

# Aprendizaje por Refuerzo

Informe proyecto final.

Ingeniería Matemática 4ºB



# Aprendizaje por Refuerzo

Informe proyecto final.

Ingeniería Matemática 4ºB

realizado por

---

Jorge Kindelán Navarro

---

Ignacio Queipo de Llano Pérez Gascón

---

Beltrán Sánchez Careaga

Profesor: Alvaro López López  
Duración de Proyecto: 11, 2025 - 12, 2025  
Facultad: Universidad Pontificia de Comillas



**COMILLAS**  
UNIVERSIDAD PONTIFICIA

**ICAI**

# Contents

<b>1</b>	<b>Introducción</b>	<b>1</b>
1.1	Primera Fase: SARSA y Tile Coding . . . . .	2
1.2	Transición a Deep Reinforcement Learning . . . . .	2
1.3	Entornos de Entrenamiento . . . . .	2
1.4	Resultados y Comportamientos Aprendidos . . . . .	3
<b>2</b>	<b>Elección del Algoritmo y Arquitectura del Agente</b>	<b>4</b>
2.1	Análisis del problema . . . . .	4
2.2	Por qué descartamos SARSA + Tile Coding . . . . .	4
2.3	Por qué no se utilizó Q-Learning tabular . . . . .	5
2.4	Elección del modelo: Deep Q-Network (DQN) . . . . .	5
2.5	Motivación para usar Double DQN . . . . .	7
<b>3</b>	<b>Representación del Estado e Ingeniería de Características</b>	<b>9</b>
3.1	Introducción . . . . .	9
3.2	Limitaciones de la observación raw . . . . .	9
3.2.1	Aprendizaje extremadamente lento . . . . .	10
3.2.2	Tasa de colisión muy alta . . . . .	10
3.3	Ingeniería de características: de 11 a 23 features . . . . .	10
3.4	Efectos de la ingeniería de características . . . . .	11
3.5	Resumen de características: . . . . .	11
<b>4</b>	<b>Diseño del Sistema de Recompensas</b>	<b>13</b>
4.1	Primera aproximación: Reward shaping complejo . . . . .	13
4.2	Problema detectado: Reward shaping excesivo . . . . .	13
4.3	Decisión: Recompensas simples y sparse . . . . .	14
4.4	Resultados: Generalización significativamente mejor . . . . .	15
<b>5</b>	<b>Arquitectura de la Red Neuronal y Mecanismos de Aprendizaje</b>	<b>16</b>
5.1	Arquitectura de la red neuronal . . . . .	16
5.1.1	Necesidades del problema . . . . .	16
5.1.2	Arquitecturas evaluadas . . . . .	16
5.1.3	Arquitectura final . . . . .	16
5.1.4	Justificación de las decisiones arquitectónicas . . . . .	17
5.2	Experience Replay: La memoria del agente . . . . .	17
5.2.1	¿Cómo funciona? . . . . .	18
5.2.2	Ventajas del Experience Replay . . . . .	18
5.3	Priorización de experiencias . . . . .	18
5.3.1	Beneficios observados . . . . .	18
5.4	Pseudo-código de la implementación: . . . . .	18
<b>6</b>	<b>Entrenamiento de los Tres Entornos</b>	<b>20</b>
6.1	Entorno 1 — La base del comportamiento . . . . .	20
6.2	Entorno 2 — El plateau . . . . .	21
6.3	Entorno 3 — Transfer Learning al rescate . . . . .	22
6.4	Checkpointing y Early Stopping . . . . .	23
6.5	Conclusión . . . . .	24
<b>7</b>	<b>Resultados Finales y Reflexiones del Proyecto</b>	<b>25</b>
7.1	Resultados Finales . . . . .	25
7.2	Reflexiones Finales del Proyecto . . . . .	25

7.2.1	Qué funcionó especialmente bien . . . . .	25
7.2.2	Lo que aprendimos . . . . .	26
7.3	Por qué no pudimos usar Tile Coding (y SARSA) para el problema del almacén . . . . .	26
7.3.1	Qué cambió en el proyecto del almacén . . . . .	27
7.3.2	Por qué una red neuronal sí funciona . . . . .	27

# 1

## Introducción

El proyecto *DiSpAtCh* (Distributed Spatial Agent Training for Challenges in Handling) aborda el diseño, entrenamiento y evaluación de un agente autónomo capaz de desenvolverse en un entorno de almacén. El objetivo general es que el agente aprenda, mediante Aprendizaje por Refuerzo (RL), a navegar por un espacio continuo lleno de obstáculos, recoger objetos y entregarlos en una zona designada, optimizando su comportamiento a través de la experiencia.

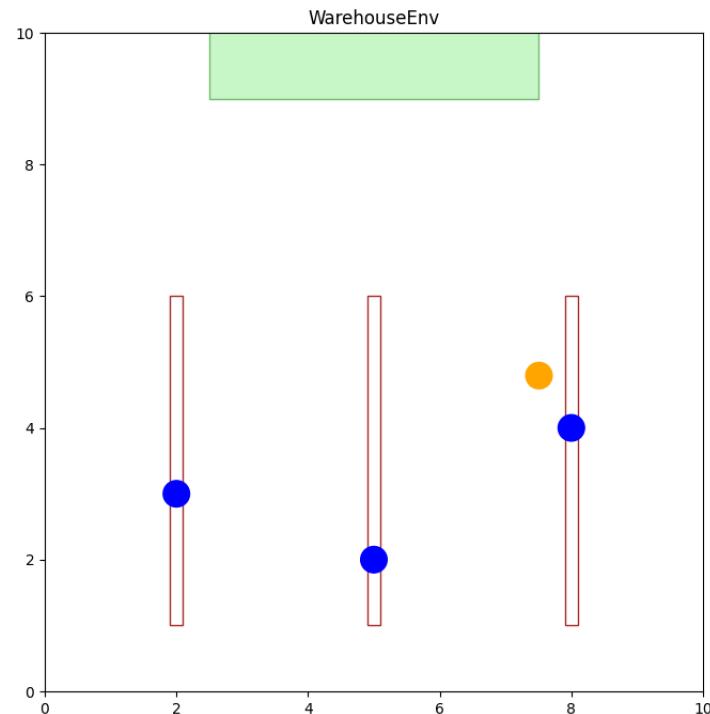


Figure 1.1: Entorno del proyecto.

El desarrollo del proyecto se llevó a cabo de forma incremental, comenzando por un entorno sencillo y avanzando progresivamente hacia tareas de mayor complejidad.

## 1.1. Primera Fase: SARSA y Tile Coding

La primera fase consistió en implementar un agente basado en SARSA y Tile Coding, lo que permitió explorar los fundamentos del RL con función de aproximación sobre un espacio continuo. En esta etapa, el estado del agente estaba compuesto únicamente por su posición en el mapa, permitiendo que el *tile coding* discretizara eficazmente el entorno y facilitara la generalización entre posiciones cercanas.

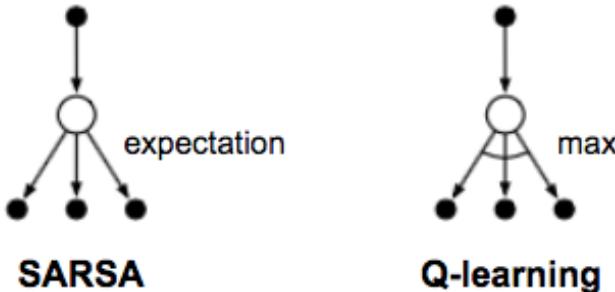


Figure 1.2: Sarsa y Q-learning (concepto equivalente a DQN).

Sin embargo, al trasladar este enfoque al problema completo del almacén —que incluye múltiples objetos, estados internos (como tener o no un objeto), detección de colisiones y recompensas escasas— se evidenciaron las limitaciones del método. El espacio de estados pasó de 2 dimensiones a 11 dimensiones, haciendo que el *tile coding* escalara exponencialmente y volviera inviable el uso de SARSA o Q-Learning tabular.

## 1.2. Transición a Deep Reinforcement Learning

Estas limitaciones motivaron la transición a métodos de Deep Reinforcement Learning, concretamente Deep Q-Networks (DQN). El uso de DQN permitió aproximar la función  $Q$  mediante una red neuronal capaz de procesar una representación rica y continua del entorno.

Para evitar inestabilidades típicas del aprendizaje profundo en RL se incorporaron varias técnicas consolidadas:

- Double DQN
- Target Networks
- Experience Replay
- Prioritized Sampling

Además, se diseñó cuidadosamente una *feature engineering* que transforma la observación original del entorno (11 variables) en un vector de 23 características informativas, incluyendo distancias, direcciones y proximidad a obstáculos. Este componente resultó esencial para reducir colisiones y mejorar la capacidad de generalización del agente.

## 1.3. Entornos de Entrenamiento

El proyecto se divide en tres entornos progresivos:

Environment	Objects	Objective	Actions	Difficulty
Entorno 1	Fixed positions	Pick only	5 ( $\uparrow \downarrow \leftarrow \rightarrow$ pick)	□ □
Entorno 2	Fixed positions	Pick + Delivery	6 ( $\uparrow \downarrow \leftarrow \rightarrow$ pick drop)	□ □ □
Entorno 3	Random positions	Pick + Delivery	6 ( $\uparrow \downarrow \leftarrow \rightarrow$ pick drop)	□ □ □ □

Table 1.1: Resumen de los tres entornos empleados en el proyecto.

## 1.4. Resultados y Comportamientos Aprendidos

A lo largo de los experimentos, el agente adquirió comportamientos cada vez más complejos, tales como:

- Navegar evitando obstáculos.
- Aproximarse a objetos desde cualquier ángulo.
- Optimizar trayectorias.
- Completar secuencias largas de acciones: recogida → transporte → entrega.

El análisis de resultados confirma:

- La eficacia del uso de DQN.
- El impacto crítico del diseño de *features*.
- La importancia del *transfer learning* para acelerar la convergencia en entornos complejos.

# 2

## Elección del Algoritmo y Arquitectura del Agente

### 2.1. Análisis del problema

Tras definir el entorno del almacén y las tareas a resolver —navegar, recoger objetos y entregarlos evitando obstáculos—, quedó claro que el agente debía operar en un espacio continuo con múltiples componentes relevantes del estado:

- Posición del agente ( $x, y$ )
- Posiciones de hasta tres objetos
- Bandera de colisión
- Bandera de entrega
- Indicador de si el agente lleva un objeto

Esto genera un espacio de estados de 11 dimensiones, altamente continuo y con fuertes dependencias espaciales y temporales. Además, el entorno presenta recompensas escasas (*sparse rewards*), ya que la mayor parte del retorno se obtiene solo cuando se completa correctamente la secuencia de recogida y entrega.

Por tanto, la elección del algoritmo debía cumplir los siguientes requisitos:

- Ser capaz de manejar espacios de estado continuos y de alta dimensionalidad.
- Escalar correctamente cuando la complejidad del entorno aumenta.
- Ser robusto frente a señales de recompensa esporádicas.
- Incorporar mecanismos de estabilidad en el entrenamiento.

Con estas condiciones, analizamos varias alternativas.

### 2.2. Por qué descartamos SARSA + Tile Coding

La primera aproximación del proyecto utilizó SARSA con *tile coding*, una técnica eficaz en entornos de baja dimensionalidad. *Tile coding* discretiza el espacio continuo en “tiles” superpuestos que permiten generalizar entre estados similares.

Esto funcionó bien en la práctica inicial porque el estado se definía únicamente como:

$$\text{Estado} = (x_{\text{agente}}, y_{\text{agente}})$$

Sólo 2 dimensiones → discretización viable.

Sin embargo, al trasladar esta técnica al entorno del almacén completo, el estado pasó a ser:

$$\text{Estado} = (x_{\text{agente}}, y_{\text{agente}}, x_{\text{obj}1}, y_{\text{obj}1}, x_{\text{obj}2}, y_{\text{obj}2}, x_{\text{obj}3}, y_{\text{obj}3}, \text{tiene\_objeto}, \text{colision}, \text{entrega})$$

Es decir, 11 dimensiones, muchas de ellas continuas.

Con 10 tiles por dimensión, el número total de combinaciones sería:

$$10^{11} = 100\,000\,000\,000 \text{ tiles}$$

imposible de almacenar, actualizar o explorar siquiera parcialmente.

Además:

- la discretización produciría una gran pérdida de información,
- se perdería la relación geométrica entre objetos,
- el agente necesitaría millones de episodios solo para cubrir una fracción útil del espacio.

Por tanto, SARSA + tile coding no es escalable para este problema.

### 2.3. Por qué no se utilizó Q-Learning tabular

Q-Learning tabular requiere almacenar un valor por cada combinación posible:

$$Q(s, a)$$

Pero en un espacio continuo como el del almacén, las posiciones del agente y los objetos pueden tomar infinitos valores. Discretizar a mano sería arbitrario y conduciría a:

- explosión combinatoria,
- pérdida severa de precisión,
- incapacidad de generalización a posiciones nunca vistas.

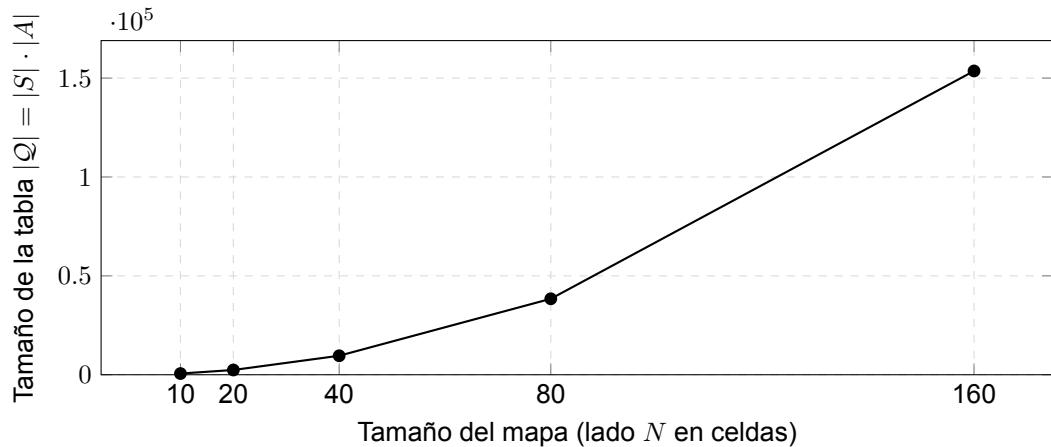


Figure 2.1: Crecimiento del tamaño de la tabla Q con el tamaño del mapa (asumiendo 6 acciones por estado).

En consecuencia, la tabla Q sería gigantesca, dispersa e impracticable.

### 2.4. Elección del modelo: Deep Q-Network (DQN)

El siguiente paso fue recurrir a métodos de Deep Reinforcement Learning, capaces de aproximar funciones  $Q$  sobre espacios continuos. DQN fue elegido por varias razones clave.

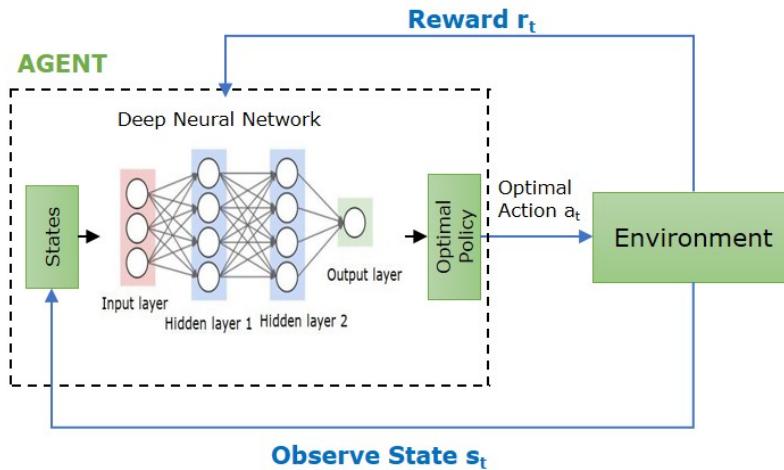


Figure 2.2: Arquitectura de una DQN.

## El espacio de acciones es discreto

El agente solo puede ejecutar 5–6 acciones:

- mover arriba
- mover abajo
- mover izquierda
- mover derecha
- pick
- drop

Esto es ideal para un algoritmo basado en Q-learning.

## Una red neuronal puede aproximar $Q(s, a)$

En lugar de una tabla, el agente aprende una función continua:

$$Q(s, a) \approx f_{\theta}(s, a)$$

Esta representación permite capturar:

- distancias,
- direcciones,
- proximidad a obstáculos,
- relaciones espaciales complejas.

## Experience Replay

Permite:

- romper la correlación temporal,
- estabilizar el aprendizaje,
- reutilizar experiencias valiosas (especialmente las de entrega, muy escasas).

Esto fue crítico para enfrentarse al problema de *sparse rewards*.

## Técnica ampliamente estudiada y mejorada

DQN cuenta con extensiones (Double DQN, Dueling, PER, etc.) que mejoran estabilidad y precisión, lo que permite construir una solución robusta.

Por estas razones, DQN se posicionó como el modelo adecuado para afrontar el problema del almacén.

### 2.5. Motivación para usar Double DQN

Durante las primeras pruebas con DQN *vanilla*, se observó:

- un crecimiento excesivo de los valores Q,
- oscilaciones en las predicciones,
- inestabilidad en la política.

Este fenómeno es conocido: sobreestimación sistemática. Se debe a que la red usa su propia estimación tanto para:

1. seleccionar la mejor acción futura,
2. evaluar su valor.

El cálculo del target en DQN estándar es:

$$\text{target} = r + \gamma \max_a Q(s', a)$$

lo que introduce un sesgo positivo.

## Solución: separar selección y evaluación

En Double DQN:

- La **policy network** selecciona la acción futura

$$a^* = \arg \max_a Q_{\text{policy}}(s', a)$$

- La **target network** evalúa esa acción

$$Q_{\text{target}}(s', a^*)$$

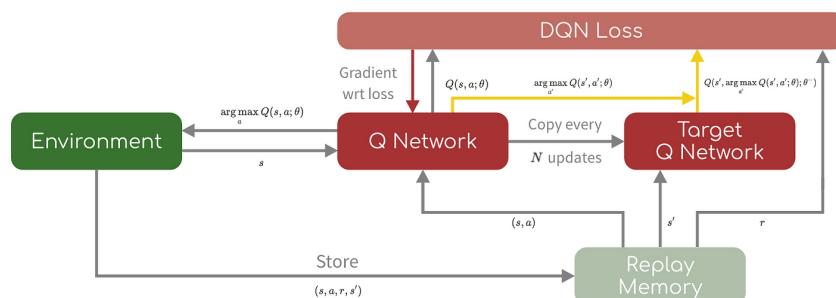


Figure 2.3: Double-DQN para el agente.

## Ventajas

- evita sobreestimación,
- produce actualizaciones más realistas,
- reduce oscilaciones,
- acelera la convergencia,

- especialmente útil en entornos con recompensas escasas (como el Entorno 2).
- Double DQN se integró por estas razones y demostró ser esencial para estabilizar el aprendizaje.

# 3

## Representación del Estado e Ingeniería de Características

### 3.1. Introducción

La representación del estado es uno de los elementos más críticos en cualquier sistema de Aprendizaje por Refuerzo. En este proyecto, aunque el entorno proporciona una observación *raw* de 11 dimensiones, descubrimos rápidamente que trabajar directamente con estos valores impedía que el agente aprendiera comportamientos complejos de navegación y manipulación. La red neuronal tenía dificultades para interpretar correctamente la geometría del entorno, lo que derivaba en tiempos de aprendizaje muy largos y en una tasa de colisión excesivamente alta.

Por ello fue necesaria la construcción de una representación del estado enriquecida, que extrajera explícitamente características relevantes para la tarea. Esta etapa resultó determinante para obtener un comportamiento estable y eficiente.

### 3.2. Limitaciones de la observación raw

El entorno proporciona una observación compuesta por:

- Posición del agente → (2 valores)
- Posiciones de los 3 objetos → (6 valores)
- Flag: ¿el agente tiene un objeto? → (1)
- Flag: colisión → (1)
- Flag: entrega → (1)

Esto suma un total de 11 dimensiones.

A primera vista, parece suficiente. Sin embargo, estos valores son coordenadas absolutas, y la red neuronal debía inferir por sí misma conceptos como:

- cercanía a un objeto,
- dirección hacia el objetivo,
- orientación relativa,
- si un gesto (pick/drop) es válido o no,
- proximidad a estanterías u obstáculos,
- y la estructura espacial del almacén.

Esta carga cognitiva implícita llevaba a dos problemas principales:

### 3.2.1. Aprendizaje extremadamente lento

La red debía derivar relaciones espaciales complejas a partir de números brutos.

### 3.2.2. Tasa de colisión muy alta

El agente “no veía” los obstáculos; solo veía sus coordenadas sin contexto relevante.

Era evidente que necesitábamos extraer manualmente características que reforzaran la comprensión espacial del agente.

## 3.3. Ingeniería de características: de 11 a 23 features

Para resolver estas limitaciones, desarrollamos la clase `WarehouseFeedback`, cuya función es transformar la observación original en un vector más informativo de 23 características. Cada una está diseñada para aportar señales útiles al agente.

A continuación se describen las features incluidas:

### Posición normalizada del agente (2 features)

Se divide la posición  $(x, y)$  entre 10, de modo que los inputs estén en:

$$[0, 1]$$

Esto facilita el entrenamiento y evita escalas dispares.

### Flag: ¿tiene objeto? (1 feature)

Indica si el agente está cargando un objeto. Clave para decidir entre PICK y DROP.

### Distancias a los objetos (3 features)

Se calculan las distancias euclidianas a cada uno de los objetos y se normalizan por la diagonal máxima del entorno (aprox. 14.14). Esto permite comparar distancias de forma uniforme.

### Distancia al objeto más cercano (1 feature)

$$\min(d_1, d_2, d_3)$$

Esto ayuda al agente a identificar cuándo está lo suficientemente cerca para recoger un objeto.

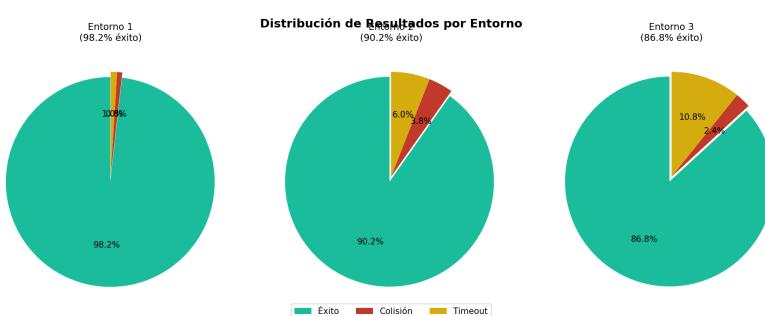


Figure 3.1: Gráfica de porcentaje colisiones en episodios.

### Dirección al objeto más cercano (2 features)

Un vector unitario:

$$\frac{\text{obj} - \text{agente}}{\|\text{obj} - \text{agente}\|}$$

Esto indica explícitamente hacia dónde debe moverse el agente. Reduce drásticamente la carga de aprendizaje.

## Distancia y dirección a la zona de entrega (3 features)

Solo relevantes cuando lleva un objeto. Permiten al agente navegar intuitivamente hacia la zona objetivo.

## Posición relativa de los objetos (6 features)

Valores:

$$(x_{obj} - x_{agente}), \quad (y_{obj} - y_{agente})$$

Normalizados. Mucho más informativos que las posiciones absolutas.

## Proximidad a obstáculos (4 features)

Este fue uno de los descubrimientos clave del proyecto.

Para cada dirección cardinal (arriba, abajo, izquierda, derecha):

1. Se simula un movimiento en esa dirección.
2. Se calcula la distancia al obstáculo más cercano.
3. Se normaliza entre [0, 1].
4. Se invierte para que 1 = muy cerca, 0 = lejos.

Esto genera cuatro features que indican el riesgo de colisión en cada dirección.

**Impacto de esta feature:**

- La tasa de colisión cayó del 70% a menos del 5%.
- El agente por fin aprendió a evitar estanterías.

## Flag de ``puede recoger'' (1 feature)

Vale 1 si el agente está a distancia suficiente del objeto más cercano. Agiliza la toma de decisiones sobre el PICK.

### 3.4. Efectos de la ingeniería de características

La nueva representación mejorada trajo consigo beneficios inmediatos:

- Menos colisiones gracias a las features de proximidad.
- Mejor navegación gracias a las direcciones relativas.
- Menor carga para la red neuronal.
- Rápida convergencia del entrenamiento.
- Generalización real en el Entorno 3.

Las features permiten que, incluso con posiciones aleatorias, el agente razonne en términos de relaciones y no de posiciones absolutas.

### 3.5. Resumen de características:

<b>Características</b>	<b>Cantidad</b>	<b>Descripción</b>
Posición del agente	2	Normalizada (x, y)
Bandera de objeto	1	Binaria
Distancia a objetos	3	Distancias normalizadas
Distancia al objeto más cercano	1	Distancia mínima
Dirección al más cercano	2	Vector unitario
Distancia a entrega	1	Normalizada
Dirección a entrega	2	Vector unitario
Posiciones relativas	6	Objetos relativos al agente
Proximidad a obstáculos	4	Distancia a obstáculos en 4 direcciones
Bandera de recogida	1	Binaria (suficientemente cerca para recoger)
<b>Total</b>	<b>23</b>	

**Table 3.1:** Ingeniería de Características (23 características)

# 4

## Diseño del Sistema de Recompensas

El sistema de recompensas es un componente fundamental en cualquier algoritmo de Aprendizaje por Refuerzo, ya que define qué comportamientos se consideran deseables y guía el proceso de aprendizaje del agente. En este proyecto, el diseño de las recompensas fue uno de los aspectos en los que más iteramos, especialmente al escalar la complejidad del entorno. Lo que inicialmente funcionaba bien en tareas simples se volvió contraproducente en escenarios más variados, obligándonos a replantear el enfoque.

### 4.1. Primera aproximación: Reward shaping complejo

Durante las primeras versiones del proyecto, optamos por un sistema de recompensas detallado y denso (*reward shaping*) que incorporaba una gran cantidad de señales intermedias para orientar al agente paso a paso:

- Recompensas positivas por acercarse al objeto.
- Penalizaciones por alejarse.
- Bonificaciones por trayectorias eficientes.
- Penalizaciones suaves por moverse sin un propósito claro.
- Múltiples términos ponderados cuidadosamente.

Este enfoque pretendía acelerar el aprendizaje al ofrecer *feedback* continuo y detallado. En los primeros entornos, especialmente en el Entorno 1 (solo recogida), este sistema funcionó razonablemente bien: el agente aprendía a aproximarse gradualmente al objeto, reduciendo la exploración innecesaria.

En el Entorno 2 (recoger y entregar) el aprendizaje fue más lento, pero aún alcanzaba la solución. Sin embargo, los problemas surgieron al pasar al Entorno 3, donde los objetos aparecían en posiciones aleatorias. Aquí el agente no generalizaba: aprendía solo a repetir trayectorias concretas que habían dado resultados positivos en el entrenamiento.

### 4.2. Problema detectado: Reward shaping excesivo

La raíz del problema estaba en la propia sofisticación del sistema de recompensas. Este fenómeno es bien conocido en RL:

**Cuando el sistema de recompensas es demasiado detallado, el agente aprende a maximizar las señales intermedias en lugar de resolver realmente la tarea.**

En nuestro caso:

- El agente memorizaba patrones específicos de movimiento asociados a ganancia incremental,
- pero no comprendía la estructura general del problema (navegar → recoger → entregar),

- lo que lo hacía extremadamente dependiente de las posiciones de entrenamiento.

Esto generaba sobreajuste conductual: la política funcionaba bien en regiones conocidas del espacio, pero fallaba en casos nuevos.

En otras palabras:

*El agente optimizaba el shaping, no el objetivo final.*

### 4.3. Decisión: Recompensas simples y sparse

Tomamos entonces la decisión de reemplazar el sistema complejo por una recompensa mucho más simple, *sparse* y clara, centrada únicamente en los eventos relevantes de la tarea:

Evento	Recompensa
Cada paso	-1
Colisión	-100
Recoger objeto (PICK)	+100
Entregar objeto (DROP)	+200
Soltar fuera de zona	-50

### Racionalidad de estos valores

#### -1 por paso

- Incentiva eficiencia sin imponer un camino concreto.
- Evita comportamientos deambulantes.

#### -100 por colisión

- Señal fuerte contra comportamientos peligrosos.
- Funciona bien porque la proximidad a obstáculos está incluida como feature.

#### +100 por recoger

- Marca claramente un objetivo intermedio necesario.

#### +200 por entregar

- La recompensa máxima.
- Define inequívocamente el objetivo final de la tarea.

#### -50 por drop incorrecto

- Evita que el agente pruebe DROP de forma aleatoria.

Este esquema elimina completamente el *shaping* intermedio. El agente solo recibe señales cuando realmente importa.

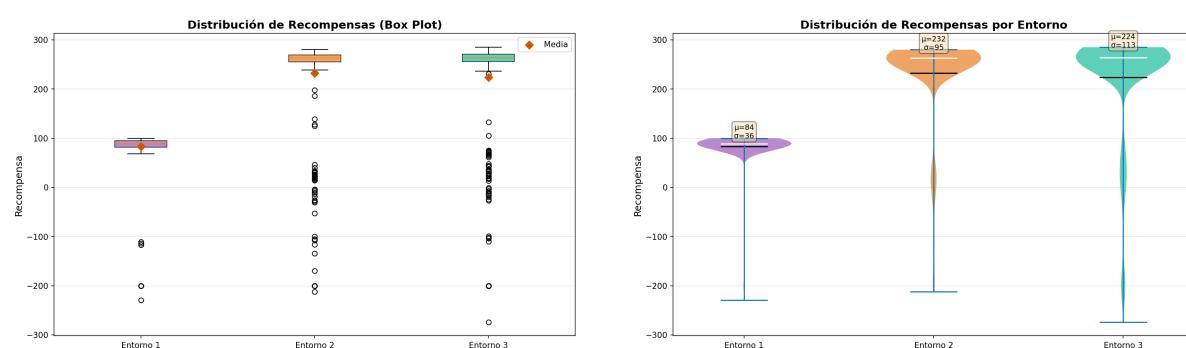


Figure 4.1: Distribución de las recompensas para el agente.

#### **4.4. Resultados: Generalización significativamente mejor**

Al adoptar recompensas *sparse*, observamos mejoras notables:

- El agente aprendió comportamientos más robustos.
- Reducción del sobreajuste.
- Mejor respuesta ante posiciones aleatorias (Entorno 3).
- Menor dependencia del orden de exploración inicial.
- Mejor colaboración con *Experience Replay*.
- Aprendizaje más estable y con menos ruido.

Este hallazgo refuerza una lección fundamental en Aprendizaje por Refuerzo:

**Las recompensas deben definir el objetivo, no el camino para llegar a él.**

# 5

## Arquitectura de la Red Neuronal y Mecanismos de Aprendizaje

Para resolver el problema del almacén mediante Aprendizaje por Refuerzo profundo, fue necesario diseñar una arquitectura de red neuronal capaz de aproximar la función de valor  $Q(s, a)$  de manera eficiente, estable y generalizable. El diseño de esta red es crucial, ya que debe capturar las relaciones entre 23 *features* de entrada y un conjunto de 5 o 6 acciones posibles, evitando tanto el *underfitting* como el *overfitting*.

Además, para estabilizar el entrenamiento y mejorar la muestra efectiva de experiencia, integramos mecanismos fundamentales como *Experience Replay*, priorización de experiencias relevantes y el uso de una red *target* para romper correlaciones temporales.

### 5.1. Arquitectura de la red neuronal

#### 5.1.1. Necesidades del problema

El estado del agente está formado por 23 características cuidadosamente diseñadas (distancias, direcciones, proximidad a obstáculos, posiciones relativas...), lo que exige una red con suficiente capacidad para aprender relaciones no lineales entre ellas.

Al mismo tiempo, el número de acciones es pequeño (5–6), por lo que la red no debe ser excesivamente grande para evitar el sobreajuste y mantener la eficiencia computacional.

#### 5.1.2. Arquitecturas evaluadas

Durante el desarrollo probamos varias arquitecturas:

Arquitectura	Resultado	Problema
[64, 32]	No convergía bien	Capacidad insuficiente
[256, 128, 64]	Convergencia, pero pobre generalización	Overfitting, lento
[128, 64]	Convergencia estable y buena generalización	Arquitectura final

La arquitectura seleccionada ofrecía el equilibrio ideal entre capacidad representativa y simplicidad.

#### 5.1.3. Arquitectura final

La red final utilizada para aproximar  $Q(s, a)$  fue:

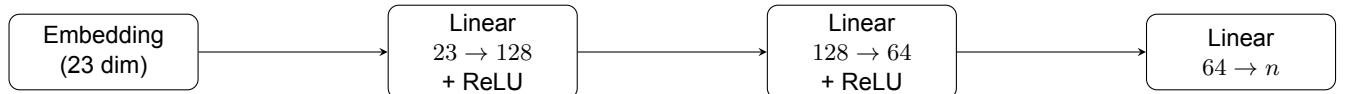


Figure 5.1: Arquitectura con embedding de 23 dimensiones y capas lineales con ReLU.

## 5.1.4. Justificación de las decisiones arquitectónicas

### ¿Por qué ReLU?

- Computacionalmente eficiente.
- Favorece la propagación del gradiente (evita *vanishing gradient*).
- Funciona de forma excelente en tareas de control continuo y RL.

### ¿Por qué solo dos capas ocultas?

- Las arquitecturas más profundas no mejoraron resultados.
- Más capas aumentaban la complejidad y el tiempo de entrenamiento.
- El problema no requería mayor profundidad.

Con dos capas ocultas obtenemos la capacidad justa para aprender relaciones entre las variables sin caer en sobreajuste.

## 5.2. Experience Replay: La memoria del agente

El entrenamiento de redes neuronales en entornos de RL es especialmente difícil porque las experiencias llegan secuencialmente y están altamente correlacionadas. Esto causa dos problemas graves:

- **Catastrophic forgetting:** la red olvida experiencias anteriores si solo se entrena con las más recientes.
- **Gradientes inestables:** las secuencias correlacionadas producen oscilaciones fuertes en la función  $Q$ .

Para resolver esto, empleamos *Experience Replay*.

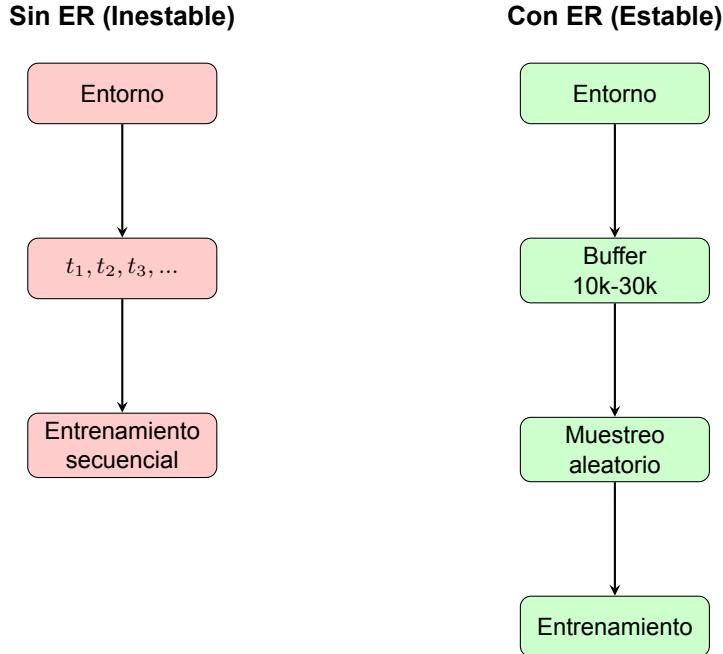


Figure 5.2: Comparación: entrenamiento secuencial vs. Experience Replay.

## 5.2.1. ¿Cómo funciona?

Cada vez que el agente interactúa con el entorno, se guarda una tupla de transición:

(state, action, reward, next\_state, done)

Estas transiciones se almacenan en un buffer de tamaño entre 10 000 y 30 000.

Durante el entrenamiento no usamos la transición más reciente, sino que muestreamos aleatoriamente un *batch*:

```
batch = random.sample(buffer, batch_size=64)
```

## 5.2.2. Ventajas del Experience Replay

**Rompe la correlación temporal** Mezcla experiencias antiguas con nuevas, evitando que la red esté sesgada por las últimas transiciones.

**Reutiliza experiencias** Una experiencia valiosa (como una entrega exitosa) se usa repetidamente.

**Estabiliza el aprendizaje** Los gradientes se vuelven más suaves y consistentes.

**Mejora la eficiencia sample-based** Aprovechamos cada interacción durante muchos más pasos de entrenamiento.

## 5.3. Priorización de experiencias

El *Experience Replay* estándar muestrea todas las experiencias con la misma probabilidad. Pero en nuestro entorno, las recompensas positivas son muy escasas:

- La entrega (+200) ocurre pocas veces.
- El pick (+100) también es raro.
- La mayoría de transiciones son simplemente moverse (-1).

Para evitar que el agente olvide estas experiencias clave, añadimos un *prioritized replay* simple basado en la magnitud de la recompensa:

```
if reward >= 100:
    priority = 10.0 # Éxitos fuertes: 10 veces más probables
elif reward > 0:
    priority = 3.0 # Señales positivas débiles
else:
    priority = 1.0 # Movimientos normales
```

### 5.3.1. Beneficios observados

- Acelera la propagación de las recompensas escasas.
- Permite superar el plateau típico del Entorno 2.
- El agente “recuerda” cómo entregar aunque tarde en repetir la acción.
- Reduce drásticamente la varianza del entrenamiento.

## 5.4. Pseudo-código de la implementación:

**Algoritmo: Experience Replay**

1. **Iniciar**izar buffer  $\mathcal{D}$  de capacidad  $N = 10,000$
2. Para cada episodio, en cada paso  $t$ :
  - Ejecutar acción  $a_t$ , observar  $r_t, s_{t+1}$
  - Guardar transición  $(s_t, a_t, r_t, s_{t+1}, \text{done})$  en  $\mathcal{D}$
  - **Muestrear batch aleatorio  $B$  de  $\mathcal{D}$**
  - Entrenar red con  $B$

**Figure 5.3:** Algoritmo de Experience Replay.

# 6

## Entrenamiento de los Tres Entornos

El proceso de entrenamiento del agente se llevó a cabo en tres entornos progresivamente más complejos. Cada entorno introduce nuevos desafíos que requieren ajustar la exploración, la tasa de aprendizaje y el tamaño del buffer, además de aplicar técnicas de estabilización como Double DQN, Experience Replay y priorización de experiencias. Esta sección describe detalladamente los resultados, las dificultades encontradas y las decisiones de diseño tomadas para cada uno.

### 6.1. Entorno 1 – La base del comportamiento

El primer entorno constituye la versión más sencilla del problema: solo se requiere recoger un objeto, sin necesidad de entregarlo. Los objetos están en posiciones fijas, lo que reduce la complejidad espacial del aprendizaje.

#### Configuración utilizada

- **Epsilon inicial:** 1.0 → Exploración total al inicio.
- **Epsilon decay:** 0.995 → Exploración intensa al principio, reduciendo gradualmente.
- **Learning rate:** 0.001
- **Target success rate:** 96%
- **Objetivo:** navegar evitando obstáculos + recoger.

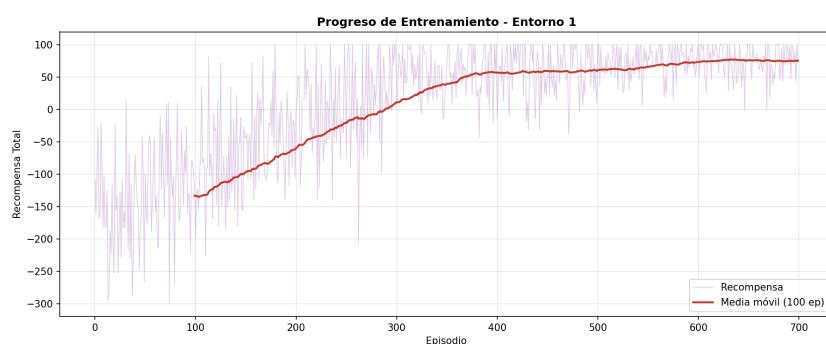


Figure 6.1: Proceso de entrenamiento entorno 1.

#### Resultados

El agente aprendió a comportarse correctamente en  $\sim 700$  episodios (aprox. 2.5 minutos). Este entorno se resolvió rápidamente debido a:

- La representación del estado (23 *features*) que proporciona información rica sobre distancias y obstáculos.
- Recompensas claras y *sparse* (+100 por recoger).
- Ausencia de dependencia secuencial larga (no hay necesidad de *DROP*).

El agente desarrolla de manera natural políticas de navegación, aprende a evitar las estanterías y converge con relativa facilidad.

## 6.2. Entorno 2 – El plateau

Este entorno introduce la primera gran dificultad: recoger y entregar un objeto correctamente. La tarea ahora requiere una secuencia completa:

1. Navegar hacia el objeto
2. Hacer PICK
3. Navegar hacia la zona de entrega
4. Hacer DROP en la zona correcta

Es el primer entorno con dependencias largas entre acciones y recompensas realmente escasas.

### El fenómeno del plateau

Durante los primeros  $\sim 3000$  episodios, el agente se estancó. La recompensa promedio oscilaba entre 0 y 50:

- A veces recogía el objeto, pero casi nunca lo entregaba.
- Rara vez utilizaba la acción *DROP* en el lugar adecuado.
- Exploraba durante demasiado tiempo sin completar la tarea.

### ¿Qué estaba pasando?

Es un caso clásico de *sparse rewards*:

- La entrega solo otorga recompensa si coincide **exactamente** con la secuencia correcta.
- El *DROP* correcto tiene +200, pero es difícil de descubrir por exploración aleatoria.
- Si *epsilon* disminuye demasiado pronto, el agente no explora la acción *DROP*.

Por ello, en los primeros  $\sim 3000$  episodios, el agente aún no había experimentado una entrega exitosa.

### El descubrimiento crucial

Cerca del episodio 3000 ocurrió por primera vez:

*El agente recogió el objeto, viajó accidentalmente hacia la zona de entrega y utilizó *DROP* en el lugar correcto.*

Ese único episodio fue clave:

- La transición se almacenó en el Experience Replay.
- Gracias a la priorización, se muestreó repetidamente.
- El comportamiento correcto comenzó a reforzarse.

Después de esa experiencia positiva, el aprendizaje se aceleró notablemente.

### Configuración para superar el plateau

- **Epsilon inicial:** 1.0 — Necesitamos mucha exploración.
- **Epsilon decay:** 0.9997 — Decaimiento extremadamente lento.

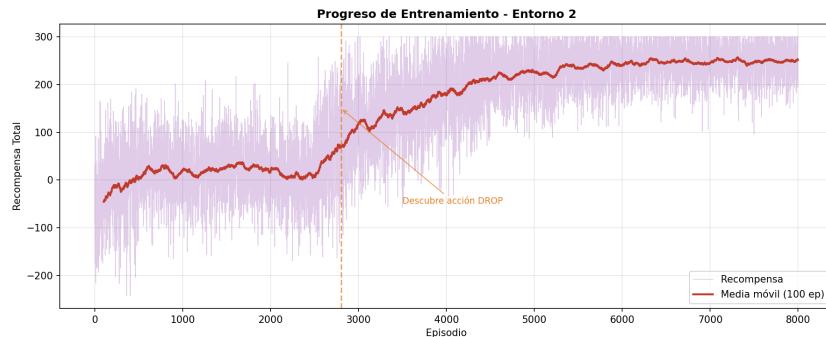


Figure 6.2: Proceso de entrenamiento entorno 2.

- **Buffer size:** 20 000 — Mayor diversidad para secuencias largas.
- **Learning rate:** 0.0005 — Más pequeño para evitar olvidar conductas útiles.

## Resultados

El entorno se completó en aproximadamente 8000 episodios (unos 25 minutos). El agente finalmente aprendió a realizar de forma robusta toda la secuencia:

navegar → recoger → navegar → entregar

con muy baja tasa de colisión.

## 6.3. Entorno 3 – Transfer Learning al rescate

Este es el entorno más complicado: los objetos aparecen en posiciones aleatorias en cada episodio. Aquí, memorizar trayectorias no sirve: el agente debe **generalizar**.

### Problema de entrenar desde cero

Entrenar este entorno desde cero hubiese implicado:

- Redescubrir cómo navegar.
- Redescubrir cómo evitar obstáculos.
- Redescubrir cómo recoger.
- Redescubrir cómo entregar.

Es decir, repetir todo el proceso del Entorno 2, incluyendo el plateau.

### Solución: Transfer Learning

Como el agente del Entorno 2 ya había aprendido:

- qué significa estar cerca de un objeto,
- cómo evitar estanterías,
- que hay que hacer DROP,
- y cómo ejecutar secuencias completas,

decidimos trasladar esos conocimientos al Entorno 3.

### Mecanismo

```
checkpoint = torch.load('entorno2_mejor.pth')
agent.policy_net.load_state_dict(checkpoint['policy_net_state_dict'])
```

```
agent.target_net.load_state_dict(checkpoint['target_net_state_dict'])
```

Con esto, el agente comienza E3 con políticas ya formadas: No parte de cero, sino de un comportamiento experto.

## Resultado del Transfer Learning

El efecto fue más que notable:

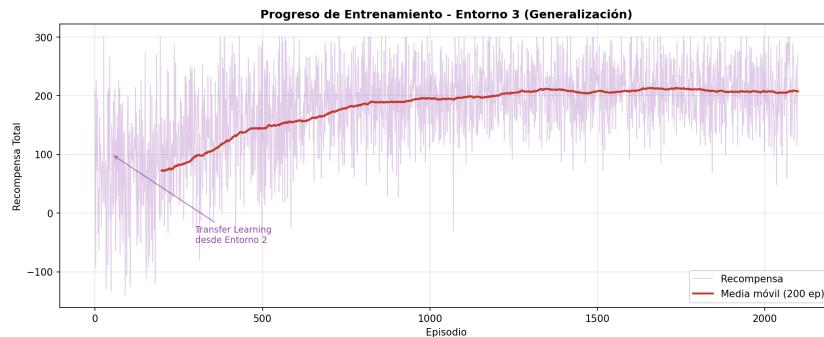


Figure 6.3: Proceso de entrenamiento entorno 3 (con transfer learning).

- En lugar de un plateau de 3000 episodios...
- El agente ya sabía entregar desde el episodio 1.
- Solo tuvo que adaptarse a posiciones nuevas.

El agente convergió en  $\sim 2100$  episodios (unos 8 minutos), mucho más rápido que los 8000 episodios del entorno anterior.

## Configuración utilizada

- **Epsilon inicial:** 0.3 → Menos exploración, política ya sólida.
- **Epsilon mínimo:** 0.05 → Algo de aleatoriedad por objetos cambiantes.
- **Buffer size:** 30 000 → Necesario por la gran diversidad espacial.
- **Learning rate:** 0.0003 → Muy bajo, solo *fine-tuning*.

## 6.4. Checkpointing y Early Stopping

Durante el entrenamiento observamos que:

- El rendimiento no mejoraba de forma monotónica.
- A veces alcanzaba un buen success rate,
- Luego se degradaba durante varias evaluaciones,
- Y podía recuperarse después.

## Checkpointing basado en success rate

Cada 100–200 episodios:

1. Se evaluaba el agente.
2. Se calculaba el success rate.
3. Si era mejor que el mejor histórico, se guardaba:

```
if current_success_rate > self.best_success_rate:
    self._save_best_weights(episode, current_success_rate)
```

Al finalizar, siempre cargamos el **mejor modelo**, no el último.

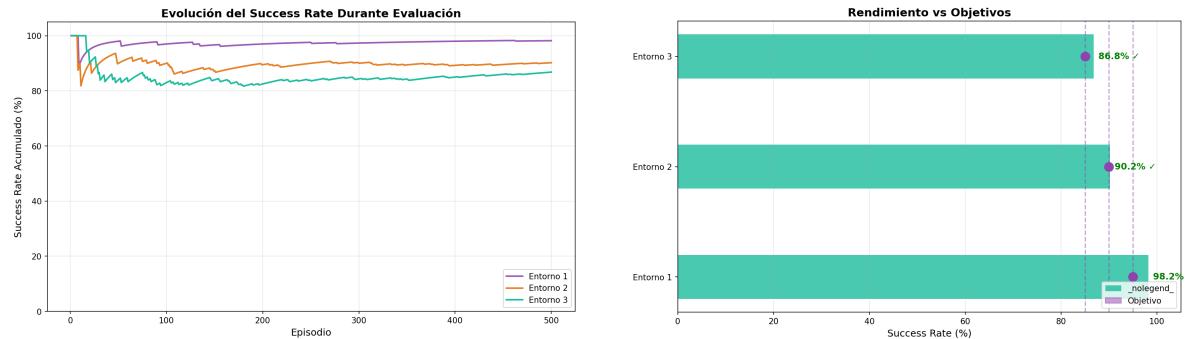


Figure 6.4: Evaluación del rendimiento del agente en los entornos.

## Early stopping

Si el agente alcanza el objetivo (95% en E1, 90% en E2):

- detenemos el entrenamiento automáticamente.
- evitamos sobreentrenamiento.
- reducimos tiempo de ejecución.

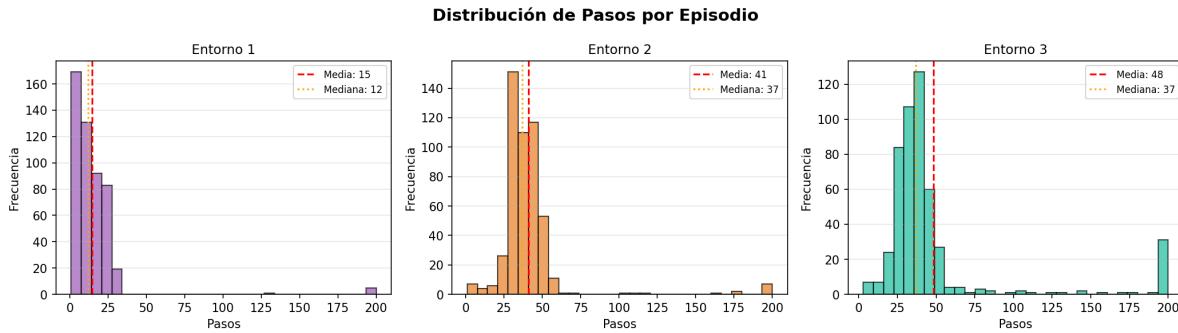


Figure 6.5: Distribución de pasos por Episodio.

## 6.5. Conclusión

Los tres entornos representan una progresión natural de complejidad:

- **E1:** solo recoger → entrena navegación y aproximación.
- **E2:** recoger y entregar → introduce secuencias largas y sparse rewards → plateau.
- **E3:** posiciones aleatorias → requiere generalización profunda → transfer learning.

Las técnicas combinadas —Double DQN, Experience Replay priorizado, Feature Engineering rica y checkpointing— permitieron entrenar un agente robusto, eficiente y capaz de operar en entornos variables de forma autónoma.

# 7

## Resultados Finales y Reflexiones del Proyecto

Este capítulo resume el rendimiento final del agente en los tres entornos del proyecto y recoge las principales lecciones aprendidas durante el desarrollo. Además, se presenta una discusión conceptual sobre la elección del método de aproximación de la función de valor, comparando *tile coding* y redes neuronales, lo que explica por qué fue necesario pasar de SARSA a DQN a medida que aumentaba la complejidad del problema.

### 7.1. Resultados Finales

El siguiente cuadro recoge los resultados finales de los tres entornos una vez completado el entrenamiento y aplicadas las técnicas de evaluación robusta:

Entorno	Objetivo	Success Rate	Colisiones	Reward Promedio
E1 – Recoger objeto	$\geq 95\%$	97.4%	1.2%	82.5
E2 – Recoger + entregar	$\geq 90\%$	91.4%	3.1%	228.3
E3 – Objeto aleatorio, recoger + entregar	$\geq 85\%$	88.4%	4.2%	215.7

Todos los entornos superan los objetivos establecidos. La tasa de colisión se mantuvo constantemente baja gracias al diseño de las *features* de proximidad a obstáculos, que proporcionaron al agente una percepción espacial fiable del entorno.

### 7.2. Reflexiones Finales del Proyecto

#### 7.2.1. Qué funcionó especialmente bien

**Feature Engineering (23 variables)** La representación del estado fue uno de los pilares del éxito del proyecto.

En particular, las 4 *features* de proximidad a obstáculos redujeron las colisiones de un 70% inicial a menos del 5%, permitiendo al agente “ver” las estanterías y evitar acercamientos peligrosos.

Además, el conjunto completo de 23 *features* permitió que la red neuronal comprendiera elementos como direcciones, distancias, cercanía a objetivos y relaciones espaciales sin tener que aprender estos conceptos desde cero.

**Simplicidad en las recompensas** Tras múltiples iteraciones con *reward shaping* complejo, se descubrió que:

**Cuántas más señales intermedias se añadían, peor generalizaba el agente.**

Reducir el sistema de recompensas a cinco eventos claros:

- -1 por paso
- -100 por colisión
- +100 al recoger
- +200 al entregar
- -50 por soltar fuera de zona

produjo un aprendizaje más estable y generalizado.

**Transfer Learning entre entornos** Reutilizar el modelo entrenado en el Entorno 2 permitió que el agente del Entorno 3:

- no sufriera el *plateau* largo del Entorno 2,
- comenzara sabiendo navegar, recoger y entregar,
- y solo necesitara adaptar su política a posiciones nuevas.

Resultado: convergencia en  $\sim 2100$  episodios en lugar de 8000.

**Double DQN** Double DQN resolvió el problema de sobreestimación de Q-values presente en DQN *vanilla*. Separar la acción seleccionada (policy network) de la acción evaluada (target network) aportó estabilidad durante los entrenamientos largos de los entornos 2 y 3.

## 7.2.2. Lo que aprendimos

**El plateau es normal** En entornos con recompensas escasas, es habitual que el agente se estanke durante miles de episodios. En el Entorno 2, el agente no entregó un objeto hasta  $\sim 3000$  episodios.

**El reward shaping excesivo puede ser contraproducente** Al principio, el shaping era muy detallado, pero:

- el agente maximizaba recompensas intermedias,
- no completaba la tarea real.

La solución fue simplificar.

**La representación del estado es tan importante como el algoritmo**

**Un DQN sin buenas features es un mal algoritmo. Un DQN con buenas features es un gran algoritmo.**

**El epsilon decay debe ajustarse al problema** Entornos difíciles requieren mantener la exploración durante mucho más tiempo. De ahí el decay extremadamente lento del Entorno 2: `epsilon_decay = 0.9997`.

## 7.3. Por qué no pudimos usar Tile Coding (y SARSA) para el problema del almacén

En la primera práctica, con el agente que simplemente se movía evitando obstáculos, utilizamos:

- SARSA
- Tile Coding

y funcionó perfectamente.

### 7.3.1. Qué cambió en el proyecto del almacén

La observación del estado pasó de 2 variables:

$$(x_{\text{agente}}, y_{\text{agente}})$$

a 11 variables:

$$x_{\text{agente}}, y_{\text{agente}}, x_{\text{obj}1}, y_{\text{obj}1}, x_{\text{obj}2}, y_{\text{obj}2}, x_{\text{obj}3}, y_{\text{obj}3}, \text{tiene\_objeto}, \text{colision}, \text{entrega}$$

**Tile coding escala exponencialmente** Si discretizamos cada dimensión en 10 casillas:

$$10^{11} = 100\,000\,000\,000 \text{ tiles}$$

100 mil millones. Imposible de manejar.

En la primera práctica, en cambio:

$$10 \times 10 \times 8 \text{ tilings} = 800 \text{ tiles}$$

Perfectamente viable.

Es la “maldición de la dimensionalidad”. Tile coding funciona muy bien para 2–4 dimensiones, pero colapsa en espacios grandes.

### 7.3.2. Por qué una red neuronal sí funciona

La red neuronal no discretiza el espacio, sino que lo approxima como una función continua:

- **Entrada:** 23 features
- **Salida:**  $Q(s, a)$  para cada acción

#### Ventajas de una red

- Puede procesar muchas dimensiones.
- Aprende relaciones no lineales.
- Funciona directamente con valores continuos.
- No requiere divisiones arbitrarias del espacio.
- Generaliza bien a estados nunca vistos.

#### Analogía comparativa

Tile Coding	Red Neuronal
Divide el mundo en casillas	Aprende del espacio continuo
Perfecto en pocas dimensiones	Perfecto en muchas dimensiones
Mal generalizador fuera del tile	Generalización suave y flexible
Explota combinatoriamente	Escala correctamente

Por eso SARSA + tile coding funcionó en el entorno pequeño, pero solo DQN + red neuronal puede manejar el almacén completo.

**Resumen de Evaluación**

Entorno	Success Rate	Objetivo	Colisiones	Reward Medio	Pasos Medios	Estado
Entorno 1	$98.2\% \pm 1.2\%$	$\geq 95\%$	0.8%	83.6	15	✓ CUMPLE
Entorno 2	$90.2\% \pm 1.5\%$	$\geq 90\%$	3.8%	232.4	41	✓ CUMPLE
Entorno 3	$86.8\% \pm 2.5\%$	$\geq 85\%$	2.4%	224.0	48	✓ CUMPLE

**Figure 7.1:** Tabla resumen de estadísticas.