UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

Sistemas Inteligentes Distribuidos

Agente de aprendizaje por refuerzo para el entorno Cliff Walking

Lluc Martínez Busquets
Eric Medina León
Àlex Rodríguez Rodríguez

2024/2025 Q2

Resumen

Este trabajo presenta un estudio detallado sobre la implementación de cuatro algoritmos de aprendizaje por refuerzo en el entorno *CliffWalking-v0* de la libreria de Python *Gymnasium*: *Value Iteration*, *Direct Estimation*, *Q-Learning* y *REINFORCE*. El entorno se configura con el modo *is_slippery* activado, lo cual introduce estocasticidad en las transiciones de estado.
El objetivo es evaluar el rendimiento de cada algoritmo, así como qué parámetros e hiperparámetros son los óptimos para su funcionamiento.

Índice

1. Introducción

El aprendizaje por refuerzo es un paradigma de aprendizaje automático en el que un agente aprende a tomar decisiones interactuando con un entorno y recibiendo recompensas o penalizaciones.
Este trabajo se centra en la implementación y evaluación experimental de cuatro algoritmos de aprendizaje por refuerzo en el entorno *CliffWalking-v0*, de la librería de Python *Gymnasium*, con el objetivo de comparar su rendimiento y eficiencia en dicho entorno. Los algoritmos implementados son: *Value Iteration*, *Direct Estimation*, *Q-Learning* y *REINFORCE*.

1.1. Caracterización del entorno

El entorno de *Cliff Walking*, propuesto originalmente por *Sutton & Barto*, es un entorno clásico para evaluar algoritmos de aprendizaje por refuerzo. En este entorno, el agente debe navegar por una cuadrícula de dimensiones 4x12 evitando caer en un acantilado, lo que representa una penalización significativa. El objetivo del agente es llegar a la esquina inferior derecha de la cuadrícula de la forma más eficiente posible, maximizando la recompensa acumulada a lo largo del tiempo y minimizando el número de pasos necesarios para alcanzar la meta.
A continuación se caracteriza de forma detallada el entorno:

- **Observabilidad**: Totalmente observable. El agente recibe en cada paso su posición exacta en la cuadrícula, sin ruido ni información oculta, por lo que tiene acceso total al estado relevante.

- **Número de agentes**: Un único agente.

- **Determinismo**: Estocástico, ya que está activado el modo `is_slippery=True`. En este caso, por cada acción que el agente toma, hay una probabilidad de aproximadamente el 66.7 % de que el agente se resvale hacia una dirección perpendicular a la acción deseada.

- **Atomicidad**: Secuencial. Las decisiones del agente tienen consecuencias que dependen de toda la historia de acciones y percepciones, y los efectos futuros de las acciones importan para maximizar la recompensa acumulada.

- **Dinamicidad**: Estático. El estado del entorno sólo cambia cuando el agente toma una acción; no hay cambios "por sí mismos" mientras el agente razona.

- **Continuidad**: Discreto. Tanto el espacio de estados (posiciones en la cuadrícula) como el de acciones (arriba, abajo, izquierda, derecha) y el tiempo de decisión son discretos.

- **Conocimiento**: Conocido. Las reglas de transición (aunque estocásticas) y la función de recompensa están definidas de antemano y son accesibles al agente.

1.1.1. Espacio de acciones

El agente dispone de un conjunto finito de acciones

$$\mathcal{A} = \{\text{Arriba, Derecha, Abajo, Izquierda}\},$$

cada una de las cuales intenta desplazar al agente una celda en la dirección indicada.

1.1.2. Estados inicial y terminal

- **Estado inicial** $s_0 = (3, 0)$, correspondiente a la esquina inferior izquierda de la cuadrícula.
- **Estado terminal** $s_T = (3, 11)$, la meta en la esquina inferior derecha; al llegar aquí, el episodio termina.

Si durante el episodio el agente cae en el acantilado, este regresa al estado inicial s_0 y continua con el episodio.

1.1.3. Función de recompensa

La señal de recompensa $R(s, a, s')$ se define como:

$$R(s,a,s') = \begin{cases} -100, & \text{si } s' \text{ es una celda de acantilado (cliff),} \\ -1, & \text{en cada transición válida que no alcance la meta ni el cliff,} \\ 0, & \text{al alcanzar el estado terminal } s_T. \end{cases}$$

De este modo, el agente está incentivado a llegar cuanto antes a la meta evitando caer en el precipicio.

1.2. Entorno experimental

Componente	**Descripción**
Sistema operativo	Ubuntu 24.04.1 LTS
Kernel Linux	6.8.0-59-generic
CPU	Intel Core i7-10750H (6 núcleos, 12 hilos, hasta 5.0 GHz)
GPU discreta	NVIDIA GeForce GTX 1650 Mobile (4 GB GDDR6)
GPU integrada	Intel CometLake-H GT2 (UHD Graphics)
Memoria RAM	16 GB DDR4-2933 MHz
Intérprete de Python	Python 3. 12. 9

Cuadro 1: Entorno de hardware y software utilizado en los experimentos

2. Value Iteration

2.1. Descripción del algoritmo

La iteración por valor es un método de programación dinámica para resolver un Proceso de Decisión de Markov (MDP) y encontrar simultáneamente la función valor óptima V^* y la política óptima π^*. Se basa en la relación de Bellman óptima:

$$V^*(s) \;=\; \max_{a \in A} \sum_{s'} P(s' \mid s, a) \big[R(s, a, s') + \gamma \, V^*(s')\big],$$

donde:

- S es el conjunto de estados.
- A es el conjunto de acciones.
- $P(s' \mid s, a)$ es la probabilidad de transición de s a s' dado a.
- $R(s, a, s')$ es la recompensa recibida al transitar.
- $\gamma \in [0, 1)$ es el factor de descuento.

A continuación se presenta el pseudocódigo genérico de Value Iteration que se ha implementado en este proyecto, seguido de las decisiones de diseño adoptadas en la implementación de Python.

Algorithm 1 Value Iteration

Require: Conjunto de estados S, conjunto de acciones A, $P(s' \mid s, a)$ y $R(s, a, s')$, factor de descuento $\gamma \in [0, 1)$, umbral de convergencia $\varepsilon > 0$
Ensure: Función valor V y política óptima π

Inicializar $V(s) \leftarrow 0$, $\forall s \in S$
repeat
  $\Delta \leftarrow 0$
  for all $s \in S$ **do**
    $V_{\text{old}} \leftarrow V(s)$
    $V(s) \leftarrow \max_{a \in A} \sum_{s'} P(s' \mid s, a)\big[R(s, a, s') + \gamma \, V(s')\big]$
    $\Delta \leftarrow \max\big(\Delta, \; |V(s) - V_{\text{old}}|\big)$
  end for
until $\Delta \le \varepsilon$
for all $s \in S$ **do**
  $\pi(s) \leftarrow \arg\max_{a \in A} \sum_{s'} P(s' \mid s, a)\big[R(s, a, s') + \gamma \, V(s')\big]$
end for
return V, π

Decisiones de diseño en la implementación Python

- **Cálculo de $Q(s, a)$ con manejo del estado terminal**: para calcular el valor de cada acción consideramos que si se ha llegado a un estado terminal, el término de arranque posterior (*bootstrap*) se anula:

$$Q(s,a) \;=\; \sum_{s'} p\left[r + \gamma\, V(s')\right] \quad \longrightarrow \quad \text{bootstrap} = 0 \text{ si } s' \text{ es terminal.}$$

  De esta forma, se garantiza que al terminar el episodio, no se incorporen erróneamente estimaciones de valor posteriores a la terminación.

- **Evaluación periódica de la política**: tras cada iteración de valor calculamos la recompensa media de la política actual en $N = 100$ episodios de longitud máxima $T_{\text{máx}} = 200$ (método `check_improvements`), tanto para monitorizar progresos como para registrar la mejor recompensa y la iteración en que ocurre. Se fijan los valores de N y $T_{\text{máx}}$ para evitar que el algoritmo se detenga por un número excesivo de episodios, lo que podría ocurrir si la política converge a una política subóptima. En este caso, el algoritmo se detendría sin haber explorado adecuadamente el espacio de estados.

2.2. Experimentación

En el caso de iteración de valor, se han decidido estudiar el efecto de diferente valores del factor de descuento γ y el parámetro de convergencia ϵ en el rendimiento del algoritmo.

2.2.1. Experimento factor de descuento & ϵ

2.2.1.1 Diseño experimental

El objetivo de este experimento es analizar cómo los parámetros γ y ϵ afectan el rendimiento del algoritmo de iteración de valor.

Observación	El rendimiento y óptimalidad de la política encontrada por *Value Iteration* se ven afectados por los valores de γ y ϵ.
Planteamiento	Para cada pareja de valores de γ y ϵ, se compara la tasa de acierto (llegar al estado final), la recompensa media, número de pasos y tiempo de entrenamiento del algoritmo.
Hipótesis	Se espera que un mayor valor de γ conduzca a una política más óptima, mientras que un menor valor de ϵ permita una convergencia más rápida con una menor precisión.
Método	▪ Se elige un conjunto de valores para γ y ϵ: $\gamma \in \{0.5, 0.7, 0.9, 0.95, 0.99\}$ y $\epsilon \in \{1 \times 10^{-1}, 1 \times 10^{-2}, 1 \times 10^{-4}, 1 \times 10^{-8}\}$. ▪ Para cada combinación de γ y ϵ, se ejecuta el algoritmo *Value Iteration* en el entorno. ▪ Se evalúa la política obtenida probándola con 500 episodios.

Cuadro 2: Value Iteration - Experimento 1 - Factor de descuento & ϵ

2.2.1.2 Resultados

3. Direct Estimation

3.1. Descripción del algoritmo

La versión de Estimación Directa que se ha implementado corresponde a un *Método Monte Carlo basado en modelo*, en el cual:

1. Se recolectan muestras de transición (s, a, s', r) jugando acciones aleatorias.
2. Se estiman empíricamente

$$\hat{T}(s,a,s') = \frac{\text{conteo}(s, a \to s')}{\sum_{s''} \text{conteo}(s, a \to s'')},$$
$$\hat{R}(s,a,s') = \frac{\sum r}{\text{veces}(s, a \to s')}.$$

3. Se aplica iteración de valor sobre el MDP estimado $(\hat{T}, \hat{R}, \gamma)$ para obtener

$$V^*(s) = \max_a \sum_{s'} \hat{T}(s,a,s') \Big[\hat{R}(s,a,s') + \gamma\, V^*(s')\Big],$$

   y de ahí la política óptima

$$\pi^*(s) = \arg\max_a \sum_{s'} \hat{T}(s,a,s') \Big[\hat{R}(s,a,s') + \gamma\, V^*(s')\Big].$$

A continuación se presenta el pseudocódigo genérico de Direct Estimation que se ha implementado en este proyecto, seguido de las decisiones de diseño adoptadas en la implementación de Python.

Algorithm 2 Estimación Directa (Model-based Monte Carlo)

Require: Factor de descuento γ, número de trayectorias N, tolerancia ε, máximo de iteraciones K

Inicializar contadores de transición y recompensa vacíos
Inicializar $V(s) \leftarrow 0$ para todo estado s
for $t = 1, \dots, K$ **do**
  Recolectar datos:
  for $i = 1, \dots, N$ **do**
    Jugar un paso aleatorio, obtener (s, a, s', r)
    Incrementar $N(s, a, s')$ y acumular recompensa en $R_{\text{sum}}(s, a, s')$
  end for
  Ajustar modelo:
  for all (s, a) **do**
    $\hat{T}(s, a, s') \leftarrow \dfrac{N(s, a, s')}{\sum_u N(s, a, u)}$
    $\hat{R}(s, a, s') \leftarrow \dfrac{R_{\text{sum}}(s, a, s')}{N(s, a, s')}$
  end for
  Iteración de valor:
  $\Delta \leftarrow 0$
  for all s **do**
    for all a **do**
      $Q(s, a) \leftarrow \sum_{s'} \hat{T}(s, a, s') \big[\hat{R}(s, a, s') + \gamma \, V(s')\big]$
    end for
    $V_{\text{nuevo}}(s) \leftarrow \text{máx}_a \, Q(s, a)$
    $\Delta \leftarrow \text{máx}\{\Delta, \, |V_{\text{nuevo}}(s) - V(s)|\}$
    $V(s) \leftarrow V_{\text{nuevo}}(s)$
  end for
  if $\Delta < \varepsilon$ **then break**
  end if
end for
return V, y derivar $\pi^*(s) = \arg \text{máx}_a \, Q(s, a)$

Decisiones de diseño en la implementación Python

- **Criterio de parada por paciencia.** Además de la tolerancia en la iteración de valor, detenemos el entrenamiento si no hay mejora en la recompensa media durante `PATIENCE` iteraciones, midiendo esto con la función `check_improvements()`.

3.2. Experimentación

3.2.1. Experimento factor de descuento & número de trayectorias

3.2.1.1 Diseño experimental

Observación	
Planteamiento	
Hipótesis	
Método	▪

Cuadro 3: Experimento 1

3.2.1.2 Resultados

3.2.2. Experimento número de episodios & *PATIENCE*

3.2.2.1 Diseño experimental

Observación	
Planteamiento	
Hipótesis	
Método	▪

Cuadro 4: Experimento 1

3.2.2.2 Resultados

4. Q-learning

4.1. Descripción del algoritmo

Q-learning es un algoritmo de aprendizaje por refuerzo. La característica fundamental de Q-learning es su capacidad para aprender de forma off-policy, es decir, puede aprender la política óptima mientras sigue una política de exploración diferente (como ε-greedy). El algoritmo actualiza iterativamente sus estimaciones $Q(s, a)$ utilizando la ecuación de Bellman. A medida que el aprendizaje progresa, las estimaciones de Q convergen hacia los valores óptimos, permitiendo derivar la política óptima como $\pi^*(s) = \arg\max_a Q(s, a)$.

A continuación se presenta el pseudocódigo genérico de Direct Estimation que se ha implementado en este proyecto, seguido de las decisiones de diseño adoptadas en la implementación de Python.

Algorithm 3 Q-Learning

Inicializar $Q(s, a) \leftarrow 0$ para todo $s \in S$, $a \in A$
for episodio $\leftarrow 1$ **to** $N_{\text{episodios}}$ **do**
    $\varepsilon \leftarrow \max(\varepsilon_{\text{mín}}, \varepsilon_0 \cdot \text{decay}^{\text{episodio}})$
    Inicializar $s \leftarrow s_0$
    for $t \leftarrow 1$ **to** $T_{\text{máx}}$ **do**
        if rand() $\leq \varepsilon$ **then**
            $a \leftarrow$ acción aleatoria
        else
            $a \leftarrow \arg\max_{a'} Q(s, a')$
        end if
        Ejecutar a, observar r, s'
        td_target $\leftarrow r + \gamma \max_{a''} Q(s', a'')$
        td_error $\leftarrow$ td_target $- Q(s, a)$
        $Q(s, a) \leftarrow Q(s, a) + \alpha$ td_error
        if s' es terminal **then**
            break
        end if
        $s \leftarrow s'$
    end for
end for

Decisiones de diseño en la implementación Python

- **Política ε-greedy con decaimiento:**

$$\pi(a \mid s) = \begin{cases} \frac{1}{|\mathcal{A}|}, & \text{con probabilidad } \varepsilon, \\ 1, & \text{si } a = \arg\max_{a'} Q(s, a'), \\ 0, & \text{en otro caso.} \end{cases}$$

Al inicio de cada episodio:

$$\varepsilon \leftarrow \text{máx}\big(\varepsilon_{\text{mín}},\ \varepsilon \cdot (\text{decay})^{\text{episodio}}\big).$$

- **Wrapper de recompensas customizado:** Se penaliza la acción `Izquierda` añadiendo una recompensa peor que la original, ya que en ningún caso interesa que el agente se desplace hacia la izquierda.

4.2. Experimentación

4.2.1. Experimento factor de descuento & ϵ decay

4.2.1.1 Diseño experimental

Observación	El factor de descuento (γ) y la tasa de decaimiento de ϵ son parámetros críticos en el algoritmo Q-Learning.
Planteamiento	Para cada pareja de valores de γ y *decay*, se compara la tasa de acierto (llegar al estado final), la recompensa media, número de pasos y tiempo de entrenamiento del algoritmo.
Hipótesis	Un mayor factor de descuento y una tasa de decaimiento de ϵ más lenta mejorarán el rendimiento del algoritmo.
Método	- Se fijan 1000 episodios de entrenamiento, tasa de aprendizaje $\alpha = 0.1$ y ϵ inicial de 0.9. - Se eligen los siguientes valores para γ y *decay*: $\gamma \in \{0.5, 0.7, 0.9, 0.95, 0.99\}$ y *decay* $\in \{0.1, 0.2, 0.5, 0.8\}$. - Para cada combinación de γ y *decay*, se ejecuta el algoritmo Q-Learning en el entorno. - Se evalúa la política obtenida probándola con 500 episodios. - Se repite el proceso para cada combinación de γ y *decay* 20 veces.

Cuadro 5: Q-Learning - Experimento 1 - Factor de descuento & ϵ decay

4.2.1.2 Resultados

4.2.2. Experimento tasa de aprendizaje & ϵ

4.2.2.1 Diseño experimental

Observación	
Planteamiento	
Hipótesis	
Método	▪

Cuadro 6: Experimento 1

4.2.2.2 Resultados

4.2.3. Experimento número de episodios

4.2.3.1 Diseño experimental

Observación	
Planteamiento	
Hipótesis	
Método	▪

Cuadro 7: Experimento 1

4.2.3.2 Resultados

4.2.4. Experimento penalización de la acción izquierda

4.2.4.1 Diseño experimental

Observación	
Planteamiento	
Hipótesis	
Método	▪

Cuadro 8: Experimento 1

4.2.4.2 Resultados

5. Reinforce

5.1. Descripción del algoritmo

5.2. Experimentación

Observación	
Planteamiento	
Hipótesis	
Método	▪

Cuadro 9: Experimento 1

6. Conclusiones

7. Bibliografia

Referencias

[1] Farama Foundation. Cliff walking environment. https://gymnasium.farama.org/environments/toy_text/cliff_walking/, 2025. Accedido: 15 de abril de 2025.

[2] Farama Foundation. Gymnasium: A reinforcement learning library. https://github.com/Farama-Foundation/Gymnasium, 2025. Accedido: 15 de abril de 2025.

8. Apéndices