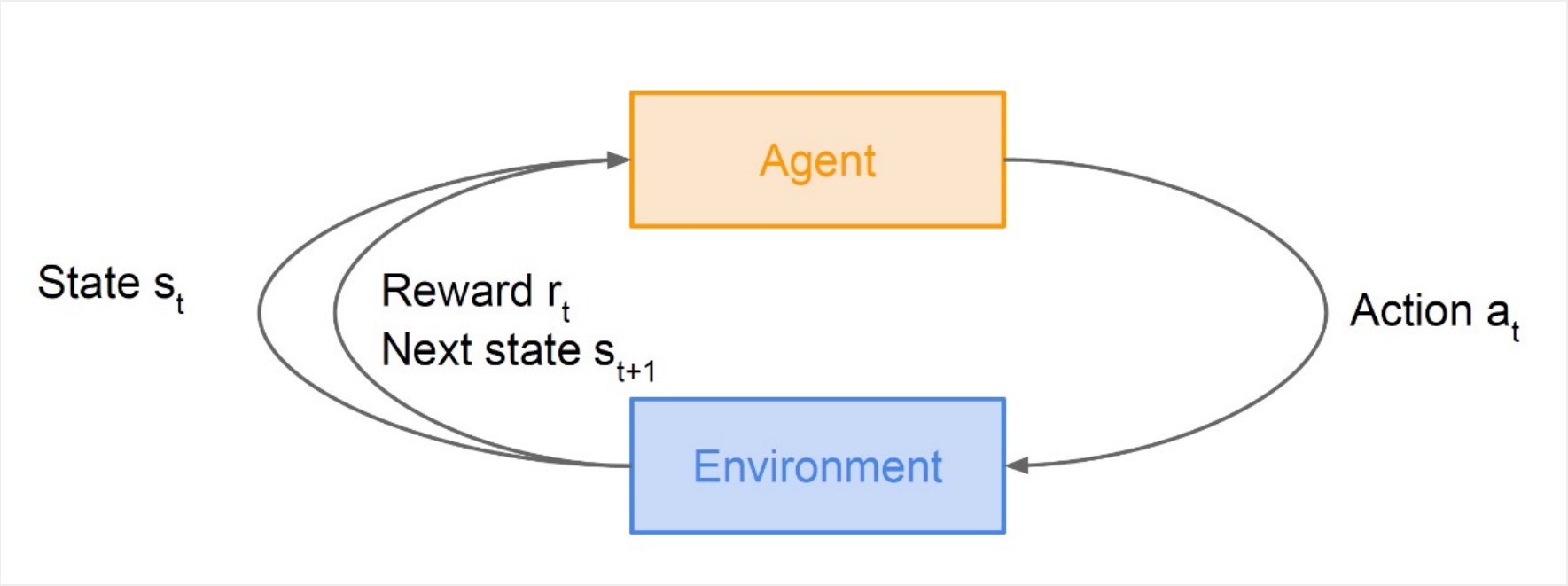
Final Teoría de Juegos

Sebastian Castaño Cruz y Germán Wiens
5 de Agosto de 2022
Aprendizaje por refuerzo para resolución de Juegos con Deep Q Learning

Reinforcement Learning

Que es el Reinforcement Learning (RL)?

- RL es un área del machine learning que se ocupa de cómo los **agentes** inteligentes deben realizar **acciones** en un **entorno** o **ambiente** para maximizar la noción de **recompensa** acumulativa.
- El RL es uno de los tres paradigmas básicos del machine learning, junto con el aprendizaje supervisado y el aprendizaje no supervisado.
- Objetivo: Aprender a como tomar acciones de tal forma de maximizar la recompensa.



Tenemos un agente y un ambiente

1. El ambiente le da al agente un estado.

2. El agente ejerce una acción.

3. El ambiente va a devolver una recompensa así como el siguiente estado.

y asi sucesivamente... $(s_0, a_0, r_1, s_1, a_1, r_2 \ldots)$

Formalización Matemática del RL

La formalización matemática de RL se hace mediante un Markov Desicion Process (MDP).

- Cumple la Propiedad de Markov que nos dice que un estado actual caracteriza completamente el estado del mundo.
- El MDP esta definido por una tupla de objetos: $(\mathcal{S},\mathcal{A},\mathcal{R},\mathbb{P},\gamma)$

$\mathcal{S}$ = Conjunto de todos los posibles estados.

$\mathcal{A}$ = Conjunto de todas las posibles acciones.

$\mathcal{R}$ = Distribución de probabilidad de los premios dados el par estado-acción.

$\mathbb{P}$ = Probabilidad de transición i.e. distribución sobre el siguiente estado dados. el par (estado, acción).

γ = Factor de descuento.

Policy y Función Valor

Policy:

- Una policy π es una función de $\mathcal{S}$ a $\mathcal{A}$ la cual especifica que acción tomar (o con que probabilidad tomar cada acción) en cada estado s

$$\pi = \{p(a_i|s) \ / \ \forall a_i \in \Delta_\pi \wedge \sum_i p(a_i|s) = 1\}$$

- El objetivo es encontrar la policy π^* que maximice la recompensa acumulada descontada.

$$R_t = \sum_{i=0}^{T-t-1} \gamma^i r_{t+i+1}$$

- Cómo encontramos la policy óptima π^*?
  Usamos funciones de valor que miden que "tan bueno" es un estado, o un par estado-acción.

Funciones de Valor: Son el valor esperado de la política π dado el estado s o el par (s, a)

- Función Valor del estado s:

$$V_\pi(s) = \mathbb{E}(R|s_t = s, \pi)$$

- Función Valor del par estado-acción (s, a):

$$Q_\pi(s, a) = \mathbb{E}(R|s_t = s, a_t = a, \pi)$$

Estas funciones nos permiten comparar dos policies π, π'

$$\pi \leq \pi' \Leftrightarrow [V_\pi(s) \leq V_{\pi'}(s)] \vee [Q_\pi(s, a) \leq Q_{\pi'}(s, a)]$$

Ecuaciones de Bellman

Basados en la recompensa acumulada podemos expandir $V_\pi(s)$ y $Q_\pi(s,a)$ para representar la relación entre dos estados consecutivos $s = s_t$ y $s^{'} = s_{t+1}$.

$$V_\pi(s) = \sum_a \pi(s,a) \sum_{s'} p(s'|s,a)(\mathbb{W}_{s \to s'|a} + \gamma V_\pi(s'))$$

$$Q_\pi(s,a) = \sum_{s'} p(s'|s,a)(\mathbb{W}_{s \to s'|a} + \gamma V_\pi(s'))$$

Donde: $\mathbb{W}_{s \to s'|a} = \mathbb{E}[r_{t+1}|s_t = s, a_t = a, s_{t+1} = s']$

Resolviendo alguna de estas ecuaciones, podemos hallar, la funcion valor $V(s)$ o $Q(s,a)$, respectivamente.

Policy Óptima π^*

Las ecuaciones de Bellman sobre la policy óptima π^* son:

$$V_{\pi^*}(s) = \max_a \sum_{s'} p(s'|s,a)(\mathbb{W}_{s \to s'|a} + \gamma V_{\pi^*}(s'))$$

$$Q_{\pi^*}(s,a) = \sum_{s'} p(s'|s,a)(\mathbb{W}_{s \to s'|a} + \gamma \max_{a'} Q_{\pi^*}(s',a'))$$

Si se conocen todas las dinámicas del problema se pueden aproximar las soluciones usando programación dinámica (sobre cada estado o par estado-acción). Pero para ambientes complejos y con muchos estados es impracticable.

Tipos de métodos de RL

Model-Free:

- El agente no conoce las probilidades de las transiciones.
- Para que el agente aprenda las transiciones entre estados, debe explorarlos y recorrer varias veces cada estado y acción.
- Es útil para los ambientes con dinámicas difíciles de modelar.

Model-Based:

- Requiere un modelo de las dinámicas del ambiente que pueda representar todas las transiciones entre estados.
- El agente explora sólo para mejorar sus policies.
- Puede acelerar el entrenamiento.

Tipos de métodos de RL

Métodos Off-Policy:
Buscan estimar el valor al tomar una cierta acción en un cierto estado, para después escoger la mejor opción.

- Es necesario exploración.
- Busca aprender la función Q.
- Tiene más eficiencia muestral.
- DQN, HER

Métodos On-Policy:
Buscan estimar la mejor policy para cada estado π^*.

- Toma como input la representación del mundo y devuelve como output una acción, donde la acción es estocástica.
- Tiene menos eficiencia muestral.
- Policy Optimization, A2C, PPO.

Método de Montecarlo

- Es model-free
- Simular episodios y guardar el valor promedio para cada estado s o cada par (s, a).

$$V_\pi^{MC}(s) = \lim_{i \to +\infty} \mathbb{E}[r^i(s_t) \mid s_t = s, \pi]$$

$$Q_\pi^{MC}(s, a) = \lim_{i \to +\infty} \mathbb{E}[r^i(s_t, a_t) \mid s_t = s, a_t = a, \pi]$$

- Se utiliza una estrategia $\varepsilon - \textit{greedy}$ para mejorar las policies.

$$R : \pi \to \pi' = \begin{cases} 1 - \varepsilon + \frac{\varepsilon}{|\Delta_\pi|} & a_i = a_j \ \wedge \ j = \arg\max_k Q_\pi(s, a_k) \\ \frac{\varepsilon}{|\Delta_\pi|} & \forall a_i \in \Delta_\pi \ \wedge \ a_i \neq a_j \end{cases}$$

Método de Diferencias Temporales

- También es model-free.
- No espera a terminar un episodio para actualizarse sino que lo hace a cada paso (puede converger más rápido).
- Estima el valor como una combinación convexa entre los dos lados de la ecuación de Bellman en el paso anterior.

$$V^i(s_t) \leftarrow \alpha V^{i-1}(s_t) + (1-\alpha)(r_{t+1} + \gamma V^{i-1}(s_{t+1}))$$

$$Q^i(s_t, a_t) \leftarrow \alpha Q^{i-1}(s_t, a_t) + (1-\alpha)(r_{t+1} + \gamma Q^{i-1}(s_{t+1}, a_{t+1}))$$

- Q-learning busca estimar el valor de la función Q sobre la policy óptima π^*

$$Q^i(s_t, a_t) \leftarrow \alpha Q^{i-1}(s_t, a_t) + (1-\alpha)(r_{t+1} + \gamma \max_{a^j_{t+1}} Q^{i-1}(s_{t+1}, a^j_{t+1}))$$

Pero...

Tanto el método de Montecarlo como el de diferencias temporales son TABULARES

$\Longrightarrow$ NECESITAN DEMASIADA MEMORIA :(

Método DQN

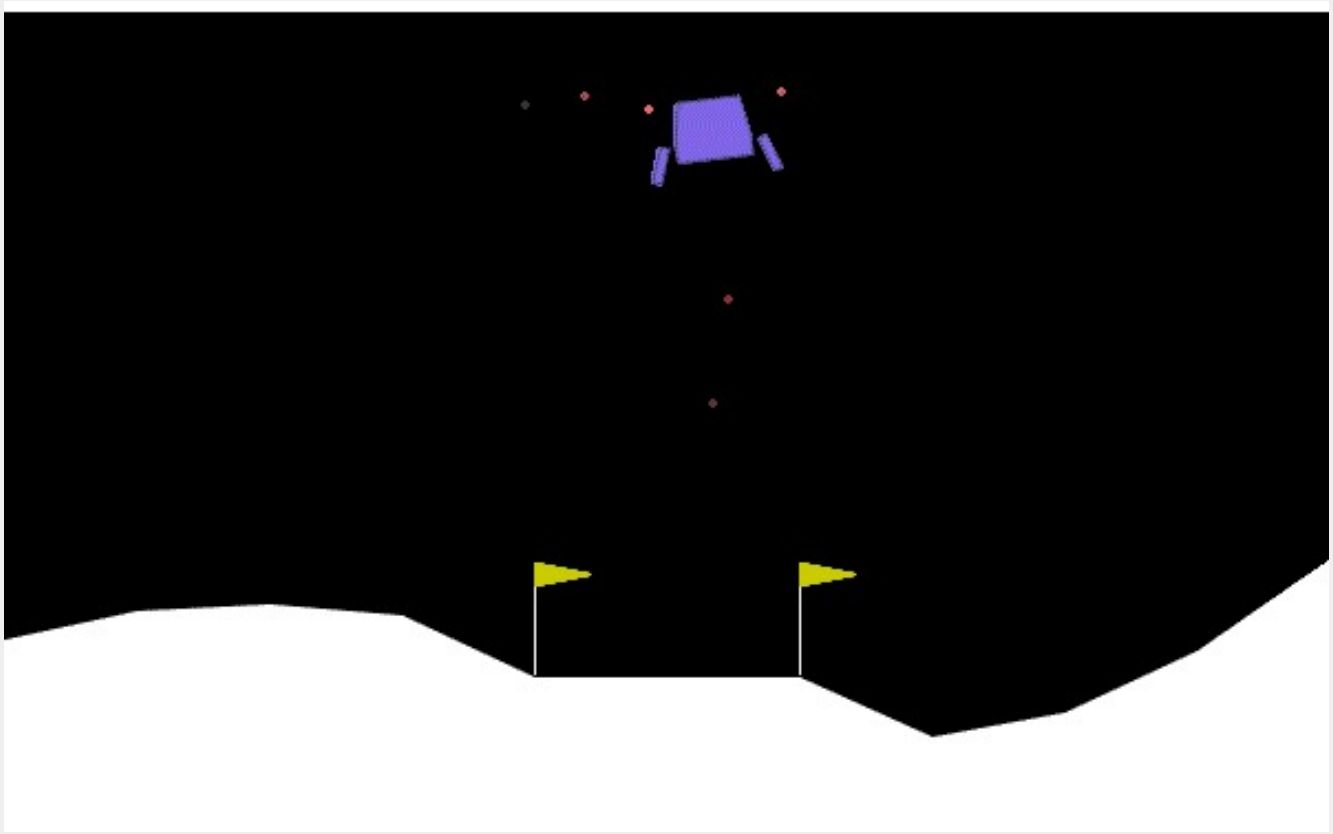
- La idea es estimar el valor de la función Q sobre la policy óptima mediante una red profunda $Q(s, a|\theta)$.
- A partir de la ecuación de Bellman se define la función de pérdida

$$\mathcal{L}(\theta) = \mathbb{E}[(r_{t+1} + \gamma \max_{a'} Q(s_{t+1}, a'|\theta) - Q(s_t, a_t|\theta))^2]$$

- Usar la misma red para entrenar y fitear es inestable $\implies$ se usa una red *target* $Q(s, a|\theta^-)$ que se actualiza cada N pasos.
- El entrenamiento se realiza sobre la memoria de los estados, acciones y recompensas recorridas. Para evitar correlación se toma una muestra aleatoria de la memoria y se entrena sobre ella.

$$\begin{cases} \mathcal{L}(\theta) = \mathbb{E}[(r_{t+1} + \gamma \max_{a'} Q(s_{t+1}, a'|\theta^-) - Q(s_t, a_t|\theta))^2] \\ \theta^- \leftarrow \theta \ \ \textit{cada} \ \ N \ \ \textit{pasos} \end{cases}$$

Juego: Lunar Lander



Objetivo: Aterrizar el cohete en el espacio delimitado por las banderas.

Descripcion del Juego

$\mathcal{A}$= Conjunto de acciones.

- No hacer nada.
- Encender el motor a la izquierda.
- Encender el motor a la derecha.
- Encender el motor principal (ir hacia arriba).

$\mathcal{S}$= Conjunto de Estados.

- Coordenadas de la nave (x, y)
- Velocidades lineales en x y en y.
- Ángulo y velocidad angular
- Dos booleanos que representan si cada pierna del cohete tocó el suelo o no.

Premios:

- La recompensa por moverse desde la parte superior de la pantalla hasta la plataforma de aterrizaje y detenerse es de aproximadamente 100-140 puntos. Si la nave se aleja de la plataforma de aterrizaje, pierde el premio.
- Si la nave se estrella, recibe -100 puntos adicionales.
- Si llega a descansar, recibe +100 puntos adicionales.
- Cada pierna en contacto con el suelo es +10 puntos.
- Disparar el motor principal es -0.3 puntos cada frame.
- Disparar algún motor lateral es -0.03 puntos cada frame.
- Se considera resuelto con 200 puntos o más.

Estado Inicial y Terminación

Estado Inicial: El módulo de aterrizaje comienza en el centro superior de la ventana gráfica con una fuerza inicial aleatoria aplicada a su centro de masa.

Cada nuevo ambiente tiene una nueva fuerza inicial y un nuevo relieve en el piso.

En nuestro caso siempre iniciamos el ambiente con la misma semilla, es decir, nuestro agente sólo conoce un estado inicial y un relieve de piso. Lo hicimos para acelerar el proceso.

Terminación:

- Si la nave se estrella (el cuerpo del módulo de aterrizaje entra en contacto con la luna).
- Si la nave sale de la ventana gráfica (la coordenada x es mayor que 1).
- Si no se mueve y no choca con ningún otro cuerpo.

Nuestra DQN

- Para estimar Q usamos una red neuronal con dos capas ocultas cada una de 64 neuronas y una capa final de cuatro salidas. En total tiene 4996 parámetros. El input es una capa de 8 entradas.

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 64)                576

 dense_1 (Dense)             (None, 64)                4160

 dense_2 (Dense)             (None, 4)                 260

=================================================================
Total params: 4,996
Trainable params: 4,996
Non-trainable params: 0
```

$$Parmetros = output_{size} * (input_{size} + 1)$$

Nuestra DQN

- la entrada corresponde a los 8 estados distintos. La salida tiene que ver con las 4 acciones posibles.
- Un episodio termina si el juego termina o si dura 1000 frames (pasos).
- Las experiencias se guardan en una lista que tiene un tamaño máximo de 100000.
- De la memoria se seleccionan aleatoriamente 64 experiencias y se utilizan para entrenar la red.
- Cada 10 pasos la red *target* se actualiza. La actualización es suave:

$$\theta^- = \tau\theta + (1 - \tau)\theta^-$$

- Utilizamos una estrategia $\varepsilon - greedy$. Después de cada episodio el ε se reduce según un ε_{decay} hasta alcanzar un ε_{min}
- Consideramos que el agente resolvió el juego si el promedio de los últimos 10 episodios supera los 200 puntos.

Algorithm 1 DQN

Inicializar memoria vacía $\mathcal{D}$ con capacidad 100000
Inicializar red que aproxima Q y *target* con pesos aleatorios θ
$\varepsilon \leftarrow 1$
for $e = 1, N$ **do**
    Resetear ambiente y obtener estado s_1
    for $t = 1, 1000$ **do**
        Encontrar la acción a_t actuando sobre s_t
        Ejecutar el ambiente sobre a_t y obtener r_{t+1}, s_{t+1}
        Guardar en $\mathcal{D}$ la transición $(s_t, a_t, r_{t+1}, s_{t+1})$
        Entrenar la red sobre una muestra aletoria de $\mathcal{D}$
        $s_t \leftarrow s_{t+1}$
        Cada 10 pasos actualizar suave la red *target*
    end for
    if $\varepsilon > \varepsilon_{min}$ **then**
        $\varepsilon \leftarrow \varepsilon \times \varepsilon_{decay}$
    end if
end for

Tiempo de Ejecución

- El entrenamiento toma bastante tiempo.
- Usar siempre la misma semilla hace que el agente aprenda más rápido (el problema es más simple).
- A medida que ε se reduce cada episodio tarda más (cada juego dura más y hay que "elegir" cada acción).
- Aumentar el tamaño de la memoria no afecta significativamente el tiempo de ejecución.
- Usar actualización suave permite actualizar la red *target* con más frecuencia.
- Adaptamos el código para poder usar 3 procesadores $\Longrightarrow$ Acelera mucho la ejecución.

Resultados Obtenidos

Escogemos testear con diferentes parámetros: ε_{decay}, γ. Las configuraciones que hemos probado son:

Con $\gamma = 0,99$

ε_{decay}	Episodios	ε final
0,95	No CV	$\leq 0,01$
0,975	470	$\leq 0,01$
0,99	437	0,01237
0,995	449	0,105
0,9995	2774	0,249

Con $\varepsilon_{decay} = 0,995$

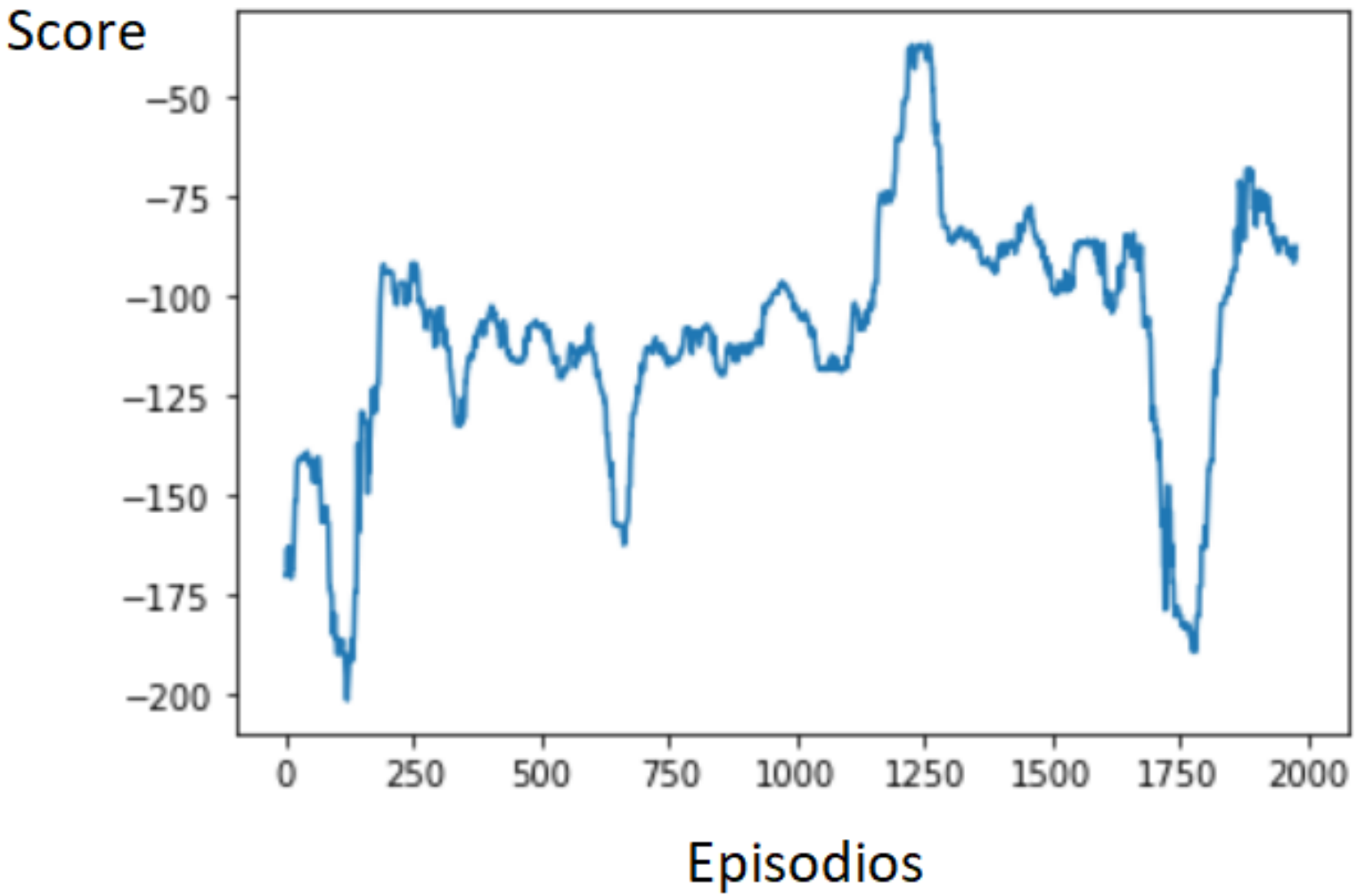
γ	Episodios	ε final
0,9	No CV	$\leq 0,01$
0,95	No CV	$\leq 0,01$
0,975	834	0,015
0,99	449	0,105
0,995	481	0,089

Para algunos de ellos el método logró resolver el juego y en otros no se lograba solución.

Cómo no converge?

```
update target model
episode: 1043   score: -127.4332526723157   epsilon: 0.00998645168764533 Factor Discount: 0.9
update target model
update target model
update target model
update target model
update target model
update target model
update target model
update target model
update target model
update target model
update target model
update target model
update target model
update target model
update target model
update target model
update target model
update target model
update target model
update target model
update target model
update target model
update target model
update target model
update target model
update target model
update target model
update target model
update target model
update target model
update target model
update target model
update target model
update target model
episode: 1044   score: -121.92629828219688   epsilon: 0.00998645168764533 Factor Discount: 0.9
update target model
```

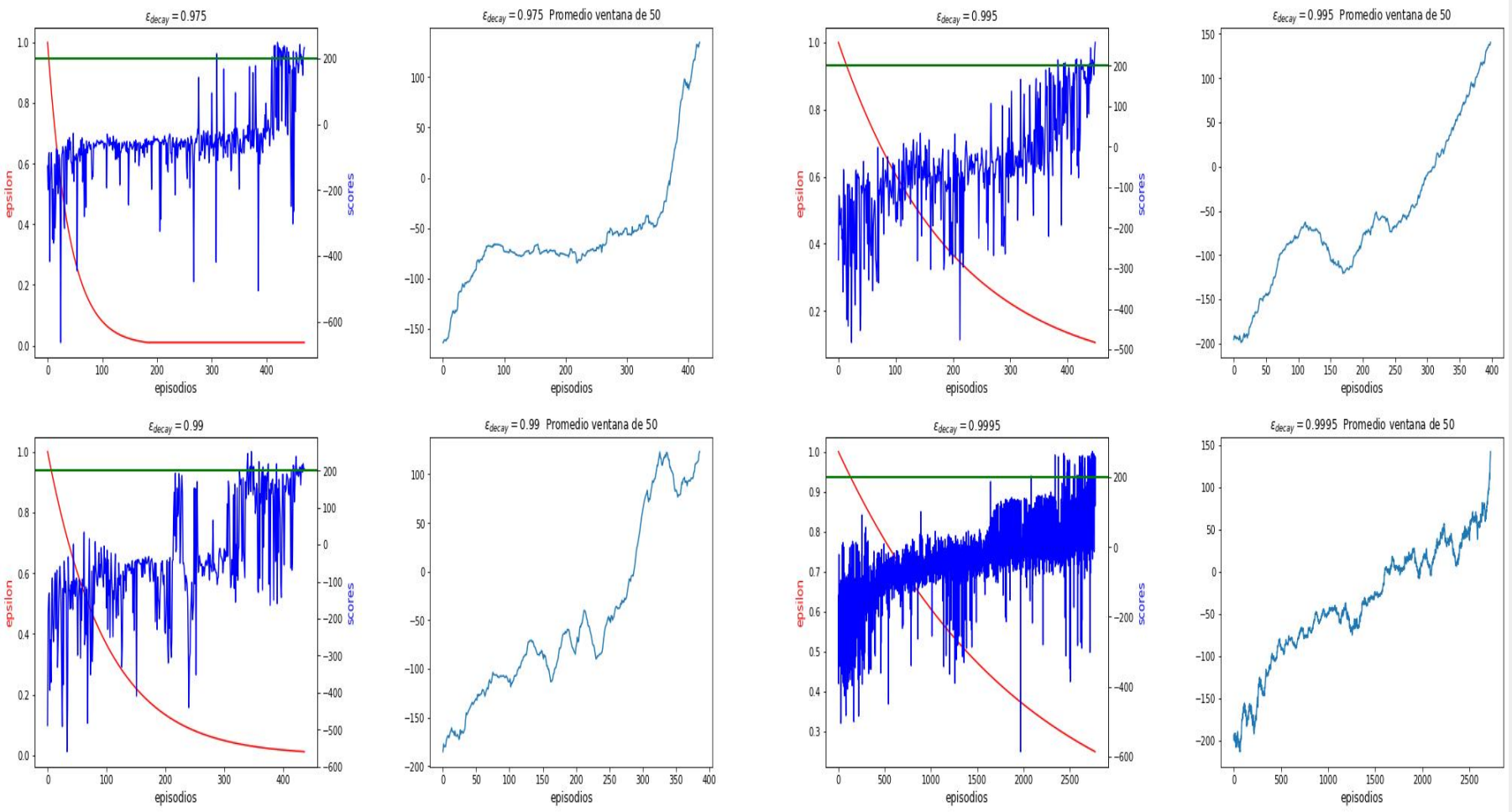
Cómo no converge?



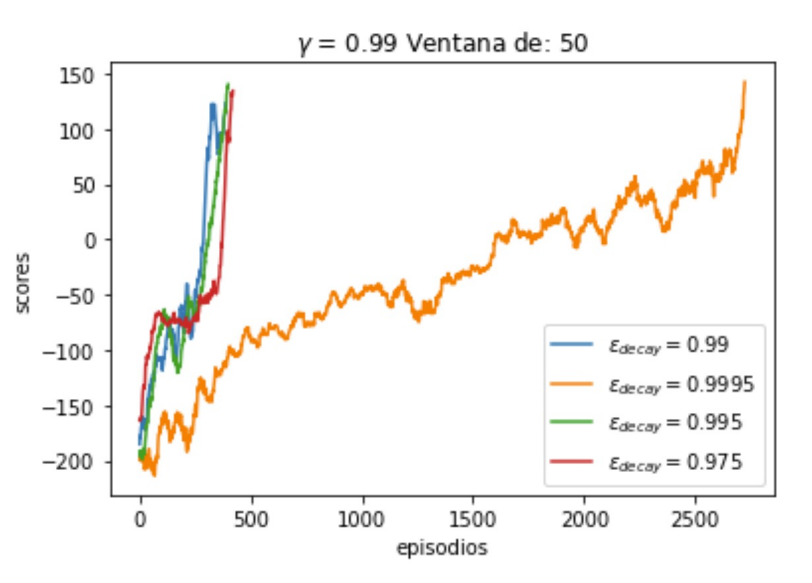
RESOLUCIÓN

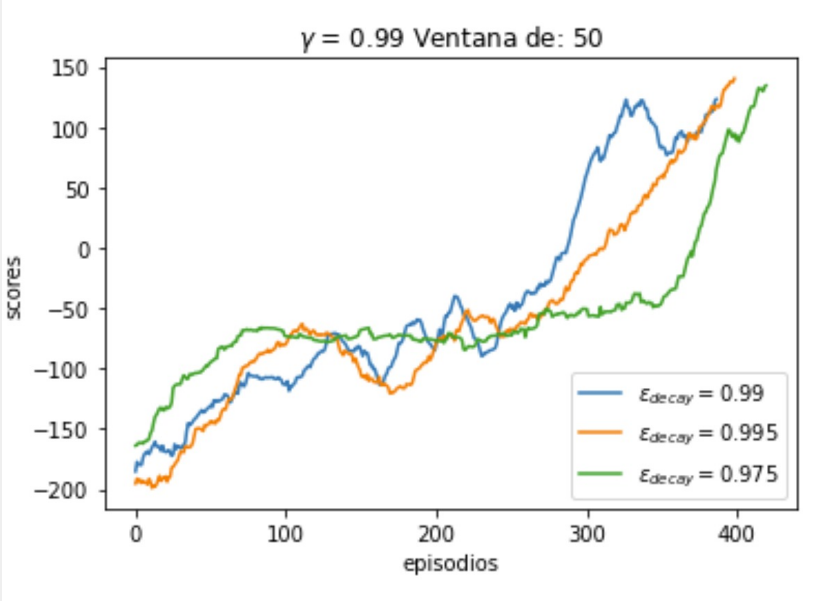
VEAMOS COMO NUESTRO AGENTE RESOLVER EL JUEGO!

Gráficos


$\varepsilon_{decay} = 0.975$
epsilon
scores
episodios
$\varepsilon_{decay} = 0.975$ Promedio ventana de 50
episodios
$\varepsilon_{decay} = 0.995$
epsilon
scores
episodios
$\varepsilon_{decay} = 0.995$ Promedio ventana de 50
episodios
$\varepsilon_{decay} = 0.99$
epsilon
scores
episodios
$\varepsilon_{decay} = 0.99$ Promedio ventana de 50
episodios
$\varepsilon_{decay} = 0.9995$
epsilon
scores
episodios
$\varepsilon_{decay} = 0.9995$ Promedio ventana de 50
episodios

Gráficos


γ = 0.99 Ventana de: 50
scores
episodios
150
100
50
0
−50
−100
−150
−200
0
500
1000
1500
2000
2500
ε_{decay} = 0.99
ε_{decay} = 0.9995
ε_{decay} = 0.995
ε_{decay} = 0.975


γ = 0.99 Ventana de: 50
scores
episodios
150
100
50
0
−50
−100
−150
−200
0
100
200
300
400
ε_{decay} = 0.99
ε_{decay} = 0.995
ε_{decay} = 0.975

Conclusiones

- Instalar las dependencias y paquetes para el juego toma su tiempo.
- Los detalles tienen mucha importancia.
- No está claro qué efecto tiene la arquitectura de la red en la resolución del problema. Se podría probar con más capas ocultas o más perceptrones por capas.
- Cantidad de episodios $\neq$ tiempo de ejecución.
- Mucha memoria es malo y poca memoria también.
- El tuning del ε_{decay} y γ tienen mucho impacto en los tiempos de entrenamiento.
- El agente logró generalizar a pesar de conocer sólo un ambiente.

Bibliografía

- Thanh Thi Nguyen, Ngoc Duy Nguyen, and Saeid Nahavandi. *Deep Reinforcement Learning for Multi-Agent Systems: A Review of Challenges, Solutions and Applications*. 2019.
- Volodymyr Mnih, Koray Kavukcuoglu, David Silver. *Human-level control through deep reinforcement learning*. 2015.
- Volodymyr Mnih, Koray Kavukcuoglu. *Playing Atari with Deep Reinforcement Learning*. 2013.
- Dimitri P. Bertsekas *Reinforcement Learning and Optimal Control*. MIT. 2019

FIN