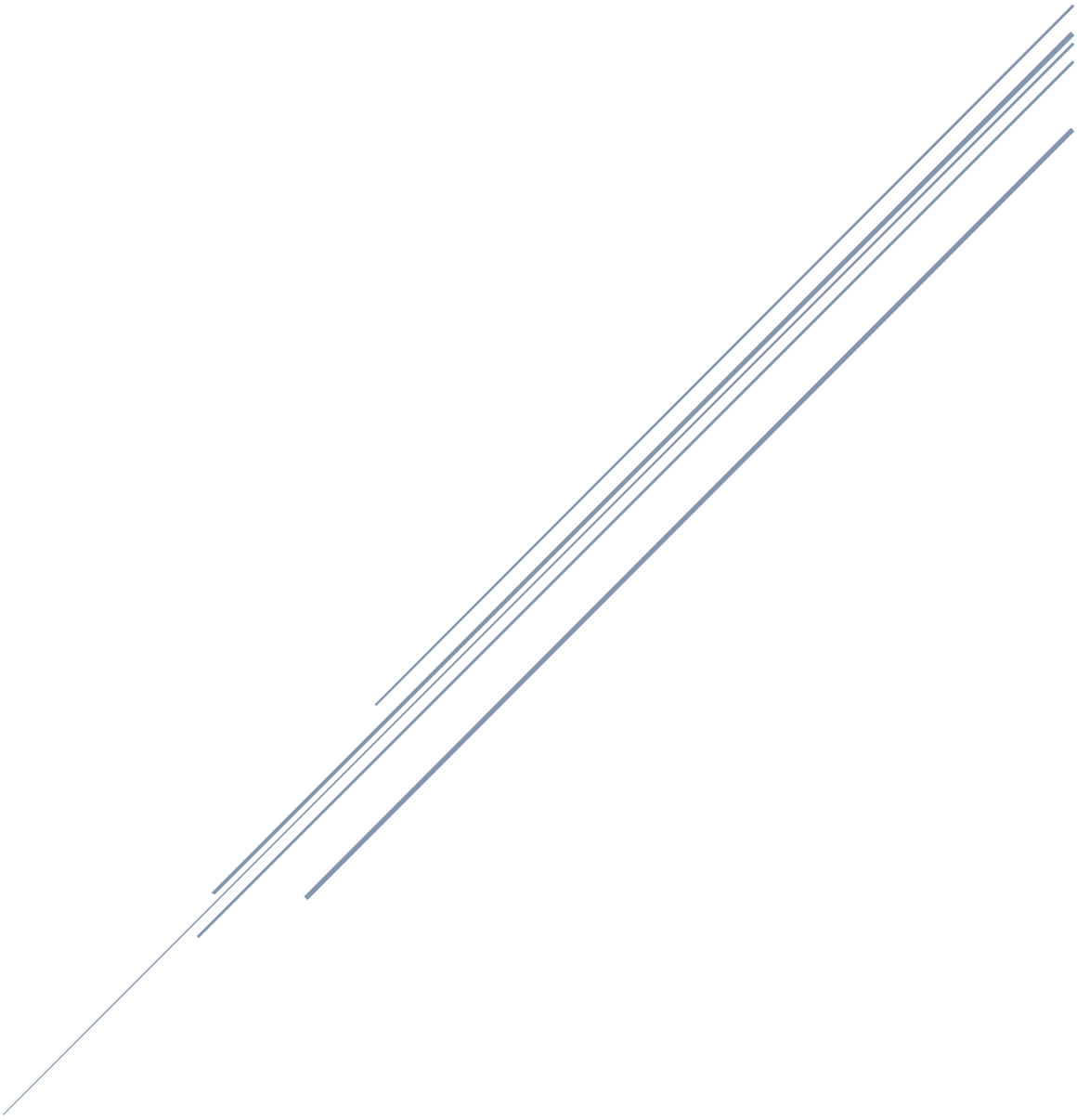
APRENDIZAJE AUTOMÁTICO

Tutorial 4

Universidad Carlos III Madrid, Ingeniería Informática



IVAN AGUADO PERULERO – 100405871
JORGE SERRANO PÉREZ – 100405987

Contenido

1.- INTRODUCCIÓN

El objetivo principal de esta práctica es familiarizarse con el software donde se tiene que implementar *Q-learning* en el dominio de *GridWorld*. Este se trata de un algoritmo de aprendizaje automático por refuerzo, en el que el agente aprende una serie de normas que le dicen que acción tomar ante distintas circunstancias.

En cuanto al presente documento, en primer lugar tendremos la portada con los miembros del grupo y el índice. A continuación, los ejercicios con una serie de pasos a ejecutar que se nos proponen en el enunciado, en los que hemos respondido a una serie de preguntas (eliminando las que no requieren respuesta), y por último unas conclusiones finales sobre las dificultades encontradas.

2.- EJERCICIO 1

1. ¿Cuántas celdas/estados aparecen en el tablero? ¿Cuántas acciones puede ejecutar el agente? Si quisieras resolver el juego mediante aprendizaje por refuerzo, ¿cómo lo harías?

Los estados posibles del tablero son 4*3-1 = 11, ya que se trata de una matriz 4x3 con una casilla en gris a la que no se puede acceder. Las acciones posibles que puede ejecutar el agente son 5 (*WEST, SOUTH, EAST, NORTH, EXIT*).

En caso de querer resolver el juego por refuerzo, dado que es un proceso iterativo de prueba y error, habría que recompensar aquellas acciones que le acercan al estado final bueno y penalizar los movimientos hacia la casilla con el -1 que hacen perder la partida.

2. Abrir el fichero *qlearningAgents.py* y buscar la clase *QLearningAgent*. Describir los métodos que aparecen en ella.

La clase *QLearningAgent* se corresponde, como indica su nombre, con el agente que hace uso de *Q-learning*. Posee los siguientes métodos:

- ***__init__***: Inicializa la *tabla q* y sus valores iniciales.
- ***readQTable***: Lee la *tabla q* de la memoria haciendo uso de un bucle para extraer los datos.
- ***writeQtable***: Escribe la *tabla q* en memoria.
- ***printQtable***: Muestra por pantalla la *tabla q* imprimiendo cada fila.
- ***__del__***: Destructor, el cual se invoca al final de cada episodio y que cierra también el archivo de la *tabla q*.
- ***computePosition***: Calcula la fila de la *tabla q* para un estado dado.
- ***getQValue***: Devuelve el valor para un estado y una acción determinada de la *tabla q*.

- ***computeValueFromQValues***: Devuelve el mejor valor de la *tabla q* para un estado teniendo en cuenta las acciones legales. Si no las hay, devuelve 0.
- ***computeActionFromQValues***: Calcula y devuelve la mejor acción a tomar según la *tabla q*, dado un estado y dentro de las acciones legales. Si no las hay, devuelve *None*.
- ***getAction***: Elige la acción a realizar en un determinado estado. Con probabilidad *epsilon*, se elige aleatoriamente o coge la de mejor política dentro de las acciones legales. Si no las hay, devuelve *None*.
- ***update***: Muestra por pantalla la *tabla q* actualizada e indica la celda que ha sido modificada y la transición realizada con su recompensa.
- ***getPolicy***: Devuelve la mejor acción de la *tabla q* para un estado dado.
- ***getValue***: Devuelve el valor más alto de la *tabla q* para un estado dado.

4. ¿Qué información se muestra en el laberinto? ¿Qué aparece por terminal cuando se realizan los movimientos en el laberinto?

En el laberinto se muestra el valor de la *tabla q* para cada una de las acciones en todos los estados posibles. Se muestra también los dos estados finales.

Mientras tanto, para cada episodio, por la terminal se muestra el estado en el que empezó el agente en ese turno, la acción elegida, el estado al que ha transitado y la recompensa obtenida. Además, se muestra también la *tabla q* actualizada e indica la celda que ha sido modificada y la transición realizada con el estado, la acción escogida, el siguiente estado y la recompensa obtenida. Al final de cada episodio se muestran los resultados obtenidos, y al finalizarlos todos, una media de los mismos.

5. ¿Qué clase de movimiento realiza el agente anterior?

El agente realiza movimientos deterministas, pues el parámetro *-n* está puesto a 0.

En este momento los movimientos son totalmente aleatorios pues la *tabla q* tiene todos sus valores a 0. Si estos se modificaran correctamente, se tendrían en cuenta a la hora de realizar un movimiento, optando por las acciones que más recompensan según la experiencia previa del agente.

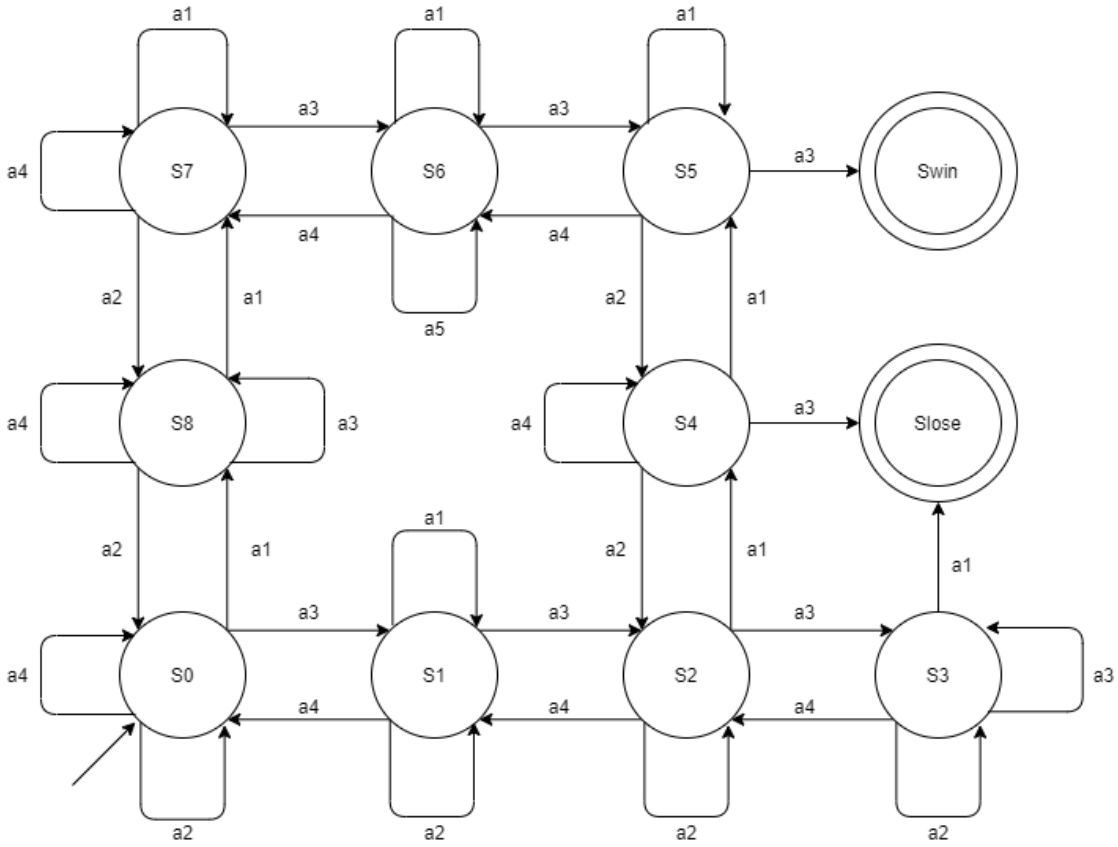
6. Dibujar el MDP considerando un entorno determinista.

Los estados han sido dispuestos simulando las casillas del laberinto a resolver.

Las acciones se identifican de la siguiente manera:

- **a1**: *North*
- **a2**: *South*
- **a3**: *East*
- **a4**: *West*

Cabe destacar que en caso de realizar una acción ilegal el agente no se moverá de la casilla actual, es decir, permanecerá en el mismo estado.



7. ¿Hay varias políticas óptimas? Describe todas las políticas óptimas para este problema.

Podemos apreciar que en este mapa existen dos políticas óptimas de manera general. Obteniéndolas de la *tabla q*, en el estado inicial el desplazarse hacia la derecha o hacia arriba son las acciones que poseen el valor máximo, para después desplazarse hacia arriba en el primer caso y para la derecha en el segundo. Por lo tanto, una política óptima sería:

- *S0 → a1*
- *S8 → a1*
- *S7 → a3*
- *S6 → a3*
- *S5 → a3*

Y la otra:

- *S0 → a3*
- *S1 → a3*
- *S2 → a1*
- *S4 → a1*
- *S5 → a3*

8. Escribir el método *update* de la clase *QLearningAgent* utilizando las funciones de actualización del algoritmo *Q-Learning*.

El método implementado cuenta con dos posibles caminos, cuando el estado actual es terminal, ya sea el terminal bueno o malo, o cuando es cualquier otro estado posible.

Si el estado en el que se encuentra el agente es terminal, es decir *reward* es 1 o -1, se actualizará el valor de la *tabla q* referente al estado y acción ejecutada según esta función:

Q(estado, acción) = (1-alfa) * Q(estado, acción) + alfa*(reward + 0)

Donde alfa es la tasa de aprendizaje y *reward* el refuerzo. Para conseguir el valor de la *tabla q* se ha utilizado la función *getQValue*.

En caso de que el estado actual no sea terminal la función de actualización será la siguiente:

Q(estado, acción) = (1-alfa) * Q(estado, acción) + alfa*(*reward* + *discount* * max(Q(estadoSiguiente, acciónSiguiente)))

Donde *discount* es la tasa de descuento.

Para conseguir el valor máximo de la tabla para el estado siguiente se hace uso de la función *computeActionFromQValues*.

9. Establece en el constructor de la clase *QLearningAgent* el valor de la variable *epsilon* a 0,05. Ejecuta nuevamente con: *python3 gridworld.py -a q -k 100 -n 0* ¿Qué sucede?

La *tabla q* se va actualizando en función del método *update* implementado, lo que se ve reflejado a su vez en el tablero. Con el paso de los episodios, se van reforzando, ya sea positivamente o negativamente, ciertas acciones para cada estado hasta que el agente aprende la política óptima para llegar al estado final bueno. De esta manera consigue llegar a él de la forma más eficiente y rápida.

10. Después de la ejecución anterior, abrir el fichero qtable.txt. ¿Qué contiene?

Contiene la *tabla q* actualizada, con los valores que se han ido calculando durante la ejecución del paso anterior, haciendo uso de la función *update*.

3.- EJERCICIO 2

Ahora utilizaremos un MDP estocástico.

1. Ejecuta y juega un par de partidas con el agente manual: *python3 gridworld.py -m -n 0.3* ¿Qué sucede? ¿Crees que el agente *QLearningAgent* será capaz de aprender en este nuevo escenario?

Las acciones son no deterministas. Es decir, las transiciones de estado y la función de refuerzo son funciones estocásticas, por lo que la misma situación puede producir distintos resultados.

Aunque le costará más, el agente sí que será capaz de aprender en el nuevo escenario, pues la función de actualización implementada en el método *update* es la no determinista, la cual también vale para el caso determinista.

4. Tras unos cuantos episodios, ¿se genera la política óptima? Y si se genera, ¿se tarda más o menos que en el caso determinista?

La política óptima llega a generarse, pero tarda más que en el caso determinista, ya que como se ha mencionado anteriormente, una misma situación puede producir distintos resultados.

4.- CONCLUSIONES

Este tutorial es muy útil para fijar los contenidos de las clases magistrales sobre aprendizaje por refuerzo, pues obliga a poner en práctica la teoría y más en concreto lo aprendido sobre el algoritmo *Q-learning*.

No consideramos esta práctica muy compleja, aunque es cierto que la parte de implementación de la función *update* nos frenó un poco, pero bastó con dar un repaso a la teoría sobre *Q-learning* para poder avanzar.

En definitiva, un tutorial claro que sirve como punto de partida para la siguiente práctica y para repasar y poner en práctica nuestros conocimientos previos.

5.- ANEXO

Las *q tablas* adjuntas en la entrega se han hecho con *epsilon* a 0,05, sin embargo, se ha probado a hacerlo también con la probabilidad de exploración a 1. El resultado fue la exploración de todos los estados y posibles acciones, por lo que el agente encontró las dos posibles políticas óptimas y reforzó sus caminos de la misma manera.