Universidad Autónoma de Nuevo León

Facultad de Ingeniería Mecánica y Eléctrica

REDES NEURONALES ARTIFICIALES
AGO-DIC 21

4.1 – algoritmo de Foundations of deep reinforcement learning

Nombre:	**Matricula:**	**Carrera:**
Luis Daniel García Leal	1857391	ITS

Nombre del profesor:

JOSE ARTURO BERRONES SANTOS

Semestre agosto – diciembre 2021

Días de la clase y Hora: Jueves (V4-V6)

San Nicolás de los Garza, N.L. Fecha: 24/11/2021

Objetivo:

El ejercicio consiste en:

1) Explicar cómo se relacionan las funciones de ambos y cómo se logra en sí la implementación de las funciones en pseudo-código del Algoritmo 1.1 en el programa ql2.py
2) Permitir al agente explorar durante 10 pasos cada episodio. Explicar la curva de aprendizaje resultante.

Reinforcement learning

El aprendizaje por refuerzo (RL) se ocupa de resolver la problemática de la toma de decisiones secuencial. Existen muchos problemas del mundo real que aplican con ello como lo son: jugar videojuegos, deportes, conducir, optimizar inventario, control robótico etc. Estas son cosas que los humanos y las máquinas pueden hacer.

El aprendizaje por refuerzo estudia problemas de forma y métodos artificiales por los cuales los agentes aprenden a resolverlos. Es un subcampo de la inteligencia artificial que se remonta a la teoría de control óptimo y procesos de decisión de Markov (MDP). Fue trabajado por primera vez por Richard Bellman en la década de 1950 en el contexto de la programación dinámica y cuasilineal.

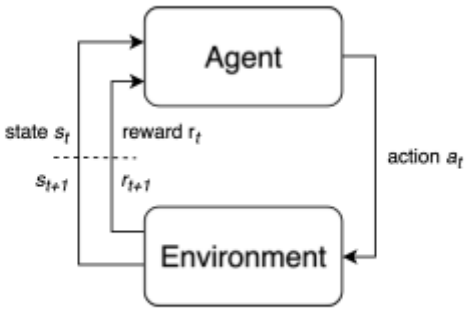


Figure 1.2 The reinforcement learning control loop

Esencialmente, un sistema de aprendizaje por refuerzo es un bucle de control de retroalimentación donde un agente y un entorno interactúan e intercambian señales, mientras que el agente intenta maximizar el objetivo. Las señales intercambiadas son (st, at, rt), que representan estado, acción y recompensa, respectivamente, y t denota el paso de tiempo en el que ocurrieron estas señales.

QL2 Pseudo-codigo

Algorithm 1.1 MDP control loop

```
1: Given an env (environment) and an agent:
2: for episode = 0, ..., MAX_EPISODE do
3:     state = env.reset()
4:     agent.reset()
5:     for t = 0, ..., T do
6:         action = agent.act(state)
7:         state, reward = env.step(action)
8:         agent.update(action, state, reward)
9:         if env.done() then
10:            break
11:        end if
12:    end for
13: end for
```

El algoritmo 1.1 expresa las interacciones entre un agente y un entorno sobre muchos episodios y pasos de tiempo. Al comienzo de cada episodio, el entorno y el agente se restablece (líneas 3–4).

```
3:     state = env.reset()
4:     agent.reset()
```

Al reiniciar, el entorno produce un estado inicial. Luego al comenzar a interactuar: un agente produce una acción dado un estado (línea 6) y, a continuación, el entorno produce el siguiente estado y recompensa dada la acción (línea 7), entrando al siguiente paso.

```
6:         action = agent.act(state)
7:         state, reward = env.step(action)
```

El ciclo agent.act-env.step continúa hasta el paso de tiempo máximo T se alcanza o el entorno termina. Aquí también vemos un nuevo componente, agent.update (línea 8), que encapsula el algoritmo de aprendizaje de un agente. Sobre varios pasos de tiempo y episodios, este método recopila datos y realiza el aprendizaje internamente para maximizar el objetivo.

```
8:         agent.update(action, state, reward)
```

Este algoritmo es genérico para todos los problemas de aprendizaje por refuerzo, ya que define una interfaz coherente entre un agente y un entorno. La interfaz sirve como un base para implementar muchos algoritmos de aprendizaje por refuerzo bajo un sistema unificado por un framework.

QL2 Código

Librerías utilizadas:

Numpy: NumPy es una librería de Python que da soporte al usuario y le permite crear vectores y matrices grandes multidimensionales, junto con una gran colección de funciones matemáticas de alto nivel para operar con ellas.

Matplotlib: Es una librería enfocada a la generación de gráficos a partir de datos contenidos en listas o arrays en el lenguaje de programación Python y su extensión matemática NumPy.

Copy: Las instrucciones de asignación en Python no copian objetos, crean enlaces entre un destino y un objeto. Para las colecciones que son mutables o contienen elementos mutables, a veces se necesita una copia para que uno pueda cambiar una copia sin cambiar la otra y para eso utilizaremos la herramienta copy en este programa.

Modulo principal

```
179  ################################################################################
180  ### Main Program ###
181
182  MAX_EPISODE = 500
183  MAX_TIME = 3
184  agent = Agent("James Bond")
185  env = Environment("in a dangerous environment")
186  print(agent.name)
187  print(env.name)
188  l = [None] * MAX_EPISODE #list to draw the learning curve
189  #action = agent.action_reset()
190  for episode in range(MAX_EPISODE):
191      state = env.reset()
192      action = agent.action_reset() #depending on the learning problem
193      cur = 0.0 #cumulative reward for visualization purposes
194      for t in range(MAX_TIME):
195          state0 = copy.copy(state)
196          action0 = copy.copy(action)
197          state, reward = env.step(state0, action0)
198          agent.update(state0, action0, reward, state, action)
199          action = agent.act(state)
200          #print(action)
201          #print(state, reward)
202          cur = cur + reward
203          if env.done(state) == True:
204              break
205      #print(cur)
206      #print(Q.get((3,1)))
207      l[episode] = cur
208      #l[episode] = Q.get((-5,0))
209
210  #print(Q.items())
211
212  import matplotlib.pyplot as plt
213
214  fig, ax = plt.subplots()
215  ax.plot(l)
216  plt.show()
```

Como se mencionó anteriormente este algoritmo expresa las interacciones entre un agente y un entorno sobre muchos episodios y pasos de tiempo. En este caso mediante el intervalo de cada episodio, el entorno y el agente se restablecen mediante las líneas 191 y 192 después de haber declarado sus respectivos módulos anteriormente.

Después el entorno produce un estado inicial. Luego al comenzar a interactuar: un agente produce la acción en la línea número 196 y después el entorno produce el siguiente estado y recompensa dada la acción 197, entrando al siguiente paso. En esta parte del código hacemos el uso del modulo copy para tener a la mano los estados y acciones actuales, también utilizamos dichas herramientas en la clase de agente y enviroment que son clases fundamentales generales para implementar un algoritmo Q-learning.

```
### These are the generic fundamental classes for Q-learning ###################
class Agent:

    def __init__(self, name):
        self.name = name

    def action_reset(self):
        action = 0
        return action

    def act(self, state):
        z = Policy(self)
        s = copy.copy(state)
        #a = z.maxq(s, Actions)
        a = z.epsilon_greedy(s, Actions)
        #print(a)
        return a

    def update(self, state0, action0, reward, state, action):
        s0 = copy.copy(state0)
        a0 = copy.copy(action0)
        r1 = copy.copy(reward)
        s1 = copy.copy(state)
        a1 = copy.copy(action)
        z = Policy(self)
        z.updateq(s0,a0,r1,s1,a1)
        return
```

```
class Environment:

    def __init__(self, name):
        self.name = name

    def reset(self):
          state = 0
          return state

    def step(self, state, action):
        s = copy.copy(state)
        a = copy.copy(action)
        z = Envsim(self)
        s, r = z.Enviro(s,a)
        return s, r

    def done(self, state):
        if state == -10 or state == 10:
           vdone = True
        else:
            vdone = False
        return vdone
```

El ciclo agent.act-env.step continúa hasta el paso de tiempo máximo T se alcanza o el entorno termina y utilizamos la variable agent.update en la línea 198 que encapsula el algoritmo de aprendizaje de un agente. Sobre varios pasos de tiempo y episodios, este método recopila datos y realiza el aprendizaje internamente para maximizar el objetivo.

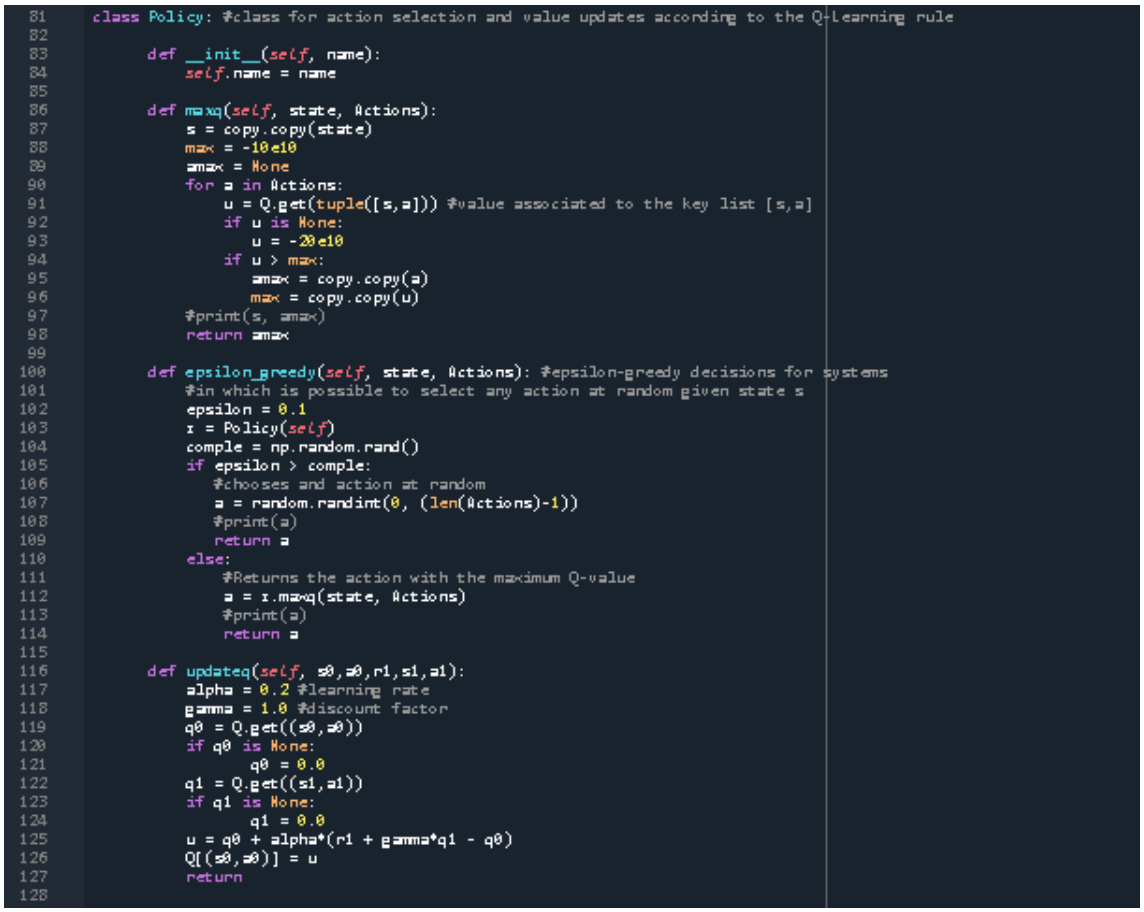
Parámetros

```
Q = {} #dictionary for the Q function [(state, action), value]

b = 0.95 #bias parameter (particular to the example)

Actions = [0, 1, 2] #list of possible agent's actions (particular to the example)

Q[(0,0)] = 0.0 ### some initializations particular to the example
amax0 = 0 ### some initializations particular to the example
```

Módulos utilizados para poder implementar los métodos de aprendizaje

```
class Envsim: # the necessary methods and data structures for the simulation of the environment

      def __init__(self, name):
        self.name = name

      def rand0(self, state):
          s = copy.copy(state)
          s = s + 2.0*(random.randint(0, 1) - 0.5)
          #print(s, round(s))
          return round(s)

      def rand1(self, state):
          s = copy.copy(state)
          u = random.uniform(0,1)
          #print(s)
          if u < b:
             s = s + 1
          else:
             s = s - 1
          #print(u,s)
          return s

      def rand2(self, state):
          s = copy.copy(state)
          u = random.uniform(0,1)
          #print(s)
          if u < b:
             s = s - 1
          else:
             s = s + 1
          #print(u,s)
          return s

```

```
      def Enviro(self, state, action):
          # Given a (state_t, action_t) pair, it generates (s_t+1, r_t+1)
          a = copy.copy(action)
          s = copy.copy(state)
          z = Envsim(self)
          r = 0.0
          ra = -1.0
          rb = 1.0
          rc = 0.0
          if a==0:
             s = z.rand0(state)
          if a==1:
             s = z.rand1(state)
          if a==2:
             s = z.rand2(state)
          else:
             s = s

          if -10 < s < 0:
             r = ra
          if 0 < s < 10:
             r = rb
          if s == 0:
             r = rc

          return s, r

```

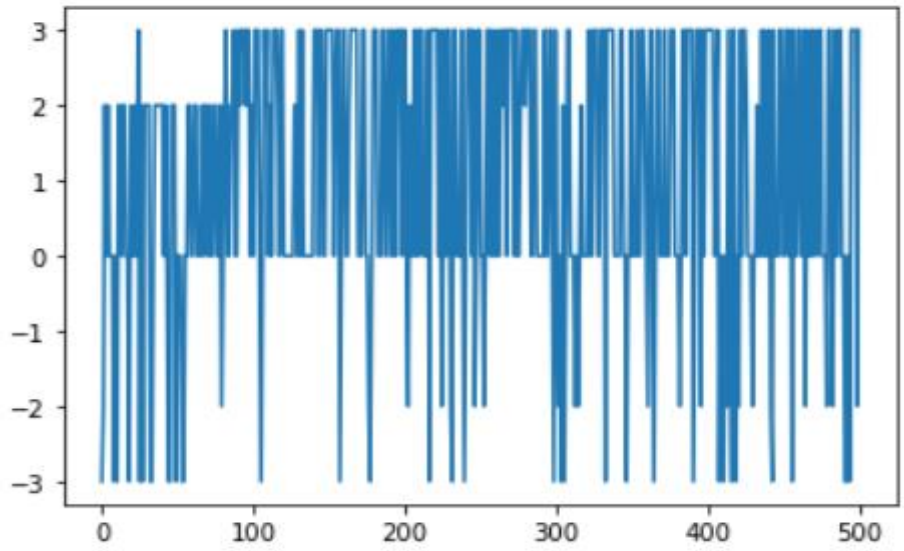


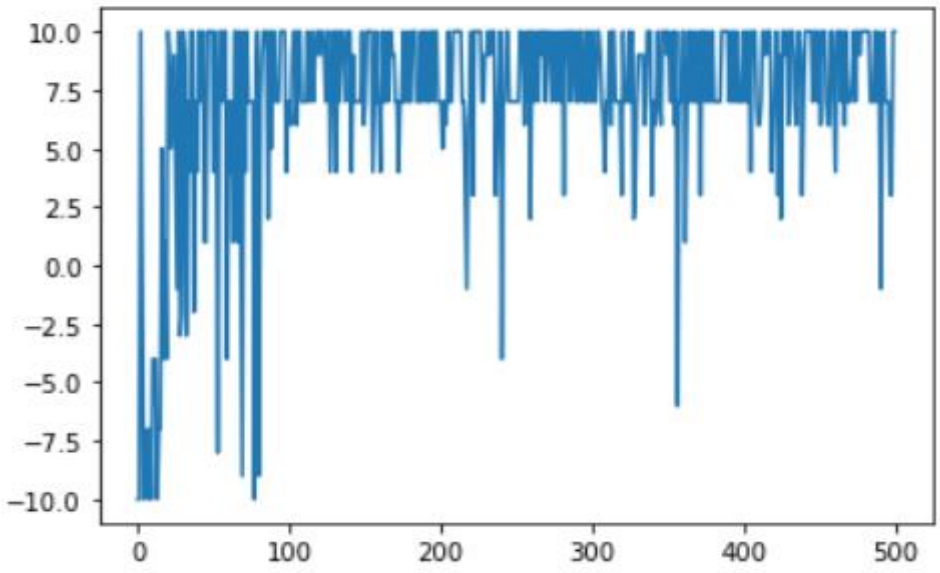
```python
class Policy: #class for action selection and value updates according to the Q-Learning rule

    def __init__(self, name):
        self.name = name

    def maxq(self, state, Actions):
        s = copy.copy(state)
        max = -10e10
        amax = None
        for a in Actions:
            u = Q.get(tuple([s,a])) #value associated to the key list [s,a]
            if u is None:
                u = -20e10
            if u > max:
                amax = copy.copy(a)
                max = copy.copy(u)
        #print(s, amax)
        return amax

    def epsilon_greedy(self, state, Actions): #epsilon-greedy decisions for systems
        #in which is possible to select any action at random given state s
        epsilon = 0.1
        r = Policy(self)
        comple = np.random.rand()
        if epsilon > comple:
            #chooses and action at random
            a = random.randint(0, (len(Actions)-1))
            #print(a)
            return a
        else:
            #Returns the action with the maximum Q-value
            a = r.maxq(state, Actions)
            #print(a)
            return a

    def updateq(self, s0,a0,r1,s1,a1):
        alpha = 0.2 #learning rate
        gamma = 1.0 #discount factor
        q0 = Q.get((s0,a0))
        if q0 is None:
                q0 = 0.0
        q1 = Q.get((s1,a1))
        if q1 is None:
                q1 = 0.0
        u = q0 + alpha*(r1 + gamma*q1 - q0)
        Q[(s0,a0)] = u
        return

```

Esta clase es la encargada de permitir la selección de acciones y actualizar los valores, lo que le permite al algoritmo de Q-Learning la toma de decisiones y actualizar el aprendizaje.

Resultados:

```
In [8]: runcell(0, 'C:/Users/Edgar/OneDrive/Documentos/ql2/ql2.py')
James Bond
in a dangerous environment
```





Observaciones y comparaciones entre el agente que explora durante 3 pasos y el agente que explora durante 10 pasos:

Como puede verse en las siguientes 2 graficas, en el primer algoritmo donde el agente puede explorar durante 3 pasos cada episodio tiene resultados un poco variados sin embargo estos se van más por recompensa con valores positivos mientras que en el segundo caso, al permitirle al agente explorar durante 10 pasos la recompensa en la mayoría de los casos es mayor, provocándole no solo una mayor recompensa a este, sino que este tiene una curva de aprendizaje más efectivo y con mejores resultados en general.