

UOC
Aprendizaje por refuerzo
PRA: Implementación de un agente para robótica espacial

Gerson Villalba Arana

Enero 2023

Índice general

1. Entorno	2
1.1. Exploración del entorno	2
1.2. Ejecución del entorno	3
2. Agente de referencia	4
2.1. Descripción del agente	4
2.2. Entrenamiento y selección de hiperparámetros	4
2.3. Prueba del agente	5
3. Propuesta de mejora	8
3.1. Selección e implementación del agente alternativo	8
3.1.1. Double DQN	8
3.1.2. Dueling DQN	9
3.1.3. NoisyNet DQN	10
3.2. Entrenamiento y selección de hiperparámetros	10
3.3. Prueba del agente y comparativa	12
4. Implementación	15

Entorno

1.1. Exploración del 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 (espacio de acciones) son discretas.

Cargando el entorno "LunarLander-2" de Gym podemos realizar una exploración inicial del entorno y observamos que el retorno fijado para resolver el entorno es de 200 puntos, además de que el rango de retorno no está limitado (-Inf, Inf). Además, se fija un número máximo de pasos por episodio de 1000, a partir del cual el episodio se finalizará automáticamente.

La dimensión del vector de observaciones es de 8, correspondiente a los siguientes valores según la documentación de OpenAI [1].

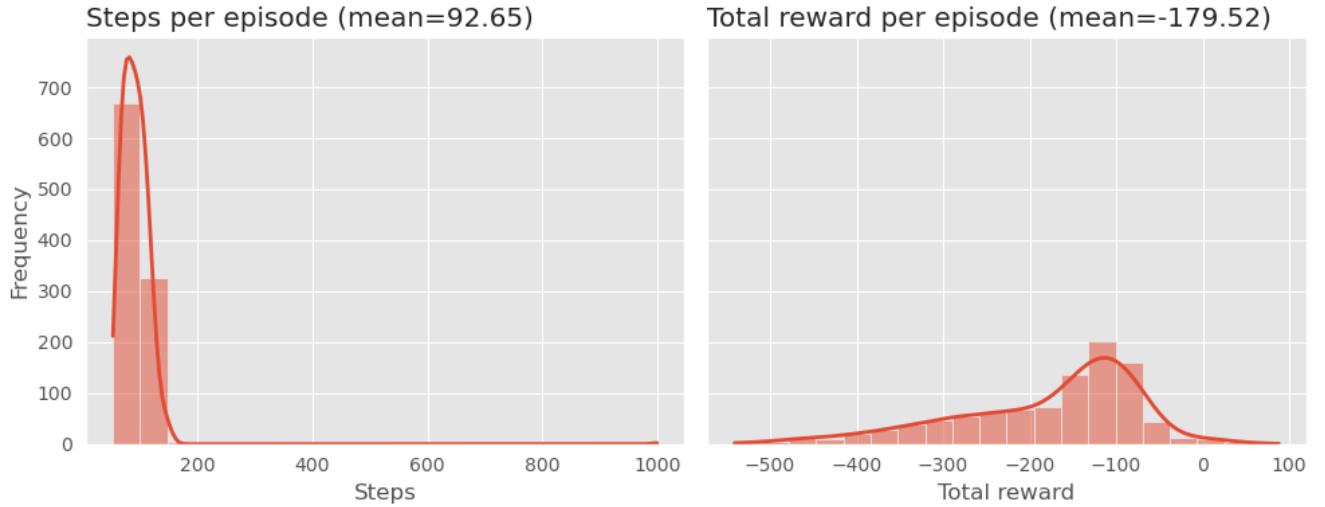
1. Posición en x
2. Posición en y
3. Velocidad en x
4. Velocidad en y
5. Ángulo de rotación del módulo
6. Velocidad angular
7. Booleano que indica si la pata izquierda está en contacto con el suelo
8. Booleano que indica si la pata derecha está en contacto con el suelo

Teniendo en cuenta la observación que tenemos como representación del estado del entorno, estamos ante un modelo de decisión de Markov, ya que la observación determina completamente el estado en el que se encuentra el entorno. Esto quiere decir que el comportamiento del entorno no va a depender de estados anteriores, sino que una observación del actual nos da información completa sobre su comportamiento futuro. Esto es así porque dentro de la observación del estado no sólo tenemos la posición y ángulo del módulo lunar, sino también su velocidad en ambas coordenadas (dirección e intensidad de movimiento) y su velocidad angular (dirección e intensidad de rotación).

Por otro lado, el tamaño del vector de acciones es de 4, correspondiente a las siguientes acciones:

1. No hacer nada
2. Encender el motor izquierdo

Figura 1.1: Ejecución aleatoria del entorno



3. Encender el motor principal (bajo la nave)
4. Encender el motor derecho

1.2. Ejecución del entorno

Para evaluar el entorno, así como las posibles recompensas, ejecutamos 1000 episodios completos con un agente que seleccione siempre una acción aleatoria de entre las cuatro posibles. El resultado de la ejecución de estos episodios se puede ver en la Figura 1.1. Como puede comprobarse, no sólo la recompensa media obtenida es muy baja y está muy por debajo de la requerida para solucionar el entorno (200), sino que en ninguna de las 1000 ejecuciones se ha llegado a dicha recompensa. Representamos en la misma figura también el histograma del número de pasos realizados por episodio, comprobando que en alguno de ellos llegamos al máximo de 1000 pasos por episodio definido por el entorno.

Agente de referencia

2.1. Descripción del agente

El agente que implementaremos inicialmente es un DQN con replay buffer y target network, que utilizaremos como agente de referencia para mejorarlo en el siguiente apartado. Como en este entorno disponemos de un estado representado por un vector de 8 variables numéricas, utilizaremos una red neuronal completamente conectada con una primera capa de entrada de 8 neuronas, dos capas ocultas con un número de neuronas que tomaremos como hiperparámetro y una capa de salida con 4 neuronas, correspondientes a las 4 acciones que tenemos disponibles en el entorno.

Por otro lado, decidimos utilizar un optimizador Adam para el entrenamiento de la red neuronal. Esta es una decisión que se toma porque este optimizador es más rápido de converger en la mayoría de los casos que Stochastic Gradient Descent y funciona bien para la mayoría de problemas.

2.2. Entrenamiento y selección de hiperparámetros

A continuación se listan los siguientes hiperparámetros que se van a modificar, así como su descripción y el valor de referencia del que partimos de cada uno de ellos.

- **LR.** Learning rate del optimizador Adam. Partimos de un valor de 0.001.
- **MEMORY_SIZE.** Tamaño del buffer de experiencias donde almacenaremos estas experiencias para tomar de él un conjunto para entrenamiento de la red. Seleccionamos este valor inicialmente a 8000.
- **DNN_UPD.** Frecuencia de actualización de la red principal. Seleccionamos este valor inicialmente a 4.
- **BATCH_SIZE.** Tamaño de muestra de experiencias que tomamos del buffer cada vez que vamos a entrenar la red (con frecuencia DNN_UPD). Seleccionamos este valor inicialmente a 32 muestras.
- **N_NEURONS.** Número de neuronas de las capas ocultas de la red neuronal. A mayor número, más capacidad tendrá la red de aprender patrones complejos de los datos. Seleccionamos este valor inicialmente a 64.
- **GAMMA.** Gamma de la ecuación de Bellman para estimar los valores Q. Seleccionamos este valor inicialmente a 0.99.

Para seleccionar la mejor combinación de hiperparámetros, modificamos uno a uno entre un rango razonable de valores para cada uno. No realizamos todas las combinaciones de ellos porque su número es muy alto y el tiempo total de entrenamiento de todas las opciones sería demasiado alto. Hay que decir también que se podrían haber incluido más hiperparámetros, como la frecuencia de sincronización de la

red, el número de capas ocultas de la red o la función de activación utilizada en la red, pero lo acotamos a estas.

Como en el entrenamiento del agente hay una fuerte componente aleatoria (el propio entorno comienza en un estado aleatorio, la selección del batch del buffer de experiencias es aleatorio, la inicialización de la red neuronal también lo es..), realizamos el entrenamiento con cada valor de hiperparámetro 5 veces, para tener una estimación más fiable de su comportamiento y poder descartar casos extremos.

En la Figura 2.1 se muestran los resultados de entrenamiento del agente DQN con distintas variaciones de estos hiperparámetros. Las gráficas muestran la recompensa media obtenida en los cien últimos episodios para cada uno de los casos estudiados. En la parte superior de ellas se muestra la distribución del número de episodios necesarios para solucionar el entorno para cada uno de los valores probados, que nos da una información muy útil. En todos los casos se ha establecido un número de episodios de entrenamiento máximo de 1000. Considerando que muchos de los casos ya se ha conseguido la resolución del entorno (200 puntos de recompensa) en este número de episodios, los casos en los que no haya sido así, se descartarán por considerarse demasiado lentos en el entrenamiento, aun en el supuesto de que acabasen resolviendo el entorno con un mayor número de episodios.

En base a lo visto en las gráficas de entrenamiento, se seleccionan como óptimos los siguientes hiperparámetros:

- **LR = 0.001.** Se puede observar un comportamiento muy errático en el entrenamiento cuando la tasa de aprendizaje es muy alta (0.01) o baja (0.0001). Nos quedamos con 0.001 porque es el que consigue solucionar el entorno en menos tiempo en media.
- **BATCH_SIZE = 64.** No parece que haya una gran ventaja a favor de ninguno de los valores que hemos probado. Seleccionamos 64 porque es el que tiene una media menor, pero es marginal.
- **N_NEURONS = 128.** Seleccionamos 128 porque tenemos claramente un entrenamiento más rápido y estable. Parece claro que con un modelo demasiado sencillo (32 neuronas en cada una de las dos capas ocultas), en muchos casos no conseguimos solucionar el entorno en 1000 episodios.
- **DNN_UPD = 2.** Parece claro que con este valor se ha conseguido una mayor velocidad de aprendizaje. No solamente ha tenido el menor número de episodios para resolverse sino que además la desviación típica es la más baja
- **MEMORY_SIZE = 8000.** Se nota un comportamiento errático en este parámetro. Los valores 8000 y 32000 son los que nos proporcionan un mejor comportamiento, pero sin embargo 16000 es considerablemente peor que estos, por lo que no sigue un patrón lógico. Seleccionamos 8000 porque nos proporciona una mayor estabilidad en el aprendizaje.
- **GAMMA = 0.99.** El entrenamiento parece muy sensible a este parámetro. Valores demasiado cercanos a 1 parecen tener un efecto errático en el entrenamiento, y valores por debajo de 0.99 limitan la capacidad de aprendizaje del agente, no llegando a resolver el entorno. Seleccionamos 0.99, que claramente es el valor que mejor resultado nos proporciona entre los probados.

2.3. Prueba del agente

Con el agente DQN entrenado del apartado anterior, procedemos a probar el comportamiento de éste, haciendo uso del modo explotación del agente. Esto quiere decir que en lugar de utilizar una política ϵ -greedy como en entrenamiento, donde elegimos con una probabilidad ϵ una acción aleatoria entre las disponibles, utilizaremos una política greedy, donde siempre elegiremos la acción que nos proporcione un retorno esperado mayor, de acuerdo a su valor en Q.

Figura 2.1: Selección de hiperparámetros para DQN

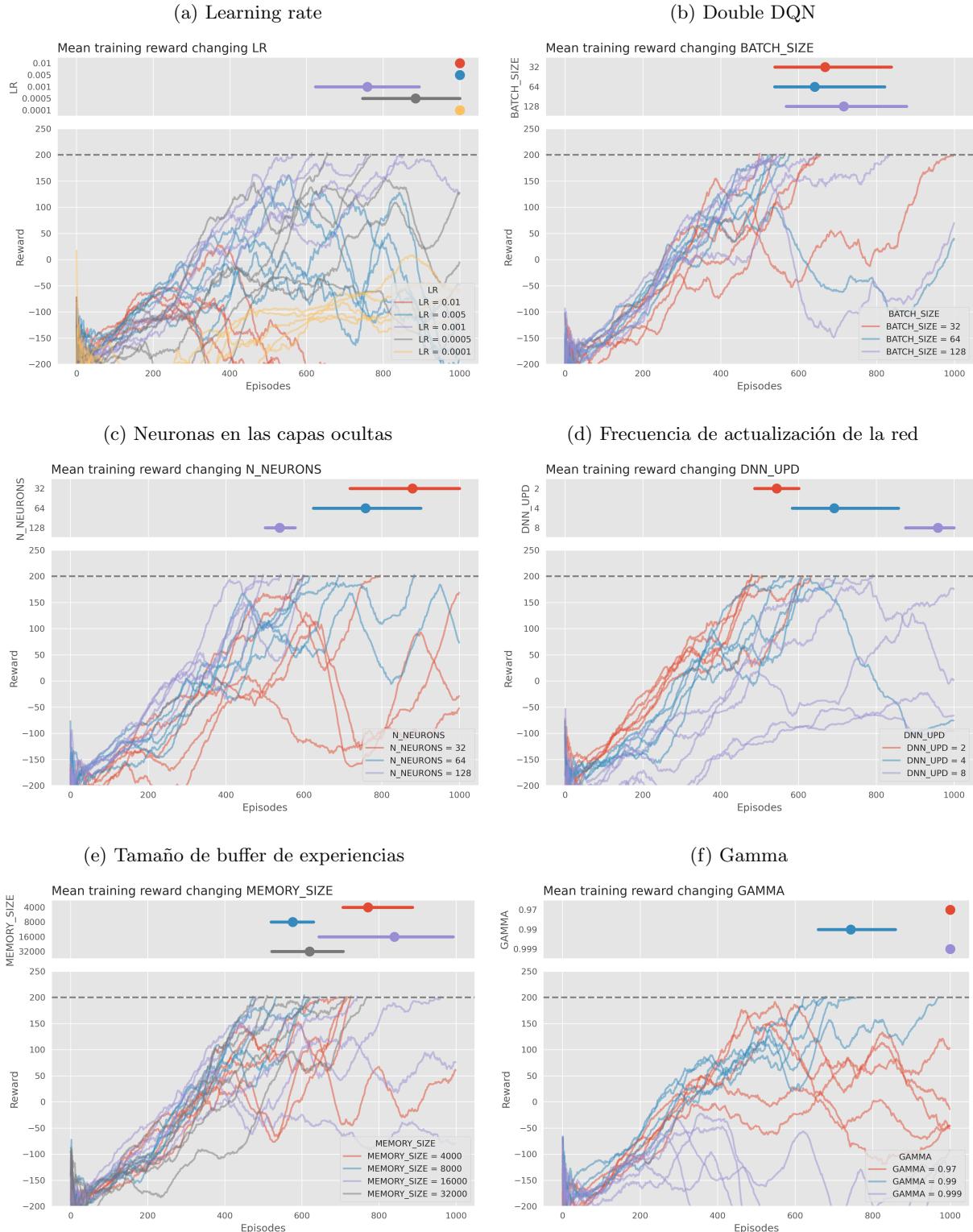
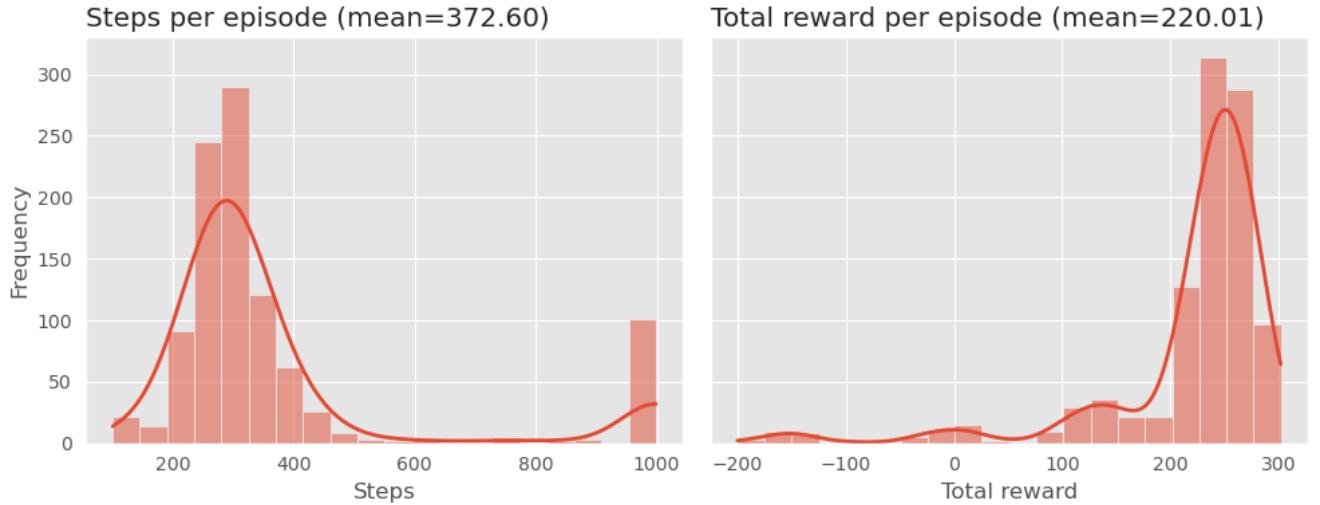


Figura 2.2: Prueba del agente entrenado



Hay que tener en cuenta que el entorno con el que estamos trabajando tiene un inicio aleatorio cada vez que lo reiniciamos, por lo que cada partida será distinta al comenzar en un estado diferente.

En la Figura 2.2 podemos ver el histograma con las recompensas obtenidas tras la ejecución de 1000 episodios distintos. Como se puede observar, la recompensa media obtenida es de 220 puntos, bastante alta si tenemos en cuenta que a partir de 200 se considera solucionado el entorno. Además, la gran mayoría de ejecuciones consiguen llegar a este umbral de solución del entorno. Tenemos que recordar que en la ejecución del entorno con un agente aleatorio habíamos conseguido un retorno medio de -180 puntos, por lo que realmente el agente entrenado se comporta de una forma muy satisfactoria.

Propuesta de mejora

3.1. Selección e implementación del agente alternativo

En primer lugar hay que indicar que con el agente DQN y haciendo una selección cuidadosa de los hiperparámetros a utilizar hemos obtenido unos resultados muy adecuados para el problema. Hemos llegado a resolver el entorno y en un plazo de tiempo (episodios jugados) relativamente corto. Además, una vez entrenado el agente, hemos obtenido una recompensa acumulada muy alta en la ejecución de partidas en modo explotación. El objetivo que se persigue conseguir ahora no es por lo tanto una mejora de la recompensa obtenida, que ya la tenemos muy buena, sino en poder tener un entrenamiento del agente más rápido y/o más estable.

En primer lugar, hay que descartar el uso de métodos tabulares para la solución de este entorno. El motivo es que las observaciones que tenemos son valores continuos (a excepción de las dos variables que nos indican si el robot toca con alguna de sus patas tierra), y por lo tanto no tenemos un conjunto discreto finito de estados. Una posible solución sería la discretización de estos valores, pero en ese caso ya estaríamos haciendo aproximaciones y a pesar de todo acabaríamos con un espacio de estados demasiado grande como para poder solucionarlo con métodos tabulares en un tiempo razonable.

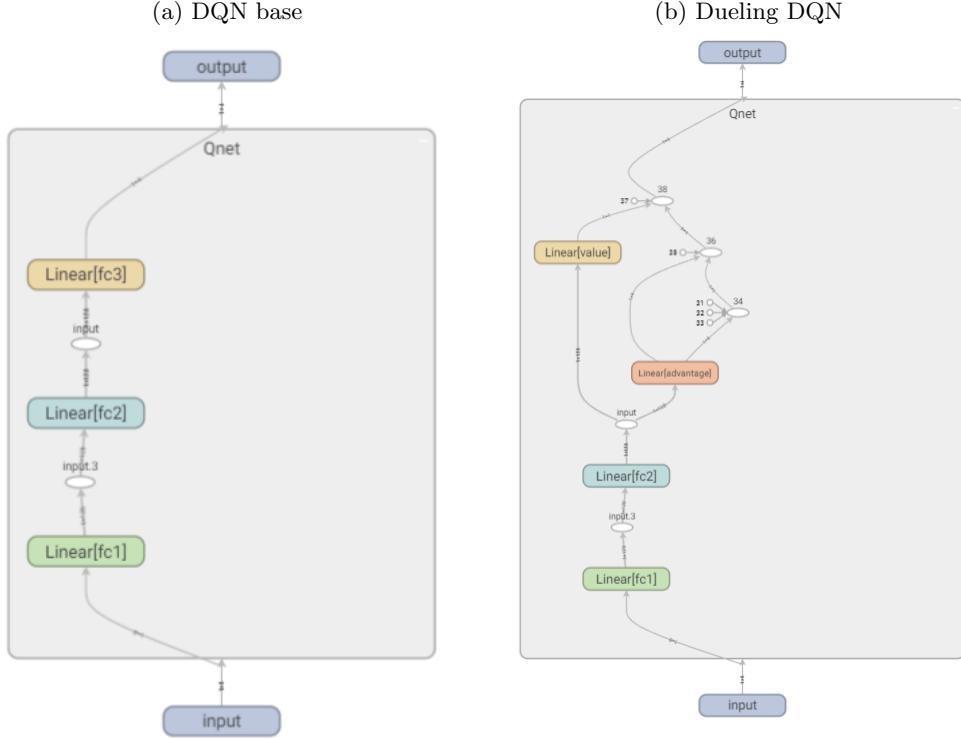
Por otro lado, nos planteamos el uso de métodos aproximados que aproximen directamente la política (policy gradients) o métodos que aproximen ambos el Q y la política. Estos métodos tienen la ventaja de poder converger a un mejor valor, pero sin embargo lo hacen generalmente en un tiempo mayor que los que se basan en aproximar la función de valor Q. Teniendo en cuenta el objetivo que nos hemos marcado, por lo tanto, no parece la mejor opción, y por ello descartamos el uso de agentes basados en política como REINFORCE, A2C o A3C, ya que no deberíamos obtener un mejor comportamiento que con DQN en cuanto a tiempo de convergencia en entrenamiento.

Nos centraremos por lo tanto en la aplicación de distintas mejoras que se han estudiado en los últimos años sobre el agente DQN para mejorar su retorno, velocidad de convergencia y estabilidad. En concreto, vamos a incluir las siguientes mejoras, que detallaremos a continuación: Double DQN, Dueling DQN y NoisyNet DQN. Realizamos un modelo común de agente parametrizado, pues las tres mejoras son compatibles entre sí y de esta forma podemos incluir una u otra o todas las combinaciones de ellas. En esencia, realizamos algo similar a lo realizado por el equipo de DeepMind con el modelo Rainbow [3], pero sólo introduciendo estas tres mejoras. Por otro lado, hay que tener en cuenta que estos modelos están generalmente estudiados en entornos mucho más complejos que el nuestro, con un espacio dimensional mucho mayor y haciendo uso en muchos casos de redes convolucionales (juegos Atari, por ejemplo). Esto es importante porque es probable que estas mejoras no tengan un efecto tan pronunciado en un entorno mucho más sencillo como el nuestro, algo que estudiaremos.

3.1.1. Double DQN

Esta mejora se plantea en [4]. El modelo Double DQN pretende mejorar el comportamiento eliminando la sobreestimación que se produce en DQN base. Esto se hace cambiando la fórmula para obtener el valor

Figura 3.1: Arquitectura de red neuronal



del valor objetivo de Q a la siguiente:

$$y = r + \gamma \hat{Q}(s', \text{argmax}_a Q(s', a; \theta); \theta^-)$$

La modificación para introducir esta mejora radica por lo tanto exclusivamente en modificar la actualización de la ecuación de Bellman, tal y como se puede ver en el fragmento de código continuación:

```
currentQ = torch.gather(self.main_network.get_qvals(states), 1, actions)
# Get max q-values for next states from target network
nextQ = self.target_network(next_states).detach().max(1)[0].unsqueeze(1)
# Q-vals to 0 in terminal states
nextQ[dones] = 0
# Bellman equation
if self.double:
    max_nextQ = torch.max(nextQ, dim=1)[0]
    max_nextQ = max_nextQ.view(max_nextQ.size(0), 1)
    expectedQ = self.gamma * max_nextQ + rewards
else:
    expectedQ = self.gamma * nextQ + rewards
# Loss calculation
loss = F.mse_loss(currentQ, expectedQ.reshape(-1,1))
```

Listing 3.1: Actualización para Double DQN.

3.1.2. Dueling DQN

Esta mejora se plantea en [5]. El modelo Dueling DQN permite que la red neuronal haga una estimación por separado del valor del estado (independientemente de la acción tomada) y de la ventaja

de tomar una acción determinada en ese estado. Esto permite a la red aprender qué estados son o no relevantes independientemente del efecto de tomar una acción, que es importante en aquellos casos en los que cualquier acción puede ser equivalente y no modifica de forma notable el entorno, y con ello se debería conseguir un entrenamiento más rápido.

Para implementar esta mejora simplemente hay que dividir la red neuronal en una primera fase común, que en este caso será la capa de entrada y la primera capa oculta, y una segunda fase, que consiste en dos redes en paralelo para el valor del estado y la ventaja. Estas dos redes de la segunda fase tendrán como entrada la salida de la primera fase y serán paralelas. La salida de la red del valor del estado tendrá una sola salida, mientras que la de la ventaja tendrá 4, una por cada acción disponible. La salida final de la red será la suma de ambos valores, pero al valor de la ventaja se le restará previamente su media, para solucionar el problema de identificabilidad que surgiría de otro modo. La estructura de la red que se ha implementado para Dueling DQN, así como la comparativa con DQN base, se muestran en la Figura 3.1.

3.1.3. NoisyNet DQN

Esta mejora se plantea en [2]. Este modelo consiste en realizar la exploración de una forma distinta a la que se hace en DQN base. En éste, la exploración se realiza tomando una política ϵ -greedy, lo que implica que con una probabilidad ϵ se va a tomar una acción aleatoria, y no la que proporcione una mayor recompensa. Típicamente esta ϵ se irá disminuyendo a lo largo del entrenamiento, para cada vez tomar con mayor probabilidad la acción que más recompensa proporcione, pero siempre será mayor que cero en el entrenamiento. NoisyNet lo que propone es utilizar siempre una política greedy, y realizar la exploración añadiendo ruido a los pesos de la red neuronal. Este ruido añadido estará definido por su desviación típica, que será un parámetro más de la red que también podremos aprender, al igual que los pesos de la red. De esta forma, a lo largo del aprendizaje aprenderemos el ruido que tenemos que añadir a la red (a cada peso) para minimizar la función de pérdida.

Hay distintos tipos de distribución de ruido que puede añadirse a los pesos de la red, pero en que utilizaremos nosotros será un ruido blanco gaussiano. La implementación de este modelo consiste simplemente en la sustitución de las capas lineales de la red neuronal por unas lineales con ruido, que tendremos que definir de manera personalizada.

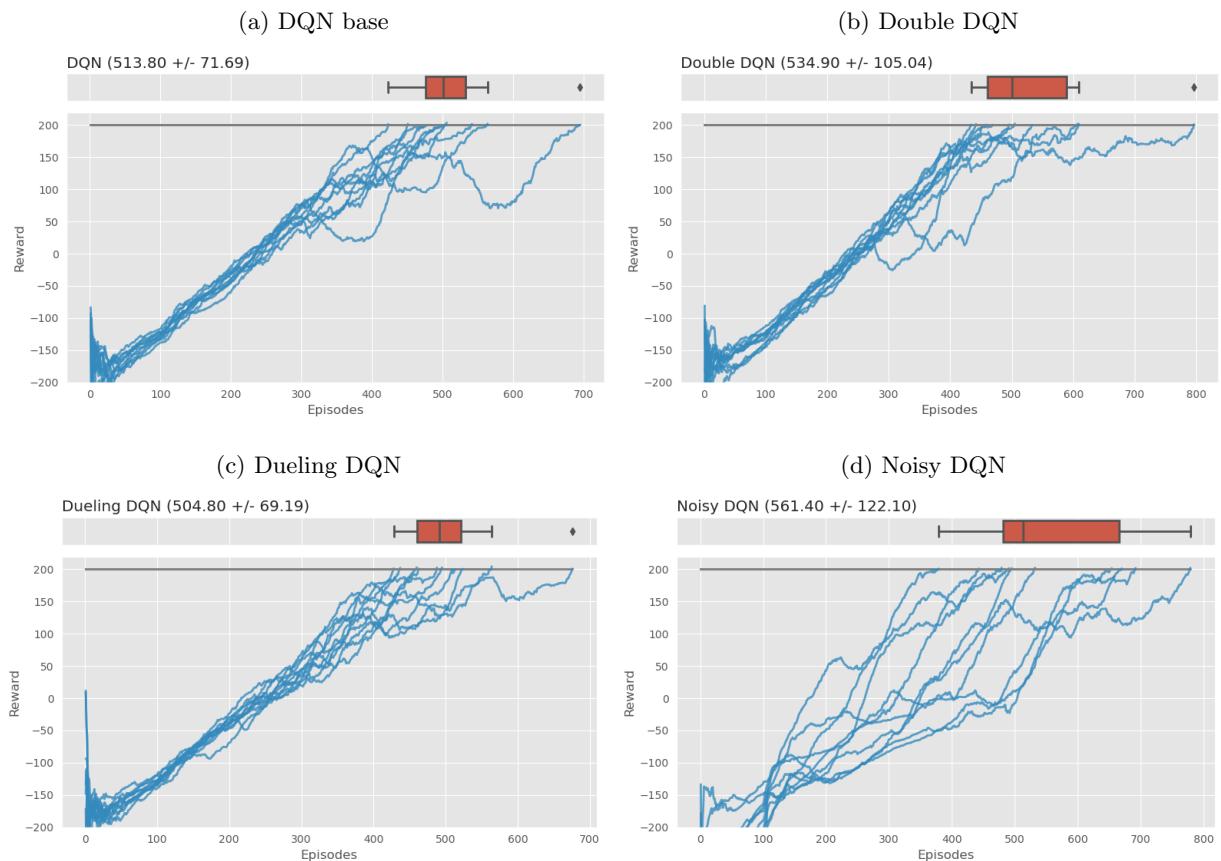
3.2. Entrenamiento y selección de hiperparámetros

En primer lugar, vamos a entrenar los modelos con las mejoras propuestas de forma individual. Para evaluar la mejora de resultado producida por cada una de ellas, mantenemos en un primer momento los mismos hiperparámetros encontrados como óptimos en el caso anterior. Además, cada caso lo entrenaremos 10 veces para poder obtener unos resultados fiables.

En la Figura 3.2 podemos ver la evolución de la recompensa media en los últimos 100 episodios en los 10 entrenamientos realizados para los casos de DQN, Double DQN, Dueling DQN y Noisy DQN. En la figura se muestra también la distribución en un diagrama de caja del número de episodios necesarios para resolver el entorno en cada uno de los casos, que como hemos comentado es el objetivo que perseguimos. Idealmente buscamos que tanto la media como la desviación típica de este número sea lo más baja posible.

En base a lo que vemos en la Figura 3.2, podemos deducir que la introducción de Noisy DQN ha resultado en un comportamiento en el entrenamiento muy errático. Es probable que sí que reporte una mejora en entornos más complejos, pero en el nuestro no parece ser así. Vamos a descartar por lo tanto el uso de esta mejora de aquí en adelante, y nos centramos en Double DQN y Dueling DQN. Los resultados con estas dos no son especialmente prometedores tampoco, aunque estamos utilizando los mismos hiperparámetros que en el caso DQN, y pueden no ser ahora los más adecuados. Según los papers tanto de Double DQN [4] como de Dueling [5] y el Rainbow [3], con el uso de estos métodos es recomendable

Figura 3.2: Arquitectura de red neuronal



reducir el learning rate del optimizador. Vamos a comprobar si efectivamente podemos mejorar su comportamiento variando este hiperparámetro. Además, volvemos a evaluar los mismos hiperparámetros que hemos contemplado con DQN, cambiando en alguno de ellos los valores probados.

En la Figura 3.3 se pueden ver los resultados de estas pruebas. Igual que hemos hecho en DQN, nos quedamos con la siguiente selección de parámetros teniendo en cuenta los resultados obtenidos:

- **LR = 0.0005.** Efectivamente, comprobamos que ahora es conveniente disminuir el valor del learning rate a 0.0005 o incluso 0.00025, con los que obtenemos no sólo una mayor velocidad de aprendizaje, sino mayor estabilidad.
- **BATCH_SIZE = 64.** No parece que haya una gran ventaja a favor de ninguno de los valores que hemos probado, igual que nos ocurría con DQN. Mantenemos la selección del valor 64.
- **N_NEURONS = 128.** Tenemos un comportamiento muy similar a DQN, y seleccionamos 128.
- **DNN_UPD = 4.** Tanto 2 como 4 proporcionan un buen comportamiento, pero nos quedamos con 4 que es ligeramente mejor.
- **MEMORY_SIZE = 32000.** Ahora vemos un comportamiento más lógico en este hiperparámetro y comprobamos como claramente 32000 es el que mejor comportamiento nos reporta.
- **GAMMA = 0.99.** Probamos dos valores nuevos para este parámetro, pero comprobamos como 0.99 sigue siendo el mejor de ellos.

3.3. Prueba del agente y comparativa

En la Figura 3.4 podemos ver el resultado en evolución de recompensa media obtenido con la opción propuesta y la comparativa de realizar lo mismo con el agente DQN original. En ambos casos hemos realizado 10 entrenamientos para tener un resultado fiable y lo hemos realizado con los hiperparámetros óptimos encontrado para los respectivos casos.

Como se puede observar, hemos mejorado el comportamiento del agente en entrenamiento, pues este es ahora más rápido y estable. Pasamos de resolver el entorno en 572 episodios en media a hacerlo en 463, una reducción notable, y demás vemos como la desviación típica de este número de episodios es también considerablemente más baja (49 vs. 87), convirtiendo este agente en más fiable en entrenamiento. Las mejoras obtenidas, sin embargo, pueden parecer bajas en relación a la que los propios papers en las que se basan ambas mejoras Double [4] and Dueling [5] dicen conseguir. Esto puede tener muchas causas, pero la que hay que considerar con más fuerza es la del diferente escenario de prueba. Ambos papers evalúan las mejoras sobre DQN en un entorno de juego Atari, donde el espacio dimensional es mucho mayor, y por lo tanto el entorno mucho más complejo de resolver. Esto hace que las mejoras sobre DQN puedan tener un efecto más llamativo en los resultados finales conseguidos que las que se puedan conseguir con un entorno mucho más sencillo como el Lunar Lander.

En la Figura 3.5, podemos ver el histograma con las recompensas obtenidas tras la ejecución de 1000 episodios distintos, y tenemos la comparativa con lo obtenido con el agente DQN sencillo. Pese a que nuestro objetivo no era mejorar el comportamiento del agente, sino optimizar su entrenamiento, haciéndolo más rápido y robusto, vemos que también hemos logrado mejorar el comportamiento del agente en modo exploración. Hemos aumentado el promedio de la recompensa obtenida en 25 puntos pero, lo más importante, vemos como ahora los casos de recompensas muy bajas (por debajo de 100 puntos) son extraordinariamente raros, mucho más que con el agente DQN simple. En el diagrama de cajas de la derecha vemos claramente esta distribución y el mejor comportamiento que presenta el agente propuesto.

Figura 3.3: Selección de hiperparámetros con Double Dueling DQN

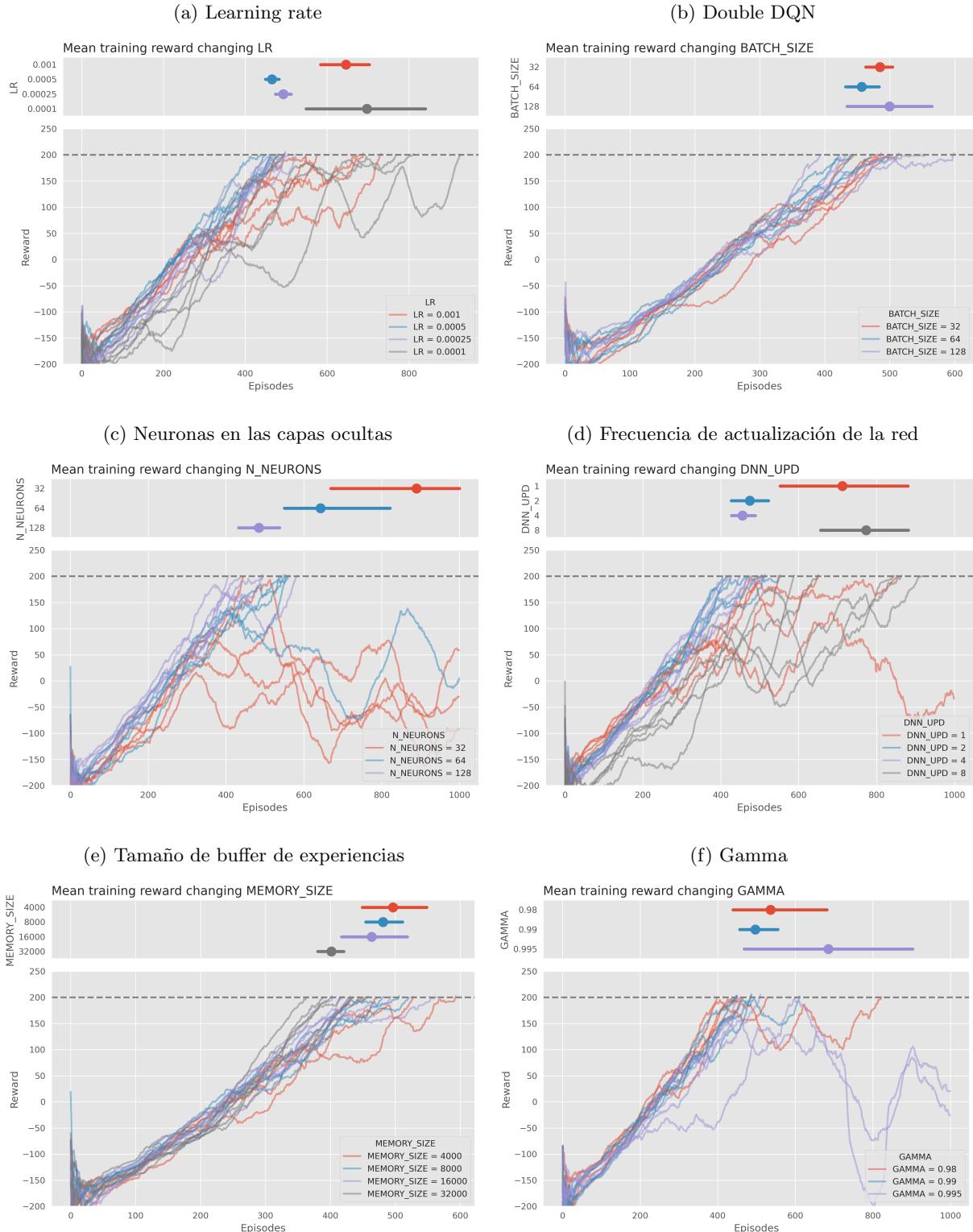


Figura 3.4: Evaluación del entrenamiento con hiperparámetros óptimos

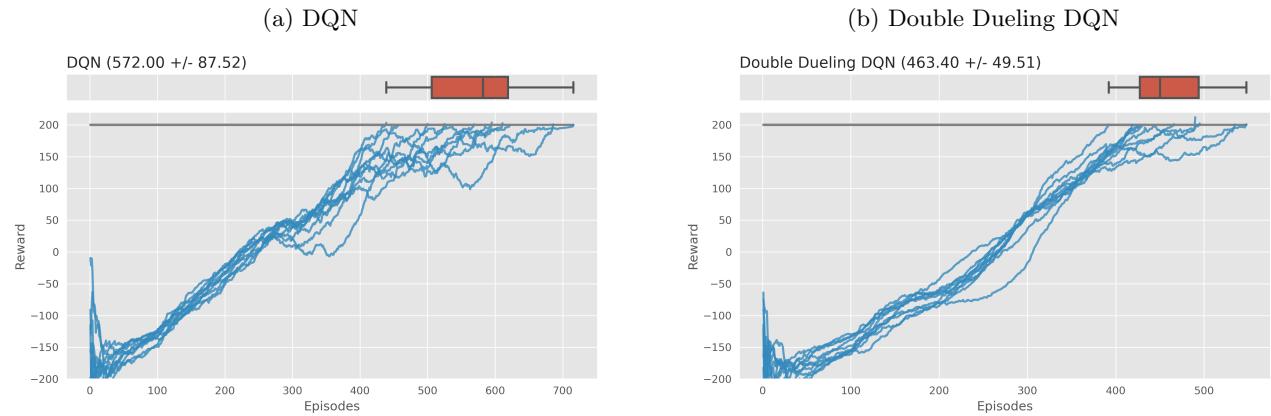
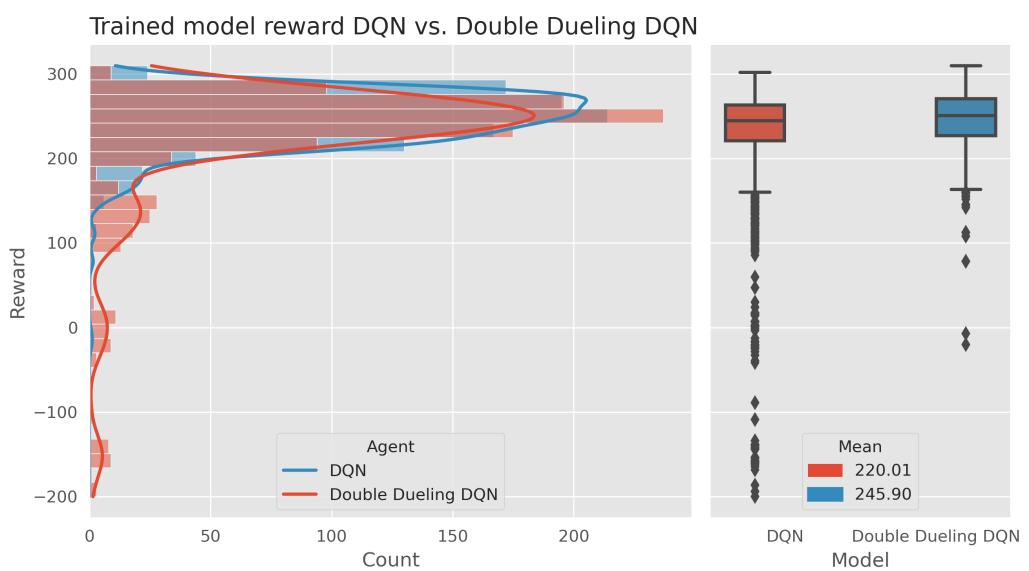


Figura 3.5: Prueba del agente entrenado



Implementación

La implementación del trabajo se ha realizado en Python, y el código se distribuye de la siguiente forma:

- **Lunar_Lander.ipynb.** Jupyter notebook utilizado para ejecutar los entrenamientos y obtener las gráficas de rendimiento asociadas. Además, en él también se incluyen ejemplos de gifs para la ejecución de episodios con cada uno de los agentes probados (aleatorio, DQN, DQN+Double+Dueling). Adicionalmente se incluye el notebook en formato HTML.
- **agents.** Directorio donde implementamos los agentes.
 - **random.py.** Implementación del agente que ejecuta acciones aleatorias.
 - **dqn.py.** Implementación del agente DQN con las mejoras propuestas en la presente práctica. Se incluyen tanto el propio agente como la clases necesarias (red neuronal, buffer de experiencias, etc).
- **utils.** Directorio donde implementamos diversas funciones de utilidad.
 - **plot.py.** Implementación de funciones para realizar los gráficos.
 - **utils.py.** Implementación de funciones accesorias, como las utilizadas por ejemplo para realizar experimentos con hiperparámetros.
- **img.** Directorio con las imágenes utilizadas en esta práctica.
- **training_stats.** Directorio con archivos json que contienen los resultados de la ejecución de los diversas selecciones de hiperparámetros.
- **videos.** Directorio con los videos resultado de la ejecución de los agentes.

Bibliografía

- [1] Lunar lander. URL https://www.gymlibrary.dev/environments/box2d/lunar_lander/.
- [2] Meire Fortunato, Mohammad Gheshlaghi Azar, Bilal Piot, Jacob Menick, Ian Osband, Alex Graves, Vlad Mnih, Remi Munos, Demis Hassabis, Olivier Pietquin, Charles Blundell, and Shane Legg. Noisy networks for exploration, 2017. URL <https://arxiv.org/abs/1706.10295>.
- [3] Matteo Hessel, Joseph Modayil, Hado van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Azar, and David Silver. Rainbow: Combining improvements in deep reinforcement learning, 2017. URL <https://arxiv.org/abs/1710.02298>.
- [4] Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning, 2015. URL <https://arxiv.org/abs/1509.06461>.
- [5] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado van Hasselt, Marc Lanctot, and Nando de Freitas. Dueling network architectures for deep reinforcement learning, 2015. URL <https://arxiv.org/abs/1511.06581>.