Entrenamiento de agente en entorno Lunar Lander de libreria Gym

Marcelo Contreras, Alejandro Del Rio, Mauricio Rivera y Mirella Rivas

Noviembre del 2022

1. Introducción

El proyecto que se desarrolla en el presente informe tiene por objetivo encontrar la política que maximice la recompensa otorgada por el entorno de Lunar Lander de la librería gym [1]. El objetivo del entorno es aterrizar una nave espacial en un área delimitada por dos banderas en el menor tiempo posible. Fueron utilizadas dos técnicas de Reinforcement Learning para la obtención de la política, las cuales fueron: Deep Q Reinforcement learning y Tile Coding. Los resultados obtenidos por ambos métodos serán comparados y se discutirá las ventajas de implementación de cada uno.

2. Ambiente de Simulación

El ambiente con el que se está trabajando en este proyecto es el Lunar Lander-v2. Este ambiente, como lo menciona [1], sirve principalmente para realizar optimización de trayectoria de una nave que tiene que aterrizar entre dos objetivos. Esto se puede observar de manera más específica en la figura 1.

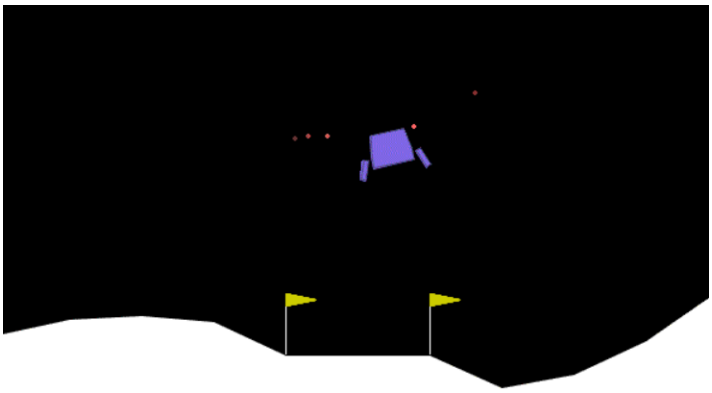

Figura 1: Ambiente de simulación de LunarLander-v2. Obtenido de [1]

Dentro de la documentación de gym se explica qué tipo de acciones y estados se disponen para el usuario. Las acciones son 4 del tipo discretas y están conformadas por: no hacer nada, propulsor izquierdo, propulsor derecho y propulsor principal. Por otro lado, los estados son del tipo continuo y es ahí donde recae la razón por la cual se utilizan los métodos mencionados en la introducción. Los estados u observaciones de este entorno representan la posición x,y, el ángulo θ, las respectivas derivadas de las anteriores variables y dos valores booleanos para cada pata de la nave que indica contacto con el suelo.

Ambos espacios se resumen en la tabla 1:

Espacio de acción	Discreto[int,4]
Espacio de observación	[float,8]
Límite superior de observación	[1.5 1.5 5. 5. 3.14 5. 1. 1.]
Límite inferior de observación	[-1.5 -1.5 -5. -5. -3.14 -5 0. 0.]

Cuadro 1: Descripción del entorno

Las recompensas que puede dar el entorno varían según el estado en el que se encuentre la nave. En primer lugar, si la nave llega a la zona de aterrizaje y se detiene se tienen de +100 a +140 puntos. Si se sale de la zona luego de aterrizar, se pierde la recompensa. En segundo punto, si la nave choca se penaliza con -100 puntos, pero si se detiene puede llegar a obtener 100 puntos. En tercer lugar, cada pata, al tener contacto con el suelo, recibe un bono de +10 puntos. Finalmente, el accionamiento del motor principal es castigado con -0.3 por cada frame donde se ejecute y de manera similar pasa para los motores de los costados con -0.03 de penalización. Al terminar todo el proceso y llegar a la zona de aterrizaje, la recompensa final es de 200 puntos.

Existen tres condiciones de parada para este entorno:

- La nave se choca
- La nave sale fuera de los límites del frame que están conformados por un cuadrado de (1,1)
- La nave no es accionada y por lo tanto, no se mueve al iniciar el programa.

3. Metodología

3.1. Diferencias temporales y Q-learning

Como el objetivo es encontrar la política que maximice la recompensa esperada, es necesario encontrar la función de estado-acción que contenga esta política en sí. Existen diferentes formas de ir actualizando esta función de valor de forma óptima considerando las recompensas que ofrece el entorno. Dentro del estado de arte se ha tenido un alto interés en la técnica de Q-learning, la cual es una extensión de diferencias temporales (TL). Este último algoritmo no tiene que esperar a que todo el episodio termine para actualizar, sino que puede hacerlo en cada conjunto (estado,acción) considerando así la influencia del pasado en la actualización.

Así mismo, no necesita un modelo matemático del entorno sobre el cual trabajar. La fórmula que rige a TL para la obtención de políticas que suele conocerse cómo SARSA es igual a:

$$Q(S_t.A_t) \leftarrow Q(S_t.A_t) + \alpha(R_{t+1} + \gamma Q(S_{t+1}.A_{t+1}) - Q(S_t.A_t))$$

donde α determina la ponderación de la actualización y γ determina la influencia del conjunto (estado,acción) del futuro.

La propuesta del Q-learning sobre DT es siempre escoger la recompensa asociada al estado S_{t+1} para todas sus acciones posibles en el componente de actualización. De esta forma se asegurar que el agente tenga una optimización más rápida mientras sigue considerando la información temporal. Por lo tanto, la fórmula Q-learning tendrá forma de:

$$Q(S_t.A_t) \leftarrow Q(S_t.A_t) + \alpha(R_{t+1} + \gamma \max_a Q(S_{t+1}.a) - Q(S_t.A_t)) \quad (1)$$

Por otra parte, al inicio de cada episodio se debe samplear una acción que se encarga de predecir la siguiente recompensa y estado. Esto sampleo se realizará siguiendo un esquema ϵ-greedy donde existe una probabilidad de sampleo asociada a la acción que máxima la recompensa para un estado en particular y otra probabilidad para las demás acciones. Esta última probabilidad puede ser actualizada por un valor de decaimiento y en sí se suele disminuir conforme avanza la simulación. De esta forma, se da la oportunidad de explorar primero el espacio de acciones y luego refinar los resultados para así asegurar la máxima convergencia posible. Su conjunto de probabilidades del esquema ϵ-greedy esta dado por:

$$\pi(a|s) = \begin{cases} 1 - \epsilon + \frac{\epsilon}{N} & . \; si \; a \; = \; \arg\max_a Q(S.a) \\ \frac{\epsilon}{N} & . \text{ para el resto de } a's \end{cases}$$

La serie de pasos que se deben ejecutar para el Q-learning puede ser resumido en el siguiente pseudocódigo:

Algorithm 1: Q-learning

Input : política inicial π,ϵ,α, cantidad de episodios
Output: Función de valor Q
Inicializar la función Q;
for *i in total episodios* **do**
    $\epsilon = \epsilon_i$;
    Obtener estado inicial S_0;
    t = 0;
    while S_t *no es terminal* **do**
        Escoger acción A_t de política Q siguiendo el esquema ϵ-greedy;
        Observar R_{t+1} y S_{t+1} al usar A_t;
        $Q(S_t.A_t) \leftarrow Q(S_t.A_t) + \alpha(R_{t+1} + \gamma \max_a Q(S_{t+1}.a) - Q(S_t.A_t))$;
        t = t + 1;
    end
end

El algoritmo de Q-learning es utilizado tanto dentro su versión Deep-Q-learning como en Tile Coding. La diferencia rádica en que Tile Coding aproxima el espacio de estados para

tener una función de valor Q tabular mientras que la formulación Deep busca aproximar la función de forma continua a través de una red neuronal donde se actualice sus pesos en cada batch.

3.2. Deep Q-Learning

En este enfoque, la función de valor Q es aproximada por una red neuronal que para propósito de este proyecto es del tipo MLP (Multi Layer percepetron) o mejor conocida como Dense, ya que se las entradas(estados) y salidas(acciones) de esta red son vectores. Dentro del entrenamiento del agente, se tendrá que enviar un par (estado, acción) donde la red debe predecir lo mejor posible la acción óptima para el estado dado. Por lo tanto, nos encontramos con una red regresora. Según se consiga este objetivo, se actualizaran los pesos dentro de la red según una función de costo y un optimizador de elección variada. La estructura de una red MLP se puede observar en 2.

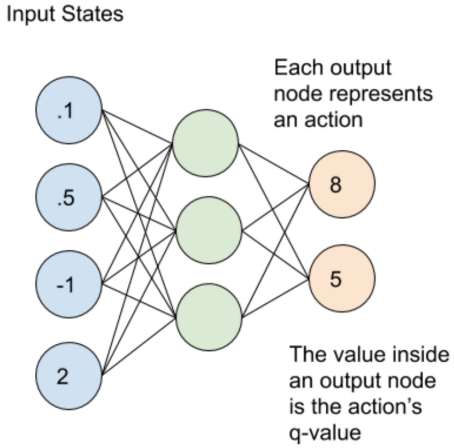

Figura 2: Esquema de red neuronal de Deep Q-learning obtenido de [2]

Las ventajas que ofrece el Deep Q-learning es el hecho de que es un aproximador de funciones generalizado en donde los pesos se combinan con las entradas de forma lineal pero se les aplica un componente de rectificación no-lineal que permite tener una actualización simple pero que se adapta a muchos casos. Asimismo, el marco de trabajo del Deep Learning se ha expandido mucho en los últimos años y se ha vuelto más fácil la implementación de redes de pequeñas dimensiones mediante librerías como Torch o Tensorflow.

Sin embargo, las desventajas es que se debe diseñar adecuadamente la red aproximadora y se debe tunear una serie de parámetros. Adicionalmente, el tiempo de entrenamiento es considerablemente mayor al de Tile Coding. Para poder obtener una gama de resultados, este trabajo ha realizado múltiples simulaciones con distintos valores de learning rate para la red que determinan que tan rápido se desea que la red aprenda.

La red en cuestión que se ha diseñado a consideración los resultados de los repositorios de [3, 4] y esta compuesta por la siguiente configuración realizada en TensorFlow.

Red	Descripción
Input	**Input.shape = (,8)**
Capa 1	**Dense.layer(128), Activación = RELU**
Capa 2	**Dense.layer(64), Activación = RELU**
Output	**Dense.layer(4). Activación = Linear**
Optimizador	RMSprop con loss = MSE
Batch-size	*64*
Learning-rate	*Variable*

Cuadro 2: Configuración de red neuronal

Parámetros de RL	Valor
γ	*0.99*
ϵ	*0.5*
ϵ_{decay}	0.998
α	*1*

Cuadro 3: Parámetros de Deep Q-learning

Finalmente cómo detalle, el enfoque Deep Q-learning necesita almacenar varios pares (estado,acción) antes de entrenar y hace uso de lo que se conoce como action-replay que es una memoria desde donde se samplea el par mencionado hasta conseguir un número dado por el batch-size y recién este conjunto se envía a la red para entrenar.

Con la red diseñada, los hiper-parámetros escogidos, la función de actualización, ϵ-greedy y el action replay, se tienen todos los elementos listos para ejecutar el Deep Q-learning

3.3. Tile Coding

El algoritmo de Deep Q-Learning descrito en la sección anterior nos permite trabajar con los valores continuos de los estados. Sin embargo, existen métodos que permiten discretizar este espacio continuo de manera que se pueda aplicar Q-Learning convencional. Para este trabajo se utilizó el algoritmo de Tile Coding.

El algoritmo de Tile Coding consiste en aplicar una serie de *tilings* o cuadrículas de tamaño uniforme al espacio continuo de los estados. Usar varias cuadrículas permite mantener la discretización precisa sin comprometer mucho el costo computacional.

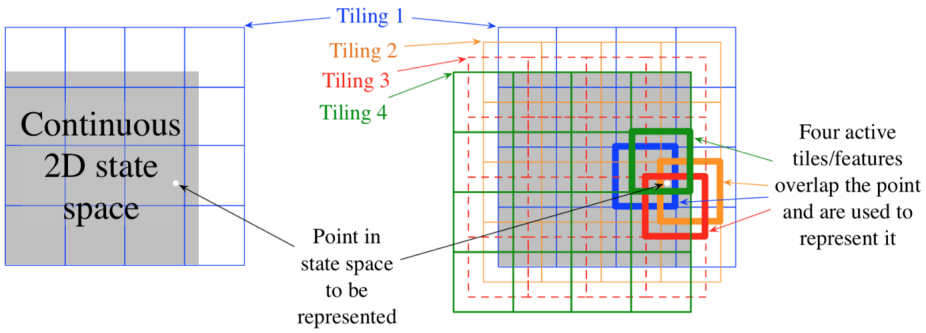

Figura 3: Ejemplo de discretización usando Tile Coding en 2 dimensiones.

Para la aplicación del Lunar Lander, los estados consisten en vectores de 8 valores: x, y, $\dot{x}$, $\dot{y}$, θ, $\dot{\theta}$ y 2 booleanos que indican si cada una de las patas tocaron el piso. Se aplicó el algoritmo a los 6 valores continuos y se agregó un índice que indique el estado de los 2 booleanos.

El número de tilings y la cantidad de tiles por cada lado fueron definidas en base al límite de almacenamiento que permite Python en una variable. Esto se debió a que se está trabajando con un espacio hexadimensional, lo que aumenta enormemente el número de estados únicos que se pueden definir. Es necesario contar con todas las variables pues todas determinan el efecto que tendrá una determinada acción en el siguiente estado, como el ángulo θ por ejemplo, del cual depende que el efecto real de encender los propulsores.

Mediante prueba y error, se determinó que 2 tilings y 4 tiles eran valores computacionalmente manejables. De esta manera, la tabla Q tiene dimensiones [4914, 9829, 4, 4], considerando las 4 posibles acciones. Esto da un total de 772 795 296 estados-acción.

Para el entrenamiento del modelo se plantea usar los siguientes parámetros:

Parámetro	Valor
γ	0.99
ϵ	0.5
ϵ_{decay}	0.0005
α	0.005

Cuadro 4: Parámetros de Q-Learning con el algoritmo de Tile Coding

4. Resultados

Para la ejecución de ambos modelos se utilizaron el entorno de Colab haciendo uso de su GPU que por defecto es una NVIDIA Tesla T4, y una laptop para correr el programa de forma local (se resalta que este último también se utilizó para realizar el renderizado del cohete utilizando el mejor modelo, dado que este no podia ser ejecutado en Colab).

4.1. Deep Q-Learning

Como los resultados finales de esta metodología dependen de un tunning adecuado de los hiper parámetros, se optó por realizar una serie de simulaciones probando distintos valores de learning-rate para luego ejecutar la red con la configuración que dio mejores resultados por una cantidad mayor de episodios.

De este primer conjunto de pruebas se obtuvieron los siguientes resultados:

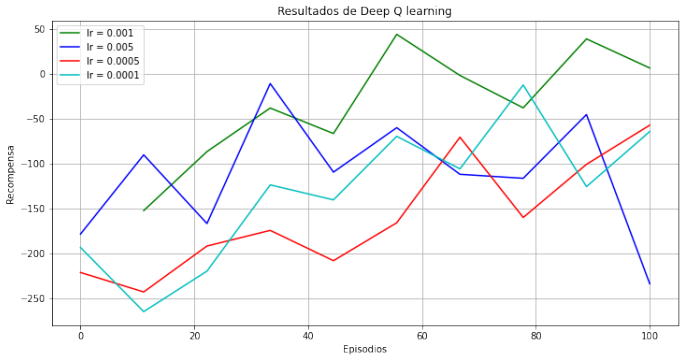

Figura 4: Simulación para distintos valores de Learning Rate

Dentro de este conjunto, el valor lr = 0.001 obtuvo los mejores resultados con una recompensa pico de 50 y una recompensa final de 5 a partir del valor inicial de -175. Se resalta que es probable que las otros valores también pudieran haber dado mejores o peores resultados para una mayor cantidad de episodios. Sin embargo, el entorno de Colab restringió la cantidad de tiempo de simulación y este también esta afectado por el learning rate en sí. Así mismo, si se deseaba mejorar aun más la recompensa final, se pudo haber hecha una serie de combinatorias con otros hiper-parámetros cómo γ o el valor inicial de ϵ, considerando que los tiempos de simulación aumentarán considerablemente debido a las combinaciones posibles.

En 5 se puede observar a mayor detalle los resultados con lr = 0.001 en donde se aplicó un filtro promediador de cada 10 muestras y la simulación se extendió hasta 150 episodios, llegando hasta una recompensa pico de 150 y un valor final de 100. Este modelo entrenado fue exportado en un archivo de formato .h5 que puede ser nuevamente cargado con el entorno y fue con él cual se realizó un vídeo mostrando los resultados del renderizado en donde el Lunar Lander llegaba a aterrizar dentro de su área límite sin desviarse considerablemente o estrellarse. Este fue el vídeo presentado durante la exposición del presente proyecto en donde se consiguió una recompensa final de 236.77.

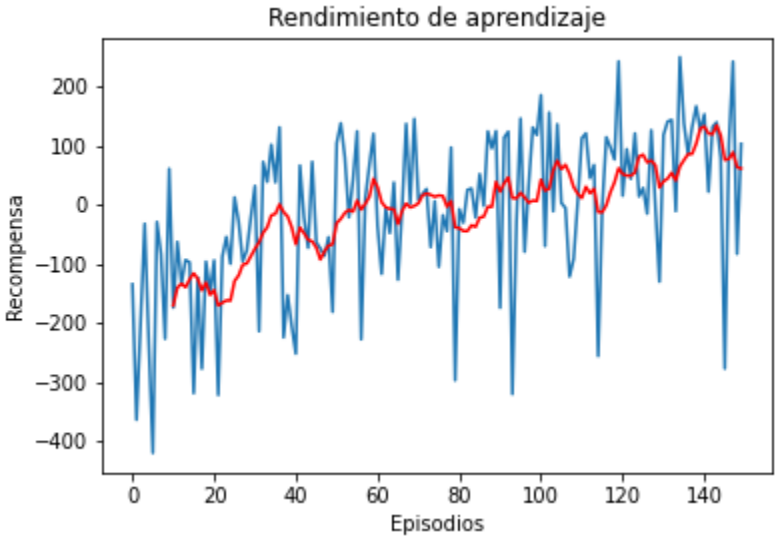

Figura 5: Detalle del resultado de entrenamiento con tasa de aprendizaje de 0.001

Adicionalmente, también se realizó una nueva simulación con la misma configuración anterior y se llegó actualizar la función de valor Q para 200 episodios consiguiendo así un valor pico de 150 para la recompensa y un valor final de 98. Esto indicaría, que la política puede seguir mejorando mientras se vaya aumentando la cantidad de episodios y así conseguir un mejor desempeño del agente.

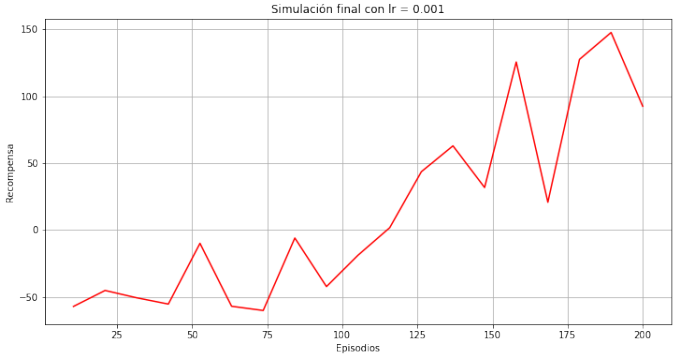

Figura 6: Simulación Final con lr=0.001

4.2. Tile Coding

El recurso computacional disponible nos permitió realizar el entrenamiento de varios modelos al mismo tiempo. En el caso particular de Tile Coding, el entrenamiento tomó 28 minutos. En la figura 7 se observa una gráfica de la recompensa promediada cada 100 episodios.

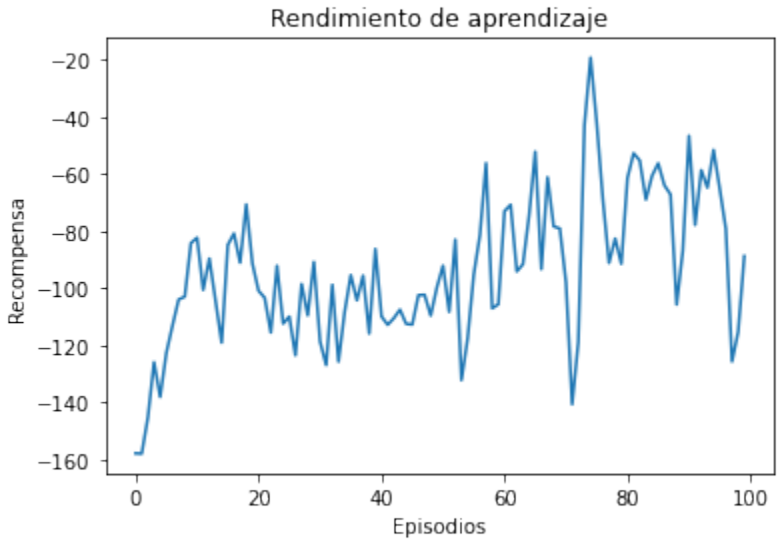

Figura 7: Recompensa obtenida con el algoritmo de Tile Coding.

Si bien la recompensa tiene una tendencia incremental a medida pasan los episodios, esta no llegó a tener valores positivos. Esto se debe a las dimensiones de la tabla Q. Cómo se mencionó en la sección anterior, esta tiene un total de 772 795 296 estados-acción. Después del entrenamiento se evaluó cuántos de estos estados habían sido visitados y sus valores modificados. Se encontró que solo 5226 estados únicos habían sido visitados. De esta manera, la gran mayoría de estados no han sido modificados y siguen teniendo un valor inicial de 0. De esta manera, si el agente llega a un estado al que nunca antes a entrado, no va a tener ninguna información de qué acción tomar. La etapa de exploración se vuelve entonces mucho más importante, sin embargo las dimensiones de la tabla Q no permiten completar el entrenamiento en un rango razonable de tiempo. Para reducir estas dimensiones, se podría realizar un tile coding adaptativo, dando más importancia a los estados donde es más probable que esté el agente.

5. Conclusiones

- Se logró conseguir la política óptima que maximiza la recompensa y que logró llevar el cohete a la posición deseada sin necesidad de hacer la elección de features que describan mejor la relación acción-estado mediante el Deep Q-learning.
- Entre los cuatro valores de tasa de aprendizaje evaluados, se concluye que el valor óptimo es de 0.001. Es crucial la correcta elección de hiper parámetros para obtener un buen rendimiento y para no requerir a una gran cantidad de episodios para el entrenamiento.
- No fue ideal usar Tile Coding para el modelo que considera muchas variables pues el número de estados acción aumenta exponencialmente con cada dimensión. No es un método escalable. Se recomienda el uso de un Tile Coding adaptativo que pueda tener

una grilla que samplee de forma más "inteligente" el espacio de estados sin sobreutilizar el recurso de memoria.

Referencias

[1] *Gym Documentation. Lunar Lander. Recuperado de https://www.gymlibrary.dev/environments/box2d/lunar*$_l$*ander/*

[2] *Verma S (2019). Train Your Lunar-Lander — Reinforcement Learning — OpenAIGYM. Recuperado de https://shiva-verma.medium.com/solving-lunar-lander-openaigym-reinforcement-learning-785675066197*

[3] *Fakemonk1. Github. Reinforcement-Learning-Lunar*$_L$*ander.Recuperadodehttps* : *//github.com/fakemonk*1*/Reinforcement* − *Learning* − *Lunar*$_L$*ander/blob/master/Lunar*$_L$*ander.py*

[4] *Wang, M. (2020). Deep Q-Learning Tutorial: minDQN. Recuperado de https://towardsdatascience.com/deep-q-learning-tutorial-mindqn-2a4c855abffc*