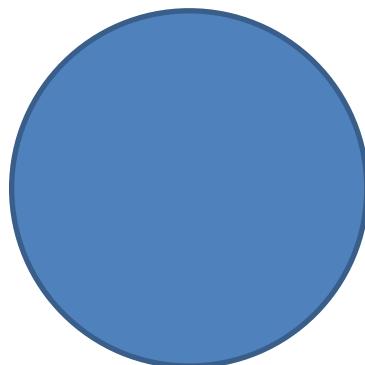


Machine Learning & Deep Learning

Algoritmos Avanzados de Machine Learning

Profesor: Carlos Moreno Morera





Contenido

01

Introducción a los métodos ensemble

02

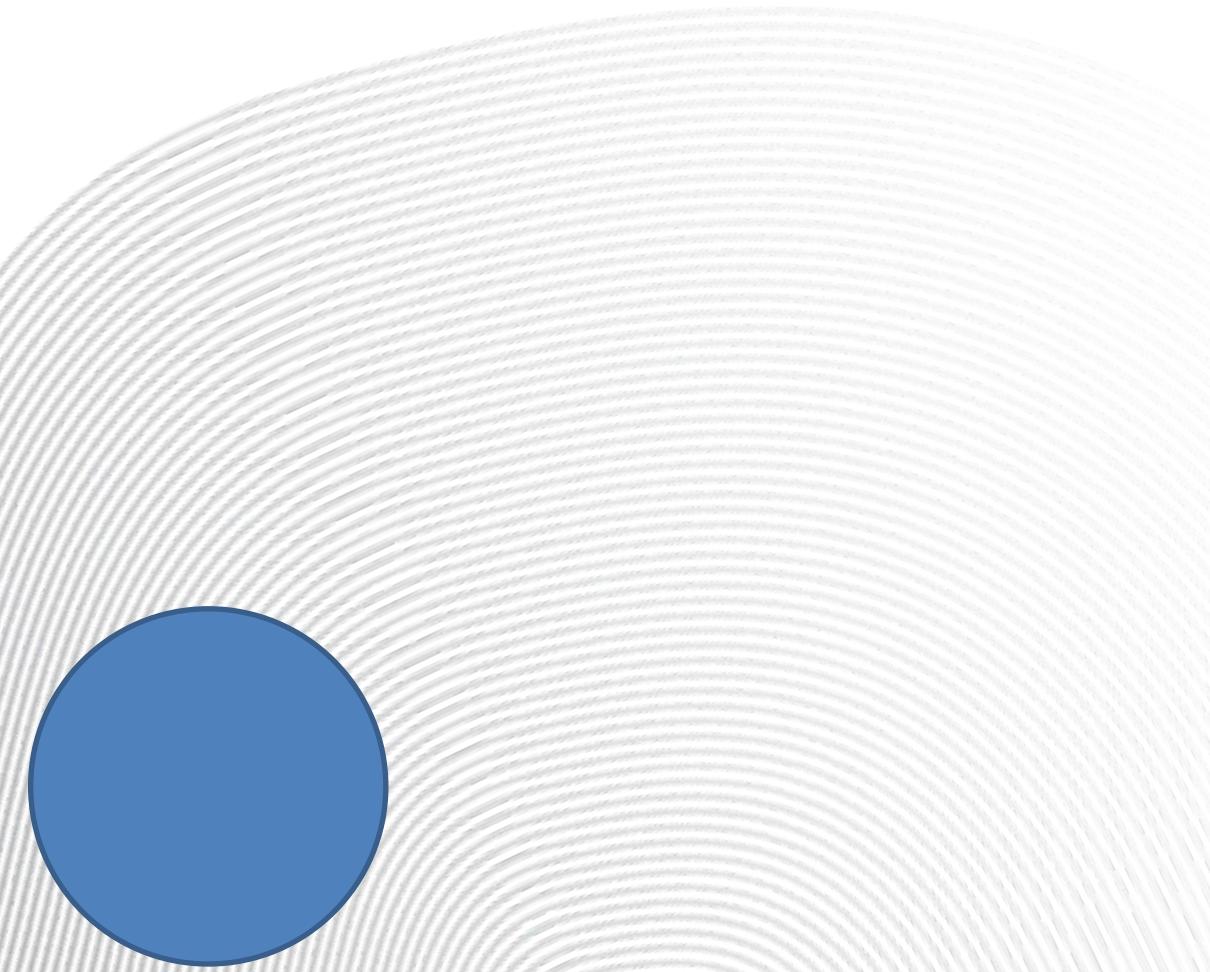
Tipos de ensambles

03

Tipos de combinaciones

04

Algoritmos de bagging





Contenido

05

Algoritmos de boosting

06

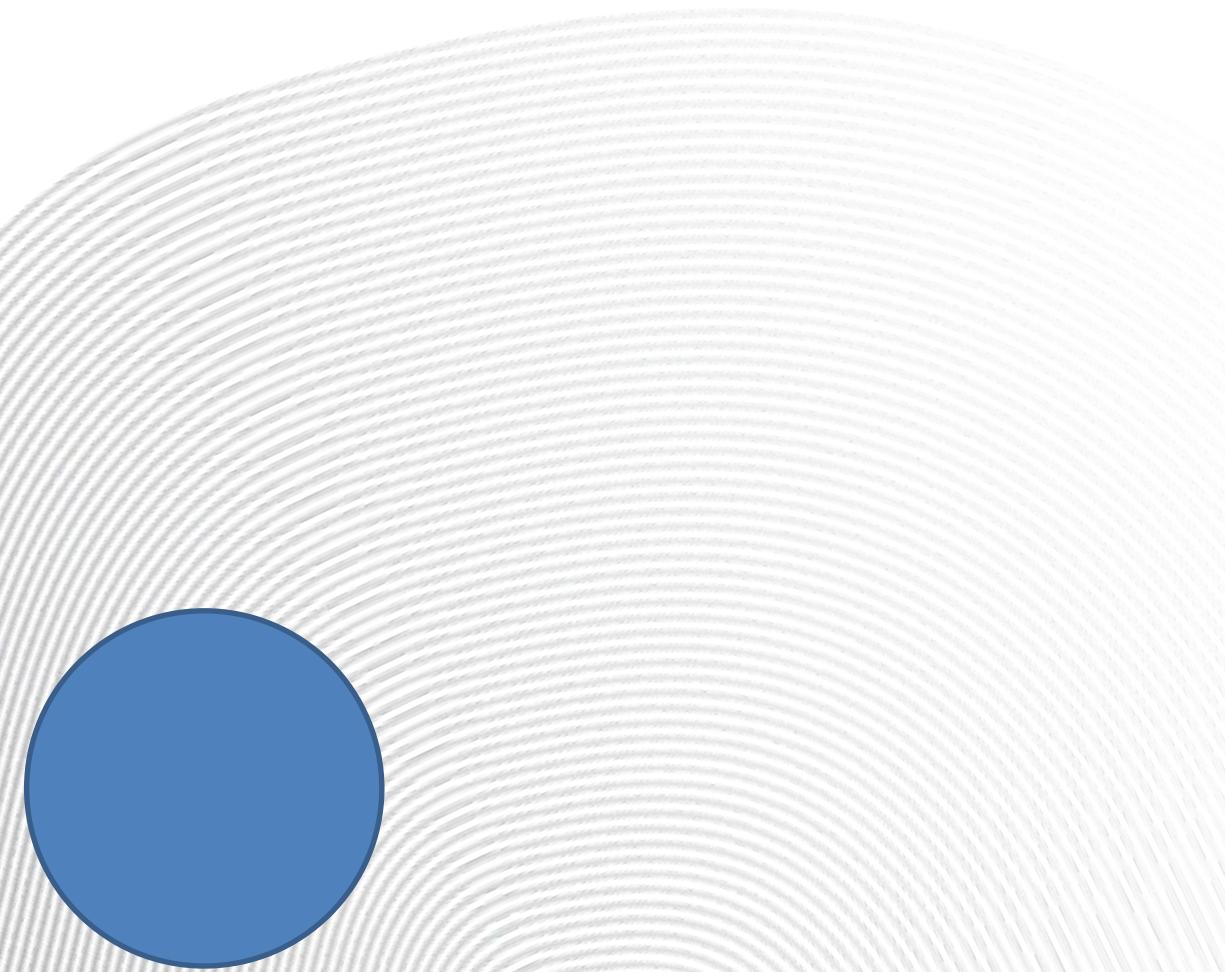
Introducción al aprendizaje por refuerzo

07

Algoritmo Q-Learning

08

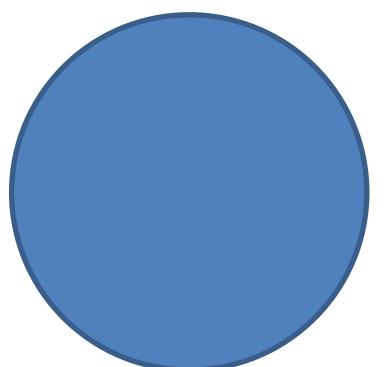
Optimización de Q-Learning



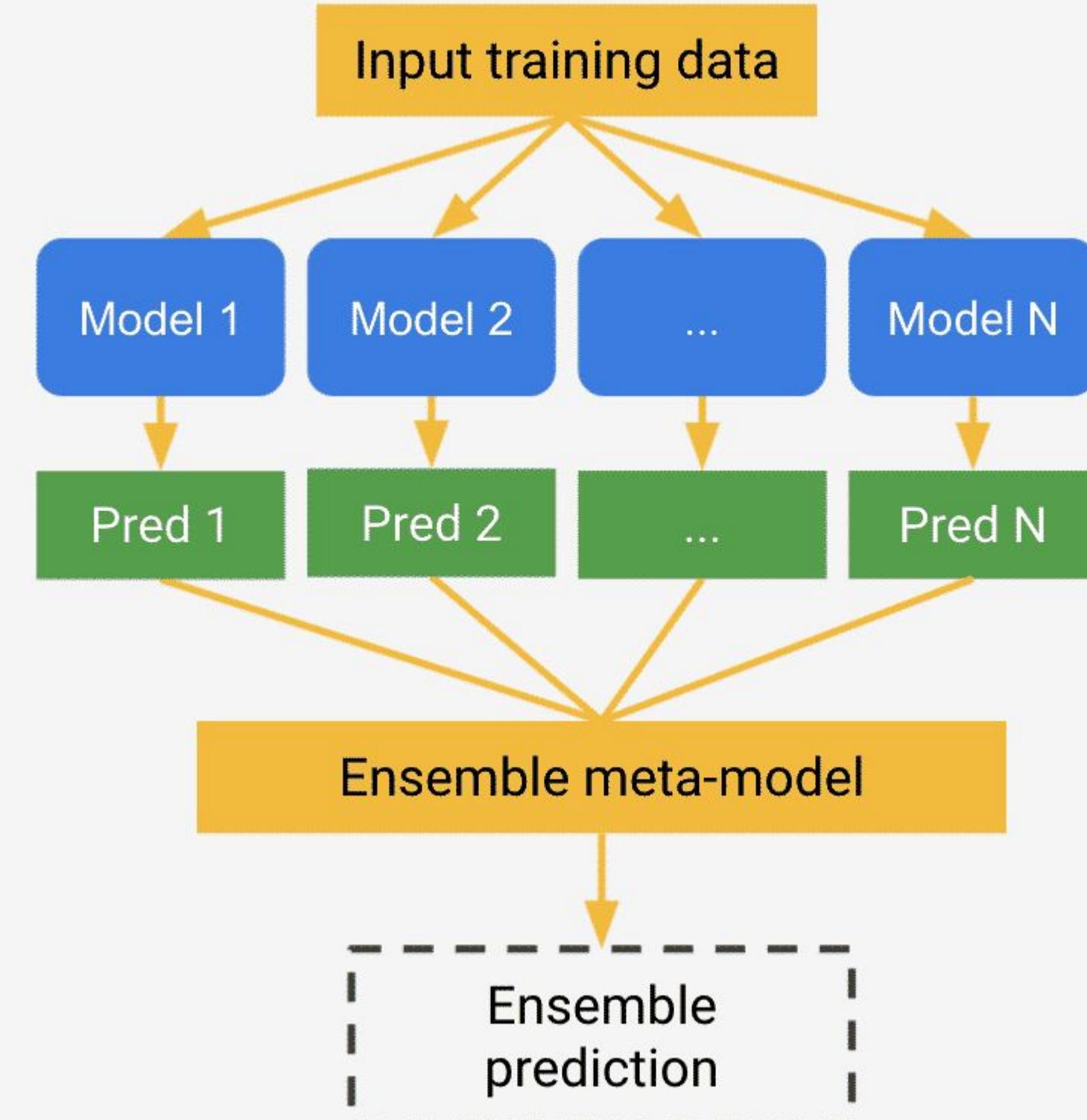
01

Introducción a los métodos ensemble

Conceptos básicos sobre los métodos ensemble



Ensemble Learning



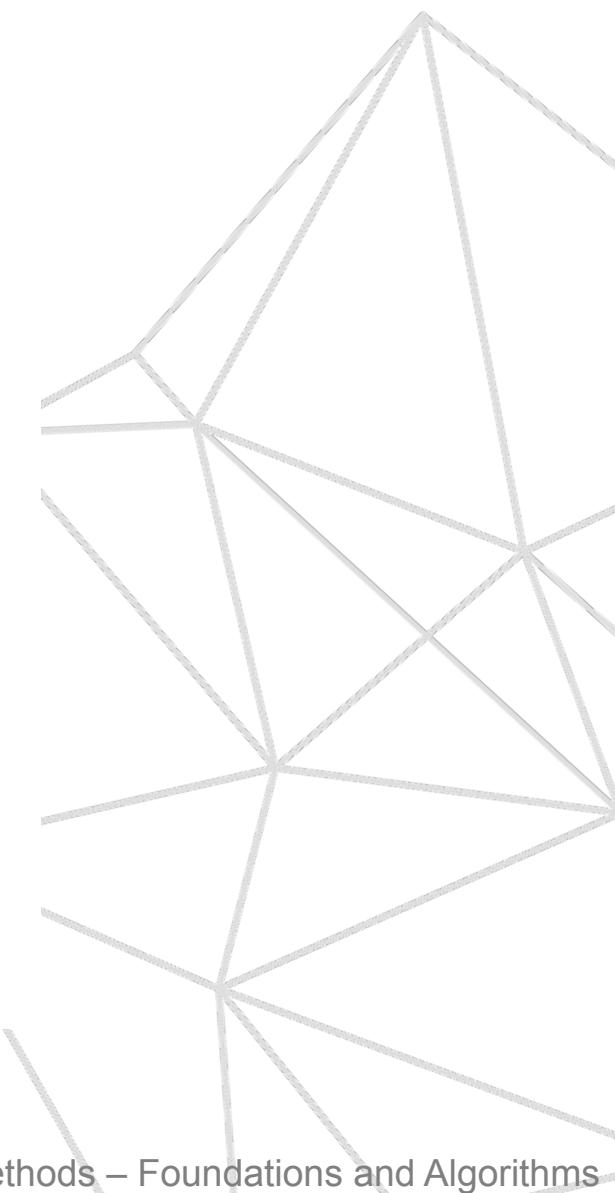
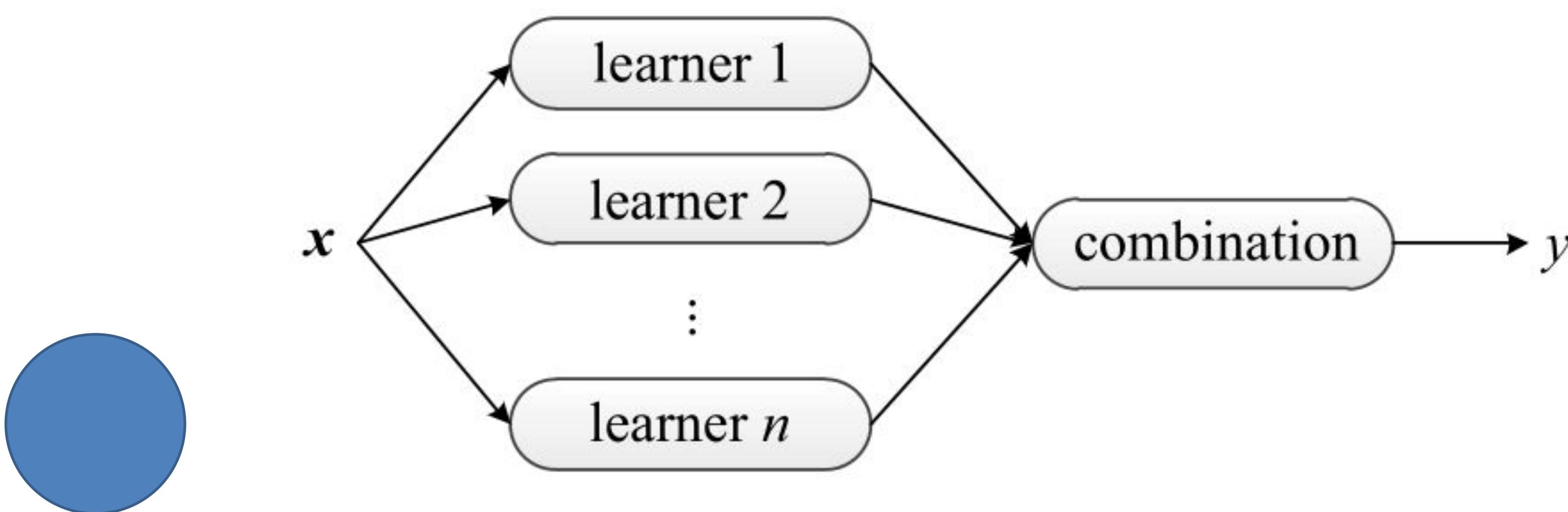


“ La unión hace la fuerza ”



Definición

Los métodos ensemble (o, en castellano, métodos de ensamblaje) son aquellos (generalmente pertenecientes al ámbito del **aprendizaje supervisado**) que utilizan **múltiples algoritmos de aprendizaje** con el fin de obtener mejores resultados predictivos que los que se consiguen de cualquiera de los algoritmos utilizados por separado. Estos métodos, son más **costosos computacionalmente**.

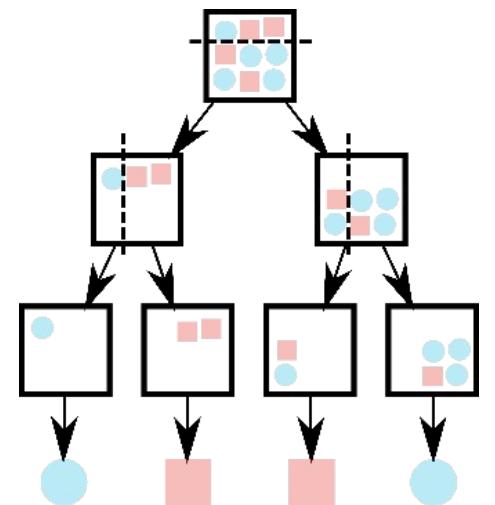




Tipos de modelos base

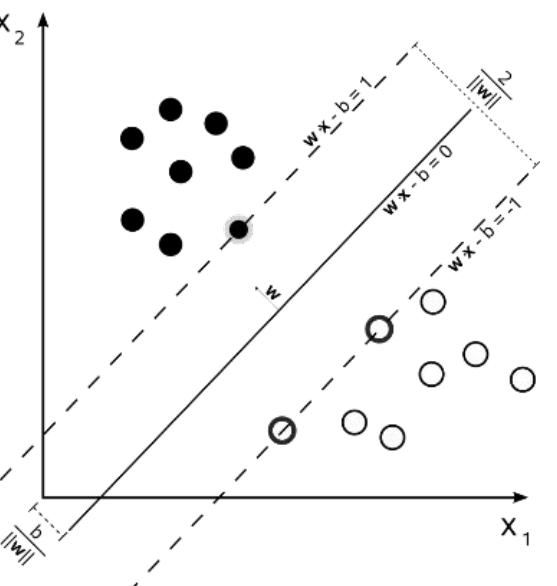
Existen dos grandes categorías de modelos base a partir de los que construir ensambles:

Modelos débiles

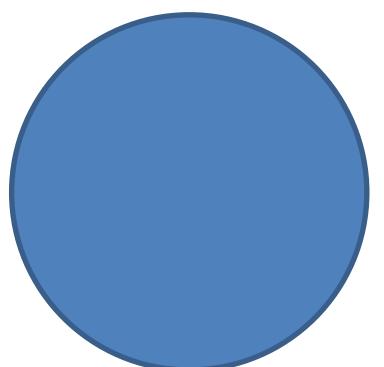


- Poco precisos en general.
- Funcionan sólo ligeramente mejor que modelos triviales basados en probabilidades a priori (clasificación) o en medias (regresión).
- Rápidos de entrenar.
- Ejemplos: árboles sencillos, regresión lineal, ...

Modelos fuertes



- Potencia suficiente para ser más precisos que los débiles.
- Funcionan considerablemente mejor que los modelos triviales.
- Entrenamiento costoso.
- Ejemplos: SVMs, redes neuronales, ...





Métricas deseables

Los modelos base de un ensemble se construyen de manera que maximice la combinación de dos métricas:

Precisión

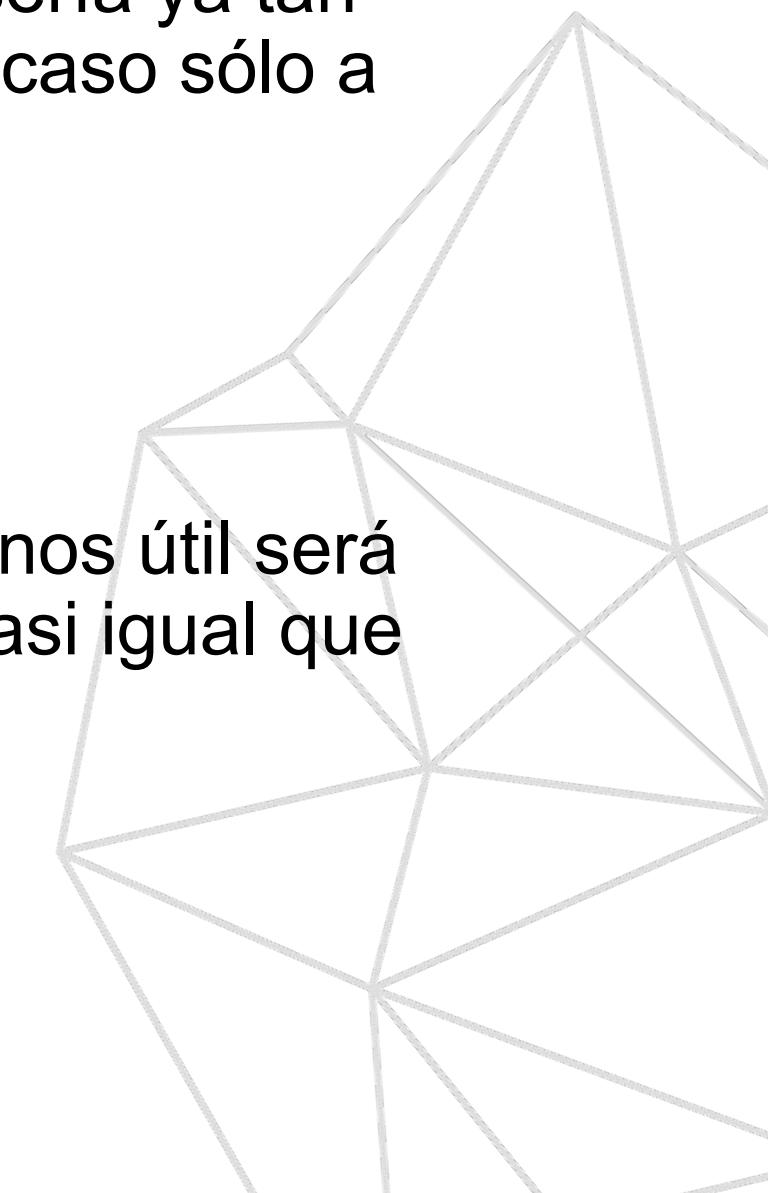
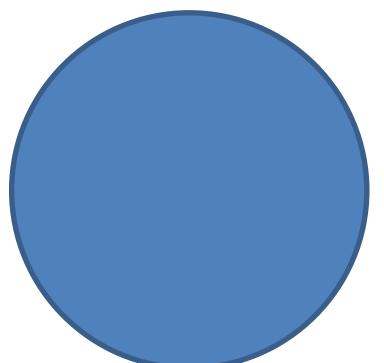


- Cuanto más precisos sean los modelos base, más precisa será la decisión combinada (porque la peor combinación sería ya tan precisa como el modelo menos preciso, al hacerle caso sólo a ese).

Diversidad



- Cuanto más parecidos sean los modelos base, menos útil será combinarlos (porque la decisión combinada será casi igual que cada una de las individuales).

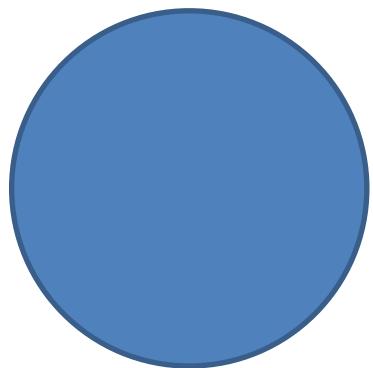




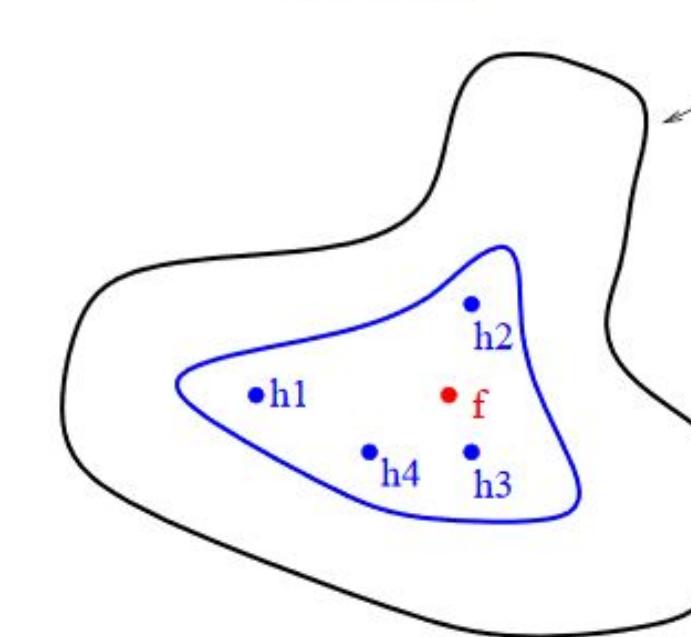
Beneficios de los ensambles

Existen tres razones principales por las que un ensemble f puede ser superior a sus modelos h_1, h_2, \dots, h_n :

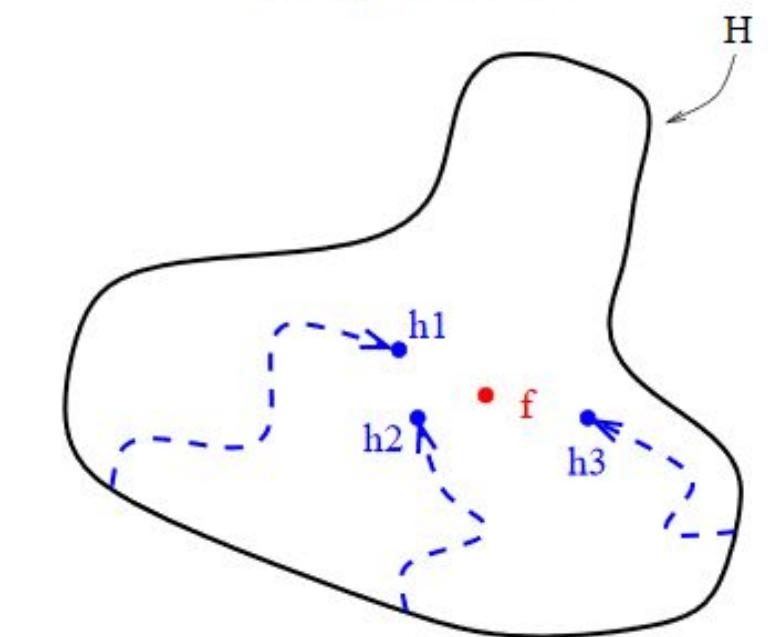
- **Estadística:** disminuye el riesgo de elegir una hipótesis alejada de la realidad (puede ocurrir que varias hipótesis expliquen adecuadamente los datos).
- **Computacional:** mayor probabilidad de encontrar extremos globales y no locales.
 - **Representativa:** puede ocurrir que la realidad no pueda representarse con la familia de modelos elegida. Al combinar varios modelos se pueden lograr representaciones imposibles con un solo modelo, extendiendo así el espacio de hipótesis.



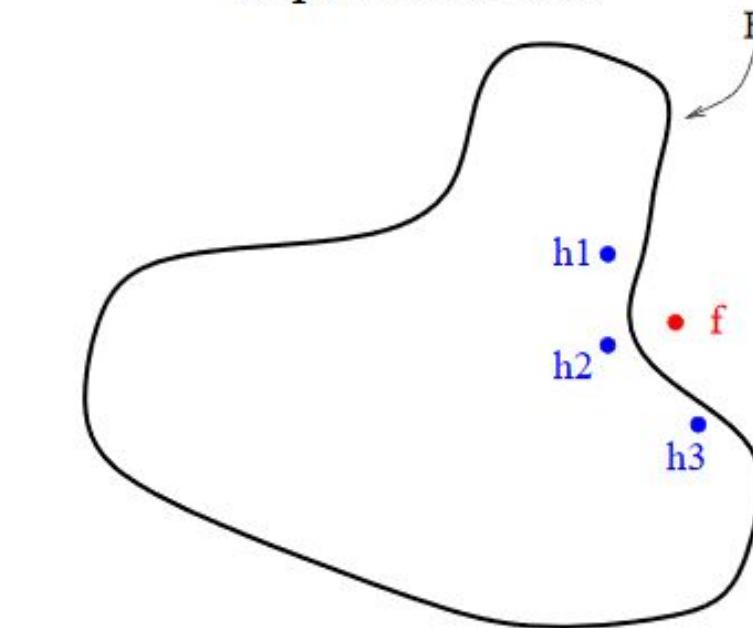
Statistical



Computational



Representational





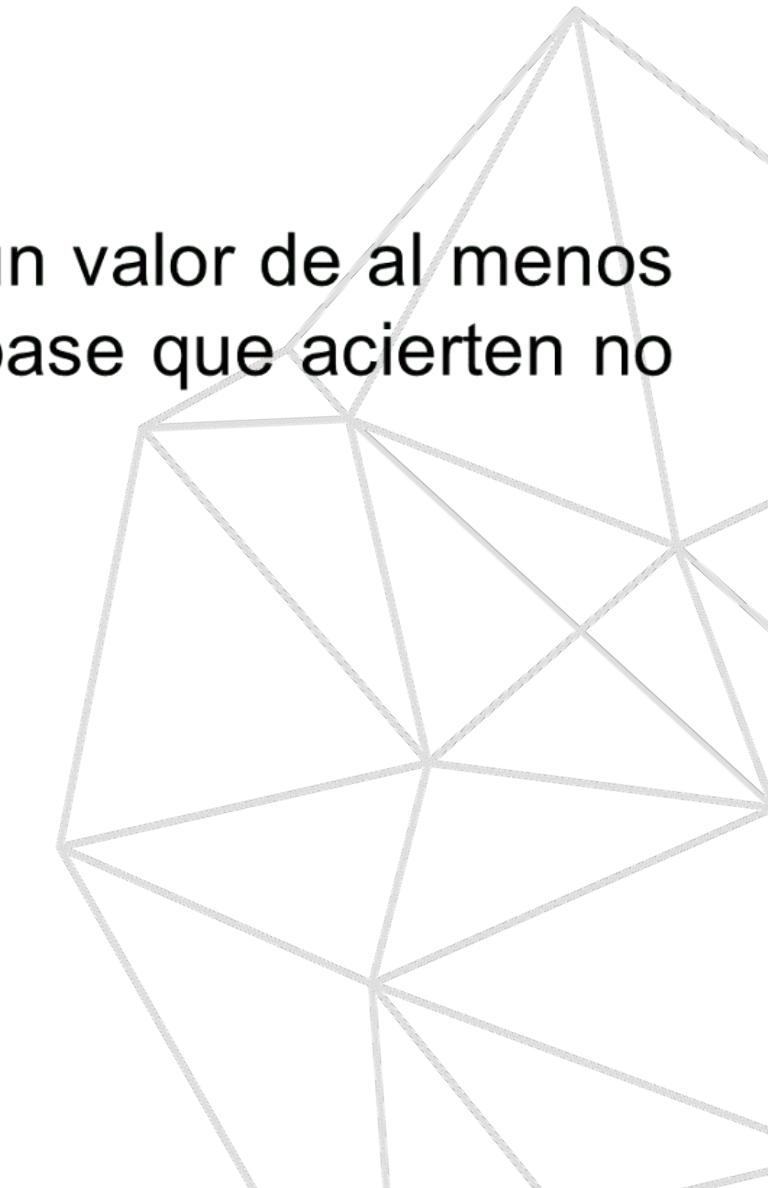
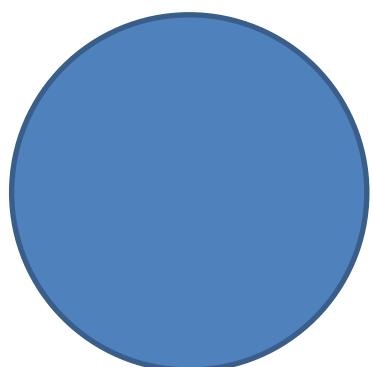
Cotas de error de los ensambles

Supongamos un problema de clasificación binaria con dos clases representadas por -1 y 1 y T clasificadores base en un ensemble que combina las predicciones por voto por mayoría:

$$f(X_i) = \text{sign} \left(\sum_{k=1}^T h_k(X_i) \right)$$

Si la probabilidad de error de los clasificadores es independiente entre ellos y tiene un valor de al menos ε , la probabilidad de que se equivoque el ensemble será la probabilidad de que los base que acierten no lleguen a ser la mitad:

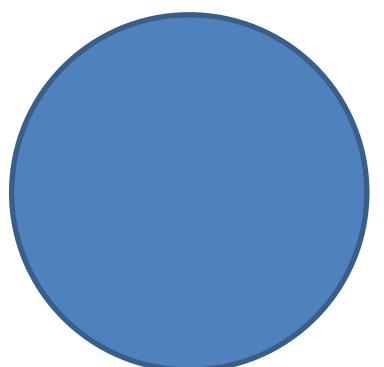
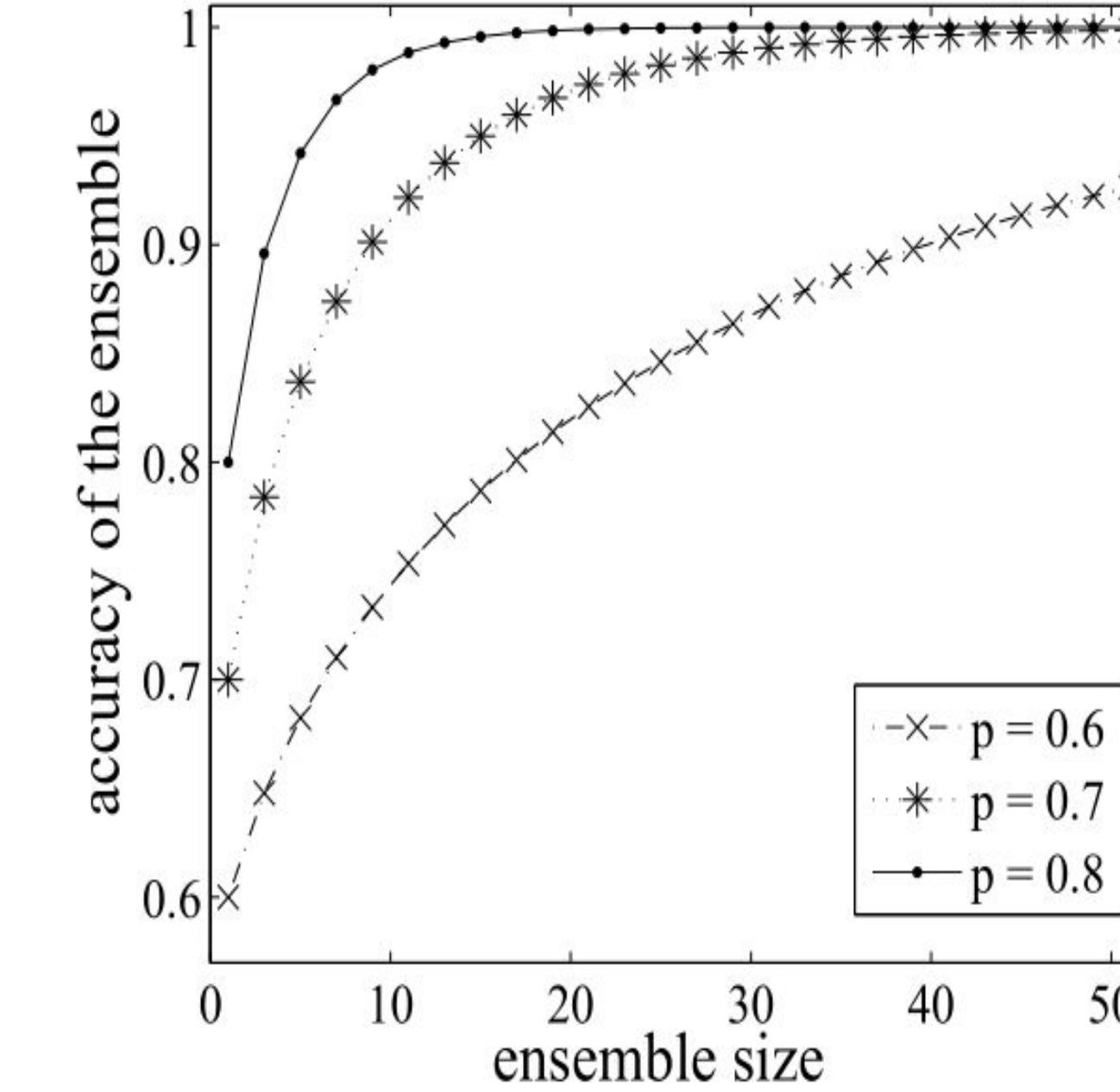
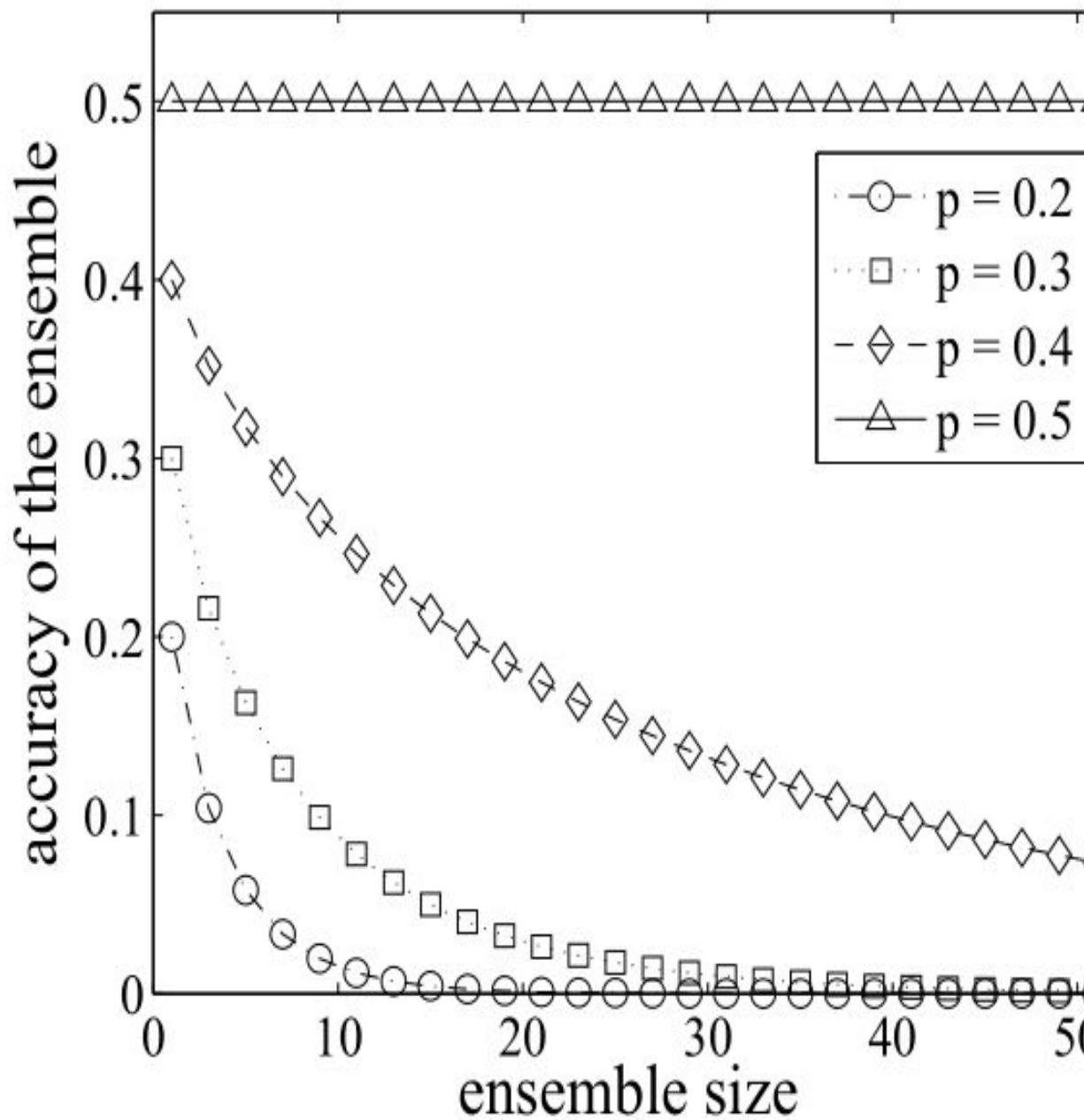
$$P(f(X_i) \neq y_i) = \sum_{k=0}^{\lfloor T/2 \rfloor} \binom{T}{k} (1 - \varepsilon)^k \varepsilon^{T-k} \leq e^{-\frac{1}{2}T(2\varepsilon-1)^2}$$





Cotas de error de los ensambles

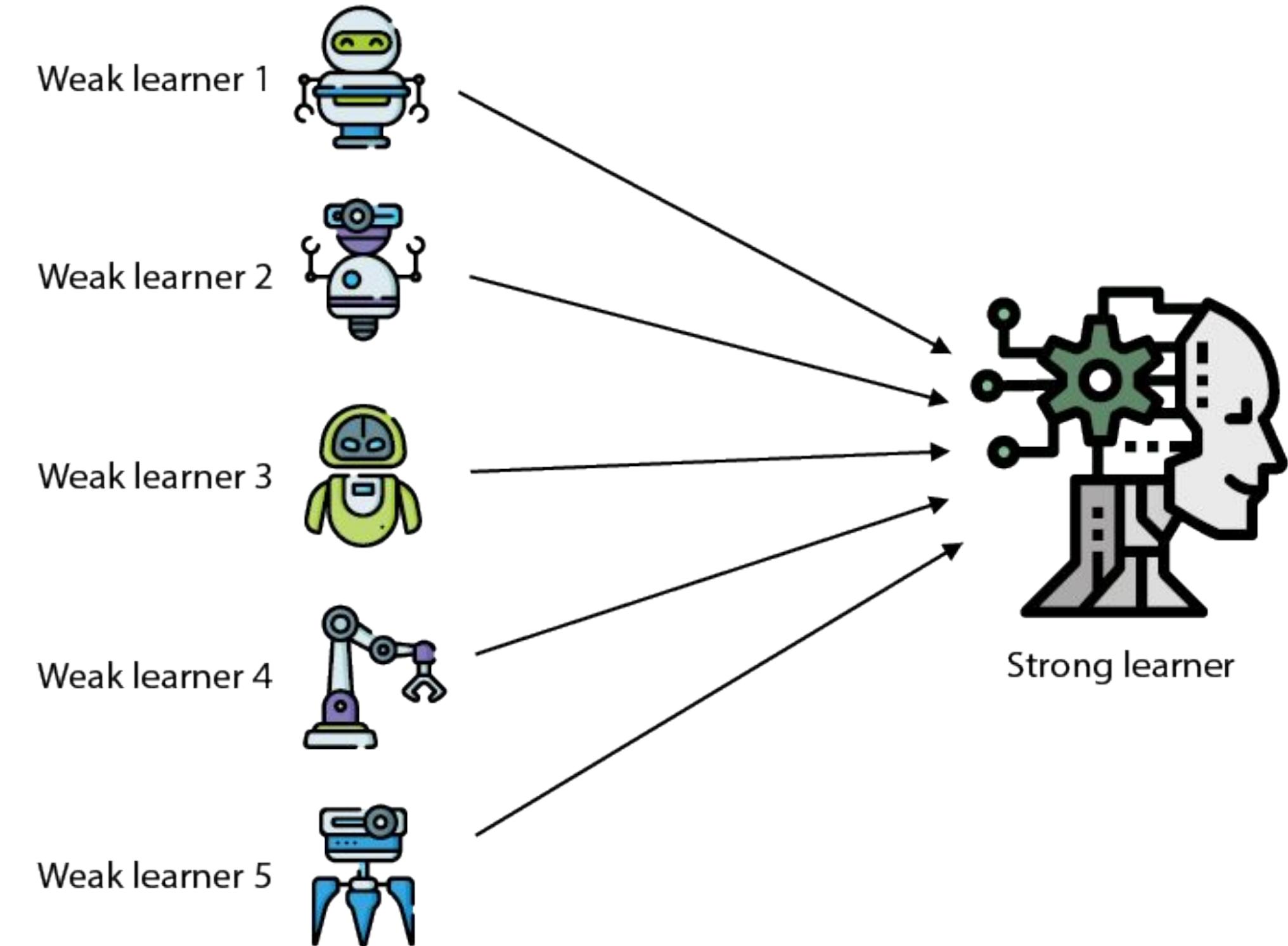
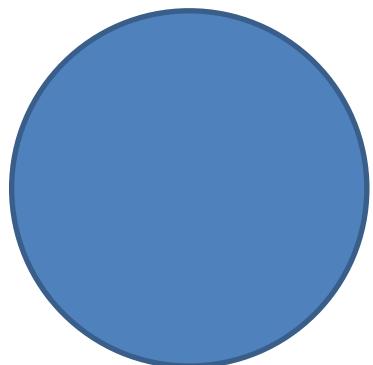
Si la probabilidad de acierto de cada clasificador base es mayor a 0.5, la precisión del ensemble tiende a 1 cuando el tamaño del ensemble tiende a infinito. Por el contrario si es menor que 0.5 la precisión tiende a 0.



02

Tipos de ensambles

Se estudian los paradigmas de construcción de los ensambles.

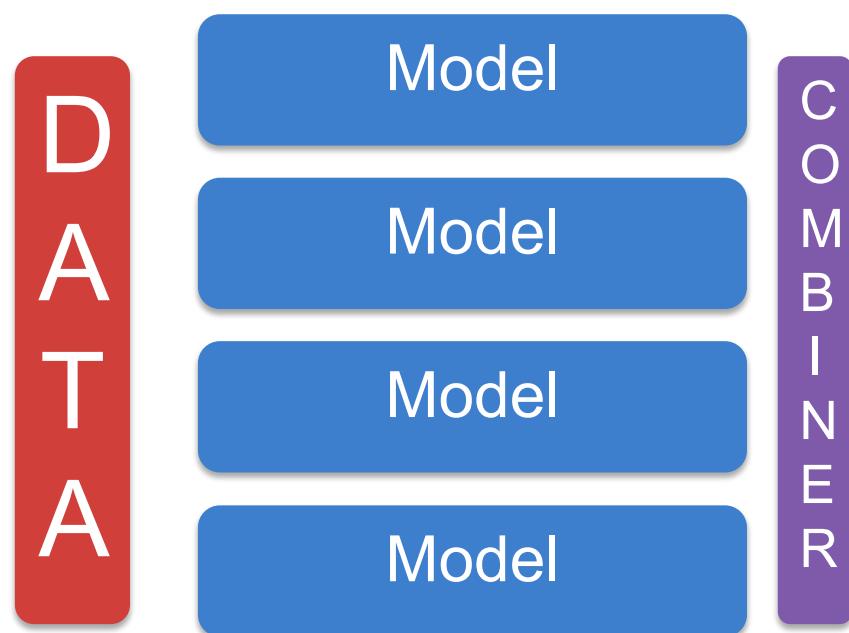




Paradigmas de construcción

Existen 2 modos principales de construcción de ensambles a partir de los datos:

Ensembles paralelos

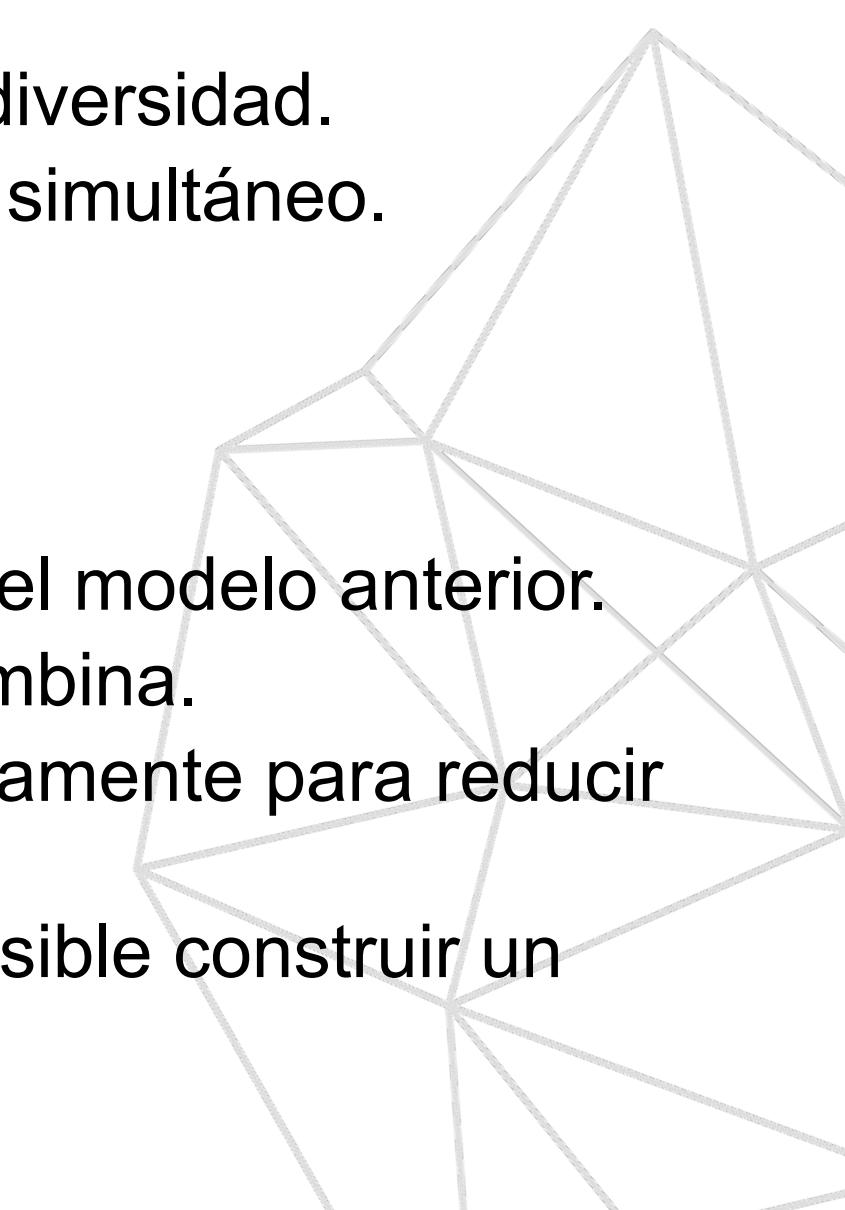


- Cada modelo base trabaja en paralelo sobre los datos.
- Los modelos no colaboran directamente entre ellos.
- La predicción de todos ellos se combina.
- Explotan la independencia, favoreciendo la diversidad.
- Permiten explotar el paralelismo y el cálculo simultáneo.
- La estrategia principal es bagging

Ensembles secuenciales



- Cada modelo trata de corregir los defectos del modelo anterior.
- La predicción de los modelos también se combina.
- Explotan la dependencia, colaborando directamente para reducir el error global.
- Se puede demostrar teóricamente que es posible construir un ensemble fuerte a partir de modelos débiles.
- La estrategia principal es el boosting





Independencia de los ensambles paralelos

En general es poco realista pensar que los modelos base de un ensemble paralelo vayan a ser realmente independientes, porque los datos se comparten. Idealmente, si tuviéramos una cantidad muy grande de datos:

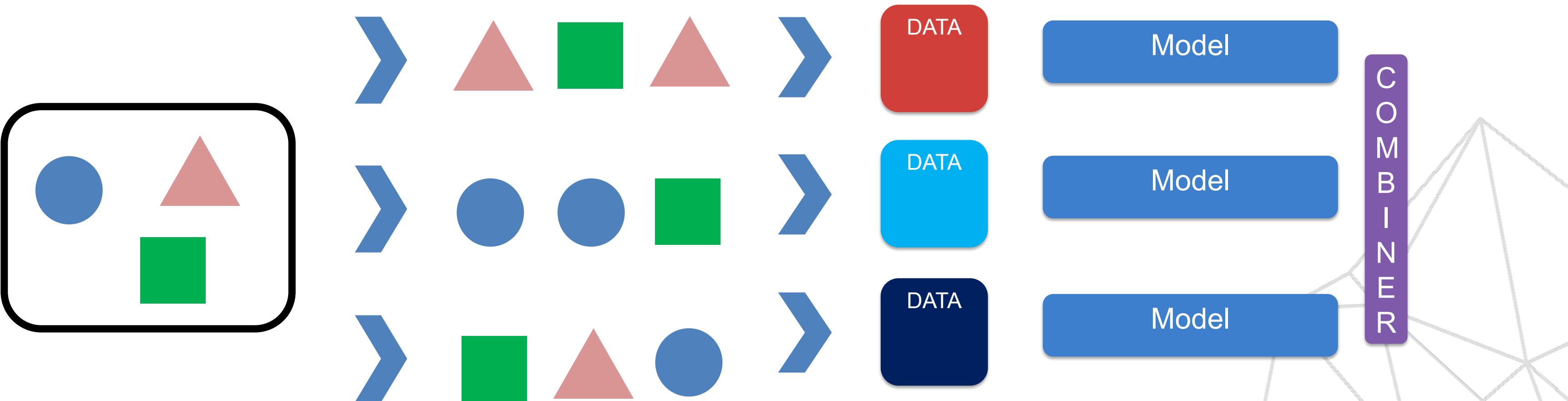


- Cada modelo base trabajaría con una muestra aleatoria de los datos.
- La independencia sería total, y cada modelo tendría suficientes datos como para no ser un modelo demasiado débil.



Ensembles paralelos: bagging

Por desgracia, casi nunca tenemos suficientes datos como para este ideal, y entonces recurrimos a la técnica de bagging (Bootstrap AGGREGatING):



- Cada modelo toma una muestra aleatoria con reemplazo del mismo tamaño que los datos originales (muestra bootstrap).
- De este modo, los datos que ve cada modelo son $\approx 1 - \frac{1}{e} = 63.2\%$ de patrones originales (y el resto duplicados).
- Se combinan los resultados de los distintos modelos en la fase de agregación.



Modelos estables e inestables

Puesto que con bagging cada modelo base comparte patrones con otros modelos base, es importante que dichos modelos base sean inestables, es decir, que sean sensibles a perturbaciones en los datos de entrenamiento (pequeños cambios en los datos provocan grandes cambios en los resultados).

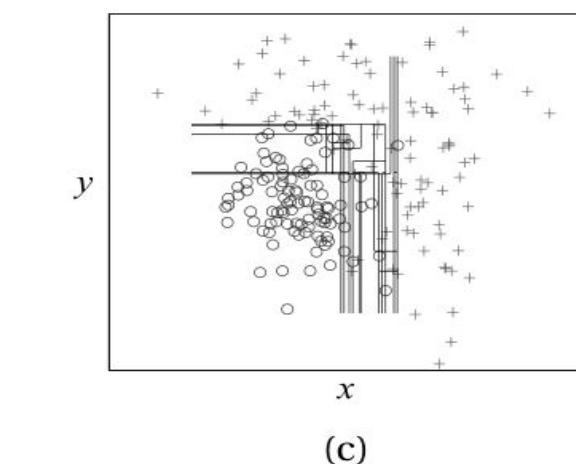
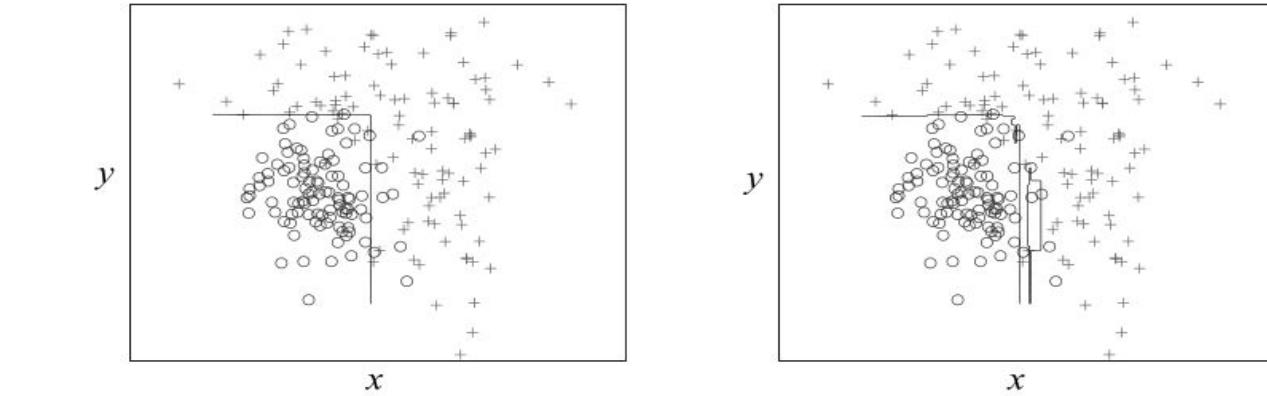
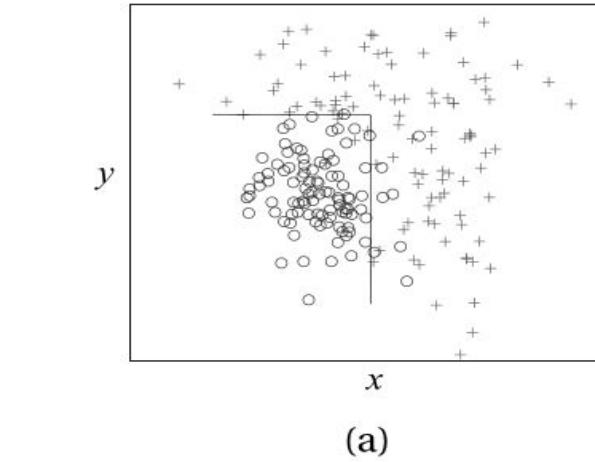
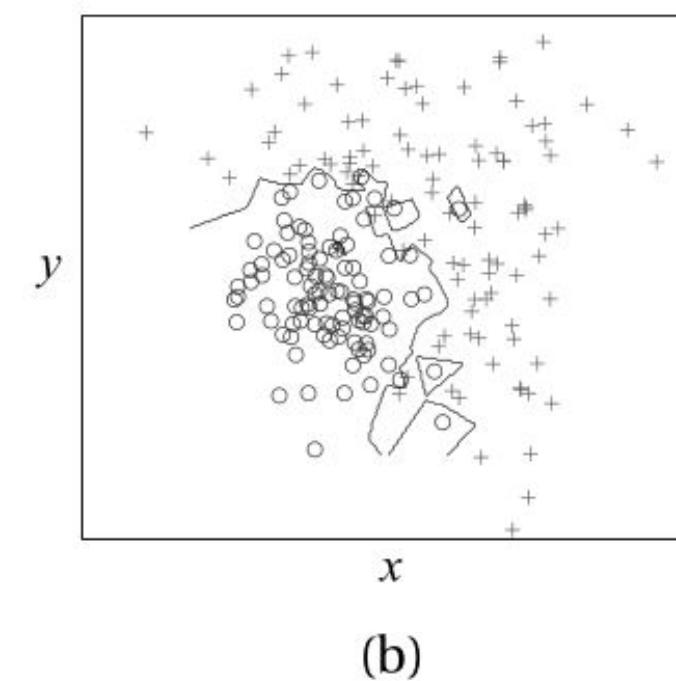
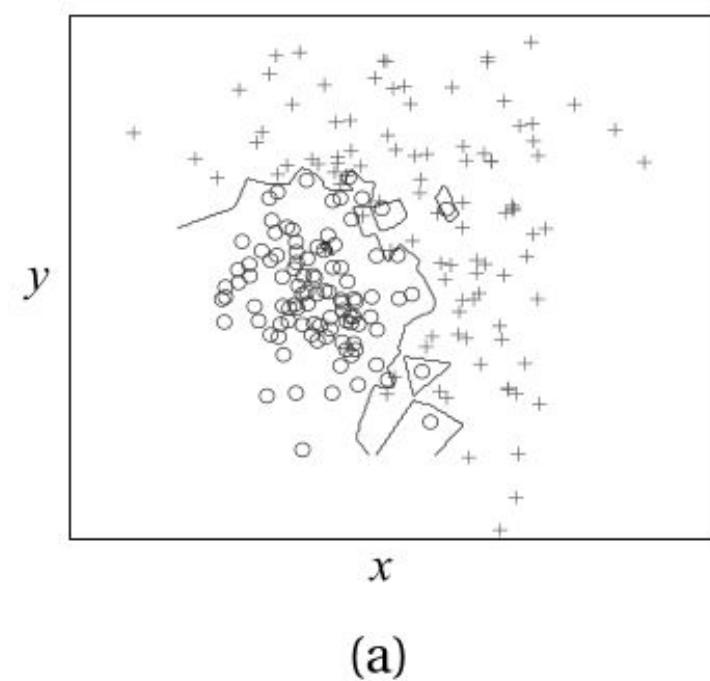


FIGURE 3.5: Decision boundaries of (a) 1-nearest neighbor classifier, and (b) Bagging of 1-nearest neighbor classifiers, on the *three-Gaussians* data set.

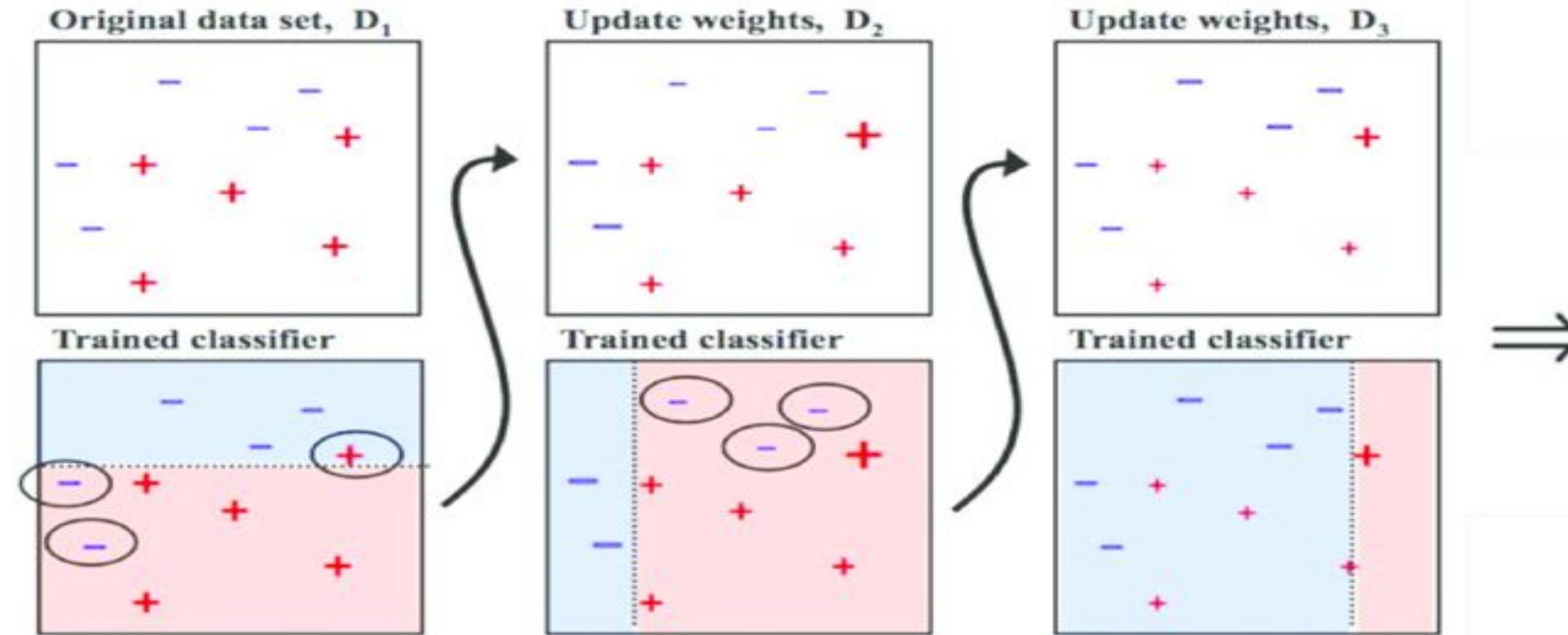
Modelos estables son K-NN, SVM, regresión lineal, etc. e inestables son los árboles decisión, RNA, ...

FIGURE 3.2: Decision boundaries of (a) a single decision tree, (b) Bagging and (c) the 10 decision trees used by Bagging, on the *three-Gaussians* data set.

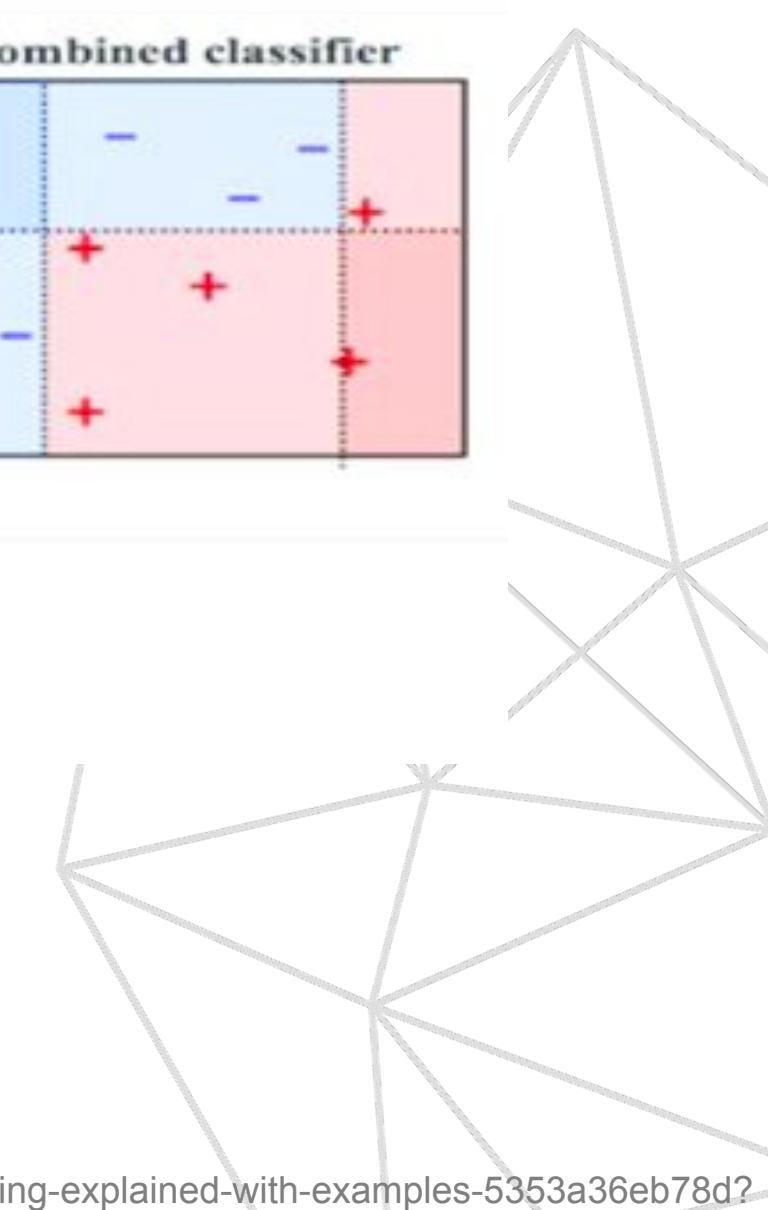
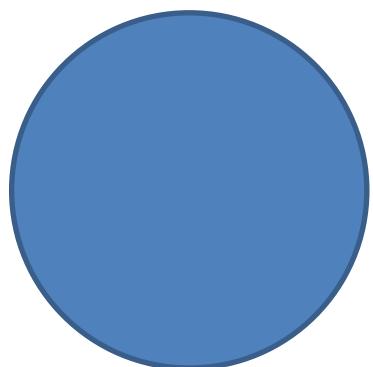


Ensembles secuenciales: colaboración

Para que un ensemble secuencial sea efectivo, un modelo base debe centrarse en los fallos que haya cometido el modelo base anterior. La manera más directa es dando mayor peso a esos patrones erróneos:



- El modelo 1 parte de cero, y se equivocará en ciertos patrones.
- El modelo 2 se centra en clasificar bien lo fallado por el modelo 1.
- El modelo 3 hace lo mismo para los fallos del 2.
- La combinación inteligente de los 3 modelos no falla.





Ensembles secuenciales: boosting

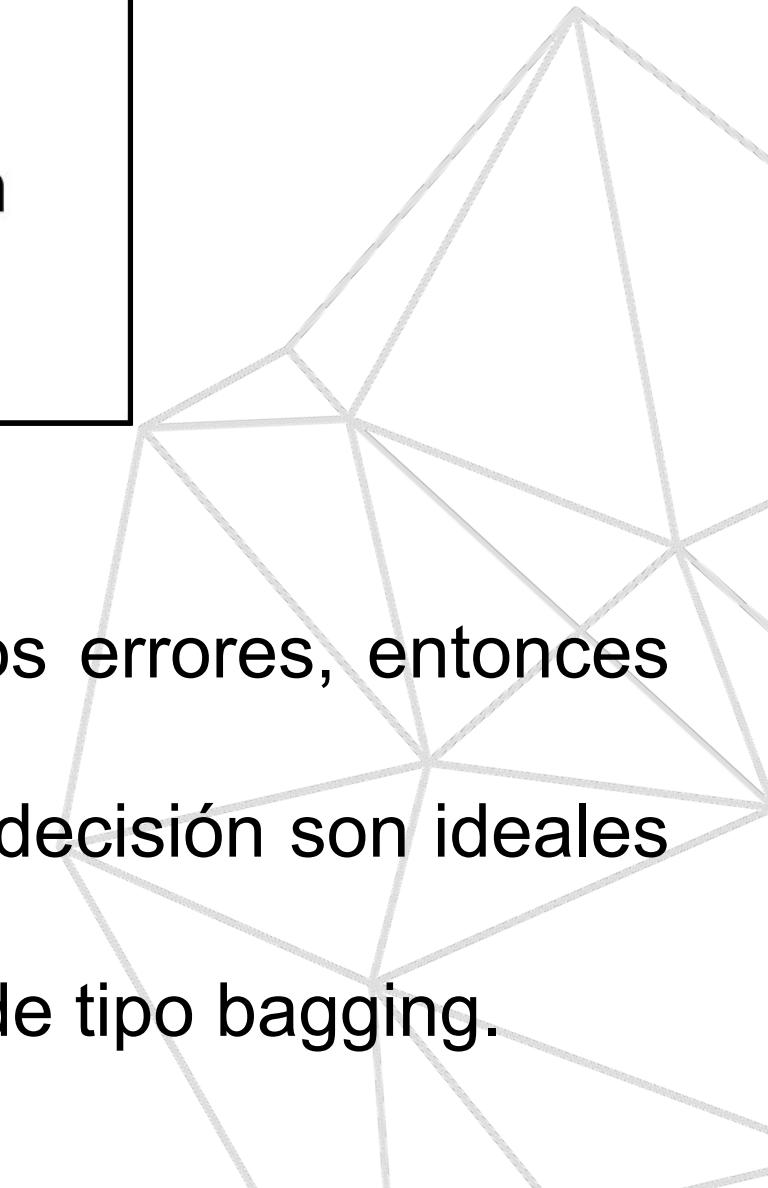
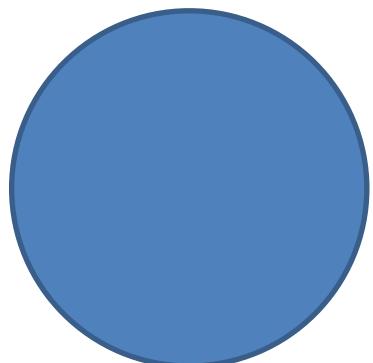
Este procedimiento es lo que se conoce como boosting:

Entrenar un modelo inicial

Repetir T veces:

1. Calcular los errores del ensemble
2. Modificar la distribución de muestreo de los datos
 Más error → más probabilidad
3. Extraer una nueva muestra de los datos con la nueva distribución
4. Entrenar un nuevo modelo con la nueva muestra
5. Añadir el modelo al ensemble

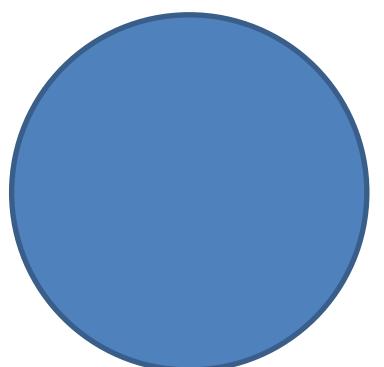
- Aquí no es tan crítico que los modelos sean inestables (pero sí deseable).
- Lo lógico es que sean también modelos débiles (si de partida hay pocos errores, entonces poco se puede corregir).
- Por ello, veremos que tanto en bagging como en boosting los árboles de decisión son ideales (débiles, inestables, rápidos de entrenar...).
- Estos modelos suelen tener más problemas de sobreaprendizaje que los de tipo bagging.



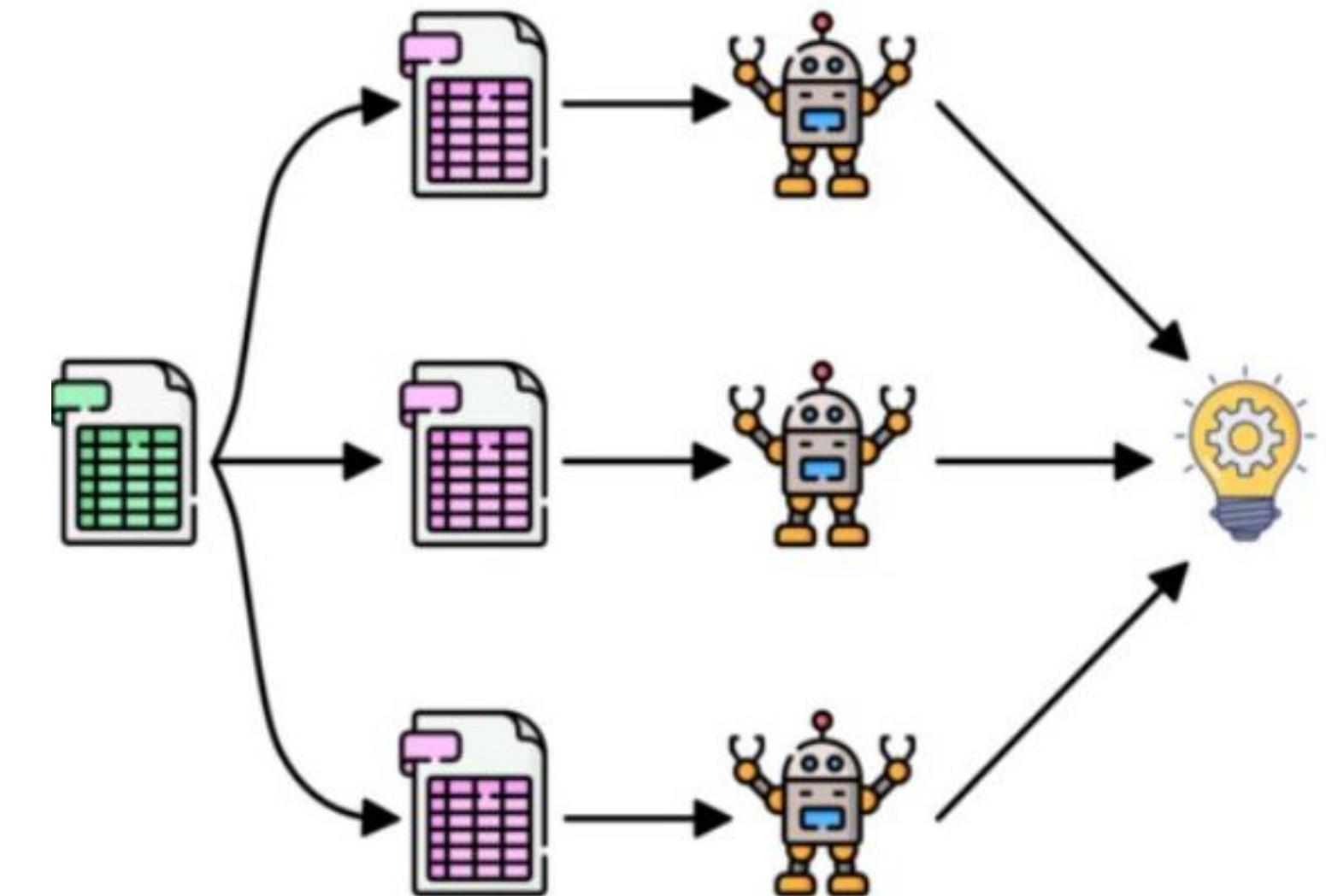
03

Tipos de combinaciones

Se estudian las distintas formas de calcular la predicción del modelo ensemble a partir de las predicciones de los modelos base.



Bagging

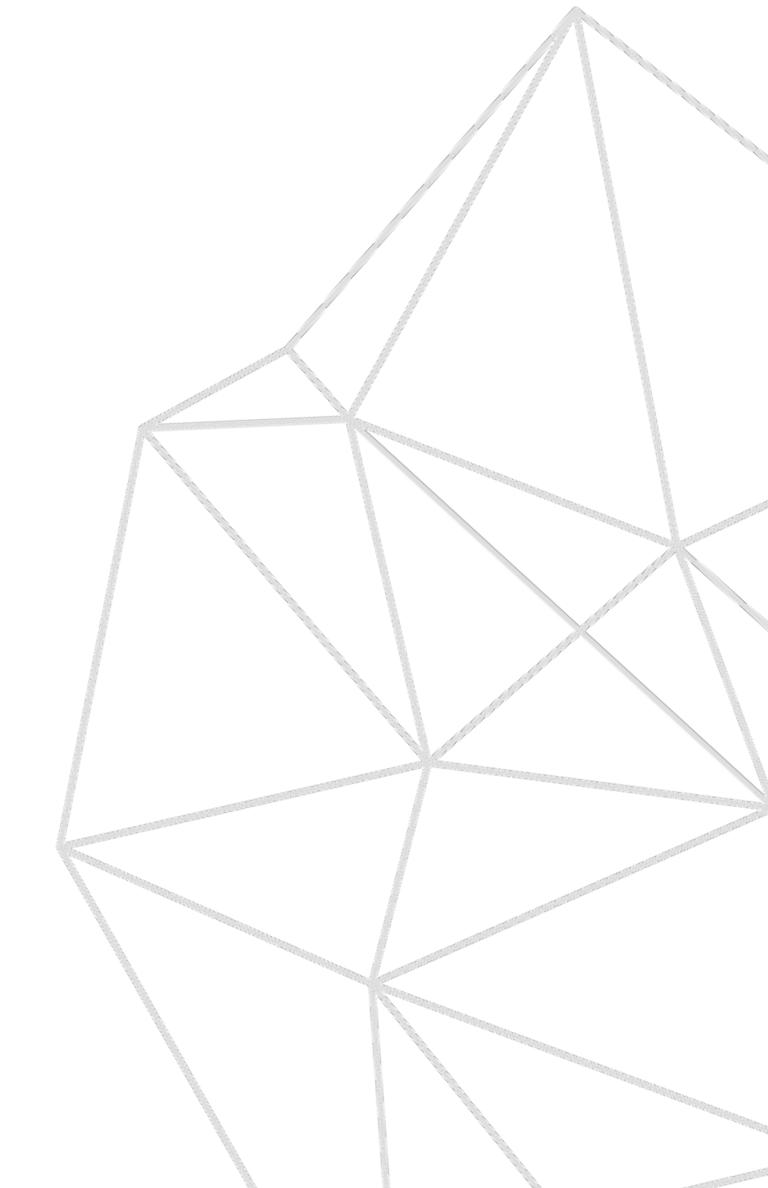
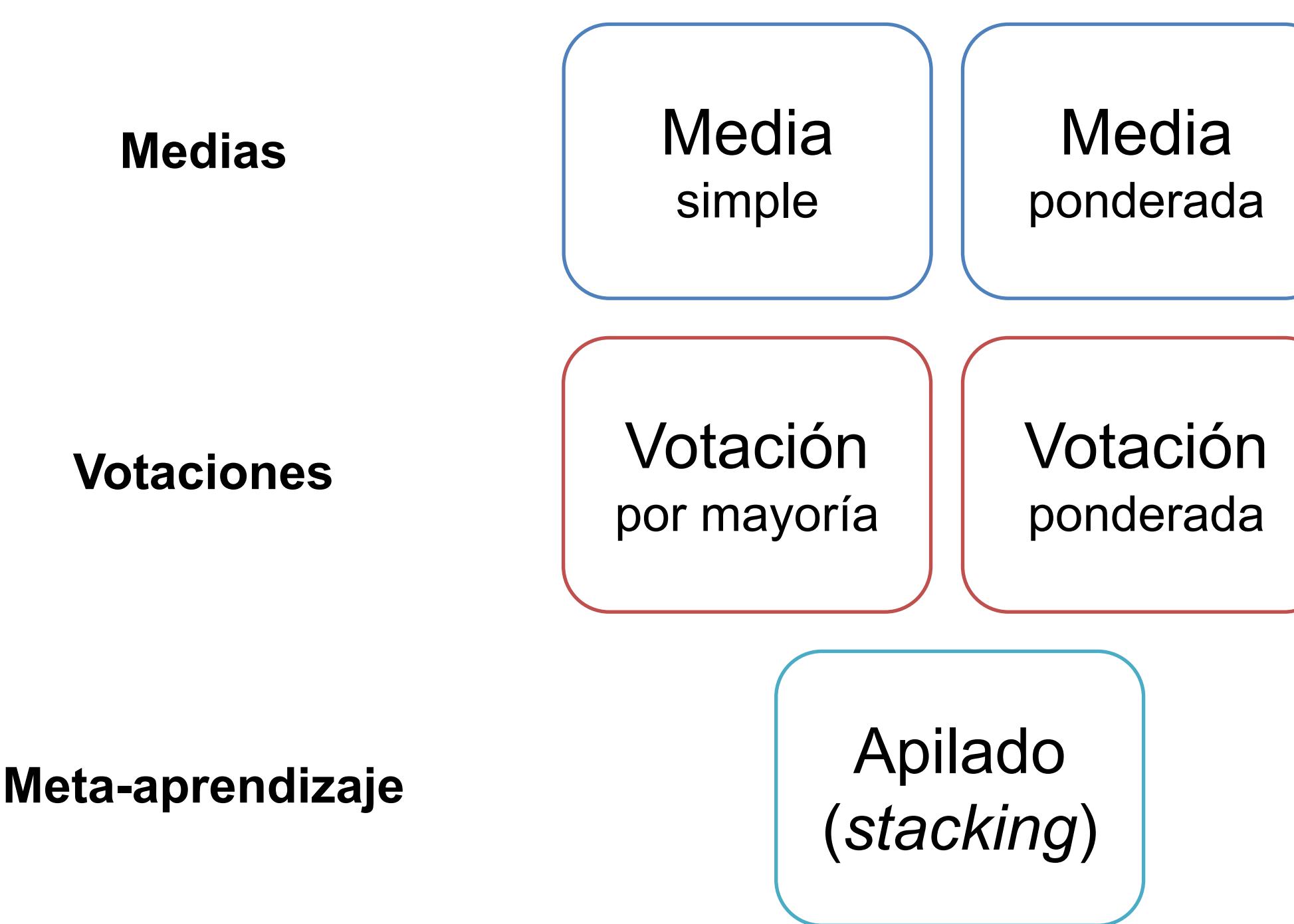
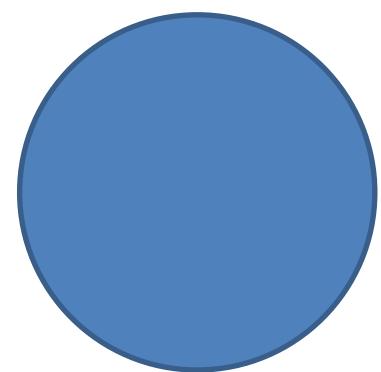


Parallel



Estrategias de combinación

Ya estemos usando un ensemble paralelo o secuencial, siempre hay que combinar al final las decisiones individuales de cada modelo base. Existen varias estrategias posibles:





Estrategias de combinación: Medias

Existen 2 tipos de medias utilizadas frecuentemente en problemas de regresión (en clasificación se usan votaciones porque la media puede no ser una clase):

Media simple

$$f(x) = \frac{1}{T} \sum_{i=1}^T h_i(x)$$

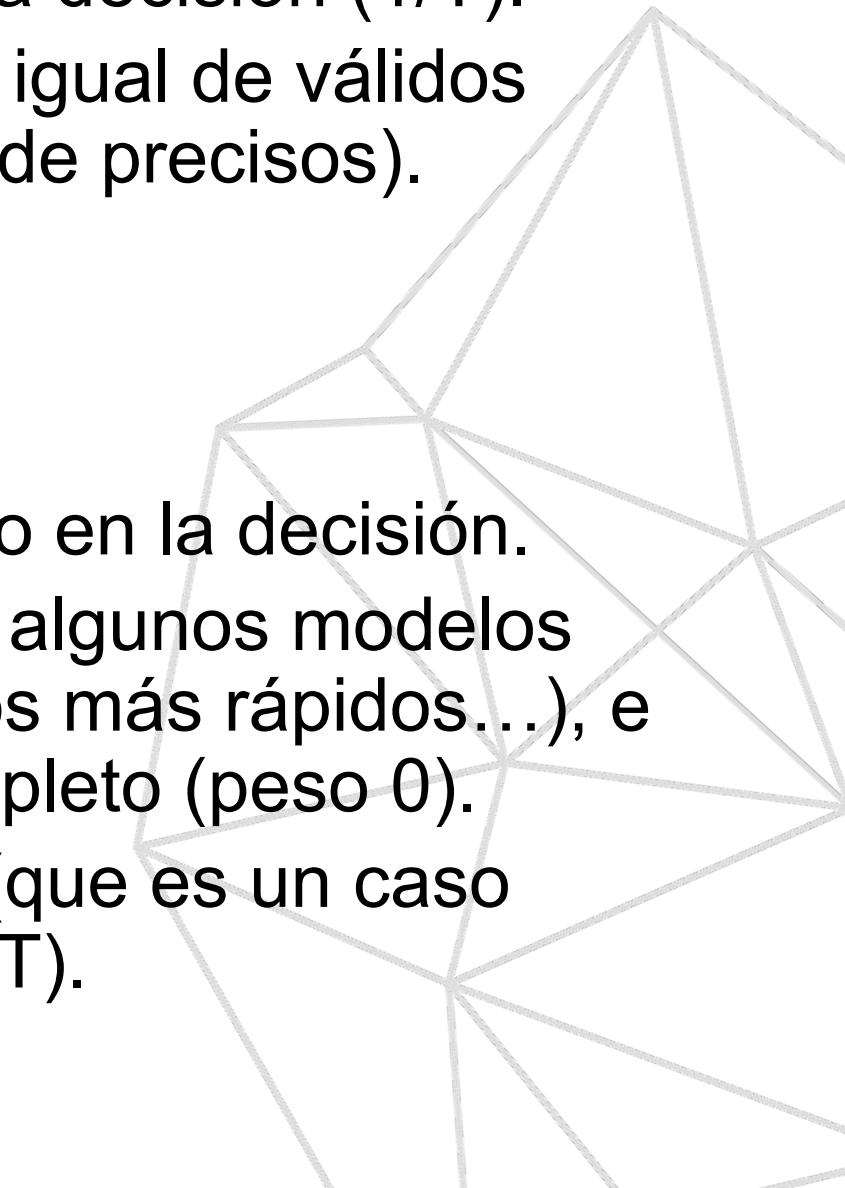
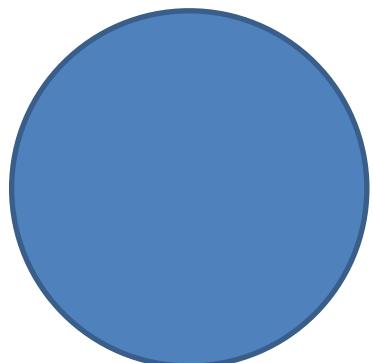
- Cada modelo tiene igual peso en la decisión ($1/T$).
- Asume tácitamente que todos son igual de válidos (aunque puede que no sean igual de precisos).

Media ponderada

$$f(x) = \sum_{i=1}^T w_i h_i(x),$$

$$w_i \geq 0 \quad \forall i, \quad \sum_{i=1}^T w_i = 1$$

- Cada modelo tiene un peso distinto en la decisión.
- Ello permite dar preponderancia a algunos modelos frente a otros (los más precisos, los más rápidos...), e incluso descartar algunos por completo (peso 0).
- Más general que la media simple (que es un caso particular con todos los pesos a $1/T$).





Estrategias de combinación: Votaciones

Existen 2 tipos de votaciones que siguen la misma filosofía que las medias. La única diferencia es que ahora los modelos base dan una salida por cada clase posible c (mientras que en regresión la salida es única):

Votación por mayoría

$$f(x) = c_{\operatorname{argmax}_j \sum_{i=1}^T h_i^j(x)}$$

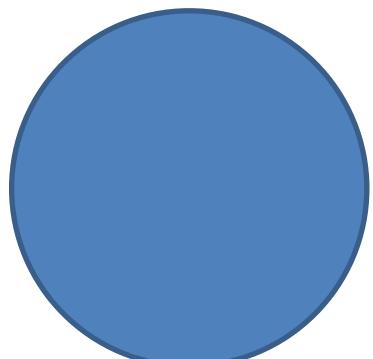
- Para un patrón x gana la clase j que haya resultado como clase estimada por más modelos.
- Asume tácitamente que todos son igual de válidos (aunque puede que no sean igual de precisos).
- Si las salidas de cada modelo son binarias hablamos de hard voting, y si no lo son (pero suman 1 al ser probabilidades de pertenencia a la clase) es soft voting.

Votación ponderada

$$f(x) = c_{\operatorname{argmax}_j \sum_{i=1}^T w_i h_i^j(x)}$$

$$w_i \geq 0 \quad \forall i, \sum_{i=1}^T w_i = 1$$

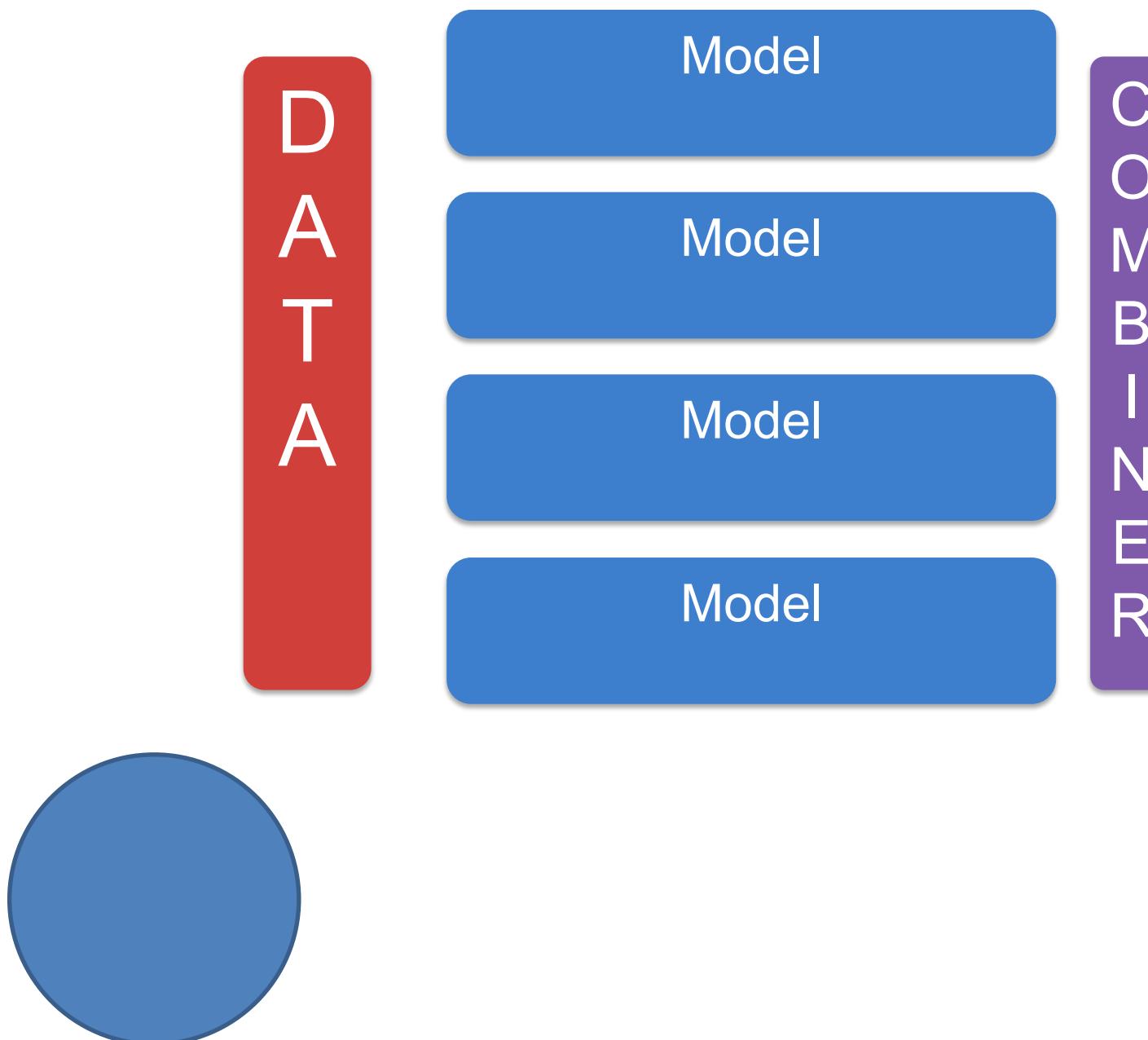
- Para un patrón x gana la clase j que haya resultado como clase estimada por más modelos, donde lo que dice cada modelo viene ponderado por un peso.
- Igual que para la votación por mayoría, hablaremos de hard o soft voting en función de si las salidas de los modelos base son binarias o probabilísticas.
- Por mayoría es el caso particular con todos los pesos a $1/T$.





Estrategias de combinación: Stacking

Claramente las versiones ponderadas en medias y votaciones son superiores a las no ponderadas.
¿Podemos de alguna forma aprender los pesos óptimos de la combinación?



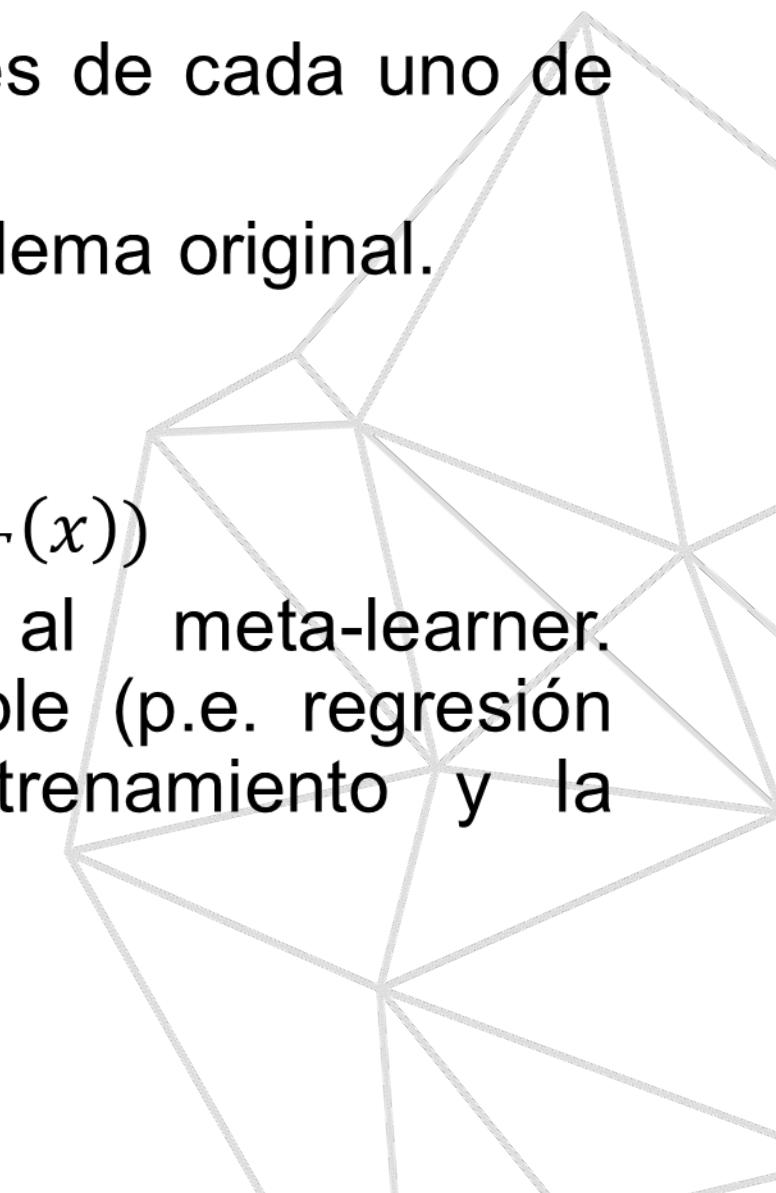
Sí, si consideramos que el combinador a su vez es un modelo (meta-learner o metamodelo) que podemos entrenar con los datos. Sus características son

- Recibe como entradas las predicciones de cada uno de los modelos base.
- Tiene como targets los targets del problema original.

Más formalmente, la salida final sería

$$H(x) = m(h_1(x), h_2(x), \dots, h_T(x))$$

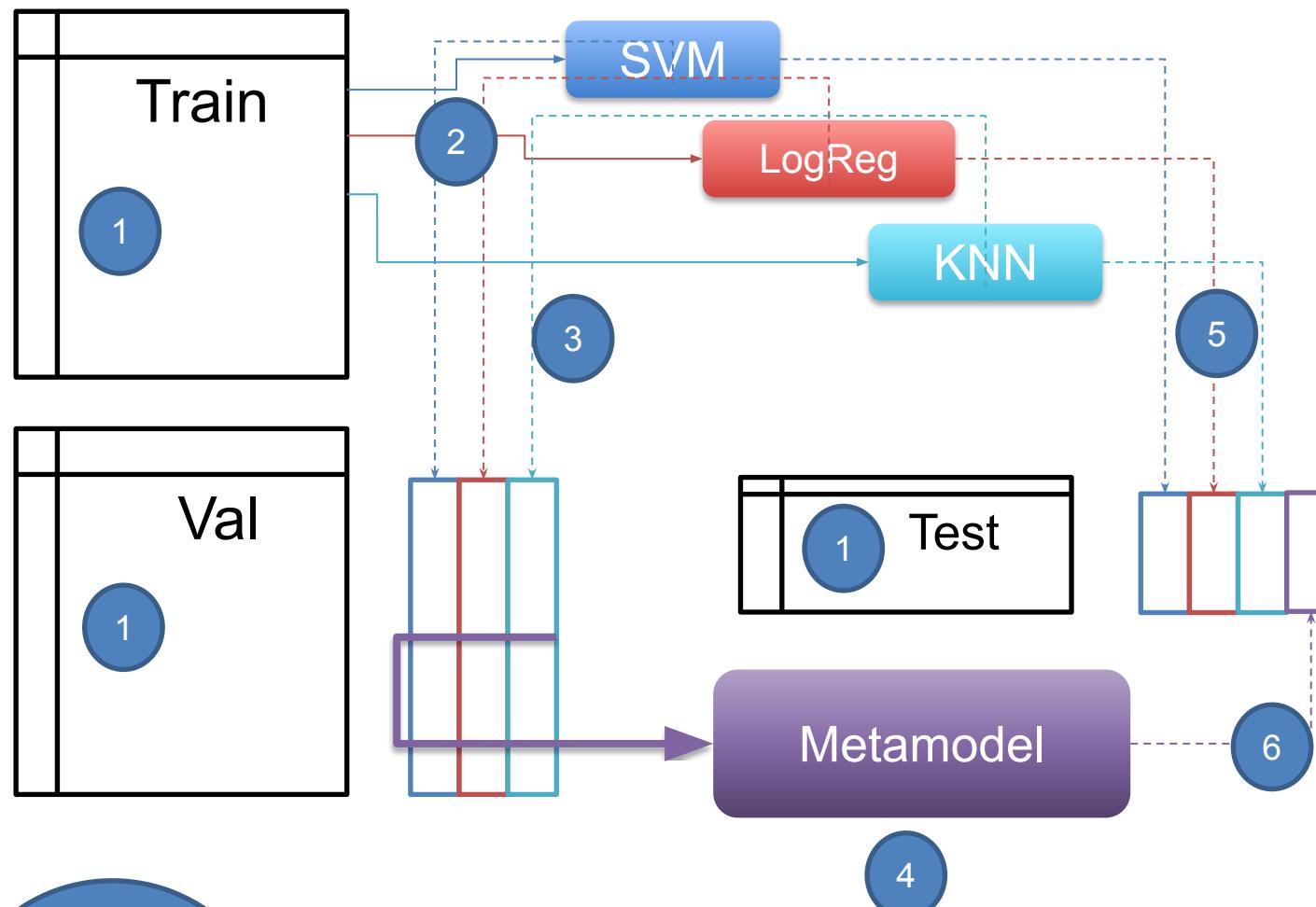
Donde la función m representa al meta-learner. Normalmente se utiliza un modelo simple (p.e. regresión logística), por la complejidad del entrenamiento y la propensión al sobreajuste del conjunto.





Entrenamiento de stacking

Si se tienen suficientes datos, conviene entrenar los modelos base con una parte de los datos (entrenamiento), y el metamodelo con la parte restante (validación). El procedimiento completo sería:



Paso previo

1. Dividir los datos en entrenamiento, validación y test.

Entrenamiento

2. Con los datos de train entrenar los modelos base (CV si se desea)
3. Con los modelos base ya entrenados, generar sus predicciones para los datos de validación.
4. Entrenar el metamodelo con dichas predicciones.

Test

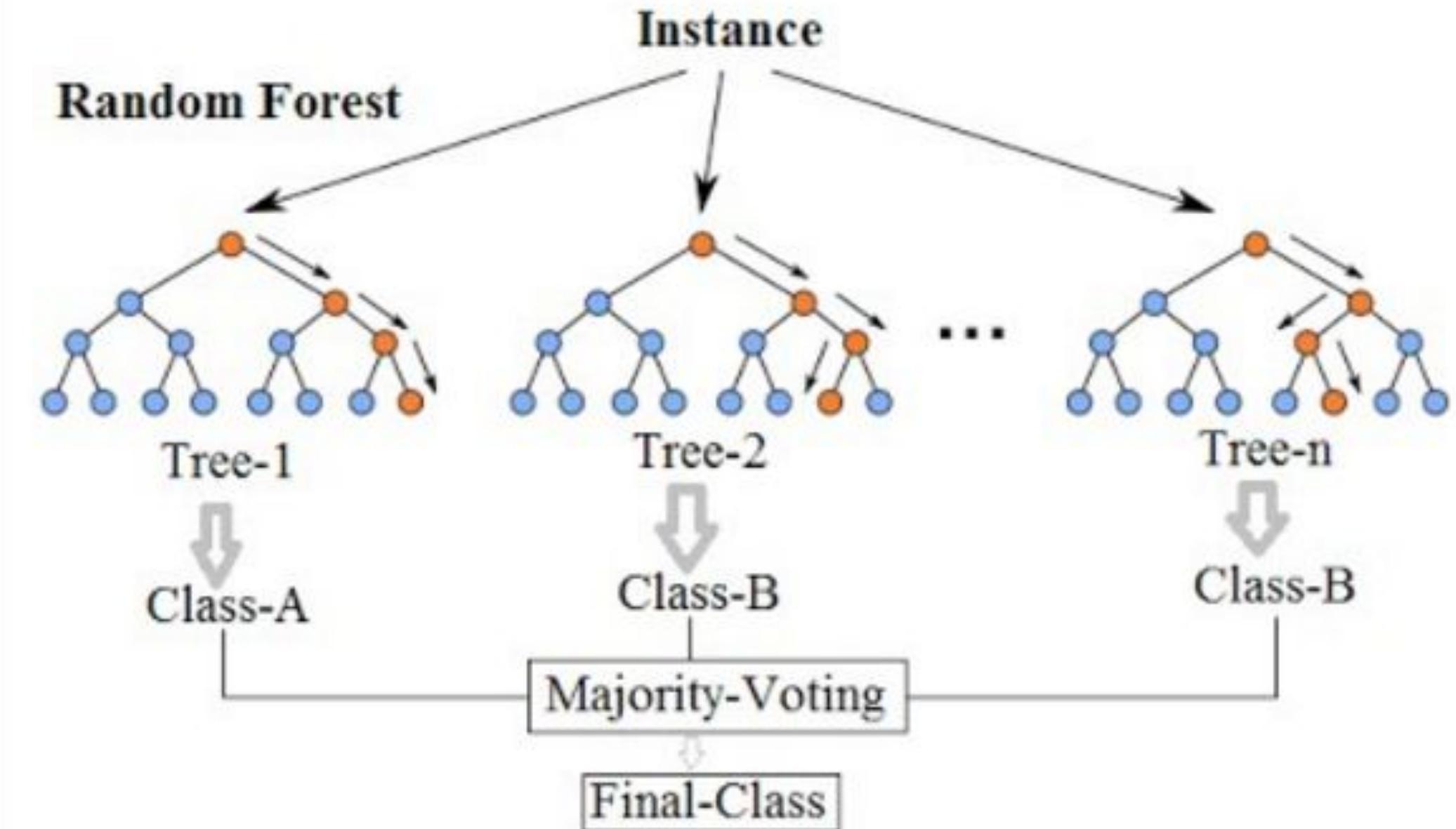
5. Emitir predicciones de los modelos base para los datos de test
6. Emitir predicciones del metamodelo a partir de las predicciones base de test como entradas.

04

Algoritmos de bagging

Presentación de Random Forest y Extremely Randomized Trees.

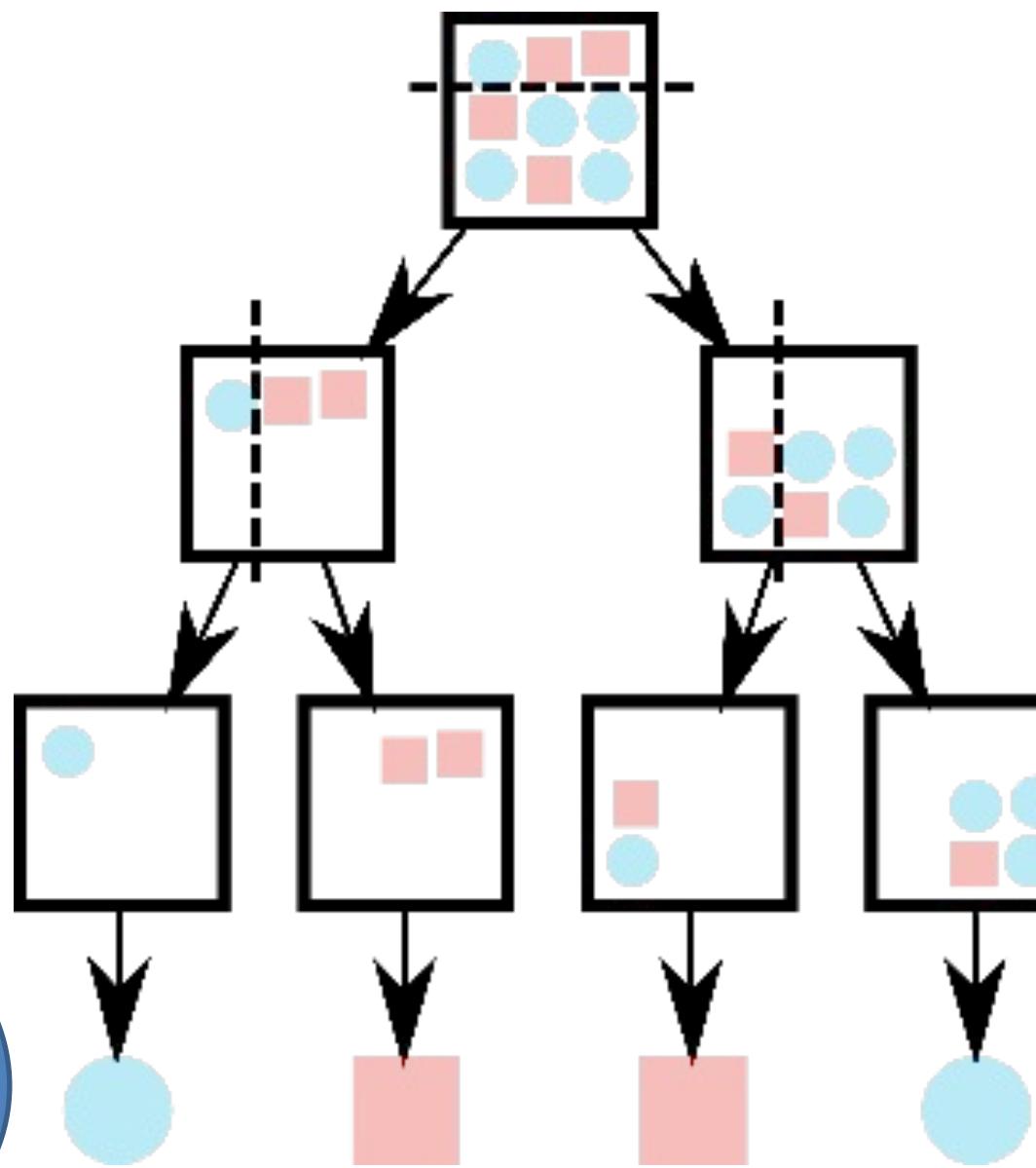
Random Forest Simplified





Random Forest: Introducción

El Random Forest (RF) es una técnica de bagging de árboles de decisión (árboles sobre muestras Bootstrap). Se caracteriza por:

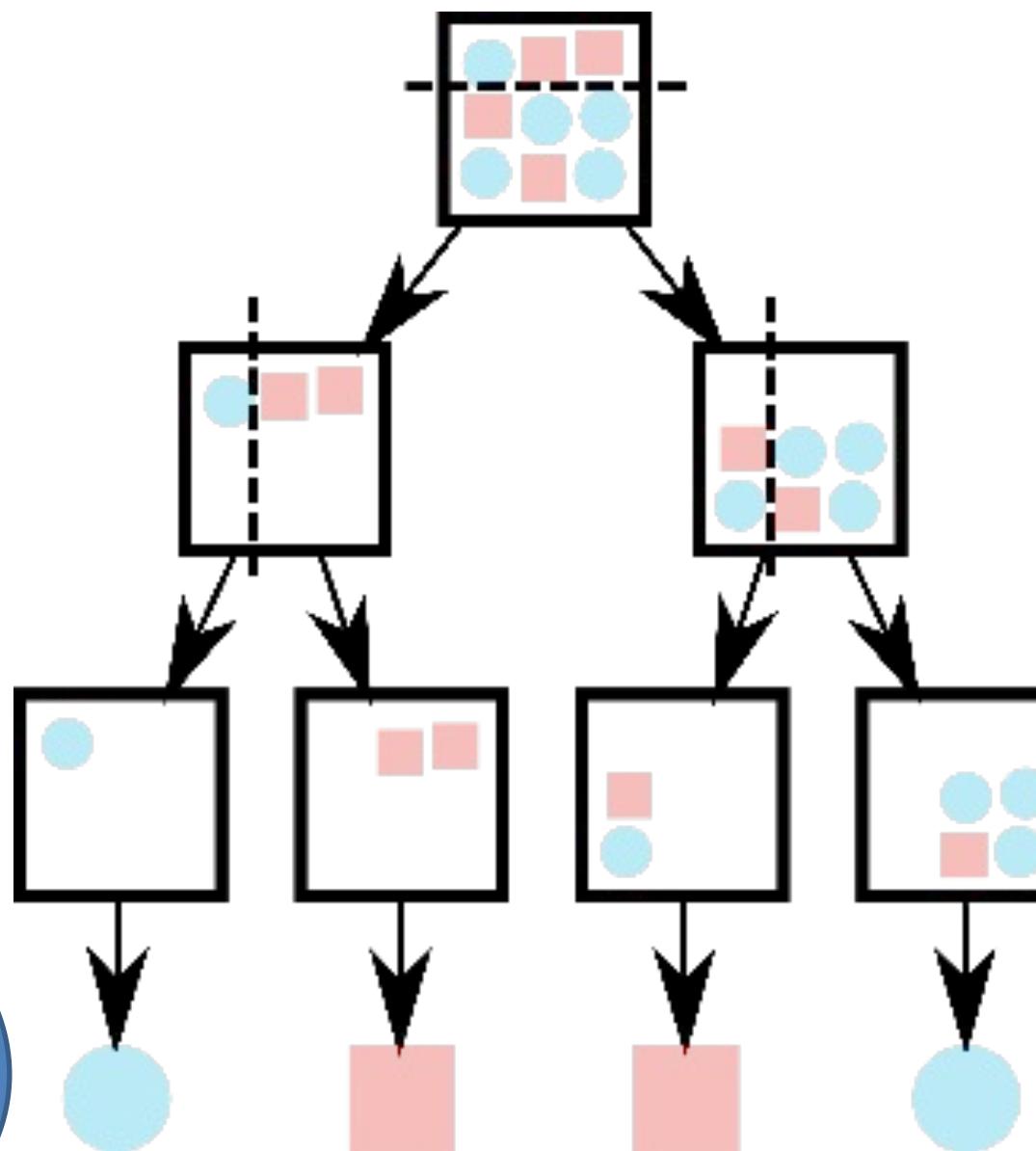


- La combinación de los árboles se realiza mediante media simple (en problemas de regresión) o votación mayoritaria (en clasificación).
- Al ser los árboles modelos inestables favorecen la diversidad del ensemble.
- Además, la ventaja de los árboles es que es fácil controlar su debilidad/fortaleza (mediante parámetros de profundidad, patrones en cada hoja, etc.): se puede regular fácilmente la precisión del ensemble.
- Los árboles son rápidos de entrenar, paralelizables y fácilmente interpretables.



Random Forest: selección de variables

Si una variable tiene mucho poder explicativo casi todos los árboles la usarán, lo que producirá que la diversidad del ensemble se resienta. Para solucionarlo, un RF hace bagging con las variables:



- Cada vez que se considera un corte en un nodo, en lugar de evaluar todas las variables se hace sobre un subconjunto aleatorio.

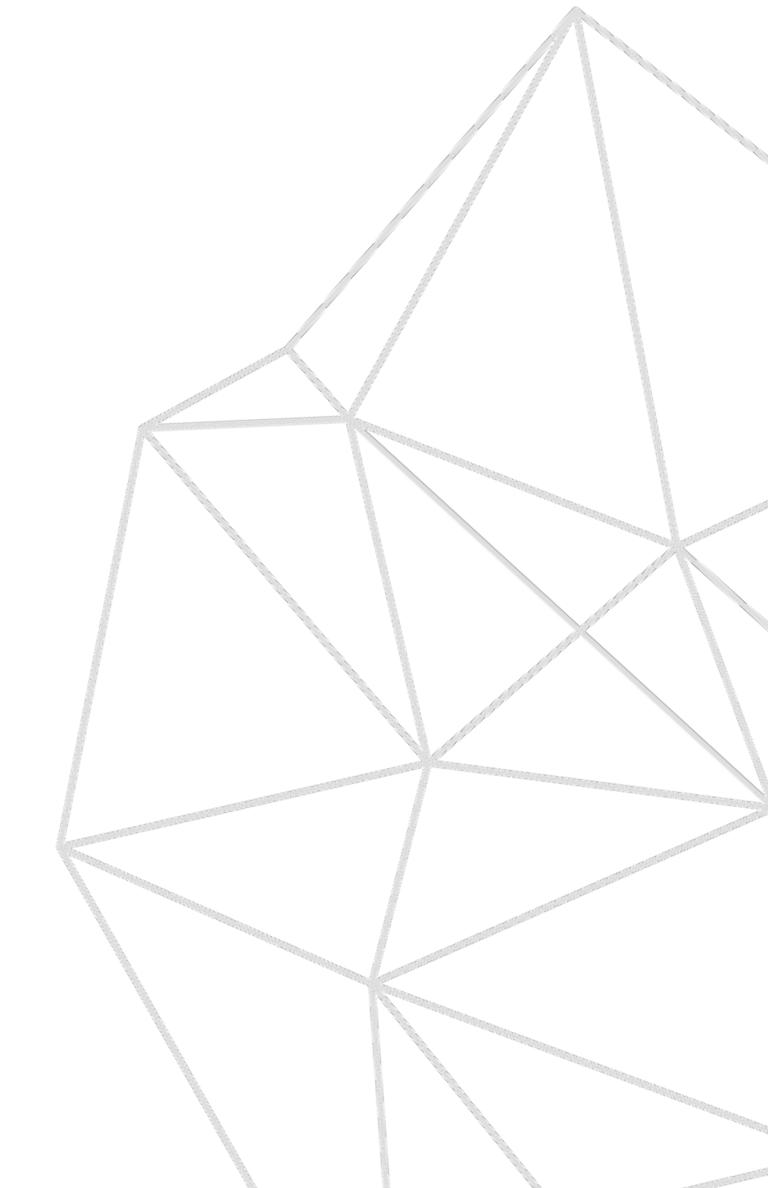
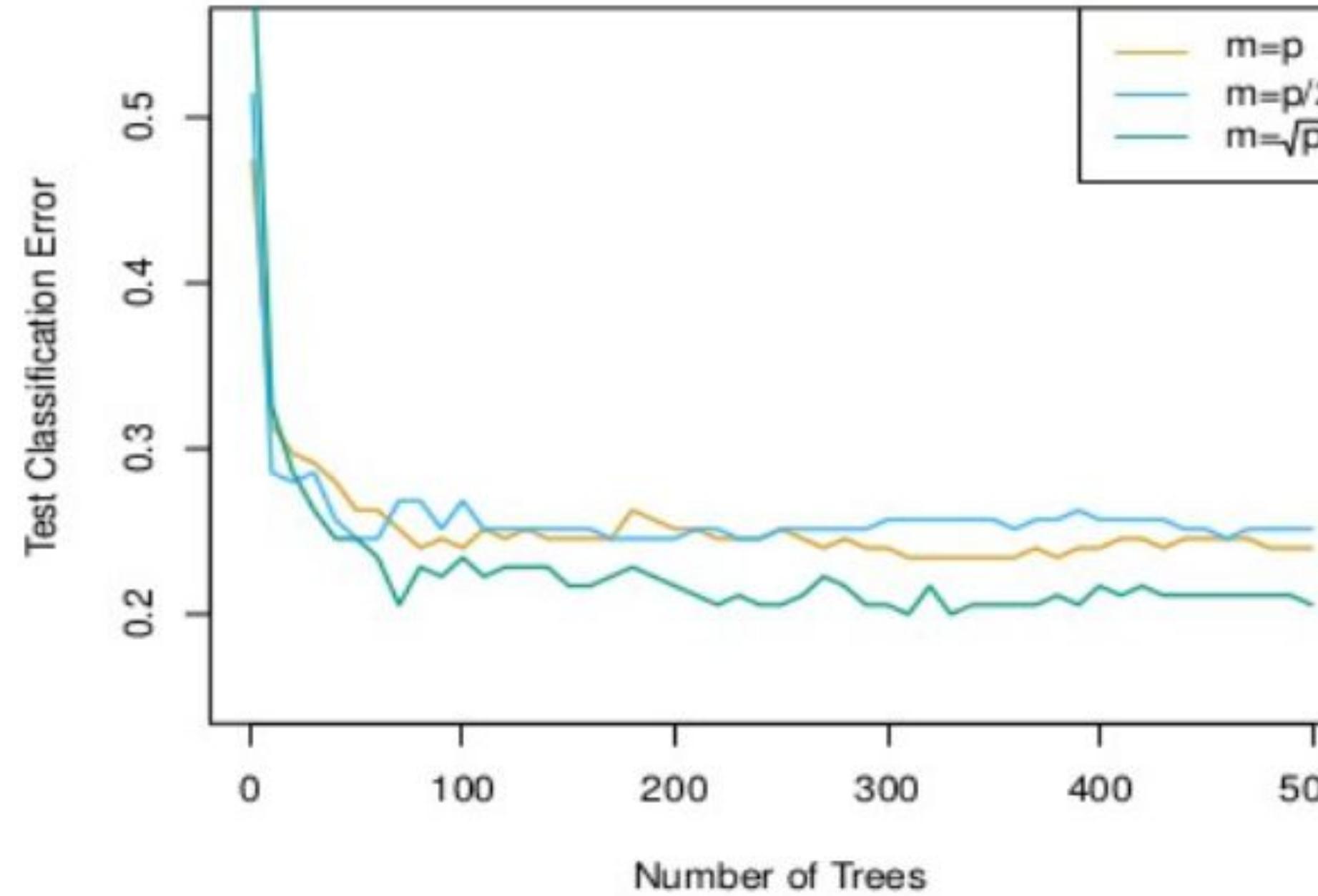
La selección de variables (*random subspace method*) mejora la diversidad de los árboles de decisión. Al decorrelar los árboles se reduce la varianza sin aumentar el sesgo.

Normalmente no es necesario podar los árboles tras entrenarlos, pero sí limitar su complejidad de antemano (ya sea en profundidad, nº de nodos/hojas, etc.). Cuantos más árboles mayor precisión, pero a costa de más tiempo.



Random Forest: selección de variables

En cada Split de la construcción del árbol de decisión se escoge un subconjunto m del conjunto total de atributos p . Por defecto, suele elegirse $m = \sqrt{p}$ o $m = \log p$.

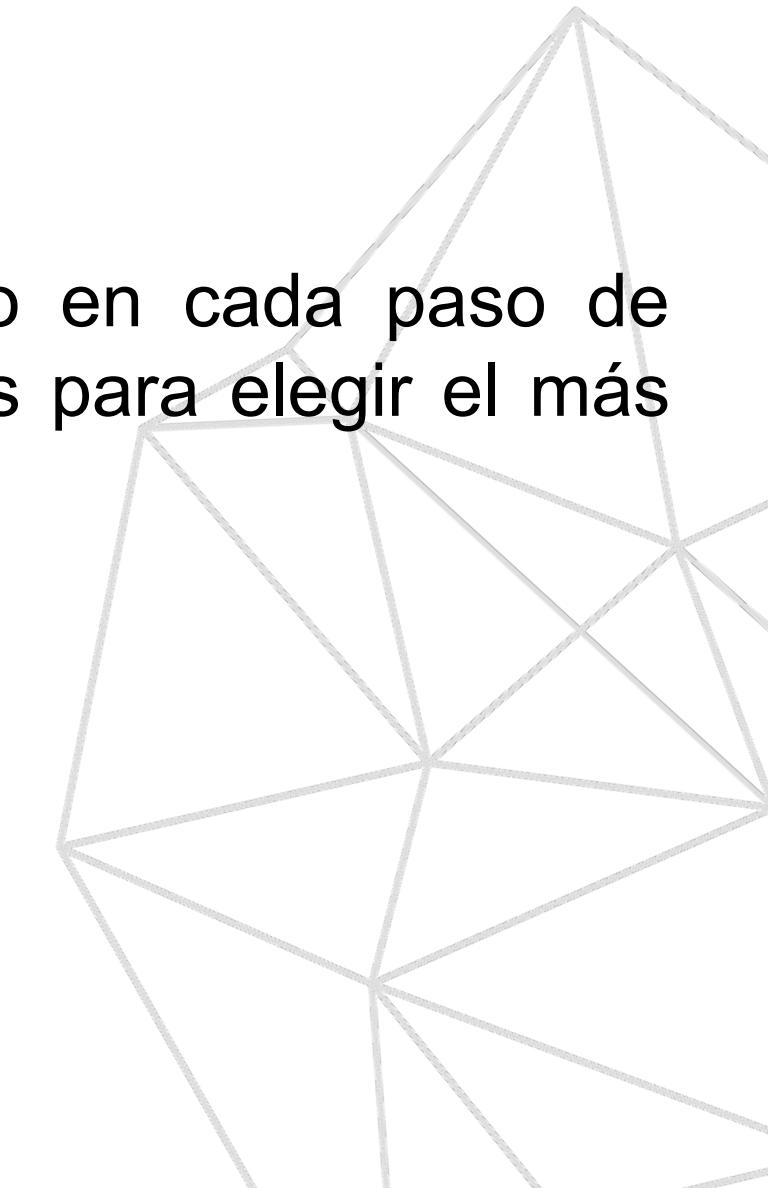
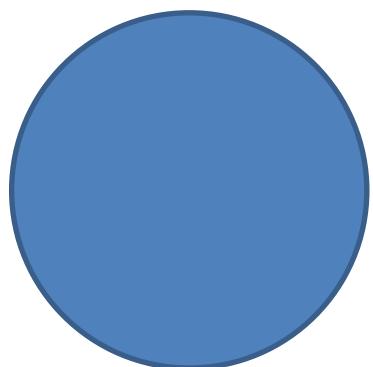




Random Forest: algoritmo

El algoritmo de entrenamiento de Random Forest, sigue el esquema general de bagging incluyendo la selección de variables explicada. Para ello debe fijarse un número de modelos base N y una profundidad máxima de los árboles P :

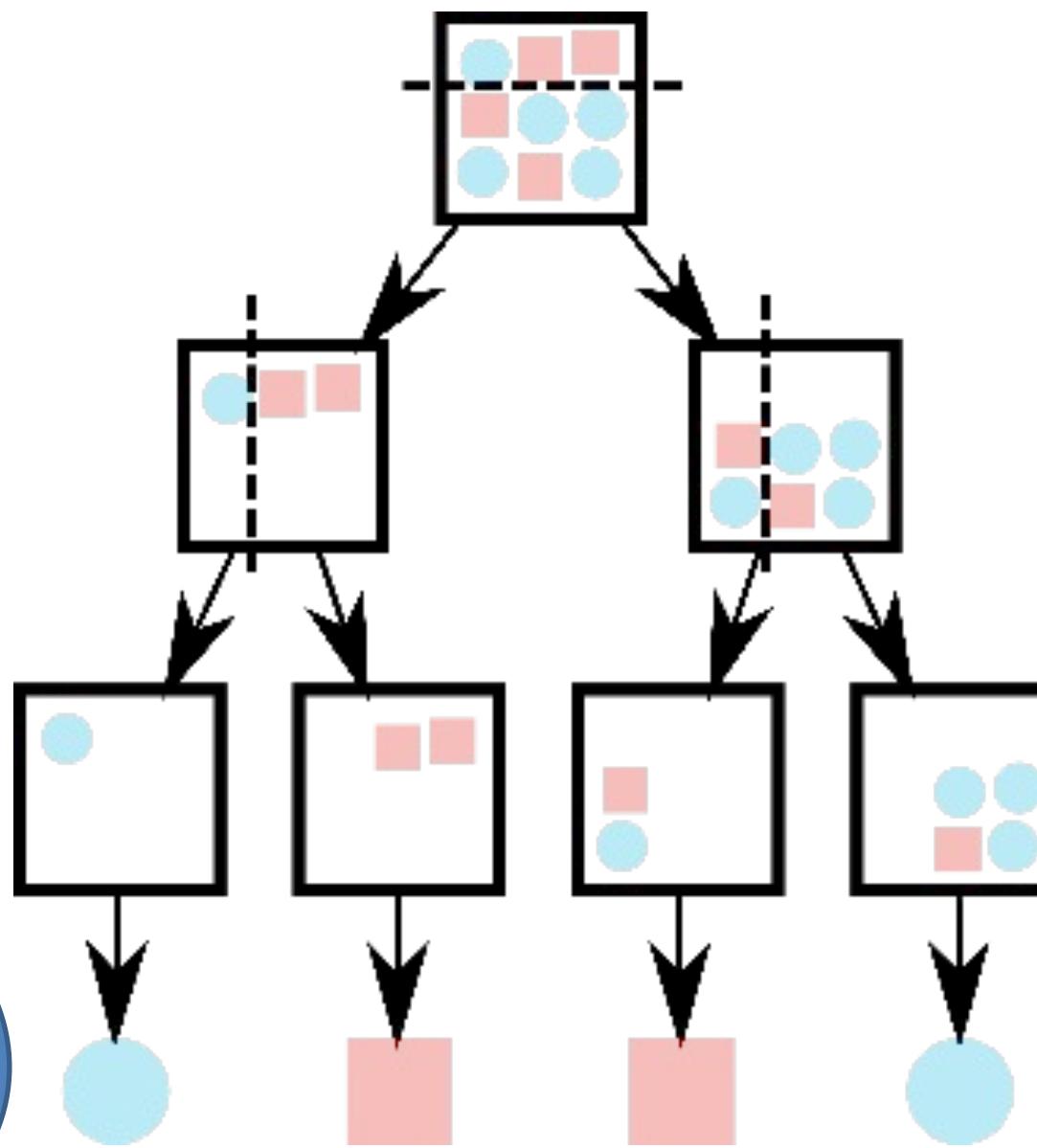
1. Inicializamos el ensemble vacío.
2. Si el ensemble contiene N árboles finalizar, si no, continuar al paso 3.
3. Extraer aleatoriamente una muestra Bootstrap de datos para entrenamiento.
4. Entrenar un árbol de decisión con los datos de entrenamiento, tomando en cada paso de selección de variables una muestra aleatoria del conjunto total de atributos para elegir el más adecuado y estableciendo una profundidad máxima P .
5. Añadir el árbol entrenado al ensemble y volver al paso 2.





Extremely Randomized Trees: Introducción

En un *Extremely Randomized Tree* (ERT) se busca mayor diversidad de los modelos base que conforman el ensemble. Por ello se caracteriza por:



- A la hora de hacer un corte se escoge un subconjunto aleatorio de variables (como en RF).
- Para cada variable elegida en cada split, en lugar de calcular el mejor punto de corte, se escoge aleatoriamente uno.
- Como corte final de cada Split se escoge el mejor de entre los generados aleatoriamente.
- Los extremely randomized trees se construyen sobre todos los datos sin sacar muestras Bootstrap.
- Son más rápidos de entrenar que RF pero requiere árboles más grandes o más modelos base.



Extremely Randomized Trees: algoritmo

El algoritmo de los ERT es igual que el de RF, solo que no se seleccionan muestras Bootstrap y cambia la función para elegir la variables del Split:

Split_a_node(S)

Input: the local learning subset S corresponding to the node we want to split

Output: a split $[a < a_c]$ or nothing

- If **Stop_split(S)** is TRUE then return nothing.
- Otherwise select K attributes $\{a_1, \dots, a_K\}$ among all non constant (in S) candidate attributes;
- Draw K splits $\{s_1, \dots, s_K\}$, where $s_i = \text{Pick_a_random_split}(S, a_i), \forall i = 1, \dots, K$;
- Return a split s_* such that $\text{Score}(s_*, S) = \max_{i=1, \dots, K} \text{Score}(s_i, S)$.

Pick_a_random_split(S, a)

Inputs: a subset S and an attribute a

Output: a split

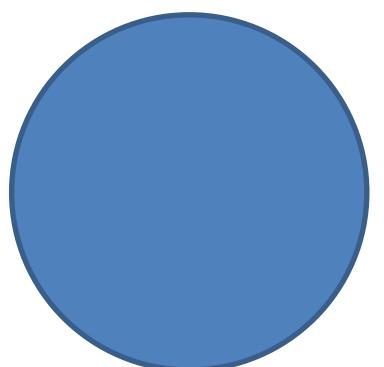
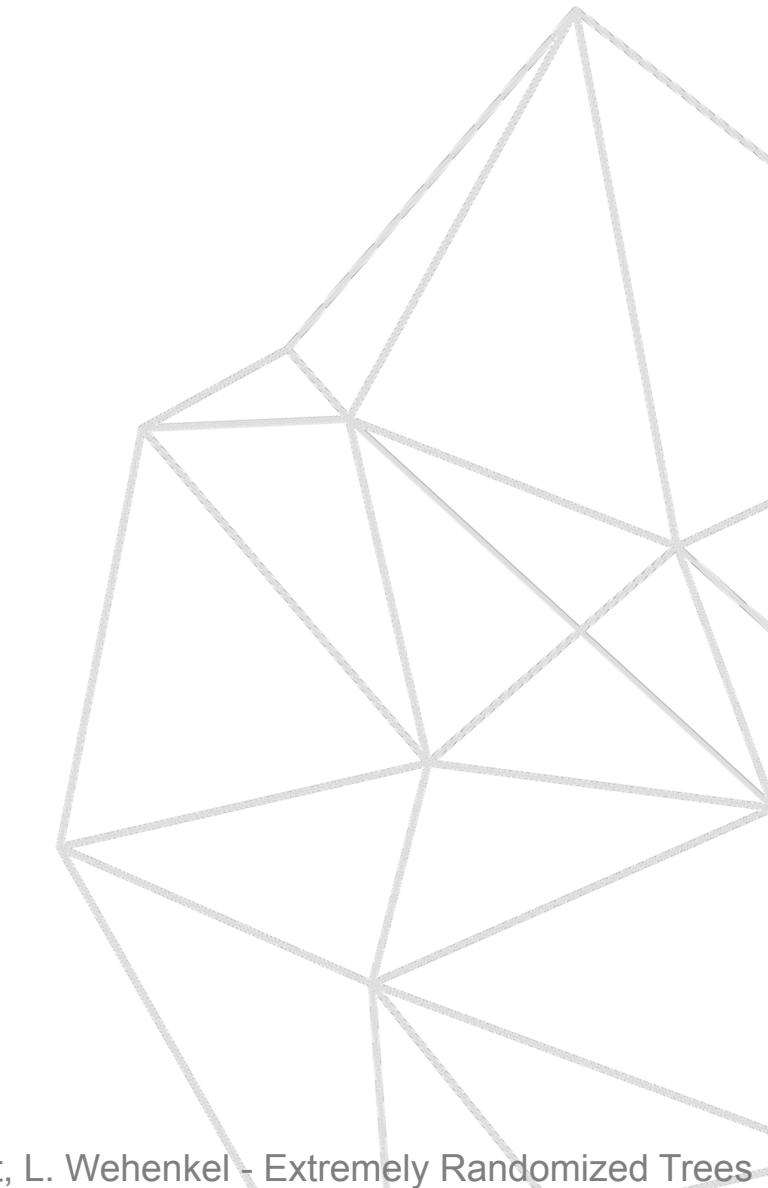
- Let a_{\max}^S and a_{\min}^S denote the maximal and minimal value of a in S ;
- Draw a random cut-point a_c uniformly in $[a_{\min}^S, a_{\max}^S]$;
- Return the split $[a < a_c]$.

Stop_split(S)

Input: a subset S

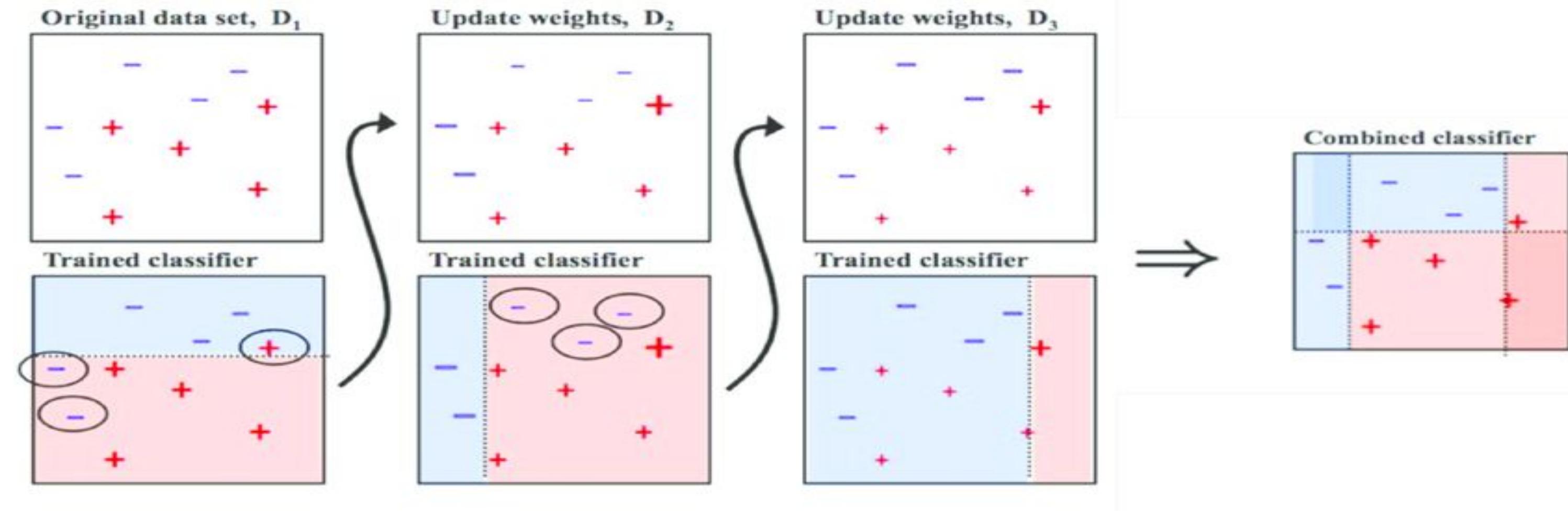
Output: a boolean

- If $|S| < n_{\min}$, then return TRUE;
- If all attributes are constant in S , then return TRUE;
- If the output is constant in S , then return TRUE;
- Otherwise, return FALSE.



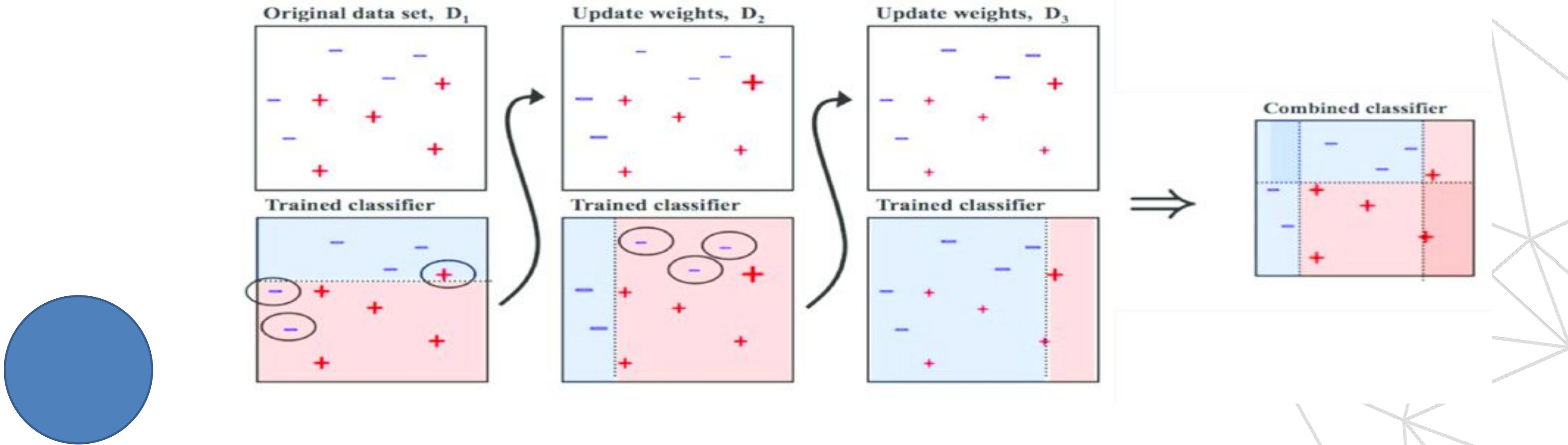
05 Algoritmos de boosting

AdaBoost y Gradient Boosting



AdaBoost: Introducción

AdaBoost es el algoritmo de tipo boosting más popular. Sus modelos base son normalmente árboles (aunque no necesariamente deben serlo, pueden elegirse otros). Todos los patrones (subconjuntos de datos) empiezan con la misma probabilidad de ser elegidos y cada modelo nuevo del ensemble se entrena con un muestreo distinto de los datos.





AdaBoost: algoritmo

Input: Data set $D = \{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \dots, (\mathbf{x}_m, y_m)\}$;

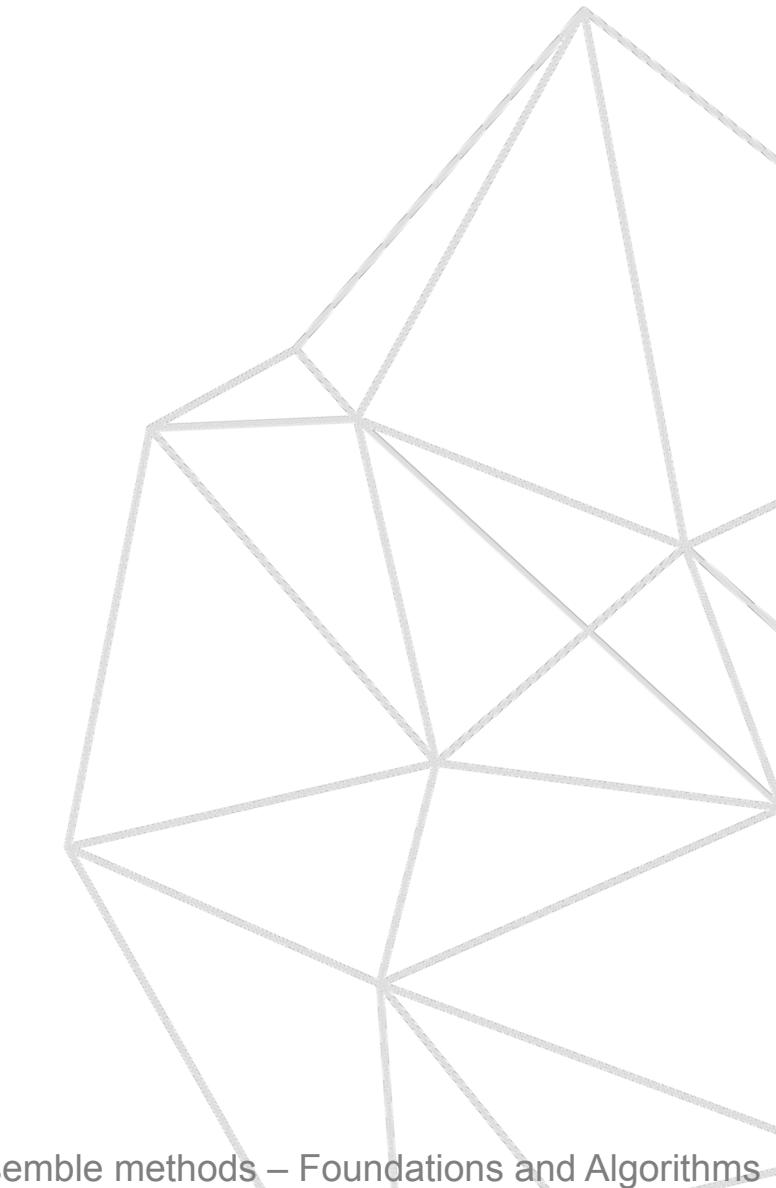
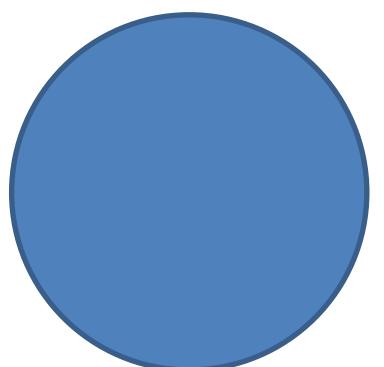
Base learning algorithm \mathcal{L} ;

Number of learning rounds T .

Process:

1. $\mathcal{D}_1(\mathbf{x}) = 1/m$. % Initialize the weight distribution
2. **for** $t = 1, \dots, T$:
3. $h_t = \mathcal{L}(D, \mathcal{D}_t)$; % Train a classifier h_t from D under distribution \mathcal{D}_t
4. $\epsilon_t = P_{\mathbf{x} \sim \mathcal{D}_t}(h_t(\mathbf{x}) \neq f(\mathbf{x}))$; % Evaluate the error of h_t
5. **if** $\epsilon_t > 0.5$ **then break**
6. $\alpha_t = \frac{1}{2} \ln \left(\frac{1-\epsilon_t}{\epsilon_t} \right)$; % Determine the weight of h_t
7. $\mathcal{D}_{t+1}(\mathbf{x}) = \frac{\mathcal{D}_t(\mathbf{x})}{Z_t} \times \begin{cases} \exp(-\alpha_t) & \text{if } h_t(\mathbf{x}) = f(\mathbf{x}) \\ \exp(\alpha_t) & \text{if } h_t(\mathbf{x}) \neq f(\mathbf{x}) \end{cases}$
 $= \frac{\mathcal{D}_t(\mathbf{x}) \exp(-\alpha_t f(\mathbf{x}) h_t(\mathbf{x}))}{Z_t}$ % Update the distribution, where
% Z_t is a normalization factor which
% enables \mathcal{D}_{t+1} to be a distribution
8. **end**

Output: $H(\mathbf{x}) = \text{sign} \left(\sum_{t=1}^T \alpha_t h_t(\mathbf{x}) \right)$

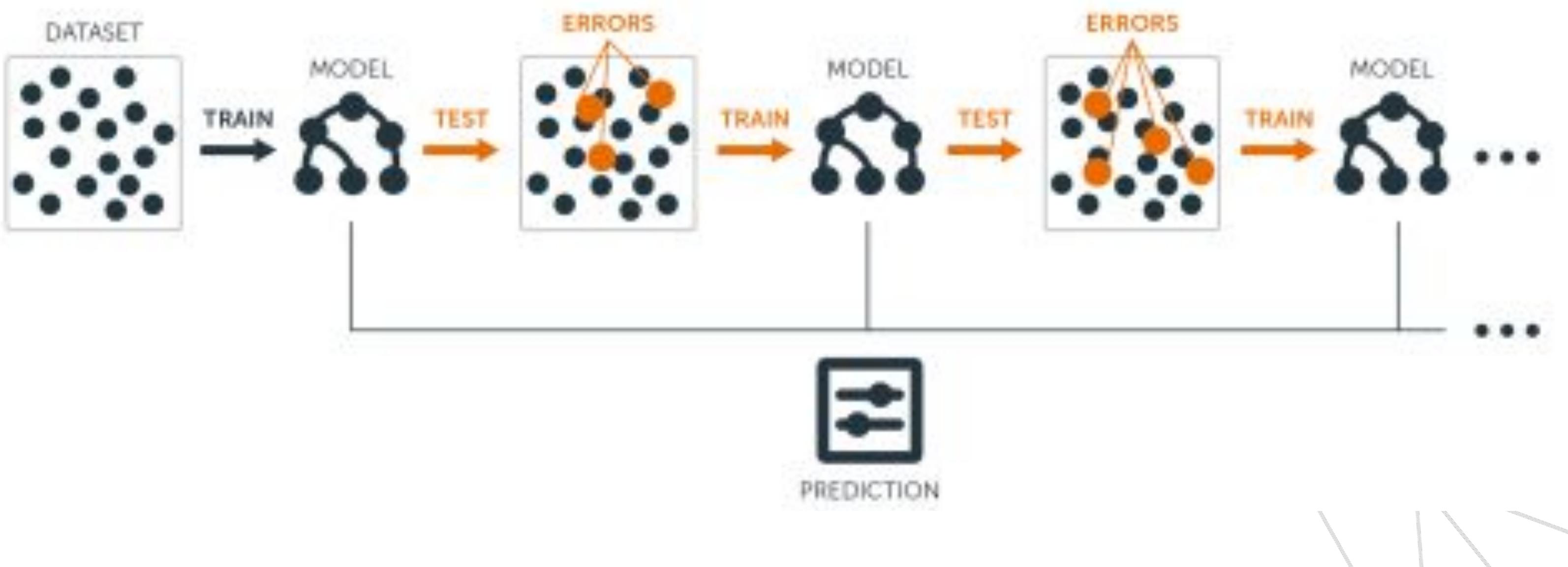




Gradient Boosting: Introducción

Gradient boosting es una alternativa a boosting más general que busca encontrar:

$$f^* = \underset{f \in F}{\operatorname{argmin}} \varphi(f) = \underset{f \in F}{\operatorname{argmin}} \frac{1}{n} \sum_{i=1}^n l(x_i, f(x_i))$$





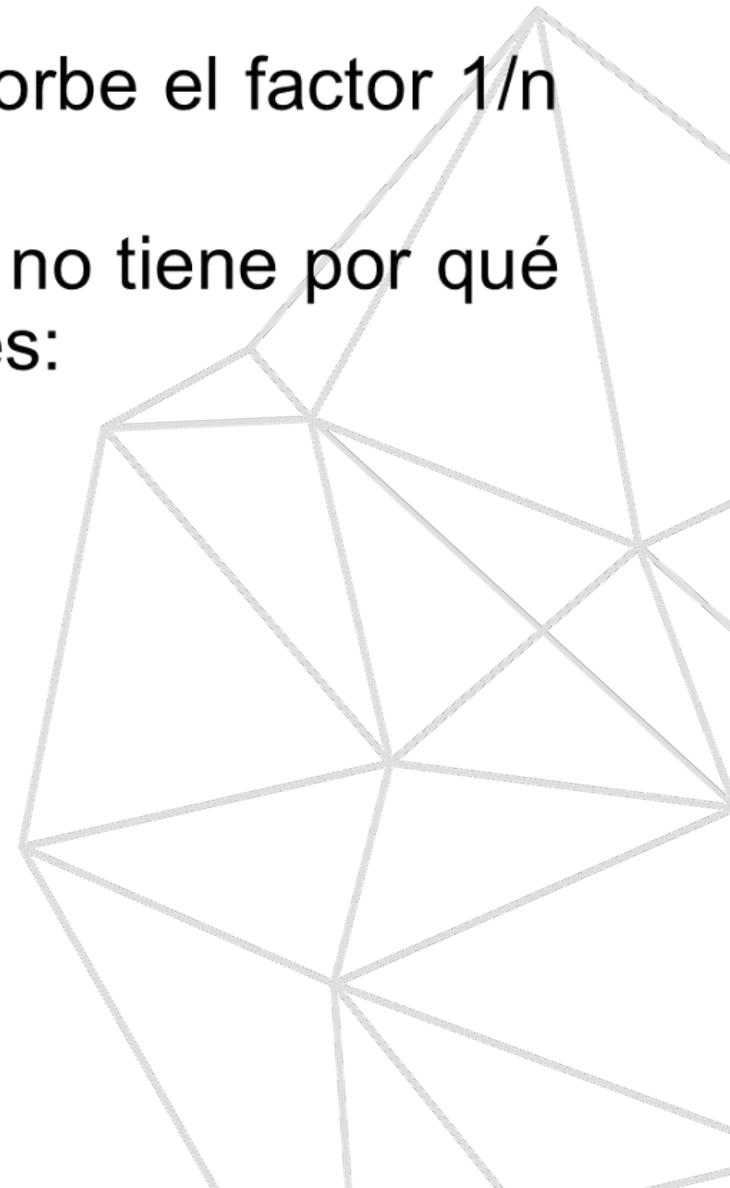
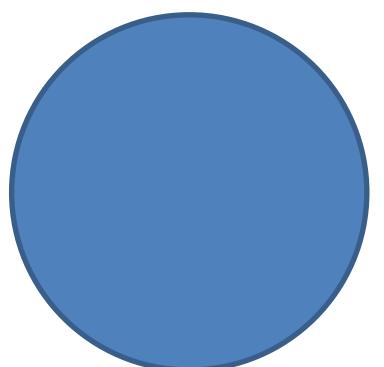
Gradient Boosting: descenso del gradiente

En concreto, el paso de descenso por gradiente sería:

$$f_t = f_{t-1} - \rho_t \nabla_F \varphi(f_{t-1}) = f_{t-1} - \rho_t \sum_{i=1}^n \nabla_F l(x_i, f_{t-1}(x_i))$$

- El modelo inicial del que se parte es f_0 y el nuevo modelo se construye a partir del anterior descendiendo por el gradiente del error.
- La longitud del paso del gradiente viene dada por la tasa de aprendizaje ρ_t (que absorbe el factor $1/n$ de la media).
- Aunque sepamos que f_{t-1} pertenece a la familia de modelos F , después del paso f_t no tiene por qué pertenecer, pero si lo que estamos optimizando es un ensemble sí, porque la familia es:

$$F = \left\{ \sum_{m=1}^M \beta_m h(a_m) \right\}$$





Gradient Boosting: algoritmo

Algorithm 1: Gradient_Boost

1 $F_0(\mathbf{x}) = \arg \min_{\rho} \sum_{i=1}^N L(y_i, \rho)$

2 For $m = 1$ to M do:

3 $\tilde{y}_i = - \left[\frac{\partial L(y_i, F(\mathbf{x}_i))}{\partial F(\mathbf{x}_i)} \right]_{F(\mathbf{x})=F_{m-1}(\mathbf{x})}, i = 1, N$

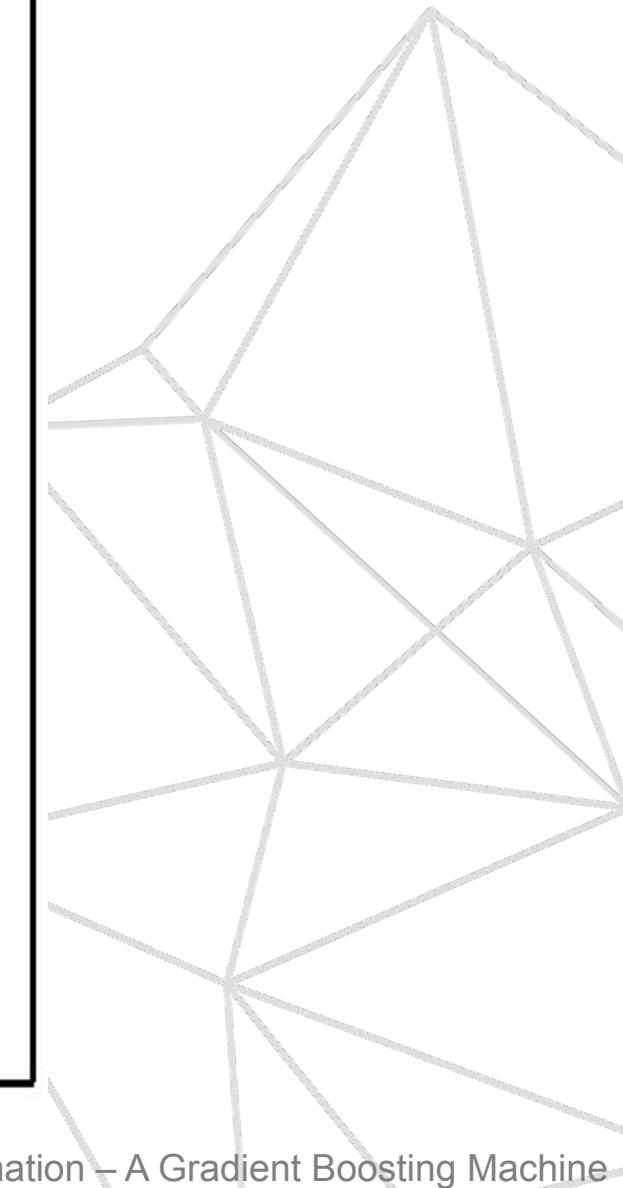
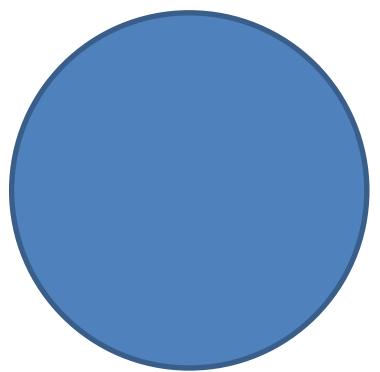
4 $\mathbf{a}_m = \arg \min_{\mathbf{a}, \beta} \sum_{i=1}^N [\tilde{y}_i - \beta h(\mathbf{x}_i; \mathbf{a})]^2$

5 $\rho_m = \arg \min_{\rho} \sum_{i=1}^N L(y_i, F_{m-1}(\mathbf{x}_i) + \rho h(\mathbf{x}_i; \mathbf{a}_m))$

6 $F_m(\mathbf{x}) = F_{m-1}(\mathbf{x}) + \rho_m h(\mathbf{x}; \mathbf{a}_m)$

7 endFor

end Algorithm





Gradient Boosting: algoritmo (error cuadrático)

Algorithm 2: LS_Boost

$$F_0(\mathbf{x}) = \bar{y}$$

For $m = 1$ to M do:

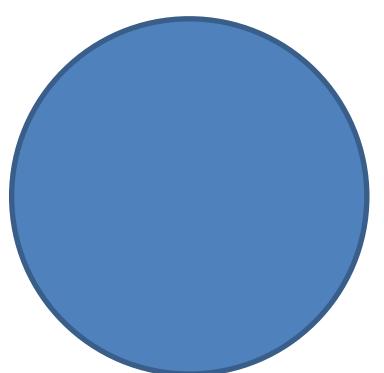
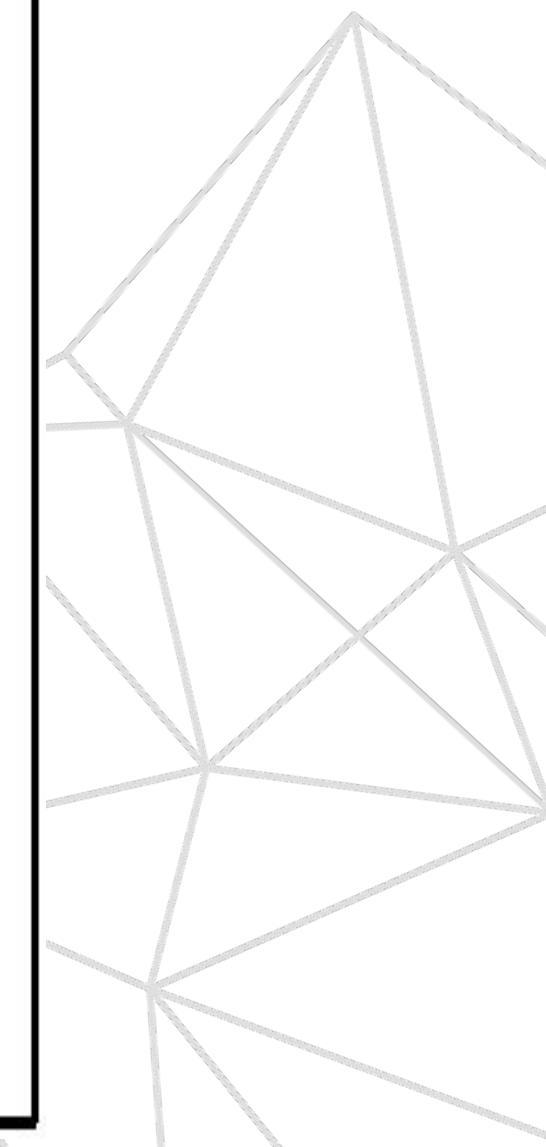
$$\tilde{y}_i = y_i - F_{m-1}(\mathbf{x}_i), \quad i = 1, N$$

$$(\rho_m, \mathbf{a}_m) = \arg \min_{\mathbf{a}, \rho} \sum_{i=1}^N [\tilde{y}_i - \rho h(\mathbf{x}_i; \mathbf{a})]^2$$

$$F_m(\mathbf{x}) = F_{m-1}(\mathbf{x}) + \rho_m h(\mathbf{x}; \mathbf{a}_m)$$

endFor

end Algorithm

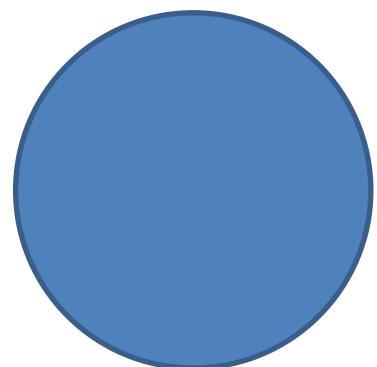




Librerías de Gradient Tree Boosting

dmlc
XGBoost

- Optimizada para volúmenes medios de datos
- Regulariza penalizando árboles con muchas hojas y predicciones extremas.
- No acepta variables categóricas.



- Las 3 permiten entrenar en paralelo en CPU y GPU, y permiten usar clústers.
- Para datasets tabulares suelen ser la mejor opción (velocidad, precisión, interpretabilidad...).

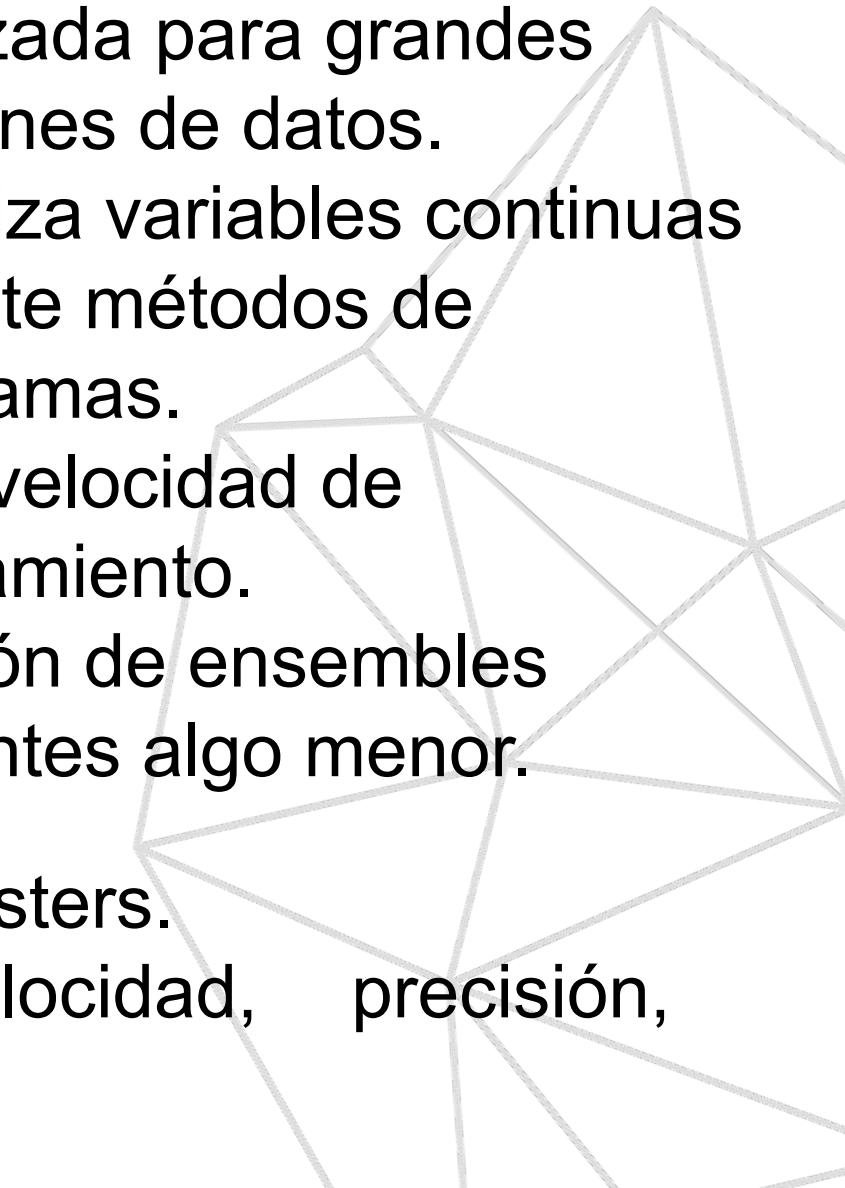


**Yandex
CatBoost**

- Optimizada para grandes volúmenes de datos.
- Optimizada para variables categóricas.
- Mayor velocidad de predicción.
- Entrenamiento en general más lento.

**Microsoft
LightGBM**

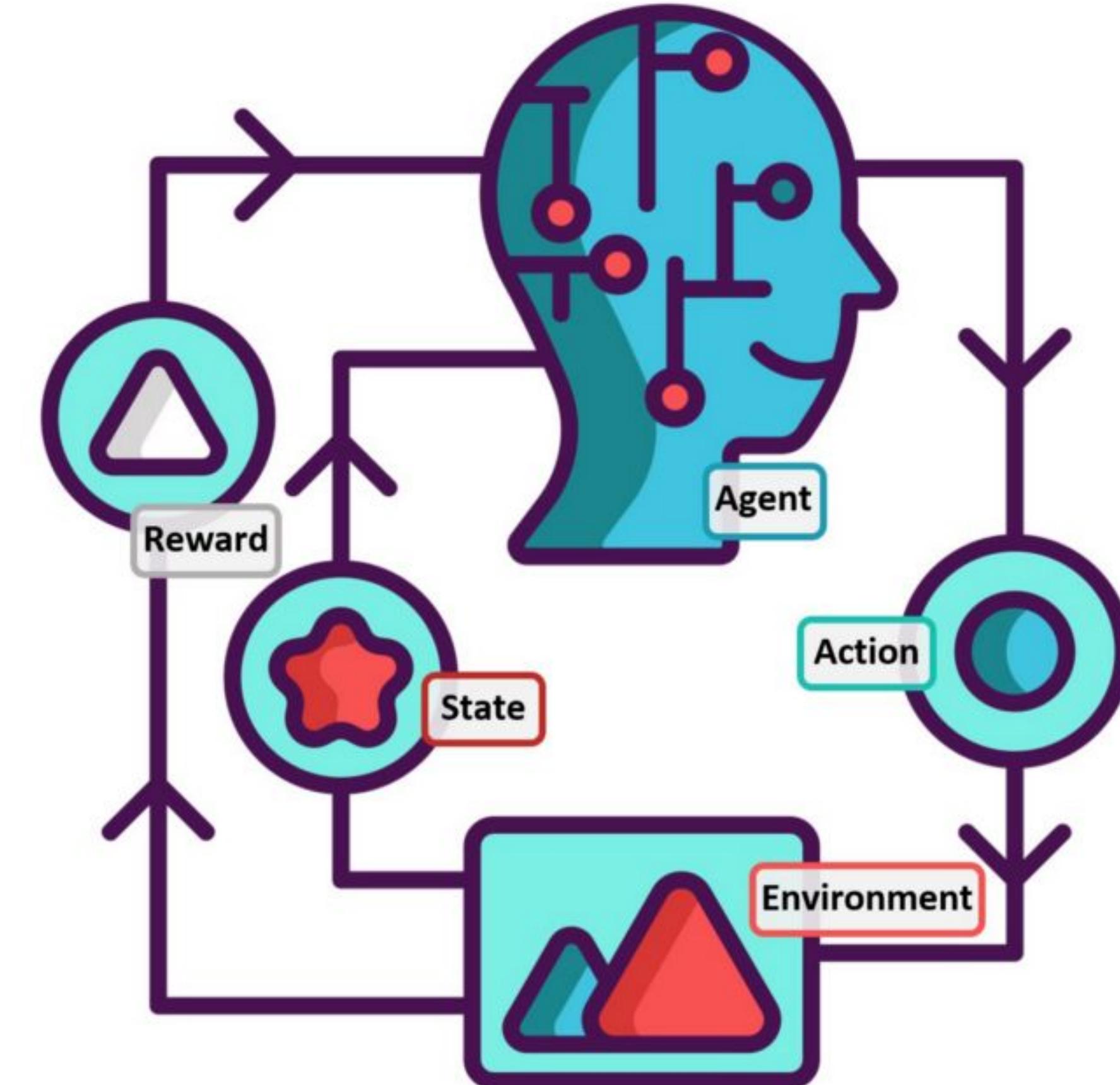
- Optimizada para grandes volúmenes de datos.
- Discretiza variables continuas mediante métodos de histogramas.
- Mayor velocidad de entrenamiento.
- Precisión de ensambles resultantes algo menor.



06

Introducción al aprendizaje por refuerzo

Conceptos básicos sobre el aprendizaje por refuerzo y los procesos de decisión de Markov



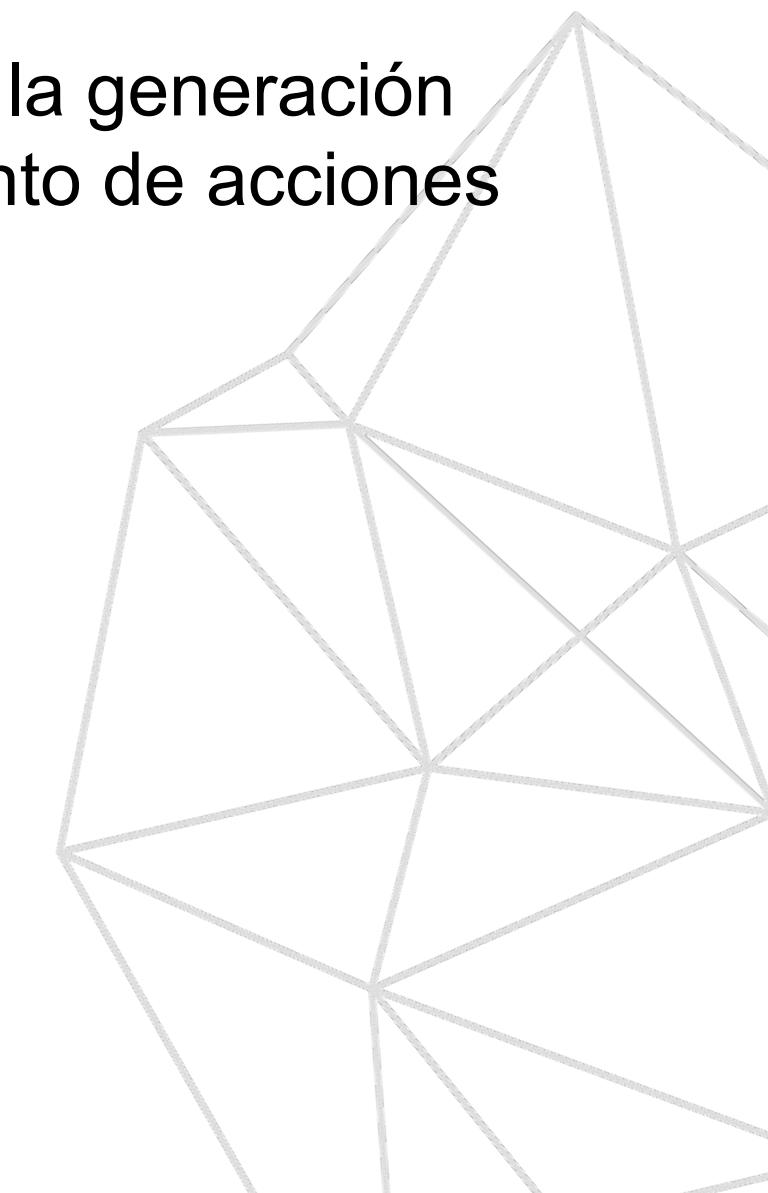
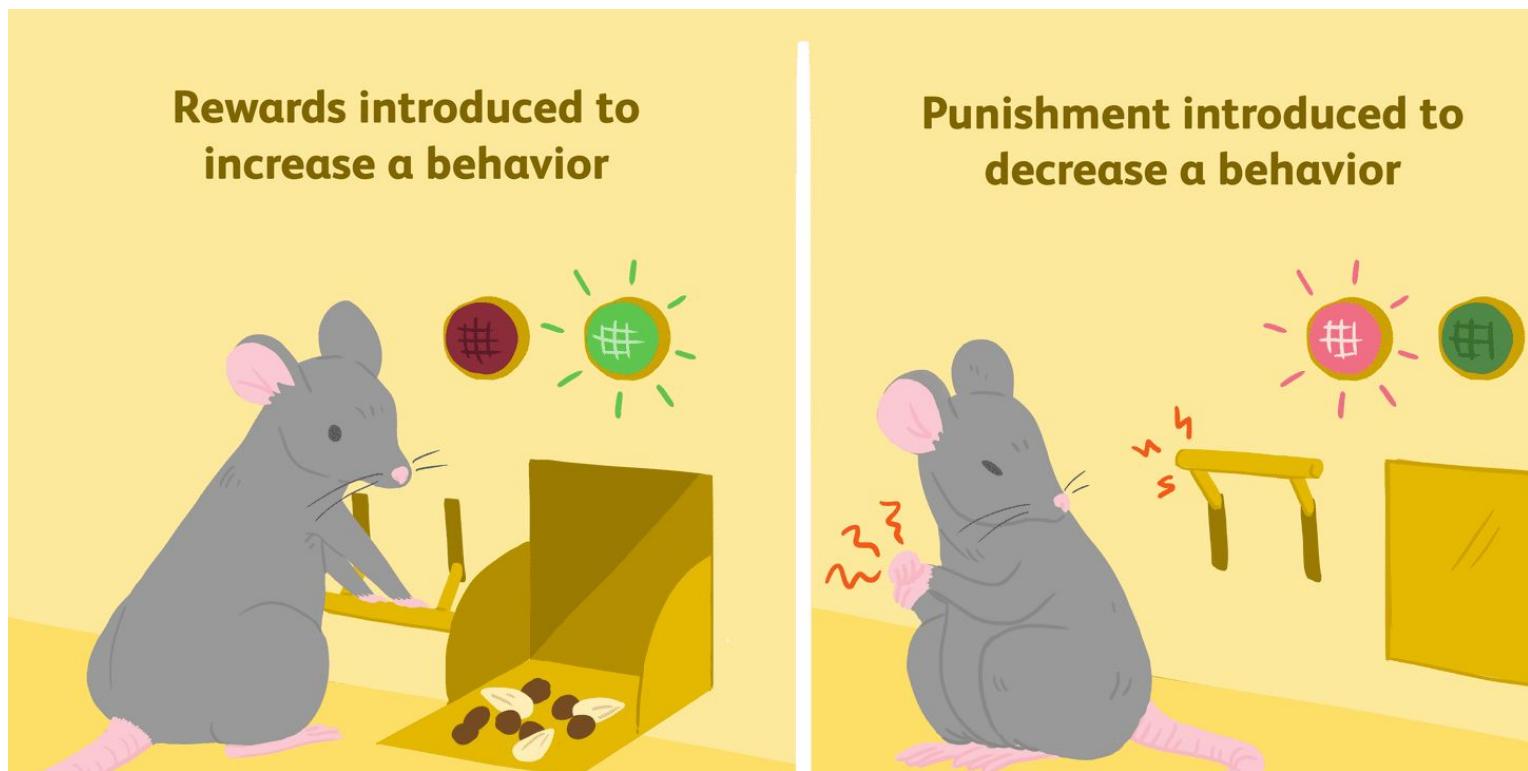


Orígenes

Siguiendo la **psicología conductivista** el aprendizaje por refuerzo (también conocido como aprendizaje reforzado o RL por su nombre en inglés *Reinforcement Learning*) engloba un tipo de aprendizaje que se basa en las recompensas:

- Obtendrás una recompensa positiva cuando se lleva a cabo una acción adecuada.
- Obtendrás una recompensa negativa (o castigo) cuando se lleva a cabo una acción inapropiada.

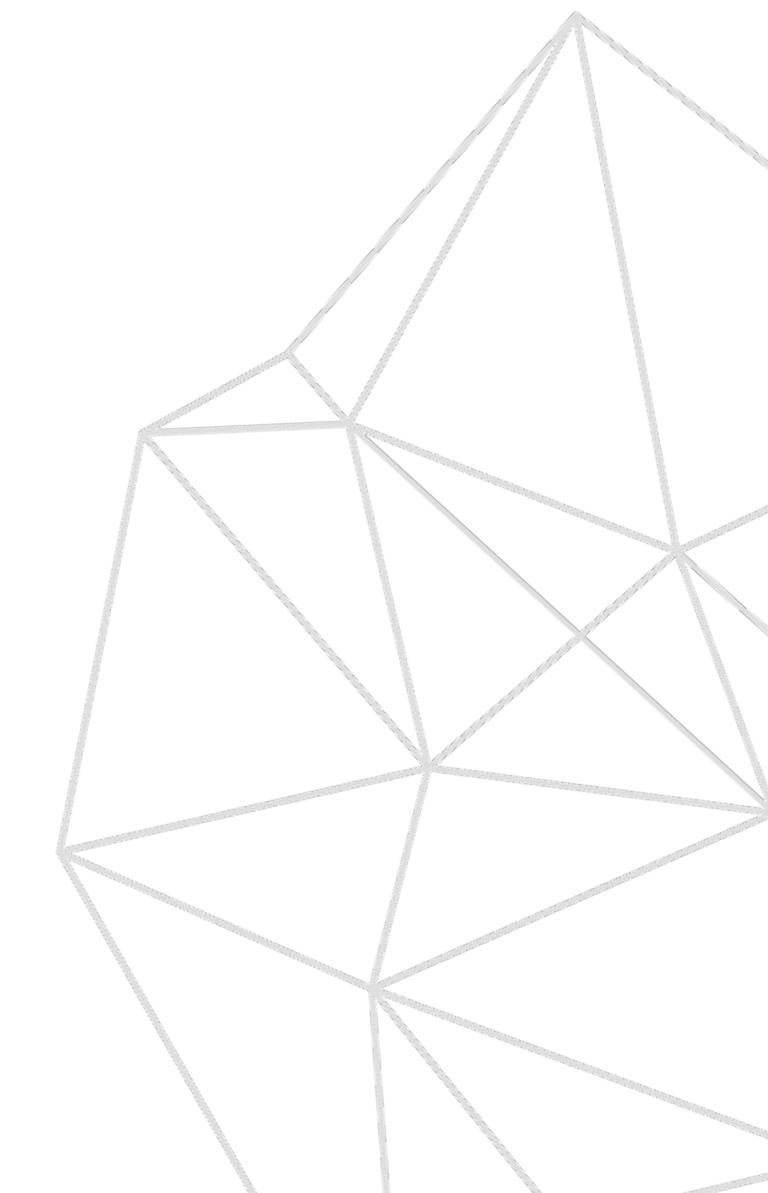
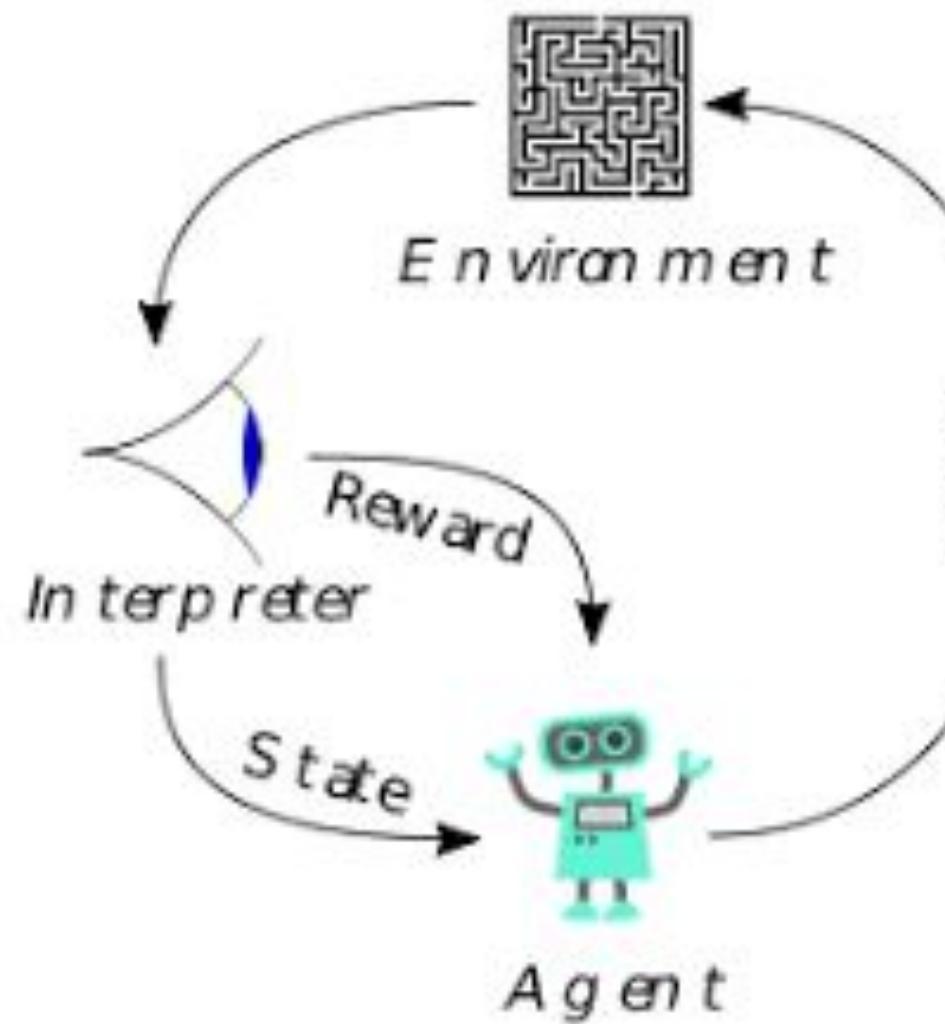
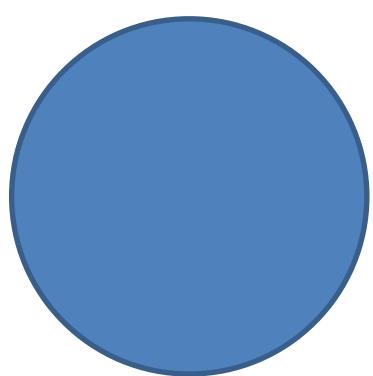
Este tipo de aprendizaje no necesitará conocimiento previo, sino una manera de dirigir la generación del conocimiento a través de recompensas. El objetivo será, por tanto, buscar el conjunto de acciones en relación al entorno que maximice la recompensa a corto y largo plazo.



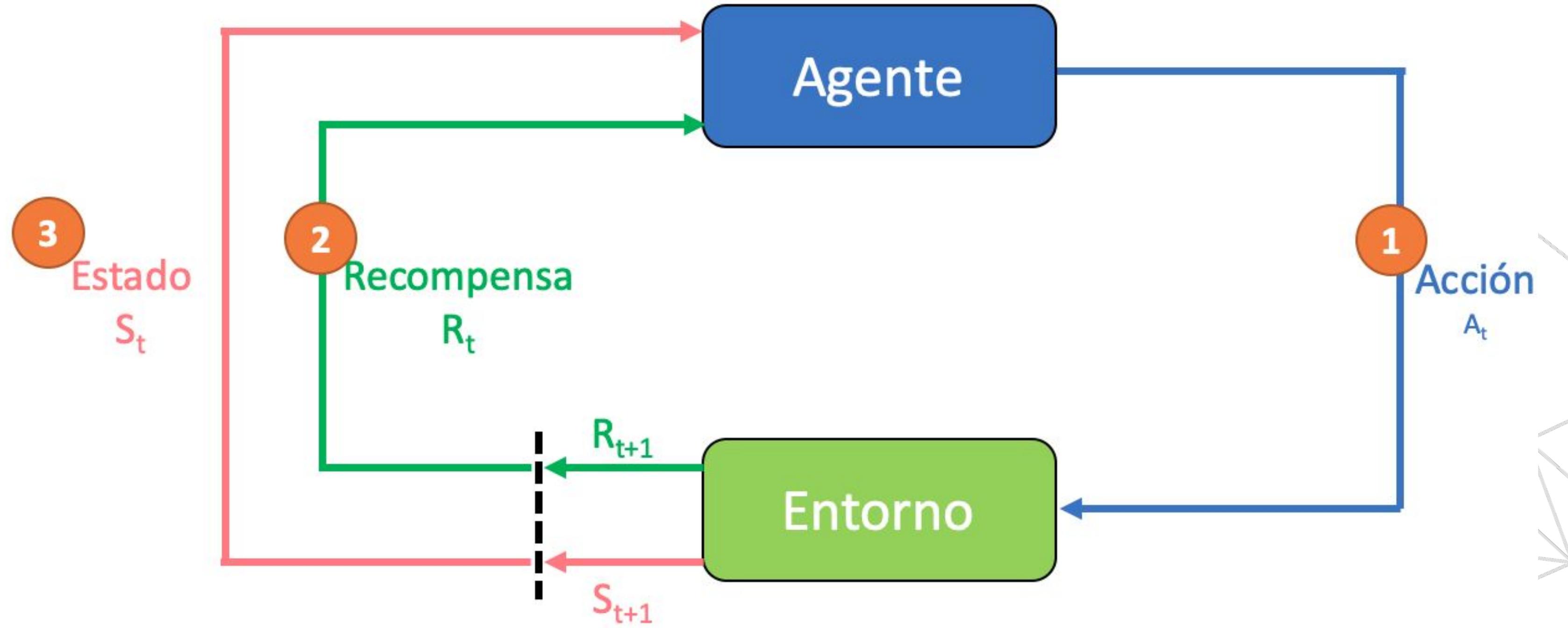
Definiciones básicas

El aprendizaje por refuerzo está basado en un **agente** que "vive" en un **entorno** y es capaz de percibir un **estado** de ese entorno. El agente puede ejecutar **acciones** en cada estado y estas acciones conllevan diferentes **recompensas**.

El objetivo es aprender una **política** encargada de tomar la decisión de qué acción tomar en cada estado con el **objetivo de maximizar la recompensa**, bien sea a corto o largo plazo.

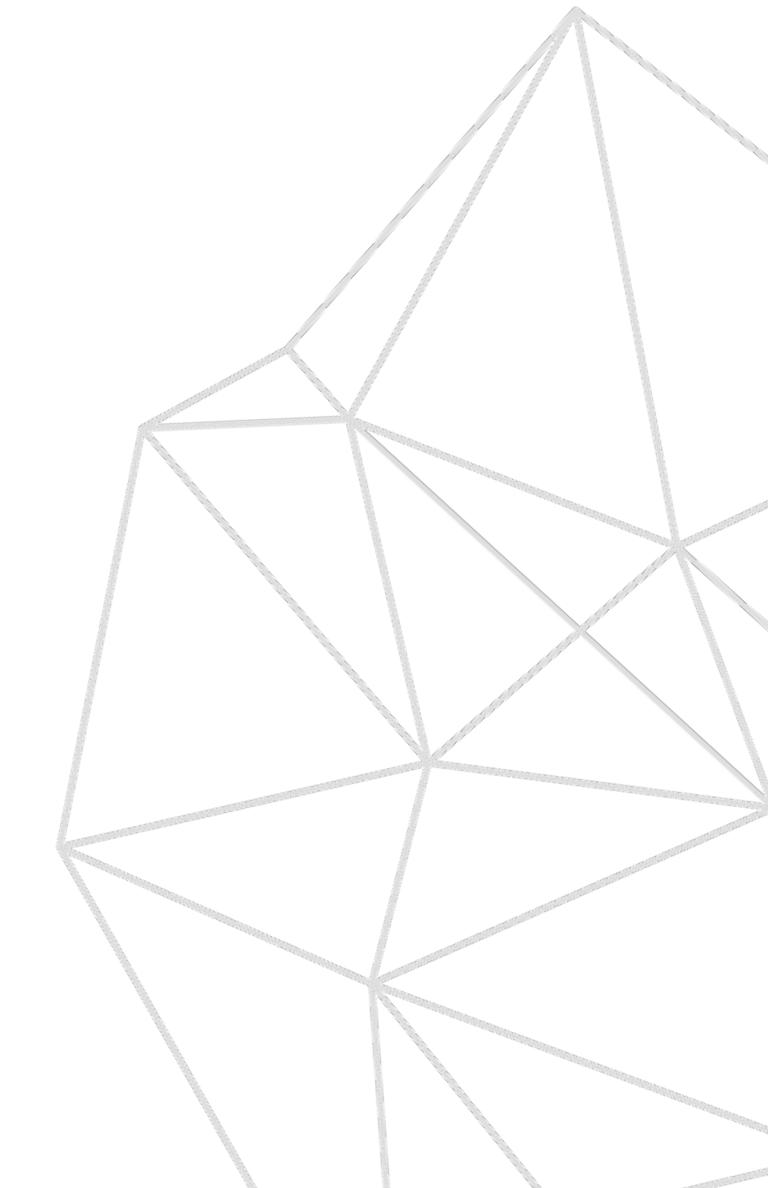
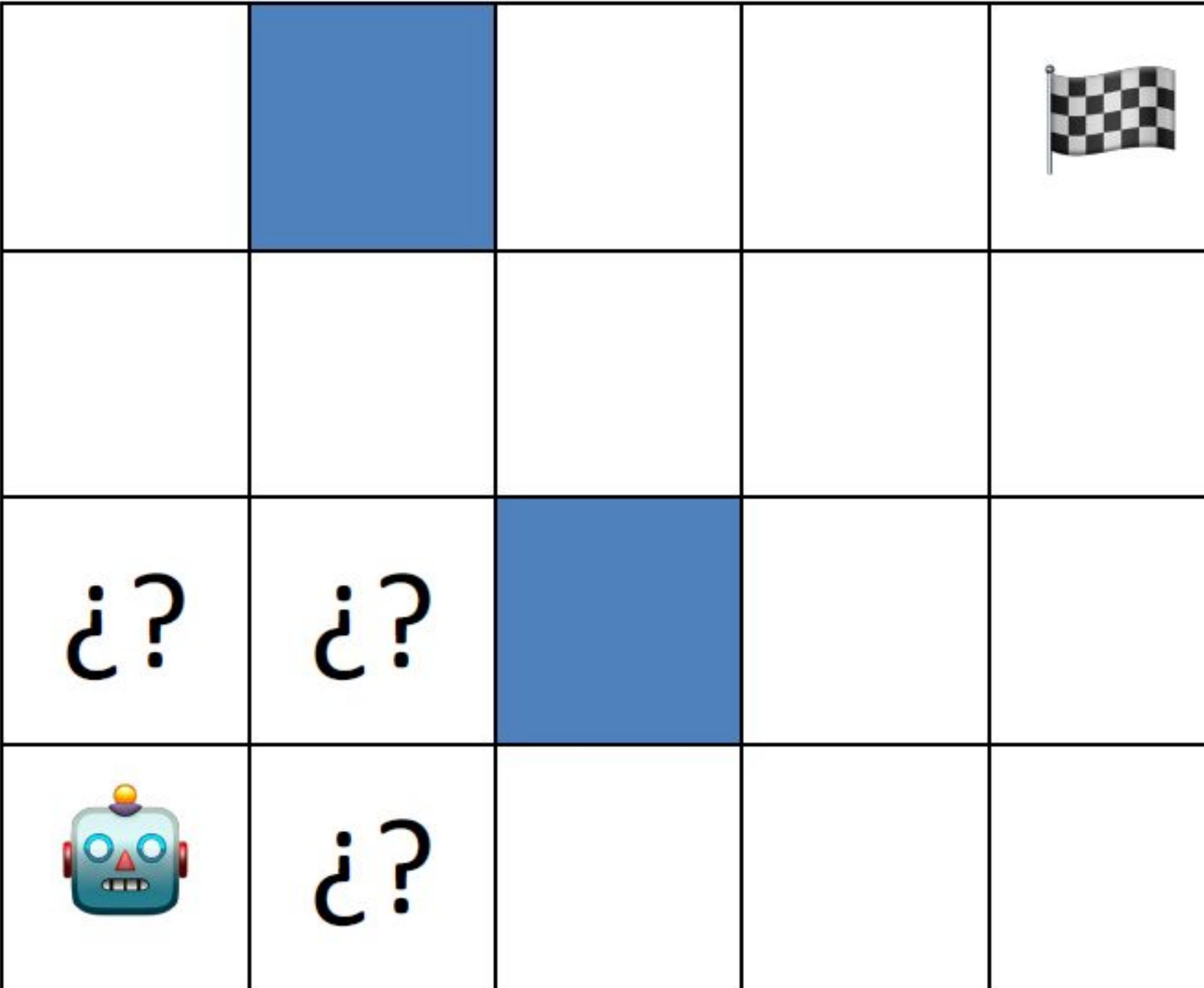
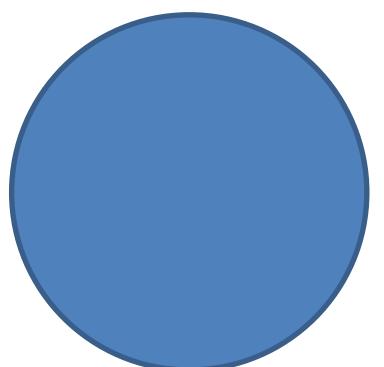


Esquema general



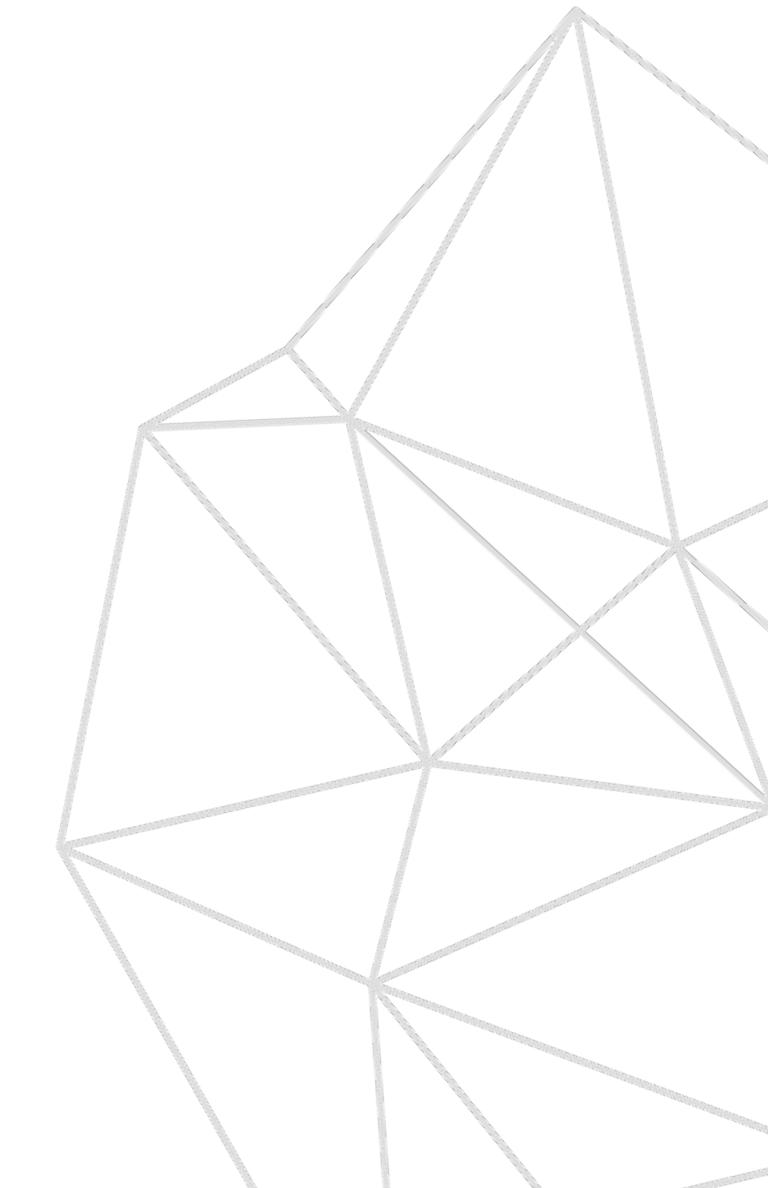
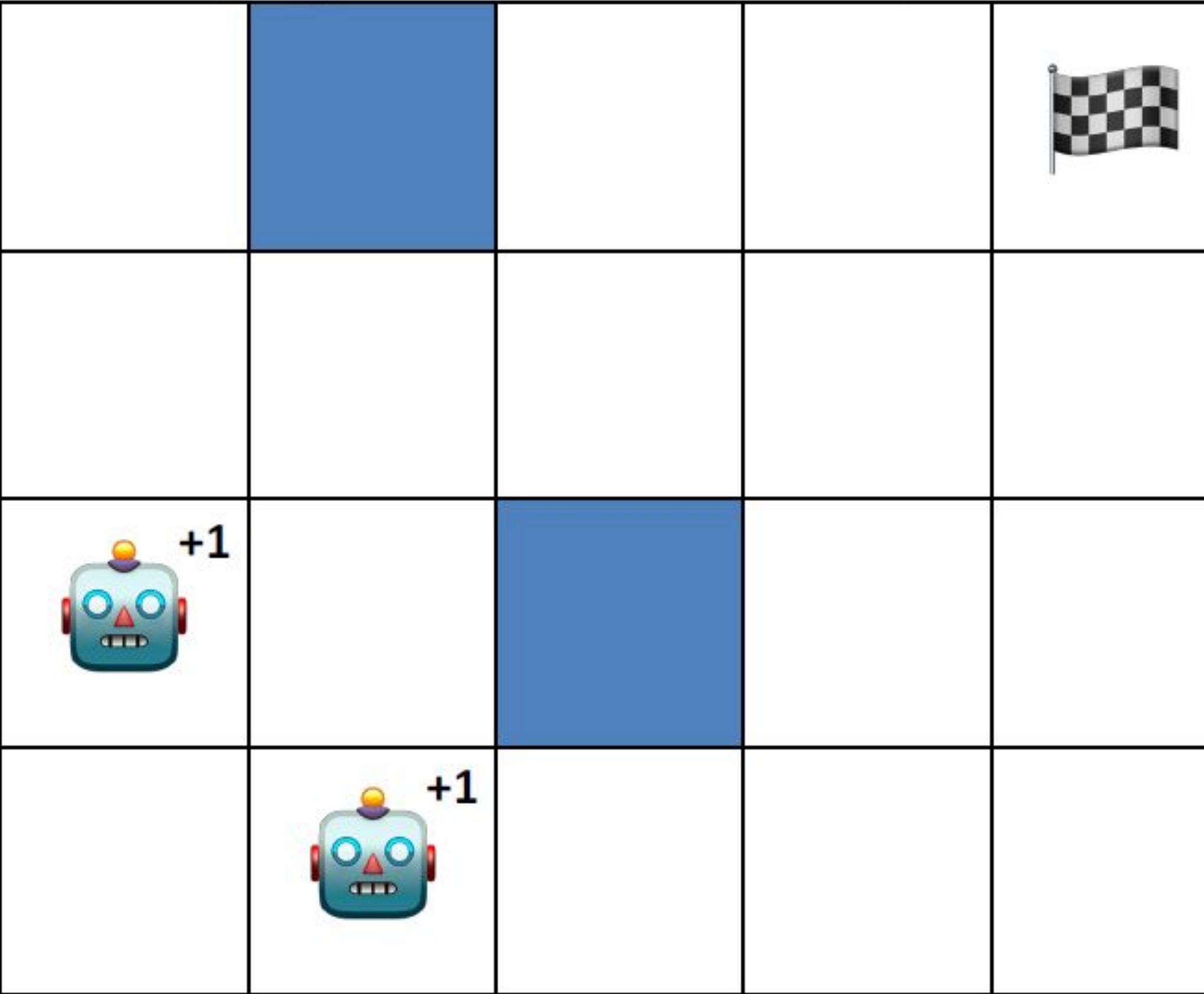
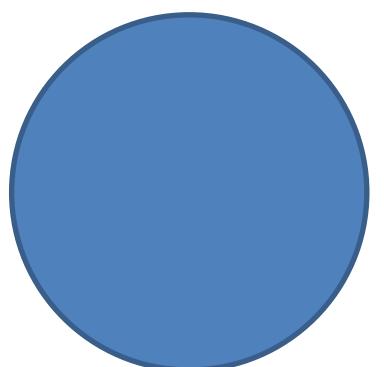


Ejemplo: ¿qué dirección tomar para llegar a la meta?



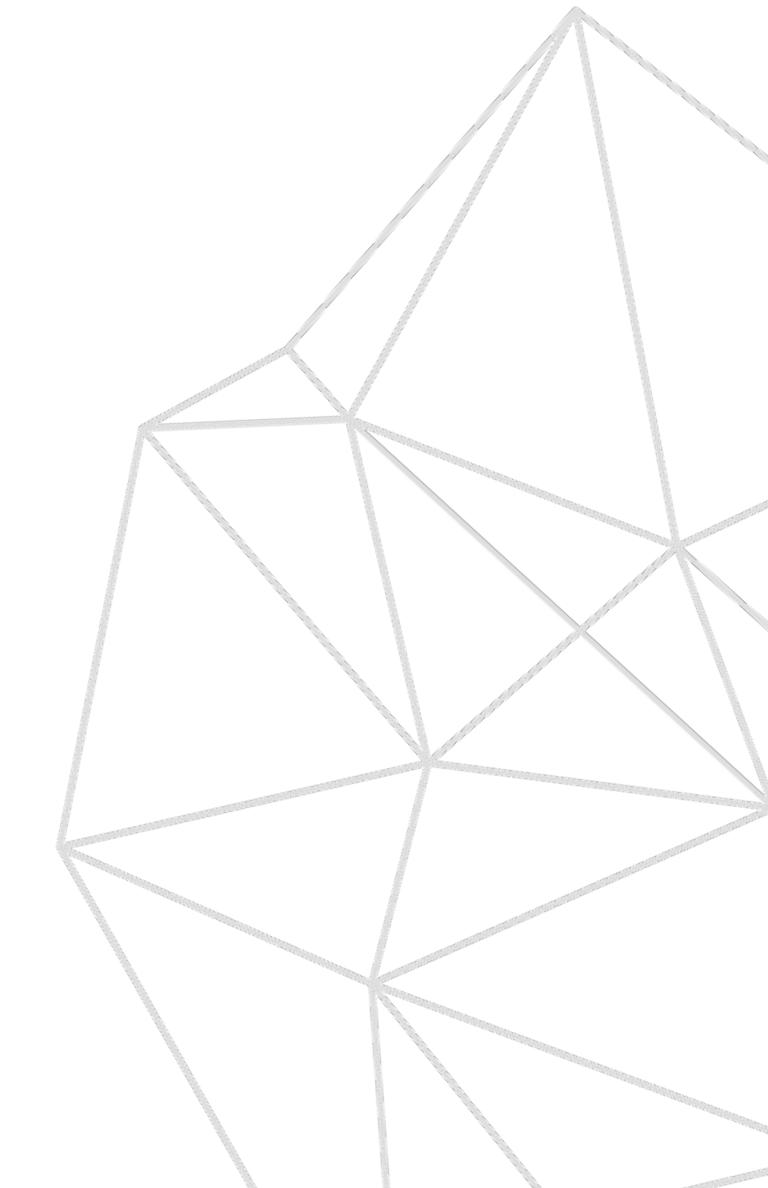
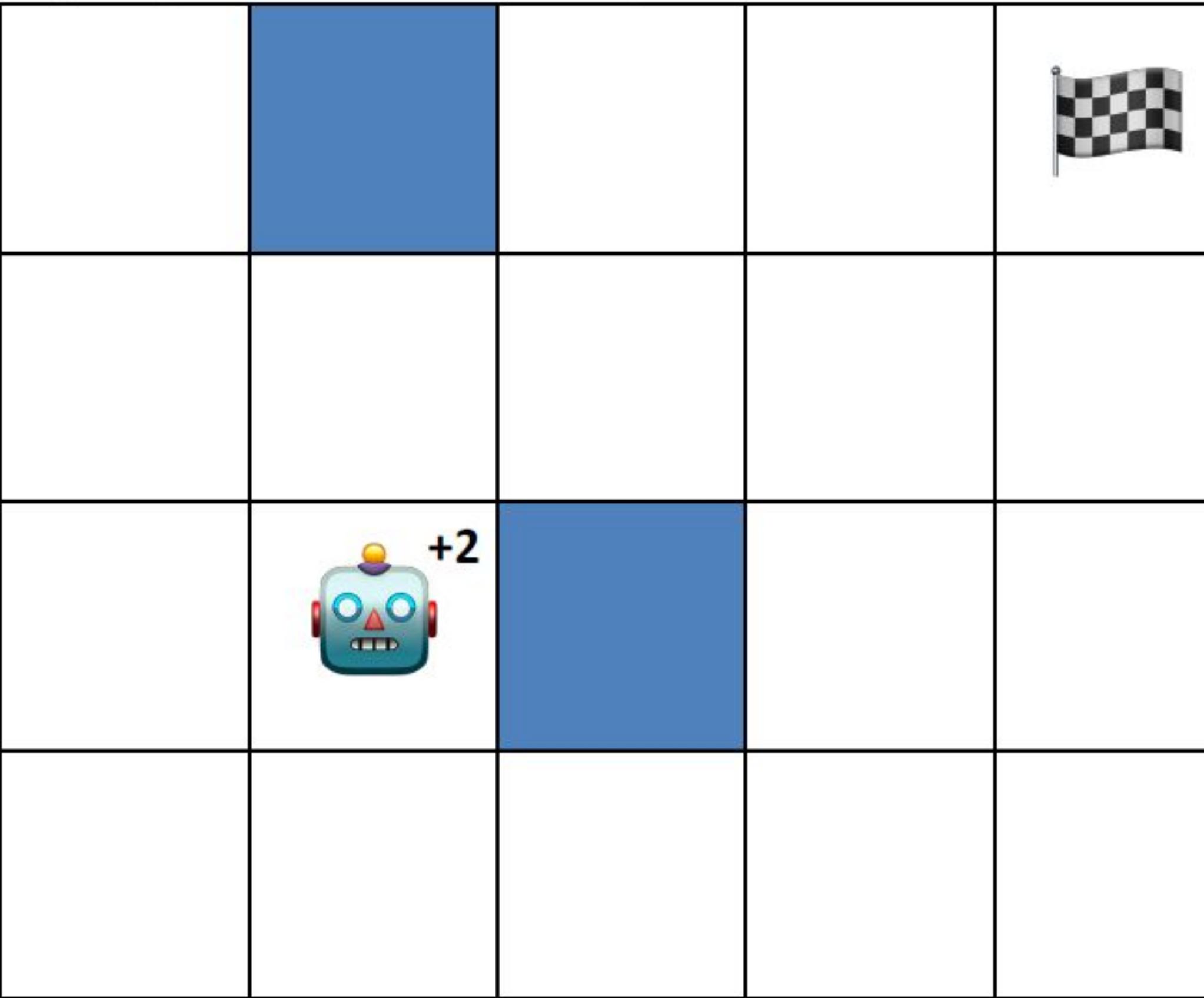


Ejemplo: ¿qué dirección tomar para llegar a la meta?

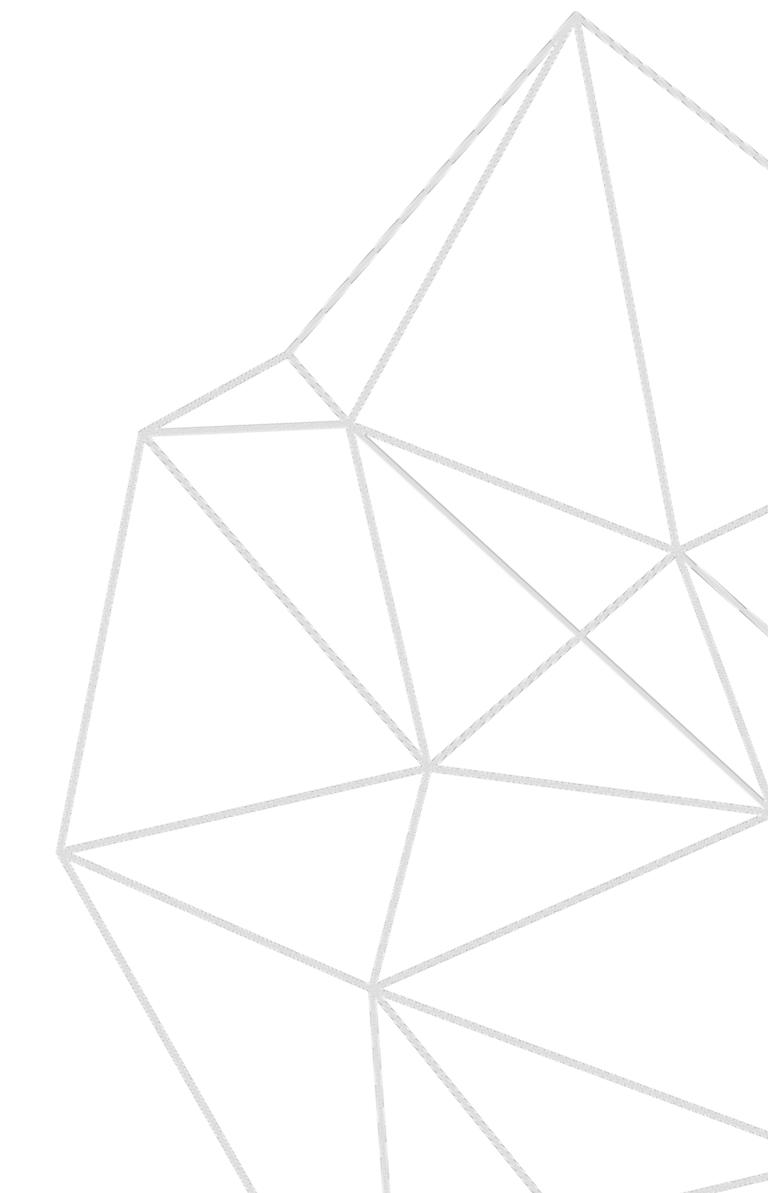
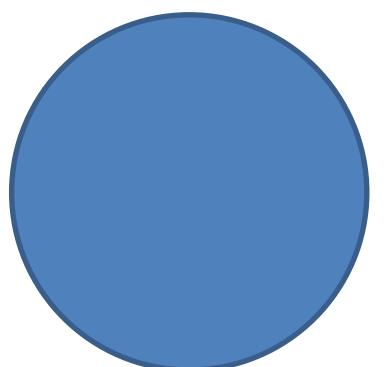
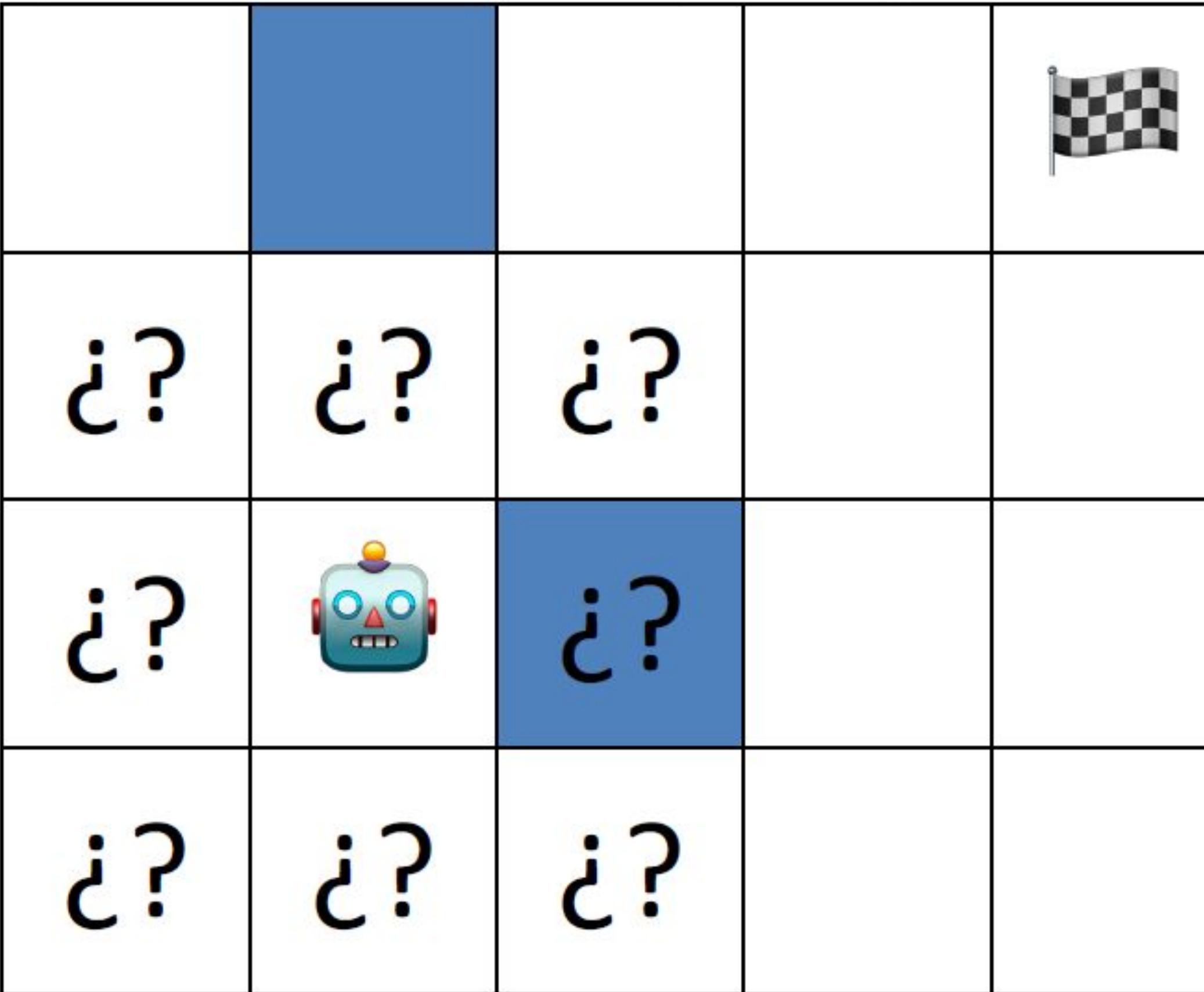




Ejemplo: ¿qué dirección tomar para llegar a la meta?

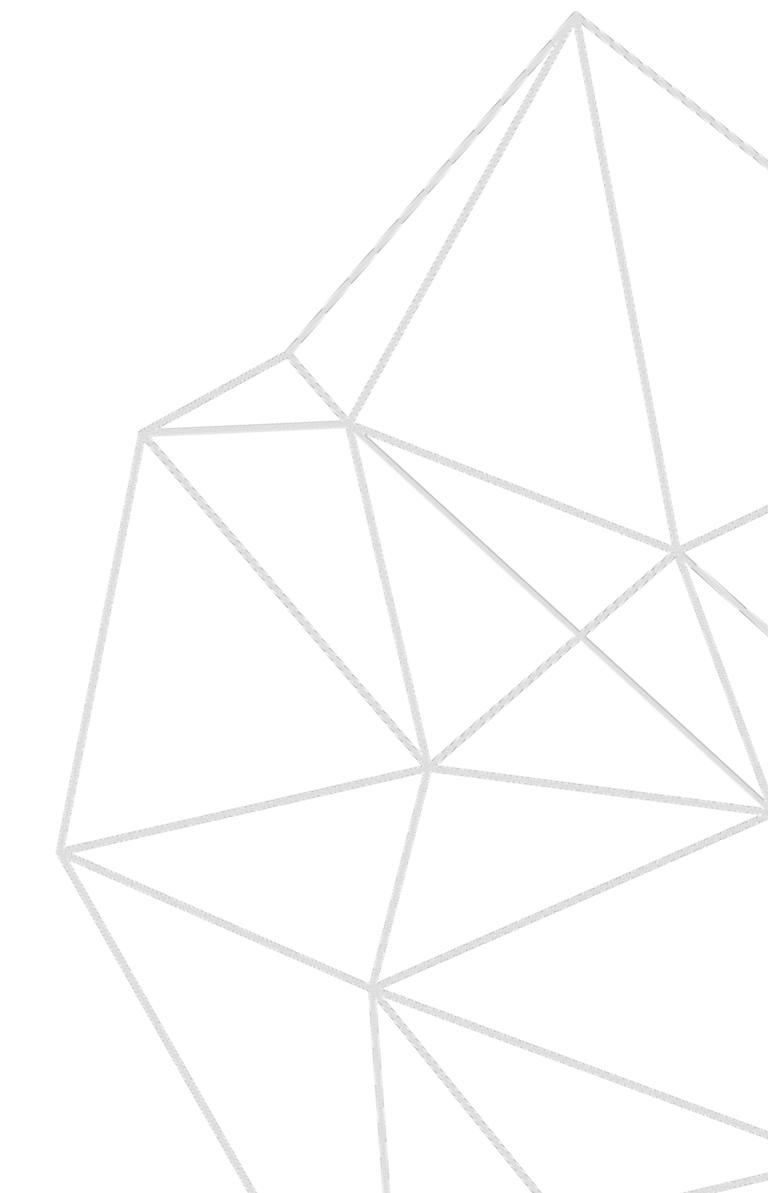
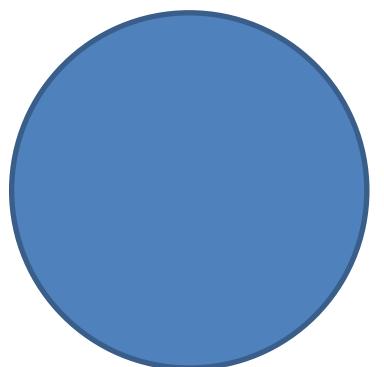


Ejemplo: ¿qué dirección tomar para llegar a la meta?



Ejemplo: ¿qué dirección tomar para llegar a la meta?

0	+1	+2			
0		0			
0	0	0			



Ejemplo: procesos de decisión de Markov (MDP)

En este entorno el robot (agente) conoce:

- Un conjunto de estados S en los que puede estar.
- Un conjunto de acciones A que pueden realizarse.

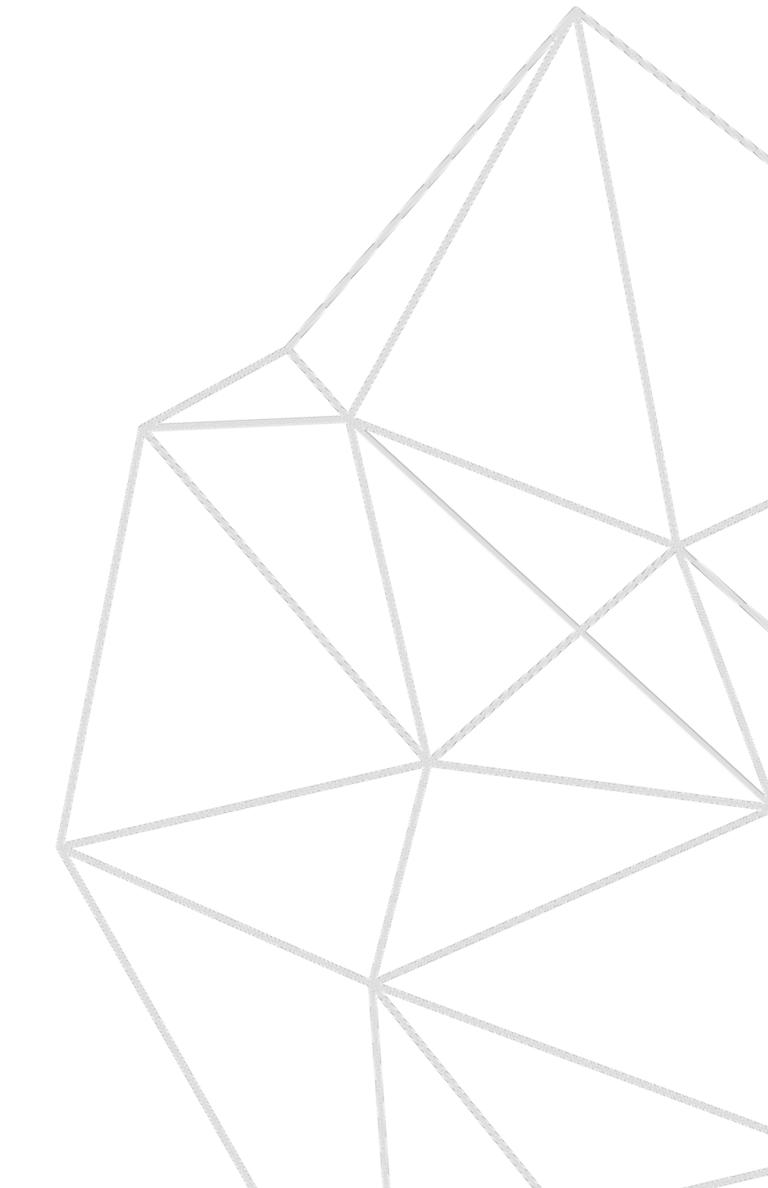
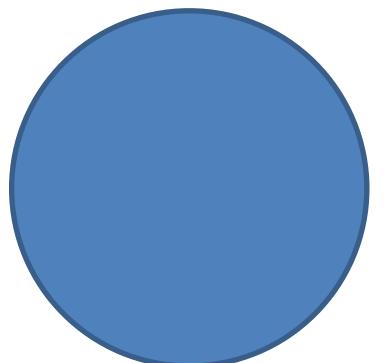
Además, para un momento dado t :

- El robot se encontrará en un estado s_t
- Y tomará la decisión de ejecutar la acción a_t

El entorno responderá:

- Dando una recompensa $r(s_t, a_t)$
- Producido una transición de estado $s_{t+1} = \delta(s_t, a_t)$

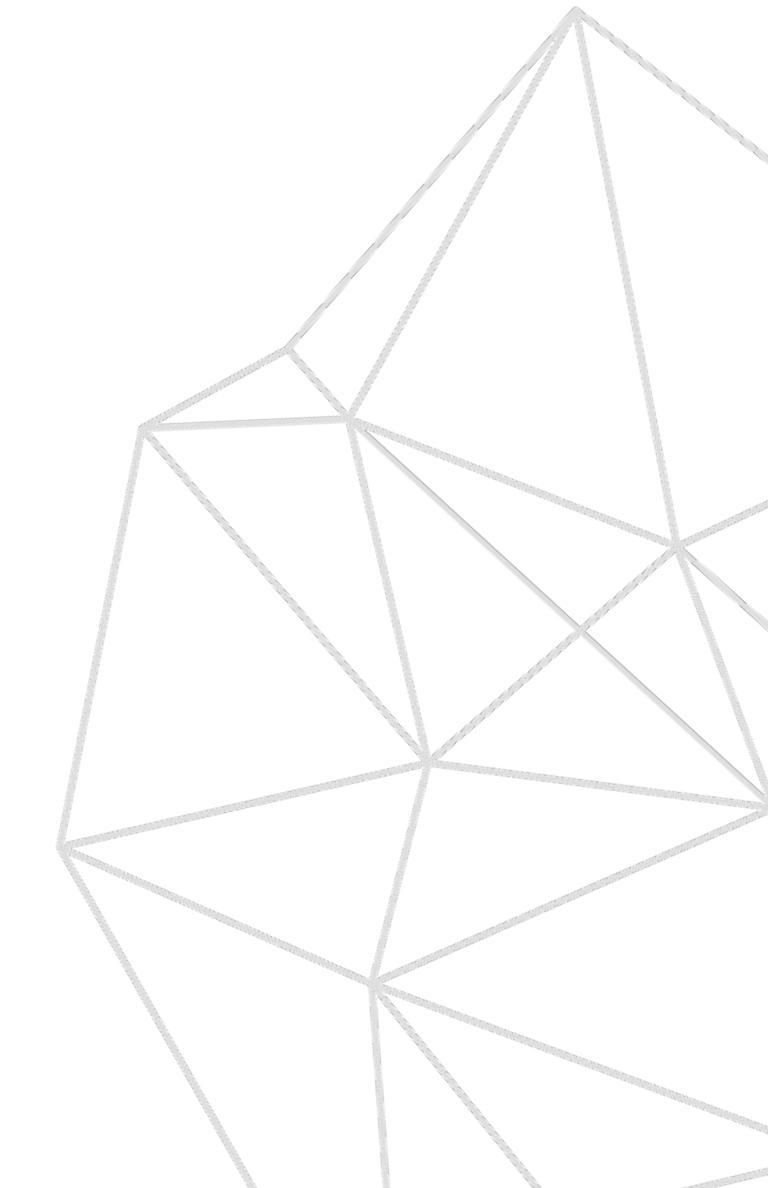
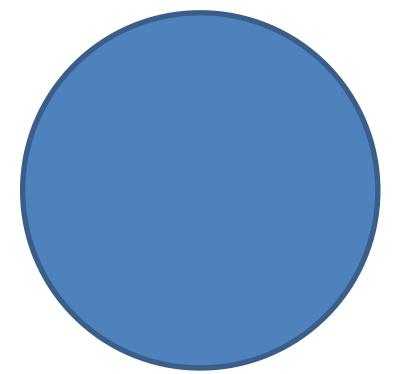
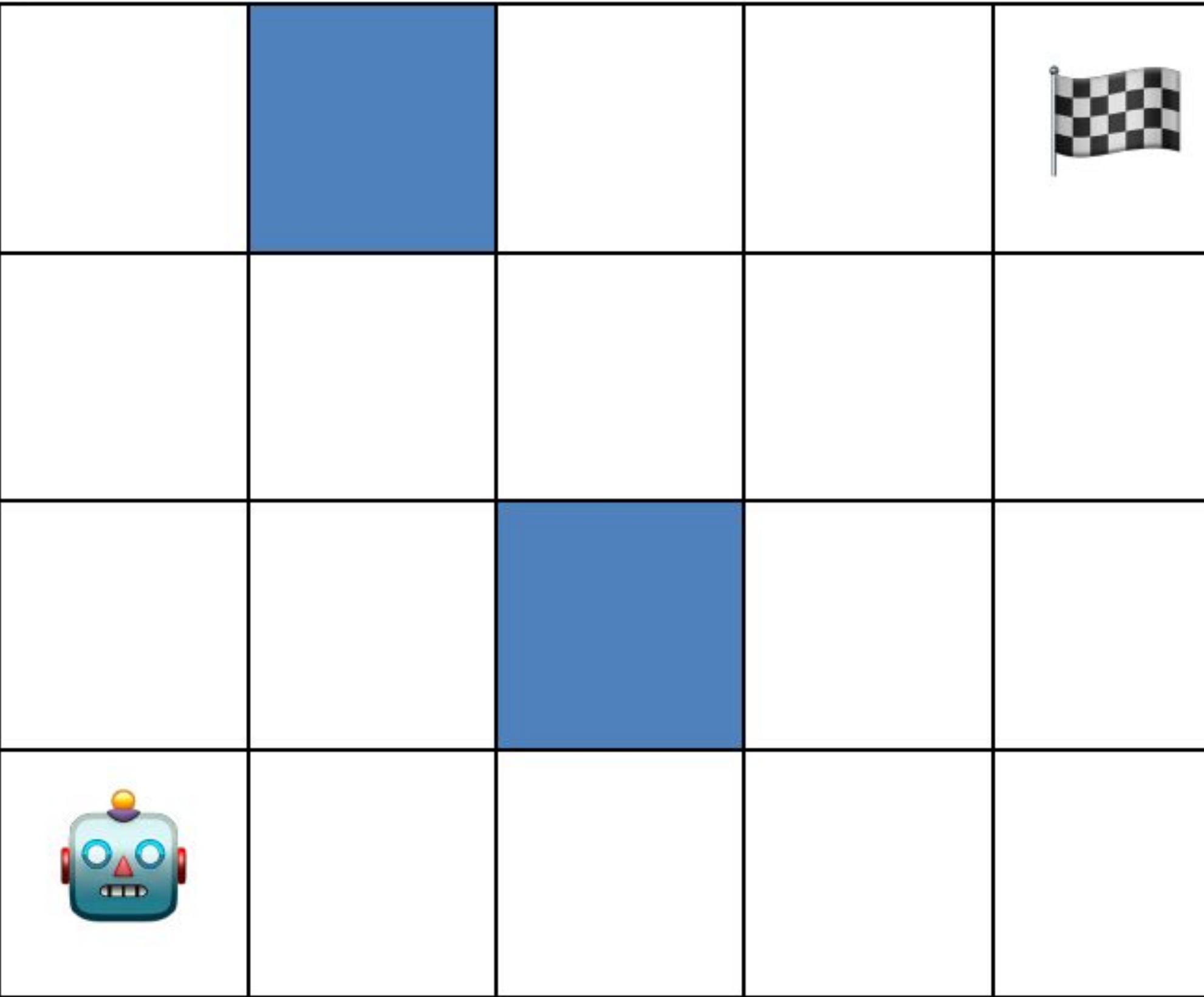
Las funciones r y δ no son necesariamente conocidas por el agente





Ejemplo: procesos de decisión de Markov (MDP)

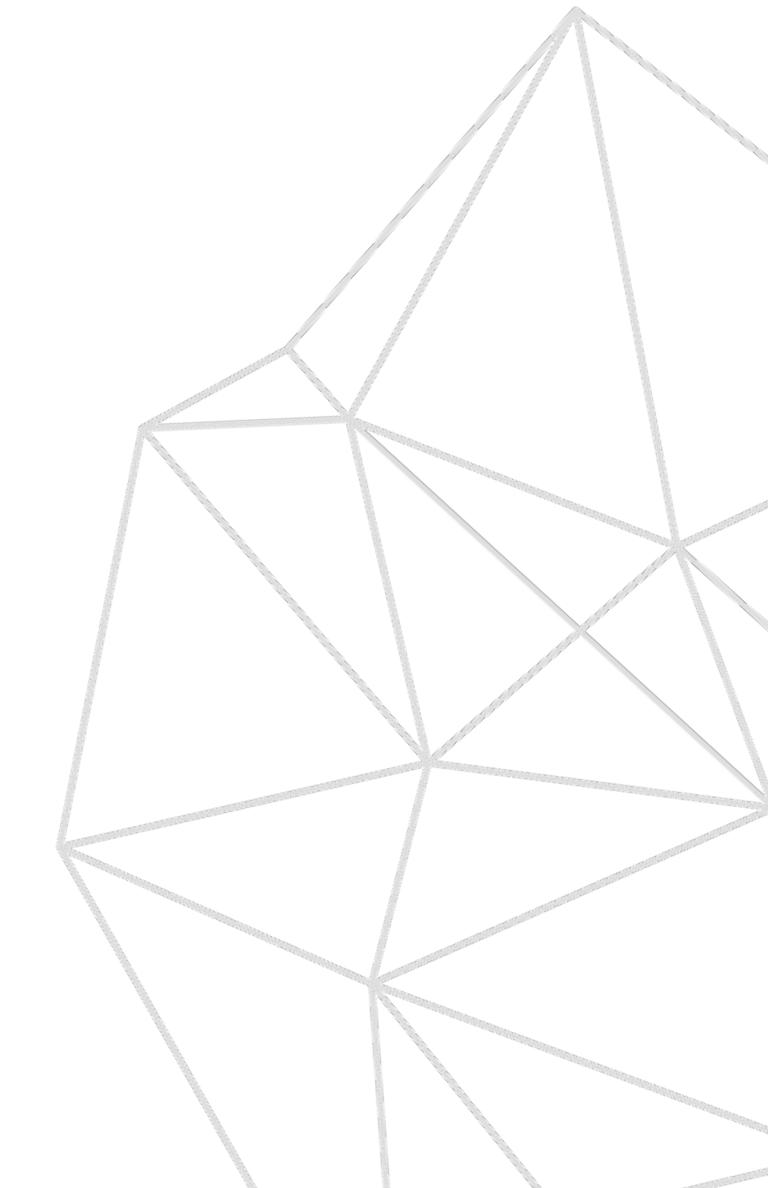
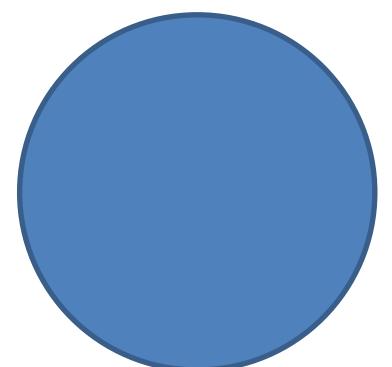
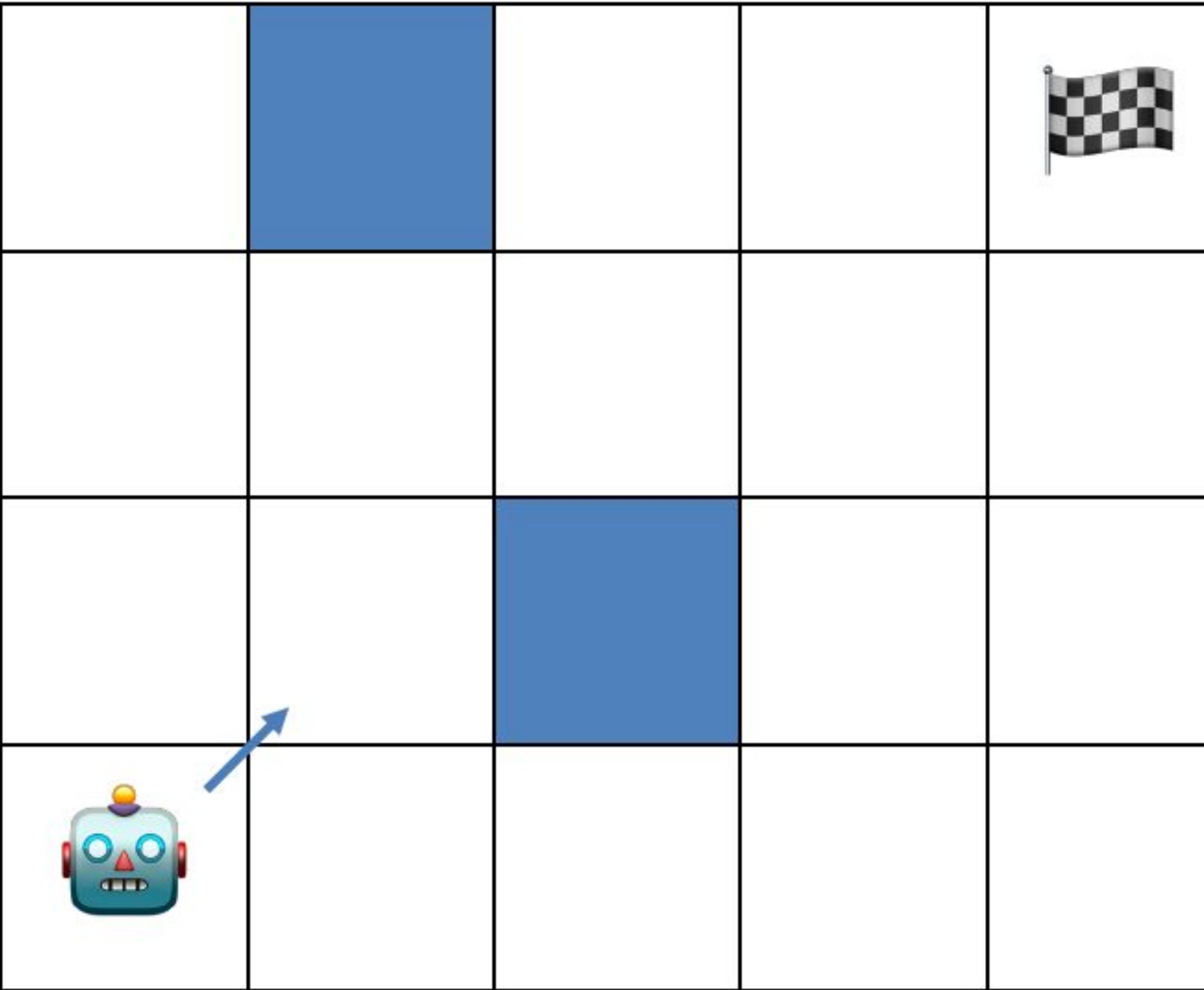
Estado s_0 :





Ejemplo: procesos de decisión de Markov (MDP)

Acción a_0 :

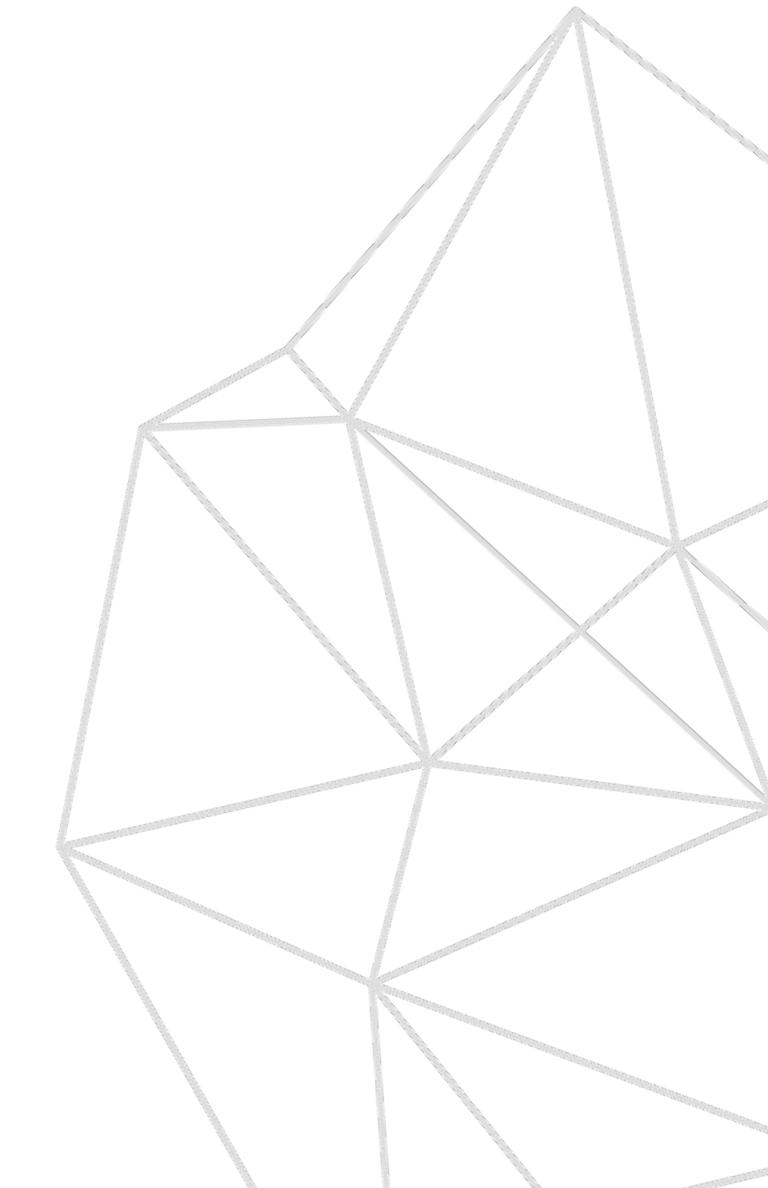
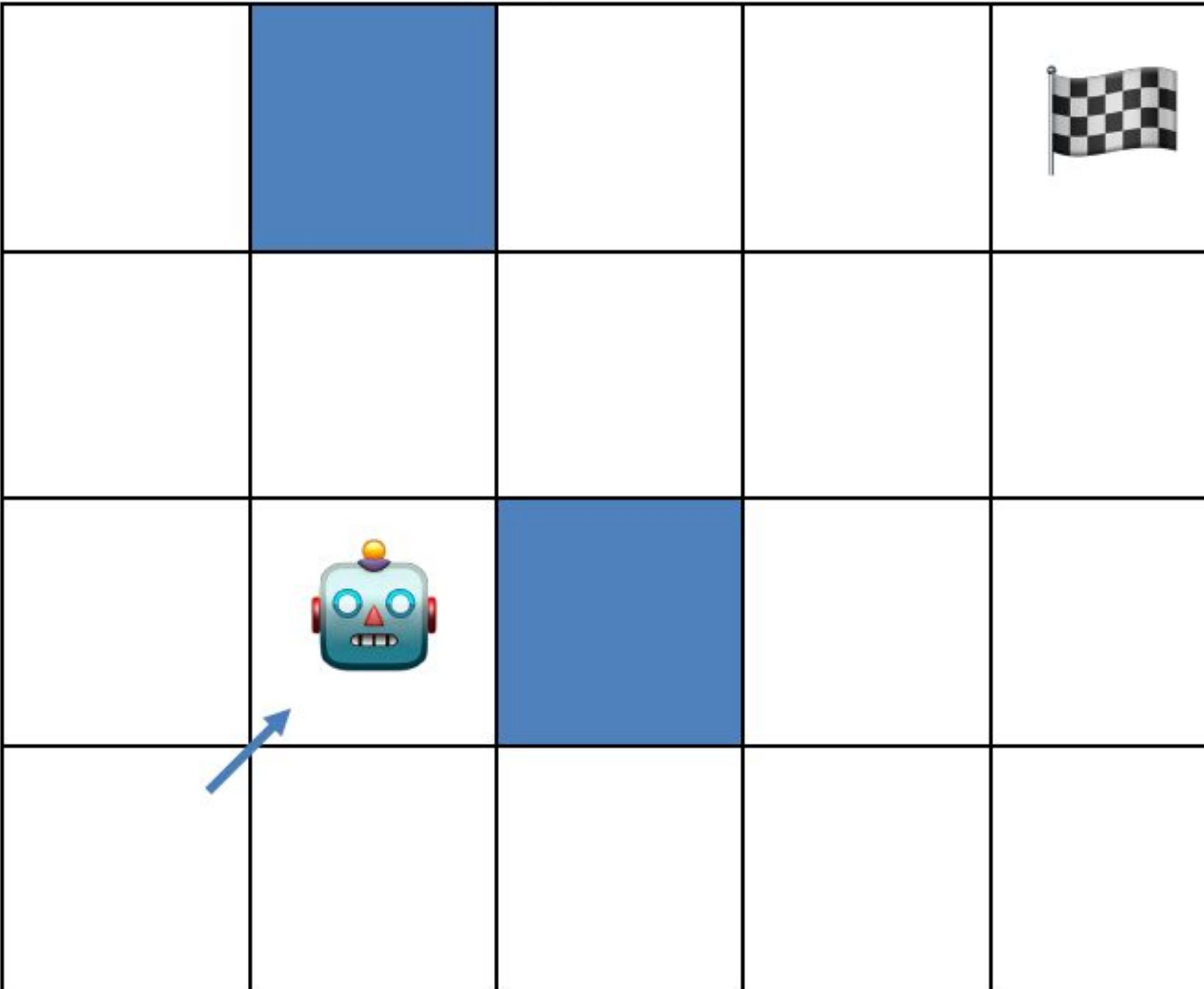
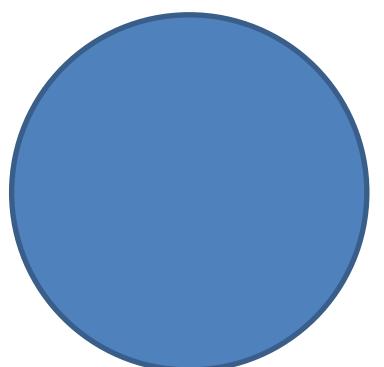




Ejemplo: procesos de decisión de Markov (MDP)

Nuevo estado:

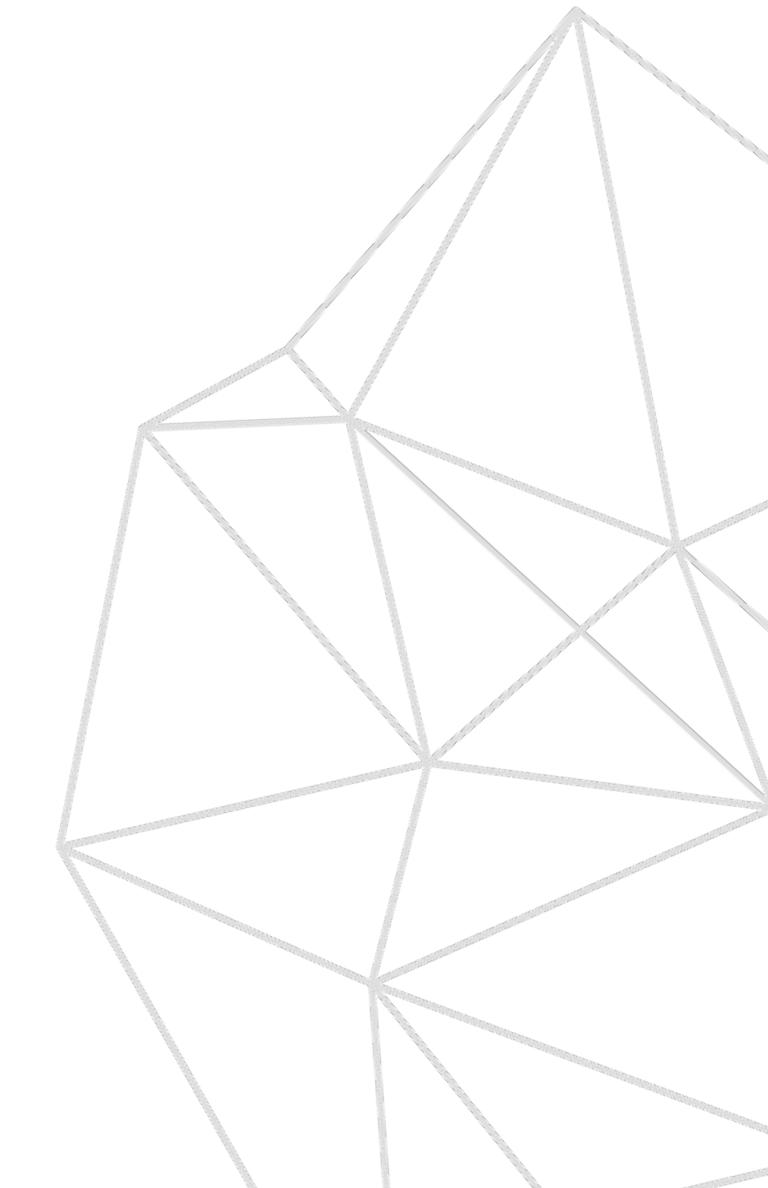
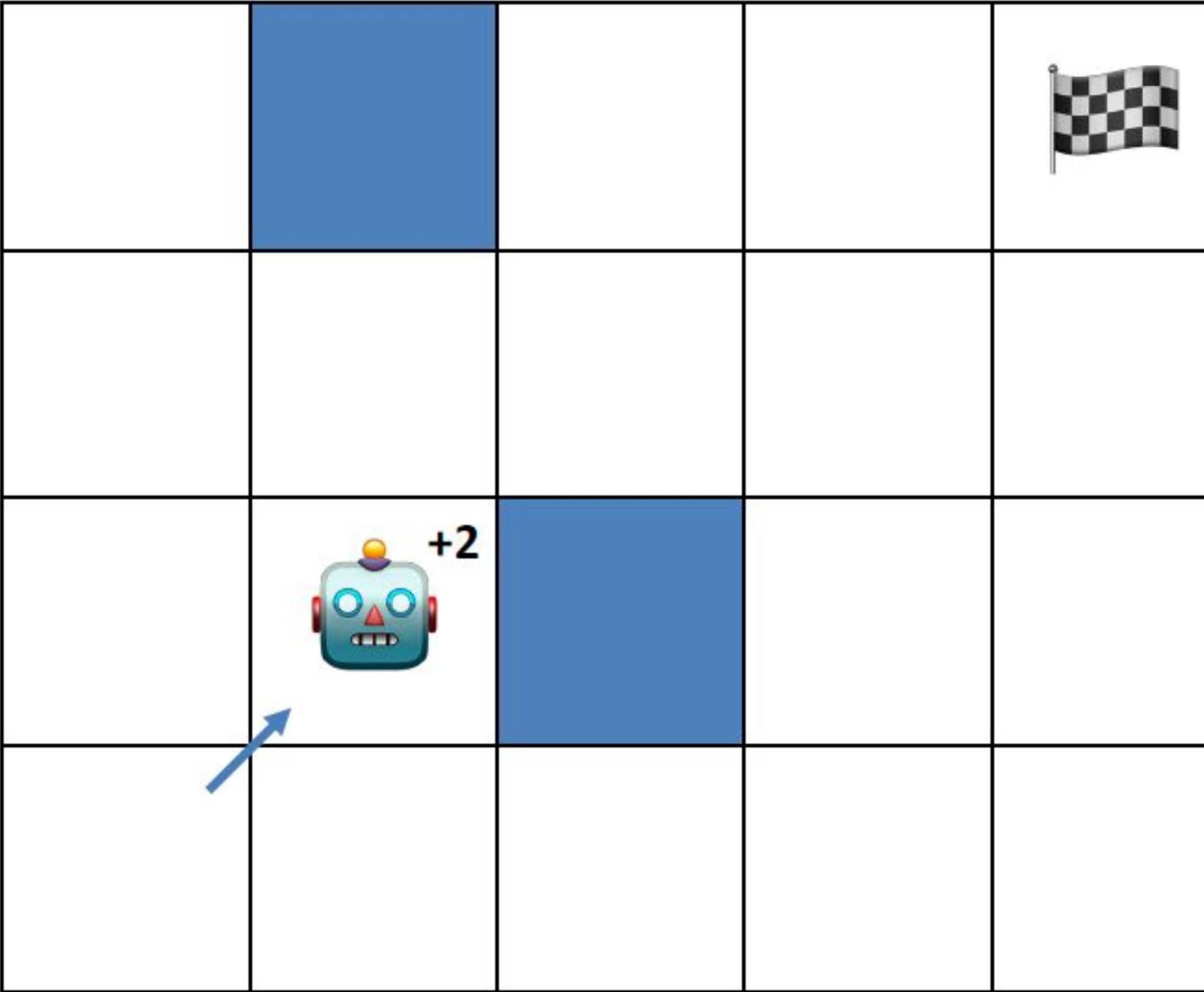
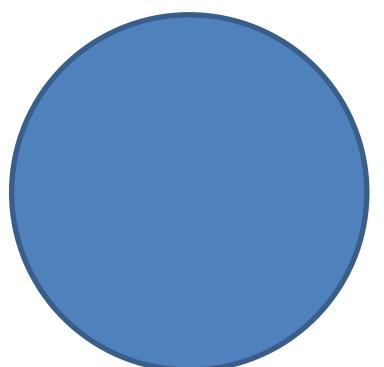
$$s_1 = \delta(s_0, a_0)$$





Ejemplo: procesos de decisión de Markov (MDP)

Recompensa:
 $r(s_0, a_0) = 2$



Procesos de decisión de Markov (MDP): Definición

Un MDP es una tupla (S, A, P, r, γ) donde:

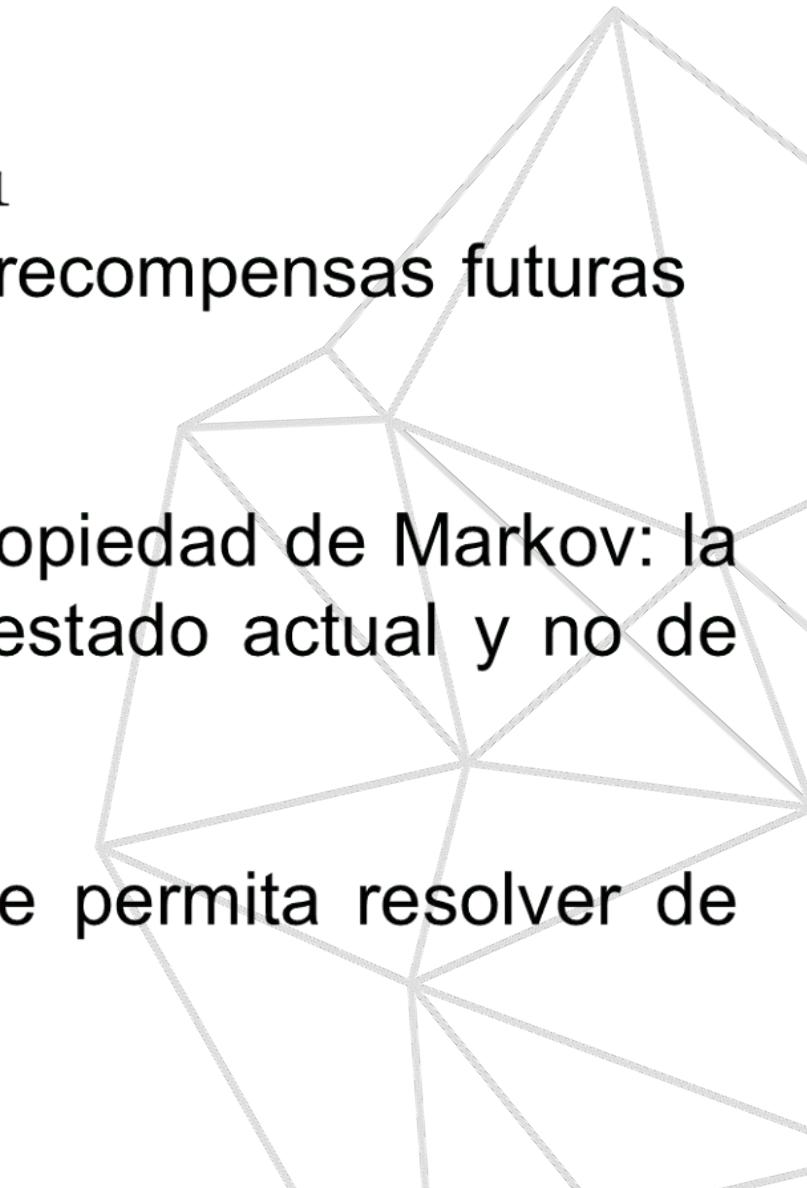
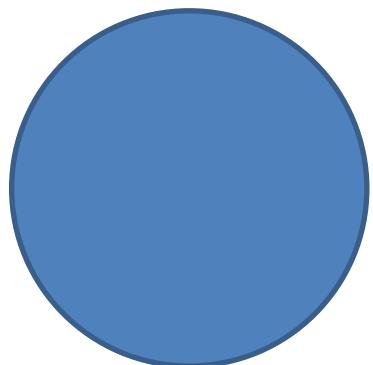
- S es el conjunto de estados posibles.
- A es el conjunto de acciones posibles
- P es la probabilidad de que se transite de un estado s_t a un estado s_{t+1} dada una acción a

$$P(s_{t+1} = s' | s_t = s, a_t = a)$$

- r es la recompensa inmediata obtenida al transitar de un estado s_t a s_{t+1}
- $\gamma \in [0,1]$ es el factor de descuento que representa la importancia de las recompensas futuras frente a las actuales

Para poder definir un proceso de decisión de Markov debe cumplirse la propiedad de Markov: la probabilidad de transitar de un estado a otro depende únicamente del estado actual y no de estados previos o futuros.

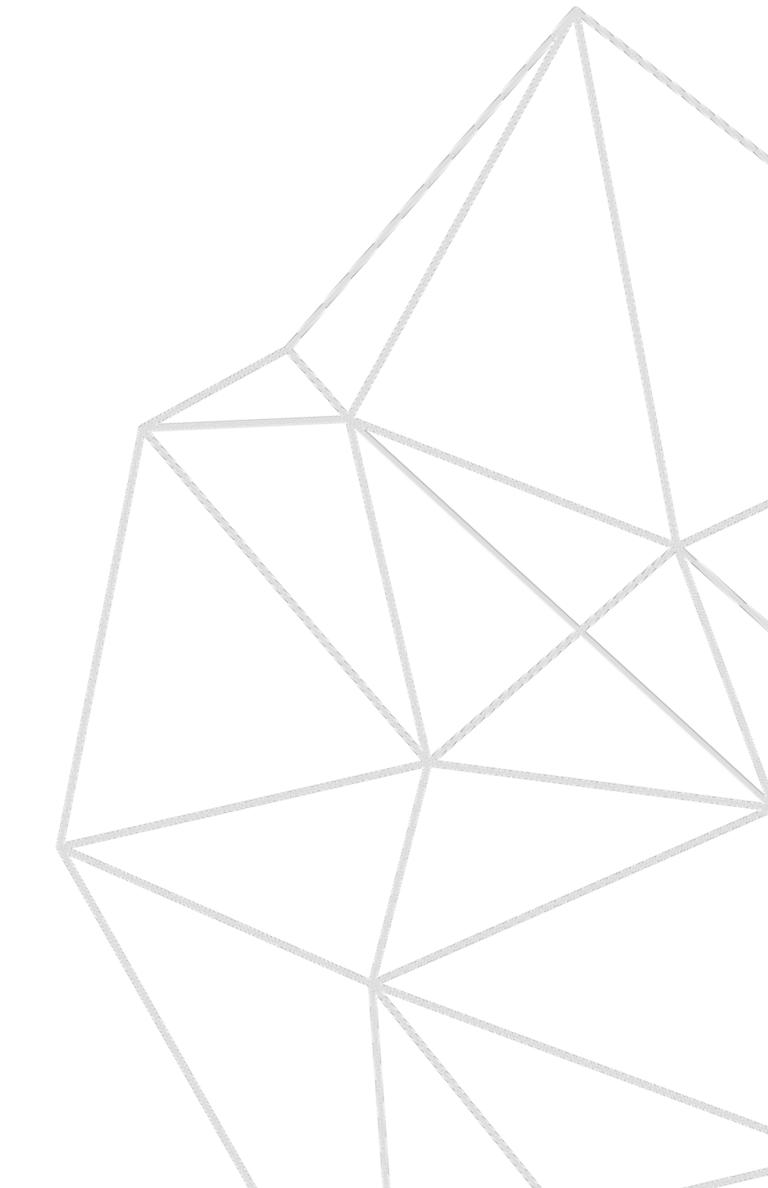
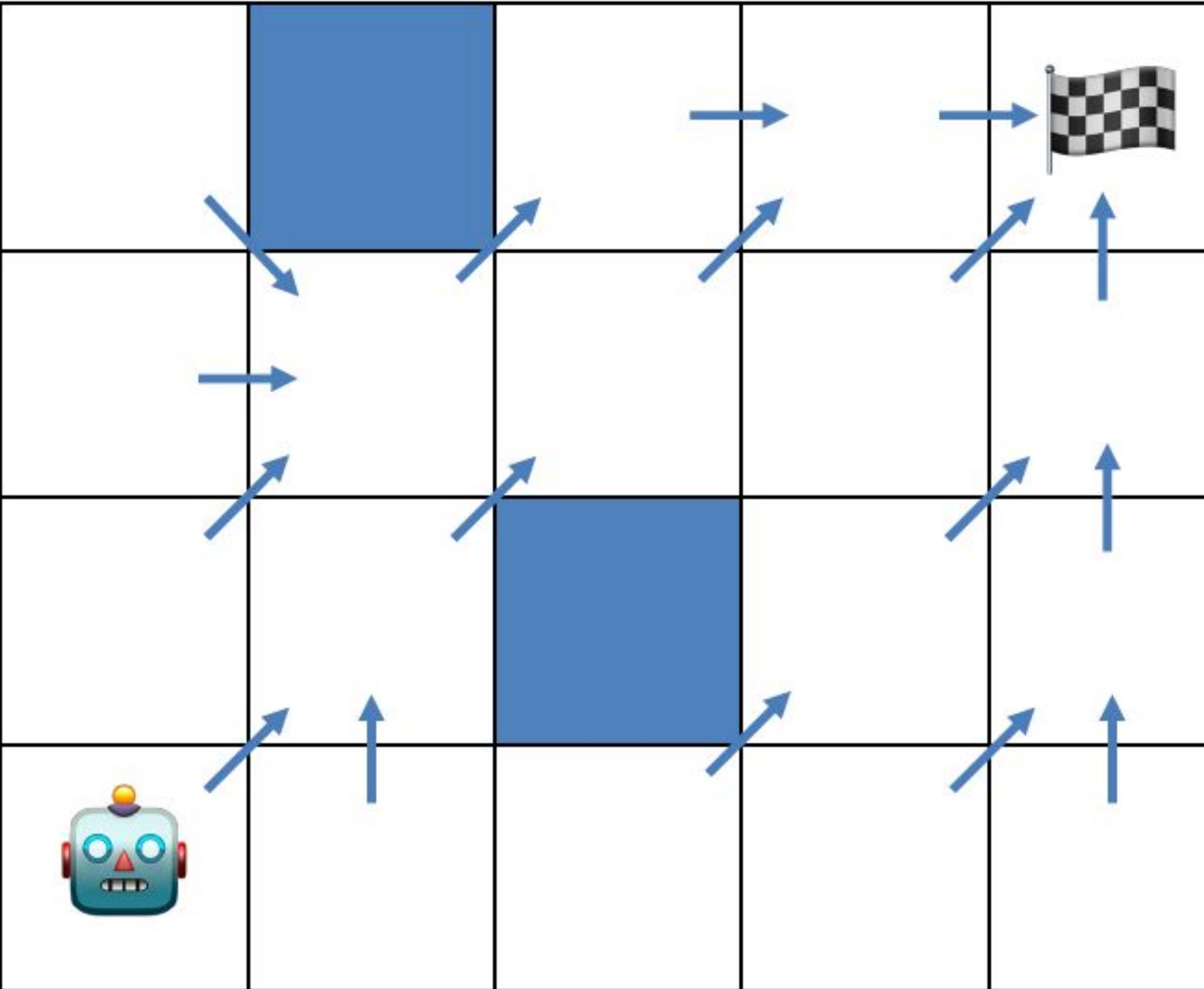
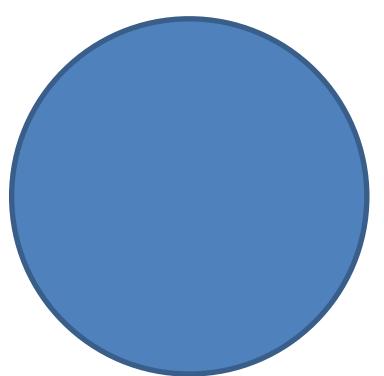
El objetivo de un agente en un MDP es encontrar la política óptima que permita resolver de forma satisfactoria el modelado ($\pi: S \rightarrow A$).



Ejemplo: procesos de decisión de Markov (MDP)

Política:

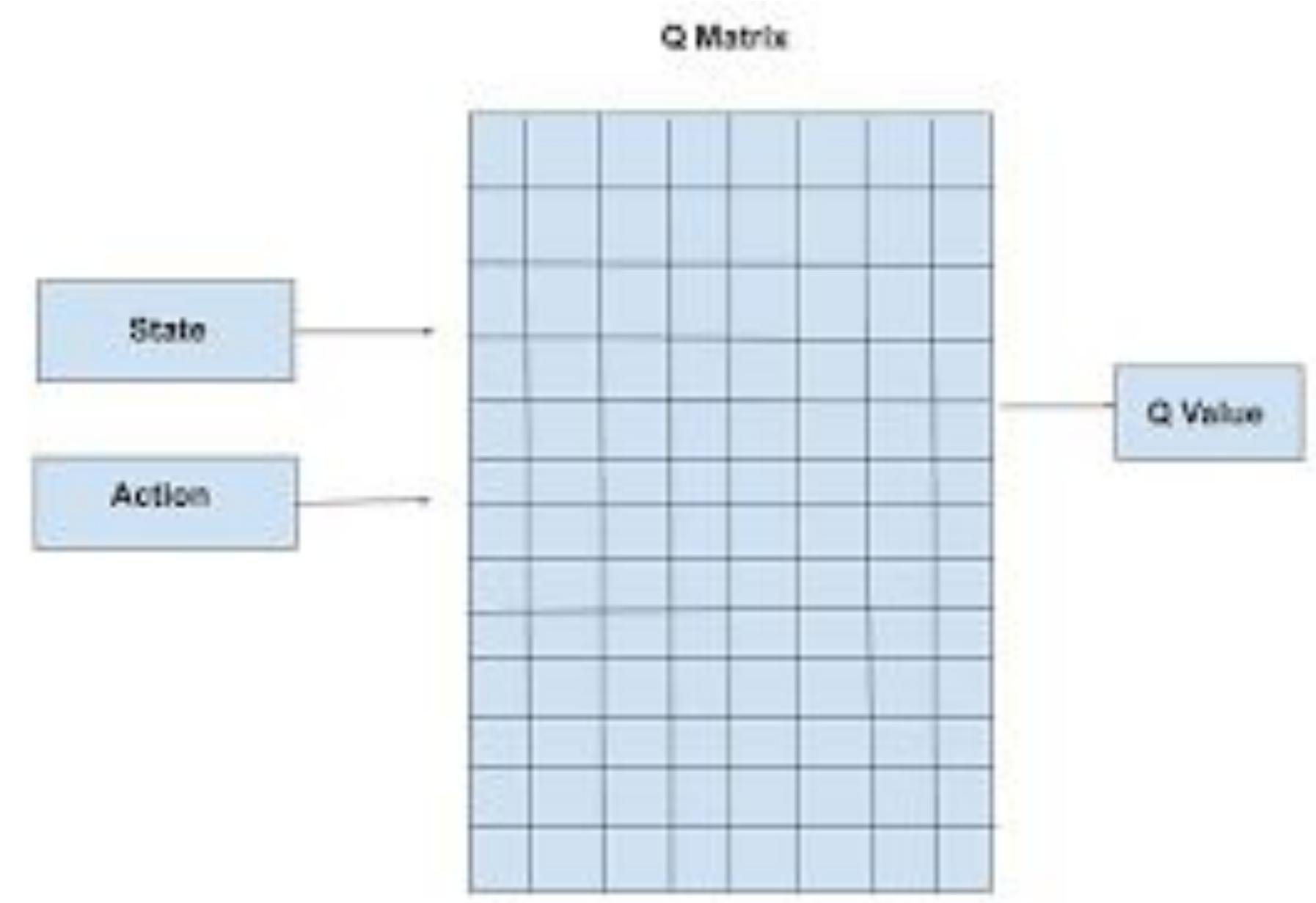
$$\pi(s_t) = a_t$$



07

Algoritmo Q-Learning

Definición de la función valor, la función Q y presentación del algoritmo general Q-Learning





MDP: Función valor

Para obtener una/la política óptima que resuelva un problema dado, se define la función de valor acumulado $V^\pi(s_t)$ para una política π y un estado s_t :

$$V^\pi(s_t) = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \dots + \gamma^n r_{t+n}$$

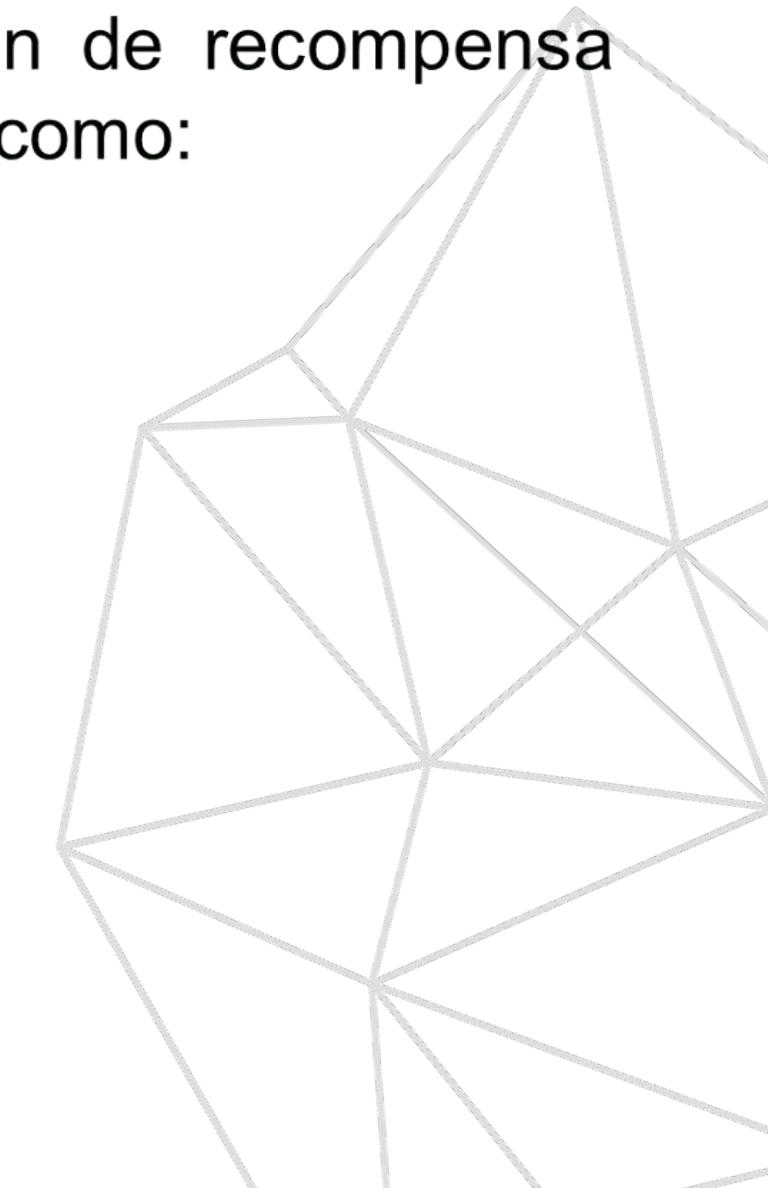
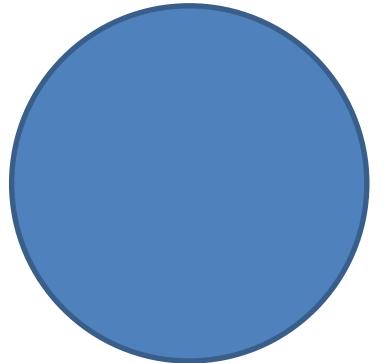
Donde $\gamma \in [0,1]$ es el factor de descuento. A esta función se le denomina función de recompensa acumulada con descuento para la política π y el estado s_t . También se puede escribir como:

$$V^\pi(s_t) = \sum_{i=0}^{\infty} \gamma^i r_{t+i}$$

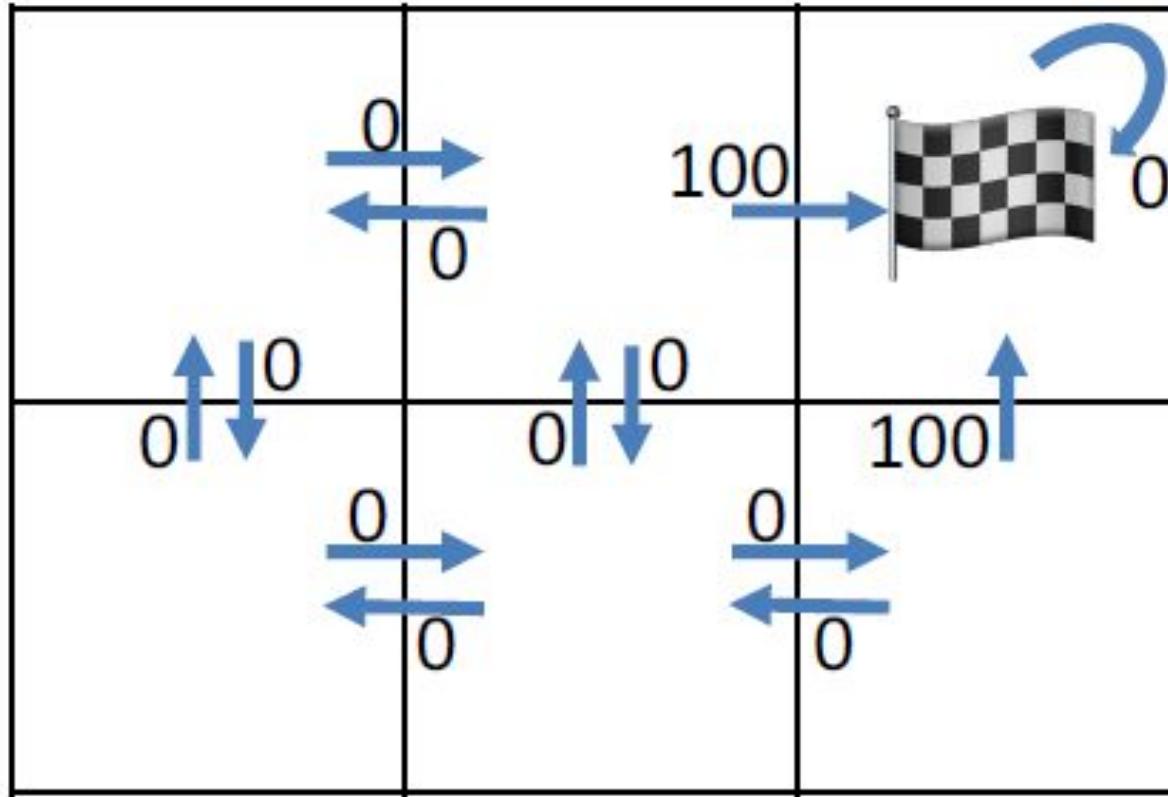
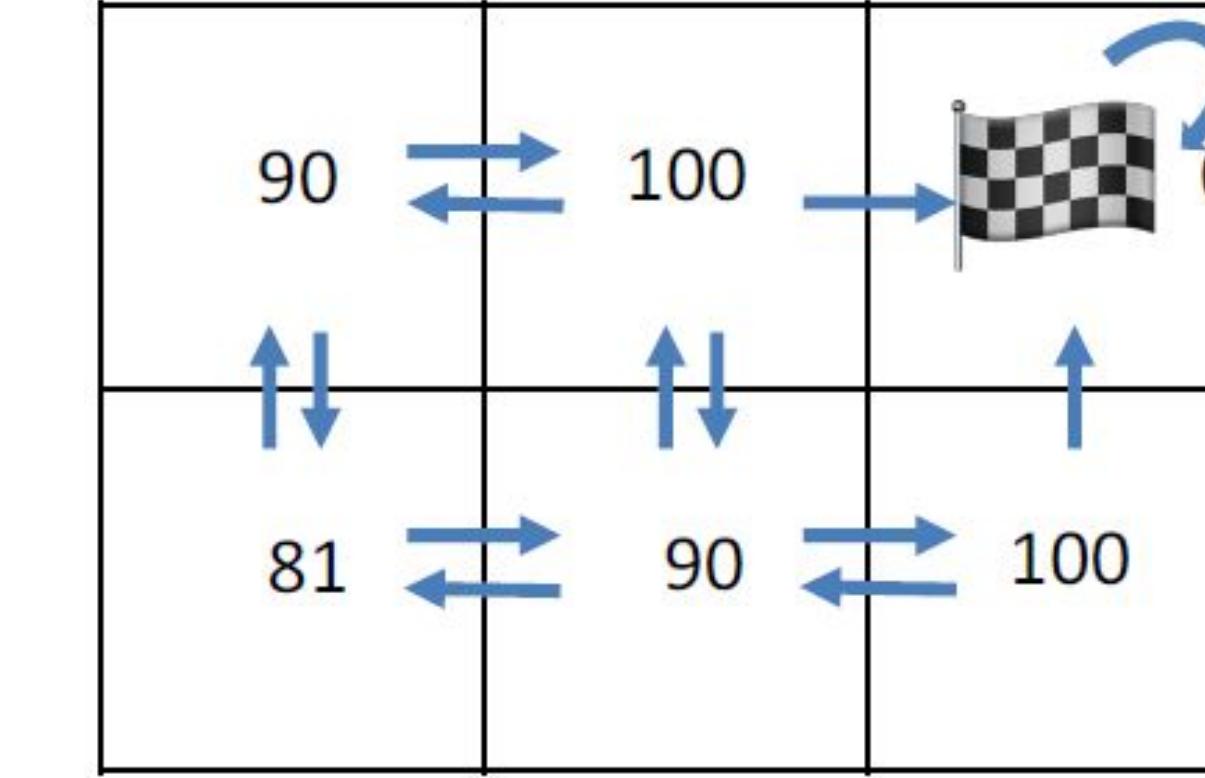
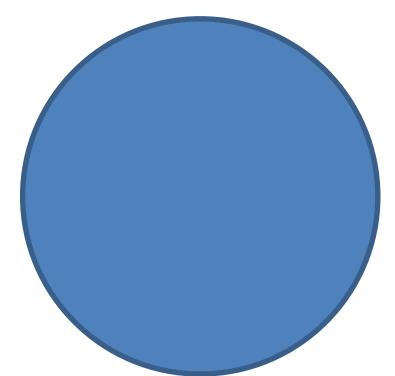
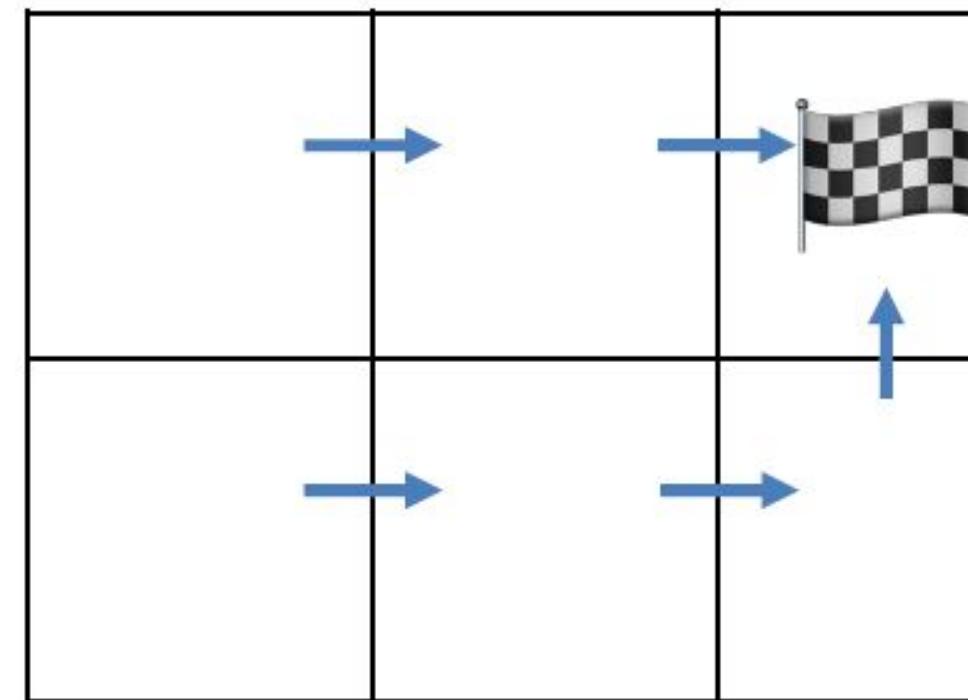
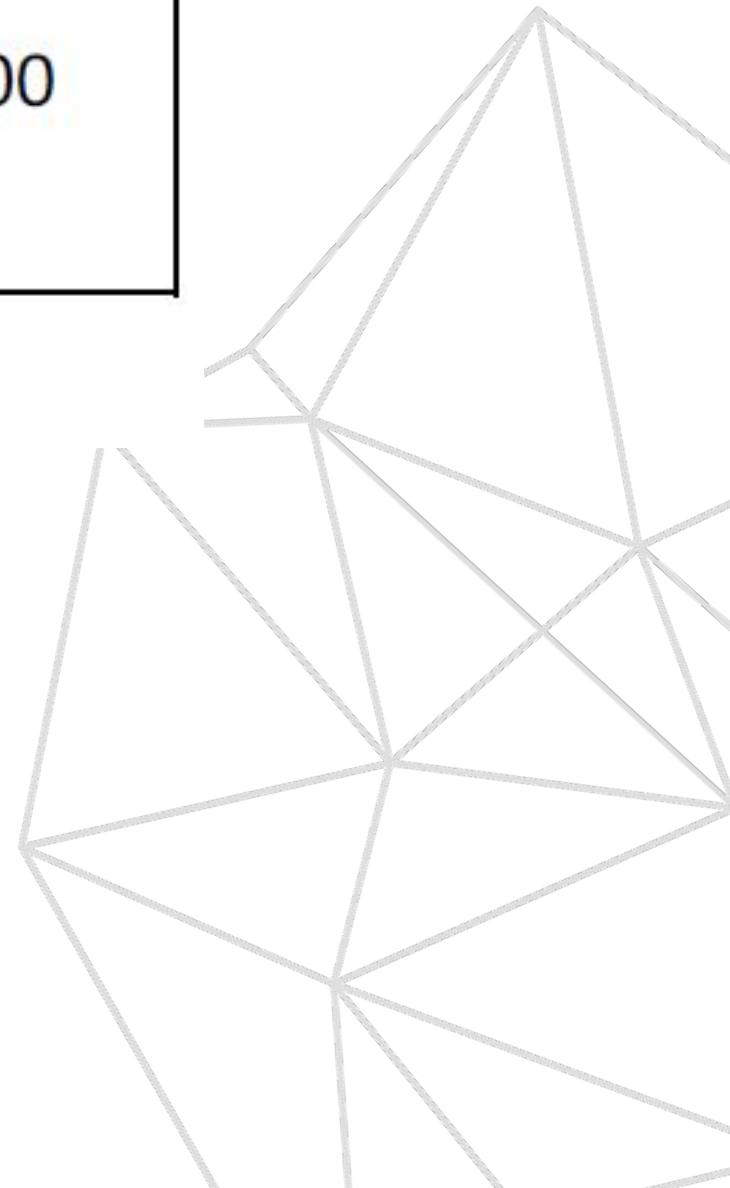
Así podemos definir un criterio para la optimización de la política tal que:

$$\pi^* \equiv \arg \max_{\pi} V^\pi(s)$$

sea la política óptima para todo estado.



MDP: Función valor (Ejemplo)

 $r(s_t, a_t)$  $V^*(s), \text{con } \gamma = 0.9$  π^* 

Tipos de aprendizaje por refuerzo

Podemos definir dos tipos de aprendizaje por refuerzo:

- **Pasivo:** se sigue una política de decisiones fija. El objetivo es obtener el valor o utilidad de cada estado para esa política, es decir, obtener la función valor para cada estado visitado.
- **Activo:** se da libertad al agente para elegir las acciones que considere oportunas. El objetivo es aprender las mejores acciones a tomar según el estado, es decir, la política. Este tipo de aprendizaje por refuerzo resulta de utilidad en aprendizaje online (como los juegos). Un ejemplo de algoritmo para este tipo de aprendizaje por refuerzo es el *Q Learning*.

	Value Iteration	RL Pasivo	RL Activo
Estados y recompensas	Se conocen todos los estados y recompensas del entorno	Se conocen los estados y recompensas visitados por el agente	Se conocen los estados y recompensas visitados por el agente
Acciones	Se conocen todas las probabilidades de acción	Se conocen las acciones elegidas por el agente	Se conocen las acciones elegidas por el agente
Decisiones		El algoritmo no elige las acciones	El algoritmo elige las acciones



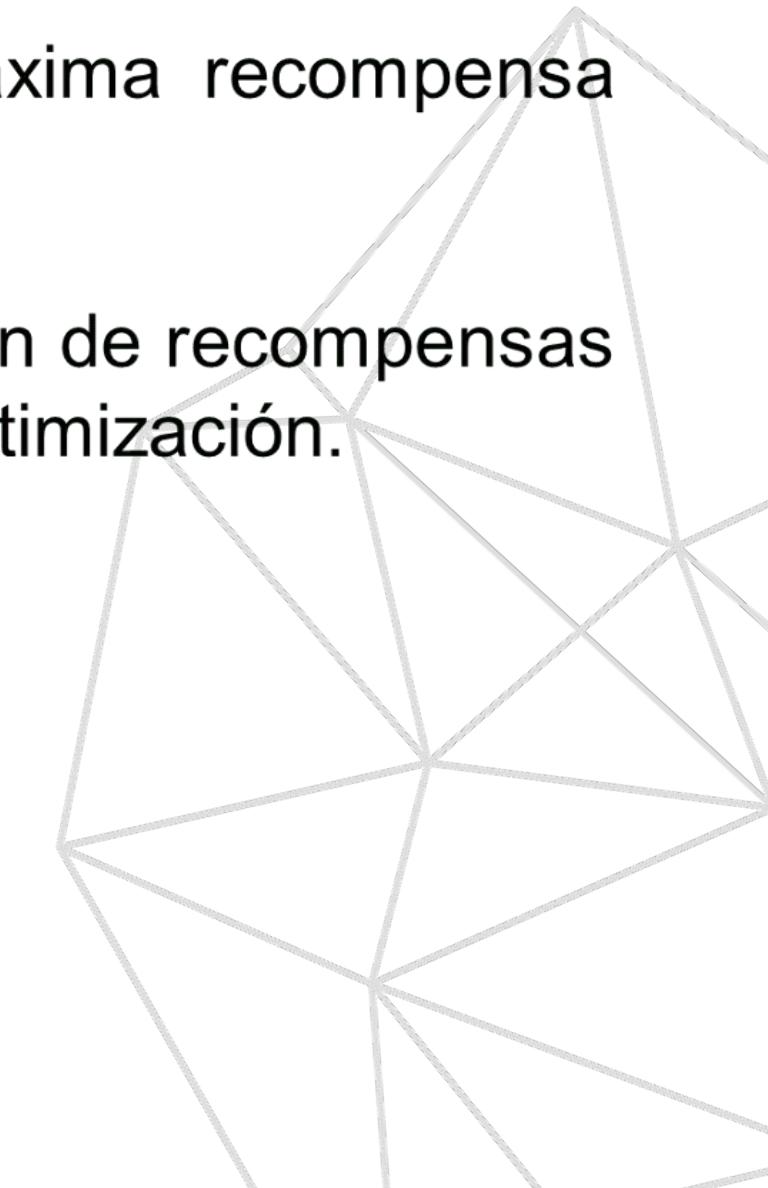
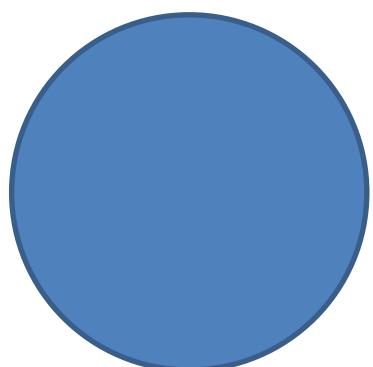
Optimización de la política

Dado un estado s_t , podemos definir el proceso de optimización de la política, de la siguiente forma:

$$\pi^*(s) \equiv \arg \max_a (r(s, a) + \gamma V^*(\delta(s, a)))$$

Siendo $V^*(s)$ la función valor de la política óptima π^* (la cual representa la máxima recompensa acumulada con descuento).

No obstante, en entornos reales, por lo general, no se tiene conocimiento de la función de recompensas ni de la función de transición de estados y, por lo tanto, no es factible realizar dicha optimización.





Función Q

Se define una nueva función (llamada función Q o *Q function*) que encapsule las funciones desconocidas de modo que podamos realizar un proceso de optimización:

$$Q(s, a) = r(s, a) + \gamma V^*(\delta(s, a))$$

De esta manera se reescribe la función de optimización como:

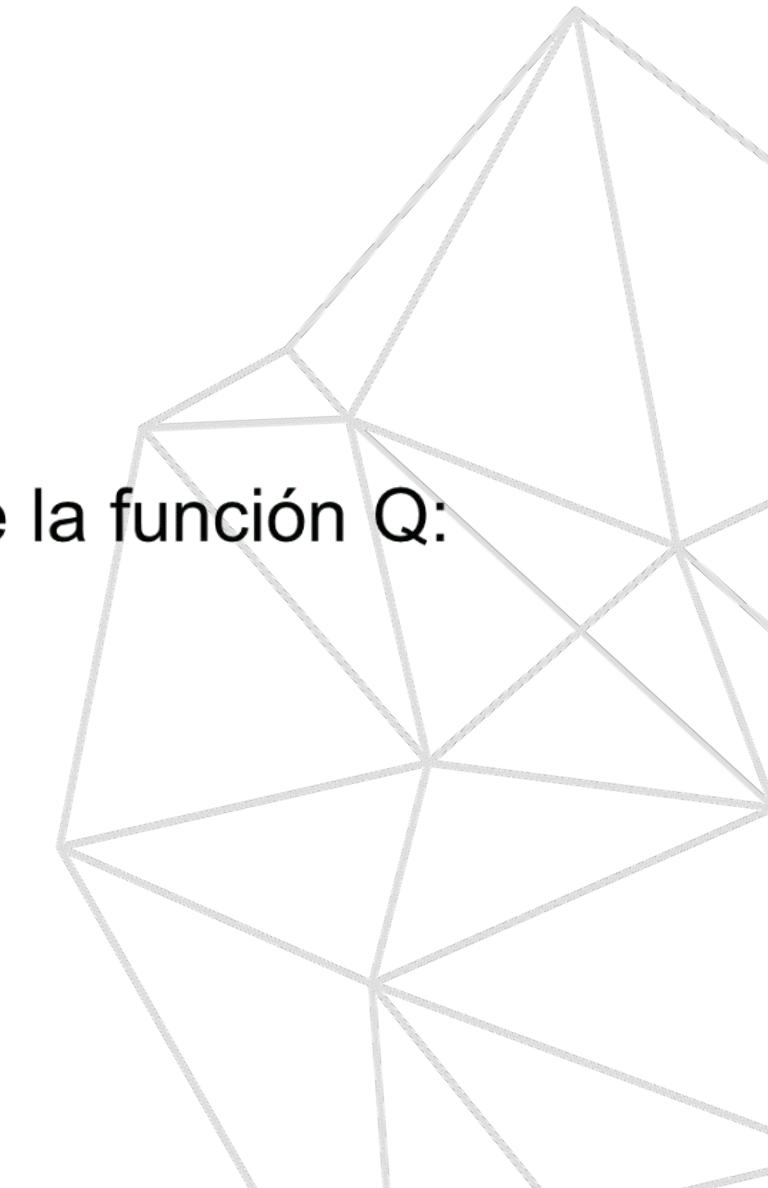
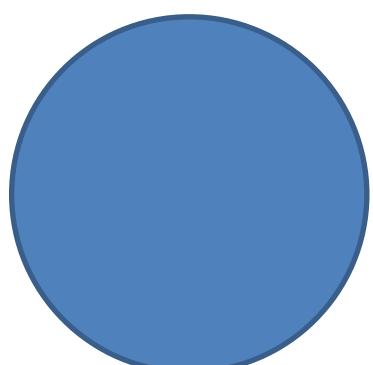
$$\pi^*(s) \equiv \arg \max_a Q(s, a)$$

Por otro lado, también puede escribirse la función valor de la política óptima a partir de la función Q:

$$V^*(s) = \max_a Q(s, a)$$

Por lo que es posible redefinir la función Q a partir de sí misma:

$$Q(s, a) = r(s, a) + \gamma \max_{a'} Q(s', a')$$





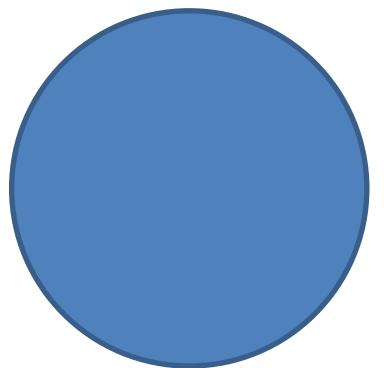
Q-Learning: Algoritmo

El objetivo será obtener el valor de $\hat{Q}(s, a)$ (una estimación de Q) para cada posible estado s y acción a :

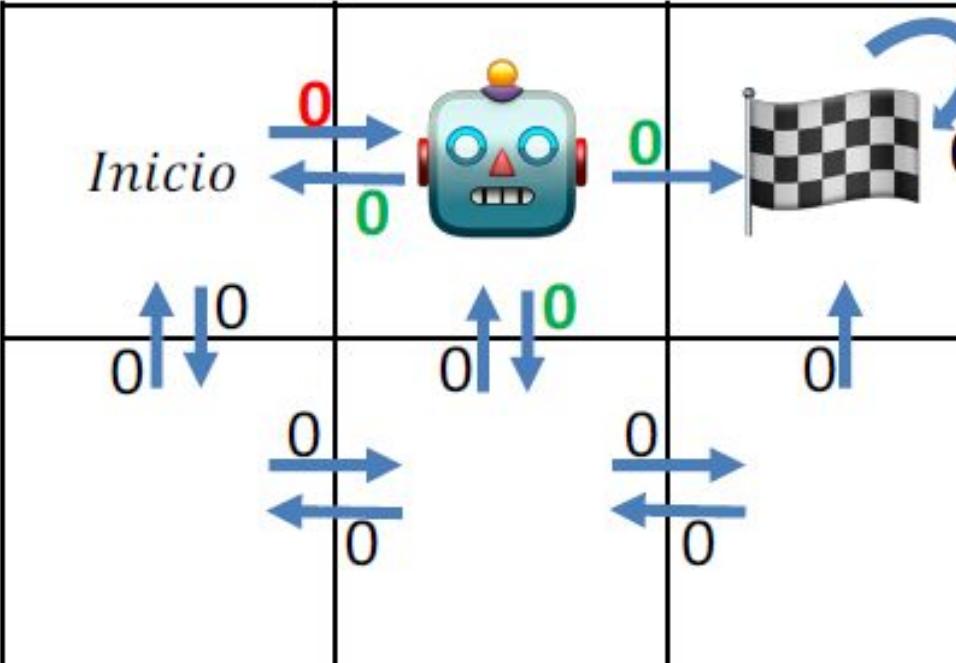
$$\hat{Q}(s, a) \leftarrow r(s, a) + \gamma \max_{a'} \hat{Q}(s', a')$$

El algoritmo será el siguiente:

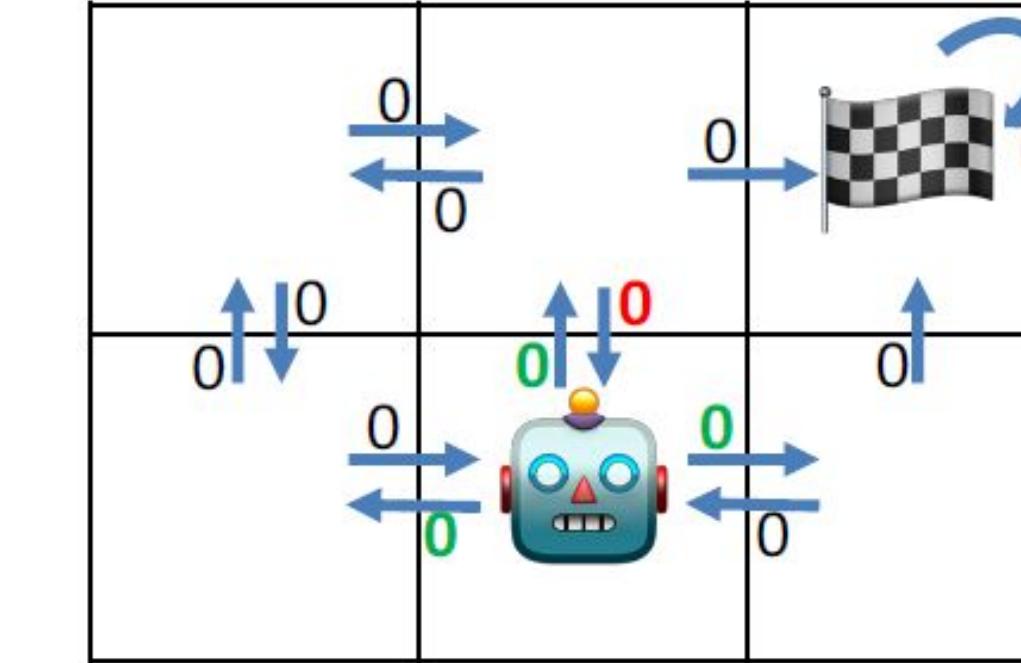
1. Para todo estado y acción inicializar $\hat{Q}(s, a)$ a 0
2. Asignar un estado inicial s de forma aleatoria
3. Ejecutar los siguientes pasos hasta que se cumpla una condición de parada:
4. Seleccionar una acción a y ejecutarla
5. Obtener la recompensa inmediata r
6. Observar el nuevo estado s'
7. Actualizar $\hat{Q}(s, a) \leftarrow r(s, a) + \gamma \max_{a'} \hat{Q}(s', a')$
8. Asignar s' a s
9. En caso de llegar a un estado final sin salida asignar un estado inicial s aleatoriamente



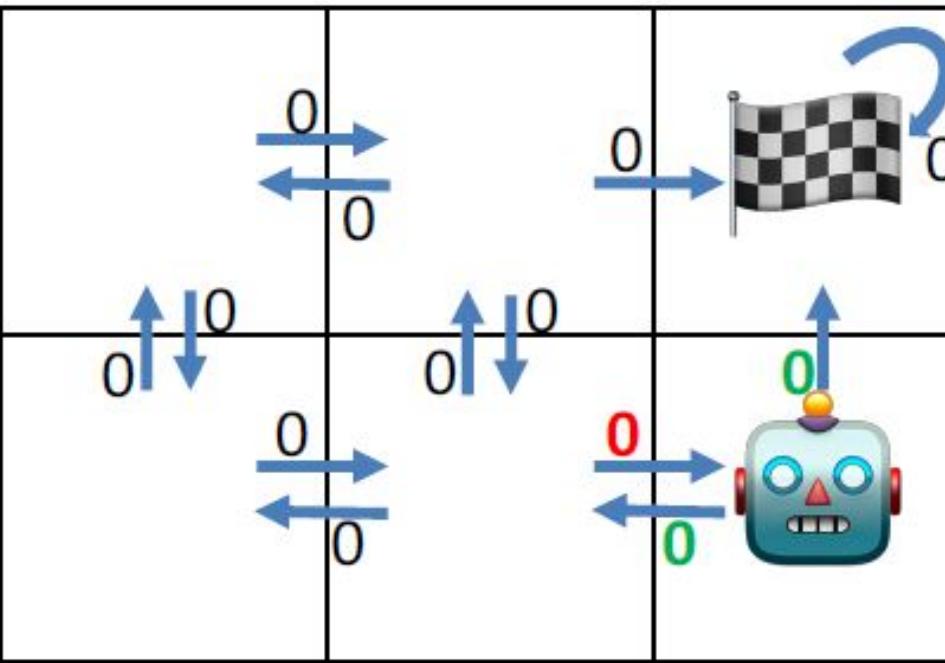
Q-Learning: Algoritmo (Ejemplo)



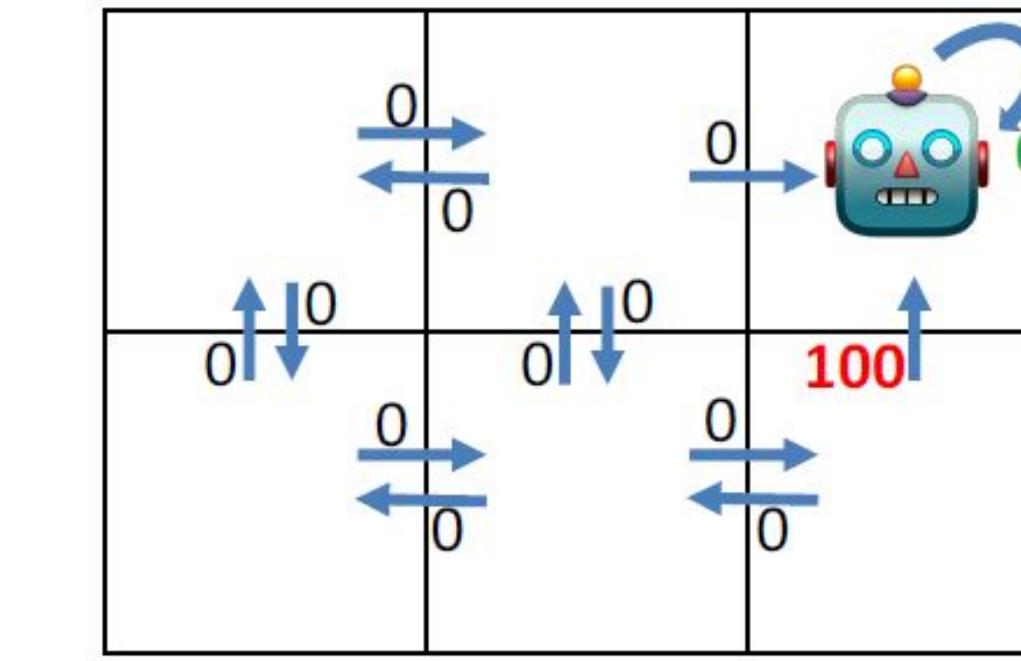
$$\hat{Q}(s_1, a_{right}) \leftarrow r + \gamma \max_{a'} \hat{Q}(s_2, a')$$



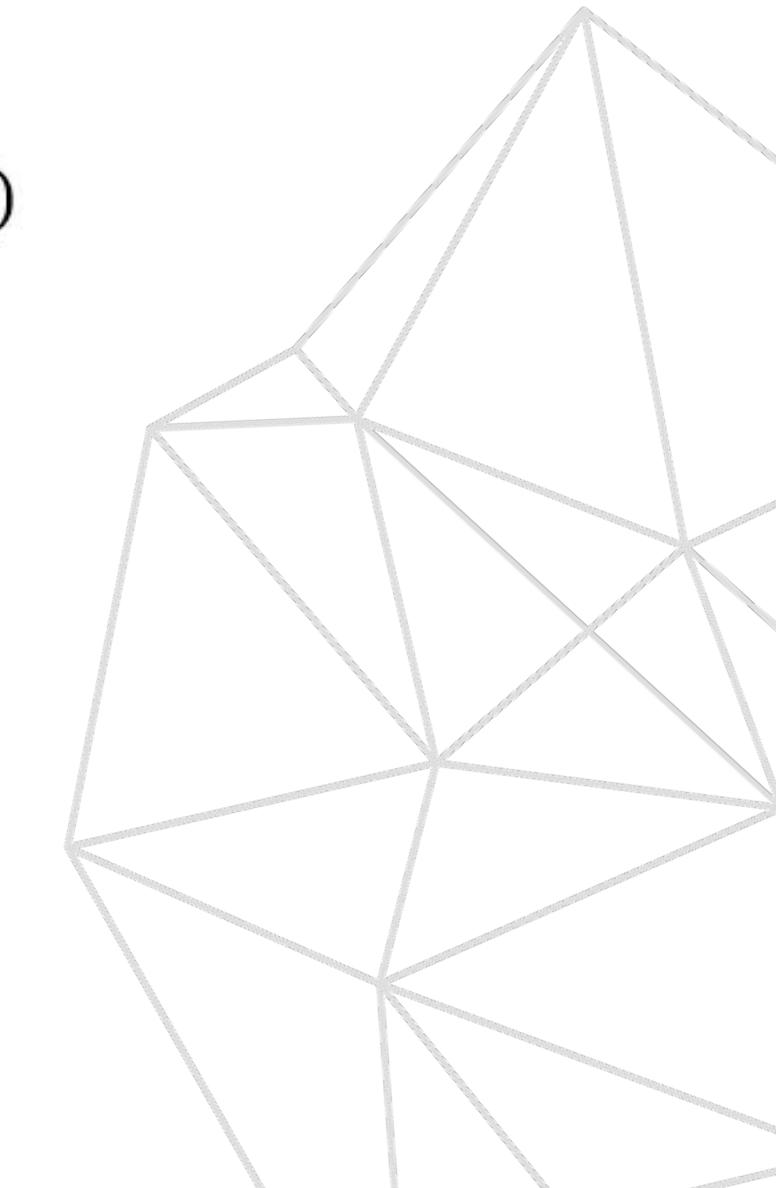
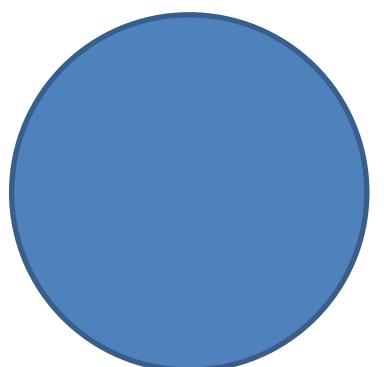
$$\hat{Q}(s_2, a_{down}) \leftarrow r + \gamma \max_{a'} \hat{Q}(s_5, a')$$



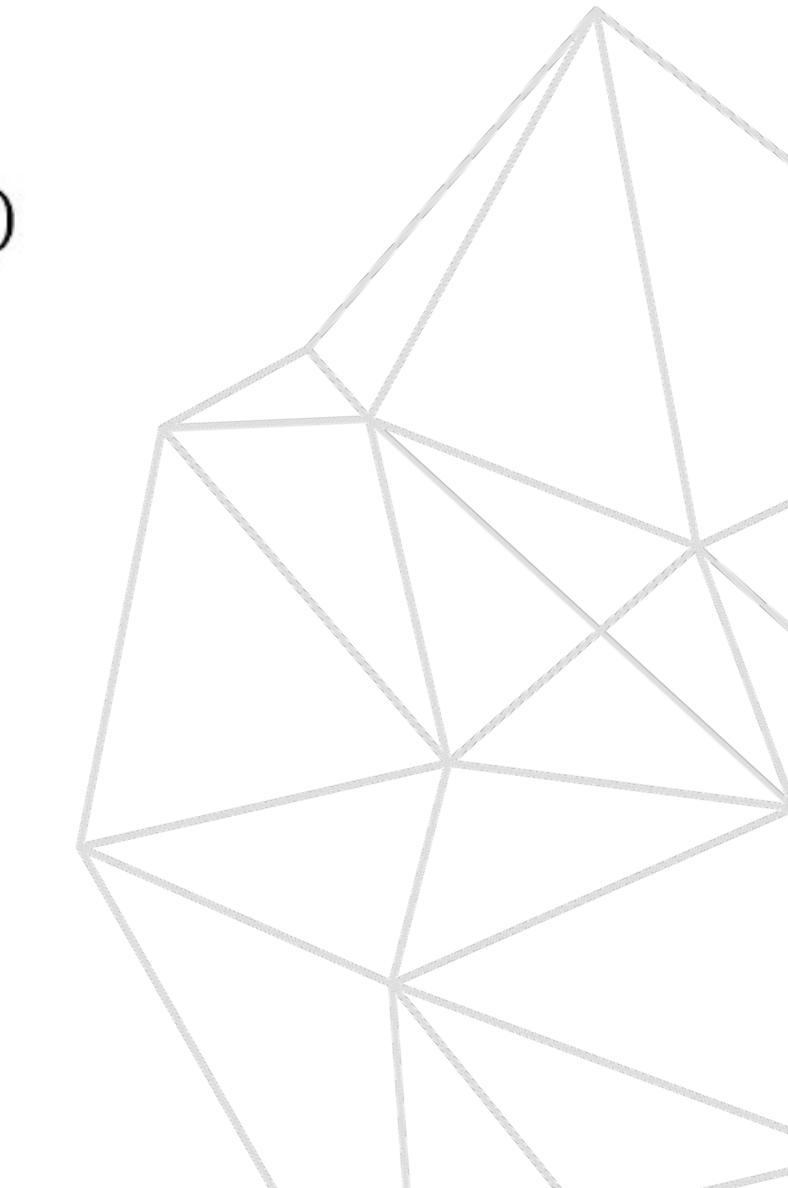
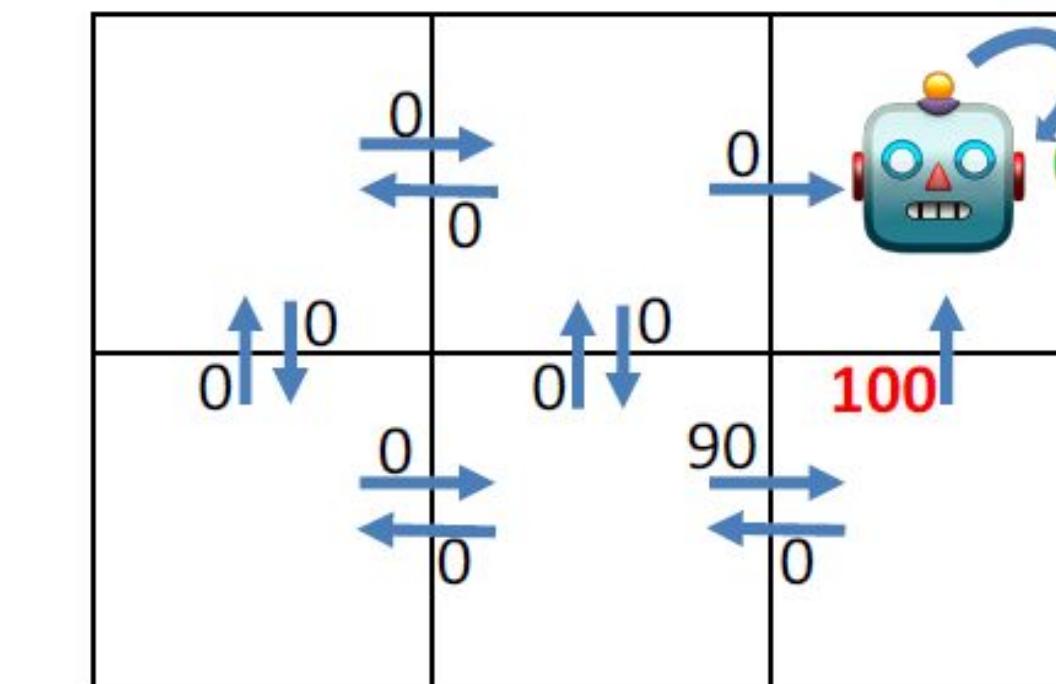
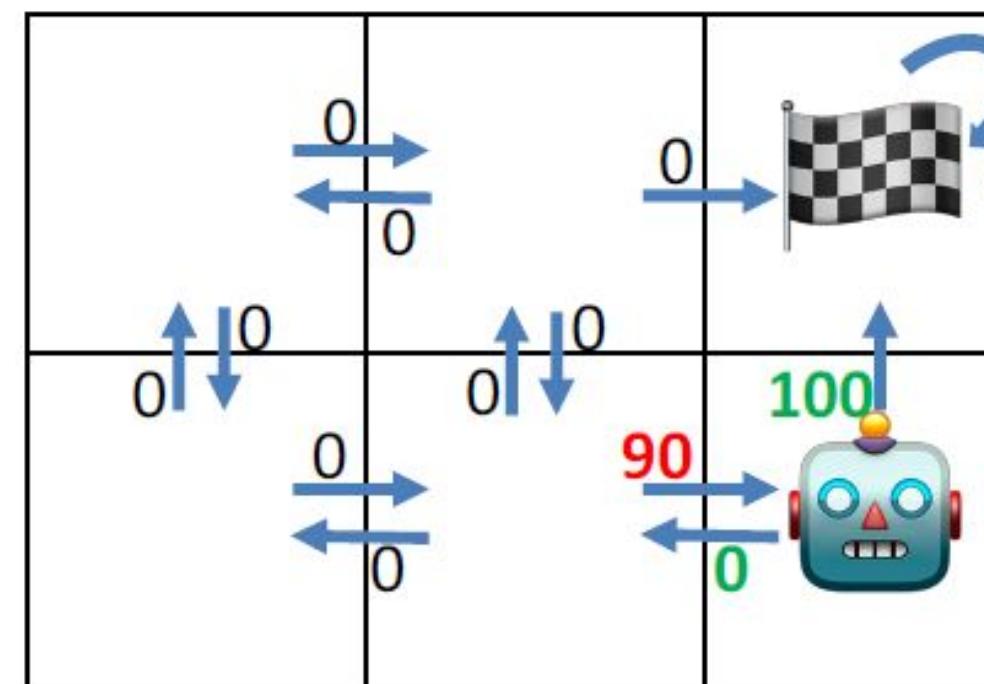
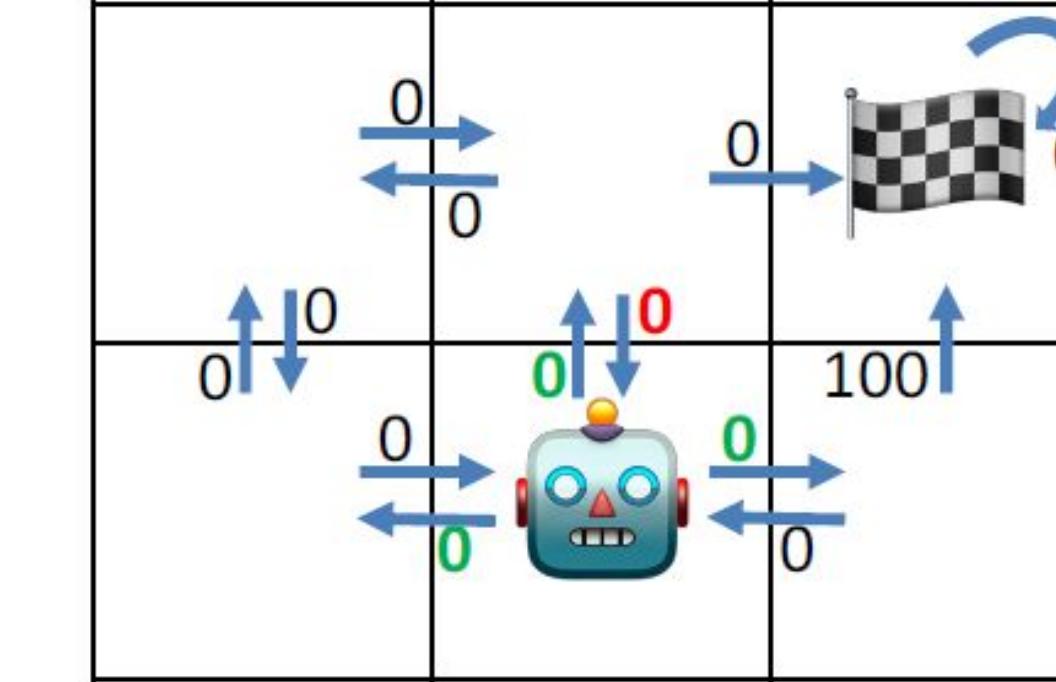
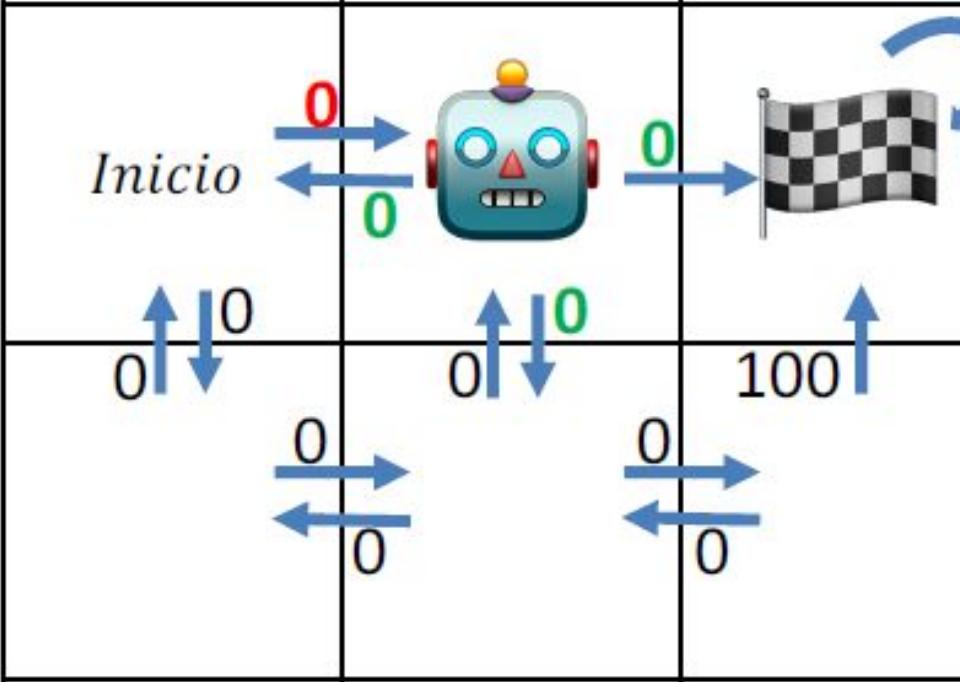
$$\hat{Q}(s_5, a_{right}) \leftarrow r + \gamma \max_{a'} \hat{Q}(s_6, a')$$



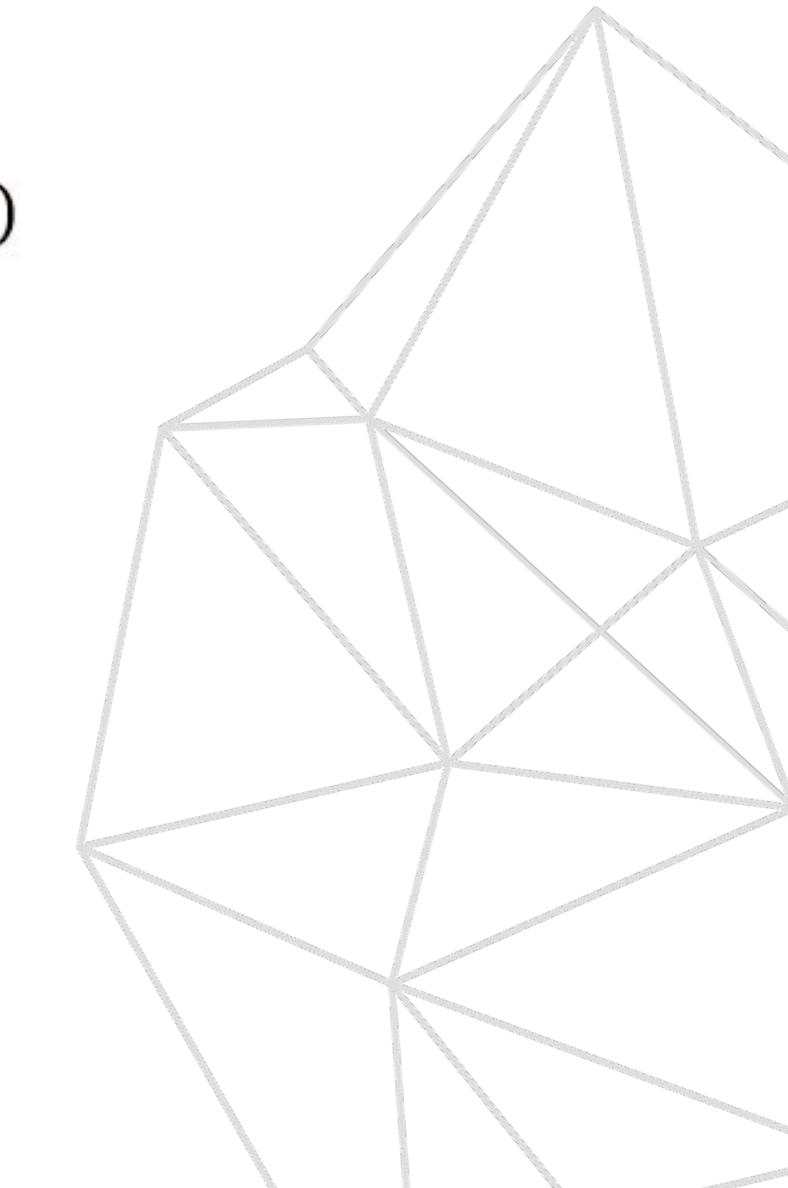
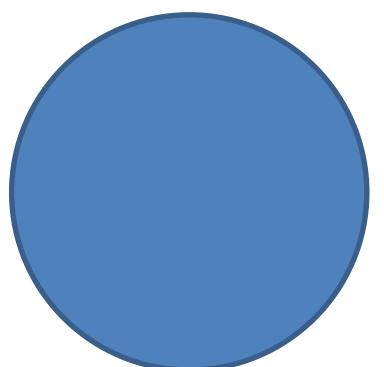
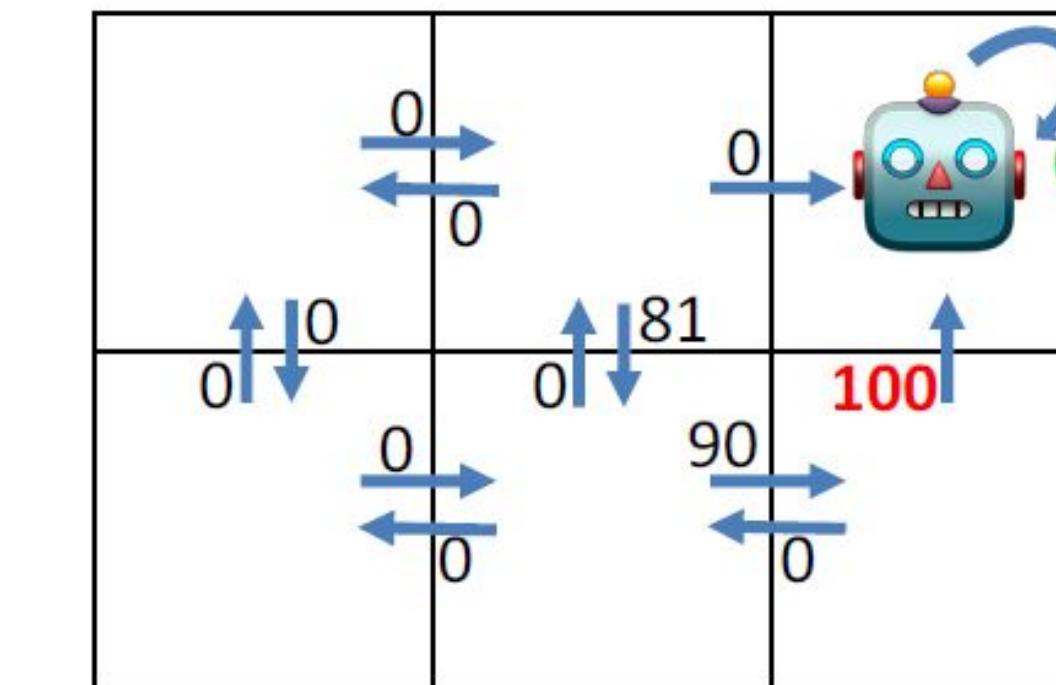
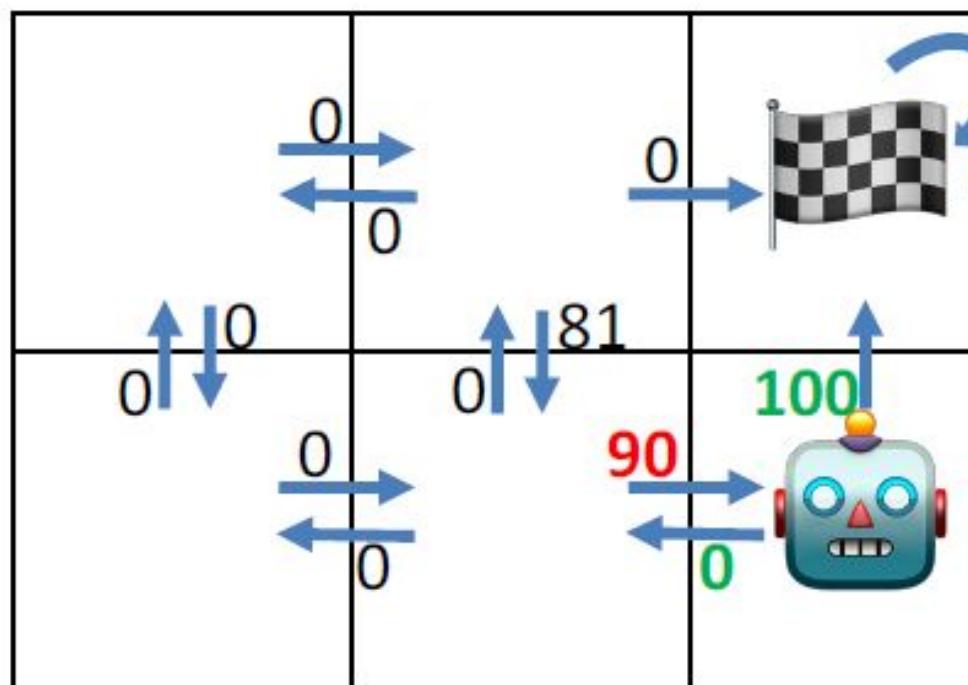
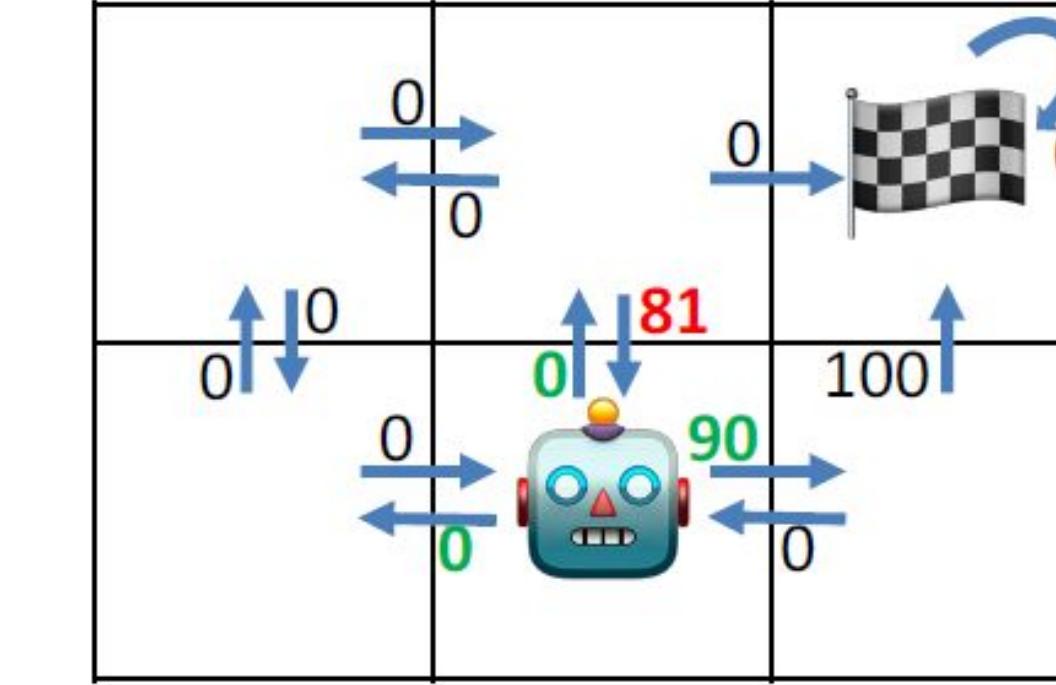
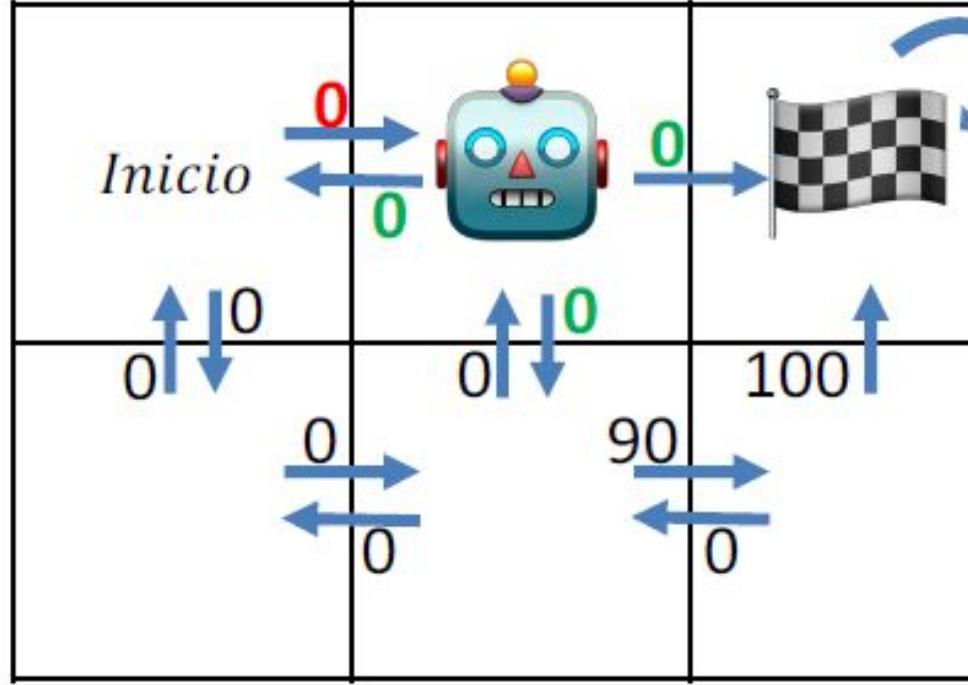
$$\hat{Q}(s_6, a_{up}) \leftarrow r + \gamma \max_{a'} \hat{Q}(s_3, a')$$



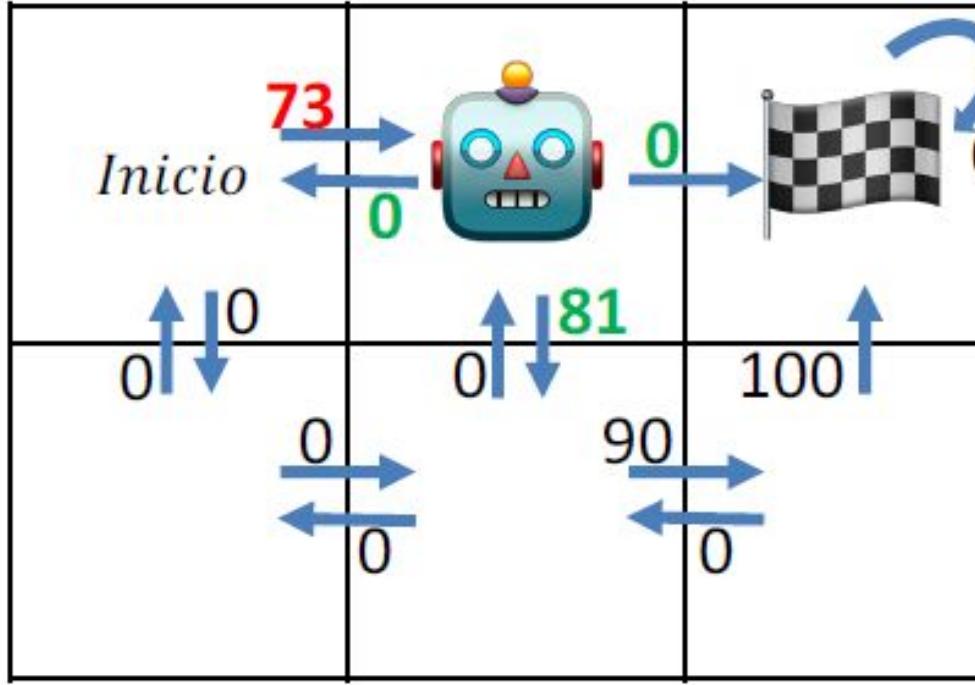
Q-Learning: Algoritmo (Ejemplo)



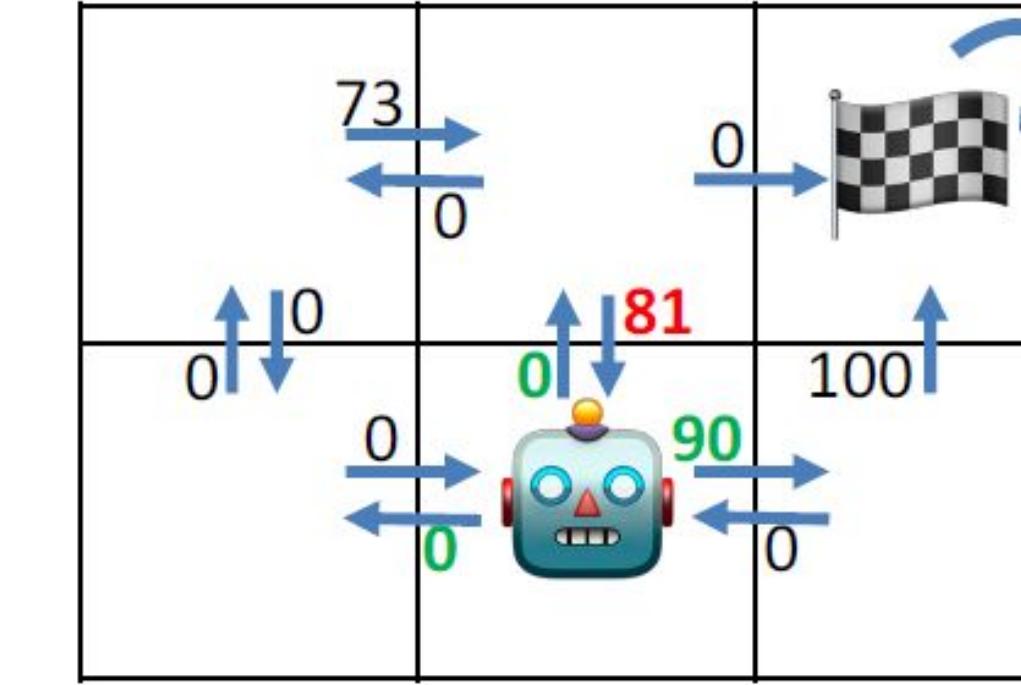
Q-Learning: Algoritmo (Ejemplo)



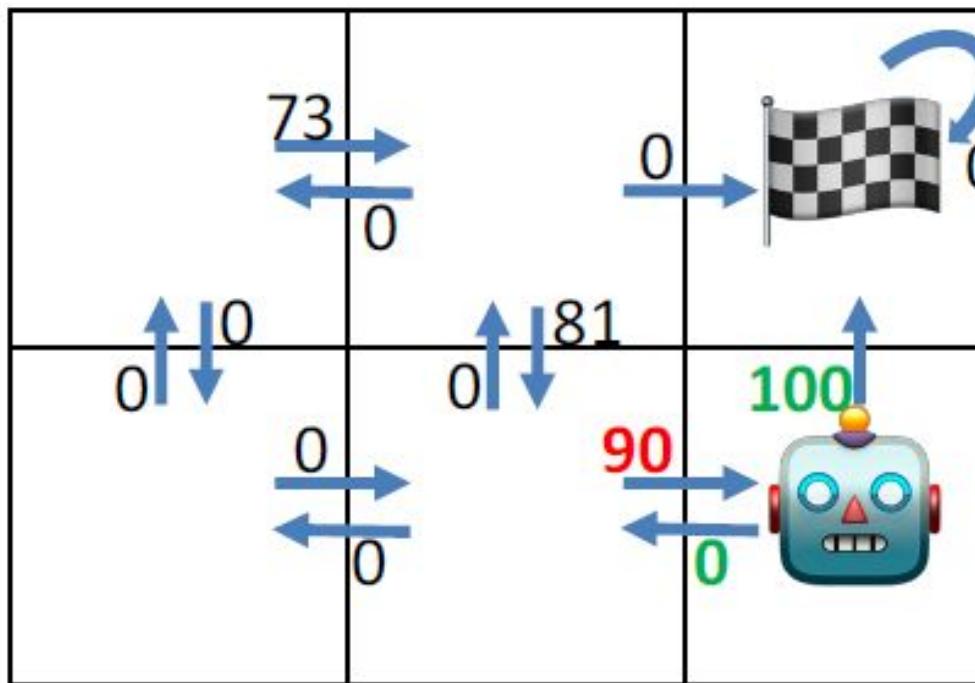
Q-Learning: Algoritmo (Ejemplo)



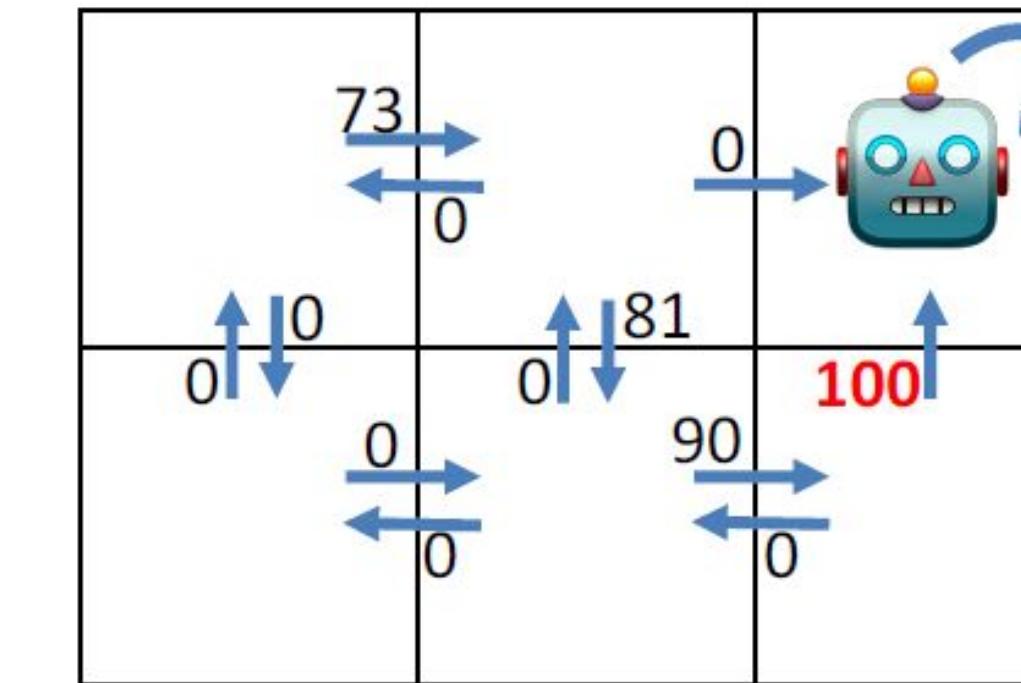
$$\hat{Q}(s_1, a_{right}) \leftarrow r + \gamma \max_{a'} \hat{Q}(s_2, a')$$



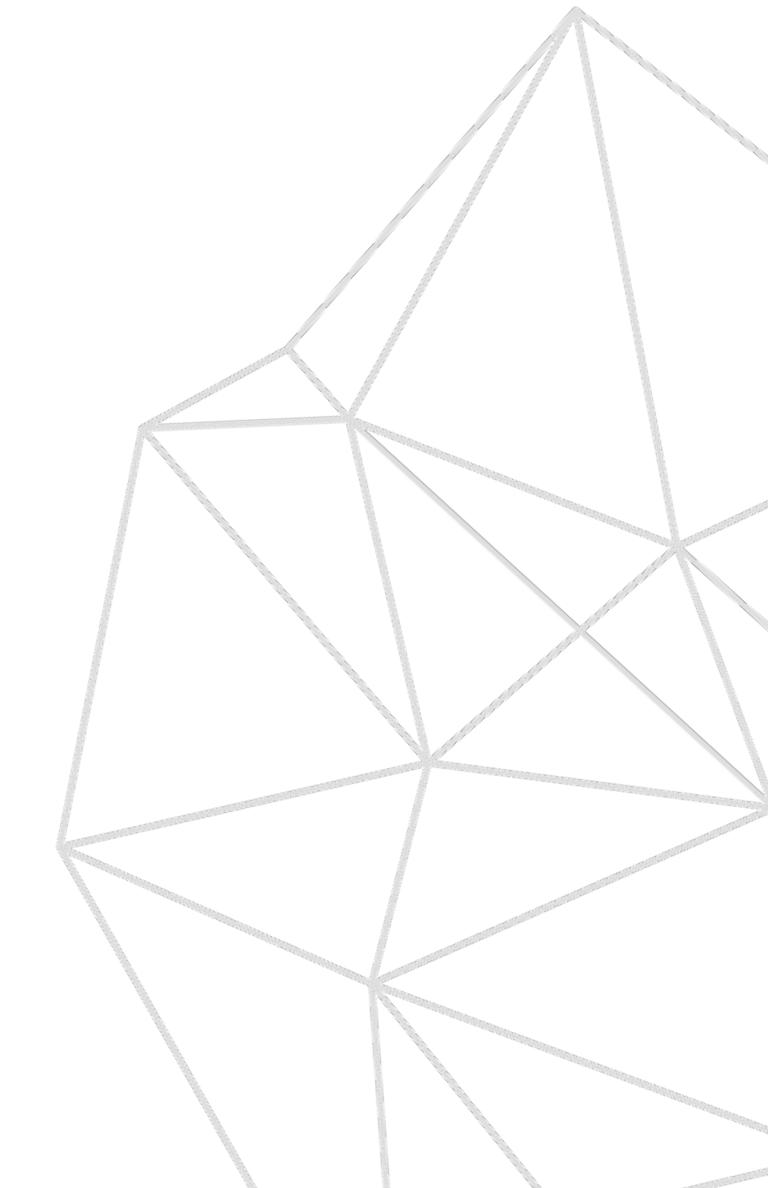
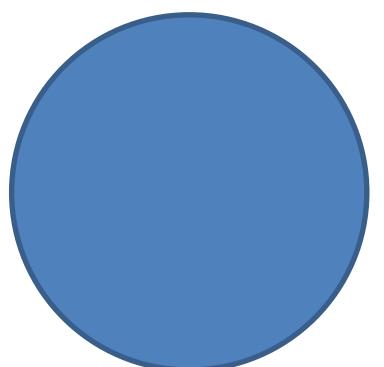
$$\hat{Q}(s_2, a_{down}) \leftarrow r + \gamma \max_{a'} \hat{Q}(s_5, a')$$



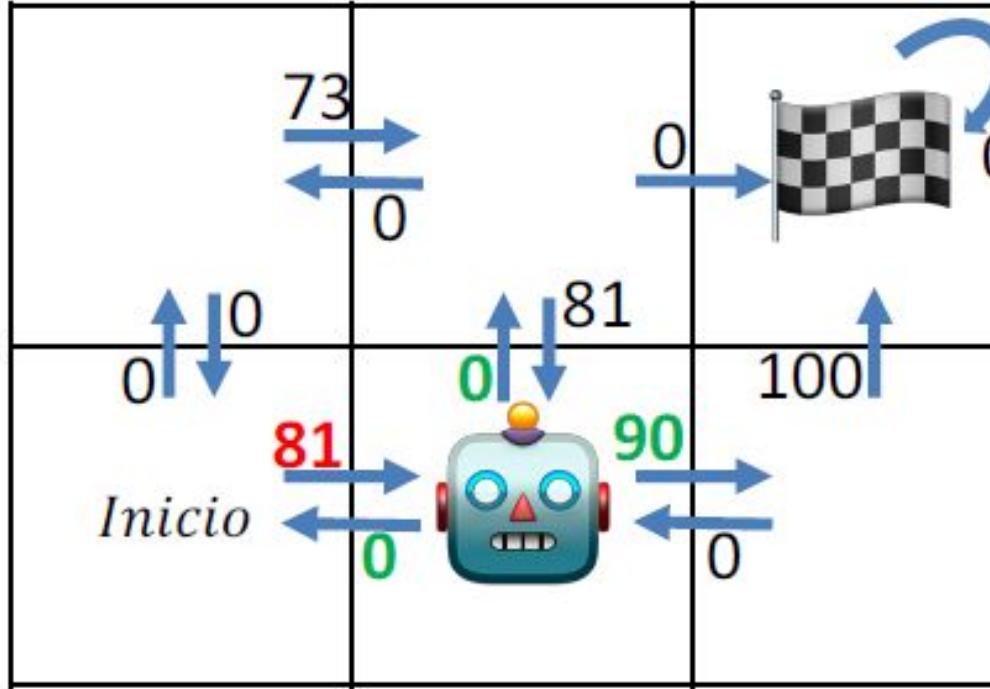
$$\hat{Q}(s_5, a_{right}) \leftarrow r + \gamma \max_{a'} \hat{Q}(s_6, a')$$



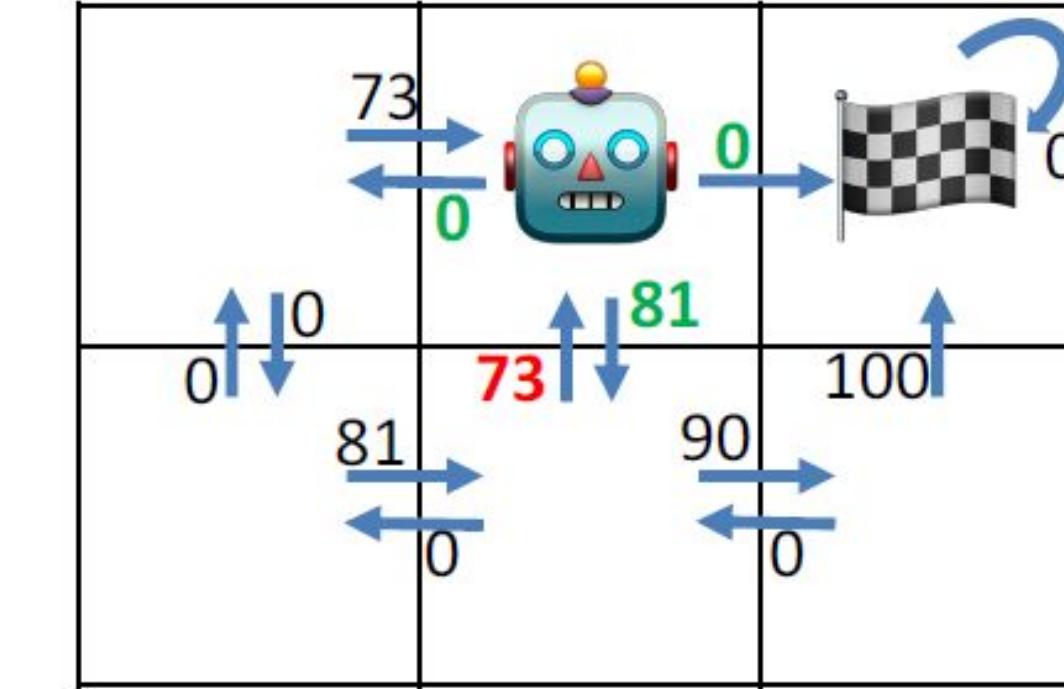
$$\hat{Q}(s_6, a_{up}) \leftarrow r + \gamma \max_{a'} \hat{Q}(s_3, a')$$



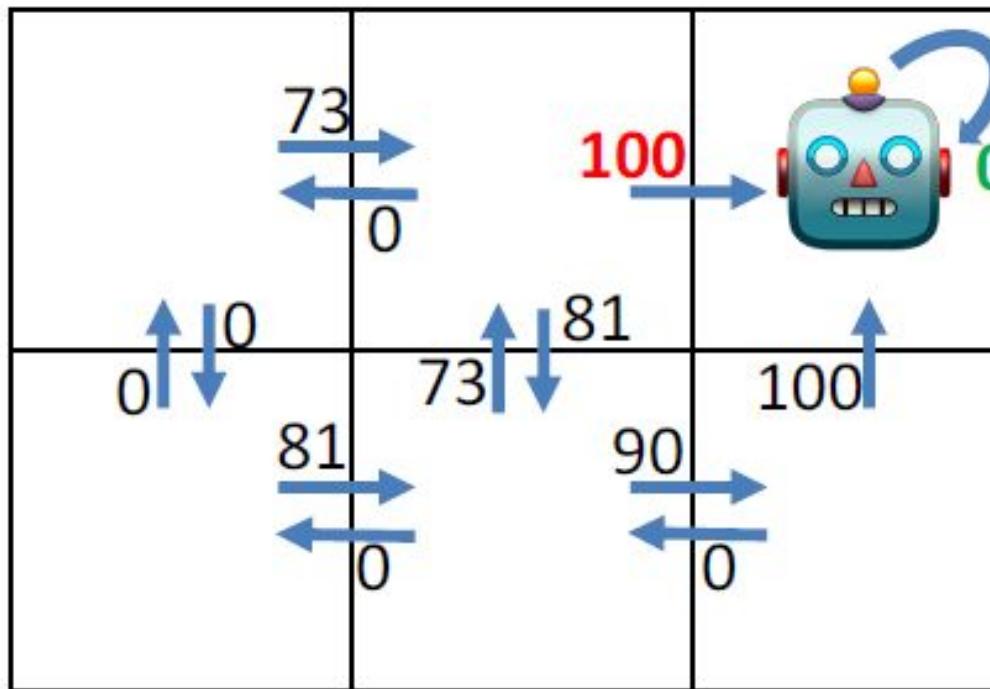
Q-Learning: Algoritmo (Ejemplo)



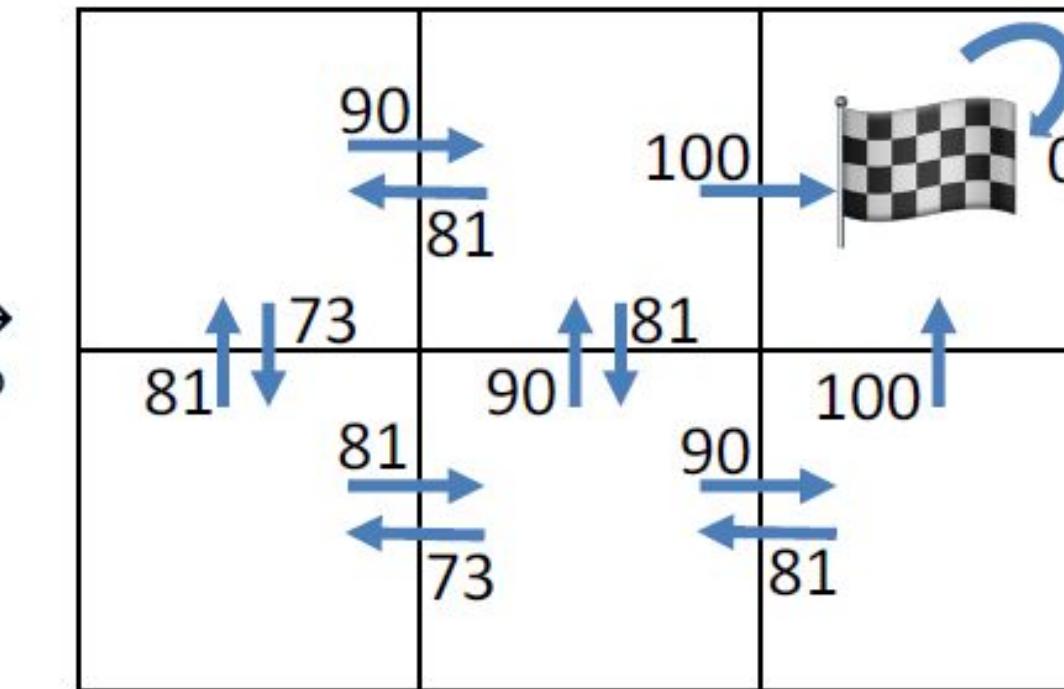
$$\hat{Q}(s_4, a_{right}) \leftarrow r + \gamma \max_{a'} \hat{Q}(s_5, a')$$



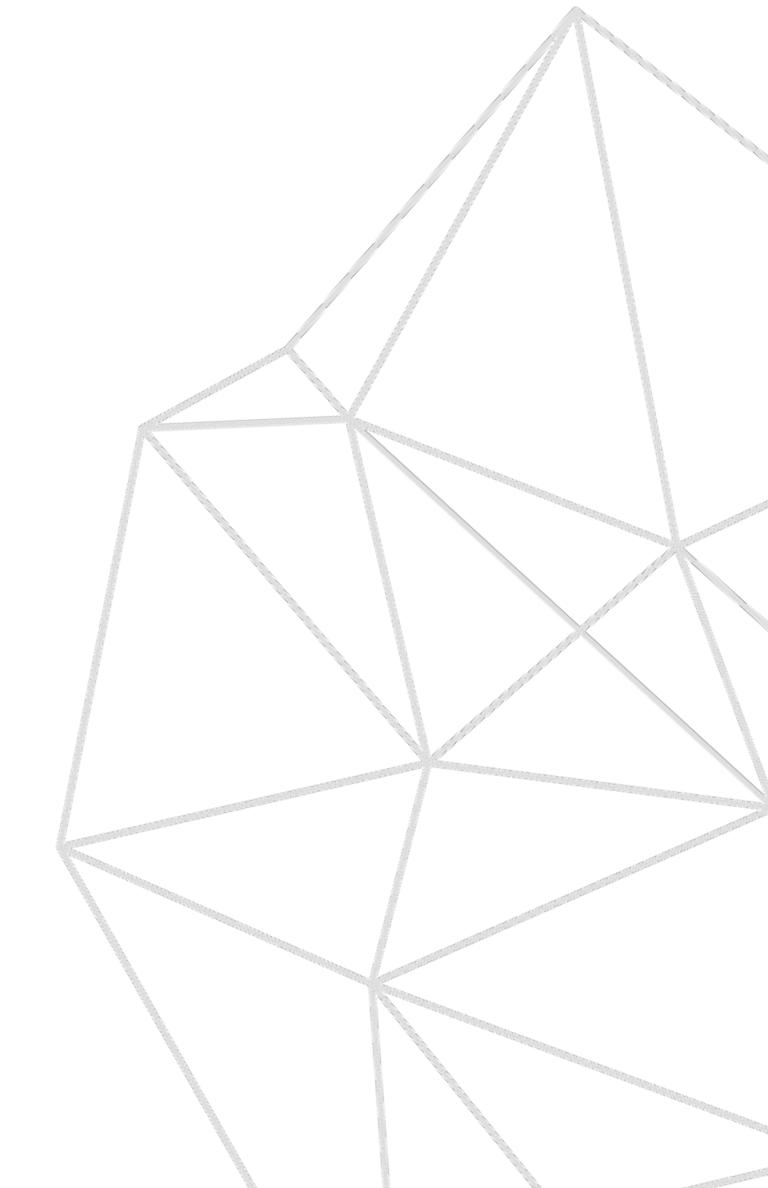
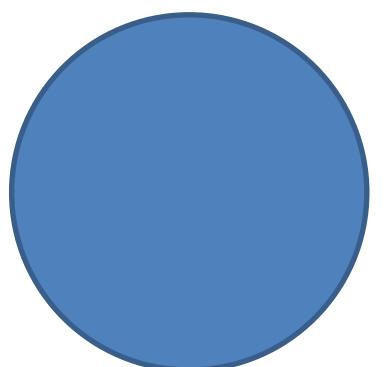
$$\hat{Q}(s_5, a_{up}) \leftarrow r + \gamma \max_{a'} \hat{Q}(s_2, a')$$



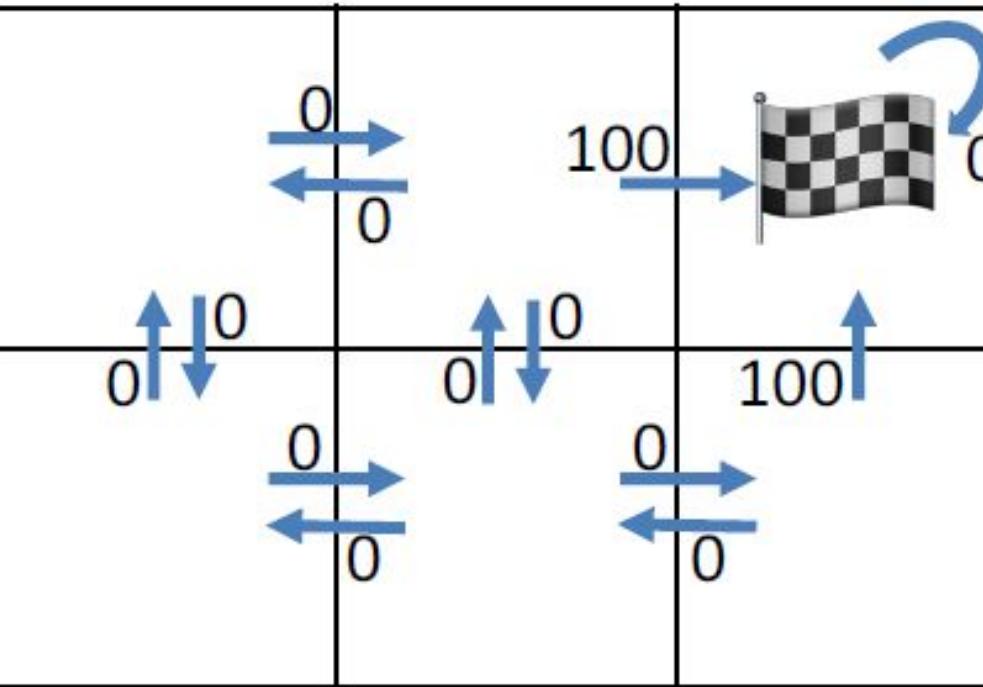
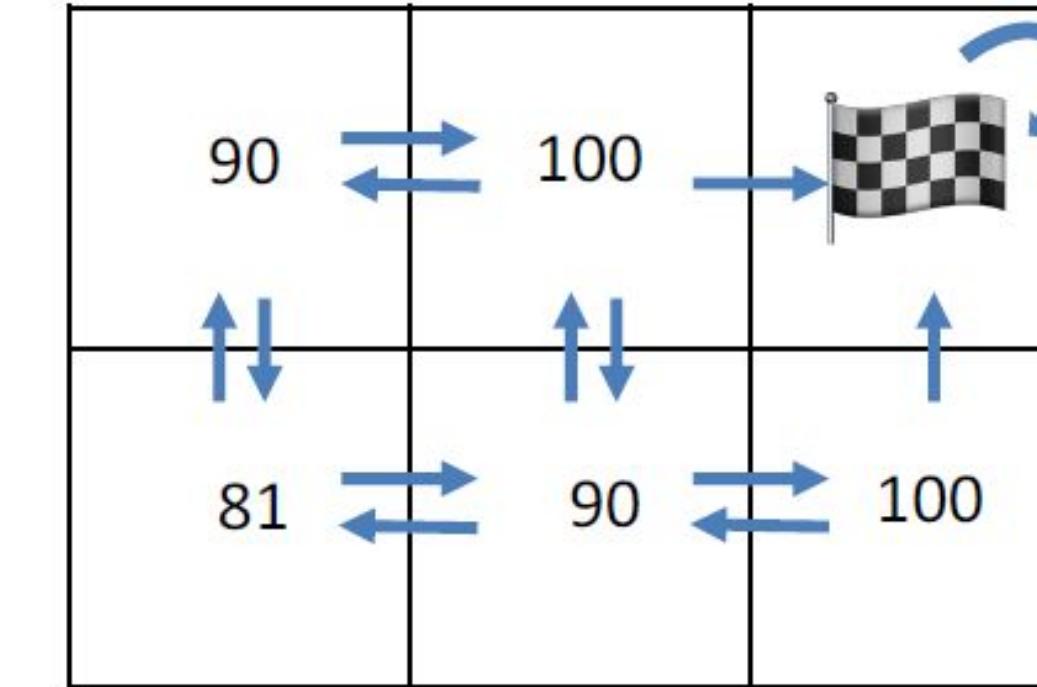
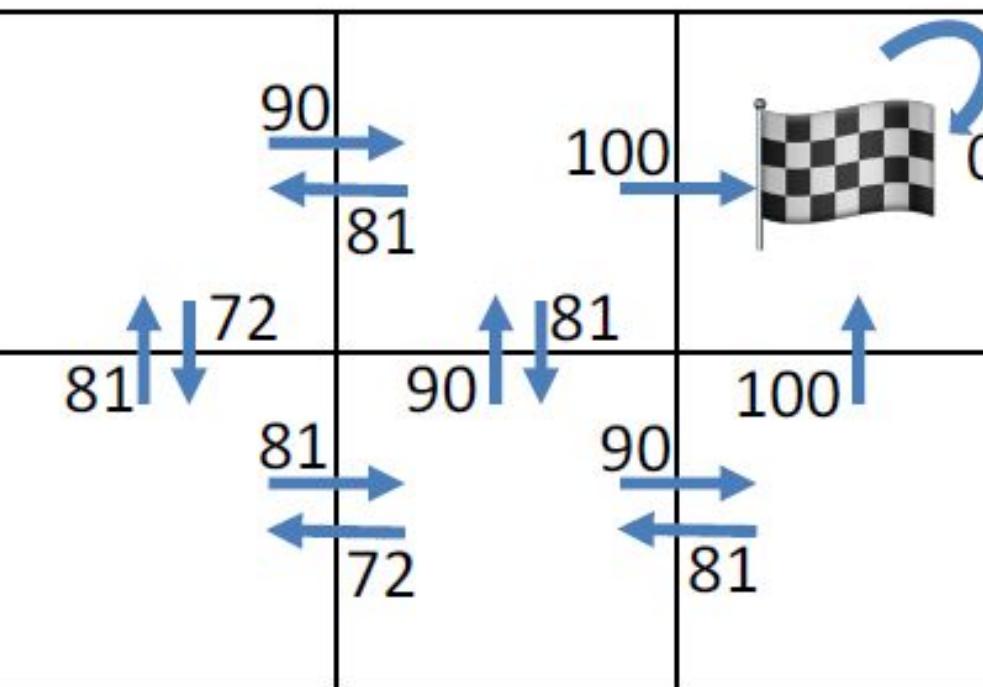
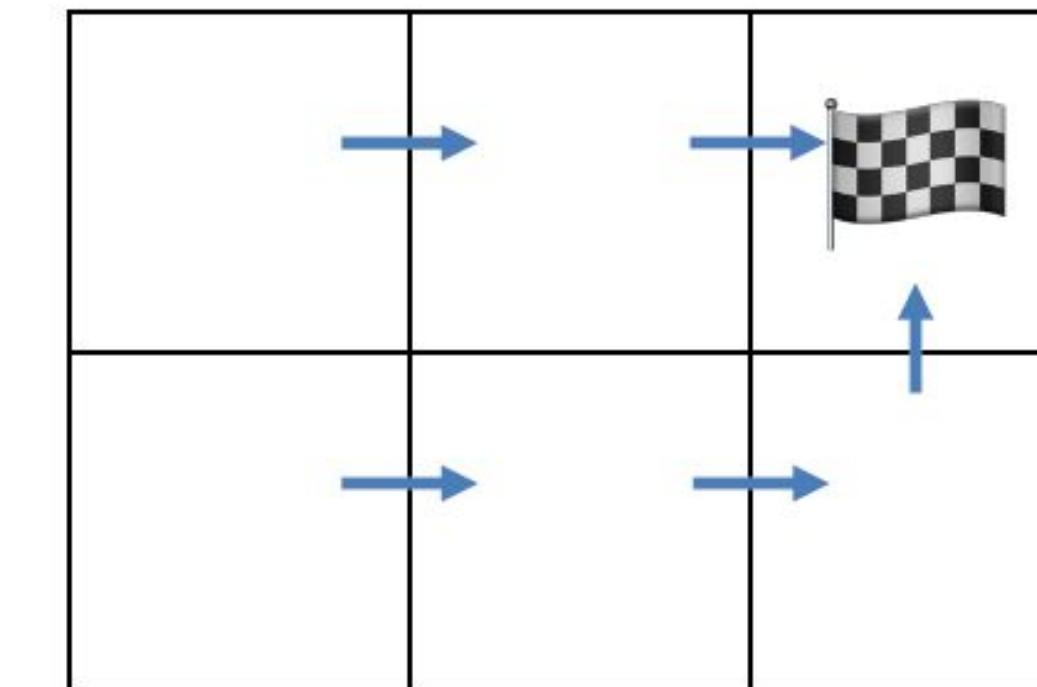
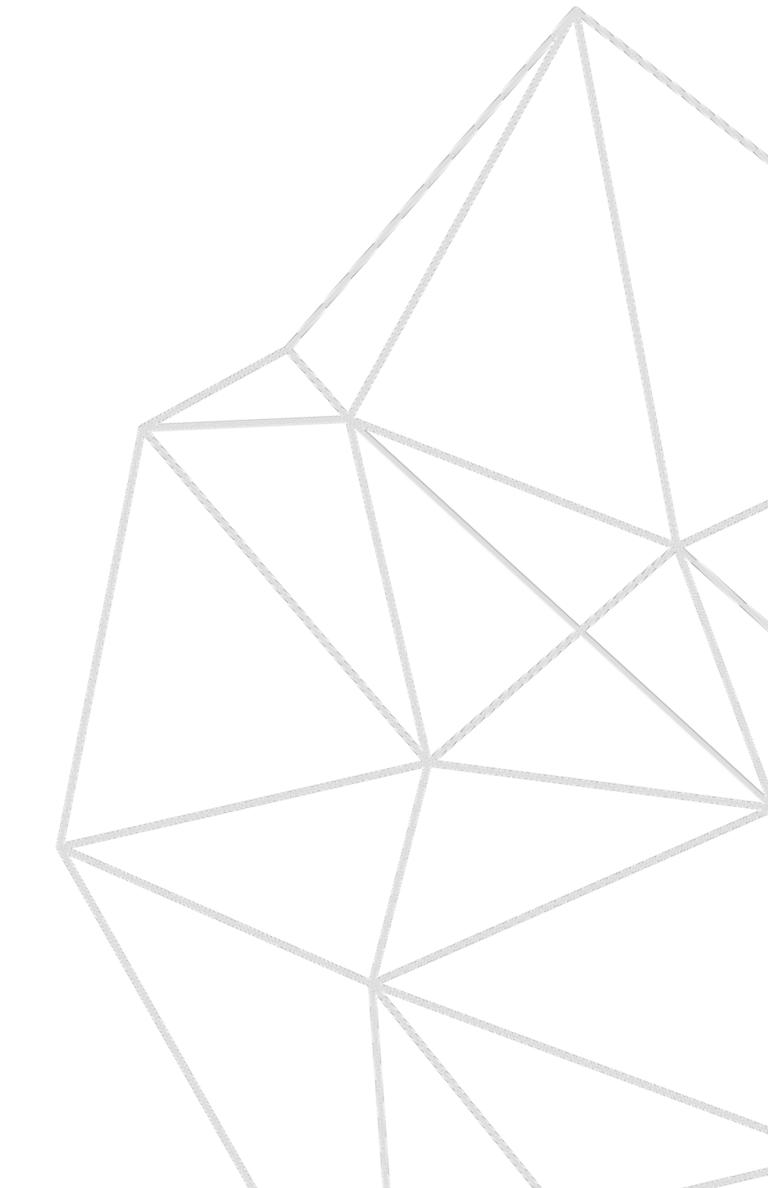
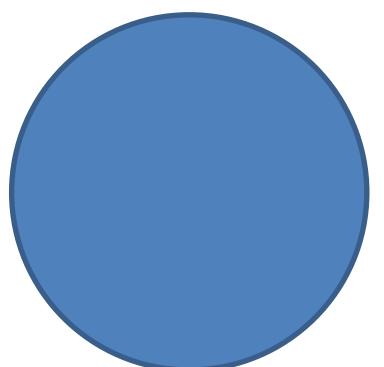
$$\hat{Q}(s_2, a_{right}) \leftarrow r + \gamma \max_{a'} \hat{Q}(s_3, a')$$



$$Q(s, a)$$



Q-Learning: Algoritmo (Ejemplo)

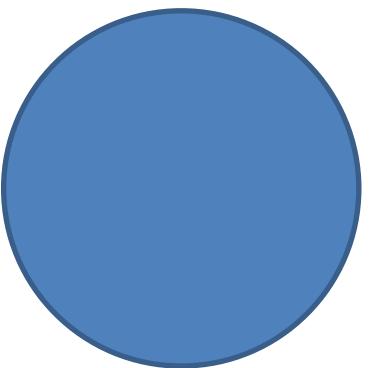
 $r(s_t, a_t)$  $V^*(s), \text{con } \gamma = 0.9$  $Q(s, a)$  π^* 



08

Optimización de Q-Learning

Mejoras de rendimiento, exploración vs. explotación y aplicación del algoritmo a problemas no determinísticos y con espacios de estados demasiado grandes

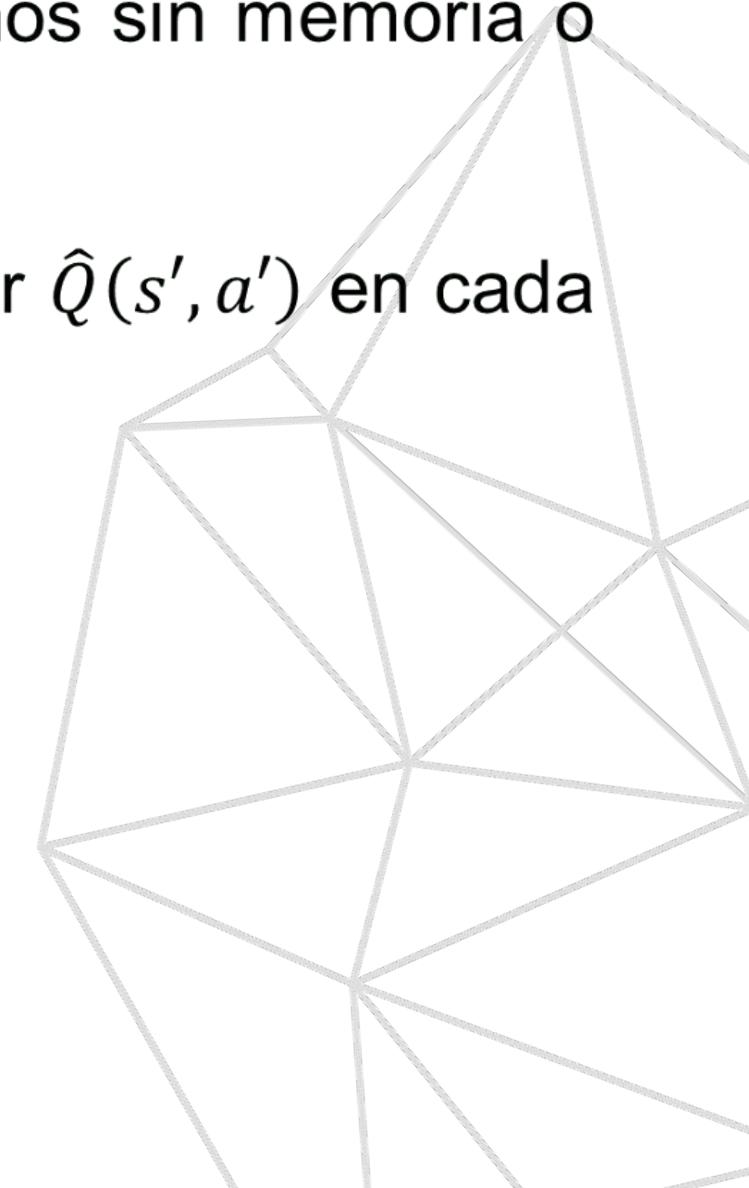
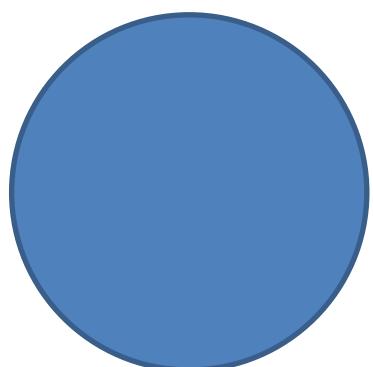




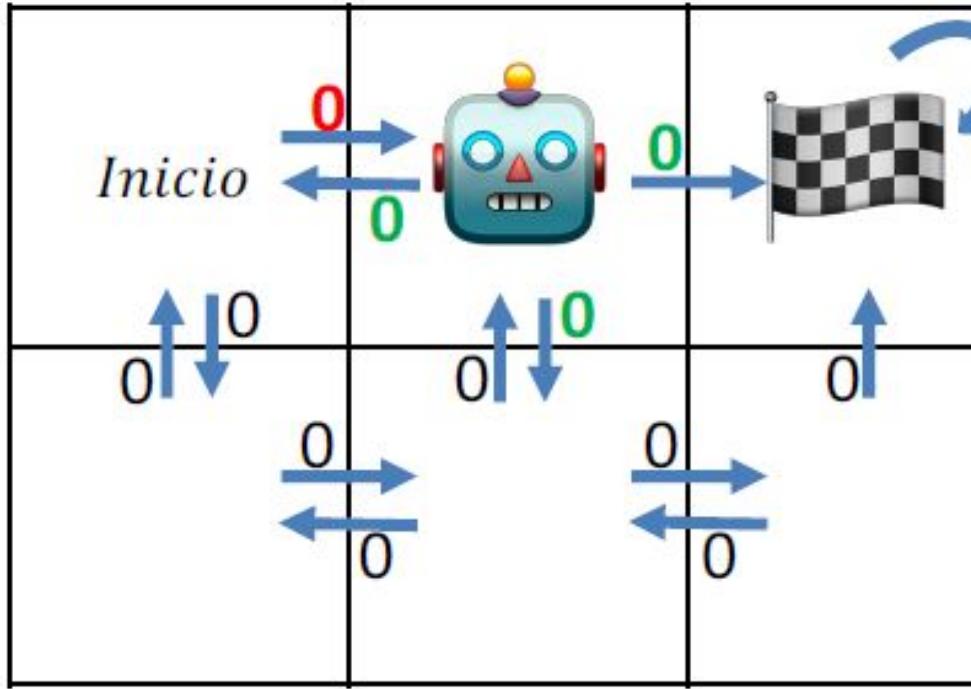
Mejoras del rendimiento

Como vimos en el ejemplo de ejecución del algoritmo de Q-Learning, se pueden realizar dos técnicas para mejorar el rendimiento:

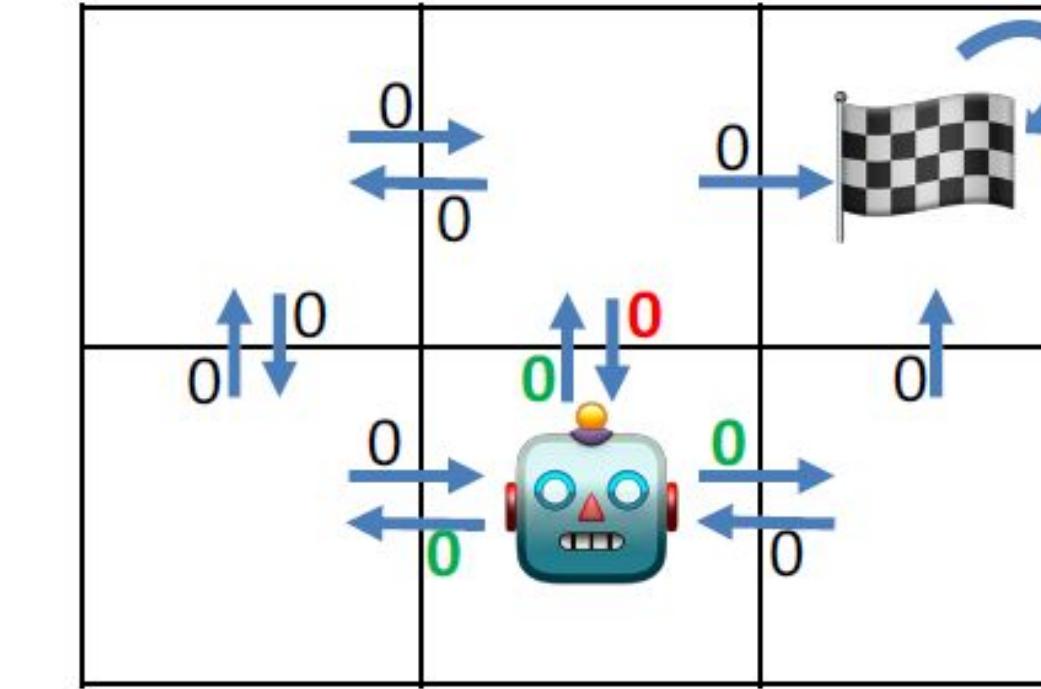
- Ejecución recursiva: antes de actualizar $\hat{Q}(s, a)$ se calcula $\hat{Q}(s', a')$ hasta llegar a un estado final o de recompensa. Se pueden almacenar pares de estado-acción pasados para actualizar sus respectivas Q en estados futuros. El gran inconveniente es que se corre el riesgo de quedarnos sin memoria o llegar al límite de llamadas recursivas.
- Almacenamiento de valores $\hat{Q}(s, a)$ y recompensas inmediatas para evitar recalcular $\hat{Q}(s', a')$ en cada iteración. Es útil en entornos reales ya que se puede simular la ejecución.



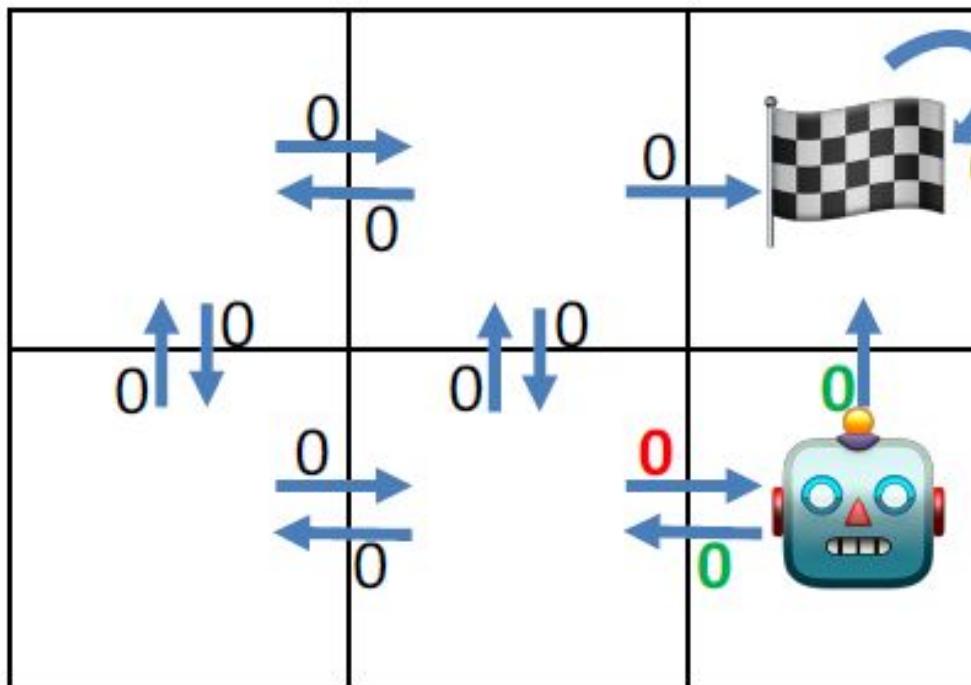
Ejemplo recursivo



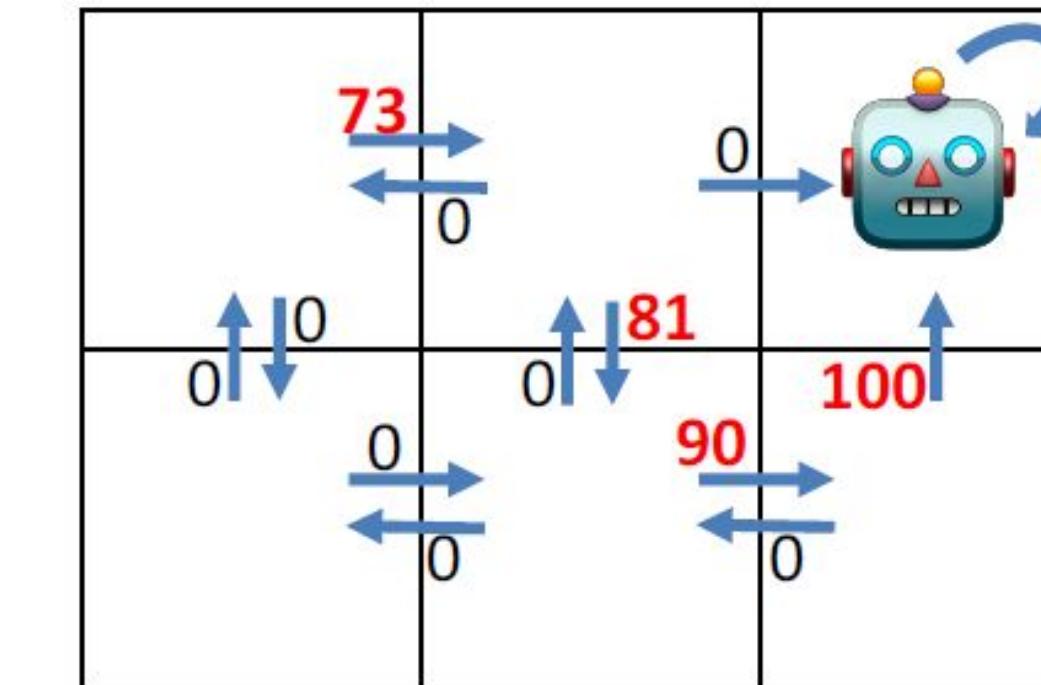
$$\hat{Q}(s_1, a_{right}) \leftarrow r + \gamma \max_{a'} \hat{Q}(s_2, a')$$



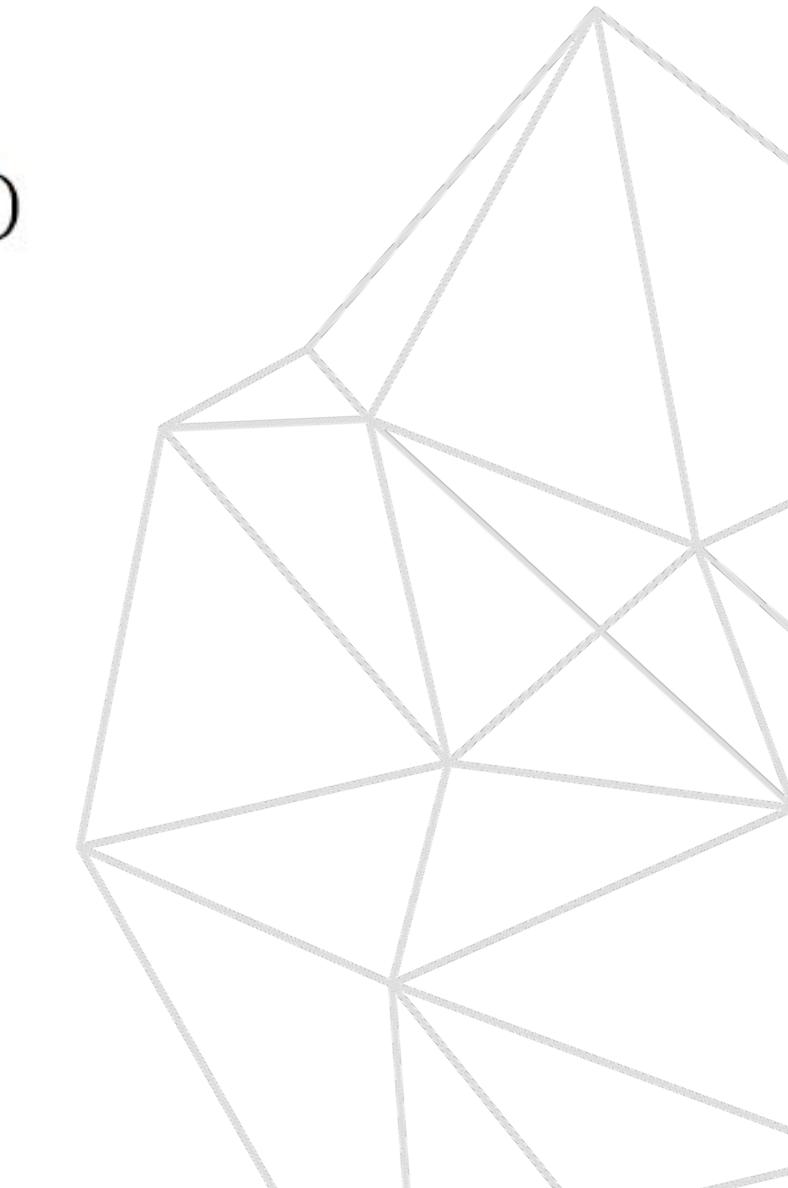
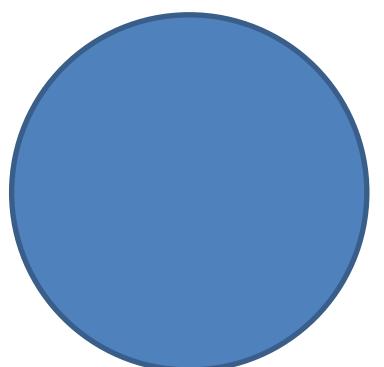
$$\hat{Q}(s_2, a_{down}) \leftarrow r + \gamma \max_{a'} \hat{Q}(s_5, a')$$



$$\hat{Q}(s_5, a_{right}) \leftarrow r + \gamma \max_{a'} \hat{Q}(s_6, a')$$



$$\hat{Q}(s_6, a_{up}) \leftarrow r + \gamma \max_{a'} \hat{Q}(s_3, a')$$





Propiedades

Para todo MDP determinista con recompensas no negativas y valores \hat{Q} inicializados a 0 los valores de \hat{Q} nunca decrecen:

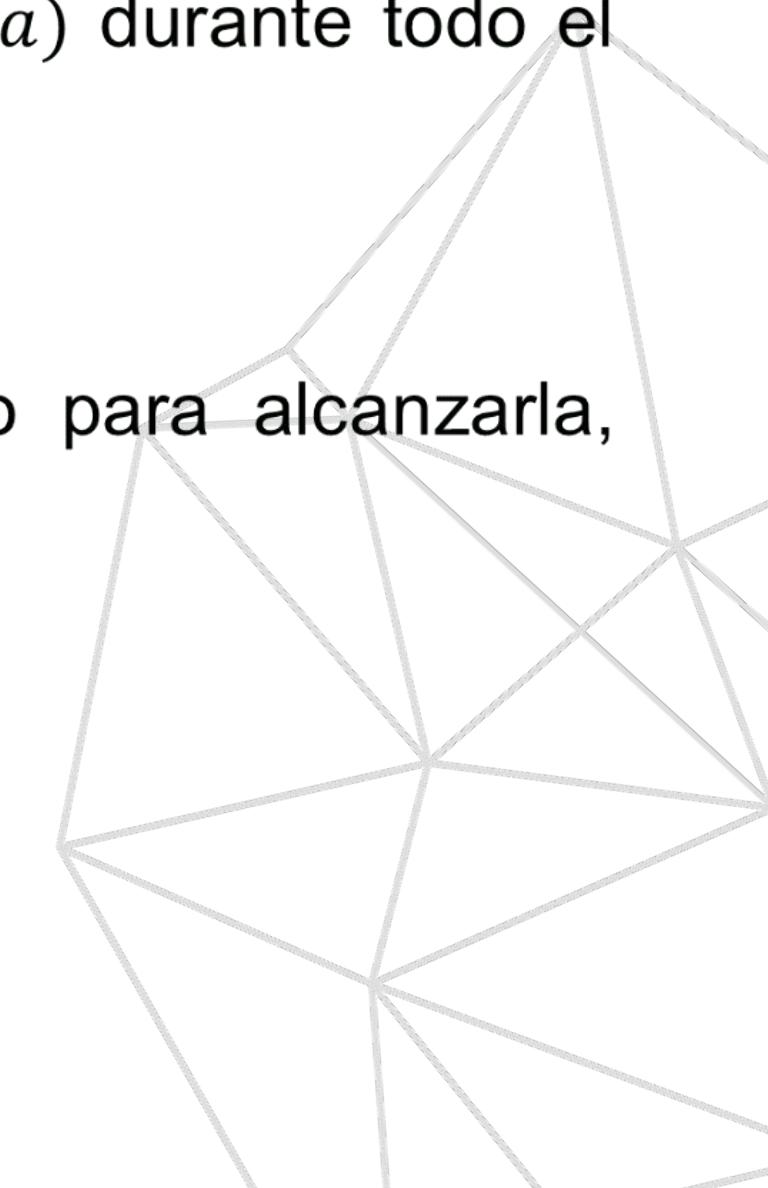
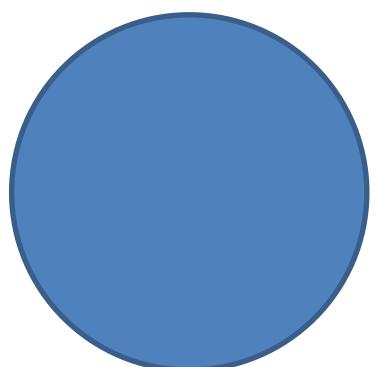
$$\forall(s, a, n) \widehat{Q_{n+1}}(s, a) \geq \widehat{Q_n}(s, a)$$

También se cumple que $\hat{Q}(s, a)$ será mayor o igual que 0 y menor o igual que $Q(s, a)$ durante todo el proceso de aprendizaje:

$$\forall(s, a, n) 0 \leq \widehat{Q_n}(s, a) \leq Q(s, a)$$

Como cabe esperar, siguiendo nuestra definición de la función Q y el algoritmo para alcanzarla, podemos afirmar que $\hat{Q}(s, a)$ converge a $Q(s, a)$ en el infinito:

$$\lim_{n \rightarrow \infty} \max_{s, a} (\widehat{Q_n}(s, a) - Q(s, a)) = 0$$



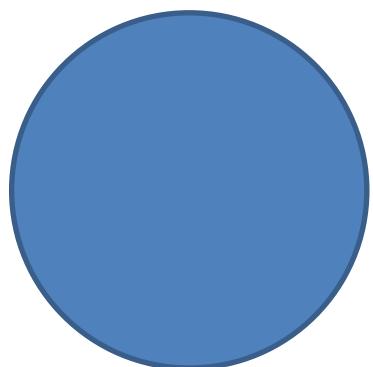
Exploración vs. Explotación

El objetivo de Q-Learning es encontrar una política que permita resolver un problema de forma óptima. La política óptima sigue una estrategia por la cual realiza la acción a que maximice el valor Q para cada estado s . No obstante, cada estrategia es contraproducente durante el proceso de aprendizaje ya que no permite **explorar** nuevas acciones (solo explota). Se debe buscar una estrategia que permita encontrar un equilibrio entre la **explotación** y la **exploración**:

$$P(a_i|s) = \frac{k \widehat{Q}_n(s, a_i)}{\sum_j k \widehat{Q}_n(s, a_j)}$$

Donde $k > 0$ es una constante que determina cuánto se favorece la explotación frente a la exploración:

- Para k grande se favorece la explotación.
- Para k pequeño se favorece la exploración.





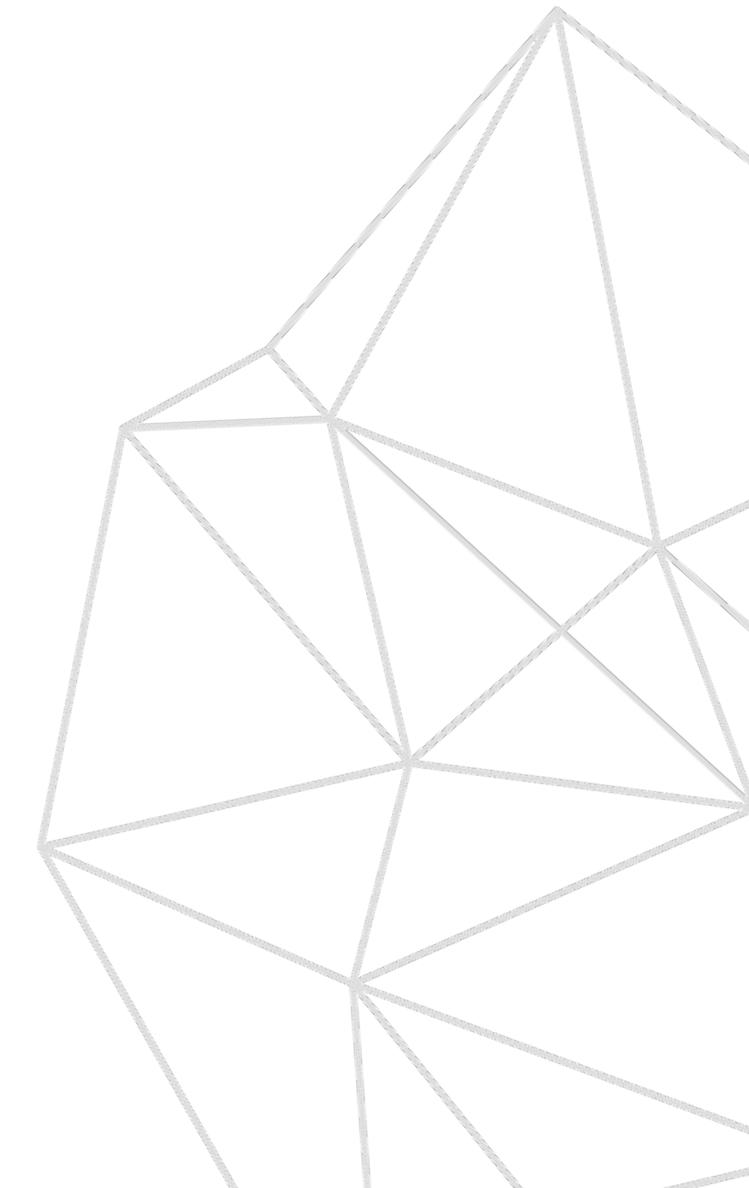
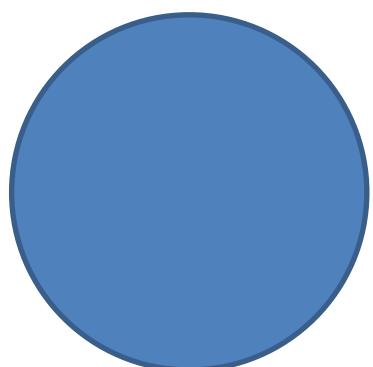
Problemas no determinísticos

Para problemas no determinísticos se aproxima $\widehat{Q}_n(s, a)$ mediante la media ponderada de su valor respecto al número de veces que se ha calculado Q para ese estado-acción:

$$\widehat{Q}_n(s, a) = (1 - \alpha_n)\widehat{Q}_{n-1}(s, a) + \alpha_n \left[r + \gamma \max_{a'} Q_{n-1}(s', a') \right]$$

Donde tenemos que:

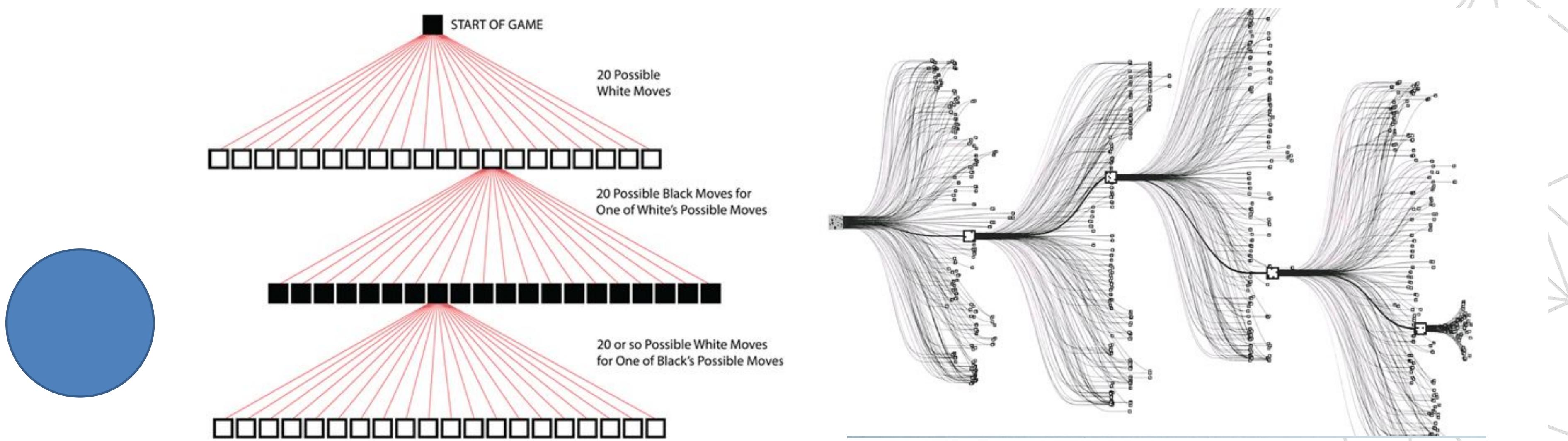
$$\alpha_n = \frac{1}{1 + \text{visitas}_n(s, a)}$$



Problemas continuos o gran dimensión

Es inviable actualizar los valores Q cuando el dominio del estado S es muy grande o infinito (como pueden ser estados con valores continuos o problemas como el ajedrez que poseen una inmensa cantidad de estados).

Para obtener los valores de Q se pueden utilizar redes de neuronas artificiales en las que las entradas son los estados S y la salida de la RNA los valores Q (esta técnica se conoce como Deep Q-Learning).



**Muchas gracias
por vuestra
atención.**

Carlos Moreno Morera
Consultor de IA en IBM
carmor06@ucm.es

