

Tema 5: Aprendizaje por Refuerzo

Q Learning

Prof. Wladimir Rodriguez

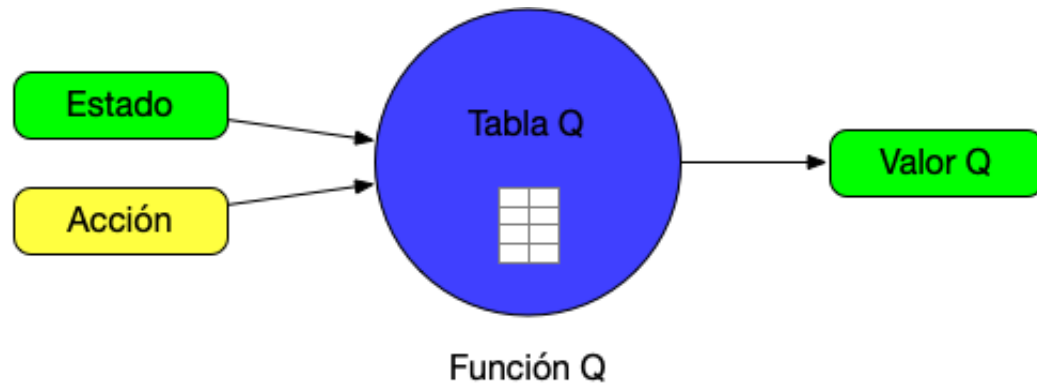
wladimir@ula.ve

Departamento de Computación

Q-learning es un algoritmo de aprendizaje por refuerzo sin modelo para aprender el valor de una acción en un estado particular. No requiere un modelo del ambiente (por lo tanto, "sin modelo"), y puede manejar problemas con transiciones estocásticas y recompensas sin requerir adaptaciones.

Para cualquier proceso de decisión de Markov finito (PDMF), **Q-learning** encuentra una política óptima en el sentido de maximizar el valor esperado de la recompensa total en todos y cada uno de los pasos sucesivos, comenzando desde el estado actual. **Q-learning** puede identificar una política de selección de acciones óptima para cualquier PDMF dado, con un tiempo de exploración infinito y una política parcialmente aleatoria. **Q** se refiere a la función que calcula el algoritmo: las recompensas esperadas por una acción realizada en un estado determinado.

















- El **Q-Learning** es el algoritmo Aprendizaje por Refuerzo que:
 - Entrena una **función Q**, que contiene, como memoria interna, una **tabla Q** la cual contiene todos los valores del par estado-acción.
 - Dado un estado y una acción, nuestra **función Q** buscará en su tabla Q el valor correspondiente.



















- Cuando finaliza el entrenamiento, tenemos una **función Q** óptima, por lo tanto, una **tabla Q** óptima.
- Y si tenemos una **función Q** óptima, tenemos una política óptima, ya que sabemos para cada estado, cuál es la mejor acción a tomar.

$$\pi^*(s) = \underset{a}{\operatorname{argmax}} Q^*(s, a)$$

Pero, al principio, nuestra **tabla Q** es inútil, ya que da un valor arbitrario para cada par estado-acción (la mayoría de las veces inicializamos la **tabla q** con 0). Pero, a medida que exploremos el ambiente y actualicemos nuestra **tabla Q**, nos dará mejores y mejores aproximaciones.

	←	↓	→	↑
	0	0	0	0
	0	0	0	0
	0	0	0	0
	0	0	0	0
	0	0	0	0
	0	0	0	0
	0	0	0	0
	0	0	0	0
	0	0	0	0
	0	0	0	0
	0	0	0	0
	0	0	0	0
	0	0	0	0
	0	0	0	0
	0	0	0	0
	0	0	0	0

Entrenamiento

	←	↓	→	↑
	0.74	0.77	0.77	0.74
	0.74	0	0.81	0.78
	0.78	0.86	0.78	0.82
	0.82	0	0.78	0.78
	0.78	0.82	0	0.74
	0	0	0	0
	0	0.90	0	0.82
	0	0	0	0
	0.82	0	0.86	0.78
	0.82	0.90	0.90	0
	0.86	0.95	0	0.86
	0	0	0	0
	0	0	0	0
	0	0.90	0.95	0.86
	0.90	0.95	1	0.90
	0	0	0	0

Ejemplo de Q learning utilizando el ambiente Frozen Lake de la librería `Gym`

Frozen Lake es un ambiente simple compuesto por mosaicos, donde el agente tiene que pasar de un mosaico inicial a uno objetivo. Los mosaicos pueden ser un lago congelado seguro o un agujero que te atrapa para siempre. El agente, tiene 4 acciones posibles: ir a la IZQUIERDA, a ABAJO, a la DERECHA o ARRIBA. El agente debe aprender a sortear los agujeros para llegar a la meta en un número mínimo de acciones. De forma predeterminada, el ambiente siempre tiene la misma configuración. En el código del ambiente, cada mosaico está representado por una letra de la siguiente manera

S F F F	(S: punto de entrada, seguro)
F H F H	(F: superficie congelada, seguro)
F F F H	(H: hueco, atrapado para siempre forever)
H F F G	(G: meta, seguro)

Importar dependencias





```
In [1]: import numpy as np
import gym

from tqdm.notebook import tqdm
```

Crear ambiente

```
In [2]: ambiente = gym.make("FrozenLake-v1", map_name="4x4", is_slippery=False)
ambiente.reset()
ambiente.render()
```

```
SFFF
FHFH
FFFH
HFFG
```

En `Frozen Lake` hay 16 mosaicos, lo que significa que nuestro agente se puede encontrar en 16 posiciones diferentes, llamadas estados. Para cada estado, hay 4 acciones posibles: ir  IZQUIERDA,  ABAJO,  DERECHA y  ARRIBA.

Espacio de Estados/Observaciones

```
In [3]: ambiente.reset()
print("____ESPACIO DE OBSERVACIONES____ \n")
print("Forma del Espacio de Observaciones", ambiente.observation_space)
print("Ejemplo de Observación", ambiente.observation_space.sample())
```

```
____ESPACIO DE OBSERVACIONES____
```

```
Forma del Espacio de Observaciones Discrete(16)
Ejemplo de Observación 2
```

Vemos con `Forma del Espacio de Observaciones(16)` que la observación es un valor que representa la posición actual del **agente como $\text{fila_actual} * \text{numero_fila} + \text{columna_actual}$ (donde tanto la fila como la columna comienzan en 0)**.

Por ejemplo, la posición del objetivo en el mapa 4x4 se puede calcular de la siguiente manera: $3 * 4 + 3 = 15$. El número de observaciones posibles depende del tamaño del mapa. **Por ejemplo, el mapa 4x4 tiene 16 posibles observaciones.**

Por ejemplo, así es como se ve estado = 0:



Espacio de Acciones

```
In [4]: print("\n ____ESPACIO DE ACCIONES____ \n")
print("Forma del Espacio de Acciones", ambiente.action_space.n)
print("Ejemplo de Acción", ambiente.action_space.sample())
```

____ESPACIO DE ACCIONES____

Forma del Espacio de Acciones 4

Ejemplo de Acción 1

El espacio de acción (el conjunto de acciones posibles que puede realizar el agente) es discreto con 4 acciones disponibles:

- 0: IR A LA IZQUIERDA
- 1: ABAJO
- 2: IR A LA DERECHA
- 3: ARRIBA

Función de recompensa:

- Alcanzar la meta: +1
- Alcanzar hoyo: 0
- Alcanzar congelado: 0

```
In [5]: espacio_de_estados = ambiente.observacion_space.n
print("Existen ", espacio_de_estados, " posibles estados")

espacio_de_accion = ambiente.action_space.n
print("Existen ", espacio_de_accion, " posibles acciones")

Existen 16 posibles estados
Existen 4 posibles acciones
```

Crear la **tabla Q** y llenarla con ceros.

```
In [6]: def inicializar_tabla_q(espacio_de_estados, espacio_de_accion):
        tablaQ = np.zeros((espacio_de_estados, espacio_de_accion))
        return tablaQ
```

```
In [7]: frozenlake_tablaQ = inicializar_tabla_q(espacio_de_estados, espacio_de_accion)
frozenlake_tablaQ
```

```
Out[7]: array([[0., 0., 0., 0.],
               [0., 0., 0., 0.],
               [0., 0., 0., 0.],
               [0., 0., 0., 0.],
               [0., 0., 0., 0.],
               [0., 0., 0., 0.],
               [0., 0., 0., 0.],
               [0., 0., 0., 0.],
               [0., 0., 0., 0.],
               [0., 0., 0., 0.],
               [0., 0., 0., 0.],
               [0., 0., 0., 0.],
               [0., 0., 0., 0.],
               [0., 0., 0., 0.],
               [0., 0., 0., 0.]])
```

Definir la Política Epsilon-Avara

Epsilon-Avara es la política de entrenamiento que maneja el compromiso entre exploración/explotación.

La idea con Epsilon Avara :

Con probabilidad $1-\epsilon$: hacemos explotación (es decir, nuestro agente selecciona la acción con el valor de par estado-acción más alto).

Con probabilidad ϵ : hacemos exploración (intentando acciones aleatorias).

Y a medida que avanza el entrenamiento vamos reduciendo progresivamente el valor de ϵ ya que cada vez necesitaremos menos exploración y más explotación.

```
In [8]: def politica_epsilon_avara(tablaQ, estado, epsilon):
# Generar un número aleatorio entre 0 y 1
numero_aleatorio = random.uniform(0,1)
# si numero_aleatorio > mayor que epsilon --> explotación
if numero_aleatorio > epsilon:
    # Tomar la acción con el valor mayor dado el estado
    accion = np.argmax(tablaQ[estado])
# else --> exploración
else:
    accion = ambiente.action_space.sample()

return accion
```

Definir los hiperparámetros

Los hiperparámetros relacionados con la exploración son algunos de los más importantes.

- Necesitamos asegurarnos de que nuestro agente **explore lo suficiente el espacio de estado** para aprender una buena aproximación de valor, para hacer eso necesitamos tener un decaimiento progresivo del ϵ .
- Si disminuye el ϵ demasiado rápido (tasa de decaimiento demasiado alta), **corre el riesgo de que su agente se quede atascado**, ya que su agente no exploró lo suficiente el espacio de estado y, por lo tanto, no puede resolver el problema.

```
In [9]: # Parámetros de Entrenamientos
n_episodios_entrenamiento = 10000 # Total de episodios de entrenamiento
tasa_de_aprendizaje = 0.7         # Tasa de aprendizaje

# Parámetros de Evaluación
n_episodios_evaluacion = 100      # Total de episodios de prueba

# Parámetros del Ambiente
nombre_ambiente = "FrozenLake-v1" # Nombre del ambiente
max_pasos = 99                    # Máximo número de pasos por episodio
gamma = 0.95                      # Taza de Descuento
semilla_evaluacion = []           # Semilla de evaluacion del ambiente

# Parámetros Exploración
max_epsilon = 1.0                 # Exploration probability at start
min_epsilon = 0.05                # Minimum exploration probability
tasa_decaimiento = 0.0005
```

Crear la funcion de entrenamiento

```
In [10]: def entrenamiento(n_episodios_entrenamiento, min_epsilon, max_epsilon, tasa_
    for episodio in tqdm(range(n_episodios_entrenamiento)):
        # Reducir epsilon (porque necesitamos menos y menos exploración)
        epsilon = min_epsilon + (max_epsilon - min_epsilon)*np.exp(-tasa_decaimiento*episodio)
        # Reiniciar el ambiente
        estado = ambiente.reset()
        paso = 0
        listo = False

        for paso in range(max_pasos):
            # Seleccionar la acción usando la epsilon de politica avara
            accion = politica_epsilon_avara(tablaQ, estado, epsilon)

            # Tomar la acción y observar el nuevo estado y la recompensa
            nuevo_estado, recompensa, listo, info = ambiente.step(accion)

            # Actualizar  $Q(s,a) := Q(s,a) + lr [R(s,a) + gamma * \max_{a'} Q(s',a') - Q(s,a)]$ 
            tablaQ[estado][accion] = tablaQ[estado][accion] + tasa_de_aprendizaje * (recompensa + gamma * max(tablaQ[nuevo_estado]) - tablaQ[estado][accion])

            # Si listo, terminar el episodio
            if listo:
                break

            # Nuestro estado es el nuevo estado
            estado = nuevo_estado
        return tablaQ
```

Entrenar el agente Q Learning


```
In [11]: frozenlake_tablaQ = entrenamiento(n_episodios_entrenamiento, min_epsilon, ma
0%|          | 0/10000 [00:00<?, ?it/s]
```

Observar la **tabla Q** resultante del entrenamiento

```
In [12]: frozenlake_tablaQ
```

```
Out[12]: array([[0.73509189, 0.77378094, 0.77378094, 0.73509189],
 [0.73509189, 0.          , 0.81450625, 0.77378094],
 [0.77378094, 0.857375   , 0.77378094, 0.81450625],
 [0.81450625, 0.          , 0.77378094, 0.77378094],
 [0.77378094, 0.81450625, 0.          , 0.73509189],
 [0.          , 0.          , 0.          , 0.          ],
 [0.          , 0.9025    , 0.          , 0.81450625],
 [0.          , 0.          , 0.          , 0.          ],
 [0.81450625, 0.          , 0.857375   , 0.77378094],
 [0.81450625, 0.9025    , 0.9025    , 0.          ],
 [0.857375   , 0.95      , 0.          , 0.857375   ],
 [0.          , 0.          , 0.          , 0.          ],
 [0.          , 0.          , 0.          , 0.          ],
 [0.          , 0.9025    , 0.95      , 0.857375   ],
 [0.9025     , 0.95      , 1.          , 0.9025     ],
 [0.          , 0.          , 0.          , 0.          ]])
```

Definir la función de evaluación

```
In [15]: def evaluar_agente(ambiente, max_pasos, n_episodios_evaluacion, Q, semilla):

    recompensa_episodio = []
    for episode in tqdm(range(n_episodios_evaluacion)):
        if semilla:
            estado = ambiente.reset(seed=semilla[episodio])
        else:
            estado = ambiente.reset()
        paso = 0
        listo = False
        recompensa_total_episodio = 0

        for paso in range(max_pasos):
            # Take the action (index) that have the maximum expected future reward
            accion = np.argmax(Q[estado][:])
            nuevo_estado, recompensa, listo, info = ambiente.step(accion)
            recompensa_total_episodio += recompensa
            if n_episodios_evaluacion == 1:
                print(nuevo_estado)
            if listo:
                break
            estado = nuevo_estado
        recompensa_episodio.append(recompensa_total_episodio)
    recompensa_media = np.mean(recompensa_episodio)
    recompensa_desviacion_estandar = np.std(recompensa_episodio)

    return recompensa_media, recompensa_desviacion_estandar
```

Evaluar a nuestro agente Q-Learning

- Normalmente deberías tener una recompensa media de 1.0
- Es relativamente fácil ya que el espacio de estado es realmente pequeño (16)

```
In [27]: recompensa_media, recompensa_desviacion_estandar = evaluar_agente(ambiente,
print(f"Recompensa media={recompensa_media:.2f} +/- {recompensa_desviacion_e

0%|          | 0/100 [00:00<?, ?it/s]
Recompensa media=1.00 +/- 0.00
```

```
In [16]: recompensa_media, recompensa_desviacion_estandar = evaluar_agente(ambiente,
print(f"Recompensa media={recompensa_media:.2f} +/- {recompensa_desviacion_e

0%|          | 0/1 [00:00<?, ?it/s]
4
8
9
13
14
15
Recompensa media=1.00 +/- 0.00
```



In []: