**Práctica 4: Aprendizaje por refuerzo**

Pacman busca recompensa.  
¿Debe comer o debe correr?  
Para resolver sus dudas, él debe q-aprender.

**Dates d'entrega:**  
  
Grup B (Dimarts de 19 a 21): *Entregueu fins****dia 17/12/2012***  
Grup A (Divendres de 17 a 19):*Entregueu fins****dia 20/12/2012***   
Grup F (Divendres de 17 a 19): *Entregueu fins****dia 20/12/2012***

**Introducción**

En esta práctica implementaréis la iteración de valores y el q-aprendizaje. Probaréis vuestros agentes primero en el mundo rejilla (Gridworld) y después los aplicaréis a un controlador de robot simulado (Crawler) y al Pacman.

La práctica contiene 9 cuestiones, 7 obligatorias y 2 opcionales:

* Cuestiones obligatorias (10 puntos): 1 (2,5p), 2 (0,5p), 3 (2p), 4 (2p), 5 (1p), 8 (0,5p), 9 (1,5p)
* Cuestiones opcionales (1 puntos): 6 (0.5p), 7 (0.5p)

De manera que si completáis las obligatorias podéis obtener hasta 10 puntos, y si además abordáis las opcionales podéis llegar a 11 puntos.

Es importante que expliquéis en la memoria detalladamente qué habéis hecho, cómo y porqué en cada cuestión para puntuarla positivamente.

El código para este proyecto contiene los siguientes archivos, que están disponibles [aquí](https://campusvirtual2.ub.edu/pluginfile.php/326412/mod_resource/content/5/reinforcement.zip):

**Ficheros que editaréis**

|  |  |
| --- | --- |
| [valueIterationAgents.py](https://campusvirtual2.ub.edu/pluginfile.php/326412/mod_resource/content/5/docs/valueIterationAgents.html) | Un agente que implementa iteración de valores para resolver PDMs conocidos. |
| [qlearningAgents.py](https://campusvirtual2.ub.edu/pluginfile.php/326412/mod_resource/content/5/docs/qlearningAgents.html) | Agentes que implementan Q-aprendizaje para Gridworld, Crawler y Pacman |

**Ficheros que debéis leer pero NO editar**

|  |  |
| --- | --- |
| [mdp.py](https://campusvirtual2.ub.edu/pluginfile.php/326412/mod_resource/content/5/docs/mdp.html) | Define métodos generales sobre PDMs. |
| [learningAgents.py](https://campusvirtual2.ub.edu/pluginfile.php/326412/mod_resource/content/5/docs/learningAgents.html) | Define la clase base ValueEstimationAgent yQLearningAgent, que vuestros agentes extenderán. |
| [util.py](https://campusvirtual2.ub.edu/pluginfile.php/326412/mod_resource/content/5/docs/util.html) | Utilidades, incluyendo util.Counter, que son particularmente útiles para q-aprendedores. |
| [gridworld.py](https://campusvirtual2.ub.edu/pluginfile.php/326412/mod_resource/content/5/docs/gridworld.html) | Implementación del Gridworld |
| [featureExtractors.py](https://campusvirtual2.ub.edu/pluginfile.php/326412/mod_resource/content/5/docs/featureExtractors.html) | Clases para extraer características en pares (estado,accion). Utilizadas por el agente q-aprendedor aproximado (en qlearningAgents.py). |

**Ficheros que podéis ignorar**

|  |
| --- |
|  |
| [environment.py](https://campusvirtual2.ub.edu/pluginfile.php/326412/mod_resource/content/5/docs/environment.html) | Clase abstracta para entornos generales de aprendizaje por refuerzo. Utilizada por gridworld.py. |
| [graphicsGridworldDisplay.py](https://campusvirtual2.ub.edu/pluginfile.php/326412/mod_resource/content/5/docs/graphicsGridworldDisplay.html) | Interficie gráfica para Gridworld. |
| [graphicsUtils.py](https://campusvirtual2.ub.edu/pluginfile.php/326412/mod_resource/content/5/docs/graphicsUtils.html) | Utilidades gráficas. |
| [textGridworldDisplay.py](https://campusvirtual2.ub.edu/pluginfile.php/326412/mod_resource/content/5/docs/textGridworldDisplay.html) | Plug-in para la interficie textual del Gridworld. |
| [crawler.py](https://campusvirtual2.ub.edu/pluginfile.php/326412/mod_resource/content/5/docs/crawler.html) | Código del crawler. Lo ejecutaréis pero no lo editaréis. |
| [graphicsCrawlerDisplay.py](https://campusvirtual2.ub.edu/pluginfile.php/326412/mod_resource/content/5/docs/graphicsCrawlerDisplay.html) | GUI para el robot crawler. |

**¿Qué debéis hacer?:** Rellenaréis porciones de [valueIterationAgents.py](https://campusvirtual2.ub.edu/pluginfile.php/326412/mod_resource/content/5/docs/valueIterationAgents.html), [qlearningAgents.py](https://campusvirtual2.ub.edu/pluginfile.php/326412/mod_resource/content/5/docs/qlearningAgents.html), y [analysis.py](https://campusvirtual2.ub.edu/pluginfile.php/326412/mod_resource/content/5/docs/analysis.html). Sólo debéis enviar estos ficheros (junto con el documento). Por favor no modifiquéis ningún otro fichero.

**PDMs**

Para comenzar, ejecutar el Gridworld en modo control manual, utilizando las flechas para controlarlo:

python gridworld.py -m

Veréis el tablero con dos salidas que os expliqué en clase. El punto azul es el agente. Cuando presionáis *arriba*, el agente solo se mueve hacia el norte el 80% de las veces. ¡La vida es así para un agente en el mundo rejilla!

Podéis controlar diversos aspectos de la simulación. Obtendréis una lista completa de opciones ejecutando:

python gridworld.py -h

El agente por defecto se mueve de forma aleatoria

python gridworld.py -g MazeGrid

Veréis que el agente se mueve aleatoriamente por la rejilla hasta que se encuentra con una salida. No es el agente más inteligente que uno se pudiera imaginar.

*Nota:* El PDM Gridworld funciona de forma que primero entramos en un estado pre-terminal (con doble recuadro en la GUI) y a continuación realizamos la acción especial 'exit' antes de que termine el episodio (en el estado terminal real, llamado TERMINAL\_STATE, que no se muestra en la GUI). Si ejecutáis un episodio manualmente, la recompensa total puede ser menos de la que esperáis debido a la tasa de descuento (podéis cambiarla con -d; por defecto es 0.9).

Mirad la salida por consola que acompaña a la salida gráfica (o usad -t para funcionar en modo texto). Os dice cada transición que el agente realiza (para quitarlo, utilizad -q).

Como en Pacman, las posiciones se representan mediante coordenadas cartesianas (x,y) y los arrays están indexados por [x][y], siendo 'north' la dirección de ycreciente, etc. Por defecto la mayoría de las transiciones reciben una recompensa de cero, aunque esto se puede cambiar con la opción -r.

***Cuestión 1 (2,5 puntos)***

Escribir un agente que ejecute iteración de valores en ValueIterationAgent, que se os da parcialmente especificado en [valueIterationAgents.py](https://campusvirtual2.ub.edu/pluginfile.php/326412/mod_resource/content/5/docs/valueIterationAgents.html). Vuestro agente de iteración de valores es un planificador "offline", no un agente de refuerzo, de forma que la opción relevante para el aprendizaje es el número de iteraciones que se deben ejecutar (opción -i) durante su fase de planificación inicial. ValueIterationAgent recibe un PDM en construcción y ejecuta iteración de valores durante el numero de iteraciones especificado antes de que el constructor retorne.

La iteración de valores computa estimaciones de el valor óptimo en k pasos , Vk. Además de ejecutar iteración de valores, implementad los siguientes métodos enValueIterationAgent utilizando Vk.

* getValue(state) retorna el valor de un estado.
* getPolicy(state) retorna la mejor acción en un estado de acuerdo con los valores calculados hasta ahora.
* getQValue(state, action) retorna el q-valor de un par (estado, acción).

Todas estas cantidades se muestran en la GUI: Los valores son números en los cuadrados, los q-valores son números en los laterales del cuadrado y las políticas son flechas.

*Importante:* Utilizad la versión "batch" de la iteración de valores donde cada vector Vk se computa a partir de un vector fijo Vk-1 (como en clase), no la versión "online" donde se reescribe un único vector. La diferencia se discute en [Sutton & Barto](http://www.cs.ualberta.ca/~sutton/book/ebook/node41.html) en el sexto párrafo de la sección 4.1.

*Nota:* Una política creada a partir de los valores a profundidad k (que reflejan las siguientes k recompensas) reflejará las siguientes k+1 recompensas (es decir, devolveréis πk+1). De forma similar, los q-valores también reflejarán una recompensa más que los valores (devolveréis Qk+1). Podéis asumir que 100 iteraciones es suficiente para alcanzar la convergencia en las cuestiones que siguen.

El siguiente comando carga vuestro ValueIterationAgent, que calculará una política y la ejecutará 10 veces. Presionar una tecla par visualizar valores, q-valores y la simulación. Veréis que el valor del estado inicial (V(start)) y la media de recompensas se encuentran muy próximas.

python gridworld.py -a value -i 100 -k 10

*Ayuda:* En el BookGrid por defecto, correr iteración de valores 5 iteraciones debe daros la siguiente salida:

python gridworld.py -a value -i 5

*Ayuda:* Utilizar la clase util.Counter en [util.py](https://campusvirtual2.ub.edu/pluginfile.php/326412/mod_resource/content/5/docs/util.html), que es un diccionario con un valor por defecto de cero. Métodos como totalCount deberían simplificar vuestro código. De todas formas, tener cuidado con argMax: el argmax que queréis pudiera ser una clave que no estuviera en el contador!

***Cuestión 2 (0.5 puntos)***

En BridgeGrid con el descuento por defecto de 0.9 y el valor de ruido por defecto de 0.2 la política óptima no cruza el puente. Cambiad sólo UNO de los parámetros de descuento y ruido de forma que la política óptima haga que el agente intente cruzar el puente. Incluid vuestra respuesta en el documento de la práctica. (Ruido se refiere a cuan a menudo un agente termina en un estado sucesor al que no quería dirigirse cuando realiza una acción) El valor por defecto se corresponde con:

python gridworld.py -a value -i 100 -g BridgeGrid --discount 0.9 --noise 0.2

***Cuestión 3 (2 puntos)***

En el DiscountGrid, dar una asignación de parámetros par el descuento, el ruido y la recompensa por vivir que genere los siguientes tipos de políticas óptimas o decir que la política es imposible. La política por defecto se corresponde con:

python gridworld.py -a value -i 100 -g DiscountGrid --discount 0.9 --noise 0.2 --livingReward 0.0

1. Escoger la salida cercana (+1), arriesgándose a caer en el acantilado (-10)
2. Escoger la salida cercana (+1), pero evitando el acantilado (-10)
3. Preferir la salida lejana (+10), arriesgándose a caer en el acantilado (-10)
4. Preferir la salida lejana (+10), pero evitando el acantilado (-10)
5. Evitar ambas salidas (también evitando el acantilado)

*Nota:* Podéis verificar vuestras políticas en la GUI. Por ejemplo, utilizando una respuesta correcta a 3(a), la flecha en (0,1) debería apuntar al este, la flecha en (1,1) debería también apuntar al este y la flecha en (2,1) debería apuntar al norte.

**Q-aprendizaje**

Daos cuenta que vuestro agente que usa value iteration no aprende de la experiencia. Mas bien, analiza su modelo PDM para alcanzar una política complete antes de comenzar a interactuar con el entorno real. Cuando interactúa con el entorno, simplemente sigue la política pre-calculada (se convierte en un agente reflejo). Esta distinción puede ser sutil en un entorno simulado como el Gridword, pero es muy importante en el mundo real, donde el PDM real no suele estar a nuestra disposición antes de comenzar.

***Cuestión 4 (2 puntos)***

Debéis escribir un agente con q-aprendizaje, que planifique muy poco inicialmente pero que en su lugar aprenda por ensayo y error a través de las interacciones con el entorno mediante su método update(state, action, nextState, reward). Un esquema del q-aprendedor aparece QLearningAgent en[qlearningAgents.py](https://campusvirtual2.ub.edu/pluginfile.php/326412/mod_resource/content/5/docs/qlearningAgents.html), y podéis seleccionarlo con la opción '-a q'. Para esta cuestión, debéis implementar los métodos update, getValue, getQValue, ygetPolicy.

*Nota:* Para getValue y getPolicy, deberíais romper los empates de forma aleatoria para que se comportase mejor. La función random.choice() puede seros útil. En un estado particular, las acciones que vuestro agente *no* ha visto antes también tienen un Q-valor óptimo, específicamente un Q-valor de cero, y si todas las acciones que vuestro agente *ha* visto antes tienen Q-valor negativo una acción no vista puede ser óptima.

Con el q-aprendizaje implementado, podéis observar vuestro q-aprendedor bajo control manual, utilizando el teclado:

python gridworld.py -a q -k 5 -m

Recordad que -k controlará el número de episodios que vuestro agente recibe para aprender. Observad como el agente aprende sobre el estado en el que estaba, no sobre el estado al que llega, de forma que va "dejando apredizaje por allá por donde pasa".

***Cuestión 5 (1 punto)***

Completad vuestro agente q-aprendedor implementando la selección de acciones epsilon-voraz action selection en getAction, que selecciona acciones aleatorias con una probabilidad epsilon, y sigue la acción con mejor q-valor en otro caso.

python gridworld.py -a q -k 100

Vuestros q-valores finales deberían parecerse a los del agente de iteración de valores, especialmente a lo largo de caminos muy transitados. A pesar de ello, vuestros retornos medios serán más bajos que los q-valores predichos por la selección aleatoria de acciones en la fase inicial de aprendizaje.

Podéis elegir un elemento de una lista de forma aleatoria utilizando la función random.choice. Podéis simular una variable binaria con probabilidad p de éxito utilizandoutil.flipCoin(p), que retorna True con probabilidad p y False con probabilidad 1-p.

***Cuestión 6 (0.5 puntos) (OPCIONAL)***

Primero, entrenad un q-aprendedor completamente aleatoria con la ratio de aprendizaje por defecto en el BridgeGrid sin ruido durante 50 episodios y observad si encuentra la política óptima o no.

python gridworld.py -a q -k 50 -n 0 -g BridgeGrid -e 1

Ahora probad el mismo experimento con una epsilon de 0. ¿Existen una epsilon y un ratio de aprendizaje para la que sea muy probable (más del 99%) que la política óptima sea aprendida después de 50 iteraciones? Epsilon se controla con -e, el ratio de aprendizaje con -l.

***Cuestión 7 (0.5 puntos) (OPCIONAL)***

Sin código adicional, deberíais ser capaces de ejecutar un robot crawler con q-aprendizaje:

python crawler.py

Si no funciona, probablemente habéis escrito algún código demasiado específico para el problema del GridWorld y deberíais hacerlo más general para cualquier PDM.

Jugad con los diferentes parámetros de aprendizaje para ver como afectan a las acciones y políticas del agente. Daos cuenta de que el retardo es un parámetro de la simulación, mientras que la ratio de aprendizaje y la epsilon son parámetros de vuestro algoritmo de aprendizaje y el factor de descuento es una propiedad del entorno.

**Q-aprendizaje aproximado y abstracción de estados**

***Cuestión 8 (0,5 punto)***

Es el momento de jugar al Pacman!!! Pacman jugará los juegos en dos fases. En la primera fase, *aprendizaje*, Pacman comenzará a aprender sobre los valores de las posiciones y de las acciones. Debido a que lleva mucho tiempo aprender q-valores precisos incluso para rejillas pequeñas, las partidas de aprendizaje de Pacman se ejecutarán en modo silencioso por defecto, sin GUI ni consola. Una vez que el aprendizaje de Pacman se haya completado, el entrará en modo *testing*. En este modo, laself.epsilon y la self.alpha del Pacman se fijarán a 0.0, deteniendo el q-aprendizaje y evitando la exploración de nuevas alternativas, para que Pacman pueda explorar la política que ha aprendido. Los juegos de Test se muestran en la GUI por defecto. Sin ningún cambio deberíais ser capaces de ejecutar q-aprendizaje con Pacman para grids muy pequeños como sigue:

python pacman.py -p PacmanQAgent -x 2000 -n 2010 -l smallGrid

Daos cuenta que el PacmanQAgent ya está definido en función del QLearningAgent que ya habéis escrito. PacmanQAgent solo es diferente en que tiene los parámetros de aprendizaje por defecto que son más efectivos para el problema del Pacman (epsilon=0.05, alpha=0.2, gamma=0.8). Vuestro agente debería ganar el 80% de las últimas 10 partidas.

*Ayuda:* Si vuestro QLearningAgent funciona para [gridworld.py](https://campusvirtual2.ub.edu/pluginfile.php/326412/mod_resource/content/5/docs/gridworld.html) y [crawler.py](https://campusvirtual2.ub.edu/pluginfile.php/326412/mod_resource/content/5/docs/crawler.html) pero no parece aprender una buena política para Pacman en smallGrid, puede ser porque vuestros métodos getAction y/o getPolicy no consideran en algunos casos adecuadamente las acciones no vistas. En particular, porque acciones no vistas tienen por definición un Q-valor de cero, si todas las acciones que *hemos* visto tienen Q-valores negativso, una acción no vista puede ser óptima.

*Nota:* Si queréis experimentar con parámetros de aprendizaje, podéis utilizar la opción -a, por ejemplo -a epsilon=0.1,alpha=0.3,gamma=0.7. Estos valores serán accesibles como self.epsilon, self.gamma y self.alpha dentro del agente.

*Nota:* Aunque se jugarán un total de 2010 partidas, las primeras 2000 no se mostrarán debido a la opción -x 2000, que designa las primeras 2000 partidas para aprendizaje (sin salida). Así que únicamente veréis a Pacman jugar las últimas 10 partidas. El número de juegos de aprendizaje también se puede enviar a vuestro agente mediante la opción numTraining.

*Nota:* Si queréis ver 10 juegos de aprendizaje para ver lo que está sucediendo, utilizad el comando:

python pacman.py -p PacmanQAgent -n 10 -l smallGrid -a numTraining=10

Durante el aprendizaje, veréis un mensaje cada 100 partidas con estadísticas sobre como se comporta Pacman. Epsilon es positivo durante el aprendizaje, así que Pacman jugará mal incluso después de haber aprendido una buena política. Esta es la razón por la que de vez en cuando hace un movimiento exploratorio hacia un fantasma. Como prueba, debería aprender durante 1,000 partidas antes de que las recompensas de Pacman para un segmento de 100 episodios se hagan positivas, reflejando que el ha comenzado a ganar más frecuentemente de lo que pierde. Al final del aprendizaje, debería mantenerse positiva y ser relativamente alta (entre 100 y 350).

Aseguraos de que entendéis lo que está sucediendo: el estado del PDM es *exactamente* la configuración del tablero ante el que Pacman se encuentra, con las transiciones describiendo un paso completo del juego. Las configuraciones intermedias en que Pacman se ha movido pero los fantasmas no lo han hecho aún *no* son estados del PDM.

Si probáis en el [mediumGrid](https://campusvirtual2.ub.edu/pluginfile.php/326412/mod_resource/content/5/layouts/mediumGrid.lay), la cosa puede no funcionar tan bien. En nuestra implementación, las recompensas de Pacman se mantienen negativas a lo largo de todo el aprendizaje. En testing, juega fatal, probablemente perdiendo todas las partidas. Encima, el aprendizaje tarda mucho rato. .

Pacman no gana en escenarios grandes porque cada configuración del tablero es un estado separado con q-valores separados. El no tiene forma de generalizar que tirarse encima de un fantasma es malo para todas las posiciones. Obviamente, esta aproximación no escala.

***Cuestión 9 (1.5 puntos)***

Implementad una q-aprendedor aproximado que aprenda pesos para las características de los estados, donde diversos estados comparten las mismas características. Escribid vuestraa implementación en la clase ApproximateQAgent en [qlearningAgents.py](https://campusvirtual2.ub.edu/pluginfile.php/326412/mod_resource/content/5/docs/qlearningAgents.html), que es una subclase de PacmanQAgent.

*Nota:* El q-aprendizaje aproximado asume la existencia de una función de caracterización de los estados f(s,a) sobre pares (estado, acción) que retorna un vector f1(s,a) .. fi(s,a) .. fn(s,a) de características. Os presentamos algunas funciones de características en [featureExtractors.py](https://campusvirtual2.ub.edu/pluginfile.php/326412/mod_resource/content/5/docs/featureExtractors.html). Los vectores de características son objetos util.Counter(como un diccionario) conteniendo parejas de característica y valor para los que el valor es no nulo; todas las características que no aparecen tienen valor cero.

La q-función aproximada toma la siguiente forma:

donde cada pesot wi se asocia con una característica particular fi(s,a). En vuestro código, deberíais implementar el vectors de pesos como un diccionario que mapea las características (que retornan los extractores de características) a valores de los pesos. Actualizaréis vuestros vectores de pesos de forma similar a como actualizábais vuestros q-valores:

Fijaos que el termino de corrección es el mismo que en el Q-aprendizaje normal.

Por defecto, ApproximateQAgent usa el IdentityExtractor, que asigna una sola característica a cada par (estado, acción). Con este extractor de características, vuestro agente q-aprendedor aproximado debería trabajar de forma idéntica a PacmanQAgent. Podéis probarlos con el siguiente comando:

python pacman.py -p ApproximateQAgent -x 2000 -n 2010 -l smallGrid

*Importante:* ApproximateQAgent es una subclase de QLearningAgent, y comparte algunos métodos como getAction. Aseguraos de que vuestros métodos en QLearningAgentllaman a getQValue en lugar de acceder a los q-valores directamente, de forma que cuando sobrescribáis getQValue en vuestro agente aproximado, los nuevos q-valores aproximados se usen para calcular acciones.

Una vez que tengáis confianza en que vuestro aprendedor aproximado funciona correctamente con las características identidad, ejecutadlo con nuestro propio extractor de características, que puede aprender a ganar con facilidad:

python pacman.py -p ApproximateQAgent -a extractor=SimpleExtractor -x 50 -n 60 -l mediumGrid

Incluso situaciones mucho más complejas no deberían ser problema para vuestro ApproximateQAgent. (*cuidado*: puede tardar unos minutos en aprender)

python pacman.py -p ApproximateQAgent -a extractor=SimpleExtractor -x 50 -n 60 -l mediumClassic

Si no tenéis errores, vuestro q-aprendedor aproximado debería ganar casi cada vez con estas características simples, incluso con solo 50 partidas de aprendizaje.

*¡Felicidades! ¡Ahora tenéis un agente Pacman que aprende!*