Práctica

Implementación de un agente para la robótica espacial

por

Ander Elkoroaristizabal

Índice

1. Introducción: el entorno Lunar-Lander

Estamos trabajando sobre un problema de robótica espacial y en particular queremos solucionar el problema de aterrizaje propio, por ejemplo, de drones autónomos. Para ello, se elige **Lunar-Lander** [1, 2] como entorno simplificado.

El entorno Lunar-Lander consiste en una nave espacial que debe aterrizar en un lugar determinado del campo de observación. El agente conduce la nave y su objetivo es conseguir aterrizar en la pista de aterrizaje, coordenadas $(0, 0)$, y llegar con velocidad 0.

La nave consta de tres motores (izquierda, derecha y el principal que tiene debajo) que le permiten ir corrigiendo su rumbo hasta llegar a destino.

Las acciones que puede realizar la nave (el espacio de acciones) son discretas.

Las recompensas obtenidas a lo largo del proceso de aterrizaje dependen de las acciones que se toman y del resultado que se deriva de ellas:

- Desplazarse de arriba a abajo, hacia la pista de aterrizaje, puede resultar en $[+100, +140]$ puntos. La recompensa aumenta/decrece cuanto más cerca/lejos está la nave de la pista de aterrizaje, aumenta/decrece cuanto más lento/rápido va la nave, y decrece cuanto más escorada (no horizontal) está.
- Si se estrella o sale por los laterales de la pantalla recibe -100 puntos.
- Si consigue aterrizar y quedarse quieto (velocidad 0), recibe $+100$ puntos.
- El contacto de una pata con el suelo recibe $+10$ puntos (si se pierde contacto después de aterrizar, se pierden puntos).
- Cada vez que enciende el motor principal pierde $-0{,}3$ puntos.
- Cada vez que enciende uno de los motores de izquierda o derecha, pierde $-0{,}03$ puntos.

La solución óptima es aquella en la que el agente, con un desplazamiento eficiente, consigue aterrizar en la zona de aterrizaje (0,0), tocando con las dos patas en el suelo y con velocidad nula. Se considera que el agente ha aprendido a realizar la tarea (i.e.el "juego" termina) cuando obtiene una media de al menos 200 puntos durante 100 episodios consecutivos.

1.1. Exploración del entorno y partida aleatoria

El código utilizado en este apartado está contenido en el *script* `env_exploration.py`, donde:

1. inicializamos el entorno LunarLander-v2,
2. extraemos sus características más relevantes,
3. reproducimos varias partidas aleatorias para analizar visualmente los diferentes escenarios que pueden darse, y las comparamos también con un agente heurístico,
4. guardamos una partida aleatoria como gif , y
5. jugamos 100 partidas aleatorias, para así analizar la recompensa media y el número de pasos por episodio de un agente aleatorio.

A partir de los atributos del entorno se extrae la siguiente información:

- El valor del umbral de recompensa definido en el entorno (el valor a partir del cual un agente se considera que ha resuelto el entorno) es 200,
- El número máximo de pasos por episodio es 1000,
- El rango de las recompensas es $(-\text{ínf}, \text{ínf})$,
- El espacio de acciones consta de 4 acciones, y
- El espacio de observaciones es un vector con 8 elementos.

De la observación de múltiples partidas aleatorias vemos que:

- La posición de la pista de aterrizaje, como ya hemos comentado, no cambia de una partida a otra: siempre está en el centro y a la misma altura. Esto es importante para el agente, dado que como veremos más adelante, las observaciones no contienen información de la posición de la meta.
- Los alrededores de la pista de aterrizaje, en cambio, si que cambian de una partida a otra, como podemos observar en la figura 1. Esto también es importante, dado que las observaciones tampoco incluyen información de los alrededores de la pista de aterrizaje, y como podemos ver en la imagen este puede variar considerablemente, llegando incluso a afectar al rendimiento del agente.
- Como se indica en la documentación del entorno [1] (y también podemos ver en los estados de la figura 1), la dirección y velocidad con la que la nave comienza la partida también varían de una partida a otra.

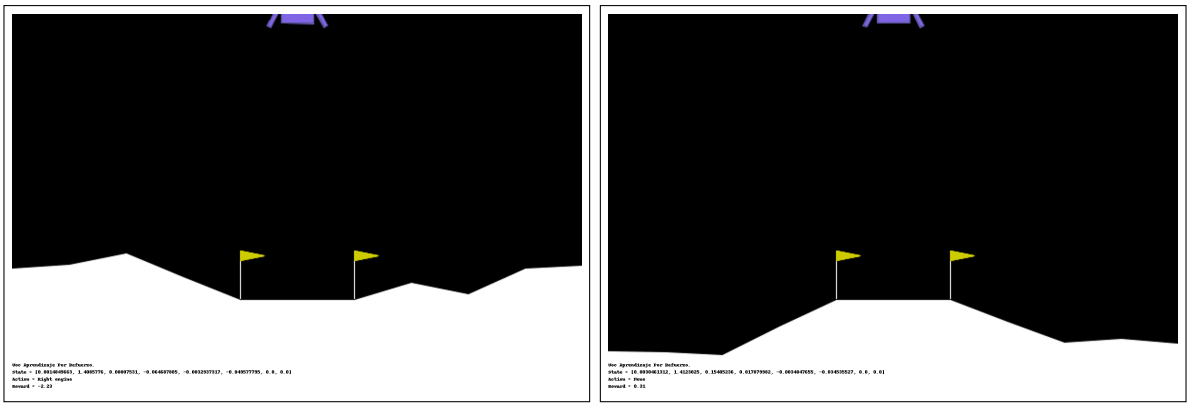

Figura 1: Dos inicializaciones aleatorias del entorno

La figura 1 contiene el primer *frame* de dos ejecuciones aleatorias. Durante la construcción del gif añadimos la información del estado, la acción tomada y la recompensa. Teniendo en cuenta cómo se rigen las recompensas del entorno, donde en cada instante el agente recibe una recompensa positiva al acercarse y decelerar hacia la pista de aterrizaje, y donde aterrizar y estrellarse tienen una gran recompensa (positiva en el primer caso y negativa en el segundo), parece un entorno donde, si los estados que se reciben son lo suficientemente informativos, el agente debería aprender con rapidez. En contraste, en las 100 partidas de un agente aleatorio analizadas se ha visto que un agente aleatorio es completamente incapaz de no estrellarse en este entorno, obteniendo recompensas raramente positivas y con episodios de menos de 150 pasos.

Por último, cabe remarcar las siguiente clasificación heurística sobre la recompensa total de los episodios, que se extrae de la definición de las recompensas dadas y de la observación de múltiples episodios de agentes ya entrenados:

1. Cuando la recompensa acumulada de un episodio es menor que 100 la nave espacial no ha conseguido aterrizar. Puede haberse estrellado, haber salido por los márgenes, o que se haya alcanzado el límite de tiempo sin que la nave aterrice, salga o se estrelle.
2. Cuando la recompensa acumulada de un episodio está entre 100 y 200, la nave ha conseguido aterrizar, pero o no en la pista de aterrizaje, o no mediante un desplazamiento eficiente.
3. Cuando la recompensa acumulada está por encima de los 200 puntos la nave ha conseguido aterrizar en la pista de aterrizaje con un desplazamiento eficiente.

Esta clasificación de las posibles puntuaciones del entorno resulta de especial importancia a la hora de definir los criterios de evaluación de los agentes.

1.2. Explicación de los espacios de observaciones y acciones

El **espacio de observaciones** es 8 dimensional, donde de cada vector:

1. El primer elemento es la coordenada x de la nave, cuyo valor está contenido en el rango $[-1{,}5; 1{,}5]$.
2. El segundo elemento es la coordenada y de la nave, cuyo valor está contenido en el rango $[-1{,}5; 1{,}5]$.
3. El tercer elemento es la velocidad lineal de la nave en el eje x, cuyo valor está contenido en el rango $[-5, 5]$.
4. El cuarto elemento es la velocidad lineal de la nave en el eje y, cuyo valor está contenido en el rango $[-5, 5]$.
5. El quinto elemento es el ángulo de la nave (como de girada está), cuyo valor está contenido en el rango $[-\pi, \pi]$.
6. El sexto elemento es la velocidad angular, cuyo valor está contenido en el rango $[-5, 5]$.
7. El séptimo elemento es un indicador de si la pata izquierda está en contacto con el suelo.
8. El octavo elemento es un indicador de si la pata derecha está en contacto con el suelo.

El **espacio de acciones** es discreto, y consta de 4 opciones: no hacer nada (acción 0), encender el motor izquierdo (acción 1), encender el motor principal (acción 2) y encender el motor derecho (acción 3).

1.3. Criterio de evaluación de los agentes

En este apartado determinamos el criterio que utilizamos para decidir qué agente de los obtenidos es mejor. Para ello nos basamos en lo que para la aplicación del aprendizaje por refuerzo a este entorno creemos tiene más sentido.

Como ya hemos comentado, nuestro objetivo al utilizar este entorno es enseñar a una nave espacial cómo aterrizar en un lugar determinado del campo de observación. Esta nave muy posiblemente se encuentre en un planeta o cuerpo celeste lo suficientemente lejano para que no podamos guiarlo a distancia, razón por la cual necesitamos enseñarle a que lo haga por si misma. Por el mismo motivo la reparación de esta nave es probablemente costosa, o incluso imposible, lo cual nos da un primer y prioritario criterio de evaluación: **nuestro mejor agente es el que más veces aterrice sin estrellarse** durante las pruebas posteriores al entrenamiento.

Además de esto, también es importante que el agente aprenda a aterrizar en la pista de aterrizaje, dado que de lo contrario el agente estaría malgastando un precioso combustible cada vez que tenga que desplegar para acercarse a la pista objetivo. En otras palabras, como segundo criterio más importante tenemos que **nuestro mejor agente debe aterrizar en la pista de aterrizaje el máximo de ocasiones** durante las pruebas posteriores al entrenamiento.

Por último, se considera que el agente debe aprender rápidamente. En otras palabras, como tercer criterio más importante tenemos que **el tiempo de entrenamiento del agente debe ser lo más corto posible**.

2. Agente de referencia: DQN

En la tercera parte de la asignatura ha sido introducido el agente DQN con política *epsilon-gready*, *replay buffer* y *target network*, que resulta ser un buen candidato para la solución del problema de robótica que estamos analizando, visto que permite controlar entornos con un número elevado de estados y acciones de forma eficiente. En este apartado tratamos de resolver el entorno utilizando este agente.

2.1. Implementación

El código específico de este apartado está contenido en el *script* `dqn.py`, en el cual:

1. Definimos la clase `DQN` de la red neuronal.
2. Definimos la clase `DQNAgent` del agente DQN.
3. Inicializamos el entorno.
4. Entrenamos el agente con unos hiperparámetros definidos.
5. Imprimimos las gráficas de recompensas y pérdidas del entrenamiento del agente.
6. Guardamos los pesos de la red.
7. Imprimimos la gráfica de evaluación del agente.
8. Imprimimos el número de episodios que ha resuelto, en los que ha aterrizado y en los que se ha estrellado.
9. Analizamos visualmente algunos episodios del agente.
10. Y guardamos un gif de su comportamiento.

Para la implementación de las clases `DQN` y `DQNAgent` se ha partido del ejemplo del capítulo 9 en el repositorio de la asignatura [3, Cap. 9]. Las dos principales modificaciones de la clase `DQNAgent` vienen dadas por la actualización de la API de `gym` [4], tras la cual los métodos `reset` y `step` devuelven 5 valores, en vez de los 4 de las versiones previas [5], importante a la hora de tener en cuenta el número máximo de pasos por episodio. Además el método `get_action()`, que en los ejemplos forma parte de la clase de la red, es ahora un método del agente.

La red DQN implementada sigue la siguiente arquitectura:

1. Una primera capa completamente conectada (representada en pytorch por `nn.Lineal`) de 256 neuronas y `bias=True`, con activación `ReLU`.
2. Una segunda capa igual: completamente conectada (representada en pytorch por `nn.Lineal`), de 256 neuronas, `bias=True`, y con activación `ReLU`.
3. Una última capa completamente conectada y `bias=True`. Esta será nuestra capa de salida y por lo tanto tendrá tantas neuronas como dimensiones tiene nuestro espacio de acciones (una salida por cada acción posible).

Utilizamos el optimizador Adam para entrenar la red debido a que normalmente muestra una velocidad de convergencia mayor que la de otros optimizadores.

Como implementación del *replay-buffer* utilizamos la clase `experienceReplayBuffer` también presente en el ejemplo del capítulo 9 en el repositorio de la asignatura [3, Cap. 9].

Para hacer reproducible el entrenamiento se fija una semilla aleatoria en las librerías `random`, `Pytorch` y `numpy`, por una parte, y en el entorno y su espacio de acciones, por otra, que además es recreado en cada ocasión.

2.2. Entrenamiento

Los hiperparámetros que impactan el entrenamiento del agente junto con su explicación y el valor utilizado -obtenido mediante una búsqueda en rejilla detallada a continuación- se muestran en la tabla 1. Estos hiperparámetros determinan tanto el *buffer* utilizado como el agente y su entrenamiento. Aunque no se muestre en esta tabla, la arquitectura de la red puede considerarse también un hiperparámetro. Por el contrario, aunque la semilla aleatoria utilizada puede impactar en el entrenamiento, esta no debe considerarse un hiperparámetro.

Hiperparámetro	**Variable**	**Descripción**	**Valor**
Tamaño del *buffer*	MEMORY_SIZE	Número máximo de estados almacenados en el *buffer*	10000
Burn in	BURN_IN	Número de estados aleatorios en el *buffer* antes de empezar a entrenar	1000
Epsilon inicial	INIT_EPSILON	Valor de epsilon al comenzar el entrenamiento del agente	1
Decrecimiento de epsilon	EPSILON_DECAY	Factor de decrecimiento de epsilon en cada episodio	0.985
Epsilon mínimo	MIN_EPSILON	Valor mínimo de epsilon	0,01
Máximo de episodios	MAX_EPISODES	Número máximo de episodios de entrenamiento del agente	1000
Gamma	GAMMA	Valor gamma de la ecuación de Bellman	0.99
Batch size	BATCH_SIZE	Tamaño del *batch* de entrenamiento de la red neuronal	32
Learning rate	LR	Tasa de aprendizaje utilizada en el entrenamiento	0.001
Frecuencia de actualización	DNN_UPD	Número de pasos entre una actualización de la red principal y la siguiente	1
Frecuencia de sincronización	DNN_SYNC	Número de pasos entre dos sincronizaciones de la red principal con la *target*	1000

Tabla 1: Hiperparámetros del aprendizaje del agente DQN

Búsqueda de los hiperparámetros utilizados

Los hiperparámetros mostrados en la tabla 1 son aquellos que dan un mejor agente según el criterio de evaluación de agentes establecido en el apartado 1.3. Para encontrar los hiperparámetros utilizados se ha realizado una búsqueda en rejilla partiendo de las siguientes consideraciones:

1. El tamaño del *buffer* y el *burn in* no son hiperparámetros con un impacto especialmente significativo en los resultados, siempre que se mantengan dentro de un rango con sentido [6], por lo que los fijamos a cantidades que consideramos razonables: 1000 estados de *burn in* (al menos un episodio) y 10000 estados máximos en el *buffer* (10 episodios o más).
2. El epsilon inicial se fija a 1, de forma que el agente comienza el entrenamiento únicamente explorando. El mínimo se fija a 0,01, de forma que al final siga habiendo cierta exploración.
3. Para el decrecimiento de epsilon se ha probado con los valores 0,98, 0,985, y 0,99, con los cuales la fase de exploración (cuando el valor de epsilon es superior al mínimo) dura 228, 305 y 459 episodios respectivamente. Con todos los valores se llega a resolver el entorno, pero decidimos mostrar la búsqueda en rejilla correspondiente al valor 0,985, por ser la que contiene el agente que mejor rendimiento ha dado.
4. El máximo de episodios no es especialmente importante en nuestro caso, dado que dejándolo suficientemente grande el agente suele conseguir resolver el entorno. Lo fijamos a 1000,

de forma que el agente realice tanto la exploración como varios cientos de episodios de explotación.

5. El valor de la constante gamma de la ecuación de Bellman la fijamos a 0,99, dado que la recompensa total del entorno es la relevante, pero las recompensas inmediatas son orientativas y también sirven de cara al objetivo final (además de entrar en la recompensa total). Como ejemplo, con este valor de gamma la recompensa de aterrizar sólo vale un punto al de 458 pasos. Cabe pensar que los valores más pequeños de gamma incentivan que la nave aterrice en menos tiempo, dado que hacen decrecer a mayor velocidad la recompensa de aterrizar, lo cual en nuestro entorno puede aumentar la puntuación, pero también facilitan que se estrelle, dado que la recompensa negativa de esta acción también decrece con mayor rapidez, pudiendo provocar incluso que no converja, razón por la cual nos quedamos con un valor conservador con el cual se han alcanzado buenos resultados.
6. El *batch size* y el *learning rate* son hiperparámetros con una gran influencia en el entrenamiento de la red neuronal, por lo que siempre se deben probar diferentes combinaciones de los mismos. Para el *batch size* pueden probarse los valores 16, 32 y 64, por ejemplo, al tratarse de datos estructurados, y para el *learning rate* valores en el rango 0,001-0,0001 cuando se utiliza el optimizador Adam como hacemos nosotros.
7. Al tratarse de una DQN con *target network* los hiperparámetros DNN_UPD y DNN_SYNC son de gran importancia y deben estudiarse también, dado que además dependen mucho del entorno. La idea a tener en mente es que la frecuencia de actualización debe ser elevada cuando el entorno da información valiosa (en contraste a repetitiva) también con una frecuencia elevada, y una frecuencia de actualización elevada ayuda a la estabilidad del entrenamiento del agente, pero puede enlentecer el entrenamiento. Como nuestro entorno retorna información valiosa en cada paso probaremos con frecuencias de actualización bajas, 1 o 3, y frecuencias de sincronización bajas pero estables, 1000 y 2000 pasos concretamente.
8. Por último, la arquitectura de la red utilizada también afecta en gran medida al potencial del agente, pudiendo además ser dificil de ajustar debido a la cantidad de posibilidades. Tras realizar algunas pruebas y comprobar que una red *feed-forward* completamente conectada de 2 capas ocultas con 256 neuronas cada una era lo suficientemente compleja para resolver el entorno se ha decidido utilizar siempre esta red.

Los resultados completos de la búsqueda en rejilla, ordenados por el criterio definido y con las columnas que lo reflejan destacadas en negrita, pueden observarse en la tabla 2. El código para realizar la búsqueda en rejilla está contenido en el *script* `grid_search_dqn.py`.

Resultados del entrenamiento

Entrenamos el agente manteniendo un registro de la recompensa obtenida en cada episodio, la media de los 100 últimos episodios y la pérdida media en las actualizaciones en cada episodio. Con esta información y el umbral de recompensa del entorno realizamos la gráfica de recompensas (figura 2) y de la pérdida durante el entrenamiento (figura 3). A la hora de visualizar y evaluar estas gráficas cabe recordar que debido al decrecimiento de epsilon definido (0,985) el valor de epsilon decrece de manera exponencial hasta el episodio 305, tras el cual se alcanza el epsilon mínimo.

Como podemos ver en la figura 2 el entrenamiento no ha sido muy estable: si bien comienza bien, mejorando de manera progresiva hasta alrededor de los 180 puntos en unos 225 episodios, tras mantenerse en esta puntuación durante 75 episodios más (cuando comienza la explotación, sorprendentemente) el agente empeora considerablemente, llegando a obtener recompensas medias casi negativas. Después consigue volver a mejorar, y de manera menos progresiva llega a solucionar el entorno en 667 episodios y 4 **minutos y** 24 **segundos de entrenamiento** en un procesador M1 Pro.

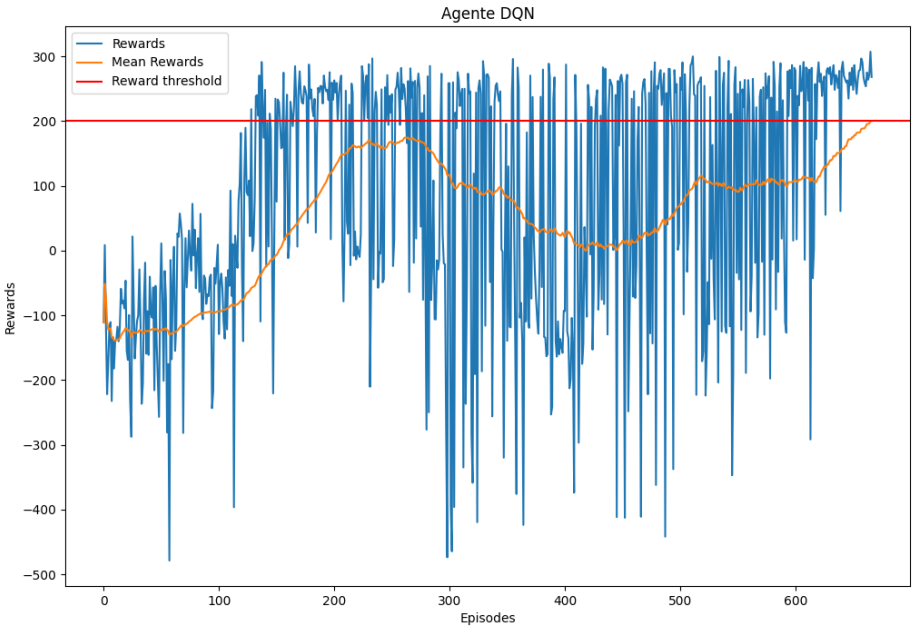

Figura 2: Recompensas obtenidas durante el entrenamiento

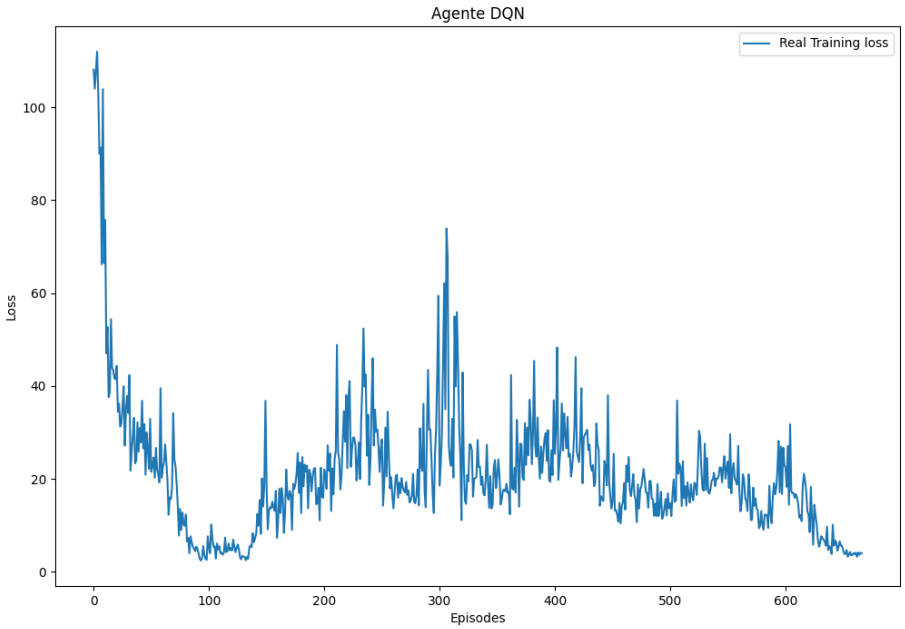

Figura 3: Pérdida media en cada episodio del entrenamiento

Como podemos ver en la figura 3 las pérdidas durante el entrenamiento tampoco son muy estables, con pérdidas por encima de los 15 puntos durante gran parte del entrenamiento y una tendencia similar a la vista en la figura 2, como podemos por ejemplo observar en el pico del error alrededor del episodio 300. En contraste, cabe destacar que el entrenamiento concluye en un momento en el que el error es mínimo y estable, posible indicador de que el agente ha alcanzado un punto de cierta estabilidad.

2.3. Prueba del agente entrenado

Evaluamos el agente obtenido en el anterior apartado jugando 100 partidas con epsilon igual a cero, es decir de manera 100 % *greedy*[1]. En la figura 4 podemos ver la recompensa total obtenida en cada episodio, la media, la mediana y el valor del umbral de recompensa.

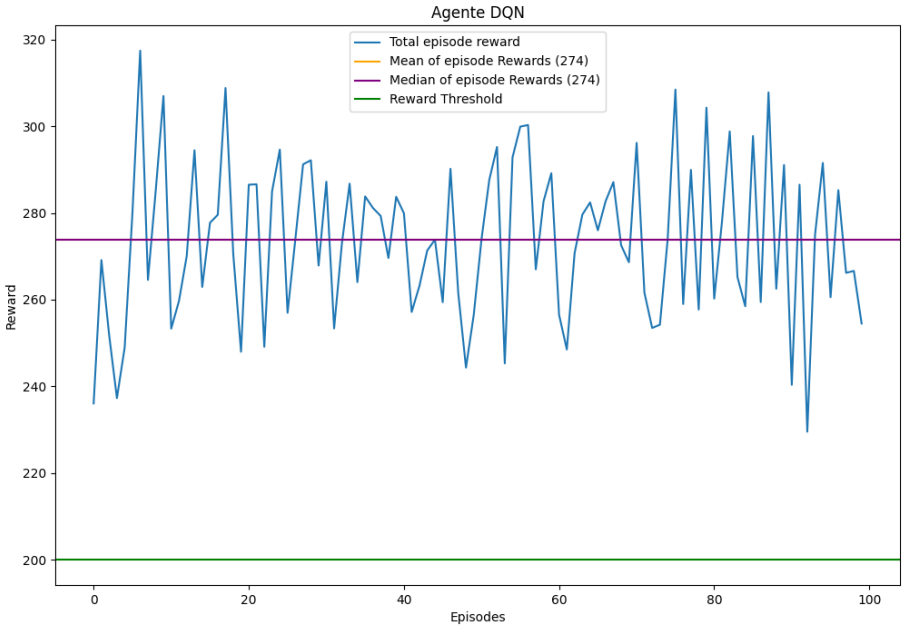

Figura 4: Recompensas en episodios de evaluación

Como podemos ver el agente supera con creces el umbral en todas las partidas, obteniendo por lo tanto una media, mediana e incluso recompensa mínima claramente superiores al umbral. Esto significa que **el agente ha sido capaz de aterrizar mediante una trayectoria eficiente en todas las partidas, sin estrellarse nunca**. Se trata por lo tanto de un agente casi óptimo según los criterios de evaluación que hemos definido, siendo el tiempo de entrenamiento el único criterio con margen de mejora. Como nota final, este agente entraría entre los 500 mejores agentes del ranking [7].

[1]Podemos probar el agente con epsilon cero porque el entorno tiene un componente aleatorio en la inicialización y no modifica su comportamiento en función del comportamiento del agente o el tiempo.

2.4. Resultados completos búsqueda en rejilla

Solved	Training episodes	**Training time**	Mean evaluation rewards	Median evaluation rewards	**Well landed evaluation episodes**	Landed evaluation episodes	**Crashed evaluation episodes**	LEARNING RATE	DNN UPD. FREQ.	DNN SYNC. FREQ.	BATCH SIZE	MAX. EPISODES
True	667	4,4	274,21	273,9	100	0	0	0,001	1	1000	32	1000
True	556	4,61	270,95	276,87	96	4	0	0,001	1	1000	16	1000
True	398	5,71	254,54	259,52	94	6	0	0,0001	1	2000	16	1000
True	327	4,23	239,96	249,7	87	13	0	0,0001	1	2000	32	1000
True	231	2,9	252,59	260,6	94	5	1	0,0001	1	1000	32	1000
True	342	2,27	243,14	252,93	90	8	2	0,0005	3	1000	64	1000
True	458	4,65	230,38	243,89	89	9	2	0,001	1	2000	16	1000
True	332	5,62	225,82	239,68	82	16	2	0,0001	1	2000	64	1000
True	350	2,74	224,68	242,87	74	24	2	0,001	3	2000	64	1000
True	481	3,37	242,29	258,53	91	6	3	0,0001	3	1000	64	1000
True	713	4,9	234,7	253,09	87	9	4	0,001	3	2000	16	1000
True	417	4,48	233,87	250,04	86	10	4	0,0005	1	2000	16	1000
True	307	3,85	228,11	250,42	81	15	4	0,0005	1	1000	16	1000
True	268	4,48	220,86	248,53	73	23	4	0,0001	1	1000	64	1000
True	444	3,27	224,09	241,1	87	8	5	0,0005	3	2000	32	1000
True	574	3,44	229,0	255,47	83	12	5	0,001	3	2000	32	1000
True	902	15,65	230,51	243,13	89	4	7	0,0001	3	1000	32	1000
True	729	11,23	227,13	251,16	81	12	7	0,0001	3	2000	32	1000
True	433	5,62	225,6	249,24	80	13	7	0,0001	3	2000	64	1000
True	639	4,97	204,84	235,7	78	15	7	0,0005	3	2000	64	1000
True	320	2,24	241,79	264,31	91	1	8	0,0005	3	1000	16	1000
True	364	3,08	239,81	257,92	89	3	8	0,001	3	1000	32	1000
True	314	4,7	223,6	242,81	80	11	9	0,0001	1	1000	16	1000
True	736	4,72	193,07	197,34	48	42	10	0,0005	3	2000	16	1000
True	491	4,0	238,2	263,39	86	3	11	0,0005	1	1000	32	1000
True	885	8,93	205,23	259,07	77	12	11	0,001	1	2000	64	1000
True	325	2,67	209,3	248,41	76	13	11	0,001	3	1000	16	1000
True	358	4,11	186,76	223,42	66	21	13	0,0005	1	2000	32	1000
False	1000	10,43	185,32	220,72	58	27	15	0,0001	3	1000	16	1000
True	840	9,64	216,18	263,25	82	2	16	0,001	1	1000	64	1000
False	1000	15,45	194,44	234,31	74	7	19	0,0001	3	2000	16	1000
True	411	3,64	184,54	242,96	72	9	19	0,001	1	2000	32	1000
True	882	7,68	166,92	264,55	74	6	20	0,0005	1	2000	64	1000
True	261	2,33	176,97	229,0	67	10	23	0,0005	3	1000	32	1000
True	370	2,66	157,88	215,78	57	18	25	0,001	3	1000	64	1000
True	391	4,6	180,95	241,99	64	6	30	0,0005	1	1000	64	1000

Tabla 2: Resultado búsqueda en rejilla DQN

3. Propuesta de mejora

En este apartado implementamos otro agente, entre aquellos que hemos visto a lo largo de la asignatura, que pueda solucionar el problema de robótica espacial de forma más eficiente con respecto al agente DQN, es decir, mejor según el criterio que hemos establecido en el apartado 1.3.

Para ello analizamos las posibles mejoras del agente DQN utilizado [8] y la posibilidad de utilizar otros algoritmos estudiados [9, 10]. Como hemos visto en el anterior apartado un agente DQN es capaz de resolver con unos resultados magníficos en este entorno, siendo el tiempo de entrenamiento el criterio único con margen de mejora, que es por lo tanto el apartado que debemos tratar de mejorar. Las opciones consideradas son las siguientes:

1. El aprendizaje multipaso, que se basa en la actualización de los parámetros de la DQN mediante transiciones más largas, con las cuales pretende mejorar y reducir el tiempo de aprendizaje, siendo su principal desventaja la peor convergencia según crece el número de pasos. Podría ser una opción interesante en nuestra situación.
2. Las *Double Q-network*s, que nos permiten lidiar con el sesgo del valor de Q debido a la correlación entre estados, evitando el posible problema de convergencia a políticas subóptimas de las DQN. Son útiles en muchos escenarios, entre los cuales probablemente se encuentre nuestro entorno.
3. El *Prioritized experience replay*, que solventa el problema de no dar más importancia a las experiencias relevantes. Como en nuestro caso todas las experiencias son relevantes, y las más relevantes, aterrizar o estrellarse, ocurren cada episodio, no parece que vaya a ser de gran utilidad, como comprueban en [11, 12]. Además requiere ajustar más hiperparámetros.
4. Las *Dueling Q-network*, que ayudan a mejorar la eficiencia calculando los valores de Q únicamente en aquellos estados en los que una acción pueda dar cierta ventaja. Como en este entorno no hay apenas estados donde la acción a tomar sea de poca importancia no parece que vayan a ser de especial ayuda. Se han realizado algunas pruebas mediante el *script* `dueling_dqn.py` y comprobado que efectivamente no hay una mejora sustancial en comparación a sólo utilizar el agente *Double DQN*.
5. Las *DQN* categóricas, que obtienen una distribución de las posibles acciones a tomar. Como nuestro entorno es sencillo y no contiene ningún componente estocástico más allá del inicio de cada partida no parece una mejora especialmente útil.
6. Las *NoisyNet* (redes con ruido), que aportan una solución más eficiente al problema de la exploración. Dado que en nuestro entorno la falta de exploración no es un problema especialmente grave, considerando que la política óptima pasa por realizar un desplazamiento eficiente, no parece que sean de especial interés.
7. La *Rainbow DQN*, que combina todas las anteriores mejoras, no parece interesante en este entorno, dado que la mayoría de sus componentes ya hemos considerado que no lo son.
8. Los gradientes de política, que no parecen una buena opción en este entorno, dado que al tener que esperar al final de cada episodio para aprender la convergencia será mucho más lenta, ya que le costará descubrir buenas políticas, como puede verse por ejemplo en [13] y comprobarse mediante el *script* `reinforce.py`.
9. Los métodos actor crítico, que suelen proporcionar una mayor estabilidad y velocidad de convergencia, son también una buena opción, siendo su principal desventaja la necesidad de entrenar dos redes en vez de una, con su consecuente impacto en el tiempo de entrenamiento. Este motivo las hace poco adecuadas como mejora del agente DQN obtenido.

Se opta por implementar una ***Double Q-network*** como posible mejora, dado que es probable que mejore la convergencia del agente a la política óptima, manteniendo así un nivel similar de resultados y reduciendo el tiempo de entrenamiento.

3.1. Implementación

El código de este apartado está contenido en el *script* `double_dqn.py`, en el que:

1. Importamos la clase `DQN` del módulo `dqn.py`.
2. Definimos la clase `DoubleDQNAgent` del agente *Double DQN*.
3. Inicializamos el entorno.
4. Entrenamos el agente con unos hiperparámetros definidos.
5. Imprimimos las gráficas de recompensas y pérdidas del entrenamiento del agente.
6. Guardamos los pesos de la red.
7. Imprimimos la gráfica de evaluación del agente.
8. Imprimimos el número de episodios que ha resuelto, en los que ha aterrizado y en los que se ha estrellado.
9. Analizamos visualmente algunos episodios del agente.
10. Y guardamos un gif de su comportamiento.

La clase `DoubleDQNAgent` contiene la implementación del agente *Double DQN*, muy similar a la clase `DQNAgent`, siendo la manera de calcular el error (dentro del método `calculate_loss()`) la única diferencia.

La arquitectura de la red DQN implementada se mantiene, es decir, sigue siendo la descrita en el apartado 2.1. También seguimos utilizando el optimizador Adam para entrenar la red.

El *replay-buffer* sigue siendo también el mismo, la clase `ExperienceReplayBuffer` presente en el ejemplo del capítulo 9 en el repositorio de la asignatura [3, Cap. 9].

3.2. Entrenamiento

Los hiperparámetros que impactan el entrenamiento del agente junto con el valor utilizado se muestran en la tabla 3. Al igual que en el entrenamiento del agente DQN estos hiperparámetros se han obtenido mediante una búsqueda en rejilla. Las combinaciones estudiadas en esta búsqueda en rejilla son las mismas que en el caso de la DQN, y los resultados mostrados son los del caso en el que el decaimiento de epsilon (`EPSILON_DECAY`) es igual a 0,98, caso en el que se ha obtenido el mejor resultado. Los resultados completos de esta búsqueda en rejilla, ordenados por el criterio definido y con las columnas que lo reflejan destacadas en negrita, pueden observarse en la tabla 4. El código para realizar la búsqueda en rejilla está contenido en el *script* `grid_search_ddqn.py`.

Hiperparámetro	**Variable**	**Valor**
Tamaño del *buffer*	MEMORY_SIZE	10000
Burn in	BURN_IN	1000
Epsilon inicial	INIT_EPSILON	1
Decrecimiento de epsilon	EPSILON_DECAY	0,98
Epsilon mínimo	MIN_EPSILON	0,01
Máximo de episodios	MAX_EPISODES	1000
Gamma	GAMMA	0,99
Batch size	BATCH_SIZE	32
Learning rate	LR	0.001
Frecuencia de actualización	DNN_UPD	3
Frecuencia de sincronización	DNN_SYNC	1000

Tabla 3: Hiperparámetros del aprendizaje del agente DDQN

Resultados del entrenamiento

Entrenamos el agente manteniendo un registro de la recompensa obtenida en cada episodio, la media de los 100 últimos episodios y la pérdida media en las actualizaciones en cada episodio. Con esta información y el umbral de recompensa del entorno realizamos la gráfica de recompensas (figura 2) y de la pérdida durante el entrenamiento (figura 3). A la hora de visualizar y evaluar estas gráficas cabe recordar que debido al decrecimiento de epsilon definido (0,98) el valor de epsilon decrece de manera exponencial hasta el episodio 228, cuando se alcanza el epsilon mínimo.

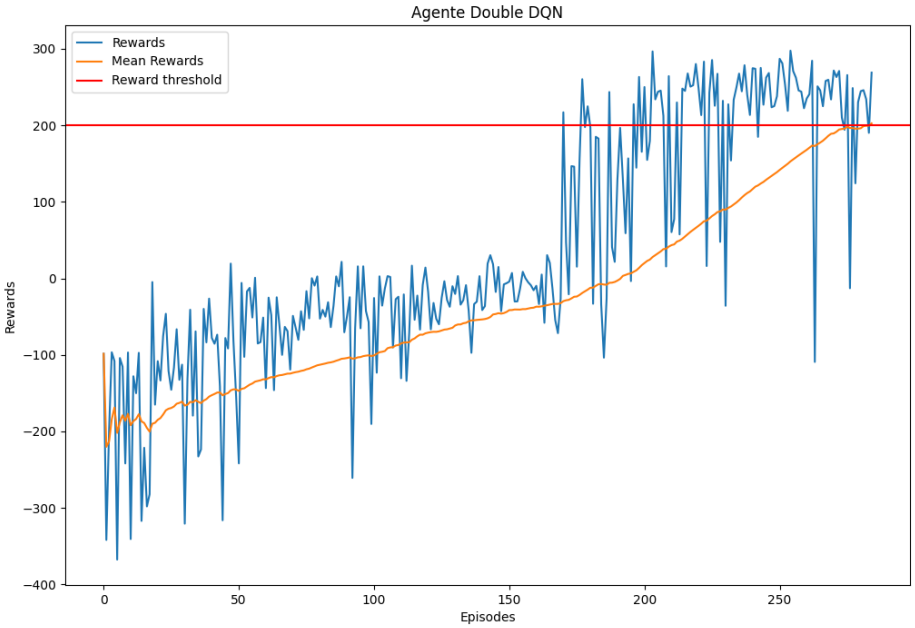

Figura 5: Recompensas obtenidas durante el entrenamiento

Como podemos ver en la figura 5 el entrenamiento ha sido mucho más estable, como puede observarse en el crecimiento casi lineal de la media de recompensas. Además se observa una mejora clara de la convergencia del episodio 170 en adelante, tras la cual el agente converge aún más rápidamente, llegando a resolver el entorno en apenas 285 episodios y 3 **minutos y** 6 **segundos de entrenamiento** en un procesador M1 Pro, un 30 % más rápido que el agente DQN.

Como podemos ver en la figura 3 las pérdidas durante el entrenamiento no son tan estables, aunque siguen siendo más estables que en el agente DQN. Durante los primeros 70 episodios las pérdidas son grandes debido a que el agente está aprendiendo y parte de cero, después hay un periodo de pérdidas muy pequeñas que dura hasta el episodio 170, que se corresponde son un periodo de estabilidad del agente observable en la figura 5, y estas vuelven a incrementarse cuando de nuevo el agente aprende a gran velocidad. En contraste al agente DQN, el entrenamiento de este agente concluye en un momento donde los errores obtenidos siguen siendo de un tamaño considerable, y por lo tanto el agente no ha acabado de estabilizarse.

3.3. Prueba del agente entrenado

Evaluamos el agente obtenido en el anterior apartado jugando 100 partidas con el epsilon mínimo. En la figura 7 podemos ver la recompensa total obtenida en cada episodio, la media, la mediana y el valor del umbral de recompensa.

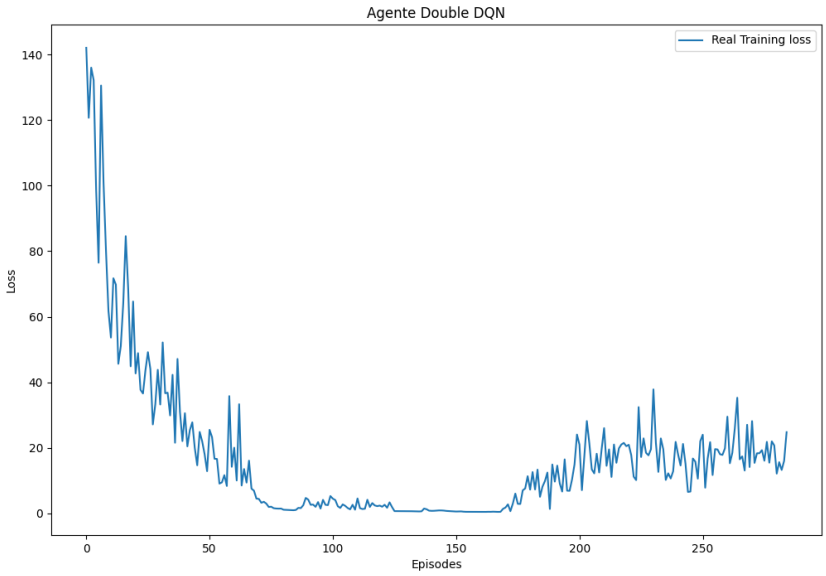

Figura 6: Pérdida media en cada episodio del entrenamiento

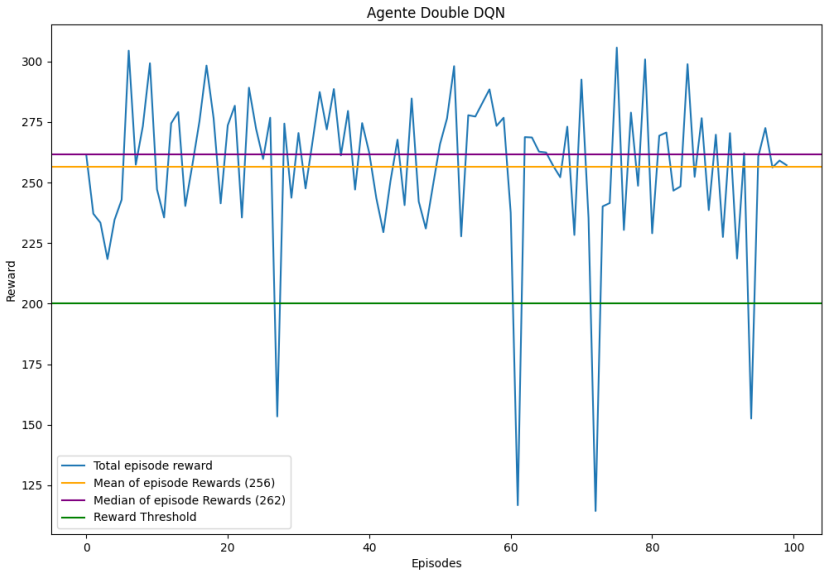

Figura 7: Recompensas en episodios de evaluación

Como podemos ver el agente supera con claridad el umbral en la mayoría de partidas, obteniendo además una media y mediana claramente superiores al umbral, si bien ligeramente inferiores al agente DQN. Concretamente, **el agente nunca se estrella**, y además **es capaz de aterrizar de manera eficiente en 96 partidas**, siendo el aterrizaje no eficiente en los 4 episodios restantes. Reproduciendo estos 4 episodios se puede observar que en todos los casos hay una pendiente descendiente a la derecha de la pista de aterrizaje, de la cual el agente no es consciente (como comentamos en el apartado 1.2), y sobre la cual aterriza parcialmente, quedándose atascado hasta agotar el tiempo tratando de ascenderla. En comparación el agente DQN aterriza más centrado en estos episodios y no se ve afectado por la pendiente.

En conclusión, según el criterio que hemos establecido en el apartado 1.3 **el agente DQN obtenido es mejor que el agente DDQN**, aunque no por mucho, dado que ambos consiguen no estrellarse en ninguna partida de evaluación, pero el agente DQN consigue además aterrizar de manera eficiente en 4 episodios más. El agente DDQN es mejor sólo en el tercer criterio de evaluación, el tiempo de entrenamiento, considerablemente inferior al del agente DQN.

3.4. Resultados completos búsqueda en rejilla

Solved	Training episodes	**Training time**	Mean evaluation rewards	Median evaluation rewards	**Well landed evaluation episodes**	Landed evaluation episodes	**Crashed evaluation episodes**	LEARNING RATE	DNN UPD. FREQ.	DNN SYNC. FREQ.	BATCH SIZE	MAX. EPISODES
True	285	3,1	256,24	261,67	96	4	0	0,001	3	1000	32	1000
True	447	12,5	262,08	266,93	95	5	0	0,0001	1	2000	32	1000
True	432	6,59	237,2	245,79	90	8	2	0,0005	3	2000	16	1000
True	433	96,42	251,25	262,63	90	8	2	0,0001	3	2000	64	1000
True	304	6,99	230,93	242,11	83	15	2	0,0001	1	2000	64	1000
True	399	4,07	234,67	248,99	91	6	3	0,0005	3	1000	32	1000
True	391	5,72	249,48	259,07	90	7	3	0,001	1	2000	32	1000
True	406	6,69	237,55	249,58	91	4	5	0,0001	1	1000	64	1000
True	507	10,75	221,67	255,22	85	10	5	0,0005	1	2000	32	1000
True	659	11,62	223,19	243,86	81	14	5	0,0001	3	1000	64	1000
True	422	7,79	223,64	247,17	81	13	6	0,001	1	2000	16	1000
True	271	2,55	220,29	240,11	80	14	6	0,0005	3	1000	64	1000
True	490	5,59	222,12	246,69	79	14	7	0,001	3	2000	16	1000
True	569	8,1	222,63	244,96	78	15	7	0,001	3	2000	64	1000
True	271	5,53	226,4	251,28	77	16	7	0,0001	1	1000	32	1000
True	339	2,97	212,25	245,27	78	13	9	0,001	3	1000	64	1000
True	366	7,76	211,92	241,05	77	13	10	0,0001	1	2000	16	1000
True	662	6,62	228,53	252,77	83	6	11	0,001	3	2000	32	1000
True	305	4,86	210,01	234,72	77	12	11	0,001	1	1000	64	1000
True	551	6,82	196,97	240,13	67	21	12	0,0005	1	1000	32	1000
True	390	6,47	198,16	242,79	68	19	13	0,0005	1	2000	16	1000
True	863	11,99	193,71	205,7	53	34	13	0,0001	3	1000	32	1000
True	388	4,52	197,52	235,24	66	20	14	0,0005	3	2000	64	1000
True	673	11,43	201,73	242,57	66	20	14	0,0001	3	1000	16	1000
True	357	6,49	167,36	188,87	37	48	15	0,0001	1	1000	16	1000
True	376	5,33	190,3	231,38	74	10	16	0,0005	1	1000	64	1000
True	783	13,42	190,19	249,37	73	11	16	0,001	1	2000	64	1000
True	617	7,48	184,54	223,66	62	19	19	0,0005	1	1000	16	1000
True	340	6,18	193,5	248,05	65	14	21	0,0005	1	2000	64	1000
True	355	4,34	165,42	257,85	74	1	25	0,001	3	1000	16	1000
False	1000	18,45	123,89	166,86	19	50	31	0,0001	3	2000	32	1000
True	344	4,38	163,39	244,9	64	2	34	0,0005	3	2000	32	1000
True	397	4,61	120,29	204,85	51	12	37	0,001	1	1000	16	1000
False	1000	18,76	137,56	193,99	49	14	37	0,0001	3	2000	16	1000
True	356	3,3	143,28	216,15	52	9	39	0,0005	3	1000	16	1000
True	287	3,78	107,04	82,07	44	6	50	0,001	1	1000	32	1000

Tabla 4: Resultado búsqueda en rejilla DDQN

Referencias

[1] Gym. Lunar Lander, . URL https://www.gymlibrary.dev/environments/box2d/lunar_lander/.

[2] Gym. Lunar Lander source code, . URL https://github.com/openai/gym/blob/0.26.2/gym/envs/box2d/lunar_lander.py.

[3] Jordi Casas. Aprendizaje por refuerzo. URL https://github.com/jcasasr/Aprendizaje-por-refuerzo.

[4] Gymnasium Documentation. v21 to v26 Migration Guide. URL https://gymnasium.farama.org/content/migration-guide/.

[5] Gym. Gym Core Documentation, . URL https://www.gymlibrary.dev/api/core/.

[6] Igor Aherne. How large should the replay buffer be? URL https://ai.stackexchange.com/a/13296.

[7] Thomas Simonini. LunarLander-v2 leaderboard. URL https://huggingface.co/spaces/ThomasSimonini/Deep-Reinforcement-Learning-Leaderboard.

[8] Laura Ruiz Dern. *Deep Q-networks*. UOC, 2021.

[9] Laura Ruiz Dern. *Gradientes de política*. UOC, 2021.

[10] Laura Ruiz Dern. *El método actor-crítico*. UOC, 2021.

[11] Guillaume Crabé. How to implement Prioritized Experience Replay for a Deep Q-Network. URL https://huggingface.co/spaces/ThomasSimonini/Deep-Reinforcement-Learning-Leaderboard.

[12] Sophia Xu. Lunar Lander - Deep Reinforcement Learning, Noise Robustness, and Quantization. URL https://xusophia.github.io/DataSciFinalProj/.

[13] Pau Labarta Bajo. How Policy Gradients can get you to the Moon. URL https://datamachines.xyz/2022/05/06/policy-gradients-in-reinforcement-learning-to-land-on-the-moon-hands-on-course/.