PONTIFICIA UNIVERSIDAD CATÓLICA DE CHILE
Facultad de Ingeniería
IIC2613 Inteligencia Artificial
2021-2

Tarea 4

1. Reinforcement Learning

En esta parte ocuparás Reinforcement Learning para entrenar una IA que sea capaz de aprender a jugar al clásico juego Snake por sí misma:

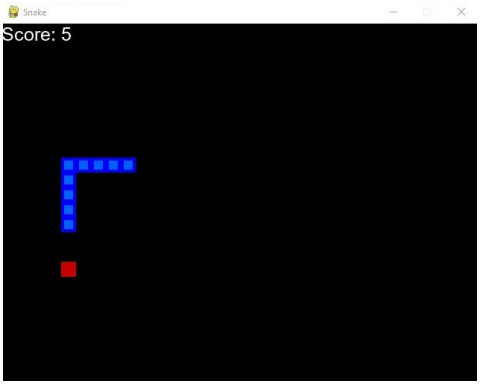

Figura 1: Imagen de la UI del juego Snake

En tu repositorio se te da el código del juego Snake, pero la serpiente se mueve de manera aleatoria. Deberás implementar el algoritmo Q-Learning para que la serpiente pueda moverse de forma "inteligente", comiendo la mayor cantidad de alimento posible en una partida.

Los archivos contenidos en el código base son los siguientes:

- `arial.ttf`: Fuente necesaria para correr el juego
- `game.py`: Código de la interfaz y funcionamiento general del juego. **No modificar**.
- `helper.py`: Código que permite graficar el desempeño del modelo en tiempo real a medida que avanzan las generaciones. **Nota:** El uso de esta funcionalidad ralentiza considerablemente los tiempos de ejecución, así que no se recomienda su uso constante al probar y entrenar el modelo.
- `snake_game_human.py`: Este archivo permite jugar al Snake clásico, controlando los movimientos (solo para divertirse).
- `snake_aleatoria.py`: Este archivo contiene el modelo de un agente que se mueve de forma aleatoria. Es aquí donde deberás trabajar y aplicar tus conocimientos de Reinforcement Learning.

Como ya se mencionó anteriormente, el archivo sobre el cual debes trabajar e implementar Q-Learning es `snake_aleatoria.py`[1]. Este archivo contiene la clase `Agent` con los siguientes métodos:

[1]Este archivo se encuentra muy bien comentado, por lo que se recomienda leerlo como parte del enunciado

- `__init__()`: Este método inicializa los atributos del agente. En el caso del agente aleatorio, solo se inicializan en 0 las partidas jugadas por el agente.
- `get_state(game)`: Este método consulta al juego por el estado actual del agente y lo retorna como una tupla de 11 elementos, donde cada una de las posiciones representa la siguiente información:

  1. **Peligro adelante:** Toma el valor 1 si en la casilla frente a la cabeza de la serpiente hay algún obstáculo que, de colisionar, pierde la partida. Este obstáculo puede ser un borde del mapa o su propio cuerpo. De no haber peligro, toma el valor de 0.
  2. **Peligro a la derecha:** Toma el valor 1 si hay peligro en la casilla a la derecha de la cabeza de la serpiente siguiendo la misma lógica que antes y 0 en caso contrario.
  3. **Peligro a la izquierda:** Toma el valor 1 si hay peligro en la casilla a la izquierda de la cabeza de la serpiente siguiendo la misma lógica que antes y 0 en caso contrario.
  4. **Movimiento a la izquierda:** Toma el valor 1 si la serpiente se está moviendo a la derecha (en la pantalla) y 0 en caso contrario.
  5. **Movimiento a la derecha:** Toma el valor 1 si la serpiente se está moviendo a la derecha (en la pantalla) y 0 en caso contrario.
  6. **Movimiento hacia arriba:** Toma el valor 1 si la serpiente se está moviendo hacia arriba (en la pantalla) y 0 en caso contrario.
  7. **Movimiento hacia abajo:** Toma el valor 1 si la serpiente se está moviendo hacia abajo (en la pantalla) y 0 en caso contrario.
  8. **Comida a la izquierda:** Toma el valor 1 si la comida se encuentra en alguna celda a la izquierda de la cabeza de la serpiente (en la pantalla) y 0 en caso contrario.
  9. **Comida a la derecha:** Toma el valor 1 si la comida se encuentra en alguna celda a la derecha de la cabeza de la serpiente (en la pantalla) y 0 en caso contrario.
  10. **Comida arriba:** Toma el valor 1 si la comida se encuentra en alguna celda arriba de la cabeza de la serpiente (en la pantalla) y 0 en caso contrario.
  11. **Comida abajo:** Toma el valor 1 si la comida se encuentra en alguna celda abajo de la cabeza de la serpiente (en la pantalla) y 0 en caso contrario.

  Nota: En la figura 3, el método `get_state` retornaría: $(0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1)$
- `get_action(state)`: Este método recibe el estado del agente y retorna un entero representando la acción que debe tomar. Las acciones posibles son:

  - `0`: Seguir en la misma misma dirección (en la pantalla).
  - `1`: Girar a la derecha (en la pantalla).
  - `2`: Girar a la izquierda (en la pantalla).

  Nota: En el caso del agente aleatorio, este no utilizará la información que entrega el estado y retornará un número al azar entre 0 y 2, incluyéndolos.

Fuera de esta clase, sólo existe una función `train()`. Esta función es la encargada de entrenar al agente y es la que se llama al correr el archivo.

Actividad 1 (0.4 pts.)

En esta actividad tendrás que pensar el problema antes de comenzar a programarlo, para ello, responde lo siguiente:

- El agente debe tener una Q-Table donde se almacenen los valores (*q-values*) asociados a cada una de las las acciones para cada estado. Sabemos que hay 3 acciones posibles y los estados se componen de tuplas de 11 elementos, donde cada elemento puede tener 2 valores posibles (0 ó 1), por lo que, en teoría, podrían haber 2^{11} estados diferentes. Sin embargo, a pesar de que podría funcionar, no es una buena idea considerar esa cantidad de estados para las filas de la Q-Table. Explique por qué.

  RESPUESTA: No es una buena idea considerar $2^1 1$ estados ya que no todos son posibles. Por ejemplo, la combinación de estados que tenga todos los elementos con el valor 1 dice que se está yendo hacia la derecha y la izquierda al mismo tiempo y eso no es posible. La cantidad de estados posibles son 256. Esto se debe a que existen 2^3 combinaciones para los elementos de peligro ya que puede haber peligro en todas las direcciones y no haber peligro en ningún lado. El movimiento tiene 4 direcciones posibles. La comida tiene 8 combinaciones posibles por los 8 puntos cardinales. Finalmente nos quedan $2^2 * 4 * 8 = 256$ estados.

- ¿Cuál es la función del *exploration rate*? ¿Cuál es el problema de tener un *exploration rate* mínimo con un valor muy bajo y un *exploration decay rate* con un valor muy alto? ¿Qué pasa si el *exploration decay rate* es demasiado bajo?
  RESPUESTA: El *exploration rate* define la probabilidad de que el agente no tome una acción basado en su Q-Table y la elija al azar. El problema de tener un *exploration rate* mínimo muy bajo y un *exploration decay rate* muy alto radica en que el agente podría encontrar una solución subóptima al inicio ya que no tuvo mucho tiempo para explorar y al haber un *eploration rate* mínimo muy bajo es muy dificil que encuentre otras soluciones y no mejore más.

Actividad 2 (2 pts.)

Basándote en lo visto en ayudantías, implementa Q-Learning para el agente del juego Snake (revisa el código base, específicamente los `#TODO`), realizando los siguientes pasos (recuerda comentar todo lo que hagas en tu código):

1. Inicializar la Q-Table del agente.
2. Modificar el método `get_action` para que el agente sea capaz de explorar el estado actual o explotar la mejor acción conocida hasta el momento.
3. Actualizar la Q-Table una vez que se ejecute la acción seleccionada por el agente.
4. En caso de terminar una partida, actualizar el *exploration rate* del agente.

Recomendación: Si tienes dudas con Q-Learning, puedes revisar este artículo.

RESPUESTA: La solución de esta parte se encuentra en `Staff/T4/snake/snake_sol.py`.

Actividad 3 (0.6 pts.)

Una vez hayas programado tu modelo y este sea capaz de aprender, juega con los hiper-parámetros y responde las siguientes preguntas:

- ¿Qué rol cumple la tasa de descuento en Q-Learning?
  RESPUESTA: La tasa de descuento es la tasa por la que descontamos las recompensas futuras y determinará el valor presente de las recompensas futuras. En Q-Learning es un hiperparámetro que descuenta la recompensa obtenida por tomar la mejor acción posible en el siguiente estado.
- Aplicado a este problema, ¿qué tasa de descuento debería entregar mejores resultados? ¿Por qué?
  RESPUESTA: Debido a que las comidas cambian de posición de forma aleatoria, el valor de las recompensas futuras solo distorsionará el verdadero valor de las acciones en los respectivos estados de la Q-Table. Es por esto que la tasa de descuento que entregue mejores valores es 0 o una muy baja (ya que se multiplica por la recompensa futura).
- ¿Para qué sirve el *learning rate*? ¿Qué valor te entregó mejores resultados? Comenta los resultados.

  RESPUESTA: El *learning rate* es un hiperparámetro que define cuánto influye el nuevo valor de tomar una acción en un estado dado al valor previo en la Q-Table. Con *learning rate* 1 se sobreescriben todos los valores anteriores en la Q-Table. Si el *learning rate* es 0 entonces la Q-Table va a quedar en 0 para siempre, tal como se inicializa. Otra forma de verlo es que este hiperparámetro define cuánto importa el nuevo conocimiento por sobre el acumulado. El resto de la pregunta es un comente por lo que quedaría a criterio del ayudante la calidad del comentario del alumno respectoa los resultados obtenidos con diferentes *learning rates*.
- En el juego, y por la forma en la que está programado el ambiente, la serpiente recibe una recompensa negativa por cada paso que da, ¿qué crees que ocurriría si esa recompensa fuera positiva?[2]
  RESPUESTA: Si la recompensa fuera positiva el agente aprendería que la mejor forma de maximizar sus recompensas es no muriendo. Por lo tanto, la serpiente se movería eternamente sin comer nada.

Referencias

[1] Loeber, P. [Python Engineer]. (20 de diciembre del 2020). *Teach AI To Play Snake! Reinforcement Learning* [Archivo de Vídeo]. YouTube. Disponible aquí

[2] *Hint*: Puedes revisar la línea 103 de `game.py` y experimentar