APRENDIZAJE POR REFUERZO: Q-LEARNING

Pablo Chantada Saborido & José Romero Conde

Introducción

Como forma de aprendizaje, decidimos usar una aproximación por épocas. De esta forma, generamos un entrenamiento ajustable (por defecto: 5 episodios con 100 épocas[1]). Añadimos un archivo de *utils.py*, con unas funciones complementarias para la tabla que salen fuera del marco del ejercicio.

Selección de Hiperparámetros

La ecuación fundamental del Q-learning, conocida como la ecuación de Bellman, se expresa como:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \cdot [r + \gamma \cdot \max_{a'} Q(s',a') - Q(s,a)] \tag{1}$$

Donde:

- $Q(s,a)$ es el valor Q actual para el par estado-acción (s,a).
- α es la tasa de aprendizaje, que determina cuánto influyen las nuevas experiencias.
- r es la recompensa recibida después de ejecutar la acción a en el estado s.
- γ es el factor de descuento, que determina la importancia de las recompensas futuras.
- s' es el nuevo estado alcanzado.
- $\max_{a'} Q(s',a')$ es el máximo valor Q posible en el nuevo estado s'.

Los valores que se ajustan en el código son: ϵ, α y γ, las dos últimas ambas aumentadas +0,2. Este aumento se realiza por una mejora en comparación con los valores recomendados en la guía (0.4 y 0.5 respectivamente). Con los valores predefinidos obteníamos una peor convergencia de las iteraciones, mientras que con los nuevos se obtiene una solución más rápido y en menos pasos. La selección de epsilon se realizo de manera arbitraria, y no se cambio con el transcurso de las iteraciones gracias a buenos resultados con los valores inciales.

Descripción de la implementación

Definición de estados y acciones

Estados

En nuestra implementación, hemos definido los siguientes estados basados en la posición de la luz respecto al robot:

- Estado 0: Luz muy a la izquierda
- Estado 1: Luz poco a la izquierda

[1] Estos hiperparámetros fueron seleccionados a base de prueba y error, las épocas son elevadas para intentar evitar obstrucciones.

- Estado 2: Luz al centro
- Estado 3: Luz poco a la derecha
- Estado 4: Luz muy a la derecha
- Estado 5: Estado no válido (sin luz visible)

Los estados se determinan utilizando los sensores de luz del robot y moviendo el "pan"para tomar lecturas en diferentes ángulos $[-90°, -45°, 0°, 45°, 90°]$. Además, implementamos una función (comentada en el código entregado) que gira el robot 4 veces y obtiene los valores de los sensores de luz; de esta forma, se genera un giro de 360^{o} devolviendo la posición más fuerte de luz. Sin embargo, no obtuvimos resultados con una mejora considerable por lo que decidimos desecharla de la ejecución final.

Acciones

Las acciones disponibles para el robot son:

- Acción 0: Girar mucho a la derecha
- Acción 1: Girar poco a la derecha
- Acción 2: Avanzar recto
- Acción 3: Girar poco a la izquierda
- Acción 4: Girar mucho a la izquierda

Sistema de recompensas

Las recompensas se calculan en función del cambio en la intensidad de luz percibida por el robot después de cada acción:

- Recompensa = 1.0 si el brillo aumenta más de 50 unidades
- Recompensa = 0.5 si el brillo aumenta entre 20 y 50 unidades
- Recompensa = 0.2 si el brillo aumenta entre 0 y 20 unidades
- Recompensa = -0.1 si el brillo cambia mínimamente (entre -5 y 0)
- Recompensa = -0.2 si el brillo disminuye más de 5 unidades
- Recompensa = 1.0 si se alcanza el objetivo (brillo 350 lux)
- Recompensas negativas adicionales para eventos de riesgo (caídas, colisiones)

Mecanismos de seguridad

Se han implementado mecanismos para detectar y evitar colisiones y caídas:

- **Detección de caídas**: Utilizando los sensores IR frontales para detectar bordes.
- **Detección de colisiones**: Mediante los sensores IR para detectar obstáculos.
- **Estrategias de evasión**: Retroceder y girar en la dirección opuesta al peligro detectado.

Análisis de resultados

Evolución del aprendizaje

Durante el entrenamiento, se observó cómo la tabla Q evolucionaba a lo largo de los episodios. Inicialmente, los valores Q son pequeños y aleatorios [2], pero a medida que el robot avanza por el entorno, estos valores se ajustan para reflejar las recompensas acumuladas.

A continuación se muestra un ejemplo de la tabla Q final después del entrenamiento:

Estado	Acción 0	Acción 1	Acción 2	Acción 3	Acción 4
0 (Muy izquierda)	0.038	0.243	0.086	0.102	**0.883**
1 (Poco izquierda)	0.071	0.121	-0.099	0.157	**1.961**
2 (Centro)	0.004	0.090	**1.464**	-0.071	0.057
3 (Poco derecha)	0.189	0.441	**0.872**	-0.025	-0.044
4 (Muy derecha)	**1.886**	0.067	-0.061	0.037	-0.006
5 (No válido)	-0.016	0.086	-0.059	-0.089	0.083

Cuadro 1: Tabla Q final después del entrenamiento

En esta tabla se puede apreciar claramente el aprendizaje del robot. Los valores más altos (resaltados en negrita) muestran las acciones preferidas para cada estado:

- Para luz muy a la izquierda (estado 0): Girar mucho a la izquierda (acción 4)
- Para luz poco a la izquierda (estado 1): Girar mucho a la izquierda (acción 4)
- Para luz al centro (estado 2): Avanzar recto (acción 2)
- Para luz poco a la derecha (estado 3): Avanzar recto (acción 2)
- Para luz muy a la derecha (estado 4): Girar mucho a la derecha (acción 0)

Este comportamiento es lógico y demuestra un aprendizaje exitoso: el robot aprende a girar hacia la dirección donde detecta más luz y a avanzar cuando la luz está centrada o ligeramente a la derecha. Los casos que difieren entre luz y acción se puede deber a la posición del robot en el mundo, obstaculos, etc.

Conclusiones

Los resultados obtenidos utilizando únicamente el Algoritmo Q-Learning son buenos, sin embargo, hay factores a tener en cuenta. El algoritmo final que obtenemos (Q-Table) sigue siendo un proceso no determinista, por ello puede haber ocasiones en las que el robot falle en la ejecución por una mala selección de acciones. Además, una implementación sola de este algoritmo puede no ser la mejor opción, una mezcla de algoritmos (i.e: Q-Learning + A*, Q-Learning + PID) seguramente genere mejores resultados. Por último, considerar que el entorno utilizado es muy simple y una implementación en la vida real podría considerar una Q-Table final extremadamente grande o incluso imposible en entornos muy complejos.

[2] Decidimos no usar una incialización con 0's, para intentar generar una mejor incialización.

Estructura del código

```
def initialize_q_table():
# Inicializa la tabla Q con valores aleatorios pequenos
def update_q_table(reward, state, new_state, action):
# Implementa la ecuacion de Bellman para actualizar la tabla Q
gamma = 0.7
alpha = 0.6
Qtable[state][action] = Qtable[state][action] + alpha * (reward + gamma *
    max(Qtable[new_state]) - Qtable[state][action])
def take_action(action):
# Ejecuta la accion seleccionada y devuelve la recompensa
def get_state():
# Determina el estado actual basado en las lecturas de los sensores
def check_collision_risk():
# Comprueba si hay riesgo de colision
def check_fall_risk():
# Comprueba si hay riesgo de caida
def handle_collision():
# Maneja la evasion de obstaculos
def select_action(state, epsilon=0.2):
# Selecciona una accion segun la politica epsilon-greedy
def train_robot():
# Funcion principal que implementa el entrenamiento por episodios
```

Listing 1: Estructura del código Q-learning

Utils

```
def initialize_q_table():
# Inicializa la tabla Q con valores aleatorios pequenos

def load_q_table(filename='q_table.npy'):
# Carga una tabla Q desde un archivo

def print_q_table(state, action, reward, q_table, test=False):
# Imprime el estado actual y la tabla Q de forma formateada
```

Listing 2: Funciones complemtarias para la tabla