

08MIAR-Aprendizaje por refuerzo

Sesión 2 – Conceptos y Terminología



Universidad
Internacional
de Valencia

De:



Planeta Formación y Universidades

Índice

Vista general

Conceptos básicos

Conceptos avanzados

Clasificación de problemas de aprendizaje por refuerzo

Jerarquía de algoritmos

Conclusiones

Bibliografía recomendada

Índice

Vista general

Conceptos básicos

Conceptos avanzados

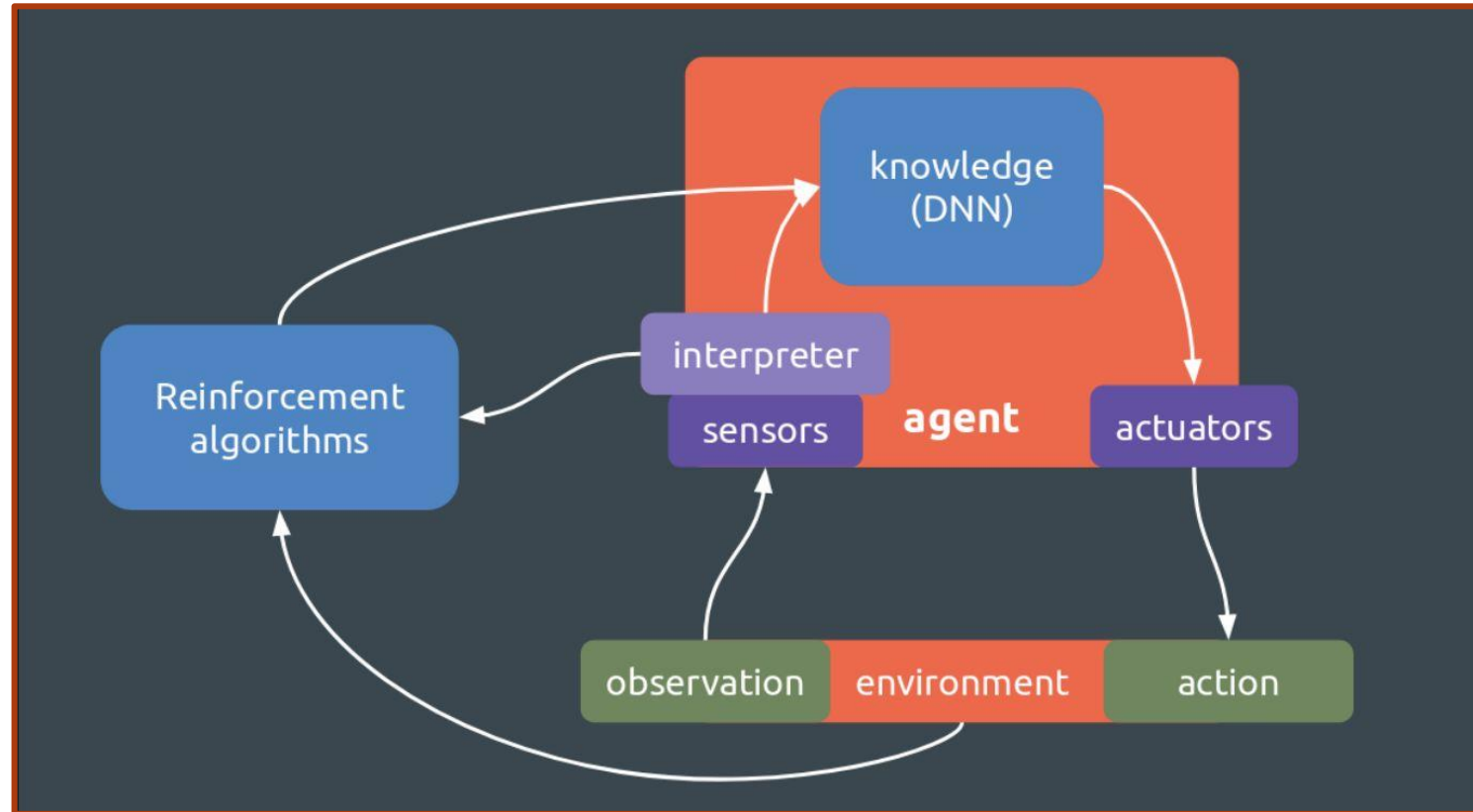
Clasificación de problemas de aprendizaje por refuerzo

Jerarquía de algoritmos

Conclusiones

Bibliografía recomendada

Vista general



Índice

Vista general

Conceptos básicos

Conceptos avanzados

Clasificación de problemas de aprendizaje por refuerzo

Jerarquía de algoritmos

Conclusiones

Bibliografía recomendada

Conceptos básicos

Para entender bien los algoritmos que veremos y cómo podríamos aplicarlos a distintos problemas, necesitamos conocer previamente los conceptos y la terminología que usaremos durante el desarrollo.

En el caso del aprendizaje por refuerzo, la terminología es muy característica y sus conceptos muy cercanos a las matemáticas y estadística que subyace en este tipo de problemas.

En esta sesión entenderemos los conceptos de una manera intuitiva y a alto nivel, así como la representación de cada uno para explotar su uso más adelante.

En las sesiones sobre los algoritmos será donde entraremos más en detalle, desde un punto de vista formal y analítico, sobre los conceptos que veremos a continuación.

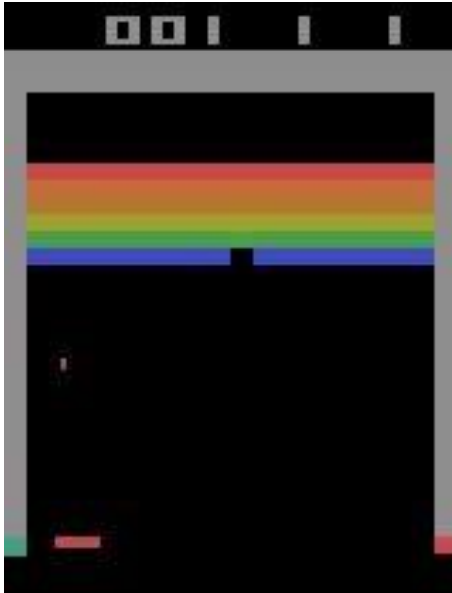
Conceptos básicos

En nuestro caso, los retos en los que trabajaremos estarán basados en videojuegos ya que conectan muy bien con el poder aplicar y probar nuestros algoritmos sobre ellos. Además, los videojuegos están definidos por reglas que rigen cómo se pueden comportar nuestros agentes en los mismos.

Por otro lado, hay que tener en cuenta que para poder demostrar empíricamente que nuestras soluciones *funcionan*, necesitamos controlar en todo momento la simulación, de ahí que los videojuegos sean una buena elección.

Para entender más fácilmente cada uno de estos conceptos vamos a usar como base un entorno digital común para todos, el videojuego de ***Breakout de Atari***.

Entorno



El entorno es el mundo donde el agente vive e interactúa con los elementos de su alrededor.

El entorno no sólo se compone de los elementos visuales de la simulación, también de la lógica que subyace y que define cómo se puntúa, tiempos disponibles, etc.

El entorno es uno de los actores principales en nuestros problemas ya que es la simulación sobre la que nuestro agente aprenderá.

Entorno

Todos los conceptos que abordaremos a alto nivel tienen una relación directa con una parte que también nos interesa, la analítica/tecnológica.

Iremos viendo cómo se representa cada uno de estos conceptos desde un punto de vista más cercano al (*pseudo*)código para poder trabajar con ellos.

En el caso del entorno, normalmente encontraremos un *wrapper* de la simulación. Al ejecutarlo, o al interactuar con el agente, nos irá devolviendo toda la información disponible.

```
# Python code

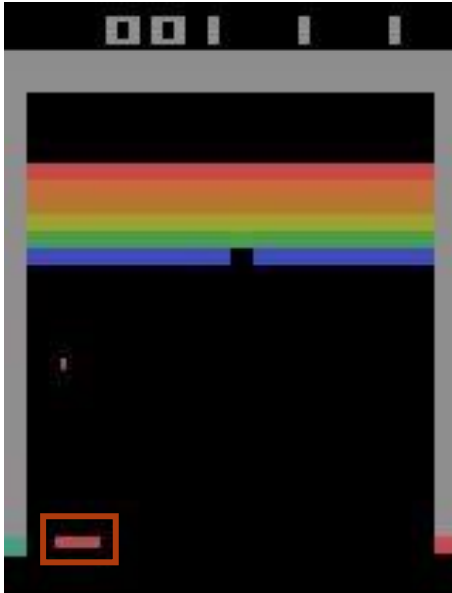
env = wrapper.open("Atari.rom")

env.action_space

env.step(action)

env.internal_variables
```

Agente



El agente es la otra pieza fundamental de nuestra solución.

Un agente es una entidad que interactúa con elementos del entorno (que estén disponibles en el momento de la interacción) para llegar a un objetivo.

Generalmente, este objetivo es maximizar una recompensa (como en el caso de la mayoría de juegos).

El agente será la entidad inteligente que irá aprendiendo durante las ejecuciones del experimento.

Agente

El agente será la pieza de nuestras soluciones en la que recaerá más responsabilidad por nuestra parte.

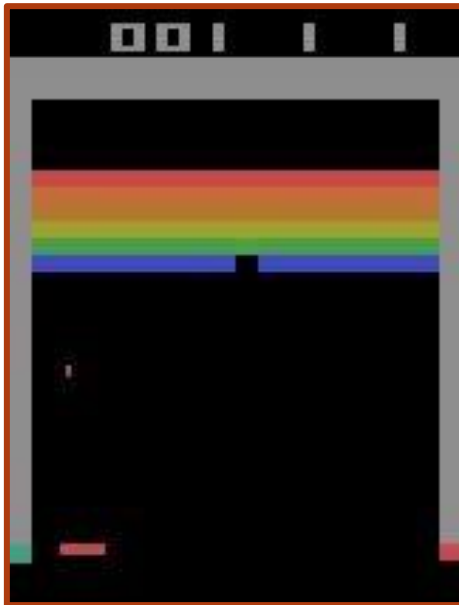
Comúnmente nos encontraremos una clase que sea la que englobe todas las funciones y comportamientos que el agente puede realizar durante su aprendizaje.

Será en el agente donde se relaciona la información que proviene del entorno con el módulo de aprendizaje que hayamos definido (en este curso, una red neuronal).

```
# Python code
```

```
class OurAgent:
    def __init__(...):
        ....
    def select_action(...):
        ...
    def preprocess_obs(...):
        ...
    def update_states(...):
        ...
```

Observación



Una vez tenemos el *feedback-loop* entre nuestro entorno y nuestro agente, ¿cómo se realiza el aprendizaje?

En primer lugar tendremos observaciones. Cada vez que el agente toma una decisión, esa decisión afecta al entorno. Cada *fotografía* del entorno tras una decisión del agente es lo que podemos entender como observación.

Cada observación contiene toda la información disponible en el entorno en ese momento.

Observación

Normalmente, sobre todo en el ámbito de los videojuegos, las observaciones van a tener la forma de un *array* o lista de elementos.

En nuestro reto, cada uno de estos elementos será un píxel de la pantalla.

Dependiendo de la información disponible, podemos encontrar información extra en estructuras de datos o en otros campos que el entorno nos devuelve.

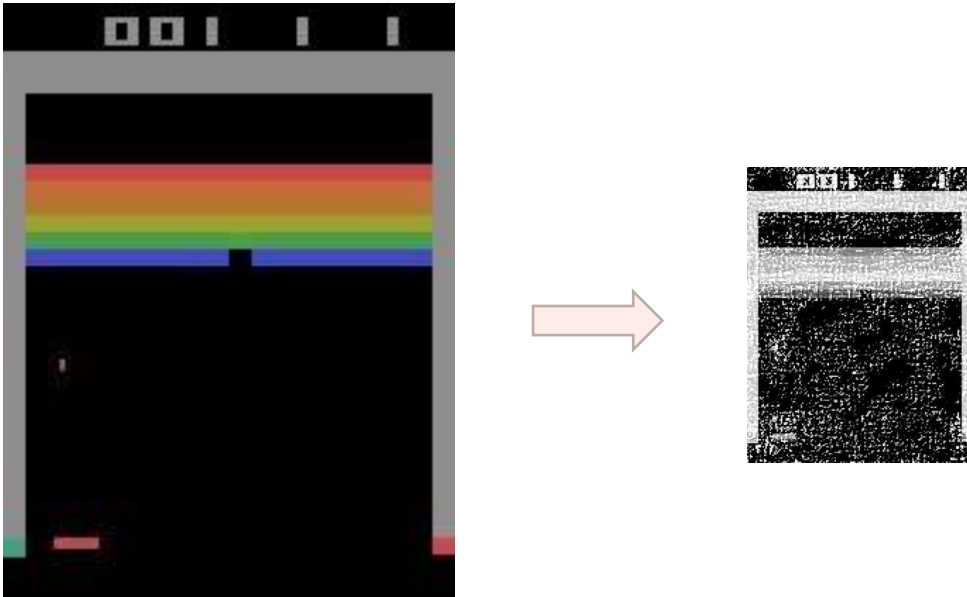
```
# Python code
```

```
obs, info, reward, done = env.step(...)
```

```
# obs
```

```
[[[255, 255, 255, 255],  
  [120, 123, 122, 123],  
  ...  
  [110, 110, 115, 115],  
  [255, 255, 255, 255]]]
```

Estado



Cuando el **entorno nos devuelve la observación** en un momento determinado de la ejecución, la información que el agente obtiene es *raw*, es decir, es tal y como el entorno la genera.

Dependiendo de cómo se defina el módulo de aprendizaje del agente, y de la disponibilidad computacional que tengamos, esta observación necesita un tratamiento.

Esto es lo que se conoce como estado. **El estado es la observación lista para ser consumida por nuestro agente y poder tomar decisiones.**

Otra posible interpretación del concepto de estado es la relación directa con la definición formal mediante cadenas de Markov.

Estado

En el caso del estado, partimos de **la información que tenemos a partir de la observación**.

A esta observación le aplicamos todo lo necesario en cuanto a **pre-procesamiento, redimensionado** de los datos, etc.

Una vez aplicado este pre-procesamiento, la nueva estructura de datos estaría lista para que el agente la pudiera usar adecuadamente.

```
# Python code
```

```
state = resize(obs)  
state = preprocess_and_rgb(State)
```

```
# state  
[[[1.0, 1.0, 1.0, 1.0],  
  [0.5, 0.5, 0.5, 0.5],  
  ...  
  [0.3, 0.4, 0.3, 0.3],  
  [1.0, 1.0, 1.0, 1.0]]]
```

Acciones



Toda **decisión** que hemos ido comentando que el agente va a realizar en el entorno es lo que se conoce como acción.

El agente siempre tiene a su disposición una lista de acciones disponibles, que puede ir cambiando conforme la simulación vaya evolucionando.

.

```
# Python code
```

```
action = agent.select_action(state)
```

```
# action
```

```
[0, 0, 0, 1] # Move left
```

```
env.step(action)
```


Acciones

En el caso del estado, **partimos de la información que tenemos a partir de la observación.**

A esta observación le aplicamos todo lo necesario en cuanto a **pre-procesamiento, redimensionado** de los datos, etc.

Una vez aplicado este pre-procesamiento, la nueva estructura de datos estaría lista para que el agente la pudiera usar adecuadamente.

```
# Python code
```

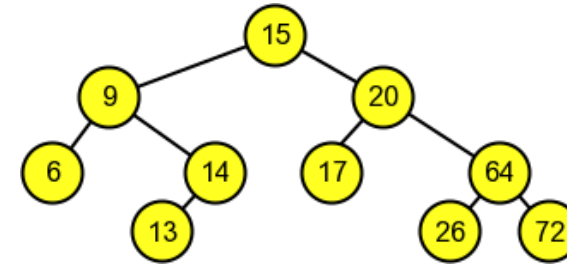
```
action = agent.select_action(state)
```

```
# action
```

```
[0, 0, 0, 1] # Move left
```

```
env.step(action)
```

Espacio de estados y acciones



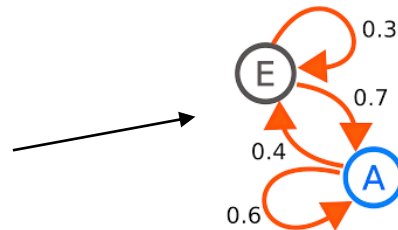
Tanto las acciones como los estados tienen características matemáticas muy interesantes. **Al conjunto de todos los estados y acciones disponibles se le conoce como espacio.**

Son estos espacios los que producen que el aprendizaje por refuerzo tenga un fuerte componente de **optimización** y que, dependiendo del problema, una búsqueda óptima en estos espacios es la parte crucial de nuestras soluciones.

Matemáticamente, el espacio de estados se puede ver como una **cadena de Markov (MDP)**, en el que cada estado va cambiando en el tiempo.

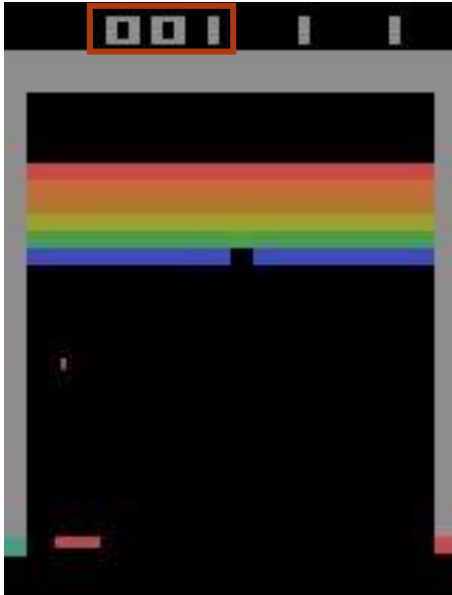
En el caso de las acciones, la búsqueda óptima está más relacionada con técnicas meta-heurísticas, *back-tracking* o búsqueda en árbol. Estas decisiones también son muy dependientes del problema que estemos modelando.

estocasticidad
aleatoriedad



MDP: en t tienes toda la información necesaria para pasar a estimar la acción en $t+1$.

Recompensa



Una vez que el agente toma una acción, ¿cómo sabe si la acción que ha tomado es buena o mala?

Para guiar el aprendizaje y analizar qué acciones son buenas o malas usamos la recompensa. La recompensa es el *feedback* que nos devuelve el entorno para evaluar cómo lo está haciendo el agente.

Por ejemplo, si jugamos a algún videojuego, la recompensa sería la puntuación del mismo. Pero también tenemos otras alternativas como: medir la barra de vida, cuántos *continues* nos quedan, velocidad a la que jugamos, etc.

Recompensa

La recompensa es un valor que nos devuelve el entorno. Dependiendo del entorno en el que estemos trabajando, este valor puede variar aunque siempre intenta ser fiel al comportamiento básico de la simulación.

El punto de complejidad respecto a la recompensa se produce cuando definimos nuestras propias funciones. Por ejemplo, dependiendo del experimento nos puede interesar potenciar ciertas acciones o incluso usar también un castigo para puntuar situaciones adversas.

```
# Python code
```

```
obs, info, reward, done =  
env.step(...)
```

```
# reward  
0.8
```

```
# Define a reward taking into  
account  
# the game counter  
def my_reward(...):  
    ...
```

Iteración (Step)

Cuando decíamos que la observación era una *fotografía* del entorno en un momento determinado, esa definición de momento es lo que se conoce como iteración o *step*.

La iteración se produce cuando congelamos todos los componentes de nuestra ejecución en un instante de tiempo.

Este componente también es muy útil para estimar la ejecución y bondad de ciertos algoritmos ya que será el valor con el que mediremos tiempos y velocidad de convergencia.

Episodio

Por último, tenemos el concepto de **episodio**. El **episodio se desarrolla desde el comienzo de una ejecución hasta que llegamos a *game over***.

- Se acaba la simulación
- Se fija un numero T máximo de iteraciones en el episodio.

Dependiendo del entorno podemos encontrarnos con que el episodio se corresponda con una fase o reto, o que se desarrolle en el tiempo hasta que termine de alguna manera.

El episodio está compuesto de un conjunto de iteraciones.

Episodio e iteración

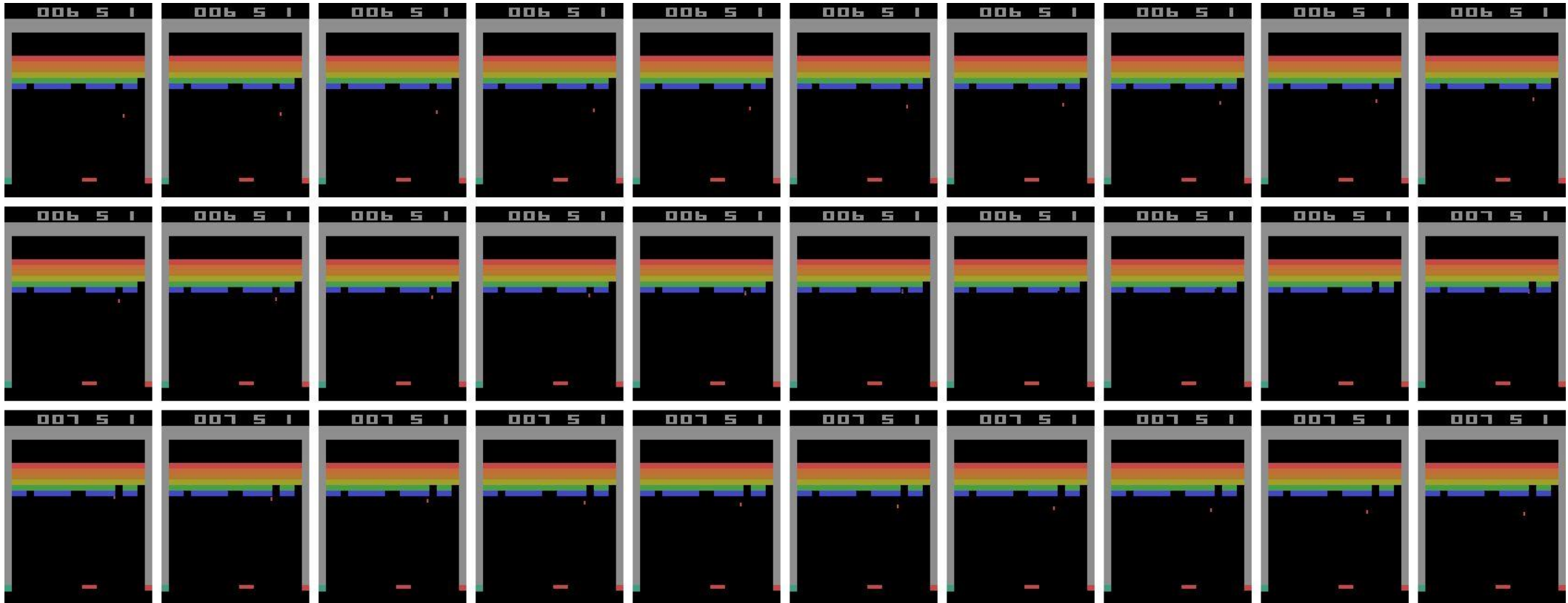
Si queremos interpretar las iteraciones en nuestro código, nos encontramos con una variable para controlar el número de ejecuciones de nuestra simulación.

Los **episodios** están un nivel por encima para definir el **número de pasadas globales (o partidas) que nuestro agente va a realizar**.

Ambos conceptos los podemos ver como **dos bucles anidados** para controlar desde lo más atómico hasta el nivel superior. Normalmente, el bucle que controlamos es el de los episodios, siendo el bucle de las iteraciones interno en la propia simulación.

```
# Bucle de episodios
for i_episode in episodes:
    step=0
    # Bucle de iteraciones
    while run:
        step += 1
        ...
        ...
    if game_over:
        run=False
    if step <=T:
        run=False
```

Episodio e iteración



Índice

Vista general

Conceptos básicos

Conceptos avanzados

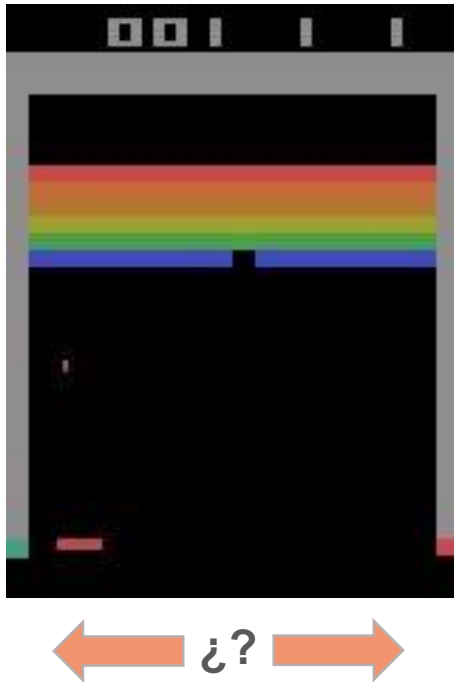
Clasificación de problemas de aprendizaje por refuerzo

Jerarquía de algoritmos

Conclusiones

Bibliografía recomendada

Estrategia (Policy)



Cuando hablamos de que el agente aprende sobre un entorno, en realidad nos referimos a que el agente encuentra una estrategia para maximizar un objetivo.

Esta **estrategia** será la que nos diga **qué acción tomar en un estado**. Nuestra meta es que nuestro agente aprenda la estrategia óptima.

La búsqueda de la mejor estrategia posible es la base del aprendizaje por refuerzo.

En inglés se usa el término *policy* por lo que iremos intercambiando ambos términos durante nuestro curso.

Estrategia (Policy)

Desde un punto de vista programático, la **estrategia** es el concepto ligado a nuestro modelo de Deep Learning.

Va a ser nuestro modelo el que se encargue de **modelar las acciones que se toman dependiendo de los estados en los que el agente se encuentra.**

Dependiendo del algoritmo podemos tener distintos comportamientos en la estrategia.

```
# Python code
```

```
action = agent_model(state)
```

```
# Q models
```

```
action_selected = np.argmax(action)
```

```
# PG models
```

```
action_selected =  
    distribution(1, prob=action)
```

Transiciones

En cada **paso de un estado a otro** se produce lo que llamamos transición. Las transiciones están compuestas de una tupla de elementos:

[Estado, Siguiendo estado, recompensa, acción] \longrightarrow (s,a,r,s)

Con esta **tupla mínima podemos modelar el paso de un estado a otro con información suficiente como para que el agente vaya aprendiendo**. Dependiendo del algoritmo usado, esta tupla puede contener otros elementos que añadan información útil al agente.

Transiciones

Lo más importante para las transiciones es seleccionar una estructura de datos adecuada.

Podemos seleccionar desde la más simple, una **lista de elementos**, a usar colas de mensajería que se encarguen de procesar y gestionar todas las transiciones.

Lo más importante es mantener la misma estructura durante todas las ejecuciones, para que no afecte desde un punto de vista computacional.

```
# Python code
```

```
transition = (state, next_state,  
              reward, action)
```

```
# Lists
```

```
transitions.append(transition)
```

```
# Queue
```

```
transitions = Queue()
```

```
transitions.push(transition)
```

Experiencia


Dependiendo del algoritmo que usemos nos interesa extraer un tipo de información u otra.

Esto es lo que se conoce como experiencia, porque va directamente relacionado con la información pasada que el agente tiene a su disposición para ir aprendiendo.


En su forma más básica se almacena la información de las transiciones, pero dependiendo del algoritmo esa información puede variar.

Experiencia = [Transiciones + Extras] en el intervalo [T-t]

(s, a, r, s)



$(a(t+1), Q\text{-values}, \dots ,)$



Experiencia

Al igual que con las **transiciones**, la experiencia también necesita de una estructura de datos adecuada.

En este caso, es común usar **colas de un tamaño fijo** ya que, como veremos en los algoritmos, la experiencia está relacionada con un conjunto fijo de número de transiciones para ir entrenando a nuestro agente.

```
# Python code
```

```
class Experience():
    def __init__():
        # Queue with 120 elements
        self.data = Queue(120)

    def add(elem):
        self.data.push(elem)

    def get():
        self.data.pop()

transition = (state, next_state,
              reward, action)
experience = Experience()
Experience.data.add(transition)
```

Exploración



EXPLORATION

Playing the other machines to see if any pay out more.

Proceso por el que el agente va adquiriendo experiencia a partir de prueba-y-error en el entorno.

Se puede **explorar de forma aleatoria al comienzo de una simulación**, para empezar el almacenamiento de las transiciones y así comenzar el aprendizaje.

Durante entrenamiento, la exploración está controlada por una **variable aleatoria que va disminuyendo en el tiempo**. Dependiendo del valor de esta variable vamos decidiendo si la acción que toma el agente es aleatoria o no.

Exploración

Como hemos indicado, el proceso de **exploración** está controlado por una **variable externa** para tomar una acción de una manera **aleatoria** o siguiendo la estrategia que se está aprendiendo.

Los valores típicos para esta variable van desde 0.99 (acción siempre aleatoria) hasta 0.05 (acción la mayoría de veces usando la *policy* aprendida).

Justo durante **el intervalo de tiempo entre estos dos valores será nuestro tiempo de exploración**.

```
# Python code

epsilon = 0.99

# During the simulations
(...)

random_number = np.random(...)

if random_number > epsilon:
    # policy action
else:
    # random action

epsilon -= 0.01
```

Explotación



EXPLOITATION

Playing the machine that (currently) pays out the most.

Una vez que el agente ha **explorado un tiempo suficiente** (esto es totalmente dependiente del problema), comienza el proceso de **explotación**.

En esta parte, **el agente ha aprendido cómo tomar decisiones a partir del proceso de aprendizaje**, sin necesidad de explorar situaciones nuevas.

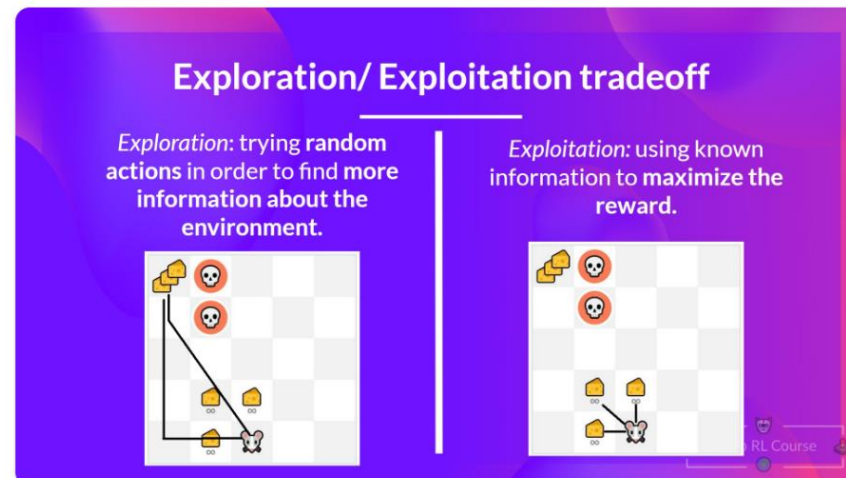
Es común **dejar siempre un grado de aleatoriedad** para la decisión de qué acción tomar, aunque esa probabilidad es muy pequeña.

Explotación

Como hemos visto en la diapositiva anterior, en **explotación** consideramos que el agente ha **aprendido lo suficiente**, por lo que las **acciones** que se seleccionan las obtenemos **directamente de nuestra *policy***.

Para ello obtenemos las acciones directamente usando nuestro modelo, que es el encargado de aproximar la **mejor *policy* aprendida**.

Este estado es la situación típica cuando desplegamos nuestra solución.



Índice

Vista general

Conceptos básicos

Conceptos avanzados

Clasificación de problemas de aprendizaje por refuerzo

Jerarquía de algoritmos

Conclusiones

Bibliografía recomendada

Clasificación de problemas de aprendizaje por refuerzo

En la primera sesión vimos distintos enfoques a la hora de clasificar los problemas y retos que nos podemos encontrar cuando queremos aplicar técnicas de aprendizaje por refuerzo. La mayoría de estos enfoques estaban definidos a alto nivel.

Si bajamos a detalle, podemos analizar los problemas de aprendizaje por refuerzo desde el punto de vista del comportamiento del agente, de la información que tenemos a nuestra disposición, etc. Este análisis nos permite saber qué algoritmo es el más adecuado acorde al reto que queramos solucionar.

En las siguientes diapositivas nos centraremos en dos clasificaciones básicas:

- **Basada en modelo - ¿Tenemos un modelo de nuestro entorno?**
- **Basada en estrategia - ¿Cómo gestionamos la memoria?**

Hay más, pero para nuestro objetivo estas dos son las más importantes.

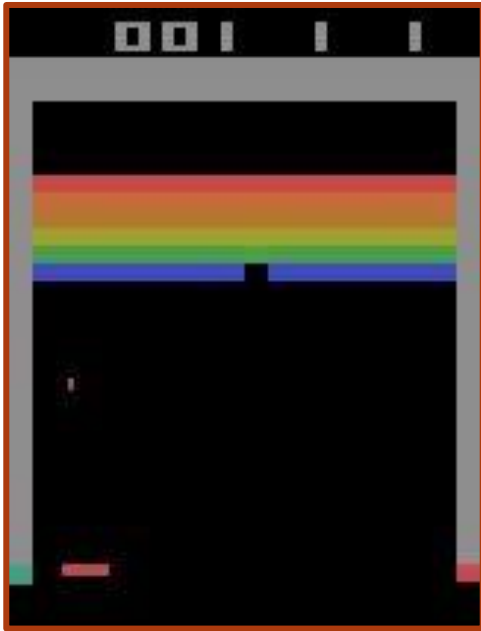
Basados en modelo

Cuando nos referimos a “algoritmos de aprendizaje por refuerzo **basados en modelo**” podemos encontrar dos posibilidades: ***model free*** y ***model based***.

En ***model free*** la simulación no conoce un modelo para “predecir” siguientes estados en el entorno. Es el caso típico que encontramos en la mayoría de los ejemplos hoy en día, como es el caso en casi todos los videojuegos. **El comportamiento de los entornos en este tipo de simulaciones no es determinista**, de ahí que no se pueda conocer un modelo.

Al no poder anticipar siguientes estados, la estrategia que busca el agente es dependiente sólo del estado actual en el que se encuentra.

Basados en modelo



model free

Basados en modelo

En el caso de *model* based sí que conocemos un modelo de cómo se comporta el entorno en relación con las posibles acciones que el agente tome durante la ejecución.

Al tener el conocimiento de este modelo, es típico **combinar los algoritmos de aprendizaje con refuerzo con otras soluciones de inteligencia artificial como técnicas de planificación y búsquedas óptimas** con objetivo de mejorar el proceso de aprendizaje.

Por ello, en este tipo de enfoque el agente no sólo se encarga de aprender la estrategia óptima sino que también esta estrategia va acompañada de algoritmos de optimización para evitar los costes computacionales de hacer búsquedas *por fuerza bruta*.

model based



Basados en estrategia

Una vez vista la primera clasificación, ahora es el turno de la otra opción: por **estrategia**. Evidentemente, cuando hablamos de estrategia nos referimos a nuestra definición de estrategia o *policy*. Al igual que con la clasificación basada en modelos, tendremos dos posibles conjuntos para identificar a los algoritmos de aprendizaje por refuerzo.

El primero de ellos es el que se conoce como ***on policy***. Durante el proceso de aprendizaje, la estrategia que nuestros agentes siguen puede ir cambiando en el tiempo. Con *on policy* nos referimos a **que el agente sólo puede usar la experiencia pasada de una estrategia específica**.

Normalmente este tipo de aprendizaje se caracteriza por más tiempo de entrenamiento para encontrar una solución óptima así como de más varianza en los datos a la hora de entrenar a nuestro agente. En este tipo de aprendizaje encontramos muchos de los algoritmos de la familia de ***Policy gradients***.

Basados en estrategia

Por otro lado encontramos la otra opción para clasificar nuestros algoritmos, una estrategia *off policy*. Este es el caso contrario, **podemos usar experiencia adquirida con estrategias distintas cada vez que queramos hacer una actualización del aprendizaje de nuestro agente.**

Esta situación está muy relacionada con conceptos vistos durante la sesión, por ejemplo, la experiencia. Si ejecutamos nuestra simulación y vamos almacenando la experiencia que el agente va adquiriendo a la vez que el agente va aprendiendo, es normal **que encontremos transiciones que pertenecen a estrategias distintas.**

Este enfoque, como todos, tiene sus pros y sus contras. El punto a resaltar es que aunque podamos usar **más información (heterogenea) pasada** para el aprendizaje, **puede conllevar a más tiempo de convergencia dependiendo de la información almacenada.** Un ejemplo de familia de algoritmos que siguen este tipo de comportamiento son las *Q-networks*.

Índice

Vista general

Conceptos básicos

Conceptos avanzados

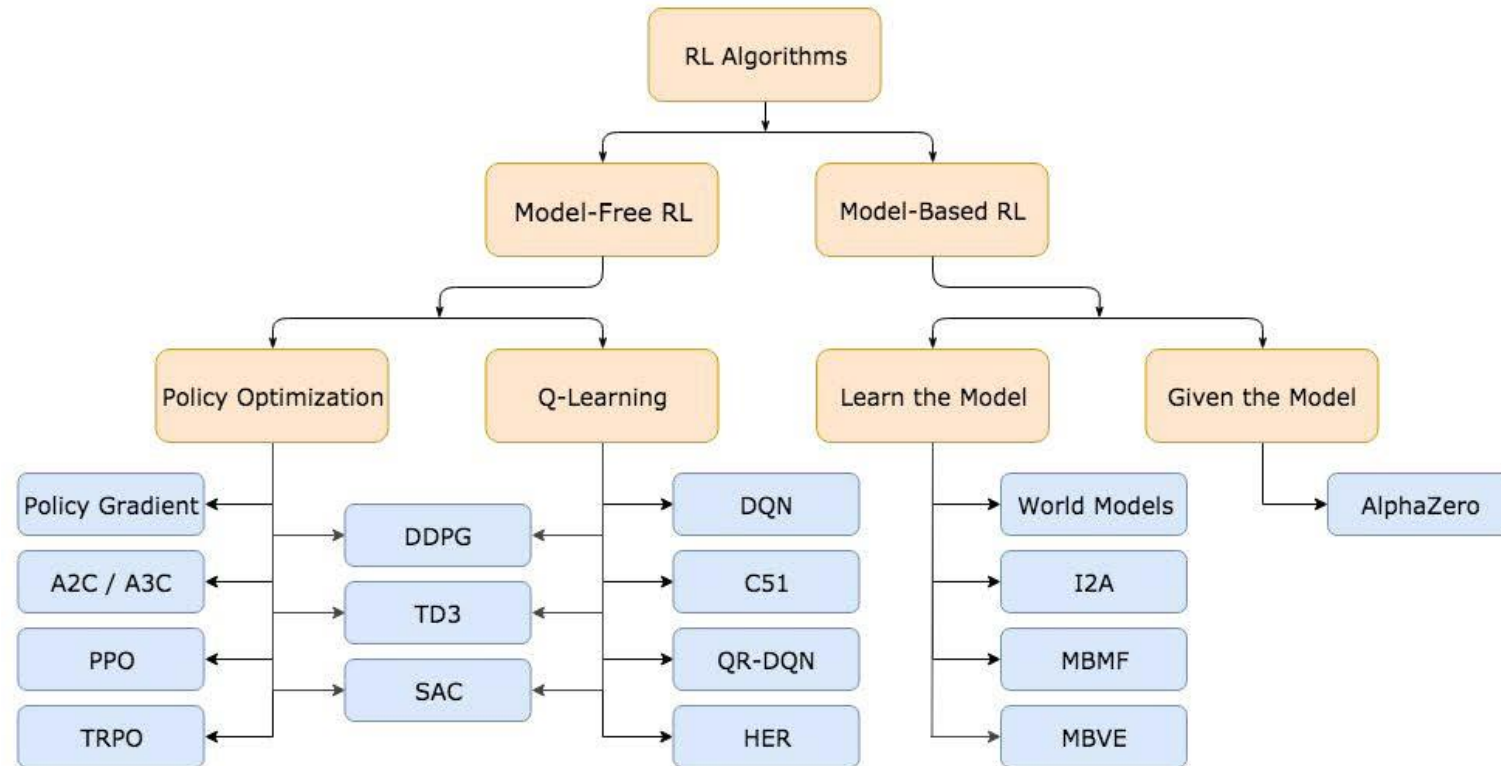
Clasificación de problemas de aprendizaje por refuerzo

Jerarquía de algoritmos

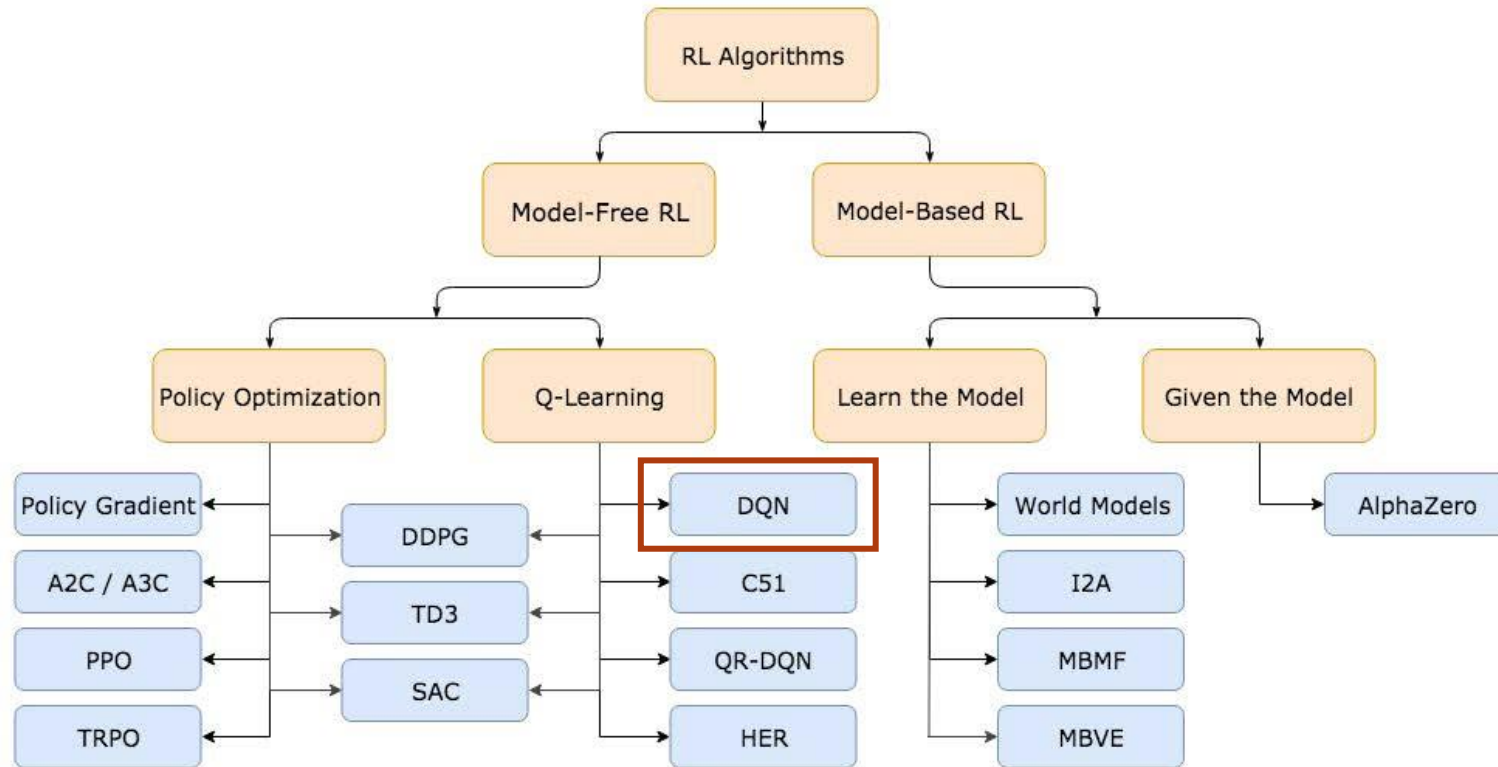
Conclusiones

Bibliografía recomendada

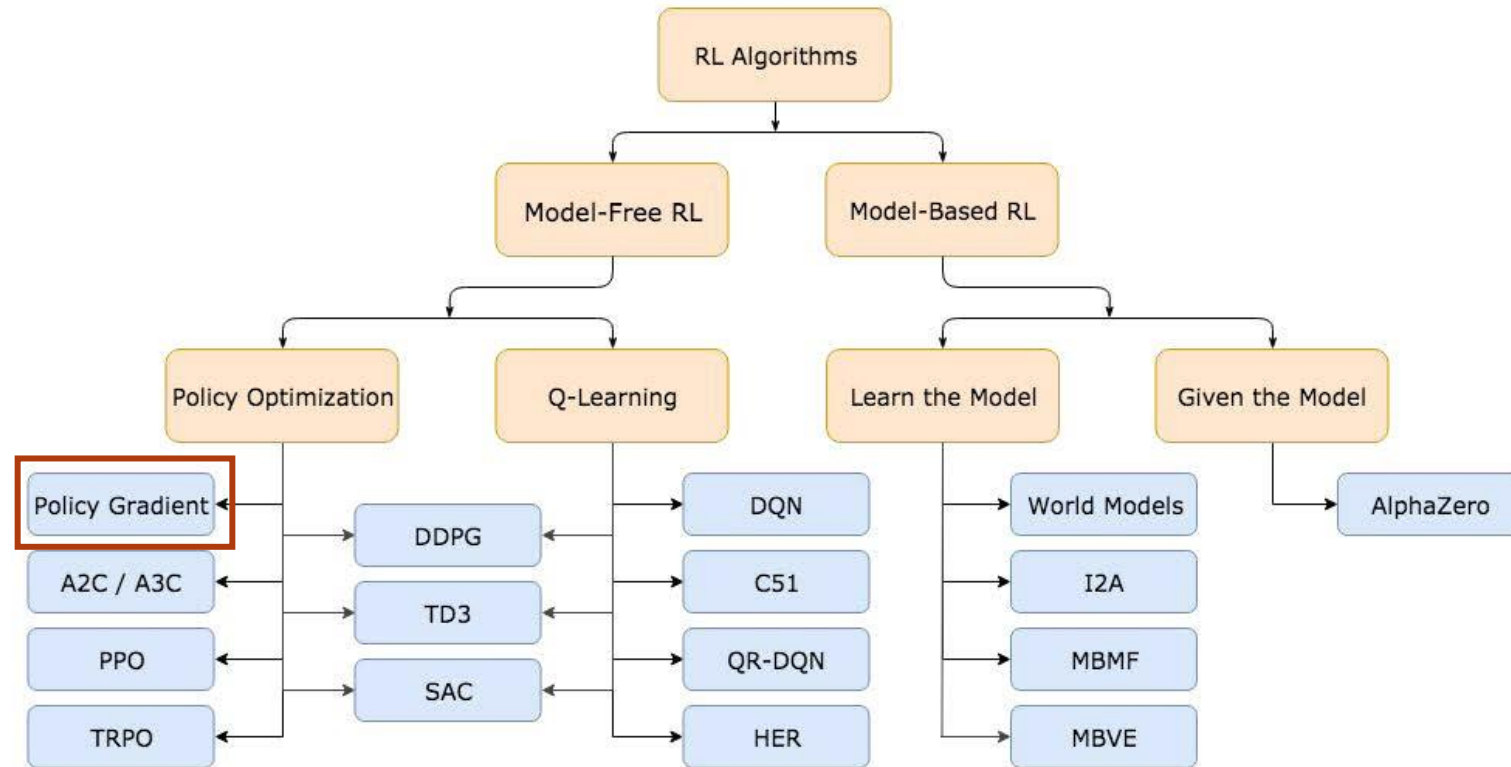
Jerarquía de algoritmos



Jerarquía de algoritmos



Jerarquía de algoritmos



Índice

Vista general

Conceptos básicos

Conceptos avanzados

Clasificación de problemas de aprendizaje por refuerzo

Jerarquía de algoritmos

Conclusiones

Bibliografía recomendada

Conclusiones

- La **terminología** y los conceptos vistos son muy específicos de aprendizaje por refuerzo. Hay que saber diferenciarlos de otras definiciones dentro de la inteligencia artificial o el mundo tecnológico.
- Todos los conceptos están muy relacionados entre sí como por ejemplo los conceptos de Observación y Estado, así como Experiencia y Transición.
- La aplicación de los conceptos vistos es muy flexible dependiendo del algoritmo usado y del reto que se quiera resolver.
- **Las clasificaciones presentadas son fundamentales** para entender qué nos ofrece cada enfoque dentro de todas las posibilidades de las soluciones de aprendizaje por refuerzo.

Índice

Vista general

Conceptos básicos

Conceptos avanzados

Clasificación de problemas de aprendizaje por refuerzo

Jerarquía de algoritmos

Conclusiones

Bibliografía recomendada

Bibliografía recomendada

“Reinforcement Learning: An introduction”, Sutton y Barto:

<http://incompleteideas.net/book/bookdraft2017nov5.pdf>

(Capítulo 2, *Multi Armed Bandits*, donde se trata el dilema de exploración-explotación)

Offline Reinforcement Learning: Tutorial, Review, and Perspectives on Open Problems

Levine S. et al, *<https://arxiv.org/pdf/2005.01643.pdf>*



viu

Universidad
Internacional
de Valencia

universidadviu.com

De:
 Planeta Formación y Universidades