# Anexos

# Anexo A

# Algoritmos de RL

A continuación se presentan brevemente dos algoritmos habituales en RL, *Q-learning* (*value iteration*) y *Policy Gradient* (*policy search*). El algoritmo de *Q-learning* es equivalente al empleado en la sección 4.3.

### Q-learning

Uno de los algoritmos basados en la función de valor más empleados es el denominado *Q-learning*, en el que se aprende la regla óptima directamente. Se trata de un método *off-policy* (puesto que además emplea, por ejemplo, *policies* $\varepsilon - greedy$ o de Boltzmann) basado también en TD (*temporal difference*) y que se ejecuta en numerosos algoritmos actuales de RL y DRL, pese a que fue propuesto hace casi tres décadas por Watkins y Dayan (1992).

Partiendo del algoritmo conocido como TD(0) propuesto por Sutton (1988) y la ecuación (3.10), se puede estimar el Q-value en cada instante a partir del anterior según la ecuación (A.1) – denominada ecuación de Bellman –, y a partir de ella definir la regla óptima.

$$Q_t(s_t, a_t) = (1-\alpha)\underbrace{Q_{t-1}(s_t, a_t)}_{\text{antiguo Q-value}} + \alpha \overbrace{\left( \mathcal{R}(s_t, a_t) + \gamma \underbrace{\max_a Q_{t-1}(s_{t+1}, a)}_{\substack{\text{estimación del Q-value} \\ \text{óptimo en } s'}} \right)}^{\text{valor aprendido}} \quad \text{(A.1)}$$

Donde $\alpha \in (0, 1]$ es el factor de aprendizaje que determina cuánto aprende el agente con cada experiencia.

En el algoritmo 1 se presenta el proceso *off-policy* de *Q-learning* en pseudocódigo, donde $\pi$ hace referencia a la *policy* $\varepsilon$*-greedy* u otra similar.

Una alternativa *on-policy* también basada en TD es SARSA (*State-Action-Reward-State-Action*), un algoritmo similar al presentado pero en el que en cada iteración se toman dos acciones y se emplea para ello la *policy* calculada.

**Algoritmo 1** Q-Learning

Parameters: $\alpha \in (0, 1]$, $\gamma \in (0, 1]$, $\varepsilon \in (0, 1]$
Initialise Q-table with arbitrary Q-values
**for** episode $\leftarrow$ 1 to max episodes **do**
    **while** $s_t$ not terminal **and** step $<$ max steps **do**
        Perceive $s_t$
        Select $a_t \leftarrow \pi(s_t|\varepsilon)$ ▷ $\varepsilon-$greedy policy
        Take $a_t$, get $r_t$ and perceive $s_{t+1}$
        **if** $s_{t+1}$ is terminal **then**
            $Q_t \leftarrow r_t$
        **else**
            $Q_t \leftarrow r_t + \gamma \max_a Q(s_{t+1}, a)$
        **end if**
        $Q(s_t, a_t) \leftarrow (1 - \alpha)Q(s_t, a_t) + \alpha Q_t$ ▷ Bellman eq.
        $s_t \leftarrow s_{t+1}$
        step $\leftarrow$ step+1
    **end while**
    $\varepsilon \leftarrow \varepsilon * \varepsilon_{decay}$
**end for**

### Policy Gradient

El núcleo de este tipo de algoritmos es la actualización iterativa de un conjunto de parámetros $\theta$ de la regla de forma que la recompensa esperada incremente. El proceso de optimización se puede formular como $\theta_{i+1} \doteq \theta_i + \Delta\theta_i$, o, en el caso de los métodos basados en el gradiente,

$$\theta_{i+1} \doteq \theta_i + \alpha \nabla_\theta J(\theta_i) \tag{A.2}$$

Donde $\nabla_\theta J$ puede estimarse mediante distintos métodos más o menos sofisticados (diferencias finitas, ratios de verosimilitud...). Brevemente se presenta la teoría sobre ratios de versimilitud (*likelihood ratios*), empleada a menudo para resolver el problema.

Sea la función objetivo (no empleada únicamente en estos métodos):

$$J(\theta) = \max_\theta \mathbb{E}\left[\sum_{t=0}^{H} R(s_t)|\pi_\theta\right] = \mathbb{E}\left[\sum_{t=0}^{H} R(s_t, a_t)|\pi_\theta\right] = \sum_\tau \mathbb{P}_\theta(\tau) R(\tau) \tag{A.3}$$

Donde $R(\tau)$ es la recompensa total en el episodio o trayectoria $\tau$, esto es, $R(\tau) = \sum_{t=1}^{H} \gamma_t r_t$, y $\mathbb{P}_\theta(\tau) \equiv \mathbb{P}(\tau|\theta)$ la distribución de probabilidad de la trayectoria.

Entonces, el objetivo es encontrar un conjunto de parámetros $\theta$ tal que la trayectoria que creen (i.e. $\tau = (s_1, a_1, ..., s_H, a_H)$) maximice la recompensa

esperada:

$$\max_{\theta} J(\theta) = \max_{\theta} \sum_{\tau} \mathbb{P}(\tau|\theta) R(\tau) \tag{A.4}$$

Puesto que se busca resolver el problema de optimización empleando el gradiente estocástico ascendente (*Stochastic Gradient Ascent*, SGA), ecuación (A.2), hay que conseguir definir el gradiente de $J$:

$$\nabla_\theta J_\theta = \nabla_\theta \left( \sum_{\tau} \mathbb{P}(\tau|\theta) R(\tau) \right) \tag{A.5}$$

$$= ... = \sum_{\tau} \mathbb{P}(\tau|\theta) \nabla_\theta \log \mathbb{P}_\theta(\tau) R(\tau) \tag{A.6}$$

$$= \mathbb{E}\left[ \nabla_\theta \log \mathbb{P}_\theta(\tau) R(\tau) \right] \tag{A.7}$$

Nótese que se ha empleado la regla de Leibniz integral para introducir el gradiente en el sumatorio, y que en la ecuación (A.6) se ha empleado el ratio de verosimilitud (*likelihood ratio*), definido como:

$$\nabla_\theta \mathbb{P}_\theta(\tau) \doteq \mathbb{P}_\theta(\tau) \nabla_\theta \log \mathbb{P}_\theta(\tau) \tag{A.8}$$

Finalmente, la ecuación (A.7) se puede expresar en función de los estados y acciones, de forma que:

$$\nabla_\theta J_\theta = \mathbb{E}\left[ \sum_{t=0}^{H} \nabla_\theta \log \pi_\theta(s_t, a_t) R(s_t, a_t) \right] \tag{A.9}$$

Donde $\pi_\theta(s_t, a_t) = \pi(a_t|s_t, \theta)$.

Redefiniendo ahora la función objetivo como:

$$J(\theta) = \sum_{s \in \mathcal{S}} d^\pi(s) V^\pi(s) \tag{A.10}$$

$$= \underbrace{\sum_{s \in \mathcal{S}} d^\pi(s)}_{\substack{\text{Probabilidad de} \\ \text{estar en } s}} \quad \underbrace{\sum_{a \in \mathcal{A}} \pi_\theta(s, a)}_{\substack{\text{Probabilidad de} \\ \text{escoger } a \text{ dados } s \text{ y } \pi}} \quad Q^\pi(s, a) \tag{A.11}$$

Donde $d^\pi(s)$ es la distribución estacionaria de la cadena de Markov para $\pi_\theta$, i.e. la probabilidad de ocurrencia del estado, $d^\pi(s) \doteq \frac{N(s)}{\sum N}$ con $N(s)$ el número de ocurrencias del estado $s$.

Se puede llegar entonces a que:

$$\nabla_\theta J_\theta = \mathbb{E}_\pi \left[ \nabla_\theta \left( \log \pi_\theta(s, a) \right) Q(s, a) \right] \tag{A.12}$$

La teoría sobre PG (*Policy Gradient*) es la base de algoritmos como REINFORCE (*Monte-Carlo Policy Gradient*), DPG (*Deterministic Policy Gradient*) o A3C (*Asynchronous Advantage Actor-Critic*). Mas detalles pueden encontrarse en (Meyer, 2018).

# Anexo B

# Algoritmos de DRL

En este Anexo se presentan los algoritmos de DRL empleados en la sección 4.4, a saber: *Deep Q-Network*, *Double Deep Q-Network* y *Dueling Double Deep Q-Network*.

### DQN

*Deep Q-Network* aproxima el cálculo de los *Q-values* en *Q-learning* mediante una red neuronal (profunda), habitualmente con capas convolucionales si su entrada son imágenes. Este tipo de redes, introducidas por Google DeepMind en (Volodymyr Mnih et al., 2015), actúan como una función de aproximación no lineal, lo que hace que RL sea inestable o diverja. Tres problemas principales surgen en esta red: la correlación temporal entre experiencias pasadas, el olvido de ciertas experiencias lejanas y la búsqueda continua de un objetivo móvil que desemboca en inestabilidades. Los dos primeros son solucionados con el uso de un *buffer* de memoria, denominado *Experience Replay* (ER), que guarda las experiencias conforme se van generando. En cada iteración, la optimización se realiza con un conjunto de experiencias (*batch*) que se muestrea de forma aleatoria. En cuanto al tercer problema; considérese la siguiente variación de los pesos de la red:

$$\Delta\theta = \alpha \underbrace{\left[ \underbrace{(R + \gamma \max_a Q(s', a, \theta))}_{\text{Q target}} - Q(s, a, \theta) \right]}_{\text{TD Error (loss)}} \underbrace{\nabla_\theta Q(s, a, \theta)}_{\text{Q gradient}} \tag{B.1}$$

Puede notarse se están usando los mismos pesos para calcular el *Q-value* y el *Q-target*. Como consecuencia, se genera una correlación entre la pérdida y los parámetros que se están cambiado, esto es, los *Q-values* varían pero también lo hace el objetivo al que estos intentan parecerse. DeepMind introdujo la idea de los *Q-targets* fijos. Empleando una red secundaria (*target network*) cuyos pesos $\theta^*$ cambien únicamente cada cierto número de episodios, se puede fijar

la función objetivo. De esta forma, la actualización de pesos será:

$$\Delta\theta = \alpha \left[ (R + \gamma \max_a Q(s', a, \theta^*)) - Q(s, a, \theta) \right] \nabla_\theta Q(s, a, \theta) \tag{B.2}$$

El algoritmo 2 muestra el proceso de aprendizaje de estos agentes.

**Algoritmo 2** Deep Q-learning

Parameters: $\alpha \in (0, 1]$, $\gamma \in (0, 1]$, $\varepsilon \in (0, 1]$
Initialise Memory $\mathcal{M}$ with $N$ samples
Initialise Q-network $Q$ with random parameters $\theta$
Initialise target Q-network $Q^*$ with parameters $\theta^* \leftarrow \theta$
**for** episode $\leftarrow 1$ to max episodes **do**
    **while** $s_t$ not terminal **and** step $<$ max steps **do**
        Perceive $s_t$
        Select $a_t \leftarrow \pi(s_t|\varepsilon)$ ▷ $\varepsilon-$greedy policy
        Take $a_t$, get $r_t$ and perceive $s_{t+1}$ and $e_t$
        Store tuple $(s_t, a_t, r_t, s_{t+1}, e_t)$ in $\mathcal{M}$
        Sample minibatch of tuples $(s_i, a_i, r_i, s_{i+1}, e_i)$ from $\mathcal{M}$ ▷ ER
        **for** each tuple in minibatch **do**
            **if** $s_{i+1}$ is terminal **then**
                $Q_{target} \leftarrow r_i$
            **else**
                $Q_{target} \leftarrow r_i + \gamma \max_a Q(s_{i+1}, a|\theta^*)$ ▷ DQN
            **end if**
        **end for**
        Perform optimization on $(Q_{target} - Q(s, a|\theta))^2$ w.r.t. $\theta$
        Every $U$ steps set $\theta^* \leftarrow \theta$ ▷ Target network update
        $s_t \leftarrow s_{t+1}$
        step $\leftarrow$ step+1
    **end while**
    $\varepsilon \leftarrow \varepsilon * \varepsilon_{decay}$
**end for**

### DDQN

*Double Deep Q-Network*, propuesta por (Van Hasselt et al., 2016), actúa sobre el problema de la sobreestimación realizada por la operación máx(·) de la función objetivo:

$$Q_{target} = R + \gamma \max_a Q\left(s_{t+1}, a|\theta_t\right) \tag{B.3}$$

$$= R + \gamma Q\left(s_{t+1}, \arg\max_a Q(s_{t+1}, a|\theta_t)|\theta_t\right) \tag{B.4}$$

De esta forma, las acciones objetivo se calculan como aquellas con el *Q-value* mayor, lo que puede no ser siempre cierto, especialmente al comienzo del

entrenamiento cuando no se tiene suficiente información. Esta sobreestimación puede reducirse empleando la red entrenada para escoger la que sería la mejor acción para el siguiente estado y la red objetivo para estimar su *Q-value*, esto es,

$$Q_{target} = R + \gamma Q\left(s_{t+1}, \arg\underset{a}{\text{máx}} Q(s_{t+1}, a|\theta_t)|\theta_t^*\right) \tag{B.5}$$

Esta aproximación conduce a aprendizajes más estables y rápidos.

### D3QN

Además del sobreoptimismo de las estimaciones, las redes anteriores tienen amplias limitaciones en cuanto a la ineficiencia de los datos de entrenamiento, de la memoria externa o del uso de los estados. Se ha implementado una *Dueling Double Deep Q-Network* a partir de la red DQN original, pese a que existían otras alternativas como A3C, DDPG (*Deep Deterministic Policy Gradient*) o *Noisy Nets Exploration*, una interesante aproximación donde se añade ruido paramétrico entrenable a la red a la vez que se sigue una *greedy policy* (Fortunato et al., 2017).

De nuevo, la red recibe las medidas láser, que pasan por capas densas con activación LeakyReLU y por una capa de salida densa con activación lineal. Los *Q-values* de la función objetivo se calculan de forma dual empleando las redes online y objetivo, ver ec. (B.5). Además, se han realizado otras modificaciones sobre la red inicial:

- Arquitectura *dueling*: los *Q-values* se calculan ahora partir de $V(s)$ (el valor de estar en un estado) y $A(s,a)$ (la ventaja de tomar una acción en un estado con respecto a otras acciones), que se estiman de forma separada en la red de forma que ésta puede además aprender el valor de cada estado sin necesidad de aprender el efecto de cada acción en dicho estado. Este hecho es particularmente útil cuando no todas las acciones afectan forma significativa en el entorno (Wang et al., 2015). Tras la capa de estimación de $V(s)$ y $A(s,a)$, hay una capa de agregación que no hace la suma directa de ambas para obtener $Q(s,a)$, puesto que concurriría en un problema del algoritmo *backpropagation* al existir más de un posible valor para $V(s)$ y $A(s,a)$ dado $Q(s,a)$. Esta operación es la siguiente:

$$Q(s,a,\theta) = \underbrace{V(s,\theta)}_{\substack{\text{Value}\\ \text{stream}}} + \underbrace{A(s,a,\theta) - \frac{1}{\dim(A)}\sum_{a'} A(s,a',\theta)}_{\text{Advantage stream}} \tag{B.6}$$

- *Prioritised Experience Replay* (Schaul et al., 2015): ahora los lotes de información extraidos de la memoria en cada aprendizaje no son escogidos de forma aleatoria sino con un cierto criterio para evitar correlaciones temporales. Cuando una tupla $(s_t, a_t, R_t, s_{t+1}, e_t)$ se va a introducir en la

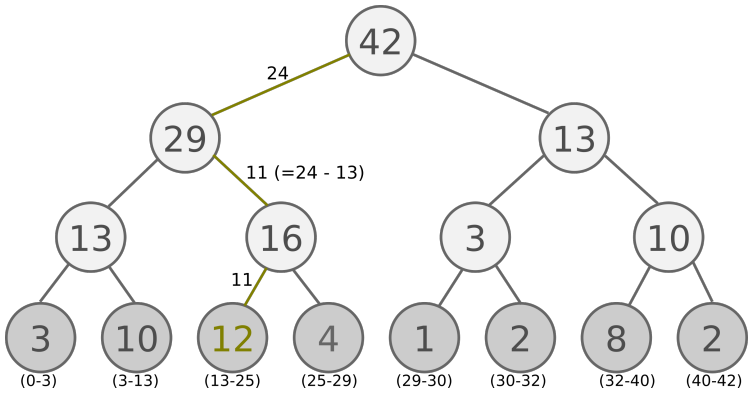


**Figura B.1:** Ejemplo de búsqueda del nodo del que se extrae la tupla con $s = 24$. De (Janisch, 2016).

memoria, se calcula el error TD (i.e. $Q - Q_{target}$) y se asigna una prioridad a dicha tupla en función del mismo:

$$\mathfrak{p} = (TD_{error} + \varepsilon)^{\alpha} \tag{B.7}$$

Siendo $\varepsilon$ una constante pequeña para evitar la prioridad 0, y $0 \leq \alpha \leq 1$. Todas las tuplas son ordenadas en un arbol binario con estructura `sum-tree`, de forma que el valor de cada nodo del arbol contiene la suma de las prioridades de sus hijos. Para extraer un lote de $k$ muestras, se divide $\sum \mathfrak{p}$ en $k$ intervalos, escogiendo de cada uno un valor $s$ según una distribución uniforme. Con este valor se recorre el árbol hacia abajo hasta encontrar el nodo del que se extraerá la tupla, según:

```
def retrieve(n,s): # Search can start at n = 0 (top node)
  if n is leaf_node: # Tuples are stored in leave nodes
    return n
  if n.left.value >= s: # Move down to left child
    return retrieve(n.left,s)
  else: # Move down to right child & update s
    return retrieve(n.right, s-n.left.value)
```

Como puede verse en la figura B.1, aquellos nodos con mayor prioridad tiene más probabilidad de ser seleccionados (última fila de la imagen). De esta forma la red aprende de aquellas experiencias cuyas estimaciones más se desvian del valor real. Para garantizar que la red emplee todas las experiencias al menos una vez, las tuplas se introducen en la memoria con prioridad máxima, y cada vez que son extraídas, su prioridad en el árbol es actualizada con el nuevo error TD.

A continuación se muestra el algoritmo de este agente:

**Algoritmo 3** Dueling Double Deep Q-learning

1: Parameters: $\alpha \in (0, 1]$, $\gamma \in (0, 1]$, $\varepsilon \in (0, 1]$
2: Initialise Memory $\mathcal{M}$ with $N$ samples, $\delta \approx 0$ and $\alpha' \in (0, 1]$
3: Initialise Dueling Q-network with random parameters $\theta$
4: Initialise target Dueling Q-network with parameters $\theta^* \leftarrow \theta$
5: **for** episode $\leftarrow 1$ to max episodes **do**
6:  **while** $s_t$ not terminal **and** step $<$ max steps **do**
7:  Perceive $s_t$
8:  Select $a_t \leftarrow \pi(s_t|\varepsilon)$ ▷ $\varepsilon$-greedy policy
9:  Take $a_t$, get $r_t$ and perceive $s_{t+1}$ and $e_t$
10:  Store tuple $(s_t, a_t, r_t, s_{t+1}, e_t)$ in $\mathcal{M}$ with max $\mathfrak{p}$
11:  Sample minibatch of tuples $(s_i, a_i, r_i, s_{i+1}, e_i)$ from $\mathcal{M}$ ▷ PER
12:  **for** each tuple in minibatch **do**
13:  **if** $s_{i+1}$ is terminal **then**
14:  $Q_{target} \leftarrow r_i$
15:  **else**
16:  $Q_{target} \leftarrow r_i + \gamma Q(s_{i+1}, \arg\max_a Q(s_{i+1}, a|\theta)|\theta^*)$ ▷ DDQN
17:  **end if**
18:  $\Omega_i \leftarrow Q_{target} - Q(s_i, a_i|\theta)$ ▷ $\text{TD}_{\text{error}}$
19:  Update $\mathfrak{p}$ of i$^{\text{th}}$-tuple in $\mathcal{M}$ such that $\mathfrak{p}_i \leftarrow (\Omega_i + \delta)^{\alpha'}$
20:  **end for**
21:  Perform optimization on $\Omega^2$ w.r.t. $\theta$ ▷ Quadratic loss
22:  Every $U$ steps set $\theta^* \leftarrow \theta$ ▷ Target network update
23:  $s_t \leftarrow s_{t+1}$
24:  step $\leftarrow$ step+1
25:  **end while**
26:  $\varepsilon \leftarrow \varepsilon * \varepsilon_{decay}$
27: **end for**

Otras mejoras pasarían por cambiar la función de pérdida cuadrática por la de Huber o la regla $\varepsilon - greedy$ por la de Boltzmann, sustituyendo las líneas 1, 8, 21 y 26 del algoritmo por, respectivamente:

1: Parameters: $\alpha \in (0, 1]$, $\gamma \in (0, 1]$, $\tau \in [0.5, 5]$

8: **for** each action $a \in \mathcal{A}$
 $\mathbb{P}[a] \leftarrow \frac{e^{Q(s_t,a)/\tau}}{\sum_i e^{Q(s_t,a)/\tau}}$ ▷ Boltzmann policy
 **end for**
 Select $a_t \leftarrow \pi(s_t|\tau)$ with that probability distribution

21: Perform optimization on $\log(\cosh(\Omega))$ w.r.t. $\theta$ ▷ Huber loss

26: $\tau \leftarrow \tau * \tau_{decay}$