

# **APRENDIZAJE POR REFUERZO**

Aproximación de funciones de valor

---

Antonio Manjavacas

[manjavacas@ugr.es](mailto:manjavacas@ugr.es)

# CONTENIDOS

---

1. Predicción *on-policy* aproximada
2. Gradiente estocástico y métodos semi-gradiente
3. Métodos lineales
4. Aproximación de funciones no lineales mediante redes neuronales
5. Control *on-policy* aproximado
6. Recompensa media
7. Trabajo propuesto

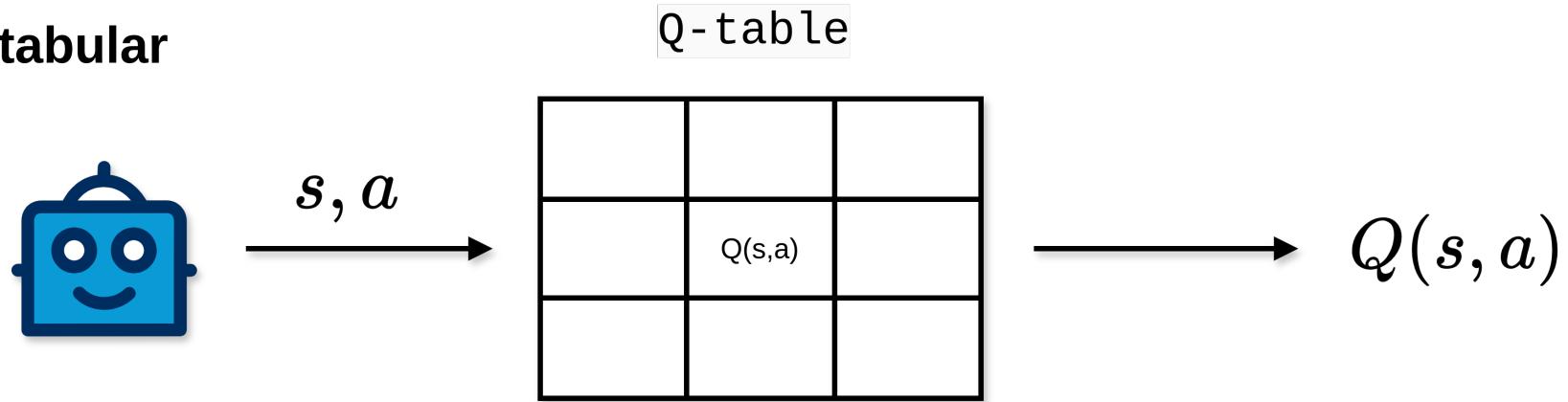
Hemos visto cómo los métodos **tabulares** presentaban una serie de limitaciones:

1. **Escalabilidad.** Si el espacio de estados o acciones es demasiado grande, las tablas de valores de  $v(s)$  o  $q(s, a)$  se vuelven inmanejables.
2. **Problemas continuos.** Los métodos tabulares no pueden manejar espacios de estados/acciones continuos, o bien requieren *discretizar* estos espacios.
3. **Generalización.** Los métodos tabulares tratan cada estado o par acción-estado de forma independiente. Esto les impide generalizar el conocimiento aprendido de un estado a otros similares.
  - Si tenemos dos estados cercanos con dinámicas o recompensas similares, los métodos tabulares no pueden sacar provecho de esta relación.
  - Esto *ralentiza* el aprendizaje, porque implica visitar todos los estados o pares acción-estado para actuar correctamente.

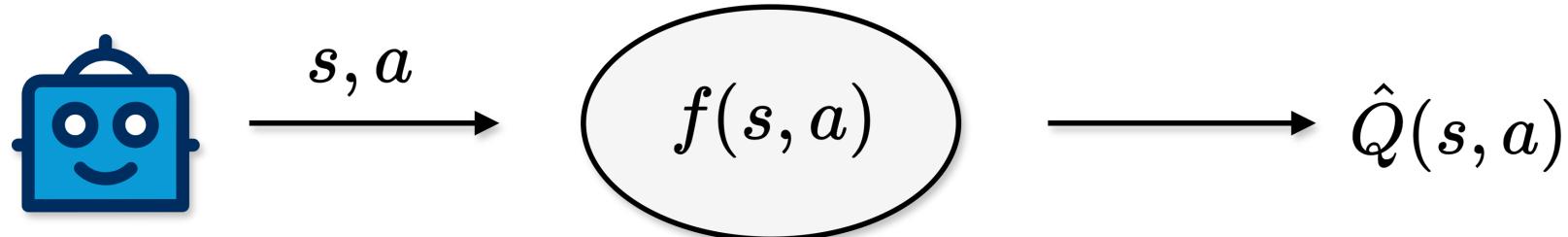
La idea principal tras los métodos *aproximados* es sustituir las **tablas** por **funciones**.

- Estas **funciones de valor aproximadas** toman como entrada un estado o par acción-estado, y nos devuelven su **valor aproximado** ( $v, q$ ).
- El **aprendizaje** de las funciones será otro aspecto importante a abordar, ya que se trata de **funciones parametrizadas**.

## Método tabular



## Función de valor aproximada



# PREDICCIÓN ON-POLICY APROXIMADA

---

**Idea principal:** buscamos obtener una **función estado–valor estimada**,  $\hat{v} \simeq v_\pi$ , a partir de experiencia generada siguiendo una política  $\pi$ .

- **Aprendizaje *on-policy*.** Utilizamos datos obtenidos mediante la interacción con el entorno para *aprender*  $\hat{v}$  de forma *on-policy*.
- La novedad es que la función de valor ya no se representa como una tabla, sino como una **función parametrizada**. Es decir, tenemos un vector de pesos  $w \in \mathbb{R}^d$  tal que:

$$\hat{v}(s, w) \approx v_\pi(s)$$

$$\hat{v}(s, \mathbf{w}) \approx v_\pi(s)$$

$\hat{v}(s, \mathbf{w})$  es el **valor aproximado** del estado  $s$  dado el vector de pesos  $\mathbf{w}$ .

- Ej.  $\hat{v}$  es la función representada por un polinomio de grado  $n$ , siendo  $\mathbf{w}$  el vector de coeficientes.
- Ej.  $\hat{v}$  es la función representada por una red neuronal, siendo  $\mathbf{w}$  el vector con los pesos de la red.

Normalmente, el número de pesos en  $w \in \mathbb{R}^d$  es **mucho menor que el número de estados**  $|\mathcal{S}|$ , ( $d \ll |\mathcal{S}|$ ) por lo que modificar un peso da lugar a una alteración del valor estimado de múltiples estados.

- De la misma forma, cuando el valor de un estado cambia, esta actualización repercute en el valor de otros estados.
- Este tipo de **generalización** permite un aprendizaje más *eficiente*, pero también potencialmente más complejo de realizar e interpretar.

La extensión de RL a la aproximación de funciones facilita su aplicación en problemas **parcialmente observables**.

Representaremos la **actualización del valor de un estado** de la siguiente forma:

$$s \mapsto u$$

donde  $s$  es el estado cuyo valor se actualiza, y  $u$  es el **valor objetivo** (*update target*) hacia el que se dirige el valor estimado de  $s$  (*backed-up value*).

- Cada actualización es un ejemplo del comportamiento esperado para la función de valor (dado un estado  $s$ , obtenemos una salida  $u$ , que es su valor objetivo).

- Actualización en **Monte Carlo**:

$$S_t \mapsto G_t$$

- Actualización en **TD(0)**:

$$S_t \mapsto R_{t+1} + \gamma \hat{v}(S_{t+1}, w_t)$$

- Actualización en ***n*-step TD**:

$$S_t \mapsto G_{t:t+n}$$

- Evaluación de la política en **programación dinámica**:

$$s \mapsto \mathbb{E}_\pi [R_{t+1} + \gamma \hat{v}(S_{t+1}, w) \mid S_t = s]$$

La actualización  $s \mapsto u$  denota que el valor estimado de  $s$  debería **acercarse o ser similar** al valor objetivo  $u$ .

- Esto refleja el **comportamiento entrada–salida** deseado para una función de valor.

Hasta ahora, las actualizaciones eran triviales:

- El valor de  $s$  en la tabla se modifica parcialmente en dirección a  $u$ , y el resto de estados quedan intactos.

Pero ahora permitimos que **métodos arbitrariamente complejos** lleven a cabo esta actualización.

- Además, actualizar  $s$  supone una **generalización** que da lugar a la modificación de los valores estimados de otros estados.

El aprendizaje de la función de valor es similar al **aprendizaje supervisado**.

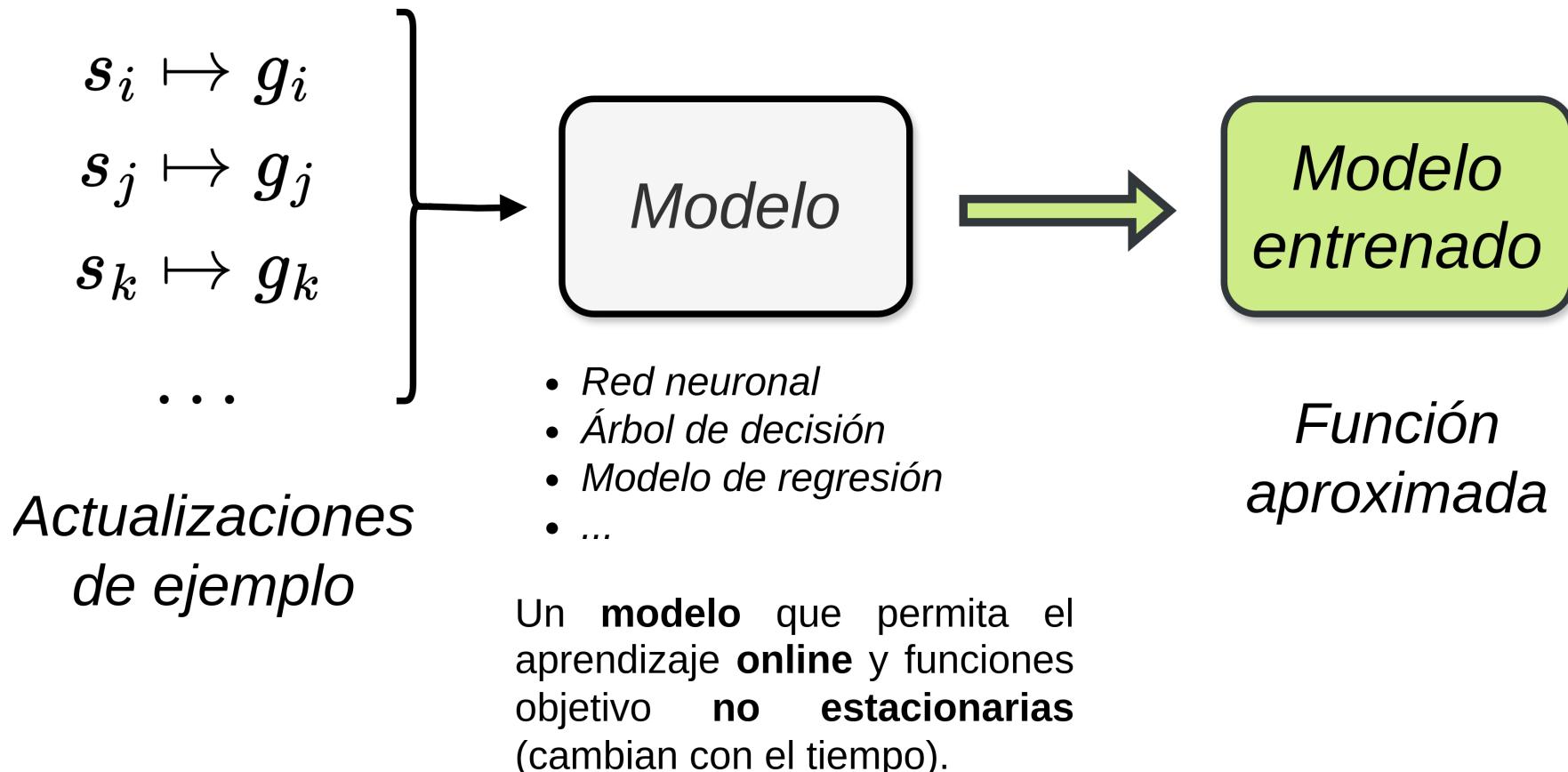
Ajustamos un **modelo** mediante ejemplos de entrada–salida:

## Aprendizaje supervisado

$(X_1, Y_1), (X_2, Y_2), (X_3, Y_3), \dots$

## En nuestro caso...

$(S_1, v_\pi(S_1)), (S_2, v_\pi(S_2)), \dots$



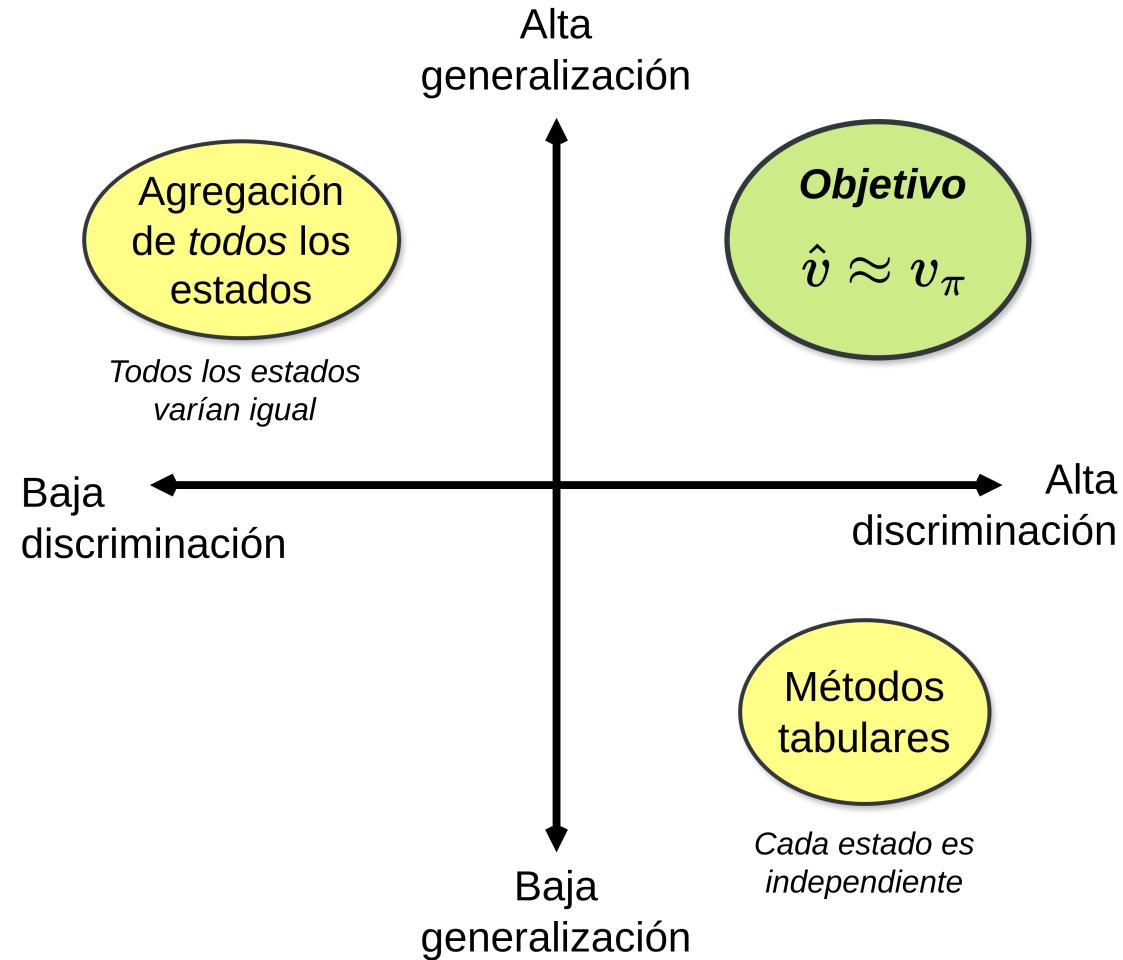
En los **métodos tabulares** no necesitábamos una **medida de calidad** de las predicciones.

- Cada estado podía llegar a su valor verdadero  $v_\pi(s)$  de forma independiente, sin afectar al valor de otros estados.

Sin embargo, empleando **aproximación de funciones**, la actualización del valor de un estado repercute sobre el resto.

- Dada esta interdependencia, no es posible obtener los **valores exactos** de todos los estados.

# Discriminación vs. generalización



# Error de predicción

El **error de estimación** mide la diferencia entre el valor real  $v_\pi(s)$  y el valor aproximado  $\hat{v}(s, \mathbf{w})$ .

- Se denomina **error de valor cuadrático medio** (**VE**, *mean squared value error*, MSVE) y se define tal que:

$$\overline{\text{VE}}(\mathbf{w}) = \sum_{s \in \mathcal{S}} [v_\pi(s) - \hat{v}(s, \mathbf{w})]^2$$

- Esta es nuestra **función objetivo**, es decir, aquella que trataremos de optimizar/minimizar.

Podemos decidir **qué estados son más relevantes**, de tal forma que el **error** en la estimación de su valor se tenga más en cuenta:

$$\underbrace{\mu(s) \geq 0}_{\text{importancia del error de estimación para el estado } s}, \sum_s \mu(s) = 1$$

De esta forma, tenemos:

$$\overline{\text{VE}}(\mathbf{w}) = \sum_{s \in \mathcal{S}} \mu(s) [v_\pi(s) - \hat{v}(s, \mathbf{w})]^2$$

$$\overline{\text{VE}}(\mathbf{w}) = \sum_{s \in \mathcal{S}} \mu(s) [v_\pi(s) - \hat{v}(s, \mathbf{w})]^2$$

- La raíz cuadrada de esta medida nos dice **cuánto difieren** las estimaciones de los valores reales.
- Ponderando por la importancia asignada al error de cada estado, según  $\mu(s)$ .

Una buena aproximación es que  $\mu(s)$  venga dado por la cantidad de tiempo invertido en  $s$ .

- Por ejemplo, definamos de la siguiente forma la **frecuencia de visita** a un estado  $s$ :

$$\eta(s) = h(s) + \sum_{s'} \eta(\bar{s}) \sum_a \pi(a \mid s') p(s \mid s', a), \quad \forall s, s' \in \mathcal{S}$$

donde:

- $\eta(s)$  es el número de *timesteps* que, de media, el agente pasa en el estado  $s$  durante un episodio, ya sea porque comienza en  $s$ , o porque transiciona desde otro estado  $s'$ .
- $h(s)$  es la probabilidad de que el episodio comience en el estado  $s$ .

Denominamos **distribución *on-policy***  $\mu(s)$  a la **fracción de tiempo** invertido en un estado  $s$ :

$$\mu(s) = \frac{\eta(s)}{\sum_{s'} \eta(s')}, \quad \forall s \in \mathcal{S}$$

Este valor indica cuánto se ha visitado  $s$  con respecto al resto de estados  $s' \in \mathcal{S}$  a lo largo de un episodio. Sirve para ponderar el **error**.

- $\mu(s)$  es un valor normalizado, tal que  $\sum_{s \in \mathcal{S}} \mu(s) = 1$

Empleamos  $\overline{\text{VE}}$  para ajustar progresivamente la **función de valor aproximada**  $\hat{v}(s, w)$ .

Un objetivo ideal en términos de  $\overline{\text{VE}}$  podría ser encontrar un **óptimo global**  $w^*$  tal que:

$$\overline{\text{VE}}(w^*) \leq \overline{\text{VE}}(w), \forall w$$

- Encontrar  $w^*$  puede ser viable en modelos sencillos, pero **no** en modelos complejos como redes neuronales o árboles de decisión.

La aproximación de funciones mediante modelos complejos puede converger en **óptimos locales**, tales que:

$$\overline{\text{VE}}(\boldsymbol{w}^*) \leq \overline{\text{VE}}(\boldsymbol{w}), \forall \boldsymbol{w} \text{ próximo a } \boldsymbol{w}^*$$

- Pueden ser soluciones subóptimas, aunque normalmente son **suficientes** para obtener una **buenas políticas**.
- Incluso hay problemas de RL donde las soluciones óptimas son inalcanzables.

# **GRADIENTE ESTOCÁSTICO Y MÉTODOS SEMI-GRADIENTE**

---

Consideramos que  $\hat{v}(s, \mathbf{w})$  es una **función diferenciable** con respecto al vector de pesos  $\mathbf{w}$ ,  $\forall s \in \mathcal{S}$ .

Nuestro objetivo será obtener  $\mathbf{w} \simeq \mathbf{w}^*$ , tal que  $\hat{v}(s, \mathbf{w}) \simeq v(s)$ .

$$\hat{v}(s, \mathbf{w})$$

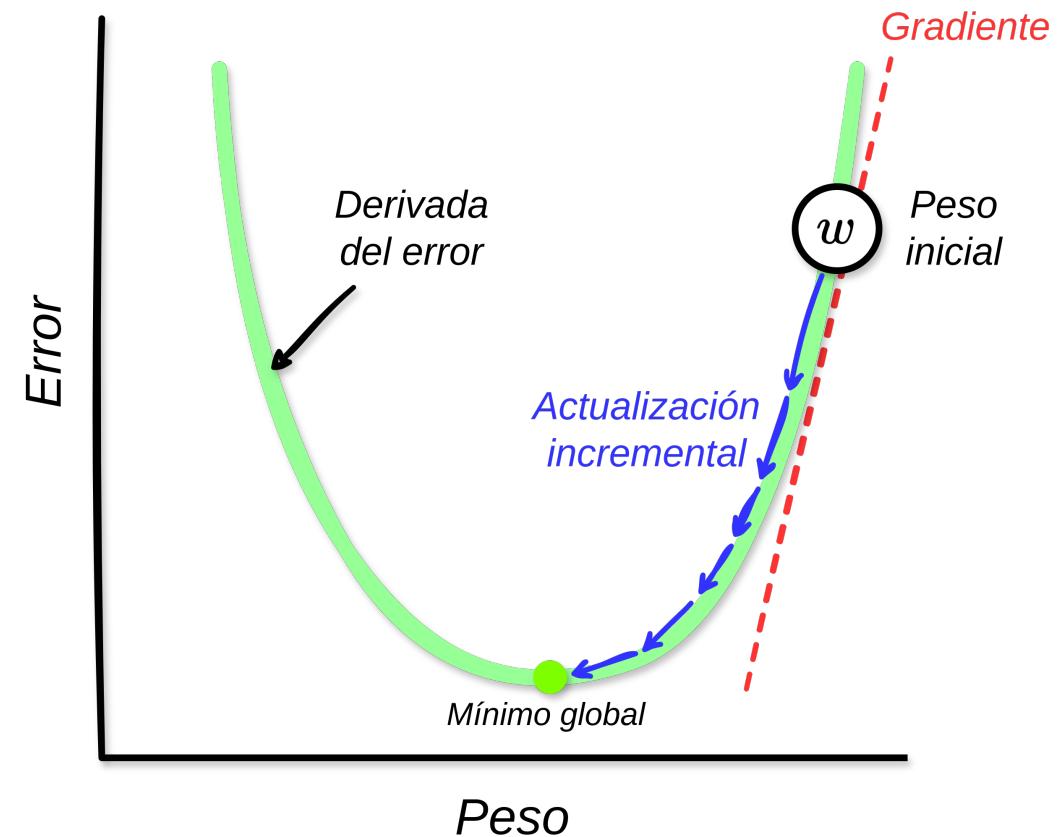
$$\mathbf{w} = \begin{pmatrix} w_1 \\ w_2 \\ \dots \\ w_n \end{pmatrix}$$

- Como inicialmente no contamos con un vector de pesos  $\mathbf{w}$  capaz de proporcionar el valor correcto para todos los estados, trataremos de ajustarlo de forma **iterativa**.

- El vector de pesos  $w$  se actualizará en cada *timestep*, por lo que emplearemos  $w_t$  para denotar los pesos del modelo en el instante  $t = 0, 1, 2, 3, \dots$
- Asumimos que los estados son seleccionados de acuerdo a la misma distribución,  $\mu$ , y que buscamos minimizar el error  $\overline{VE}$ .
- Emplearemos **gradiente estocástico descendente** (*stochastic gradient descent, SGD*) para llevar a cabo la optimización de  $w$ .

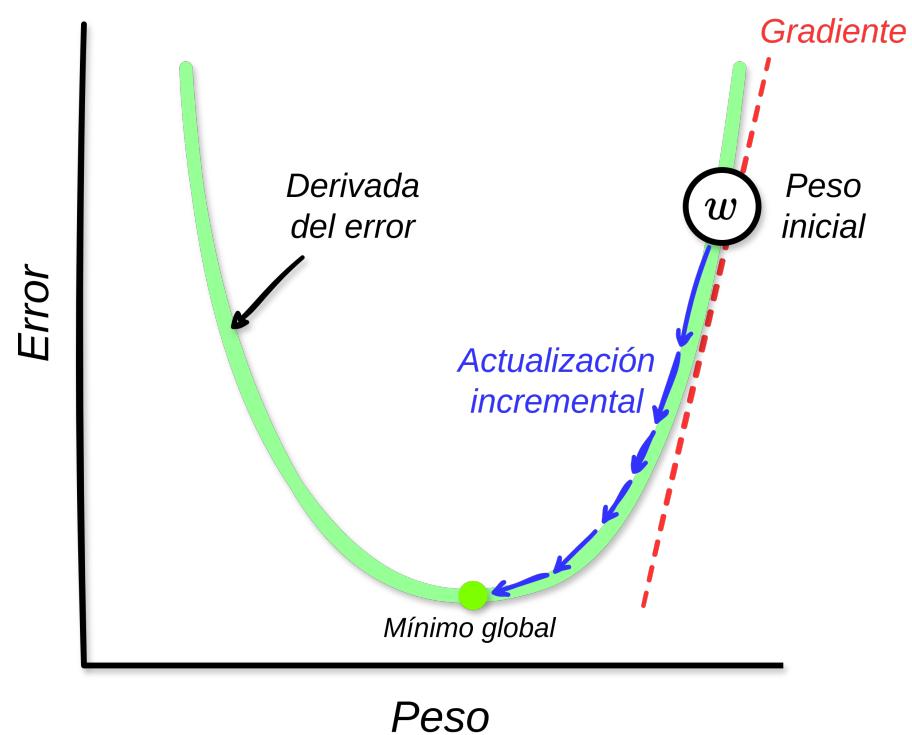
# Gradiente estocástico descendente

**SGD** es un algoritmo de optimización empleado para minimizar una función de **error** (pérdida, *loss*) ajustando los parámetros  $w$  de un modelo.



- La **función de error**  $E(w)$  indica cómo de buena es la predicción del modelo con respecto al valor real.
- El **gradiente de la función de error** con respecto a los pesos es un vector que apunta en la **dirección del mayor incremento del error**:

$$\nabla E(w) = \left[ \frac{\partial E}{\partial w_1}, \frac{\partial E}{\partial w_2}, \dots, \frac{\partial E}{\partial w_d} \right]^T$$



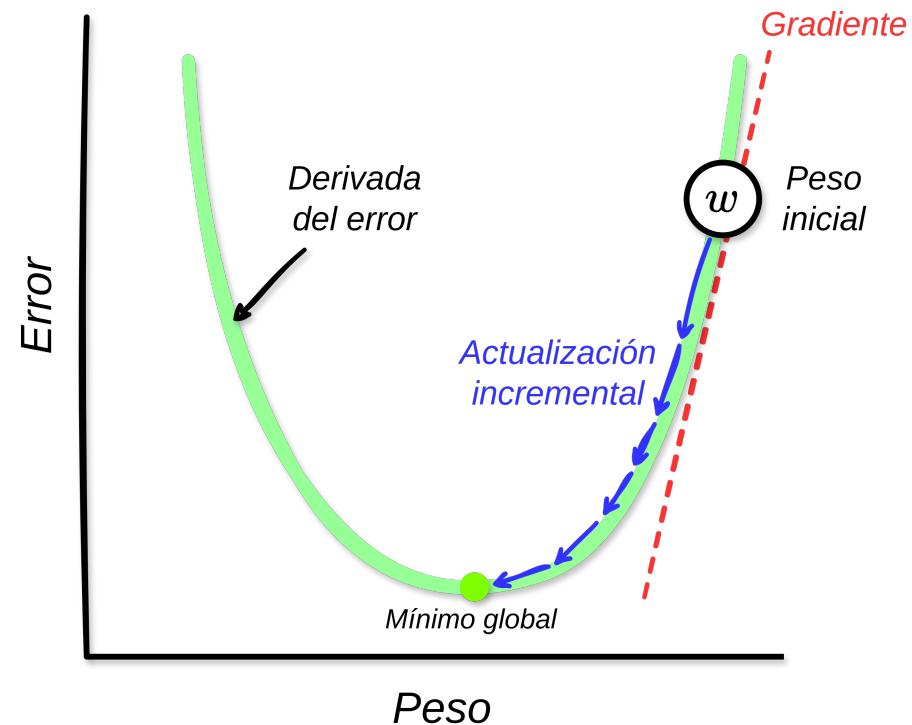
Para minimizar el error, nos moveremos en la **dirección opuesta al gradiente**.

- Esta es la **dirección en la que el error disminuye más rápidamente**.

La actualización de los pesos se lleva a cabo de la siguiente forma:

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha \nabla E(\mathbf{w})$$

donde  $\alpha$  es la *tasa de aprendizaje* que controla el *tamaño* de la actualización.



Aplicado a nuestro  
problema...

El proceso general empleado para aproximar  $w$  es el siguiente:

1. En cada *timestep*, observamos un ejemplo  $S_t \mapsto v_\pi(S_t)$ , dado un estado aleatorio  $S_t$  y su valor verdadero  $v_\pi(S_t)$ .
2. Comparamos  $v_\pi(S_t)$  con el valor predicho por  $\hat{v}(S_t, w)$ , obteniendo así el error de predicción  $\overline{\text{VE}}$ .
3. Ajustamos  $w$  en una pequeña fracción,  $\alpha$ . La dirección de dicho ajuste vendrá guiada por la minimización de  $\overline{\text{VE}}$  según el gradiente:

$$\begin{aligned} \mathbf{w}_{t+1} &= \mathbf{w}_t - \frac{1}{2}\alpha \nabla [v_\pi(S_t) - \hat{v}(S_t, \mathbf{w}_t)]^2 \\ &= \mathbf{w}_t - \alpha [v_\pi(S_t) - \hat{v}(S_t, \mathbf{w}_t)] \nabla \hat{v}(S_t, \mathbf{w}_t) \end{aligned}$$

\* Consideramos  $\overline{\text{VE}}(w) = \frac{1}{2}[v_\pi(S_t) - \hat{v}(S_t, w)]^2$ . El valor  $\frac{1}{2}$  es opcional, y se emplea para simplificar derivadas.

$$w \leftarrow w - \alpha \left[ \underbrace{v_\pi(S_t) - \hat{v}(S_t, w_t)}_{\text{Tamaño del error}} \right] \underbrace{\nabla \hat{v}(S_t, w_t)}_{\text{Dirección de la actualización}}$$

- En nuestro caso, el vector de derivadas parciales es:

$$\nabla \overline{VE}(\mathbf{w}) = \begin{pmatrix} \frac{\partial \overline{VE}}{\partial w_1} \\ \frac{\partial \overline{VE}}{\partial w_2} \\ \vdots \\ \frac{\partial \overline{VE}}{\partial w_d} \end{pmatrix}$$

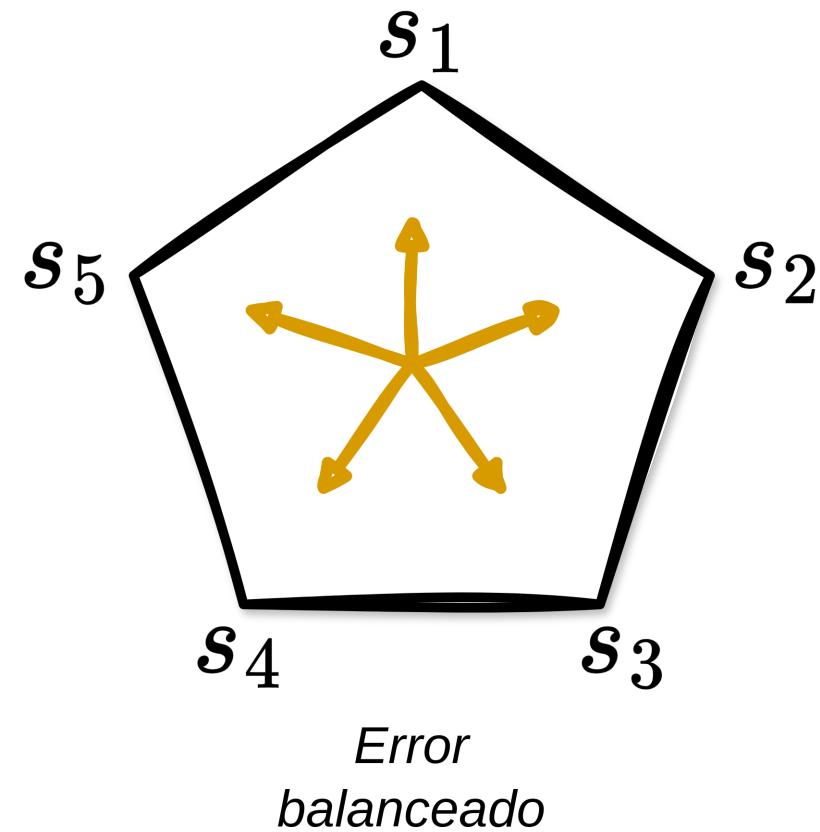
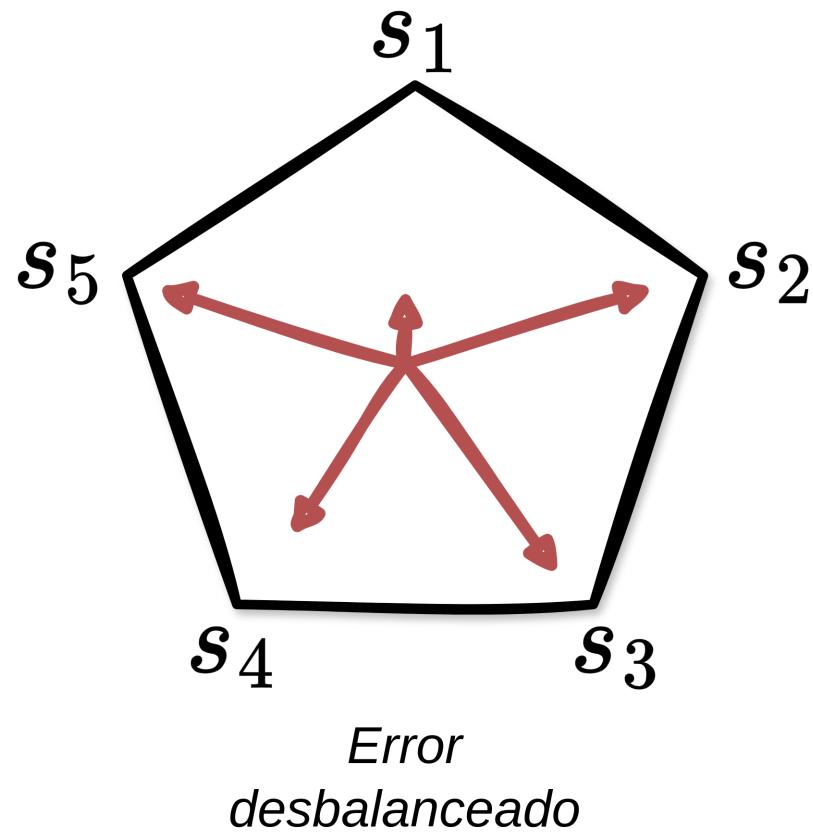
- SGD es **estocástico** porque el estado observado en cada *timestep* es aleatorio.
- A lo largo de múltiples observaciones, se van sucediendo observaciones y pequeños pasos hacia la reducción del error  $\overline{VE}$ , mejorando la aproximación de la función de valor estimada  $\hat{v}$ .

La reducción del error (actualización de  $w$ ) se realiza en pasos pequeños porque **no** esperamos encontrar una solución perfecta, sino más bien una aproximación que suponga un **error balanceado** para todos los estados.

- Si corrigiésemos completamente el error para el estado actual en cada *timestep*, nunca obtendríamos dicho balance.
- Perderíamos capacidad de **generalización**.

No obstante, para asegurar la convergencia de SGD, debemos permitir que  $\alpha$  (*step size*) **se reduzca con el tiempo**.

- Si esto ocurre, SGD **siempre convergerá en un óptimo local**.



¿Y si no tenemos  $v_\pi$ ?



¿Qué ocurre si no contamos con  $v_\pi$  para calcular el **error**?

- ¿Podríamos actualizar correctamente  $w$  si en vez de utilizar  $v_\pi$  para calcular  $\overline{VE}$  utilizamos un **valor aproximado**?

Es decir...

¿Qué ocurriría si el **valor objetivo** (*target output*)  $U_t \in \mathbb{R}$  tal que  $S_t \mapsto U_t$ , no es el valor exacto de  $v_\pi(S_t)$ ?

- Supongamos que es un **valor aleatorio aproximado**, similar a  $v_\pi(S_t)$  pero con cierto ruido, o un valor estimado obtenido mediante *bootstrapping* a partir de  $\hat{v}$ .

En ese caso, la actualización de  $w$  no será exacta, porque no conocemos  $v_\pi(S_t)$ .

Igualmente, podemos aproximarla sustituyendo  $v_\pi(S_t)$  por  $U_t$ , dando lugar a la siguiente actualización de  $w$ :

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \alpha [U_t - \hat{v}(S_t, \mathbf{w}_t) \nabla \hat{v}(S_t, \mathbf{w}_t)]$$

siendo  $U_t$  una **estimación no sesgada** de  $v_\pi(S_t)$ .

Si  $\mathbb{E}[U_t \mid S_t = s] = v_\pi(s), \forall t = 0, 1, 2, \dots$   
se garantiza que  $\mathbf{w}_t$  **convergerá en un óptimo local** empleando SGD.

En resumen, estamos sustituyendo  $v_\pi$  en:

$$S_1 \mapsto v_\pi(S_1), S_2 \mapsto v_\pi(S_2), \dots$$

por un **objetivo aproximado** tal que:

**MC**

$$\begin{aligned} S_1 &\mapsto G_1, \\ S_2 &\mapsto G_2, \\ &\dots \end{aligned}$$

**TD**

$$\begin{aligned} S_1 &\mapsto R_2 + \gamma \hat{v}(S_2, \mathbf{w}), \\ S_2 &\mapsto R_3 + \gamma \hat{v}(S_3, \mathbf{w}), \\ &\dots \end{aligned}$$

# Estimación Monte Carlo de $\hat{v}$

**Monte Carlo** aproxima  $\hat{v}$  con garantías porque su *target*  $U_t = G_t$  es una estimación **no sesgada** (*unbiased target*) de  $v_\pi$ . Es decir, estamos sustituyendo:

$$w \leftarrow w + \alpha[v_\pi(S_t) - \hat{v}(S_t, w)]\nabla\hat{v}(S_t, w)$$

por:

$$w \leftarrow w + \alpha[G_t - \hat{v}(S_t, w)]\nabla\hat{v}(S_t, w)$$

teniendo en cuenta que

✓  $v_\pi(s) = \mathbb{E}_\pi[G_t \mid S_t = s]$

La estimación del gradiente sigue siendo la misma.

- Utilizar retornos estimados obtenidos a través de *sampling* no afecta al objetivo del descenso del gradiente.

$$w \leftarrow w + \alpha [G_t - \hat{v}(S_t, w)] \nabla \hat{v}(S_t, w)$$

**Idea intuitiva:** sustituyo el valor real  $v(S_t)$  de cada estado por la recompensa total  $G_t$  que espero obtener en base a mi experiencia.

- Como nos basamos en experiencia obtenida a partir de trayectorias reales, decimos que  $U_t = G_t$  es una estimación **no sesgada** de la función de valor.

La idea general es la siguiente:

### Gradient MC

1. Generar episodio  $S_0, A_0, R_1, S_1, A_1, R_2, \dots$  siguiendo  $\pi$ .
2. Para cada *step* del episodio, actualizar  $w$  tal que:

$$w \leftarrow w + \alpha[G_t - \hat{v}(S_t, w)\nabla\hat{v}(S_t, w)]$$

basándonos en retornos muestrados ( $G_t$ ).

### Gradient Monte Carlo Algorithm for Estimating $\hat{v} \approx v_\pi$

Input: the policy  $\pi$  to be evaluated

Input: a differentiable function  $\hat{v} : \mathcal{S} \times \mathbb{R}^d \rightarrow \mathbb{R}$

Algorithm parameter: step size  $\alpha > 0$

Initialize value-function weights  $\mathbf{w} \in \mathbb{R}^d$  arbitrarily (e.g.,  $\mathbf{w} = \mathbf{0}$ )

Loop forever (for each episode):

Generate an episode  $S_0, A_0, R_1, S_1, A_1, \dots, R_T, S_T$  using  $\pi$

Loop for each step of episode,  $t = 0, 1, \dots, T - 1$ :

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha [G_t - \hat{v}(S_t, \mathbf{w})] \nabla \hat{v}(S_t, \mathbf{w})$$

# Estimación de $\hat{v}$ mediante *bootstrapping*



¿Y podríamos combinar TD con aproximación de funciones?

En *Gradient MC*, la actualización de los pesos se hacía conforme a  $U_t = G_t$ , pero realmente el *target* podría ser cualquier otra estimación del valor.

Por ejemplo...

- ***n-step return:***

$$U_t = G_{t:t+n}$$

- ***DP target:***

$$U_t = \sum_{a,s',r} \pi(a|S_t)p(s', r | S_t, a)[r + \gamma \hat{v}(s', w_t)]$$

¿Qué ocurre si nuestro objetivo  $U_t$  es una estimación basada en otros valores estimados? ➔ **bootstrapping**

Empleando métodos basados en *bootstrapping*, se actualiza  $w$  a partir de estimaciones.

- Por ejemplo, para **semi-gradient TD(0)**, el valor objetivo (*bootstrap target*) es:

$$U_t = R_{t+1} + \gamma \hat{v}(S_{t+1}, w)$$

- El objetivo depende de  $w$ , por lo que **varía constantemente**, cada vez que se actualiza  $w$ .
- Por eso decimos que el objetivo está **sesgado**.

$$U_t = R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w})$$

Si el objetivo está **sesgado**, **no se produce un verdadero descenso del gradiente**.

Debido a esto, a los métodos que estiman  $\hat{v}$  mediante *bootstrapping* se les denomina **métodos semi-gradiente** (*semi-gradient methods*).

- No podemos garantizar la convergencia en un óptimo local.
- No obstante, convergen en la mayoría de los casos.
- El sesgo se reduce a medida que las estimaciones mejoran.

## Semi-gradient TD(0) for estimating $\hat{v} \approx v_\pi$

Input: the policy  $\pi$  to be evaluated

Input: a differentiable function  $\hat{v} : \mathcal{S}^+ \times \mathbb{R}^d \rightarrow \mathbb{R}$  such that  $\hat{v}(\text{terminal}, \cdot) = 0$

Algorithm parameter: step size  $\alpha > 0$

Initialize value-function weights  $\mathbf{w} \in \mathbb{R}^d$  arbitrarily (e.g.,  $\mathbf{w} = \mathbf{0}$ )

Loop for each episode:

    Initialize  $S$

    Loop for each step of episode:

        Choose  $A \sim \pi(\cdot | S)$

        Take action  $A$ , observe  $R, S'$

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha [R + \gamma \hat{v}(S', \mathbf{w}) - \hat{v}(S, \mathbf{w})] \nabla \hat{v}(S, \mathbf{w})$$

$S \leftarrow S'$

    until  $S$  is terminal

Los métodos semi-gradiente no convergen de forma tan robusta, pero ofrecen ciertas **ventajas** que los hacen preferibles:

-  **Velocidad de aprendizaje.** Las actualizaciones presentan una menor varianza.
-  **Aprendizaje *online*.** No es necesario esperar a obtener  $G_t$ .
-  **Problemas continuados.** Son aplicables a problemas sin un *final* de episodio.

👉 En la práctica el aprendizaje rápido es más útil que el rendimiento asintótico, lo que hace que TD sea mejor que MC en la mayoría de problemas.

Si ampliamos TD(0) al caso *n-step* tendremos el siguiente algoritmo...

## $n$ -step semi-gradient TD for estimating $\hat{v} \approx v_\pi$

Input: the policy  $\pi$  to be evaluated

Input: a differentiable function  $\hat{v} : \mathcal{S}^+ \times \mathbb{R}^d \rightarrow \mathbb{R}$  such that  $\hat{v}(\text{terminal}, \cdot) = 0$

Algorithm parameters: step size  $\alpha > 0$ , a positive integer  $n$

Initialize value-function weights  $\mathbf{w}$  arbitrarily (e.g.,  $\mathbf{w} = \mathbf{0}$ )

All store and access operations ( $S_t$  and  $R_t$ ) can take their index mod  $n + 1$

Loop for each episode:

    Initialize and store  $S_0 \neq \text{terminal}$

$T \leftarrow \infty$

    Loop for  $t = 0, 1, 2, \dots$ :

        If  $t < T$ , then:

            Take an action according to  $\pi(\cdot | S_t)$

            Observe and store the next reward as  $R_{t+1}$  and the next state as  $S_{t+1}$

            If  $S_{t+1}$  is terminal, then  $T \leftarrow t + 1$

$\tau \leftarrow t - n + 1$  ( $\tau$  is the time whose state's estimate is being updated)

        If  $\tau \geq 0$ :

$G \leftarrow \sum_{i=\tau+1}^{\min(\tau+n, T)} \gamma^{i-\tau-1} R_i$

            If  $\tau + n < T$ , then:  $G \leftarrow G + \gamma^n \hat{v}(S_{\tau+n}, \mathbf{w})$   $(G_{\tau:\tau+n})$

$\mathbf{w} \leftarrow \mathbf{w} + \alpha [G - \hat{v}(S_\tau, \mathbf{w})] \nabla \hat{v}(S_\tau, \mathbf{w})$

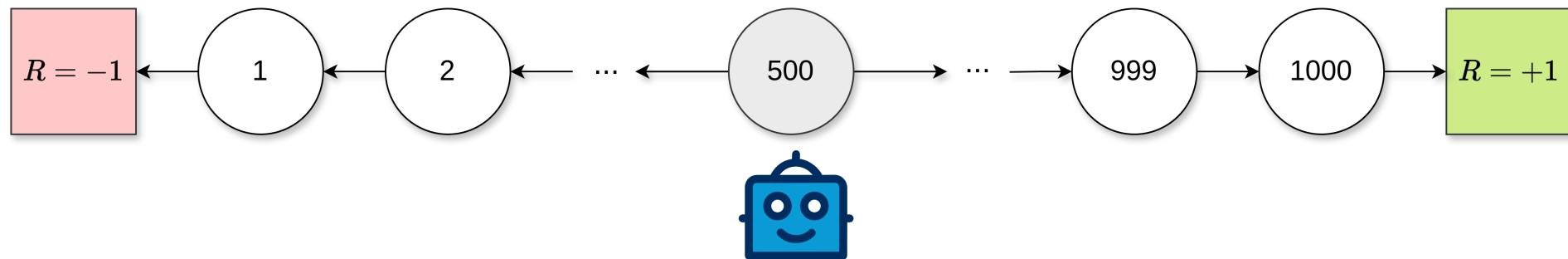
    Until  $\tau = T - 1$

# Agregación de estados

Cabe destacar el concepto de **agregación de estados** (*state aggregation*).

- Se trata de una técnica que permite generalizar la aproximación de funciones, de tal forma que los estados **se agrupan** de acuerdo a un mismo valor estimado / vector de pesos  $w$ .
  - El valor de cada estado se actualiza junto al valor de los estados del mismo grupo.
  - Es un caso especial de SGD donde el gradiente  $\nabla \hat{v}(S_t, w)$  solo considera los estados en el grupo de  $S_t$  y se ignora para el resto.
-  Simplifica el número de parámetros del modelo.
-  Necesario establecer un criterio de agrupamiento. Puede perderse información.

Consideremos el siguiente ejemplo (*random walk*), con  $\pi = \{\rightarrow : 0.5, \leftarrow : 0.5\}$ ,  $\gamma = 1$ .

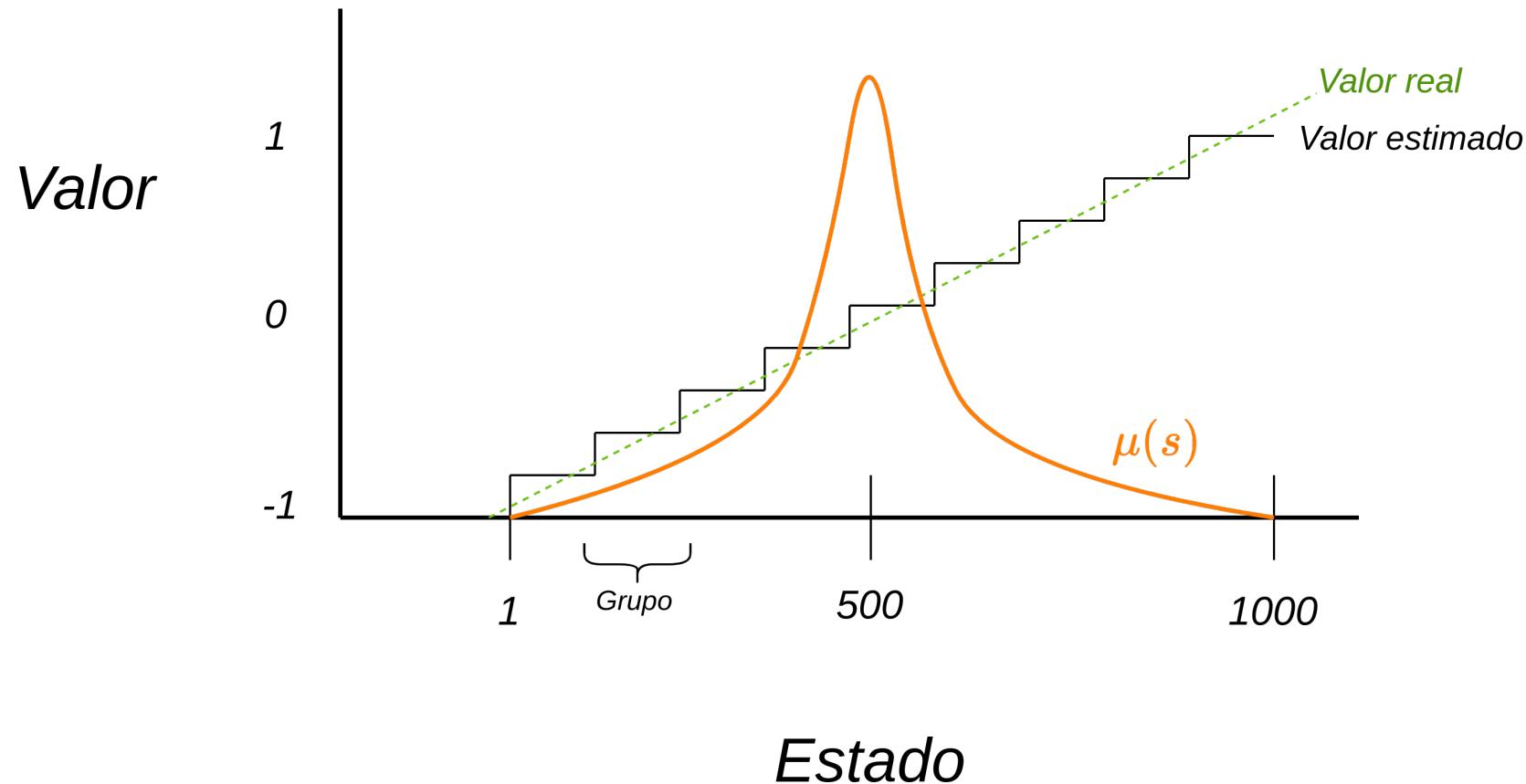


Cada **acción** supone saltar a un estado aleatorio entre los 100 estados inmediatamente anteriores/posteriores.

Por ejemplo, desde el estado 500, la acción  $\rightarrow$  podría llevar al agente a cualquier estado en  $[501, 600]$ .

- Podemos agrupar los estados de 100 en 100, dando lugar a 10 grupos.

Tras  $N$  episodios, estos son los resultados obtenidos...



# Pregunta...

? ¿En qué se diferencian?

**Discretización** del espacio de estados

**Agregación** de estados

## Discretización

**División** de un espacio continuo en valores discretos.

- Ej. Temperatura  $\in \mathbb{R} \rightarrow \{\text{baja, media, alta}\}$ .

## Agregación

Agrupamiento de las **actualizaciones** de múltiples estados.

- Ej. ubicaciones cercanas en un mapa se consideran el mismo estado y se actualizan igual.

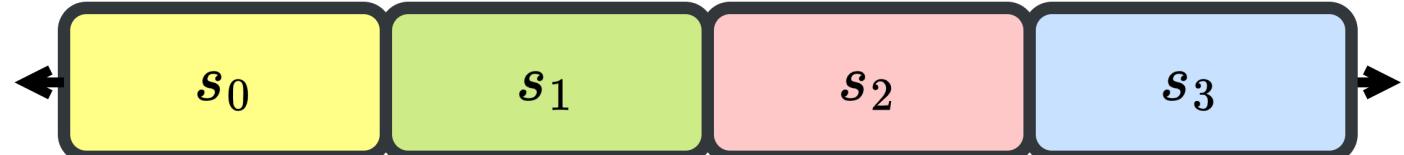
Estados infinitos  
espacio de estados continuo



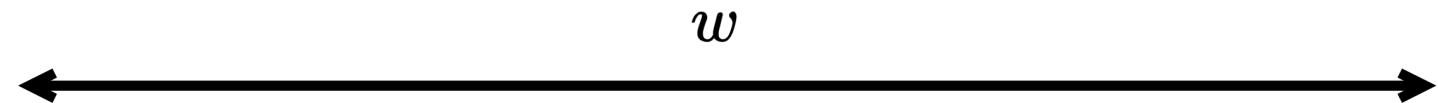
*Discretización*



Estados finitos  
espacio de estados discreto



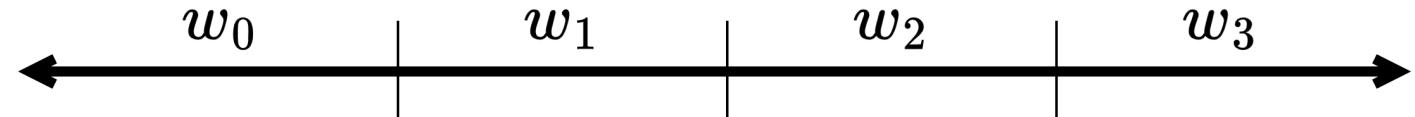
Estados infinitos  
espacio de estados continuo



*Agregación  
de estados*



Estados infinitos  
espacio de estados continuo



# MÉTODOS LINEALES

---

Un caso relevante en aproximación de funciones es aquel en el cual  $\hat{v}(\cdot, w)$  es una **función lineal** de  $w$ .

Esto es, para cada estado  $s$  existe un vector de valores  $x$  tal que:

$$x(s) = \begin{pmatrix} x_1(s) \\ x_2(s) \\ \dots \\ x_d(s) \end{pmatrix}$$

Este vector **contiene el mismo número de elementos** que  $w$ .

→ Como  $|x| = |w|$ , existe **una valor  $x_i$  por peso  $w_i$** .

Los métodos lineales aproximan la función estado–valor empleando el producto escalar entre  $w$  y  $x(s)$ :

$$\begin{aligned}\hat{v}(s, w) &= w^T x(s) \\ &= \sum_{i=1}^d w_i x_i(s)\end{aligned}$$

Denominamos a  $x(s)$  **vector de características** (*feature vector*) del estado  $s$ .

- Cada componente  $x_i(s)$  en  $x(s)$  es el valor de una función  $x_i : \mathcal{S} \rightarrow \mathbb{R}$ .

Podemos emplear SGD para aproximar funciones lineales. El gradiente en este caso sería:

$$\nabla \hat{v}(s, \mathbf{w}) = \mathbf{x}(s)$$

Y la actualización de pesos se reduce a:

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha [U_t - \hat{v}(S_t, \mathbf{w}_t)] \mathbf{x}(S_t)$$

En la aproximación de funciones lineales, sólo existe un óptimo, por lo que cualquier método que pueda converger en un **óptimo local** lo hará también en el **global**.

- **Monte Carlo** converge en el óptimo global de  $\overline{VE}$  siguiendo una aproximación de función lineal siempre que  $\alpha$  tienda a 0 con el tiempo.
- ¿Y **TD**?

# Aproximación de funciones lineales con TD

Partimos de la regla de actualización previamente presentada:

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha \delta_t \nabla \hat{v}(S_t, \mathbf{w})$$

donde  $\delta_t$  es el *TD-error* definido tal que:

$$\delta_t = \underbrace{\frac{R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w}) - \hat{v}(S_t, \mathbf{w})}{\text{TD-target}}}_{\text{TD-error}}$$

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha \delta_t \nabla \hat{v}(S_t, \mathbf{w})$$

- Los pesos  $w$  se ajustan en la dirección del *TD-error*  $\times$  el gradiente de la función de valor aproximada.

En el caso lineal, dicho gradiente es simplemente el **vector de características**:

$$\hat{v}(S_t, \mathbf{w}) = \mathbf{w}^T \mathbf{x}(S_t)$$

$$\nabla \hat{v}(S_t, \mathbf{w}) = \mathbf{x}(S_t)$$

Por tanto, tenemos la siguiente actualización de pesos:

$$\begin{aligned}\mathbf{w}_{t+1} &= \mathbf{w}_t + \alpha \delta_t \nabla \hat{v}(S_t, \mathbf{w}_t) \\ &= \mathbf{w}_t + \alpha \delta_t \mathbf{x}(S_t)\end{aligned}$$

- Los pesos se actualizan en la dirección del *TD-error*  $\times$  el vector de características. Esta actualización se denomina ***semi-gradiente lineal***.

La decisión sobre qué características emplear es, por tanto, muy importante.

TD lineal converge en el llamado **TD fixed-point** (ver Sutton & Barto p. 206 ).

- Es decir, **no optimiza de forma precisa** la función objetivo (el mínimo de  $\overline{VE}$ ), sino que el objetivo es ese **punto fijo**.

¿Cuál es la relación entre la solución encontrada por TD y la solución óptima global?

$$\underbrace{\overline{VE}(w_{TD})}_{\text{TD fixed-point}} \leq \frac{1}{1 - \gamma} \underbrace{\min_w \overline{VE}(w)}_{\text{valor óptimo}}$$

Dependiendo de la calidad de las **características (features)**, el *TD fixed-point* y el óptimo global serán más o menos cercanos.

¿Por qué es interesante la aproximación de funciones lineales?

1. TD lineal es una **generalización** de TD tabular y TD con agregación de estados.
2. Son simples de entender y analizar.
3. Eficaces si contamos con buenas características, que pueden venir dadas por un experto (*domain knowledge*).

**Sin embargo**, las funciones lineales también presentan ciertas limitaciones...

- Son incapaces de capturar **relaciones complejas entre variables**.
- Requieren de una **definición precisa y manual de las características** que definen un estado.

Esto motiva el uso de métodos **no-lineales** de aproximación de funciones, tales como las **redes neuronales**.

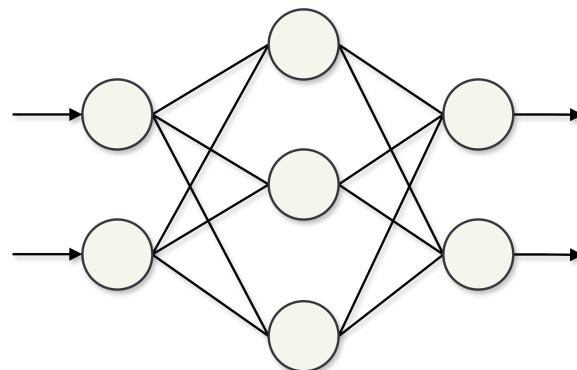
- Permiten capturar relaciones complejas entre variables / *features*.

# **APROXIMACIÓN DE FUNCIONES NO LINEALES MEDIANTE REDES NEURONALES**

---

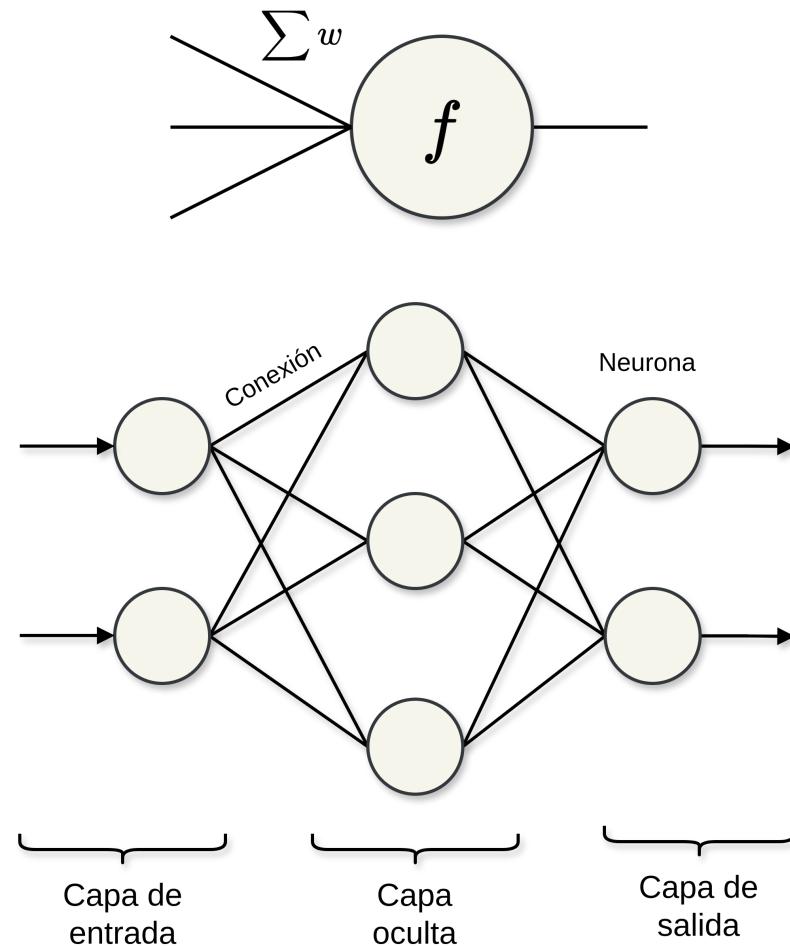
Las **redes neuronales artificiales** (NNs) son modelos matemáticos conexionistas inspirados en el funcionamiento del cerebro humano.

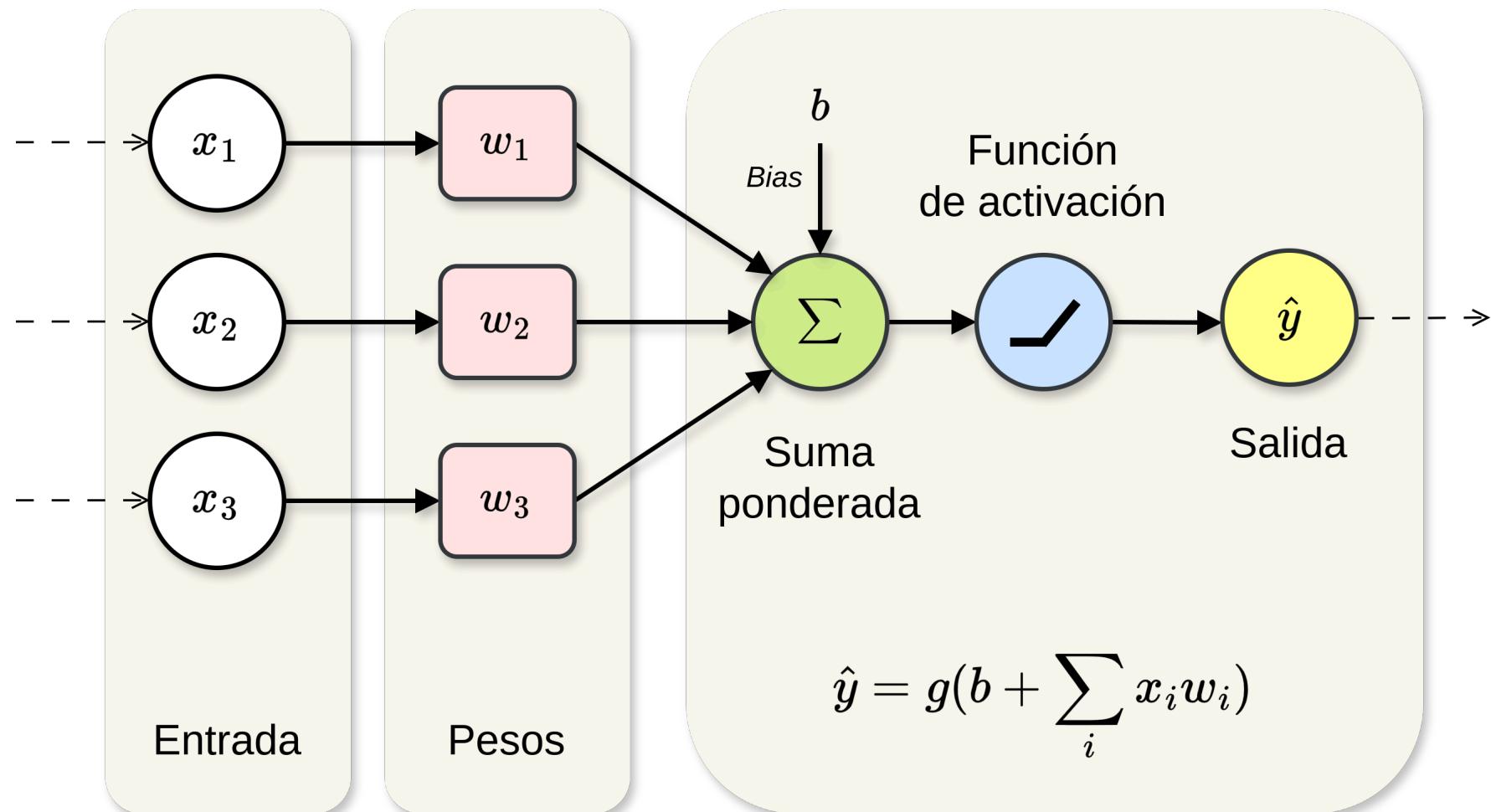
Se componen de una serie de **neuronas** interconectadas y organizadas en **capas**.



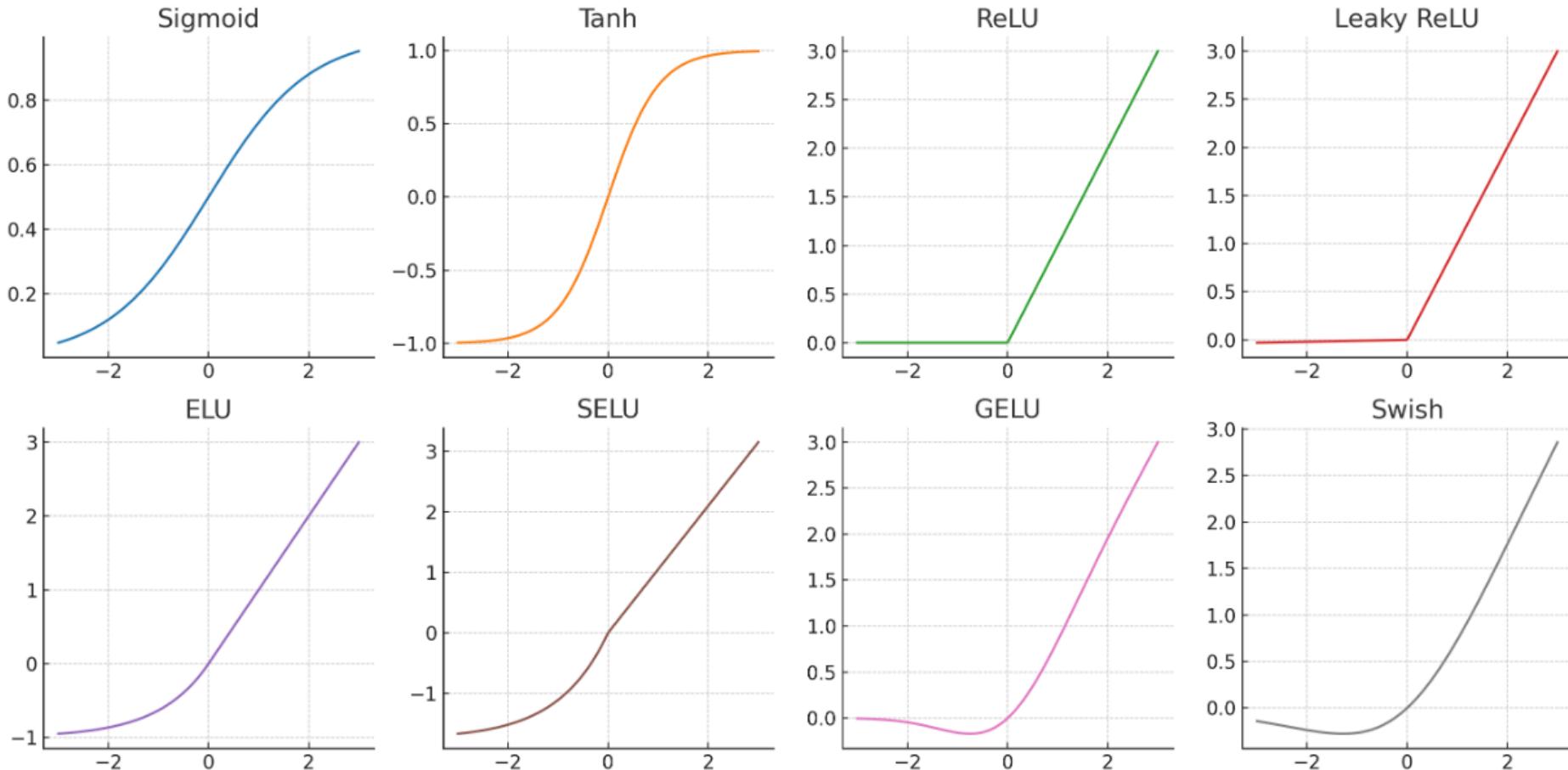
- Son **aproximadores generales de funciones**, especialmente útiles para ajustar **funciones no lineales**.

- Dentro de cada **neurona** hay una **función de activación**.
- La entrada de cada neurona es la suma ponderada de las salidas anteriores. Dicha ponderación viene dada por los **pesos** de las conexiones.
- Su arquitectura más común es la de **red *feedforward***.
- Distinguimos entre capa de **entrada**, capa de **salida** y capas **ocultas**.





# Funciones de activación



La salida de una red neuronal se obtiene como la combinación de

1. Los **valores** de las características (entrada de la red).
2. Los **pesos** asignados a las conexiones entre neuronas.
3. Los ajustes realizados por las **funciones de activación** a lo largo de las capas de la red.

- El **entrenamiento** de una red neuronal implica ajustar los pesos de las conexiones entre neuronas para minimizar progresivamente el error (**loss**) entre la predicción/salida de la red y el valor real esperado.
- En RL, el error puede ser  $\overline{VE}$ . Utilizamos el *TD-error* para aproximar  $v(s)$ .

Generalmente, el entrenamiento de NNs se realiza mediante **SGD**, integrado en el algoritmo de **backpropagation**.

## Algoritmo de *backpropagation*

1. Pasada hacia delante (*forward pass*) para obtener la salida de la red.
2. Calcular el error (*loss*).
3. Pasada hacia atrás (*backward pass*) para calcular los gradientes de la función de pérdida con respecto a los pesos.
4. Actualización de los pesos (ej. mediante SGD).

✓ Existen diferentes formas de **mejorar** el entrenamiento de una red neuronal...

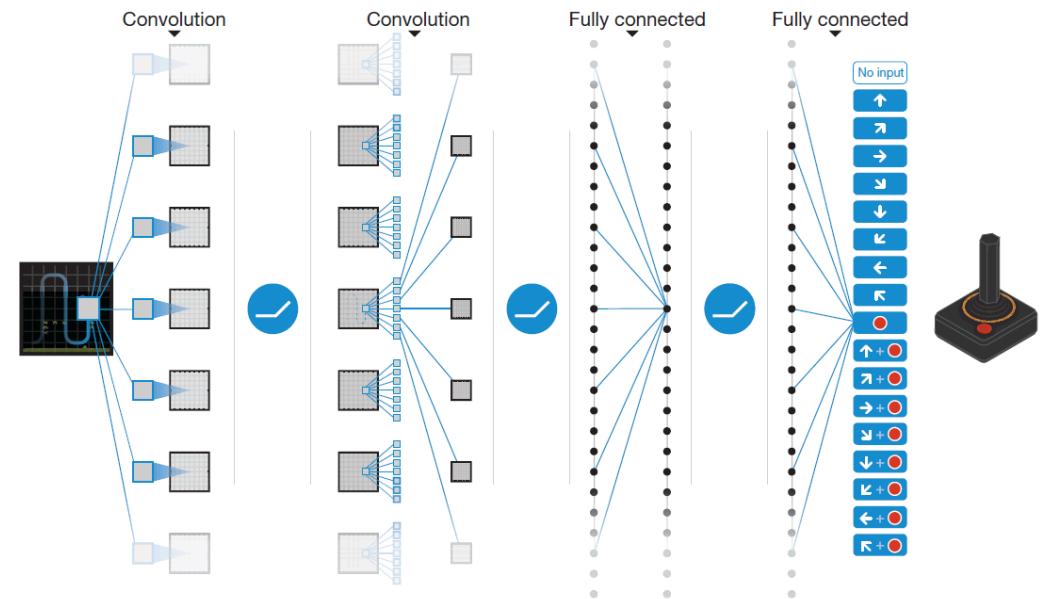
- Ajuste de hiperparámetros (ej. *step-size*, número de capas, número de neuronas por capa, funciones de activación, ...).
- Inicialización de pesos.
- *Momentum*.
- *Dropout*.
- Regularización, normalización.
- *Early stopping*.
- ...

✗ También debemos tener en cuenta algunos **problemas** típicos de las redes neuronales...

- Coste computacional.
- Sobreaprendizaje (*overfitting*).
- Explicabilidad.
- Calidad de datos.
- Desvanecimiento del gradiente.
- ....

💡 Aparte de las *feedforward*, existen múltiples tipos de redes neuronales. Su uso dependerá del problema que tratemos de abordar.

- Redes convolucionales (CNNs).
- Redes recurrentes (RNNs).
- GANs.
- LSTMs.
- *Transformers*.
- ...



# **CONTROL ON-POLICY APROXIMADO**

---

Hemos visto cómo estimar la función estado-valor  $v(s)$  haciendo uso de métodos aproximados.

## PREDICCIÓN → *semi-gradient TD(0)*

Nuestro **objetivo** ahora será aproximar la función **acción-valor**  $q(s, a)$ , de la cual derivaremos la política óptima.

## CONTROL → *semi-gradient SARSA*

Es decir, buscamos obtener **de forma *on-policy***:

$$\hat{q}(s, a, \mathbf{w}) \simeq q_*(s, a)$$

Aproximaremos la función acción-valor  $\hat{q} \simeq q_\pi$  empleando muestras de entrenamiento de la forma:  $S_t, A_t \mapsto U_t$ , donde  $U_t$  es nuestro **valor objetivo (update target)**.

- Es decir,  $U_t$  es una aproximación de  $q_\pi(S_t, A_t)$ .

En este caso, la actualización basada en descenso del gradiente para aproximar  $q(s, a)$  es:

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha[U_t - \hat{q}(s, a, \mathbf{w}_t)]\nabla\hat{q}(S_t, A_t, \mathbf{w})$$

Por ejemplo, la versión semi-gradiente de **SARSA** emplearía la siguiente actualización de pesos:

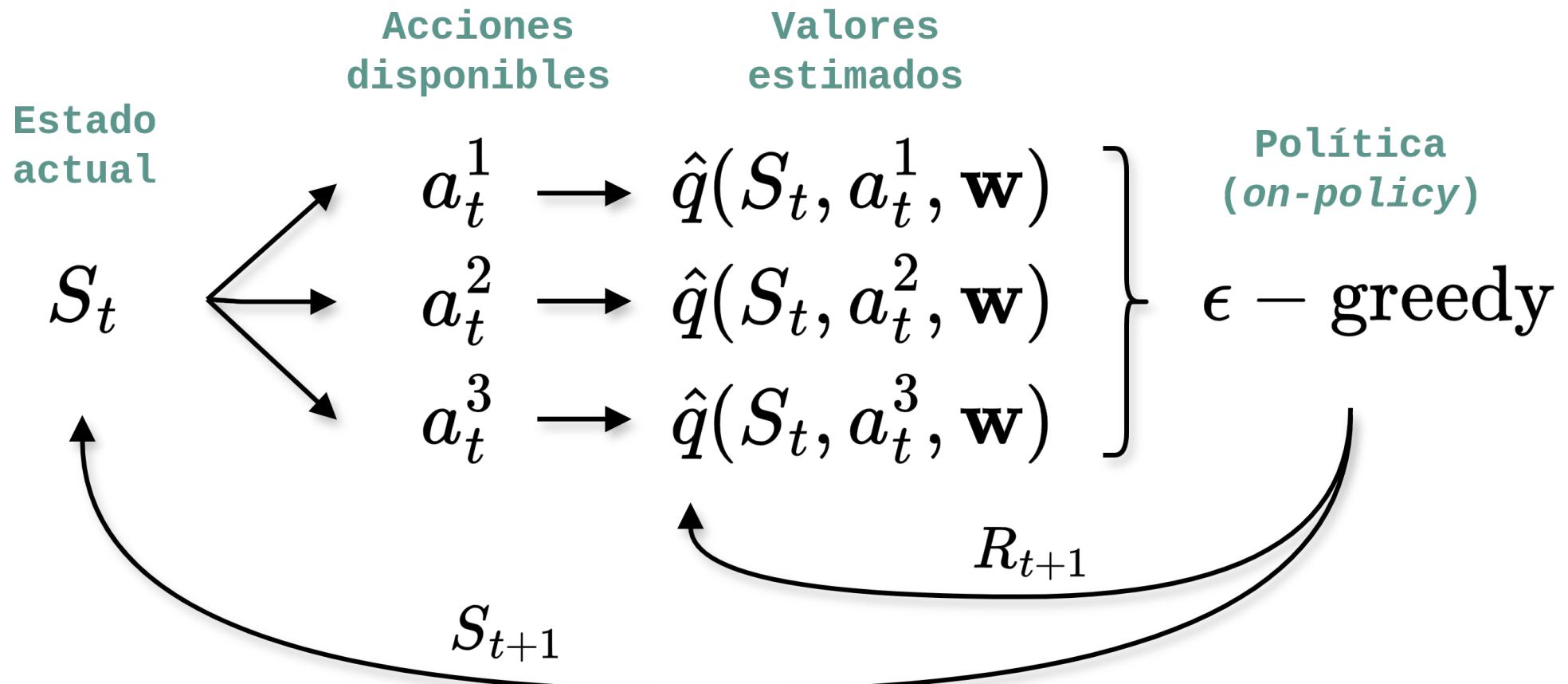
$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha [R_{t+1} + \gamma \hat{q}(S_{t+1}, A_{t+1}, \mathbf{w}_t) - \hat{q}(S_t, A_t, \mathbf{w}_t)] \nabla \hat{q}(S_t, A_t, \mathbf{w}_t)$$

Llamamos a esto **SARSA 1-step semi-gradiente episódico** (*episodic semi-gradient 1-step SARSA*).

- Ofrece las mismas garantías de convergencia y *fixed-point* que TD(0).

Se resume en:

1. Predecir el valor de los pares estado-acción disponibles.
2. Elegir acción (ej.  $\varepsilon$ -greedy).
3. Actualizar el valor estimado.



## Episodic Semi-gradient Sarsa for Estimating $\hat{q} \approx q_*$

Input: a differentiable action-value function parameterization  $\hat{q} : \mathcal{S} \times \mathcal{A} \times \mathbb{R}^d \rightarrow \mathbb{R}$

Algorithm parameters: step size  $\alpha > 0$ , small  $\varepsilon > 0$

Initialize value-function weights  $\mathbf{w} \in \mathbb{R}^d$  arbitrarily (e.g.,  $\mathbf{w} = \mathbf{0}$ )

Loop for each episode:

$S, A \leftarrow$  initial state and action of episode (e.g.,  $\varepsilon$ -greedy)

Loop for each step of episode:

Take action  $A$ , observe  $R, S'$

If  $S'$  is terminal:

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha [R - \hat{q}(S, A, \mathbf{w})] \nabla \hat{q}(S, A, \mathbf{w})$$

Go to next episode

Choose  $A'$  as a function of  $\hat{q}(S', \cdot, \mathbf{w})$  (e.g.,  $\varepsilon$ -greedy)

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha [R + \gamma \hat{q}(S', A', \mathbf{w}) - \hat{q}(S, A, \mathbf{w})] \nabla \hat{q}(S, A, \mathbf{w})$$

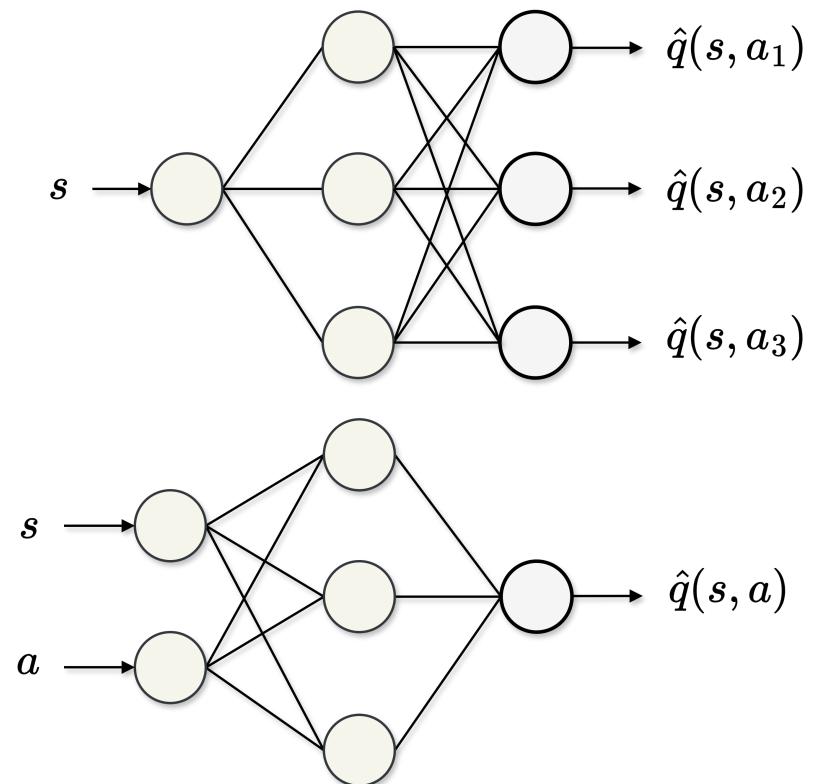
$$S \leftarrow S'$$

$$A \leftarrow A'$$

Si empleamos una **red neuronal** para representar  $\hat{q}(s, a, w)$ , esta puede definirse de varias formas...

- Mientras que la primera forma es apropiada cuando el espacio de acciones es **finito**, la segunda es más apropiada en problemas con valores de acción **continuos**.
- La última capa oculta representa los *features*  $x$  tales que:

$$\hat{q}(s, a, w) = w^T x(s, a)$$



# Garantías de exploración

Para garantizar la **exploración**, empleamos  $\varepsilon$ -greedy.

- $A_t = \operatorname{argmax}_a \hat{q}(S_t, a, w)$  con probabilidad  $1 - \varepsilon$ .
- $A_t$  aleatoria con probabilidad  $\varepsilon$ .

Recordemos que (*semi-gradient*) SARSA es un método **on-policy**, de tal manera que siempre habrá cierta probabilidad de actuar de forma sub-óptima a menos que  $\varepsilon$  se reduzca con el tiempo.

- No se garantiza una exploración sistemática (vs. valores iniciales optimistas), sino que se basa en la aleatoriedad.



¿Pero por qué no es posible emplear **valores iniciales optimistas** en problemas con **aproximación de funciones no-lineales**?

- En problemas con funciones **lineales** tenemos que  $\hat{q}(s, a, \mathbf{w}) = \mathbf{w}^T \mathbf{x}(s, a)$ , por lo que una forma de asegurar valores iniciales optimistas es asignar pesos iniciales lo suficientemente grandes:

$$\mathbf{w} \leftarrow \begin{pmatrix} 1000 \\ 1000 \\ 1000 \\ \dots \end{pmatrix}$$

- Pero en problemas **no-lineales**, ¿cómo sabemos qué pesos iniciales de una red neuronal son apropiados? **Es por esto por lo que empleamos  $\varepsilon$ -greedy.**

$$\hat{q}(s, a, \mathbf{w}) = \text{RN}(s, a, \mathbf{w})$$

$$\mathbf{w} \leftarrow ???$$

# Otros métodos

¿Podemos generalizar para el caso de *Expected SARSA*?

La regla de actualización de pesos para *Expected SARSA* empleando una función acción-valor aproximada sería la siguiente:

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha \left[ R_{t+1} + \gamma \sum_{a'} \pi(a'|S_{t+1}) \hat{q}(S_{t+1}, a', \mathbf{w}) - \hat{q}(S_t, A_t, \mathbf{w}) \right] \nabla \hat{q}(S_t, A_t, \mathbf{w})$$

¿Y en el caso de *Q-learning*?

La regla de actualización de pesos para *Q-learning* empleando una función acción-valor aproximada sería la siguiente:

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha [R_{t+1} + \gamma \max_{a'} \hat{q}(S_{t+1}, a', \mathbf{w}) - \hat{q}(S_t, A_t, \mathbf{w})] \nabla \hat{q}(S_t, A_t, \mathbf{w})$$

- Utilizamos el valor máximo en lugar del valor esperado.
- Recordemos que *Q-learning* es un caso especial de *Expected SARSA*.

# **RECOMPENSA MEDIA**

---

Hasta el momento, hemos visto dos formas de plantear el **objetivo** de un MDP:

- Retorno **episódico**, empleado en problemas finitos:

$$G_t = \sum_{k=0}^{T-t} r_{t+k+1}$$

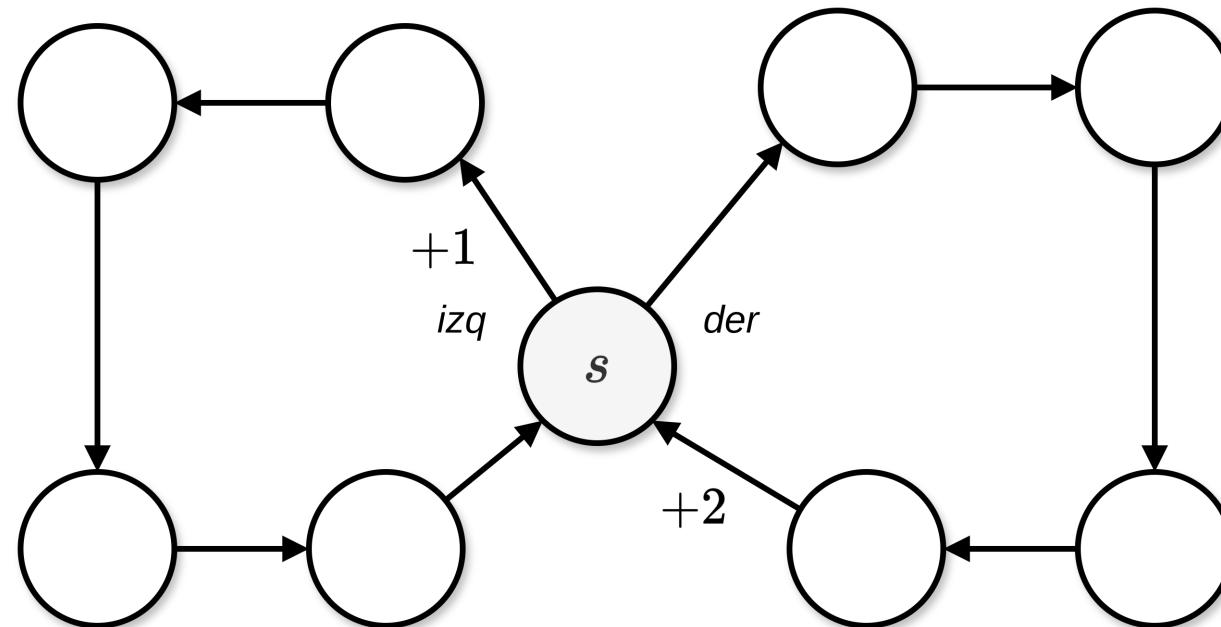
- Retorno **descontado**, empleable en problemas continuados:

$$G_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}$$

El retorno *descontado* es útil en entornos tabulares, pero **puede ser problemático** en algunos casos.

Veamos un ejemplo...

- Problema continuado. Partimos de  $s$  y podemos seguir  $\pi_{\text{izq}}$  o  $\pi_{\text{der}}$ .
- Todas las transiciones tienen como recompensa +0, excepto las que se indican en la figura.



- La política  $\pi_{\text{izq}}$  da lugar a:

$$v_{\text{izq}}(s) = 1 + \gamma \cdot 0 + \gamma^2 \cdot 0 + \gamma^3 \cdot 0 + \gamma^4 \cdot 0$$

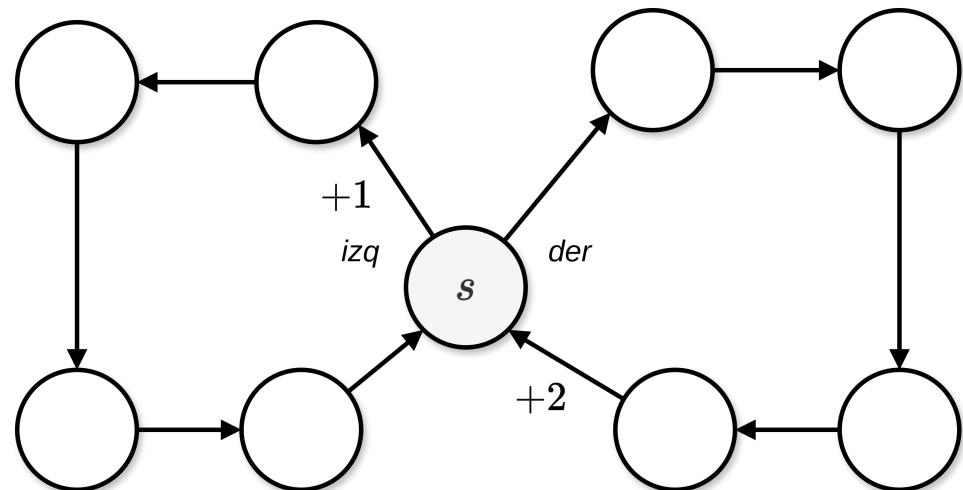
$$+ \gamma^5 \cdot 1 + \gamma^6 \cdot 0 + \dots +$$

$$+ \gamma^{10} \cdot 1 + \gamma^{11} \cdot 0 + \dots$$

$$= 1 + \gamma^5 + \gamma^{10} + \gamma^{15} + \dots$$

$$= \sum_{k=0}^{\infty} \gamma^{5k} = \frac{1}{1 - \gamma^5} \text{ (serie geométrica)}$$

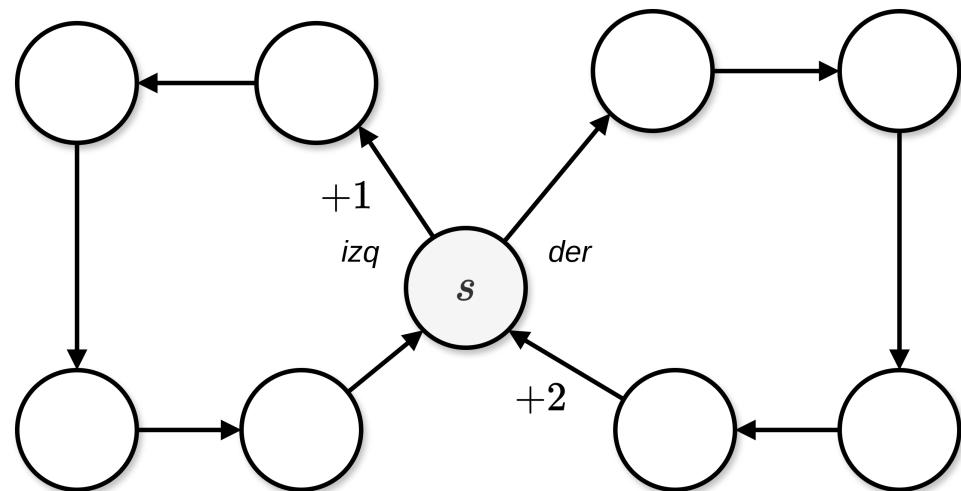
$$v_{\text{izq}}(s) = \frac{1}{1 - \gamma^5}$$



- La política  $\pi_{\text{der}}$  da lugar a:

$$\begin{aligned}v_{\text{der}}(s) &= 0 + \gamma \cdot 0 + \gamma^2 \cdot 0 + \gamma^3 \cdot 0 + \gamma^4 \cdot 2 \\&\quad + \gamma^5 \cdot 0 + \dots + \gamma^9 \cdot 2 + \\&\quad + \gamma^{10} \cdot 0 + \dots + \gamma^{14} \cdot 2 + \dots \\&= \sum_{k=0}^{\infty} \gamma^{4+5k} \cdot 2 = \frac{2\gamma^4}{1-\gamma^5}\end{aligned}$$

$$v_{\text{der}}(s) = \frac{2\gamma^4}{1-\gamma^5}$$



$$v_{\text{izq}}(s) = \frac{1}{1 - \gamma^5}$$

$$v_{\text{der}}(s) = \frac{2\gamma^4}{1 - \gamma^5}$$

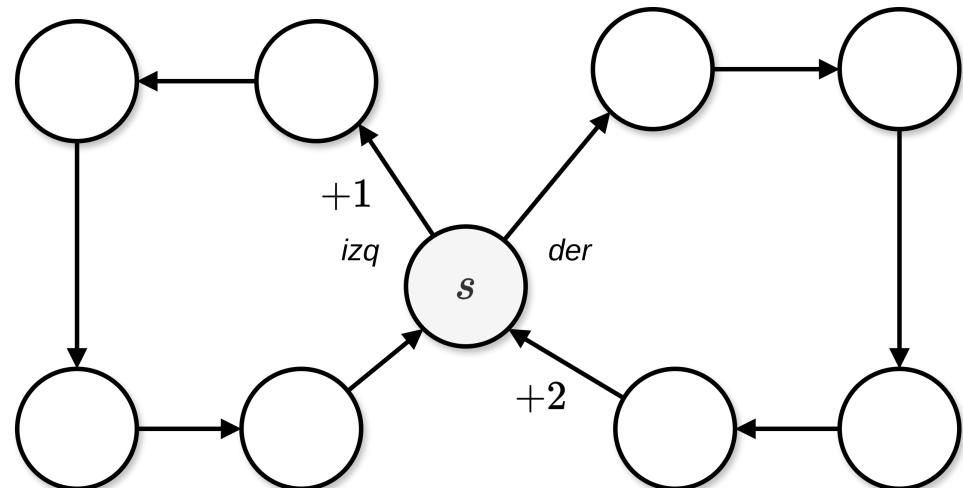
Probemos diferentes valores de  $\gamma$ ...

$\gamma = 0.5 \rightarrow v_{\text{izq}}(s) \simeq 1, v_{\text{der}}(s) \simeq 0.1$

$\pi_{\text{izq}}$  es preferible (  $\times$  )

$\gamma = 0.9 \rightarrow v_{\text{izq}}(s) \simeq 2.4, v_{\text{der}}(s) \simeq 3.2$

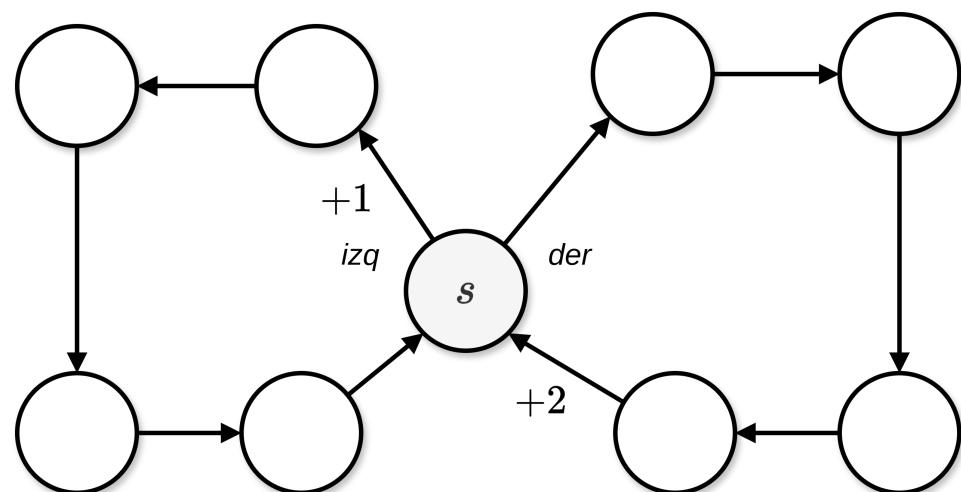
$\pi_{\text{der}}$  es preferible (  $\checkmark$  )



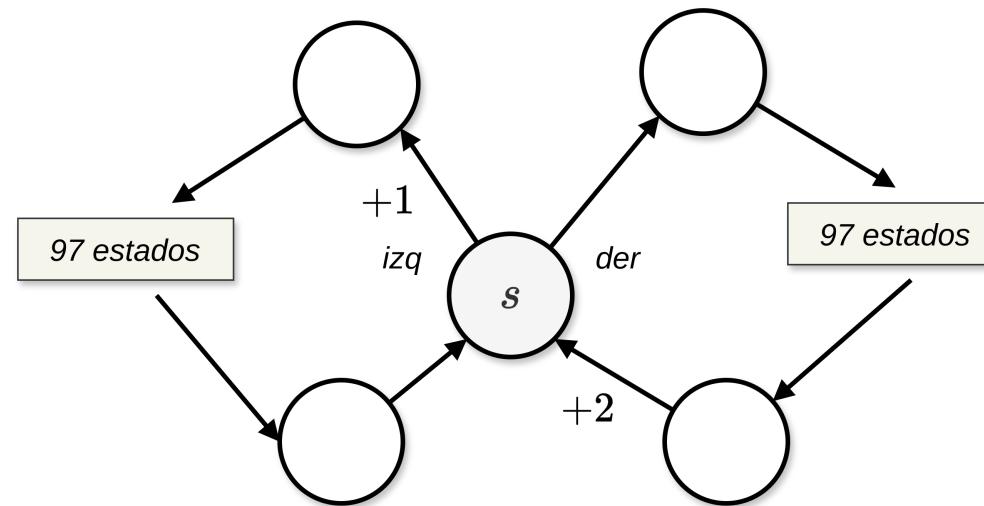
- Si queremos encontrar el valor de  $\gamma$  mínimo que nos lleve a una solución válida:

$$v_{\text{der}}(s) > v_{\text{izq}}(s) \text{ cuando } \gamma > 2^{-\frac{1}{4}} \simeq 0.841$$

El problema ante el que nos encontramos es que la magnitud del valor de descuento  $\gamma$  depende del número de estados del problema.



En este caso, necesitaríamos que  $\gamma > 2^{-\frac{1}{99}} \simeq 0.993$



A medida que aumenta el número de estados,  $\gamma$  tiende a 1 asintóticamente.

- ¡Pero  $\gamma$  no puede ser = 1 al tratarse de un problema continuado!
- Y si es cercano a 1, la suma tiende a infinito.

Para evitar este tipo de problemas, se plantea una alternativa: la **recompensa media**.

$$\begin{aligned} r(\pi) &= \lim_{h \rightarrow \infty} \frac{1}{h} \sum_{t=1}^h \mathbb{E}[R_t \mid S_0, A_{0:t-1} \sim \pi] \\ &= \lim_{t \rightarrow \infty} \mathbb{E}[R_t \mid S_0, A_{0:t-1} \sim \pi] \\ &= \sum_s \mu_\pi(s) \sum_a \pi(a|s) \sum_{s',r} p(s',r \mid s,a)r \end{aligned}$$

donde  $\mu_\pi(s)$  es la distribución de visitas a un estado  $s$ , tal que:

$$\mu_\pi(s) = \lim_{t \rightarrow \infty} \Pr\{S_t = s \mid A_{0:t-1} \sim \pi\}$$

De esta forma, la calidad de una política  $\pi$  viene dada por la **recompensa media por timestep** obtenida bajo esa política.

$$r(\pi) = \lim_{h \rightarrow \infty} \frac{1}{h} \sum_{t=1}^h \mathbb{E}[R_t \mid S_0, A_{0:t-1} \sim \pi]$$

- Toda política que maximice  $r(\pi)$  es una política óptima.
- El **retorno** pasa a denominarse **retorno diferencial**, y se define tal que:

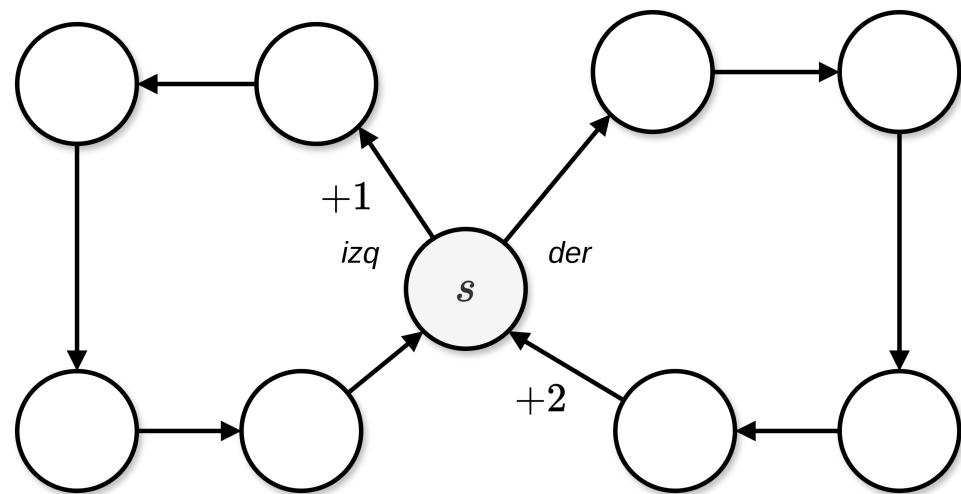
$$G_t = R_{t+1} - r(\pi) + R_{t+2} - r(\pi) + R_{t+3} - r(\pi) + \dots$$

Volviendo al ejemplo anterior...

$$r(\pi_{izq}) = \frac{1}{5} = 0.2$$

$$r(\pi_{der}) = \frac{2}{5} = 0.4$$

La recompensa media nos indica directamente qué política es mejor.



¿Por qué en problemas con aproximación de funciones es preferible la recompensa media al enfoque descontado?

- En entornos descontados, es necesario definir un **factor de descuento**  $\gamma$  ajustado de forma «manual».
- En función del valor de  $\gamma$  elegido, nuestra representación/modelo de la función de valor  $v, q$  puede variar significativamente. Esto hace que nuestra aproximación sea **inestable y dependiente de  $\gamma$** .
- Si empleamos el **retorno diferencial**, contamos con **funciones de valor independientes de  $\gamma$**  (solamente dependientes de la recompensa media).
- El uso de la **recompensa media nos permite comparar políticas sin depender del valor de  $\gamma$** .

Las **ecuaciones de Bellman** con retorno diferencial se definen de la siguiente forma:

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{r,s'} p(s', r|s, a) [r - r(\pi) + v_\pi(s')]$$

$$q_\pi(s, a) = \sum_{r,s'} p(s', r|s, a) \left[ r - r(\pi) + \sum_{a'} \pi(a'|s') q_\pi(s', a') \right]$$

$$v_*(s) = \max_a \sum_{r,s'} p(s', r|s, a) \left[ r - \max_\pi r(\pi) + v_*(s') \right]$$

$$q_*(s, a) = \sum_{r,s'} p(s', r|s, a) \left[ r - \max_\pi r(\pi) + \max_{a'} q_*(s', a') \right]$$

También contamos con una definición de **TD-error** empleando retorno diferencial:

$$\delta_t = R_{t+1} - \bar{R}_t + \hat{v}(S_{t+1}, \mathbf{w}_t) - \hat{v}(S_t, \mathbf{w}_t)$$

$$\delta_t = R_{t+1} - \bar{R}_t + \hat{q}(S_{t+1}, A_{t+1}, \mathbf{w}_t) - \hat{q}(S_t, A_t, \mathbf{w}_t)$$

donde  $\bar{R}_t$  es la estimación en el instante  $t$  de la recompensa media  $r(\pi)$ .

Empleando estas definiciones alternativas, los algoritmos vistos hasta el momento pueden adaptarse a la formulación de recompensa media sin apenas cambios.

## Differential semi-gradient Sarsa for estimating $\hat{q} \approx q_*$

Input: a differentiable action-value function parameterization  $\hat{q} : \mathcal{S} \times \mathcal{A} \times \mathbb{R}^d \rightarrow \mathbb{R}$

Algorithm parameters: step sizes  $\alpha, \beta > 0$

Initialize value-function weights  $\mathbf{w} \in \mathbb{R}^d$  arbitrarily (e.g.,  $\mathbf{w} = \mathbf{0}$ )

Initialize average reward estimate  $\bar{R} \in \mathbb{R}$  arbitrarily (e.g.,  $\bar{R} = 0$ )

Initialize state  $S$ , and action  $A$

Loop for each step:

    Take action  $A$ , observe  $R, S'$

    Choose  $A'$  as a function of  $\hat{q}(S', \cdot, \mathbf{w})$  (e.g.,  $\varepsilon$ -greedy)

$\delta \leftarrow R - \bar{R} + \hat{q}(S', A', \mathbf{w}) - \hat{q}(S, A, \mathbf{w})$

$\bar{R} \leftarrow \bar{R} + \beta\delta$

$\mathbf{w} \leftarrow \mathbf{w} + \alpha\delta\nabla\hat{q}(S, A, \mathbf{w})$

$S \leftarrow S'$

$A \leftarrow A'$

Dada la siguiente definición del **retorno diferencial n-step**:

$$G_{t:t+n} = R_{t+1} + \overline{R}_{t+n-1} + \dots + R_{t+n} - \overline{R}_{t+n-1} + \hat{q}(S_{t+n}, A_{t+n}, \mathbf{w}_{t+n-1})$$

la versión **n-step del TD-error diferencial** es:

$$\delta_t = G_{t:t+n} - \hat{q}(S_t, A_t, \mathbf{w})$$

Esto nos permite formular la versión diferencial n-step de SARSA semi-gradiente...

## Differential semi-gradient $n$ -step Sarsa for estimating $\hat{q} \approx q_\pi$ or $q_*$

Input: a differentiable function  $\hat{q} : \mathcal{S} \times \mathcal{A} \times \mathbb{R}^d \rightarrow \mathbb{R}$ , a policy  $\pi$

Initialize value-function weights  $\mathbf{w} \in \mathbb{R}^d$  arbitrarily (e.g.,  $\mathbf{w} = \mathbf{0}$ )

Initialize average-reward estimate  $\bar{R} \in \mathbb{R}$  arbitrarily (e.g.,  $\bar{R} = 0$ )

Algorithm parameters: step size  $\alpha, \beta > 0$ , a positive integer  $n$

All store and access operations ( $S_t$ ,  $A_t$ , and  $R_t$ ) can take their index mod  $n + 1$

Initialize and store  $S_0$  and  $A_0$

Loop for each step,  $t = 0, 1, 2, \dots$ :

    Take action  $A_t$

    Observe and store the next reward as  $R_{t+1}$  and the next state as  $S_{t+1}$

    Select and store an action  $A_{t+1} \sim \pi(\cdot | S_{t+1})$ , or  $\varepsilon$ -greedy wrt  $\hat{q}(S_{t+1}, \cdot, \mathbf{w})$

$\tau \leftarrow t - n + 1$    ( $\tau$  is the time whose estimate is being updated)

    If  $\tau \geq 0$ :

$$\delta \leftarrow \sum_{i=\tau+1}^{\tau+n} (R_i - \bar{R}) + \hat{q}(S_{\tau+n}, A_{\tau+n}, \mathbf{w}) - \hat{q}(S_\tau, A_\tau, \mathbf{w})$$

$$\bar{R} \leftarrow \bar{R} + \beta \delta$$

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha \delta \nabla \hat{q}(S_\tau, A_\tau, \mathbf{w})$$

# **TRABAJO PROPUESTO**

---

- Estudiar las **implementaciones** de los algoritmos vistos.
  - [https://github.com/manjavacas/rl-temario/tree/main/ejemplos/value\\_approx/](https://github.com/manjavacas/rl-temario/tree/main/ejemplos/value_approx/)
  - Impleméntalos por tu cuenta y aplícalos en otros problemas.
- Lectura sobre **coarse coding** y **tile coding** (ver Sutton & Barto sec. 9.5 & 9.6 ).
- Investiga sobre el algoritmo **DQN** (Deep Q-Network).

## Bibliografía y recursos

- **Capítulos 9 y 10** de Sutton, R. S., & Barto, A. G. (2018). Reinforcement learning: An introduction.
- <https://youtu.be/Xg0WGzlEefY?si=r8Z2wrocsoHcyo7r>
- <https://statquest.org/neural-networks-part-1-inside-the-black-box/>
- <https://michaeloneill.github.io/RL-tutorial.html>
- <https://www.davidsilver.uk/wp-content/uploads/2020/03/FA.pdf>
- [https://www.youtube.com/watch?v=Vky0WVh\\_FSk](https://www.youtube.com/watch?v=Vky0WVh_FSk)

# **APRENDIZAJE POR REFUERZO**

Aproximación de funciones de valor

---

Antonio Manjavacas

[manjavacas@ugr.es](mailto:manjavacas@ugr.es)