Práctica 2

Laura Rodríguez Navas
rodrigueznavas@posgrado.uimp.es

15 de mayo de 2021

1. Introducción

Esta práctica se apoya en una plataforma que recrea el videojuego clásico Pac-Man. Utiliza una versión muy simplificada del juego para poder desarrollar un sistema de control automático y explorar algunas capacidades del Aprendizaje por Refuerzo aprendidas durante la asignatura. Especialmente, en esta práctica se aplica el algoritmo *Q-learning* para construir un agente que funcione de forma automática en los tres mapas disponibles: *lab1.lay, lab2.lay y lab3.lay*. El objetivo de este agente será maximizar la puntuación obtenida en una partida.

Asimismo, en este documento se describen las tareas realizadas, que se dividen en distintas fases (definición de los estados, función de refuerzo, construcción del agente y evaluación), y que se detallan a continuación.

En la sección 2 se justifica el conjunto de atributos elegido y su rango para la definición de los estados. Una vez se ha elegido el conjunto de atributos que se van a utilizar para representar cada estado, en la sección 3 se detalla el diseño de la función de refuerzo que vamos a emplear y que permitirá al agente lograr su objetivo de maximizar la puntuación obtenida en una partida. Después de esto, en la sección 4 se explica cómo se ha procedido a la construcción del agente con el fin de funcionar bien en todos los laberintos. Cuando se obtiene el agente, en la sección 5 se presentan los resultados de las puntuaciones que ha obtenido en los mapas proporcionados y se añaden comentarios sobre su comportamiento. Finalmente, en la parte final del documento se añaden las conclusiones (sección 6) y un apéndice que contiene métodos auxiliares del código desarrollado en las secciones 3 y 4.

2. Definición de los estados

En esta sección, a fin de definir los estados, hay que seleccionar unos valores adecuados para los parámetros: *self.nRowsQTable, self.alpha, self.gamma y self.epsilon*. Primero intentaremos descubrir el valor del parámetro *self.nRowsQTable*, o que es lo mismo, intentaremos descubrir cuántas filas debe tener la *QTable*. Este valor dependerá de las características seleccionadas para representar los estados, ya que según estas características tendremos un número diferente de filas en la *QTable*. Nos fijamos en la información que nos proporciona la función *printInfo*, concretamente en la información de los mapas *Walls* y *Food*. Según esta información (T/F) y de los movimientos que puede ejecutar el agente: north, east, south y west, las características elegidas son:

- nearest_ghost_north, no_wall
- nearest_ghost_south, no_wall
- nearest_ghost_north, no_food
- nearest_ghost_south, no_food

- nearest_ghost_east, no_wall
- nearest_ghost_west, no_wall
- nearest_ghost_north, wall
- nearest_ghost_south, wall
- nearest_ghost_east, wall
- nearest_ghost_west, wall
- nearest_ghost_east, no_food
- nearest_ghost_west, no_food
- nearest_ghost_north, food
- nearest_ghost_south, food
- nearest_ghost_east, food
- nearest_ghost_west, food

Se han elegido 16 características, de este modo el valor del parámetro *self.nRowsQTable* tendrá que ser igual a 16. Adaptamos el valor de *self.nRowsQTable*:

```
self.nRowsQTable = 16
```

También tenemos que adaptar los parámetros *self.alpha, self.gamma y self.epsilon*. Pero primero realizamos una pequeña descripción de estos:

- Alpha (α). Este valor toma valores entre 0 y 1. Cuando este valor es 0, entonces no hay aprendizaje y siempre se utiliza el valor de $Q(s, a)$ inicial. Cuando este valor es 1, entonces el aprendizaje no conserva ningún porcentaje del valor de $Q(s, a)$. Es decir, si el valor α se acerca a 0, el valor de $Q(s, a)$ no variará mucho; y si el valor α se acerca a 1, el valor de $Q(s, a)$ variará mucho.

- Gamma (γ). Este valor toma valores entre 0 y 1. Cuando este valor es 0, entonces el aprendizaje ignorará el valor dado por el modelo de aprendizaje y aprenderá solo con el valor de la recompensa. Cuando este valor es 1, entonces el aprendizaje utilizará el valor total dado por el modelo de aprendizaje.

- Epsilon (ϵ). Numera los pasos para disminuir el valor α. El valor α se reducirá para cada 10 decisiones tomadas. Por ejemplo, si este valor es 0.01, se reduce 0.01 el valor de α por cada 10 decisiones tomadas.

El proceso para adaptar los parámetros se realiza con el entrenamiento de una inteligencia simple (sin usar el aprendizaje automático). Esta inteligencia simple se compone de un objetivo muy singular, el agente Pac-Man atacando a uno de los fantasmas. Con esa finalidad, ejecutamos el siguiente comando con diferentes valores de *self.alpha, self.gamma y self.epsilon* y observamos el valor resultante *Average Score* en cada ejecución. Este valor nos indica el valor promedio máximo de las puntuaciones obtenidas durante las partidas, o lo que es lo mismo, el valor promedio máximo de $Q(s, a)$ durante las partidas.

```
python busters.py -p RLAgent -k 1 -l lab1.lay -n 100
```

Por ejemplo, con los valores: *alpha = 1* , *gamma = 0.8* y *epsilon = 0.05*, el valor de *Average Score* es menor que con los valores: *self.alpha = 0.2*, *self.gamma = 0.8* y *self.epsilon = 0.05*. Otro ejemplo, con los valores: *self.alpha = 0.4*, *self.gamma = 0.8* y *self.epsilon = 0.05*, el valor de *Average Score* también es menor que con los valores: *self.alpha = 0.2*, *self.gamma = 0.8* y *self.epsilon = 0.05*. Así, consideramos que la mejor asignación de parámetros es con los valores: *self.alpha = 0.2*, *self.gamma = 0.8* y *self.epsilon = 0.05*.

Durante el proceso para adaptar los parámetros, los valores de *epsilon* siempre toman valores muy próximos a 0, esto refleja un comportamiento aprendido en la práctica 1 de la asignatura. En esta práctica aprendimos que si en una estrategia ϵ-greedy el valor de ϵ es más pequeño, entonces la probabilidad de que el agente *Q-learning* tome decisiones aleatorias será menor. Así, si la aleatoriedad es menor, menor será la inestabilidad del agente Pac-Man. Como a consecuencia, el promedio del valor máximo de $Q(s, a)$ será mayor, que corresponde a una mejor puntuación en el videojuego.

Los mejores valores encontrados para los parámetros son los siguientes:

```
self.alpha = 0.2
self.gamma = 0.7
self.epsilon = 0.01
```

3. Función de refuerzo

Una vez que se han adaptado los parámetros del agente Pac-Man podemos diseñar la función de refuerzo. La función de refuerzo determinará las recompensas que obtenga el agente. Determinar las recompensas que obtiene el agente con cada acción es clave para su aprendizaje, pues marca cómo de buenas serán unas acciones frente a otras y por lo tanto determina qué tipo de comportamientos decidimos potenciar a la larga. La lista siguiente enumera los diferentes estados y sus acciones con sus valores de recompensa elegidos:

- Ganar el juego son 1000 puntos.
-

```
def getReward(self, state, nextState):
  """
  Return a reward value based on the information of state and nextState
  """
  reward = 0

  if nextState.isWin():
    return 1000

  # distancia al fantasma en el siguiente estado
  next_state_ghost_position = self.getGhostDistance(nextState)
  # distancia minima al fantasma en el siguiente estado
  min_position_ghost_next_state = min(next_state_ghost_position,
    key=lambda t: t[1])[0]
  # fantasma en el estado actual
  nghost_actual_state = len(list(filter(lambda d: d is not None,
    state.data.ghostDistances)))
  # fantasma en el estado siguiente
  nghost_next_state = len(list(filter(lambda d: d is not None,
    nextState.data.ghostDistances)))
  # paredes en el estado actual
  actual_state_has_walls = self.directionIsBlocked(state,
    state.getGhostPositions()[min_position_ghost_next_state])
  # paredes en el estado siguiente
  next_state_has_walls = self.directionIsBlocked(nextState,
    nextState.getGhostPositions()[min_position_ghost_next_state])

  # no fantasma en el siguiente estado
  if nghost_next_state < nghost_actual_state:
    reward += 100

  # fantasma en el siguiente estado y no paredes
  if not actual_state_has_walls and nghost_next_state == nghost_actual_state:
    reward += 3
```

```
  # fantasma en el siguiente y paredes
  elif actual_state_has_walls and nghost_next_state == nghost_actual_state:
    reward += -1

  # fantasma - paredes y en el estado siguiente hay paredes
  if not actual_state_has_walls and next_state_has_walls \
      and nghost_next_state == nghost_actual_state:
    reward -= 4
  # fantasma + paredes y en el estado siguiente no hay paredes
      elif actual_state_has_walls and not next_state_has_walls \
    and nghost_next_state == nghost_actual_state:
  reward += 1

  return reward
```

Los métodos auxiliares se describen en la sección 7.1.

4. Código desarrollado

```
def update(self, state, action, nextState, reward):
  """
  The parent class calls this to observe a
  state = action => nextState and reward transition.
  You should do your Q-Value update here
  """
  reward = reward + self.getReward(state, nextState)  # Calculate reward

  print "Started in state:"
  self.printInfo(state)
  print "Took action: ", action
  print "Ended in state:"
  self.printInfo(nextState)
  print "Got reward: ", reward
  print "-------------------------------------"

  position = self.computePosition(state)
  action_column = self.actions[action] - 1
  sample = reward + self.gamma * self.getValue(nextState)

  if nextState.isWin():   # If a terminal state is reached
    self.writeQtable()

  else:
    self.q_table[position][action_column] = (1 - self.alpha) * self.q_table[position
    ][action_column] + self.alpha * sample
```

Este proyecto se divide en varias tareas pequeñas que deben completarse en consecuencia. La primera tarea gira en torno al diseño de Value-Iteration Agent. Este agente puede planificar su acción dado el conocimiento del entorno antes incluso de interactuar con él. Para ello, debemos calcular los valores Q de las acciones que seguirá el agente y elegir la mejor acción que devolverá el valor Q máximo. Después de diseñar el agente de iteración de valor, necesitamos cambiar el valor de descuento, ruido y vida. recompensa para lograr la mejor política óptima.

La siguiente tarea es escribir código para Q learning agent. Este agente, a diferencia del agente de iteración de valor, necesita interactuar activamente con el entorno para que pueda aprender a través de las experiencias. Necesitamos aproximar la función Q de los pares estado-acción a partir

de las muestras de Q (s, a) que observamos durante la interacción con el medio ambiente. Después de una aproximación exitosa, el agente debería poder tomar la mejor acción para lograr el objetivo. Q-learning Pacman debería poder ganar al menos el 80 % del tiempo.

Sin embargo, a medida que intentamos utilizar este agente en diseños más grandes, por ejemplo, el mundo Grid medio El entrenamiento de PacMan de alguna manera ya no funcionará bien. Para resolver este problema, debemos optimizar nuestro agente para implementar también un agente Q-learning aproximado que aprenda los pesos de características de los estados, donde muchos estados pueden compartir las mismas características.

5. Resultados

6. Conclusiones

Todo el contenido de esta práctica se puede encontrar en el repositorio personal de GitHub: `https://github.com/lrodrin/masterAI/tree/master/A21/softpractica2`.

7. Apéndice

7.1. Métodos de la función *get_reward*

7.2. Métodos de la función *update*