ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA (ICAI)

Máster en Big Data: Tecnología y Analítica Avanzada

MACHINE LEARNING II: REINFORCEMENT LEARNING

Jorge Ayuso Martínez

Francisco Javier Gisbert Gil

Carlota Monedero Herranz

Diego Sanz-Gadea Sánchez

José Manuel Vega Gradit

Madrid
Mayo 2023

Índice

1. Introducción

En este proyecto hemos desarrollado una serie de agentes capaces de jugar al Blackjack, un juego de cartas clásico, emplando una amplia variedad de técnicas de Aprendizaje por Refuerzo (Reinforcement Learning, RL).

El Aprendizaje por Refuerzo es una disciplina dentro del campo del aprendizaje automático que se centra en entrenar agentes para enseñarles a tomar decisiones en un entorno con el objetivo de maximizar una señal de recompensa. Para implementar el juego y probar nuestros agentes, utilizaremos el entorno de la librería de OpenAI Gymnasium disponible en el siguiente enlace. El código del proyecto, así como los resultados obtenidos se encuentran el siguiente repositorio de GitHub. Junto a la memoria del trabajo se entrega un breve vídeo de menos de 1 minuto de duración con los resultados de los algoritmos ensayados.

En resumen, este proyecto demostrará de manera práctica cómo se pueden emplear las técnicas de aprendizaje por refuerzo para entrenar a un agente a jugar un juego como el Blackjack. Al comparar el rendimiento de diferentes algoritmos, se obtendrá información sobre las fortalezas y debilidades de cada técnica, así como su idoneidad para resolver distintos tipos de problemas.

1.1. El juego del BlackJack

El BlackJack es un popular juego de cartas cuyo objetivo es tratar de obtener una suma lo más próxima a 21, sin sobrepasar nunca este número. Al inicio de cada ronda, el *crupier* reparte dos cartas visibles al jugador, ambas boca arriba, y dos cartas para sí mismo, una de las cuales se encuentra boca abajo. El jugador puede decidir entre *"Pedir"* una carta adicional o "*Plantarse*", manteniendo el valor actual de su mano. El juego termina cuando el jugador o el *crupier* obtienen un valor de mano de 21 o lo más cercano posible sin superarlo.

En nuestro entorno, se mostrará la suma del valor de las cartas del jugador y una de las cartas del crupier, que se encuentra boca arriba. Además, se informará si el jugador tiene un as utilizable, cuyo valor puede ser 1 u 11 en función del valor del resto de cartas en su mano.

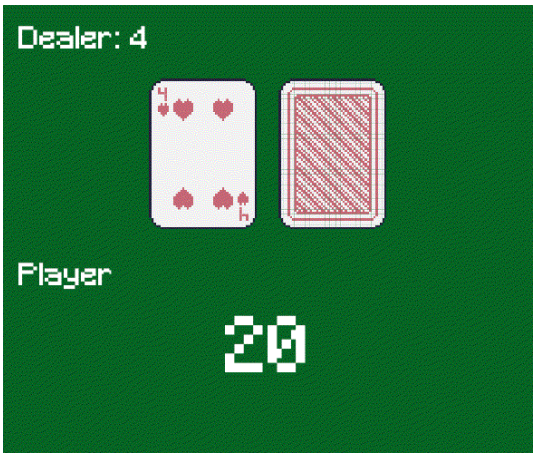

Figura 1: Ejemplo del entorno de nuestro Agente en Gymnasium.

1.2. Técnicas empleadas

En este trabajo compararemos el rendimiento de varias técnicas de Aprendizaje por Refuerzo, entre ellas Q-Learning, Deep Q-Network (DQN), Policy Gradient y Advantage Actor-Critic (A2C):

- **Q-Learning** es un algoritmo que permite la elaboración de una tabla de valores Q para cada uno de los pares acción-estado del entorno en el que se encuentra el agente, algo que consigue explorando el espacio del juego y aprendiendo de las recompensas que recibe del mismo. Adicionalmente, se emplea una política *epsilon-greedy* que permite equilibrar el compromiso entre la *exploración* y la *explotación* del conocimiento que se va adquiriendo, permitiendo al agente tomar ocasionalmente una acción aleatoria con el fin de explorar nuevas posibilidades y con ello reducir la probabilidad de caer en mínimos locales.
- ***Deep Q-Network*** (**DQN**) es un algoritmo de RL basado en *Deep Learning* que utiliza una Red Neuronal para aproximar la función Q. Combina Q-Learning con un *buffer replay* de experiencias con el fin de mejorar la eficiencia de la muestra y la estabilidad durante el entrenamiento.
- ***Policy Gradient*** es otro algoritmo de Aprendizaje por Refuerzo que, a diferencia de los algoritmos anteriormente presentados, se centra en tratar de aproximar la mejor política de manera directa a través de la maximización de la recompensa acumulada esperada. En este tipo de modelo, los parámetros de la Red Neuronal se optimizan mediante el empleo de *ascenso del gradiente*, lo que significa que da pequeños pasos en la dirección que más mejora la recompensa esperada.
- ***Advantage Actor-Critic*** (**A2C**) es un algoritmo que permite combinar la estimación de la política óptima como la aproximación de la función valor. El algoritmo básico de *Actor-Critic* se compone de dos redes neuronales: la primera de ellas (el *actor*), se encarga de tomar acciones dentro del entorno en base al estado en el cual se encuentra en cada momento. Por otra parte, el segundo aproximador (el *crítico*) aprende a estimar el valor de las acciones tomadas por el actor a través de computar sobre ellas la función valor, de manera que puede proporcionar un *feedback* directo al mismo sobre si sus acciones son óptimas o no. La mejora adicional que presenta el algoritmo de Advantage Actor Critic se centra en la descomposición del valor Q en la función valor y el valor de la ventaja A que este supone, de forma similar a como se presenta en algoritmos de *Dueling DQN* , de forma que el crítico aprenda los valores de la ventaja, reduciendo así la varianza del actor y estabilizando el modelo.

1.3. Estructura de la entrega

Nuestra entrega consta de:

- **Memoria:** Este documento
- **Vídeo:** 'results_video.gif', un vídeo de menos de 1 minuto de duración con los resultados de los algoritmos ensayados.
- **Repositorio:** La carpeta 'blackjack_rl' contiene el repositorio de GitHub con el código elaborado para el entrenamiento de los modelos.

2. Random Agent

En primer lugar se ha procedido a la creación de un **Random Agent**, el cual toma una acción aleatoria en cada uno de los pasos de la partida. El objetivo de este agente es servir como punto de referencia y comparación para el resto de algoritmos entrenados a continuación, de forma que podamos saber si éstos están suponiendo una mejora significativa o no frente a la estrategia de tomar una acción aleatoria dentro del espacio de acciones posibles.

Dado que en este caso no hay un proceso de entrenamiento del algoritmo, tan sólo se muestran los resultados relativos a la evaluación del mismo en un total de 1000 partidas.

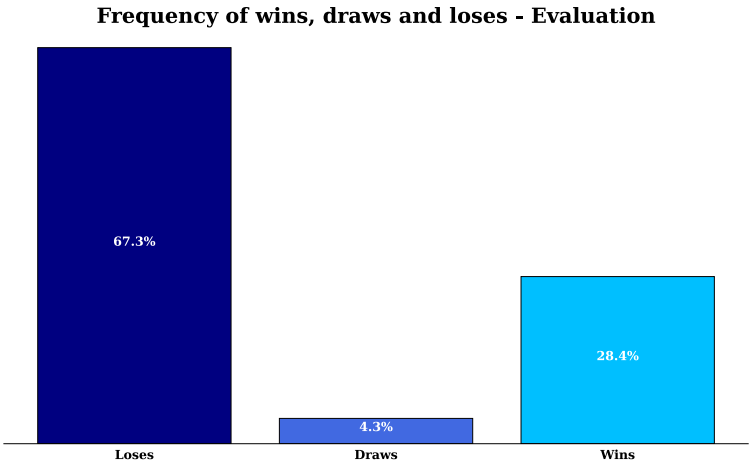

Figura 2: Estadísticas de Random Agent durante el proceso de evaluación.

Tal y como puede observarse en la figura 2, el agente pierde en la mayoría de ocasiones, saliendo victorioso en apenas el 30 % de los casos. Resulta interesante observar como el agente valora poco o nada la posibilidad de empatar, siendo puramente un suceso aleatorio para él.

Los algoritmos presentados a continuación pretenden aprender de una forma u otra una política que permita mejorar estos resultados, aumentando el número de victorias y empates así como reduciendo el número de derrotas.

3. Q-Learning

3.1. Introducción

El algoritmo de Q-learning es un método de aprendizaje por refuerzo (*Reinforcement Learning*) que permite a un agente aprender a tomar decisiones óptimas en un entorno desconocido, a través de la iteración de experiencias y actualización de una función de valor conocida como función Q.

La función Q es una tabla que representa el valor esperado de tomar una acción a en un estado s, y se actualiza iterativamente utilizando la ecuación de Q-learning. Esta ecuación se basa en la idea de que el valor de una acción en un estado dado debe ser igual a la suma de la recompensa inmediata obtenida al realizar esa acción, más la suma de los valores esperados de todas las acciones posibles en el siguiente estado ponderadas por su probabilidad de ocurrencia:

$$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha(r + \gamma \max_{a'} Q(s',a')) \tag{1}$$

Donde:

- $Q(s,a)$ es el valor de la acción a en el estado s.
- α es la tasa de aprendizaje (*learning rate*) que determina la proporción en la que el nuevo conocimiento reemplaza al viejo.
- r es la recompensa obtenida al realizar la acción a en el estado s.
- γ es el factor de descuento (*discount factor*) que determina la importancia de las recompensas futuras.
- s' es el siguiente estado que se alcanza después de realizar la acción a en el estado s.
- $\max_{a'} Q(s',a')$ es el valor máximo de la función Q en el siguiente estado s' para todas las acciones posibles a'.

El proceso de aprendizaje en Q-learning consiste en que el agente interactúa con el ambiente realizando acciones y observando las recompensas recibidas y los estados resultantes. En cada paso, el agente actualiza la tabla Q utilizando la ecuación de Q-learning y toma la acción con el valor Q más alto en el estado actual. Con el tiempo, la tabla Q se actualiza de tal manera que el agente aprende a tomar decisiones óptimas en cualquier estado dado.

El algoritmo de Q-learning utiliza un enfoque *off-policy*, lo que significa que el agente aprende una política óptima sin necesidad de seguir una política específica en el ambiente. En otras palabras, el agente puede explorar el ambiente de manera aleatoria mientras aprende una política óptima.

En nuestro juego, la matriz Q es una estructura de datos que se utiliza para almacenar los valores de la función Q, que indica el valor esperado de recompensa para cada acción en un estado determinado. En particular, en nuestro caso, los estados se representan mediante tuplas de la forma (*sum_cartas_jugador*, *sum_cartas_crupier*, *usable_ace*), donde *sum_cartas_jugador* y *sum_cartas_crupier* son las sumas de las cartas del jugador y del crupier respectivamente, y *usable_ace* es un valor binario que indica si el jugador tiene un As que se puede usar como 1 o como 11.

Para cada estado, la matriz Q almacena los valores de la función Q para cada posible acción (pedir carta o plantarse). En nuestro caso, como solo hay dos acciones posibles en cada estado,

cada fila de la matriz Q es una lista de dos elementos que representan los valores de la función Q para cada acción. Por ejemplo, la fila correspondiente al estado (13, 8, True) podría ser [4,68, 6,99], lo que indica que el valor esperado de recompensa para pedir carta es 6,99 y para plantarse es 4,68.

En nuestro caso, al iniciar el entrenamiento del agente inicializaremos los valores de la matriz Q de forma aleatoria. A medida que el agente juega partidas y recibe recompensas, se actualizarán los valores de la matriz Q utilizando el algoritmo de Q-learning. Con el tiempo, los valores de la matriz Q convergen a los valores óptimos, lo que permite al agente tomar decisiones óptimas en cada estado.

3.2. Entrenamiento

El código relativo al entrenamiento y evaluación del modelo se puede encontrar en el archivo de `notebooks/Q_learning.ipynb`. La matriz Q, que se utiliza para almacenar los valores de recompensa estimados para cada par estado-acción. En nuestro caso, la matriz Q se actualiza en cada episodio del juego utilizando la fórmula de actualización del Q-Learning. Después del proceso de entrenamiento, evaluamos el rendimiento del agente en términos de porcentaje de victorias, empates y derrotas obtenidas por el agente en un conjunto de juegos. Esto nos dará una idea de la capacidad del agente para aprender y mejorar su estrategia de juego a lo largo del proceso de entrenamiento.

En primer lugar entrenamos el `random_agent` de Q-Learning donde se inicializan los valores aleatorios de la matriz Q y directamente se pasa a jugar con 1 episodio de entrenamiento, evaluándose con un número de partidas de 20000. Se obtienen los siguientes resultados:

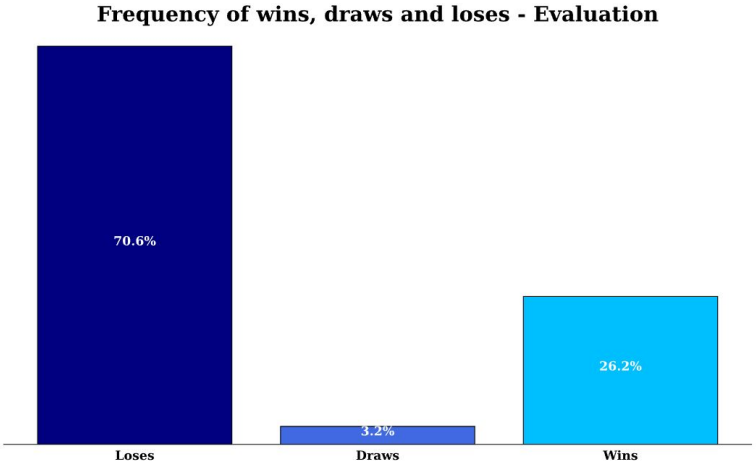

Figura 3: Estadísticas del algoritmo Random Q-Learning durante el proceso de evaluación.

Como se puede observar con esas 20000 iteraciónes de evaluación se obtiene un 70,6 % de derrotas, 3,2 % de empates y 26,2 % de victorias. Este Random Agent da lugar a resultados peores que el anterior. Podemos mejorarlo entrenando el algoritmo en las sucesivas iteraciónes de la matriz Q, como se hace a continuación:

Los paramétros de entrenamiento utilizados son `learning_rate`=0.1, `discount_factor`=0.90, `exploration_rate`=0.90, con 100000 episodios, obteniendo los siguientes resultados:

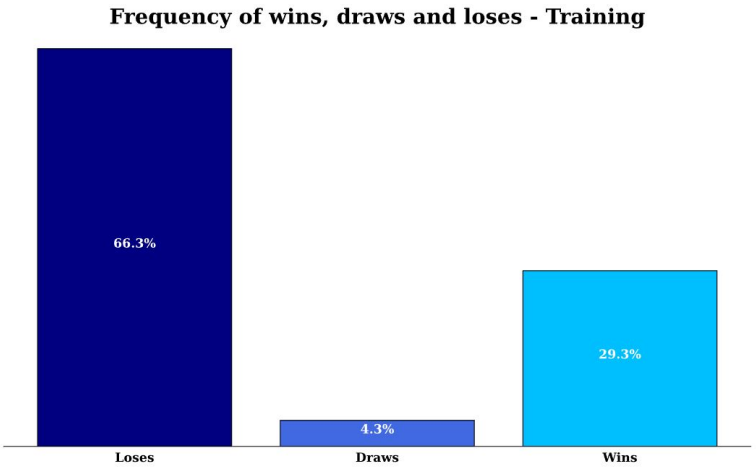

Figura 4: Estadísticas del algoritmo Q-Learning durante el proceso de entrenamiento.

Se observa que los resultados mejoran reduciendo el porcentaje de derrotas a 66,3 %, empates con un 4,3 % y victorias en un 29,3 %.

Una vez entrenado, se ha evaluado la capacidad del agente en un conjunto de otras 100000 partidas, dand lugar a los siguientes resultados:

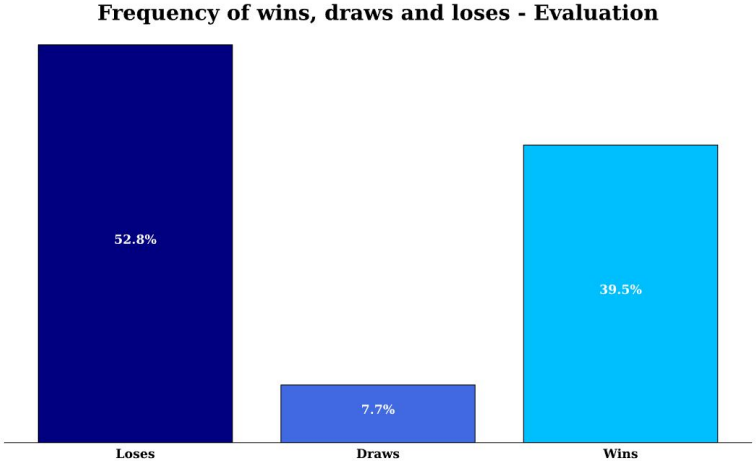

Figura 5: Estadísticas del algoritmo QLearning durante el proceso de evaluación.

Se puede observar que en la evaluación el modelo ha aprendido más, dando lugar a mejores resultados que en el entrenamiento y en el random agent, con un porcentaje de derrotas del 52,8 %, empates del 7,7 % y victorias de 39,5 %.

La matriz Q resultante del entrenamiento tiene la siguiente forma, donde dada la extensión de la misma solo podemos enseñar una pequeña parte:

```
defaultdict(<function src.agents.QAgent.__init__.<locals>.<lambda>()>,
            {(13, 10, False): array([0.76463126, 0.66313768]),
             (16, 9, False): array([-0.44455471, -0.01804687]),
             (23, 9, False): array([-0.00096045, -0.00063258]),
             (14, 1, False): array([-0.43915105,  0.34816777]),
             (20, 1, False): array([ 1.39730297, -0.7031031 ]),
             (30, 1, False): array([-0.00042219,  0.00034808]),
             (13, 6, False): array([1.33415058, 1.02084644]),
             (18, 7, False): array([4.24223229, 1.05928921]),
             (18, 5, False): array([1.79298522, 0.25648158]),
             (19, 5, False): array([ 4.93411337, -0.13025343]),
             (29, 5, False): array([6.88501550e-05, 5.44177725e-04]),
             (6, 10, False): array([0.66926298, 0.86872923]),
             (19, 2, False): array([4.13092295, 1.23349332]),
             (29, 2, False): array([-0.00015632, -0.00077888]),
             (18, 10, True): array([0.42766954, 1.9766678 ]),
             (12, 10, False): array([0.17304349, 0.64211106]),
             (17, 10, False): array([-0.37380935,  0.51221263]),
             (15, 1, False): array([-0.55941006,  0.03006493]),
```

Figura 6: Matriz Q de entrenamiento Q-Learning tras 100000 episodios

Se puede ver cómo conforme crece el valor de las cartas en mano del jugador, a recompensa obtenida por plantarse también crece. Veamos algunas filas de la matriz Q en detalle para comprender mejor sus valores.

- (12, 10, False): array([0.17, 0.64]): Esto significa que si el jugador tiene una mano con un valor de 12 y no tiene un as usable, y el crupier muestra una carta con un valor de 10, el valor esperado de recompensa para plantarse es 0,17 y para pedir carta es 0,64.
- (19, 2, False): array([4.12, 1.23]): Esto significa que si el jugador tiene una mano con un valor de 19 y el crupier muestra una carta con un valor de 2, el valor esperado de recompensa para plantarse es 4,12 y para pedir carta es 1,23.

Estos casos son bastante evidentes y las decisiones a las que ha llegado el modelo serían las mismas que tomaría un jugador que conociera la mecánica del juego. Por lo tanto podemos ver como el mecanismo de aprendizaje es el correcto.

3.3. Conclusión

Finalmente, con respecto al algoritmo de Q-learning se ha conseguido una mejora en relación al Agente Aleatorio inicial, tanto el normal, como el de Q-Learning con entrenamiento de un episodio con incialización aleatoria de la matriz. Las mejoras no son excesivas, dado el tipo de juego en el que el Crupier tiene la estadistica a su favor. Como es de esperar, el número de derrotas a nivel del proceso de entrenamiento es superior a las observadas durante el proceso de evaluación, puesto que inicialmente el agente desconoce cómo modelar la política de la forma más adecuada. El algoritmo Q-Learning es un algoritmo rápido el cuál nos ha permitido computar 100000 episodios en cuestión de segundos, por lo que si se busca un algoritmo con tiempos muy buenos de respuesta y con mejoras respecto al agente aleatorio, podría ser una buena opción.

4. DQN

4.1. Introducción

El algoritmo **DQN** es un método de aprendizaje por refuerzo que se basa en el empleo de redes neuronales para aprender una función capaz de aproximar el Q-valor de cada uno de los estados. A diferencia de métodos más tradicionales, como el Q-learning, donde cada pareja de estado-acción es asignada un valor de manera tabular, DQN se centra en encontrar un aproximador lo suficientemene bueno como para poder realizar esta estimación, sin necesidad de emplear métodos tabulares que almacenen esta información de manera estructurada. Esto resulta especialmente útil en casos en los cuales el número de estados y posibles acciones es demasiado elevado como para poder realizar un mapeo concreto de sus combinaciones, así como en entornos continuos, donde el valor añadido de encontrar una función generalizadora es evidente.

4.2. Entrenamiento

El código relativo al entrenamiento y evaluación del modelo se puede encontrar en la ruta `notebooks/DQN.ipynb`. La red neuronal entrenada se compone de una capa de entrada de 128 neuronas, dos capas ocultas de 64 y 32 neuronas y una capa de salida de 2 neuronas. Cada una de estas neuronas corresponde a una de las dos acciones posibles que el modelo puede tomar en este entorno, y el valor devuelto por la red es el correspondiente a la estimación del valor Q para cada par estado-acción dado el estado en el que se encuentra el agente en dicho momento.

Para llevar a cabo el entrenamiento se ha seleccionado un factor de descuento $\gamma = 0{,}8$, y un tamaño de *batch* de 200. Mencionar asimismo que el agente incorpora el método de ϵ***-decay*** a la hora de realizar el entrenamiento, inicializándose su valor a $\epsilon = 0{,}01$ y descendiendo de manera progresiva a medida que aumenta el número de episodios. El número total de iteraciones realizadas es de 1000.

Tras el proceso de entrenamiento se ha evaluado el porcentaje de victorias, empates y derrotas obtenidas por el agente.

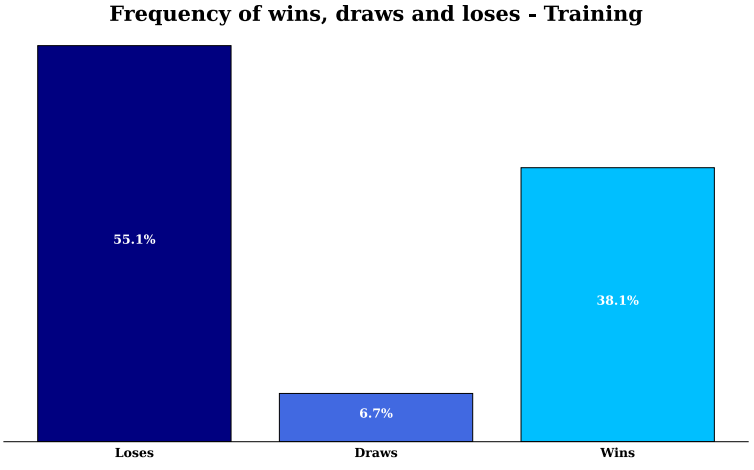

Figura 7: Estadísticas del algoritmo de DQN durante el proceso de entrenamiento.

Como se puede observar, con tan solo 1000 iteraciones de entrenamiento se ha conseguido una reducción bastante significativa del porcentaje de derrotas respecto del agente aleatorio, pasando

de casi un 67 % a un 55 %, lo cual se ve reflejado en un aumento considerable del porcentaje tanto de victorias (28 % a 38 %) y de empates (4 % a 7 %).

Una vez entrenado, se ha evaluado la capacidad del agente en un set de 1000 partidas de evaluación, dando los resultados mostrados a continuación:

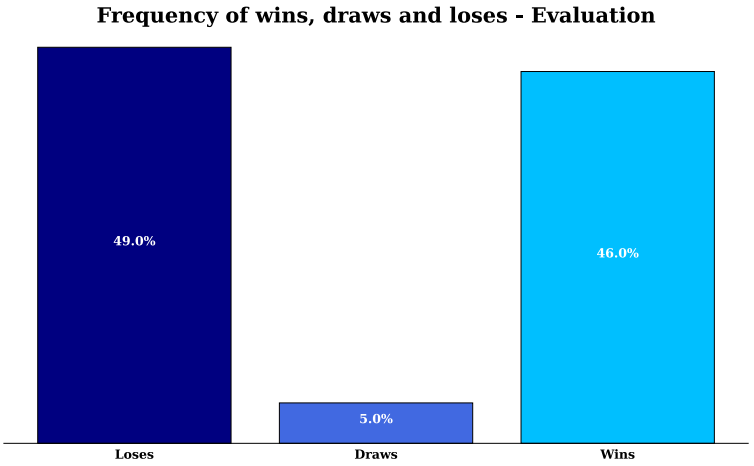

Figura 8: Estadísticas del algoritmo de DQN durante el proceso de evaluación.

El desempeño del agente es bastante bueno, consiguiendo ganar o empatar en más del 50 % de las partidas.

4.3. Conclusión

Como puede observarse, se ha conseguido una gran mejora con respecto del agente aleatorio en lo que respecta tanto al proceso de aprendizaje como al proceso de evaluación del agente dentro del juego, así como con respecto del agente de Q-Learning.

5. Policy Gradient

5.1. Introducción

Los métodos usados hasta ahora eran lo que se conoce como ***Value Based methods***. Es decir, métodos donde la política no se obtiene directamente, sino que se centran en estimar una función valor que sea capaz de guiar la toma de decisiones del agente en cada uno de los estados en los que se encuentre. Frente a ellos, los métodos denominados como ***Policy Based Methods*** son aquellos en los que se obtiene una estimación directa de la política a llevar a cabo.

Una de las principales ventajas del empleo de este tipo de algoritmos frente a los algoritmos *Value Based* es que son capaces de generar una distribución de probabilidad de las acciones, permitiéndoles aprender políticas estocásticas. Esto es esencial en el caso de entrenar agentes para que aprendan a jugar un juego, donde es fundamental que este tenga un cierto grado de aleatoriedad. Si la política a seguir por el agente fuera determinista (algo que ocurre en los *Value Based* methods, donde siempre se escoge aquella acción cuyo Q-valor es mayor), el agente sería totalmente predecible, un comportamiento opuesto al necesario para poder ganar juegos donde hay un alto componente de incertidumbre e impredecibilidad. Este punto si parece ser de especial importancia, dado que el problema a resolver por el agente es aprender a jugar a un juego (black-jack) donde aplicar políticas deterministas haría que el agente fuera predecible, y en consecuencia, no fuera un buen jugador de black-jack.

Sin embargo, este tipos de métodos también tienen alguna serie de puntos negativos, entre los que se encuentran:

- Tendencia a converger a un mínimo local y no al mínimo global, obteniendo por tanto una solución subóptima. Esto puede ser un punto relevante, siendo necesario comparar si la solución obtenida (la cual no podemos asegurar que sea un mínimo local o el global), es mejor o no que las soluciones obtenidas por el resto de algoritmos estudiados.
- Evaluar una política es típicamente ineficiente y altamente variable.

En nuestro caso, se ha entrenado un agente con el algoritmo de *Reinforce*, introducido en 1992 por Ronald Williams. Concretamente, al igual que en los casos anteriores, se ha definido una clase (`PolicyGradientAgent`) en el fichero `src/agents.py`, que hereda de la clase padre `Agent`.

5.2. Entrenamiento

El código relativo al entrenamiento y evaluación del modelo se puede encontrar en la ruta `notebooks/Policy_Gradient.ipynb`. Para entrenar el agente, en este caso se ha seleccionado un factor de descuento $\gamma = 0{,}99$. Adicionalmente, se ha entrenado el agente durante un total de 1000 iteraciones.

Tras el proceso de entrenamiento se ha evaluado el porcentaje de victorias, empates y derrotas obtenidas por el agente. En la gráfica que se muestra a continuación se pueden apreciar dichos resultados en detalle:

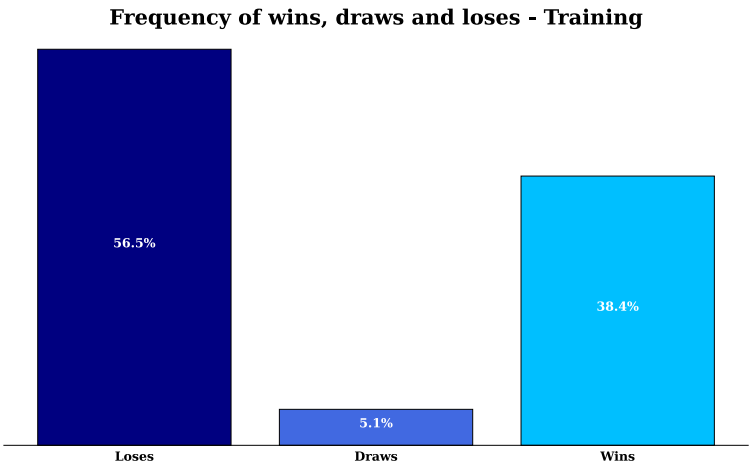

Figura 9: Estadísticas del algoritmo de Policy Gradient durante el proceso de entrenamiento.

Una vez el agente ha sido entrenado, se ha evaluado su capacidad jugando un total de 1000 partidas, y donde se han obtenido los resultados que se muestran a continuación:

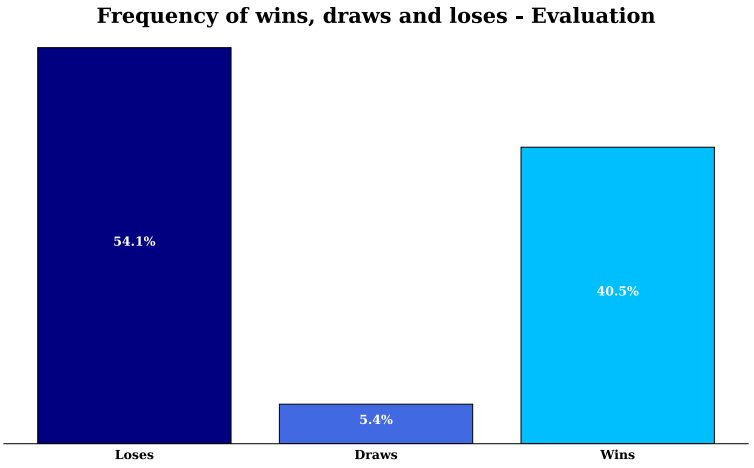

Figura 10: Estadísticas del algoritmo de Policy Gradient durante el proceso de evaluación.

5.3. Conclusión

Como se puede apreciar, existe una pequeña mejora en el proceso de evaluación con respecto al proceso de entrenamiento, dado que en el proceso de entrenamiento el modelo inicialmente no conoce cual es la política óptima, mientras que en el proceso de evaluación el algoritmo ha alcanzado la convergencia inicialmente, y conoce cual es la mejor política (si es que ha llegado al mínimo global, lo cual no podemos asegurar).

6. Advantage Actor-Critic (A2C)

6.1. Introducción

El último algoritmo evaluado en este proyecto es el ***Advantage Actor Critic*** (**A2C**), un algoritmo que permite combinar las ventajas de los métodos *Value Based* con aquellas de los métodos *Policy Based*. Como ya se comentó anteriormente, el algoritmo se compone de dos redes neuronales que interactúan entre sí de manera directa durante el proceso de entrenamiento:

1. **Actor**: El actor es la red encargada de realizar la toma de decisiones acerca de las acciones que han de llevarse a cabo en cada momento y dado el estado del agente dentro del entorno, constituyendo así el modelo *Policy Based*.

2. **Crítico**: El crítico es la red encargada de evaluar la calidad de las acciones tomadas por el actor en cada momento, ayudando de forma directa en el proceso de entrenamiento del mismo. A diferencia de la implementación básica del algoritmo Actor-Critic, en la cual el crítico trata de estimar el valor Q del par estado-acción, en el algoritmo A2C se realiza una descomposicón de este valor en dos partes: la función valor para el estado ($V(s)$) y la ventaja de tomar la acción a seleccionada por el actor en dicho estado ($A(s, a)$). El crítico se centra en la estimación del valor de la función ventaja, la cual es capaz de capturar de manera más precisa el valor real de la toma de la acción, puesto que implica su comparación con el resto de acciones posibles que hubiera podido tomar el actor en el estado s. De esta forma, el proceso de entrenamiento de la red que actúa como actor es mucho mas estable y robusto, lo que ayuda a una convergencia más rápida hacia valores óptimos.

6.2. Entrenamiento

El código relativo al entrenamiento y evaluación del modelo se puede encontrar en el archivo de `notebooks/A2C.ipynb`. Para llevar a cabo el entrenamiento se ha seleccionado un factor de descuento = 0.8 y un número de iteraciones de 1000. Tras el proceso de entrenamiento se ha evaluado el porcentaje de victorias, empates y derrotas obtenidas por el agente.

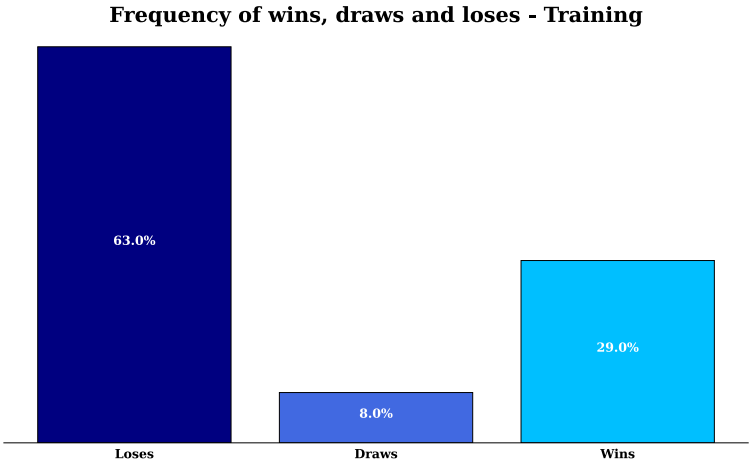

Figura 11: Estadísticas del algoritmo de A2C durante el proceso de entrenamiento.

Como se puede observar, con tan solo 1000 iteraciones de entrenamiento se ha conseguido una reducción bastante significativa del porcentaje de derrotas respecto del agente aleatorio.

Una vez entrenado, se ha evaluado la capacidad del agente en un set de 1000 partidas de evaluación, dando los resultados mostrados a continuación:

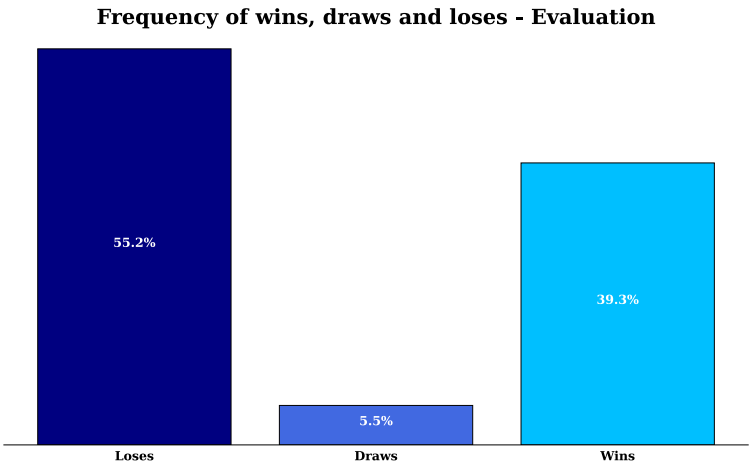

Figura 12: Estadísticas del algoritmo de A2C durante el proceso de evaluación.

6.3. Conclusión

Se puede observar un rendimiento adecuado, aunque este apenas es ligeramente superior al del agente entrenado con Q-Learning. Sería necesario explorar más este método, ya sea entrenando el modelo con un mayor número de iteraciones para explorar el espacio de acciones en más profundidad o probando distintos parámetros en las redes del actor y el crítico con el fin de llegar a una política más óptima.

7. Conclusiones finales del proyecto

7.1. Idoneidad de los algoritmos al problema

Antes de analizar el rendimiento de nuestros modelos, es necesario repasar la idoneidad de cada uno de los métodos empleados para el proyecto:

- En primer lugar, es importante destacar que Q-Learning es un algoritmo sencillo y fácil de entrenar. Sin embargo, es importante tener en cuenta que es un algoritmo determinístico, lo que significa que una vez que encuentra la política óptima, dejará de explorar el espacio de acciones y se limitará a seguirla. En el caso del Blackjack, donde existe una parte no determinista debido a que solo conocemos una de las cartas del crupier, esto puede resultar una espada de doble filo. Aunque tengamos la certeza de que siempre jugaremos de manera óptima, no podemos asegurar que esta sea verdaderamente la mejor opción, ya que desconocemos las siguientes cartas que se revelarán. Para hacer una analogía con el juego, podríamos decir que Q-Learning representa a un jugador conservador, mientras que otros algoritmos de naturaleza estocástica o aleatoria pueden representar a un jugador que toma más riesgos. Por lo tanto, es esperable que se obtengan unos resultados razonables pero que no se alcance el mejor rendimiento de entre todas las técnicas.
- En cuanto a Policy Gradient y Advantage Actor-Critic, ambos métodos tienen un enfoque estocástico en contraste con Q-Learning, lo que significa que pueden ser más adecuados para representar a un jugador más arriesgado. Por lo tanto, es razonable esperar que puedan lograr mejores resultados. Sin embargo, es importante tener en cuenta que estos métodos están mejor indicados en entornos continuos, y nuestro problema es un entorno discreto, por lo que su ventaja se ve disminuida. Además, debido a que estos métodos aprenden directamente la mejor política, es posible que requieran un tiempo de entrenamiento más prolongado y modelos más complejos. Debido a la limitación de tiempo y con el fin de evaluar todos los métodos en igualdad de condiciones, no hemos podido explorar todo el espacio de soluciones disponible.
- DQN representa un buen compromiso entre algoritmos conceptualmente menos complejos, como Q-Learning, donde se estima una función de valor y el tiempo de entrenamiento es menor que en los algoritmos basados en política. No obstante, DQN es más complejo que Q-Learning, ya que utiliza una red neuronal en lugar de una tabla de valores.

7.2. Análisis de los resultados

A continuación se presenta una tabla comparativa con los resultados de evaluación de los distintos métodos:

Modelo	% Victorias	% Empates	% Derrotas
Q-Learning	38.6 %	7.9 %	53.5 %
Advantage Actor-Critic	39.3 %	5.5 %	55.2 %
Policy Gradient	40.5 %	5.4 %	54.1 %
Deep Q-Network	46.0 %	5.0 %	49.0 %

Tabla 1: Resultados de las técnicas aplicadas.

En cuanto a los resultados obtenidos, en general, se observó que todos los modelos aprendieron a jugar al Blackjack de manera razonable. Sin embargo, hubo algunas diferencias en su rendimiento:

1. Q-Learning tuvo un desempeño ligereamente inferior en comparación con los otros modelos, lo que era esperable dada su naturaleza conservadora.

2. Si bien Policy Gradient y A2C mejoran a Q-Learning, no se observa una diferencia significativa con Q-Learning, especialmente en porcentaje de derrotas. Sería necesario explorar más estos métodos con el fin de llegar a una política más óptima.

3. El modelo que mejor rendimiento exhibe es DQN.

En resumen, los resultados obtenidos muestran que los modelos de aprendizaje reforzado tienen la capacidad de aprender a jugar al Blackjack de manera razonable, pero con algunas diferencias en su rendimiento. Los modelos más sofisticados, como Policy Gradient, A2C y DQN, muestran un mejor rendimiento que el método más básico de Q-Learning. En particular, DQN logra el mejor rendimiento en términos de porcentaje de victorias y derrotas, lo que sugiere que su arquitectura de red neuronal y su enfoque de aprendizaje basado en experiencia son altamente efectivos en este contexto. No obstante, es importante tener en cuenta que todavía hay espacio para mejoras en la eficacia y eficiencia de los modelos entrenados, y se requiere una mayor exploración para lograr un rendimiento óptimo en esta tarea.