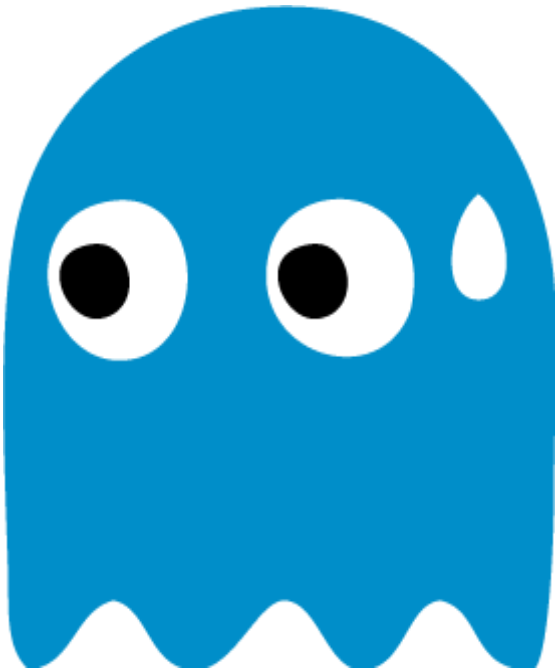
Aprendizaje Automático

# Tutorial 3



uc3m | Universidad Carlos III de Madrid

**Ricardo Prieto Álvarez** 100386267

# ÍNDICE

**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?**
   El tablero muestra 12 celdas en total. Para el agente, existe la posibilidad de ejecutar 4 acciones, siendo estas: Norte, Sur, Este, Oeste y Exit. Lo necesario es encontrar que acción maximiza la recompensa en el futuro para cada estado. Esto se define como la política óptima .

2. **Abrir el fichero qlearningAgents.py y buscar la clase QLearningAgent. Describir los métodos que aparecen en ella.**
   Métodos de la clase QLearningAgent:
   - redQtable(): Como su nombre indica, lee el input qtable.txt, de tal manera que pueda generar y devolver una lista de listas. Cada fila de qtable.txt corresponde a un estado concreto y las columnas indican las acciones posibles.
   - writeQtable(): Reescribe el fichero qtable.txt basándose en los valores del atributoq_table de la clase actual.
   - printQtable(): Imprime por pantalla cada fila de q_table.
   - computePosition(state): Devuelve la fila a la que corresponde un estado determinado.
   - getQtable(state, action): Devuelve el valor de q_table para una fila (correspondiente a un estado) y una acción a tomar.
   - computeValueFromQValues(state): Devuelve el máximo valor para las acciones posibles en un determinado estado.
   - computeActionFromQValues(state): Devuelve la mejor acción a tomar dado un estado determinado.
   - getAction(state): Determina la acción que ofrece la política óptima considerando el valor de epsilon (ruido probabilístico).
   - update(state, action, nextState, reward): Establece la actualización de los valores para Q-table dado un estado determinado y una acción escogida.
   - getPolicy(state): Se llama al método computeActionFromQValues de tal manera que se obtenga la mejor acción para un estado determinado.
   - getValue(state): Se llama al método computeValueFromQValues de tal manera que se obtenga el máximo valor en Q-table para un estado determinado.

3. **Ejecuta ahora el agente anterior con: python gridworld.py -a q -k 100 -n 0**
4. **¿Qué información se muestra en la ventana con en el laberinto? ¿Qué aparece por terminal cuando se realizan los movimientos en el laberinto?**

   En la ventana del laberinto se pueden observar 4 valores, cada uno de ellos corresponde al valor de las posibles acciones a escoger para un estado concreto. Dado que, de momento, no hay una actualización de estos mismos, estos se mantienen en 0.0 constantemente.

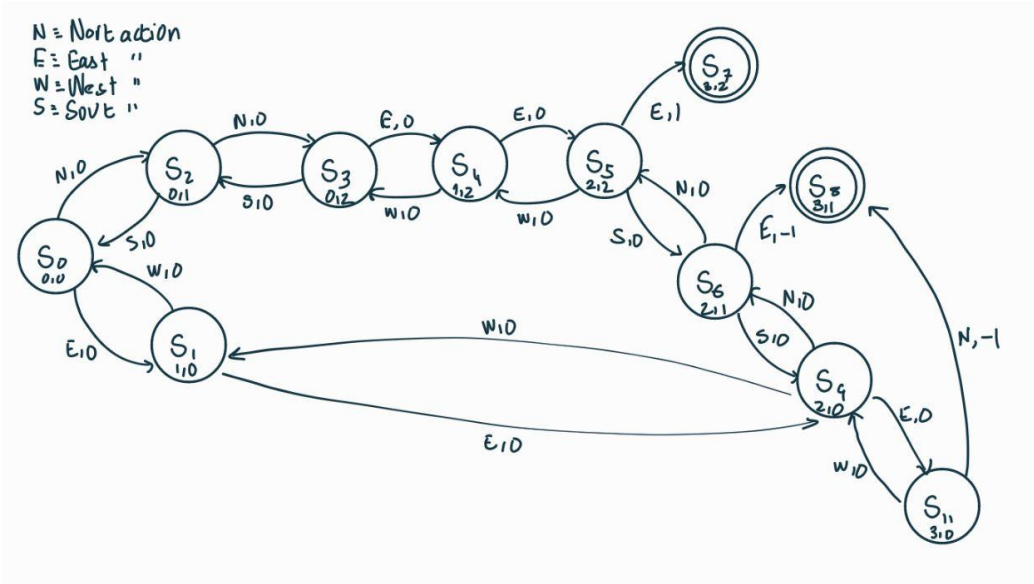
   Por otro lado, el terminal muestra 4 elementos:

- Estado en el que se inició la acción.
- La acción escogida.
- El estado resultante dada la acción escogida.
- La recompensa obtenida.

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

   Al no existir una acción con un valor que represente el máximo, el movimiento es aleatorizado.

6. **Dibujar el MDP considerando un entorno determinista.**




7. **¿Se pueden sacar varias políticas óptimas? Describe todas las políticas óptimas para este problema.**

   Por lo que se puede observar, habría dos políticas óptimas. Ambas implican pasar por el mismo número de estados.

   La primera política óptima es: S0 -> S2 -> S3 -> S4 -> S5 -> S7

La segunda política óptima es: S0 -> S1 -> S9 -> S6 -> S5 -> S7

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

   Primero se comprueba si el estado actual corresponde a un estado terminal. Si es este el caso, el valor de Q(estadoActual, actionTomada) pasa a ser actualizado según el algoritmo Q-Learning basado en una solución ε-greedy.

   - Para el caso de estado terminal, se debe realizar:
     Q(state,action) <- (1-self.alpha) Q(state,action) + self.alpha * r
     Donde el código implementado ha sido:
     self.q_table[self.computePosition(state)][self.actions[action]] = (1-self.alpha)*self.getQValue(state, action) + self.alpha*reward

   - Para el caso de no estado terminal, se debe realizar:
     Q(state,action) <- (1-self.alpha) Q(state,action) + self.alpha * (r + self.discount * max a' Q(nextState, a'))
     Donde el código implementado ha sido:
     self.q_table[self.computePosition(state)][self.actions[action]] = (1-self.alpha)*self.getQValue(state, action) + self.alpha*(reward + self.discount*self.computeValueFromQValues(nextState))

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

   Al realizar 100 ciclos de aprendizaje, se determina que la política óptima para el estado inicial corresponde a la acción North con un valor de 0.59.

   La tabla Q resulta en:

   0.59049 0.3287575213134364 0.265720499429686 0.26572049999999997 0.0
   0.0 0.0 0.0 0.49819941943195695 0.0
   0.0 0.0 0.0 0.0 0.0
   0.0 0.0 0.0 0.0 0.0
   0.6560999999999999 0.29524499999815973 0.39858074997370674 0.43896654934289525 0.0
   0.0 0.0 0.0 0.0 0.0
   0.50625 0.0 0.0 0.0 0.0
   0.0 0.0 0.0 0.0 -0.5
   0.0 0.7289999999999999 0.295244999994325 0.574087499999737 0.0
   0.0 0.8099999999999998 0.6291540527343749 0.0 0.0

0.0 0.8999999999999999 0.09112500000000001 0.36449999999999994 0.0
0.0 0.0 0.0 0.0 1.0

10. **Después de la ejecución anterior, abrir el fichero qtable.txt. ¿Qué contiene?**
    Contiene los valores para cada estado y acción de las políticas óptimas (Q(Si, Ai)). Un ejemplo de esto se observa cómo el estado inicial contiene el máximo valor para la primera columna, es decir, para la acción de North.

## Ejercicio 2

---

1. **Ejecuta y juega un par de partidas con el agente manual: python gridworld.py -m -n 0.3 ¿Qué sucede? ¿Crees que el agente QLearningAgent será capaz de aprender en este nuevo escenario?**
   Al tener un ruido del 0.3, las acciones no son deterministas, esto se ve en el momento en el que se indica una dirección a tomar desde el teclado, sin embargo, toma otra completamente diferente. También se puede observar el valor de retorno tras completar un ciclo.

   El agente QLearningAgent sí será capaz de aprender en este escenario nuevo, dado que, introducir ruido aumenta la capacidad de exploración, pero no evita encontrar la política óptima para llegar al estado terminal de recompensa 1.

2. **Reiniciar los valores de la tabla Q del fichero qtable.txt. Para ello ejecutar desde el terminal: cp qtable.ini.txt qtable.txt**
3. **Ejecutar el agente QLearningAgent: python gridworld.py -a q -k 100 -n 0.3**
4. **Tras unos cuantos episodios, ¿se genera la política óptima? Y si se genera, ¿se tarda más o menos que en el caso determinista?**
   Para los 100 episodios de entrenamiento seleccionados, se ha llegado a la política óptima. Parece que se ha llegado a encontrar esta solución dependiendo de un mayor número de episodios en comparación con el caso determinista.

   La tabla Q resulta en:
   0.44657570839203076 0.3097037798724747 0.003944288671875002 0.0640373310047435 0.0
   0.0 0.0 0.31383283789724375 0.05607364224140069 0.0

0.2797712153712609 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0
0.49950849583885415 0.21871464284858996 0.25434421296546905 0.16682456935829654 0.0
0.0 0.0 0.0 0.0 0.0
0.21188254201153947 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 -0.998046875
0.2788645915055888 0.6181788715733305 0.5279464422815223 0.522995664574899 0.0
0.3461877194532337 0.7819308035987114 0.39684234968395327 0.4850976669339766 0.0
0.5706983409615625 0.8731041445077434 0.0 0.3972124323440651 0.0
0.0 0.0 0.0 0.0 1.0