gamma utilizado. La segunda figura (Figura 12) muestra un estado donde lo más probable es que el agente pierda la partida, es por esto que la recompensa tiene varios picos bajos junto con algunos valores altos, estos se deben a que existen casos donde el oponente no coloca la ficha en el lugar que le dará la victoria. Finalmente, (Figura 13) muestra un estado donde el agente ganará sin importar el movimiento que haga el oponente, y debido a esto su valor tiende a ser muy grande, oscilando en 0.7 aproximadamente.

3.1. Efecto de los parámetros de entrenamiento

El efecto del valor de la recompensa al empatar se muestra en la figura 14, donde se observa que a medida que este valor aumenta nuestra cantidad de victorias va disminuyendo. Asimismo, podemos concluir que el mejor valor para esta recompensa es de 0.1 porque es la que maximiza la cantidad de victorias obtenidas.

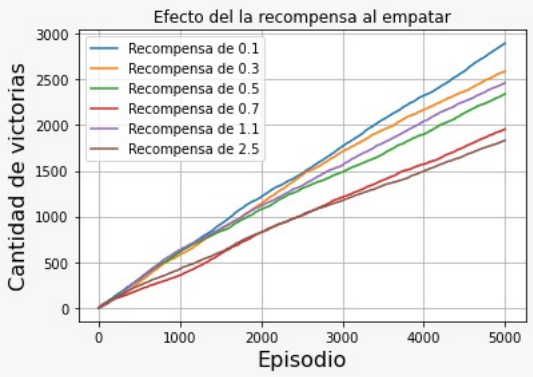


Figura 14: Variación de la recompensa al empatar

Otra modificación es la del *Learning Rate*, el cual es el tamaño de paso que tomamos en cada actualización de los estados. Pero como vemos en la figura 15 este parámetro no tiene un gran efecto en la cantidad de victorias que tiene el modelo entrenado ya que las curvas se mantienen cerca entre ellas.

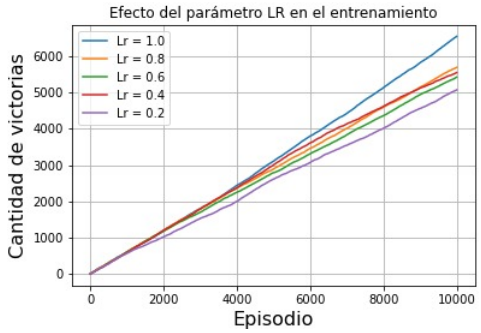


Figura 15: Efecto del parámetro del *Learning rate*

Finalmente, en la figura 16 se muestra el efecto que tiene la cantidad de episodios de entrenamiento en el primer movimiento que se realiza. En este se muestra que a medida que se aumentan los episodios el primer movimiento aumenta considerablemente de valor, lo que significa que aumentaremos nuestras probabilidades de ganar a la larga.

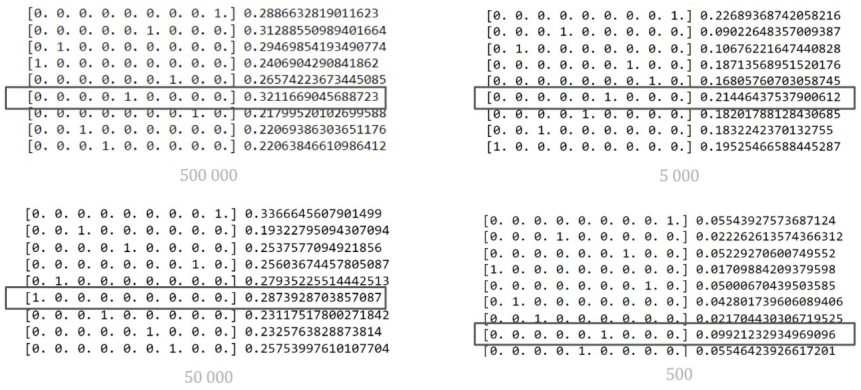


Figura 16: Efecto del parámetro del *Learning rate*

4. Aplicaciones en la robótica

Las aplicaciones del Reinforcement Learning en robótica son bastantemente amplias. Desde los años 90's se realizaron aplicaciones dentro de la navegación móvil, navegación autónoma y sobretodo en tareas repetitivas. Por ejemplo, en 1992 el robot OBELIX, un robot móvil con ruedas, que aprendió a empujar cajas con un enfoque basado en la función de valor [3]. Asimismo, el brazo robótico Zebra Zero aprendió una tarea de inserción de clavijas en un agujero con un enfoque de gradiente de política sin modelo [4]. Además de ello, el helicóptero autónomo de Carnegie Mellon aprovechó un enfoque de búsqueda de políticas basado en modelos para aprender un controlador de vuelo robusto [5]. Recientemente, el uso de un método de RL libre de modelos y basado en visión computacional fue empleado para realizar un manipulador de alta sensibilidad [6]. Actualmente, se utiliza deep RL para el entrenamiento de movimientos ágiles en robots cuadrúpedos [7].

En el caso de la teoría de juegos, se mostraron algunos ejemplos principales en la sección 1.1, en la cual los ejemplos mostrados no involucraban sistemas físicos reales. Sin embargo, en [8] recientemente realizó el diseño e implementación del juego tres en raya en el Robot IRB120 utilizando técnicas de RL, específicamente el algoritmo SARSA y Q-*learning*, lo cual demuestra que es posible utilizar este sistema físico con un sistema físico.

5. Conclusiones

- Se logró demostrar que con suficiente entrenamiento, un agente es capaz de utilizar el aprendizaje por refuerzo para dominar un juego simple como Tic Tac Toe. Nuestro agente ha utilizado la ecuación de Bellman y el algoritmo Q-learning para mejorar su estrategia de juego.
- Una gran ventaja que se tiene al trabajar con algoritmos de aprendizaje por refuerzo es el hecho de que no es necesario realizar una programación directa con todos los posibles movimientos en el juego.
- Es importante tener cuidado con los hiperparámetros definidos para el entrenamiento, principalmente con las recompensas, ya que, una mala definición de recompensas provocará que nuestro agente no logre aprender de manera correcta porque no sabría diferenciar cuando está tomando las mejores decisiones.

Referencias

[1] K. Shao, Z. Tang, Y. Zhu, N. Li, and D. Zhao, "A survey of deep reinforcement learning in video games," 2019.

[2] O. Vinyals, T. Ewalds, S. Bartunov, P. Georgiev, A. S. Vezhnevets, M. Yeo, A. Makhzani, H. Kuttler, J. Agapiou, J. Schrittwieser, J. Quan, S. Gaffney, S. Petersen, K. Simonyan, T. Schaul, H. van Hasselt, D. Silver, and T. Lillicrap, "Starcraft ii: A new challenge for reinforcement learning," 2017.

[3] S. Mahadevan and J. Connell, "Automatic programming of behavior-based robots using reinforcement learning," *Artificial Intelligence*, vol. 55, no. 2, pp. 311–365, 1992.

[4] V. Gullapalli, J. A. Franklin, and H. Benbrahim, "Acquiring robot skills via reinforcement learning," *IEEE Control Systems*, vol. 14, pp. 13–24, 1994.

[5] J. A. Bagnell and J. G. Schneider, "Autonomous helicopter control using reinforcement learning policy search methods," *Proceedings 2001 ICRA. IEEE International Conference on Robotics and Automation (Cat. No.01CH37164)*, vol. 2, pp. 1615–1620 vol.2, 2001.

[6] K. D. Katyal, "In-hand robotic manipulation via deep reinforcement learning," 2016.

[7] J. Tan, T. Zhang, E. Coumans, A. Iscen, Y. Bai, D. Hafner, S. Bohez, and V. Vanhoucke, "Sim-to-real: Learning agile locomotion for quadruped robots," 2018.

[8] S. V. Rabasco, *Diseño e implementación del juego 3 en raya mediante técnicas Reinforcement Learning en el Robot IRB120*. PhD thesis, Departamento de Ingeniería Electrónica y Automatización Industrial, Septiembre 2021.