
PONTIFICIA
UNIVERSIDAD
CATÓLICA
DEL PERÚ

# *INTELIGENCIA ARTIFICIAL (INF371)*

## *APRENDIZAJE POR REFUERZO (PARTE 3)*

Dr. Edwin Villanueva Talavera

# Contenido

- Aprendizaje por Refuerzo Activo
  - Q-learning
- Generalización en Aprendizaje por Refuerzo
- Deep Q-Networks
- Doble Deep Q-netwotks

Bibliografía:

Capitulo 21.3 y 21.4 del libro:

Stuart Russell & Peter Norvig "Artificial Intelligence: A modern Approach", Prentice Hall, Third Edition, 2010

(*) imágenes tomadas de este libro

# Aprendizaje por Refuerzo Activo

## Definición

- El agente tiene que escoger acciones
- Objetivo: aprender las acciones óptimas, aquellas que obedecen a la ecuación de Bellman:

$$U(s) = R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P(s' \mid s, a) U(s')$$

- Idea: Partiendo de un modelo de transición arbitrario
  - Refinar el modelo a través de la experiencia
  - Utilizar Programación Dinámica Adaptativa + Iteración de Valor para encontrar la política óptima para el modelo actual. Repetir.
- Crucial: tenemos que dar una manera de aprender el modelo para todos los estados y acciones

# Aprendizaje por Refuerzo Activo

## Programación Dinámica Adaptativa Codiciosa

**function** GREEDY-ADP-AGENT($percept$) **returns** an action

**inputs**: $percept$, a percept indicating the current state $s'$ and reward signal $r'$

**persistent**: $mdp$, an MDP with model $P$, rewards $R$, discount $\gamma$
$U$, a table of utilities, initially empty
$N_{sa}$, a table of frequencies for state–action pairs, initially zero
$N_{s'|sa}$, a table of outcome frequencies given state–action pairs, initially zero
$s$, $a$, the previous state and action, initially null

**if** $s'$ is new **then** $U[s'] \leftarrow r'$; $R[s'] \leftarrow r'$

**if** $s$ is not null **then**
increment $N_{sa}[s, a]$ and $N_{s'|sa}[s', s, a]$
**for each** $t$ such that $N_{s'|sa}[t, s, a]$ is nonzero **do**
$P(t \mid s, a) \leftarrow N_{s'|sa}[t, s, a] \,/\, N_{sa}[s, a]$

$U \leftarrow$ VALUE-ITERATION($mdp$)

$$\pi[s'] = \operatorname*{argmax}_{a \in A(s')} \sum_{s''} P(s'' \mid s', a) U(s'')$$

**if** $s'$.TERMINAL? **then** $s, a \leftarrow$ null **else** $s, a \leftarrow s', \pi[s']$

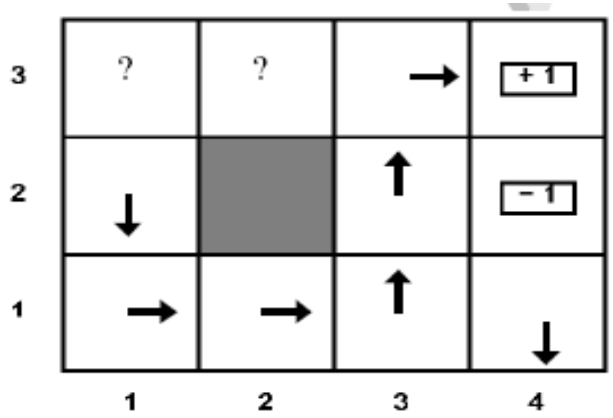
**return** $a$

*

# Aprendizaje por Refuerzo Activo

## Ejemplo: Programación Dinámica Adaptativa Codiciosa

- Imagine que el agente encuentra primero el siguiente camino:




*

- El agente siempre continuará usando esa política, nunca visitará los otros estados. ¿Por qué?
  - El agente no aprende las utilidades de las mejores regiones ya que con la política actual nunca se visita esas regiones.
  - La política es optima para el modelo aprendido, pero este no refleja el verdadero entorno

# Aprendizaje por Refuerzo Activo

## Exploración versus Explotación

- Exploración: escoger acciones sub-óptimas de acuerdo con el modelo actual para poder mejorar el modelo
- Explotación: escoger las mejores acciones para el modelo actual
- Pura explotación o pura exploración no sirve. Los agentes deben explorar más al inicio y explotar más al final.

## Posibles Soluciones

- El agente escoge acciones aleatorias una fracción de tiempo (convergencia muy lenta)
- El agente escoge acciones con probabilidad inversamente proporcional al número de veces que la acción fue ejecutada en el estado. Se implementa a través de una función de exploración

# Aprendizaje por Refuerzo Activo

## Programación Dinámica Adaptativa con Exploración

- La iteración de valor en Greedy-ADP se realiza usando la ecuación de Bellman alterada:

$$U^+(s) \leftarrow R(s) + \gamma \max_a f\left(\sum_{s'} P(s' \mid s,a)U^+(s'),\ N(s,a)\right)$$

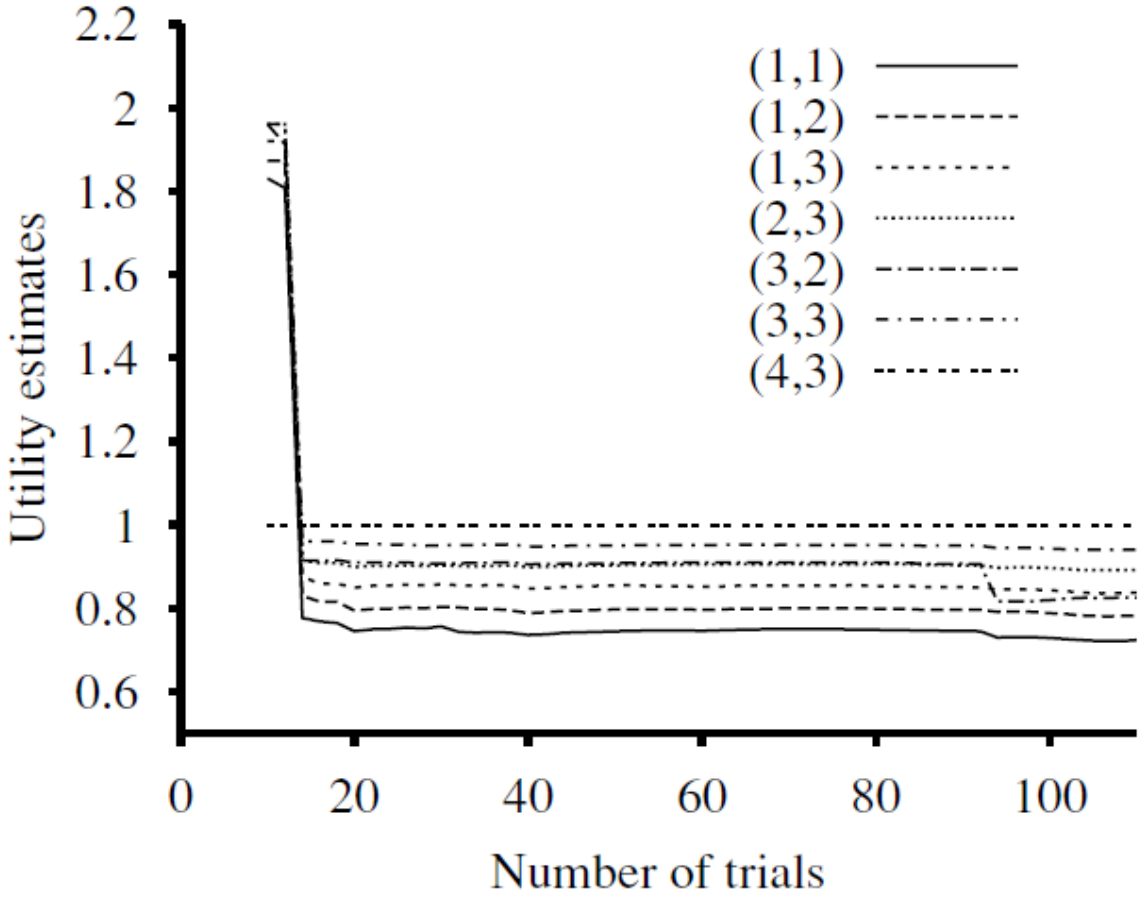
donde *f* es la función de exploración. Una forma simple es:

$$f(u,n) = \begin{cases} R^+ & \text{if } n < N_e \\ u & \text{otherwise} \end{cases}$$

- $U^+(s)$ es una utilidad optimista ya que, si el estado no ha sido visitado con frecuencia, muestra al estado con alta utilidad para hacer que el agente tome acciones que visiten dicho estado
- La acción que se toma $\pi(s)$ es la que maximiza *f*

# Aprendizaje por Refuerzo Activo

## Programación Dinámica Adaptativa con Exploración




*

Curva de aprendizaje de utilidades en el mundo 4x3 con ADP exploratorio. $R^+=2$, $N_e=5$

# Aprendizaje por Refuerzo Activo

## Diferencias Temporales en aprendizaje activo

- A diferencia del caso pasivo, en el caso activo el agente no tiene una política fija
- Una extensión natural seria aprender el modelo de transición P para poder escoger la acción que mas utilidad esperada le dé:

$$\pi(s) = \underset{a \in A(s)}{\operatorname{argmax}} \sum_{s'} P(s' \mid s, a) U(s')$$

La actualización de utilidades seria la misma forma que en el caso pasivo:

$$U^{\pi}(s) \leftarrow U^{\pi}(s) + \alpha(R(s) + \gamma\, U^{\pi}(s') - U^{\pi}(s))$$

- Existe un método alternativo que no requiere modelo: Q-learning

# Aprendizaje por Refuerzo Activo

## Q-learning

- Denotaremos $Q(s, a)$ la utilidad de ejecutarse la acción $a$ en el estado $s$. La relación con la utilidad del estado es:

$$U(s) = \max_a Q(s, a)$$

- La función $Q(s,a)$ es útil porque a partir de ella podemos calcular directamente la política, si necesitar de modelo de transicion.
- **Ecuación de actualización**: con cada transición observada $s$ -> $s'$ con acción $a$ ejecutada se actualiza $Q(s,a)$ :

$$Q(s, a) \leftarrow Q(s, a) + \alpha(R(s) + \gamma \max_{a'} Q(s', a') - Q(s, a))$$

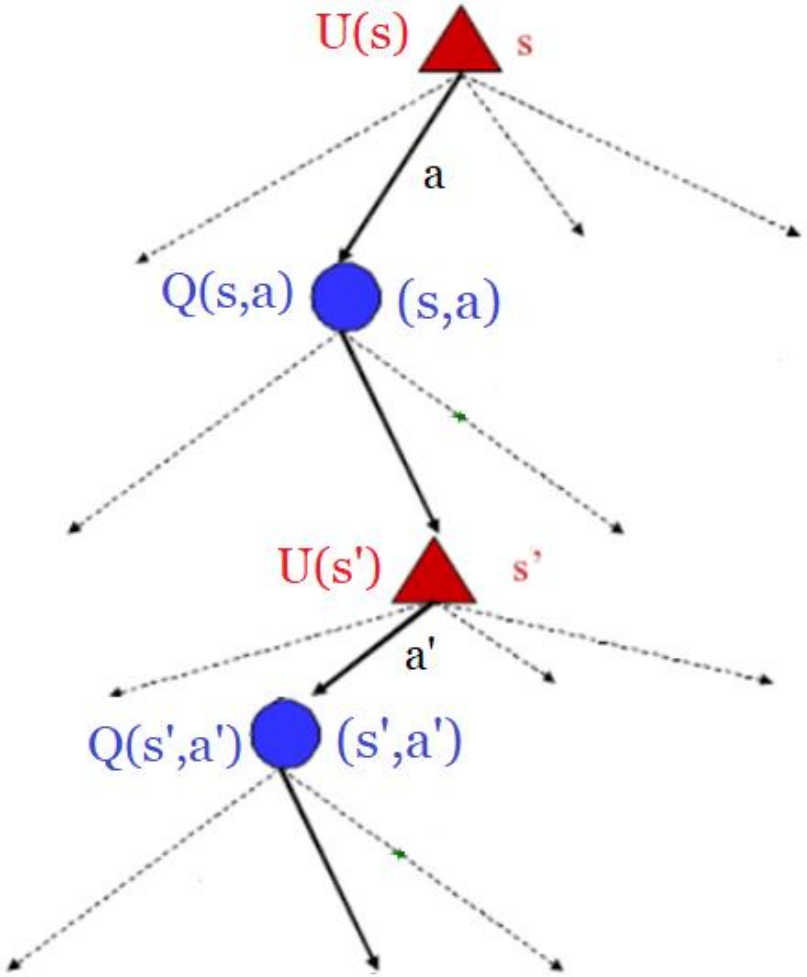
- A medida que se realiza iteraciones, los valores $Q(s,a)$ se acercan a los valores de equilibrio:

$$Q(s, a) = R(s) + \gamma \sum_{s'} P(s' \mid s, a) \max_{a'} Q(s', a')$$

# Aprendizaje por Refuerzo Activo

## Q-learning:

Opera en el espacio de Q-estados



# Aprendizaje por Refuerzo Activo

## Q-learning:

*

**function** Q-LEARNING-AGENT($percept$) **returns** an action

**inputs**: $percept$, a percept indicating the current state $s'$ and reward signal $r'$

**persistent**: $Q$, a table of action values indexed by state and action, initially 0

$N_{sa}$, a table of frequencies for state–action pairs, initially zero

$s$, $a$, $r$, the previous state, action, and reward, initially null

**if** TERMINAL?($s$) **then** $Q[s, None] \leftarrow r'$

**if** $s$ is not null **then**

increment $N_{sa}[s, a]$

$Q[s, a] \leftarrow Q[s, a] + \alpha(N_{sa}[s, a])(r + \gamma \max_{a'} Q[s', a'] - Q[s, a])$

$s, a, r \leftarrow s', \text{argmax}_{a'} f(Q[s', a'], N_{sa}[s', a']), r'$

**return** $a$

*f* es la misma función de exploración del ADP exploratorio.

$$f(u, n) = \begin{cases} R^+ & \text{if } n < N_e \\ u & \text{otherwise} \end{cases}$$

# Generalización en Aprendizaje por Refuerzo

- Los agentes vistos hasta ahora guardan las utilidades y Q valores en tablas (un valor por estado o estado-acción)
- Funcionan bien para espacios de estados pequeños. Para espacios grandes no son adecuados (ej. Backgammon, Ajedrez)
- Una forma de escalar a problemas grandes es estimar las utilidades o Q-valores con funciones aproximadas. Por ejemplo, podríamos usar una aproximación lineal de la utilidad:

$$\hat{U}_\theta(s) = \theta_1 f_1(s) + \theta_2 f_2(s) + \cdots + \theta_n f_n(s)$$

donde: $f_1, \ldots, f_n$ son atributos que describen el estado, y

$\theta_1, \ldots, \theta_n$ son parámetros ajustables para aproximar la utilidad

- Al usar una función aproximada se comprime la definición de utilidades (o q-valores) del número de estados del problema a n (# de parámetros) (ej. en ajedrez de $10^{40}$ a 20)

# Generalización en Aprendizaje por Refuerzo

- Otra ventaja de funciones aproximadas es que ellas habilitan al agente a generalizar de estados visitados a estados no visitados
- La clave está en escoger la forma funcional adecuada (hipótesis). Entre más parámetros tenga más tiempo y datos serán necesarios
- Para el caso del laberinto 3x4 y una aproximación lineal podemos usar las coordenadas *x*, *y* como atributos:

$$\hat{U}_\theta(x, y) = \theta_0 + \theta_1 x + \theta_2 y$$

- Regresion lineal puede estimar los parámetros si hay datos.
- Es preferible usar aprendizaje online al fin de cada trial. Si $u_j(s)$ es la utilidad observada en trial $j$, el error de la función aproximada será:

$$E_j(s) = (\hat{U}_\theta(s) - u_j(s))^2/2$$

# Generalización en Aprendizaje por Refuerzo

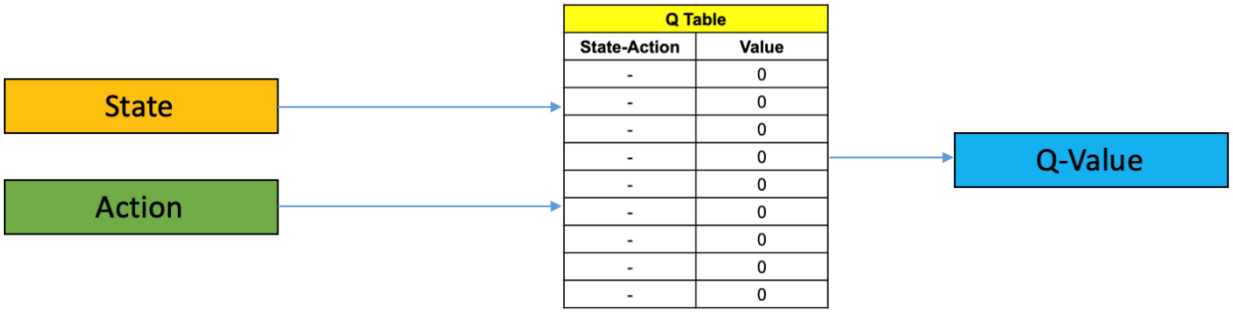
- Para el caso de Q-learning la regla de actualización online seria:

$$\theta_i \leftarrow \theta_i + \alpha \left[ R(s) + \gamma \max_{a'} \hat{Q}_\theta(s', a') - \hat{Q}_\theta(s, a) \right] \frac{\partial \hat{Q}_\theta(s, a)}{\partial \theta_i}$$

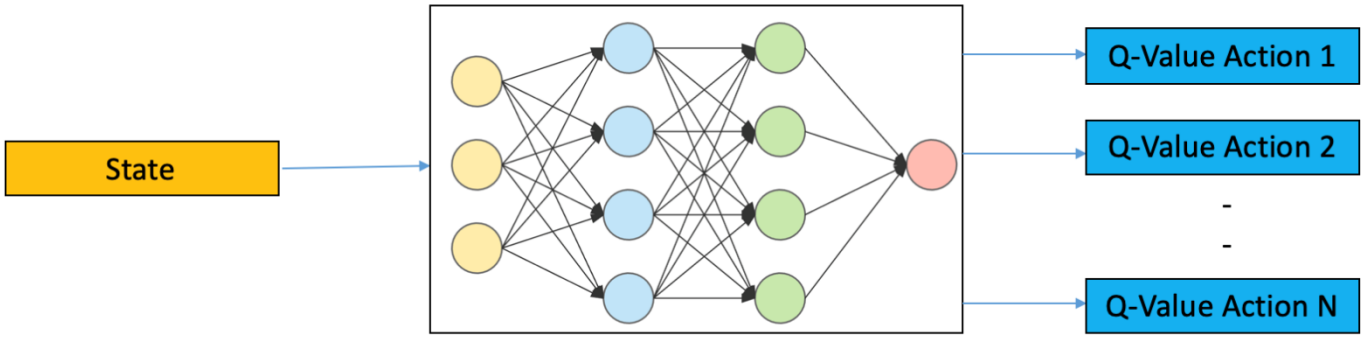
- Una buena alternativa como aproximador funcional es usar redes neuronales. Estos son modelos bastante flexibles. El cálculo de gradientes es simple si se usa funciones de activación diferenciables.
- De hecho, se ha usado redes neuronales profundas para ello, dando lugar al famoso Deep Q-learning (Mnih *et al.*, Nature 02/2015)
  - Testado en 49 juegos del Atari 2600. Entradas de 84x84 pixeles y score del juego. Nivel comparable al de un testador profesional de juegos (mismo algoritmo, arquitectura y híper-parámetros)

# Deep Q-networks

- En Deep Q-network se usa una Red neuronal para aproximar los valores Q




Q Learning




Deep Q Learning

# Deep Q-networks

- Cada experiencia simple es almacenada en memoria. Normalmente una experiencia simple es un cuarteto:

(state, action, next_state, reward)

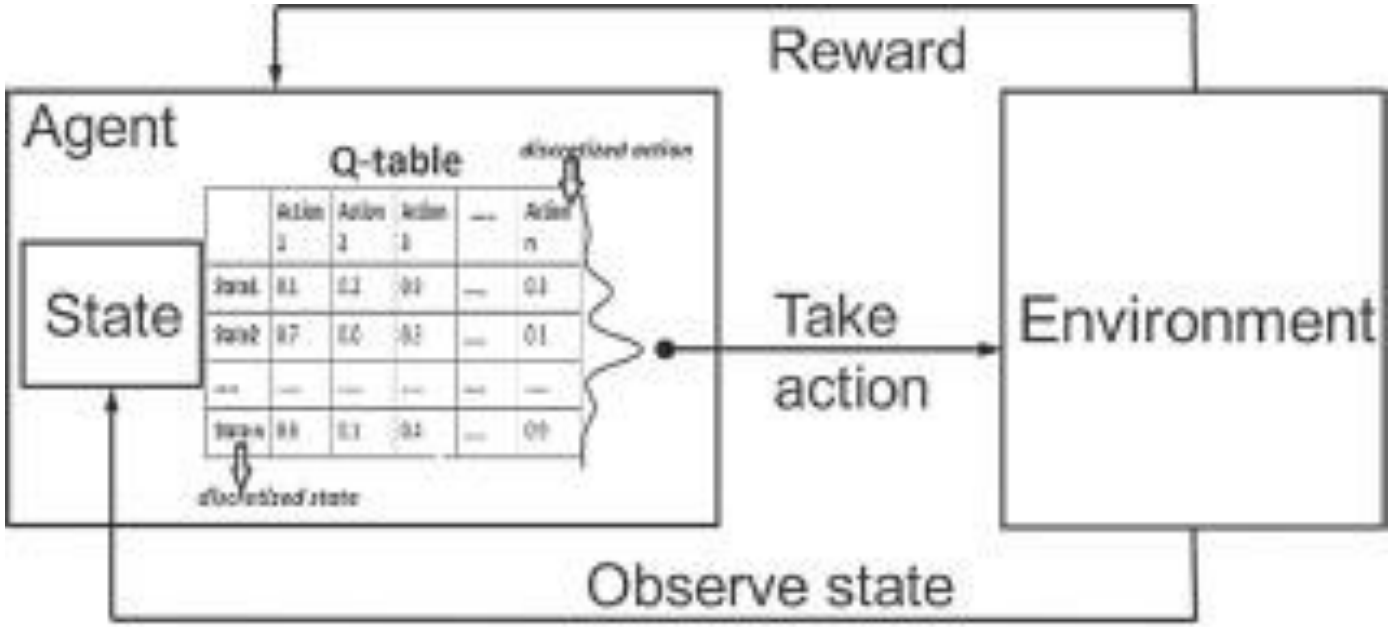
$S_t$ $A_t$ $S_{t+1}$ $R_{t+1}$

- La acción tomada por el agente es la que tiene el máximo valor Q predicho por el modelo
- La función de perdida es el Error Cuadrático Medio (MSE) del valor Q predicho y el valor target Q* . Como no se conoce el target entonces se usa como target

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\right]$$

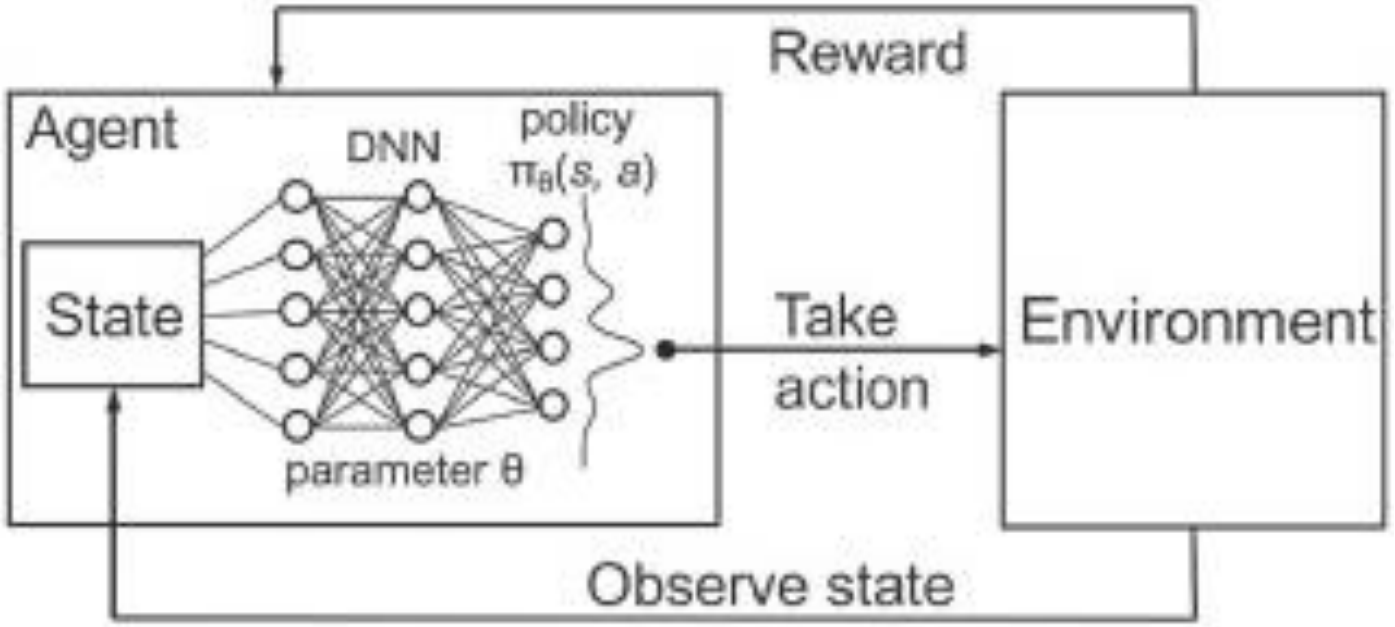
# Deep Q-networks

- Operación del Agente Q-learning



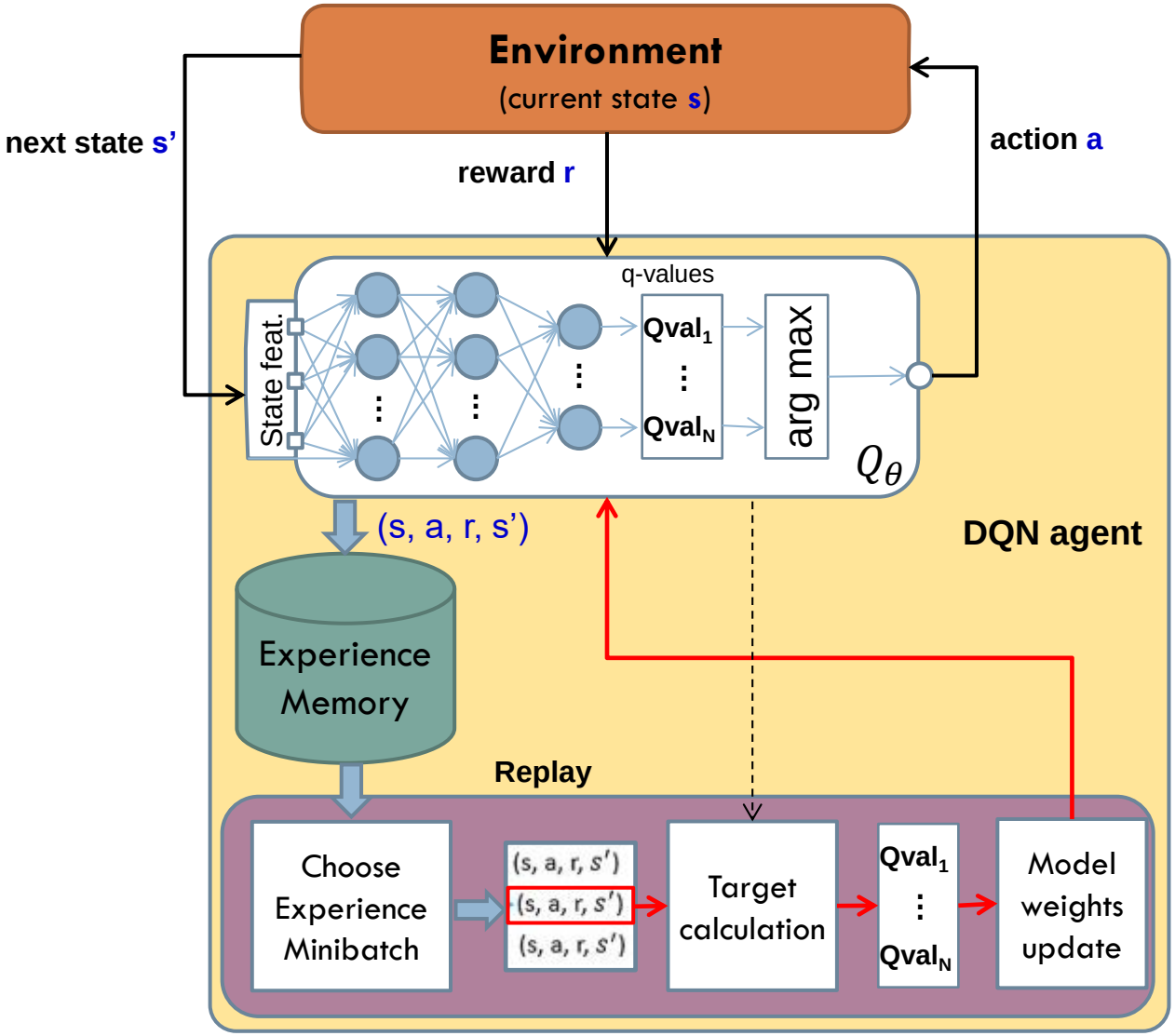
# Deep Q-networks

- Operación del Agente Deep Q-network



# Deep Q-networks

- Flujo de Entrenamiento de un DQN



# Deep Q-networks

Initialize experience memory $D$
Initialize model $Q_\theta$ with random weights $\theta$
**for** episode 1:n **do**
    *# Make an episode experience with model $Q_\theta$*
    s = reset_environment()
    **while** $s != terminal$
        select an action $a$
            with probability $\varepsilon$: $a \leftarrow random(Actions(s))$
            otherwise: $a \leftarrow argmax_{a'} Q_\theta(s, a')$
        execute action $a$ in environment and observe reward $r$ and new state $s'$
        store transition $[s, a, r, s']$ in experience memory $D$
        $s \leftarrow s'$

    *# Update the model $Q_\theta$ (replay)*
    get a random sample of experiences: $Minibatch \leftarrow Sample(D, batchsize)$
    **for each** transition $[s, a, r, s'] \in Minibatch$
        $\boldsymbol{t} \leftarrow Q_\theta(s)$ *# vector of q-values predicted by the current model*
        **if** $s' = terminal$:
            $\boldsymbol{t}_a \leftarrow r$
        **else**:
            $\boldsymbol{t}_a \leftarrow r + \gamma\, max_{a'} Q_\theta(s', a')$ *# future discounted reward obtained with the model*
        update weights $\theta$ of model $Q_\theta$ with example $\langle s, \boldsymbol{t} \rangle$
    $\varepsilon \leftarrow \varepsilon * decay$ *# decay the probability of random actions (exploration)*

# Double Deep Q-networks

- DQN ajusta el modelo $Q_\theta$ con cada experiencia (transición) revisada en el minibatch de replay. El problema es que la construcción del target de entrenamiento se realiza con el mismo modelo que esta siendo ajustado $Q_\theta$ : $\boldsymbol{t}_a \leftarrow r + \gamma \, max_{a'} \, Q_\theta(s', a')$

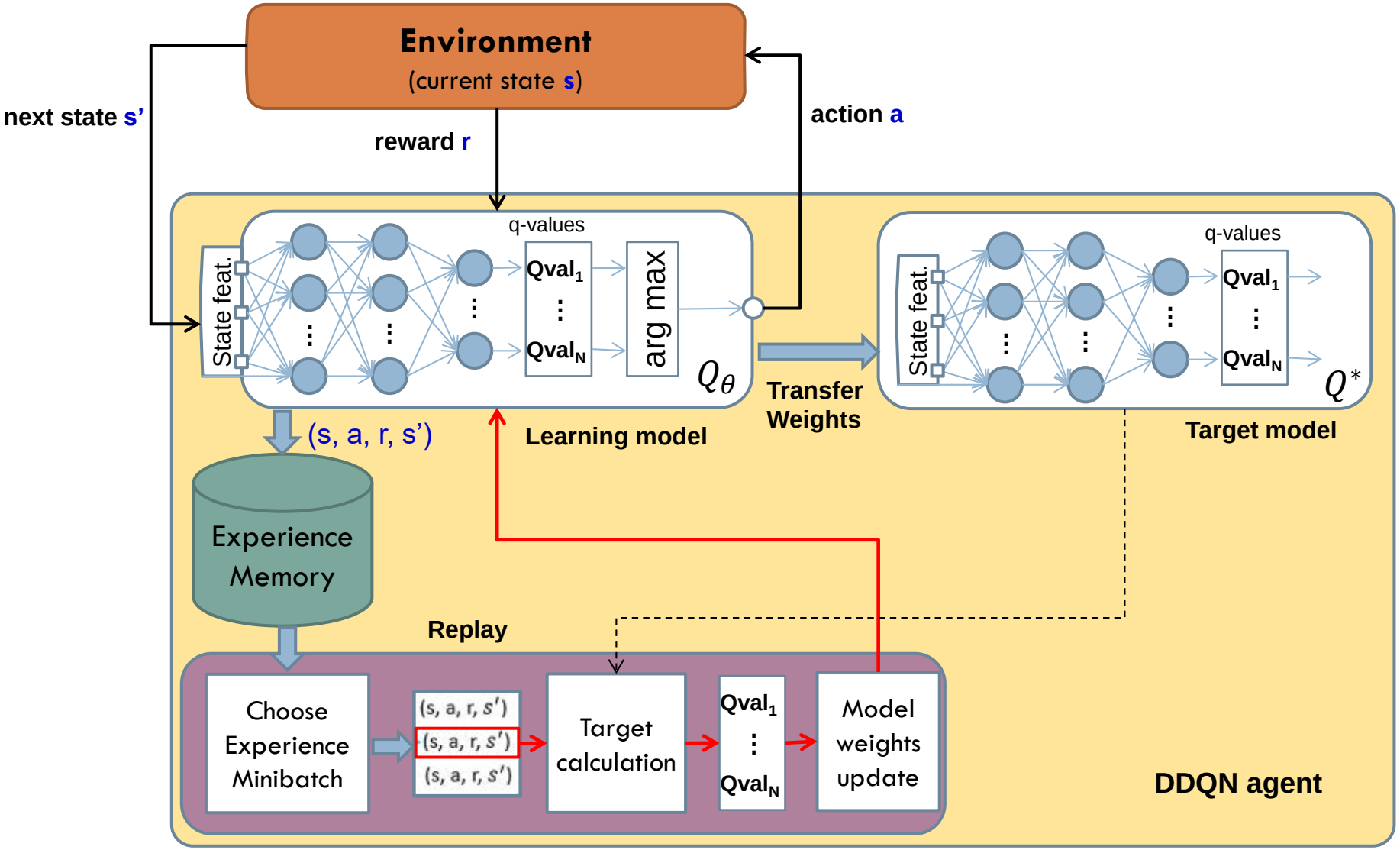
- Ello trae como consecuencia una inestabilidad de aprendizaje en DQN

- Una forma de aliviar ese problema es Double Deep Q-networks, el cual usa un modelo adicional $Q^*$ (target model) para estimar el target durante la etapa de replay: $\boldsymbol{t}_a \leftarrow r + \gamma \, max_{a'} \, Q^*(s', a')$

- Después de cada etapa de replay se transfieren los pesos de $Q_\theta$ a $Q^*$ , y este ultimo se usa como estimador de target en el siguiente replay

# Double Deep Q-networks

- Flujo de Entrenamiento de un Double DQN (DDQN)



# Double Deep Q-networks

Initialize experience memory $D$
Initialize model $Q_\theta$ with random weights $\theta$
Initialize model $Q^*$ with random weights $\theta^*$
**for** episode 1:n **do**
    *# Make a episode experience with model $Q_\theta$*
    s = reset_environment()
    **while** $s != terminal$
        select an action $a$
            with probability $\varepsilon$: $a \leftarrow random(Actions(s))$
            otherwise: $a \leftarrow argmax_{a'} Q_\theta(s, a')$
        execute action $a$ in environment and observe reward $r$ and new state $s'$
        store transition $[s, a, r, s']$ in experience memory $D$
        $s \leftarrow s'$

    *# Update the model $Q_\theta$ (replay)*
    get a random sample of experiences: $Minibatch \leftarrow Sample(D, batchsize)$
    **for each** transition $[s, a, r, s'] \in Minibatch$
        $\boldsymbol{t} \leftarrow Q_\theta(s)$ *# vector of q-values predicted by the model being learned*
        **if** $s' = terminal$:
            $\boldsymbol{t}_a \leftarrow r$
        **else**:
            $\boldsymbol{t}_a \leftarrow r + \gamma\, max_{a'} Q^*(s', a')$ *# future discounted reward obtained with the target model*
            update weights $\theta$ of model $Q_\theta$ with example $\langle s, \boldsymbol{t} \rangle$
    transfer weights to the target model : $\theta^* \leftarrow \theta$
    $\varepsilon \leftarrow \varepsilon * decay$ *# decay the probability of random actions (exploration)*

# Referencias y Material complementar



- DQN: https://web.stanford.edu/class/psych209/Readings/MnihEtAlHassibis15NatureControlDeepRL.pdf
- DDQN: https://www.aaai.org/ocs/index.php/AAAI/AAAI16/paper/download/12389/11847
- DEMYSTIFYING DEEP REINFORCEMENT LEARNING: https://neuro.cs.ut.ee/demystifying-deep-reinforcement-learning/
- Deep Reinforcement Learning: Pong from Pixels: https://karpathy.github.io/2016/05/31/rl/
- A Beginner's Guide to Deep Reinforcement Learning: https://skymind.ai/wiki/deep-reinforcement-learning

# Preguntas?