INTELIGENCIA ARTIFICIAL

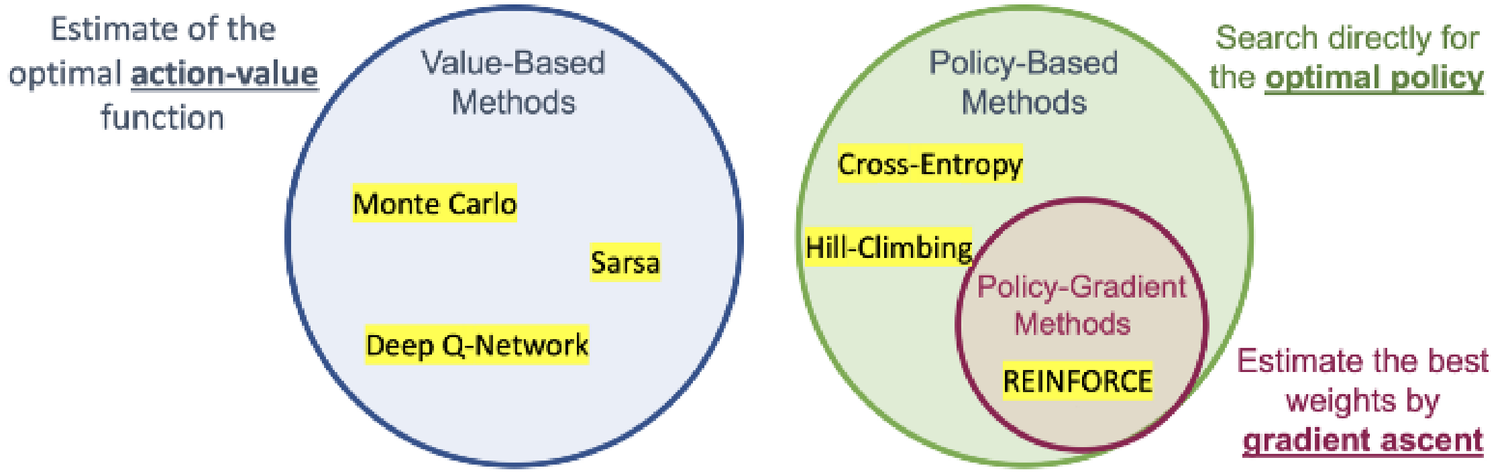
REINFORCEMENT LEARNING:
VALUE-BASED AGENTS Y SARSA

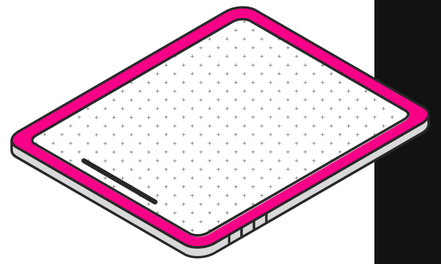
Julissa Villalobos Chaparro
Luis Carlos Parra Mendoza
Giovanny Alfonso Chávez Ceniceros

INTRODUCCIÓN

Recordemos que el objetivo del agente es encontrar una secuencia de acciones que maximicen el rendimiento; la suma de recompensas (descontadas o no descontadas, según el valor de gamma) durante un episodio o toda la vida del agente, según la tarea.

Los agentes basados en el valor (Value-based Agents) son aquellos que estiman y almacenan la función de valor y basan sus decisiones en ella.



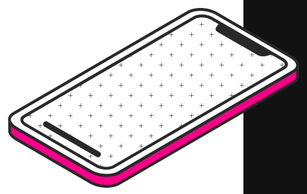


Una función de valor estima qué tan bueno es para el agente estar en un estado dado (o qué tan bueno es realizar una acción dada en un estado dado) en términos del rendimiento. Las funciones de valor se definen con respecto a formas particulares de actuar, llamadas políticas.

$$V_{\pi}(s) = E_{\pi}[R_t | s = s_t]$$

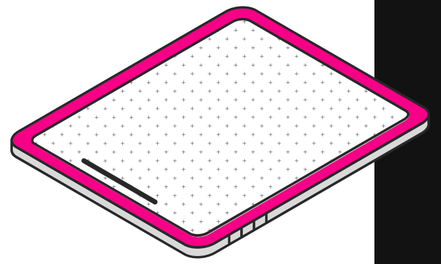
En ese sentido, ampliamos la definición de función de valor a pares de acción, estado, que se llama función de valor de acción, también conocida como función Q o simplemente Q.

$$Q_{\pi}(s, a) = E_{\pi}[R_t | s = s_t, a = a_t]$$



La política que maximiza la recompensa acumulada total se denomina política óptima.

$$V_*(s) = max\ V_\pi(s)$$
$$Q_*(s,a) = max\ Q_\pi(s,a)$$



VALUE-BASED AGENTS

DP - DYNAMIC PROGRAMMING

La ecuación de Bellman nos permite definir la función valor de forma recursiva y se puede resolver mediante programación dinámica. Para ello necesitamos recorrer todos los estados en cada iteración y además requiere que conozcamos la probabilidad de transición entre estados.

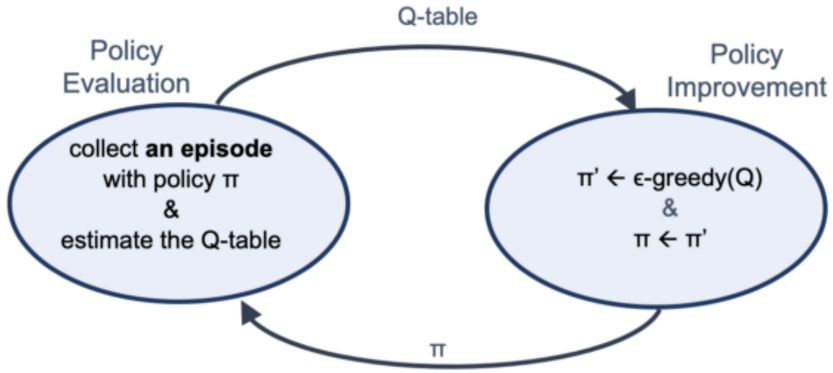
MC - MONTE CARLO

Se recopila una gran cantidad de episodios para estimar la función de valor para los estados que se vayan produciendo. Estos valores se almacenan en una estructura denominada Q-table. Después se usa la tabla para generar una política respecto a dicha tabla. La forma en que MC estima la función de valor consiste en recopilar las recompensas y promediarlas.

Uno de los principales inconvenientes de MC es el hecho de que el agente tiene que esperar hasta el final de un episodio para obtener la recompensa antes de poder actualizar y realizar mejoras en la estimación de la función de valor.

TD - TEMPORAL DIFFERENCES

Una alternativa es el enfoque conocido diferencias temporales, que implica estimar el rendimiento y aprender de los episodios incompletos, empleando la diferencia entre predicciones sucesivas temporalmente. Esta forma de actualizar una estimación con una estimación se denomina bootstrapping y tiene cierta similitud con lo que hace DP.



Este enfoque tuvo un gran impacto en el avance de los métodos de aprendizaje por refuerzo, y el aprendizaje TD es el precursor de técnicas como SARSA, DQN y más.

SARSA

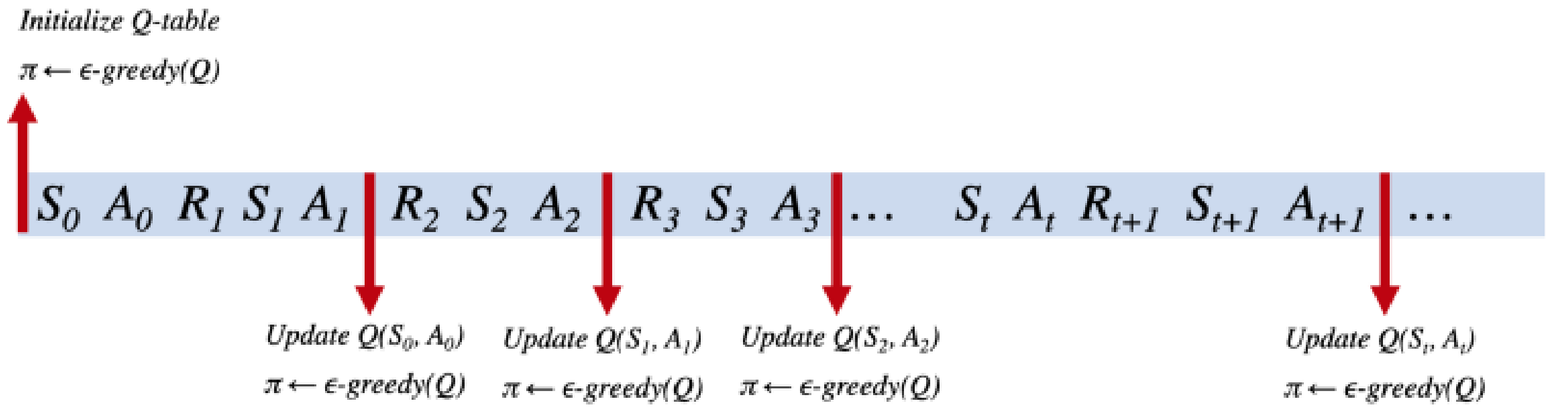
Este método actualiza el valor de la Q-table correspondiente al par estado-acción anterior. Sin embargo, en lugar de usar el rendimiento como una estimación alternativa para actualizar la tabla Q-table, usamos la suma de la recompensa inmediata R1 y el valor descontado del siguiente par acción-estado Q(S1, A1) multiplicado por el factor gamma:

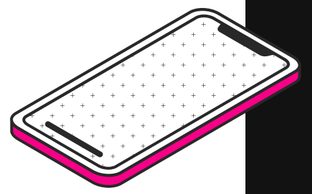
$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(R_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t))$$

ALGORITMO SARSA

- comenzamos inicializando todos los valores de acción a cero al determinar la política correspondiente.
- el agente comienza a interactuar con el entorno y recibe el primer estado SO.
- se utiliza la política para elegir la acción AO. Inmediatamente después, recibe una recompensa Rl y el siguiente estado Sl.
- el agente vuelve a utilizar la misma política para elegir la siguiente acción Al.

Esta serie de pasos define la secuencia:



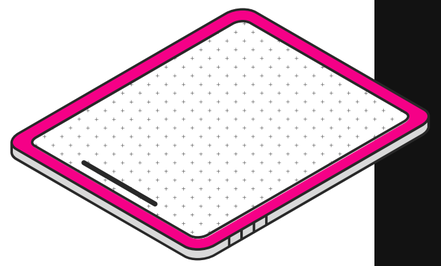


IMPLEMENTACIÓN SARSA

MÉTODO DE APRENDIZAJE/ENTRENAMIENTO

```python
class SARSAAgent:
    ...

    def learn(self, episodes, timesteps):
        # para cada epoca de entrenamiento
        for episode in range(episodes):
            # obtenemos el estado actual del ambiente y lo redimensionamos
            state = self._env.reset()
            state = self._mapState(state)
            # decaer el exploration rate por episodio
            self._decayEpsilon(episode)
            # durante X pasos generaremos una acción sobre el ambiente
            # y obtendremos una recompensa.
            for t in range(timesteps):
                # AGENTE EJECUTA ACCIÓN DADO EL ESTADO ACTUAL
                action = self._chooseAction(state)
                # obtendremos el estado nuevo del ambiente ante el estimulo dado por el agente
                # y lo redimencionamos también
                newState, reward, done, _ = self._env.step(action)
                newState = self._mapState(newState)
                # AGENTE EJECUTA ACCIÓN DADO EL NUEVO ESTADO
                newAction = self._chooseAction(newState)
                # actualizamos la Q-table
                self._learn(state, newState, action, newAction, reward)
                # actualizamos el estado actual con el nuevo
                state = newState
                # si terminó la época, salimos
                if done:
                    break
            # cerramos el ambiente
            self._env.close()
```



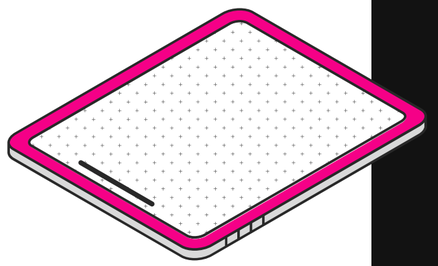
IMPLEMENTACIÓN SARSA

DETERMINAR LA FUNCIÓN DE VALOR

```
def _learn(self, state, newState, action, newAction, reward):
    # actualizar la matriz Q con la ecuación de Bellman
    self._QValues[state][action] += self._lr*(reward + self._gamma*(self._QValues[newState][newAction] - self._QValues[state][action]))
```

DETERMINAR LA ACCIÓN

```
def _chooseAction(self, state):
    # aleatoriamente regresamos una acción aleatoria para ayudar al
    # agente a que no caiga en un minimo local
    if np.random.rand() <= self._epsilon:
        return random.randrange(self._actionSize)
    # retornamos la mejor accion posible
    return np.argmax(self._QValues[state])
```



IMPLEMENTACIÓN SARSA

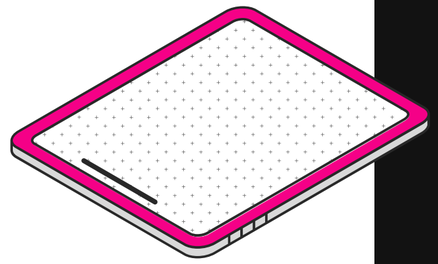
EJECUCIÓN

```python
class SARSAAgent:
    ...

    def run(self):
        t = 0
        done = False
        # obtenemos el estado actual del ambiente y lo redimensionamos
        state = self._env.reset()
        state = self._mapState(state)

        while not done:
            self._env.render()
            t = t + 1
            # AGENTE EJECUTA ACCIÓN DADO EL ESTADO ACTUAL
            action = self._chooseAction(state)
            # obtendremos el estado nuevo del ambiente ante el estimulo dado por el agente
            # y lo redimencionamos también
            newState, reward, done, _ = self._env.step(action)
            newState = self._mapState(newState)
            # actualizamos el estado actual con el nuevo
            state = newState

        # cerramos el ambiente
        self._env.close()
        return t
```

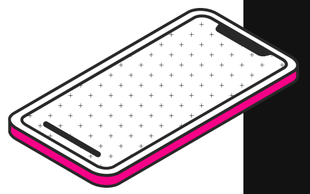


COMPARACIÓN ENTRE DQN Y SARSA

DQN es una optimización de otra variación de TD (tal y como lo es SARSA) denominada Q-Learning:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(R_t + \gamma \max_a Q(s_{t+1}, a_t))$$

Existe documentación de la implementación híbrida de SARSA; es decir la implementación de una red neuronal profunda en la estrategia SARSA de actualización de las funciones de valor.



COMPARACIÓN ENTRE DQN Y SARSA

En ese sentido podemos realizar las siguientes observaciones:

- Q-Learning aprende directamente la política óptima, mientras que SARSA aprende una política casi óptima mientras explora.

- Q-learning tiene una mayor varianza por muestra que SARSA y, como resultado, puede sufrir problemas de convergencia. Esto se convierte en un problema al entrenar redes neuronales a través de Q-learning.

- SARSA se acercará a la convergencia permitiendo posibles sanciones de movimientos exploratorios, mientras que Q-learning las ignorará.

FUENTES DE CONSULTA

- **Deep reinforcement learning explained series:**
  https://torres.ai/deep-reinforcement-learning-explained-series/

- **Repositorio en Github:**
  https://github.com/PonchoCeniceros/Reinforcement-Learning-Baby