Práctica 2

Laura Rodríguez Navas
rodrigueznavas@posgrado.uimp.es

22 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 Pac-Man que funcione de forma automática, con el objetivo de maximizar la puntuación obtenida en una partida por cada mapa disponible para la práctica: *lab1.lay, lab2.lay y lab3.lay*.

Asimismo, se presenta este documento donde se describen las tareas de la práctica realizadas, que se dividen en distintas fases (definición de los estados, función de refuerzo, construcción del agente Pac-Man 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 Pac-Man 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: *nRowsQTable, alpha, gamma y epsilon*. Primero intentaremos descubrir el valor del parámetro *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 nos modificará el número de filas de nuestra *QTable*. Nos fijamos en la información que nos proporciona la función *printInfo* durante una ejecución manual del agente, concretamente nos fijamos en la información de los mapas *Walls* y *Food*. Según esta información, que nos indica la presencia y ausencia de paredes y comida en los mapas *Walls* y *Food*, y la información de las direcciones que puede ejecutar el agente Pac-Man: *north, east, south y west*, elegimos 16 características.

Las 16 características elegidas son:

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

Según el número de características (16), modificamos el valor del parámetro *nRowsQTable*, de la función *registerInitialState* en la clase *RLAgent* del archivo *bustersAgents.py*, de la siguiente manera:

```
self.nRowsQTable = 16
```

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

- 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 Q inicial. Cuando este valor es 1, entonces el aprendizaje no conserva ningún porcentaje del valor Q. Es decir, si el valor α se acerca a 0, el valor Q no variará mucho, y si el valor α se acerca a 1, el valor Q 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 n decisiones tomadas. Por ejemplo, si este valor es 0.01, se reduciría 0.01 el valor de α por cada 10 decisiones tomadas, donde n=10.

El proceso para elegir los parámetros *nRowsQTable, alpha, gamma y epsilon* se ha realizado 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 en el primer laberinto disponible (*lab1.lay*). Con esa finalidad, ejecutamos el comando *python busters.py -p RLAgent -k 1 -l lab1.lay -n 100* varias veces, con diferentes valores de *alpha, gamma y epsilon* observando el valor resultante *Average Score* de las distintas ejecuciones (ver Figura 1). El valor *Average Score* nos indicará el valor promedio máximo de las puntuaciones obtenidas durante 100 partidas, o lo que es lo mismo, el valor promedio máximo Q durante 100 partidas. Si este valor es positivo habremos ganado la partida, en caso contrario, habremos perdido la partida.

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

Durante el proceso para elegir los parámetros, los valores de *epsilon* siempre han tomado 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. Si la aleatoriedad es menor, menor será la inestabilidad del agente *Q-learning*, en este caso la del agente Pac-Man. Como a consecuencia, el promedio del valor máximo Q será mayor, que es equivalente a una mejor puntuación en el videojuego alcanzando el objetivo del agente.

Los mejores valores encontrados para los parámetros son los siguientes:

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

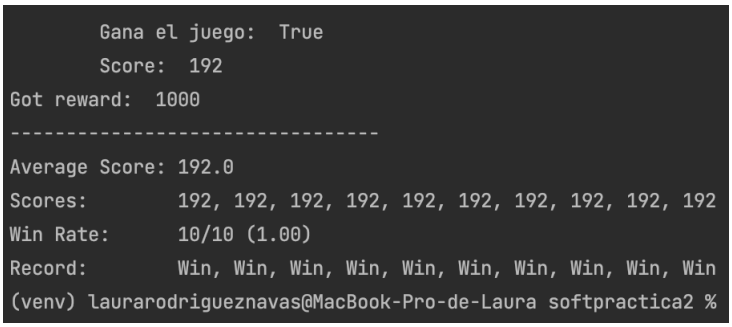

Figura 1: Ejemplo de ejecución del agente Pac-Man que nos muestra el *Average Score* obtenido.

3. Función de refuerzo

Una vez que se han adaptado los parámetros del agente Pac-Man ya 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 marcará cómo de buenas serán unas acciones frente a otras y por lo tanto determinará qué tipo de comportamientos decidimos potenciar. La siguiente lista enumera las diferentes acciones que he decidido potenciar, con sus valores de recompensa correspondientes:

- **Ganar**: este estado se muestra cuando el agente Pac-Man gana la partida. Ganamos 1000 puntos de recompensa.
- **Comer**: este estado se muestra cuando el agente Pac-Man se come a un fantasma. Ganamos 100 puntos de recompensa.
- **Lejos de un fantasma y lejos de una pared**: este estado se muestra cuando el agente Pac-Man está al menos a cinco celdas del fantasma más cercano y de la pared más cercana. En este caso, ganamos 1 punto de recompensa.
- **Lejos de un fantasma y cerca de una pared**: este estado se muestra cuando el agente Pac-Man está al menos a cinco celdas del fantasma más cercano y está a un máximo de cuatro celdas de la pared más cercana. En este caso, perdemos 1 punto de recompensa.

- **Cerca de un fantasma y lejos de una pared**: este estado se muestra cuando el agente Pac-Man está a un máximo de cuatro celdas del fantasma más cercano y está al menos a cinco celdas de la pared más cercana. En este caso, ganamos 3 puntos de recompensa.
- **Cerca de un fantasma y cerca de una pared**: este estado se muestra cuando el agente Pac-Man está a un máximo de cuatro celdas del fantasma más cercano y de la pared más cercana. En este caso, ganamos 2 puntos de recompensa.
- **Cerca de una pared**: este estado se muestra cuando el agente Pac-Man está a un máximo de cuatro celdas de la pared más cercana. En este caso, perdemos 3 puntos de recompensa.
- **Lejos de una pared**: este estado se muestra cuando el agente Pac-Man está al menos a cinco celdas de la pared más cercana. En este caso, ganamos 1 punto de recompensa.

En todas las acciones, el agente Pac-Man recuerda la posición del fantasma más cercano y elige una de las direcciones disponibles: *north, east, south o west* para llegar al fantasma por el camino más corto. Como se puede observar, se han definido las recompensas mediante valores enteros, aunque las acciones dependan de la posición de los fantasmas respecto a la posición del agente. Se ha tomado esta decisión al fin de facilitar el código desarrollado y se han definido los valores de recompensa para cada acción en particular. La elección de los valores concretos de recompensa se han realizado un poco al azar, siguiendo mi propio criterio. La función de refuerzo desarrollada se muestra a continuación:

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

  if nextState.isWin():
    return 1000

  # distancias al fantasma mas cercano en el siguiente estado
  next_state_ghost_distances = self.getGhostDistances(nextState)
  # distancias al fantasma mas cercano en el estado actual
  actual_state_ghost_distances = self.getGhostDistances(nextState)

  # distancia minima al fantasma mas cercano en el siguiente estado
  min_distance_ghost_next_State = min(next_state_ghost_distances, key=lambda t: t[1])[0]
  min_ghost_distance_next_state = nextState.data.ghostDistances[
    min_distance_ghost_next_State]
  # distancia al fantasma mas cercano en el estado actual
  min_distance_ghost_actual_State = min(actual_state_ghost_distances, key=lambda t: t[1])[0]
  min_ghost_distances_actual_state = state.data.ghostDistances[
    min_distance_ghost_actual_State]

  # numero de fantasmas en el estado actual
  number_ghost_actual_state = len(list(filter(lambda d: d is not None,
    state.data.ghostDistances)))
  # numero de fantasmas en el siguiente estado
  number_ghost_next_state = len(list(filter(lambda d: d is not None,
    nextState.data.ghostDistances)))

  # distancia a la pared mas cercana en el estado actual
  actual_state_has_walls = self.directionIsBlocked(state,
  state.getGhostPositions()[min_distance_ghost_next_State])
  # distancia a la pared mas cercana en el siguiente estado
  next_state_has_walls = self.directionIsBlocked(nextState,
  nextState.getGhostPositions()[min_distance_ghost_next_State])
```

```
# come fantasma
if number_ghost_next_state < number_ghost_actual_state:
  reward += 100

# mas lejos de un fantasma y lejos de una pared, no come
if min_ghost_distance_next_state < min_ghost_distances_actual_state \
    and not actual_state_has_walls \
      and number_ghost_next_state == number_ghost_actual_state:
  reward += 1

# mas lejos de un fantasma y cerca de una pared, no come
elif min_ghost_distance_next_state < min_ghost_distances_actual_state \
    and actual_state_has_walls \
      and number_ghost_next_state == number_ghost_actual_state:
  reward -= 1

# mas cerca de un fantasma y lejos de una pared, no come
elif min_ghost_distance_next_state > min_ghost_distances_actual_state \
    and not actual_state_has_walls \
      and number_ghost_next_state == number_ghost_actual_state:
  reward += 3

# mas cerca de un fantasma y cerca de una pared, no come
elif min_ghost_distance_next_state > min_ghost_distances_actual_state \
  and actual_state_has_walls \
    and number_ghost_next_state == number_ghost_actual_state:
  reward += 2

# cerca de una pared, no come
if not actual_state_has_walls and next_state_has_walls \
    and number_ghost_next_state == number_ghost_actual_state:
  reward -= 3

# lejos de una pared, no come
elif actual_state_has_walls and not next_state_has_walls \
    and number_ghost_next_state == number_ghost_actual_state:
  reward += 1

return reward
```

Código 1: Función de refuerzo.

Los métodos auxiliares de la función de refuerzo como el método *getGhostDistances*, que muestra las distancias entre los fantasmas y el agente Pac-Man (ver Código 3), y el método *directionIsBlocked*, que nos indica si la dirección tomada por el agente se bloquea por una pared (ver Código 4), se incluyen en la sección 7.1.

4. Código desarrollado

Una vez que hemos seleccionado los parámetros que vamos a utilizar para representar cada estado, y hemos desarrollado la función de refuerzo que vamos a emplear (ver Código 1), procedemos a la construcción del agente Pac-Man. La implementación del agente consiste en ir determinando el estado actual en que se encuentra. Dado este estado, se elige la acción de acuerdo con los valores Q. Una vez elegida la acción, se determina cuál es la dirección más apropiada para la acción dada. A la hora de elegir la siguiente acción, se actualiza el valor Q para el estado y la acción anterior, dado el estado actual obtenido y la mejor acción que devuelve la fórmula expresada en la ecuación 1, que podemos observar en la página siguiente. Finalmente dado el estado actual, el proceso comenzará de nuevo eligiendo la siguiente acción. Este procedimiento se recoge en la función *update* (ver

Código 2), que lleva a cabo el algoritmo *Q-learning*.

Como se ha visto ya en la práctica 1 de la asignatura, los valores de la tabla Q únicamente son actualizados cuando se cambia de estado, es decir, si al tomar una acción, el agente se mantiene en el mismo estado, la tabla Q no se actualiza, sino que la recompensa se acumula, de forma que se suman todas las recompensas obtenidas mientras se permanece en un estado, y es cuando se pasa al siguiente, el momento de modificar la tabla Q.

La fórmula para obtener el nuevo valor Q:

$$Q_{(t+1)}(s_t, a_t) = (1 - \alpha) * Q_t(s_t, a_t) + \alpha * (R(s_t, a_t)) + V_t(s_{t+1}) - Q_t(s_t, a_t) \quad (1)$$

donde,

$Q_{(t+1)}(s_t, a_t)$	es el nuevo valor Q dado el estado anterior s_t y la acción anterior a_t.
α	es la tasa de aprendizaje.
$R(s_t, a_t)$	es el valor de recompensa dado el estado anterior s_t y la acción anterior a_t.
$Q_t(s_t, a_t)$	es el antiguo valor de Q dado el estado anterior s_t y la acción anterior a_t.
$V_t(s_{t+1})$	es el valor de aprendizaje dado el nuevo estado s_{t+1}.

La función *update* que lleva a cabo el algoritmo *Q-learning*:

```
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)  # actualizar recompensa

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

  # buscar el estado actual en la memoria de estados
  state_position = self.computePosition(state)
  action_position = self.actions[action]  # elegir accion
  # actualizar la tabla Q con la accion elegida
  self.q_table[state_position][action_position] = (1 - self.alpha) *
    self.q_table[state_position][action_position] + self.alpha * (reward +
      self.gamma * self.getValue(nextState))

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

Código 2: Función update.

En la función *update*, primero actualizamos el valor de la recompensa. Esto se debe a que en la función *getReward* (ver Código 1), el valor de la variable *reward* inicialmente siempre es 0. Después buscamos el estado actual en la memoria de estados con la función *computePosition* (ver Código 5), método que se incluye en la sección 7.2. El agente Pac-Man aprende modificando los valores Q (valores que hay almacenados en la tabla Q), de forma que cuando se realiza la acción *action_position* desde el estado *state_position*, es el valor $Q(s, a)$ el que se ve modificado de acuerdo a la fórmula expresada en la ecuación 1. Con la finalidad de que el agente pueda aprender, necesita tener a mano todos los estados que ya haya visitado anteriormente, para poder detectar en cuál de ellos se encuentra a medida que va jugando y, en caso de encontrarse en una situación nueva, registrarla para utilizarla posteriormente. En la implementación de la función *statesMemory*,

método que se incluye en la sección 7.2, estos estados se guardan en forma de lista a medida que se van descubriendo (ver Código 6).

Una vez modificado el agente, observamos su comportamiento en uno de los laberintos disponibles de la práctica, ejecutándolo con el siguiente comando:

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

Con ello, hemos ejecutado durante 100 partidas el agente que acabamos de crear en el laberinto *lab1*, que únicamente tiene un fantasma (ver Figura). Lo volvemos a ejecutar, pero esta vez en el laberinto *lab2* que tiene dos fantasmas, con el uso del siguiente comando:

```
python busters.py -p RLAgent -k 2 -l lab2.lay -n 100
```

Finalmente, también lo ejecutamos en el último de los laberintos disponibles de la práctica, el laberinto *lab3*, que tiene 3 fantasmas, pared en el interior del laberinto y un punto.

```
python busters.py -p RLAgent -k 3 -l lab3.lay -n 100
```

El agente Pac-Man toma decisiones cada vez que llega a una intersección, momento en que mira el estado del juego, ve en qué situación se encuentra, y en base a los conocimientos aprendidos toma la decisión de moverse en una dirección u otra. También, en aquellos momentos en que se le proporciona información acerca de los fantasmas, el agente también toma decisiones en el momento en que percibe a un fantasma demasiado cerca de él, momento en el cual decide atacar, aunque no se encuentre en una intersección. El agente aprende a perseguir a los fantasmas, que son los que le reportan mayor beneficio. Elige caminos que contienen la mayor cantidad de fantasmas, desviándose a veces si ve que tiene un fantasma comestible cerca.

5. Resultados

En esta sección exponemos los resultados obtenidos en las ejecuciones realizadas en la sección anterior (ver sección 4), además sacamos algunas conclusiones con respecto a los resultados. Los resultados obtenidos son:

Mapa	Average Score
1	191.89
2	390.96
3	578

Tabla 1: Resultados de las ejecuciones del agente Pac-Man.

Podemos ver que la técnica de *Q-learning* implementada en el agente Pac-man es lo suficientemente inteligente para vencer en los tres mapas.

6. Conclusiones

Q-learning en estos casos, es sin duda un método que parece efectivo. Sin embargo, requiere realizar muchos experimentos para estudiar la viabilidad de las soluciones propuestas. En concreto lo que más problemas me ha dado ha sido la tarea de diseñar los casos, y de cómo utilizar la información que contienen para hacer que el sistema evolucione a medida que se simulan las partidas. Conseguir un equilibrio para que los casos no sean demasiado específicos (y por tanto se reutilicen

pocas veces), ni tampoco demasiado generales (y no aporten soluciones especialmente útiles), es complicado y requiere de muchas pruebas hasta encontrar los parámetros que mejor se adaptasen al objetivo perseguido.

Personalmente uno de los mayores retos de esta práctica ha sido la implementación de todo el código para operar con el agente Pac-Man. El videojuego que se utiliza parece a priori sencillo, pero representa un dominio que se puede volver extremadamente complejo cuando pretendes que un agente aprenda a jugar. Un jugador humano procesa información visual y la aplica muy rápido cuando toma decisiones, pero para conseguir que una máquina haga lo mismo, se necesita un grado de abstracción de esa información que en ocasiones se ha vuelto muy difícil de conseguir. Ha sido de gran ayuda que muchas personas hayan resuelto este problema de diferentes maneras, ya me ha dado la posibilidad de aprender y tener una implementación base con la que guiarme.

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*

```
def getGhostDistances(gameState):
  """
  Return distances to each of the ghosts on the map.
  """
  return [(i, distance) for i, (distance, alive) in enumerate(zip(
    gameState.data.ghostDistances, gameState.getLivingGhosts()[1:])) if alive]
```

Código 3: Función getGhostDistances.

```
def directionIsBlocked(gameState, ghost_position):
  """
  Return True if directions are blocked (walls) and False otherwise (no walls).
  It also returns distances to blocking elements and non-blocking elements.
  """
  walls = gameState.getWalls()
  walls_array = np.array(walls.data)
  pacman_position = gameState.getPacmanPosition()

  x_min = min(pacman_position[0], ghost_position[0])
  x_max = max(pacman_position[0], ghost_position[0]) + 1
  y_min = min(pacman_position[1], ghost_position[1])
  y_max = max(pacman_position[1], ghost_position[1]) + 1
  if y_min < 3:
    y_min = 3

  grid_beetween = walls_array[x_min:x_max, y_min:y_max]
  if len(grid_beetween) == 0:
    return False

  return np.any(np.all(grid_beetween, axis=1)) or np.any(np.all(grid_beetween, axis
    =0))
```

Código 4: Función directionIsBlocked.

7.2. Métodos de la función *update*

```
def computePosition(self, state):
  """
  Compute the row of the qtable for a given state.
  """
  pacman_ghost_direction, ghost_position = self.statesMemory(state)
  hasWall = self.directionIsBlocked(state, ghost_position)
  actions_value = 0
  for i, direction in enumerate(pacman_ghost_direction):
    actions_value += min(self.actions[direction], 2) + i * 4
  return int(hasWall) * 8 + actions_value
```

Código 5: Función computePosition.

```
def statesMemory(self, gameState):
  """
  Create states memory.
  """
  # posicion de pacman
  pacman_position = gameState.getPacmanPosition()

  # distancia minima al fantasma mas cercano
  living_ghosts_distances = self.getGhostDistances(gameState)
  min_distance_ghost_index = min(living_ghosts_distances, key=lambda t: t[1])[0]

  # posicion del fantasma mas cercano
  nearest_ghost_position = gameState.getGhostPositions()[min_distance_ghost_index]

  pacman_ghost_direction = []

  if (pacman_position[1] - nearest_ghost_position[1]) != 0 \
      and (pacman_position[1] - nearest_ghost_position[1]) > 0:
    pacman_ghost_direction.append("South")
  if (pacman_position[1] - nearest_ghost_position[1]) != 0 \
      and (pacman_position[1] - nearest_ghost_position[1]) < 0:
    pacman_ghost_direction.append("North")
  if (pacman_position[0] - nearest_ghost_position[0]) != 0 \
      and (pacman_position[0] - nearest_ghost_position[0]) > 0:
    pacman_ghost_direction.append("West")
  if (pacman_position[0] - nearest_ghost_position[0]) != 0 \
      and (pacman_position[0] - nearest_ghost_position[0]) < 0:
    pacman_ghost_direction.append("East")

  return pacman_ghost_direction, nearest_ghost_position
```

Código 6: Función statesMemory.