Redes Neuronales y Aprendizaje Profundo

Tema 8. Agentes inteligentes. Deep Reinforcement Learning

Índice

Esquema

8.1. ¿Cómo estudiar este tema?

Para estudiar este tema deberás leer las **Ideas clave** que se desarrollan a continuación.

En este tema veremos otra área que ha sido revolucionada por el aprendizaje profundo. La combinación de *deep learning* y *reinforcement learning* ha dado lugar a nuevos hitos en el mundo de la inteligencia artificial, con agentes capaces de aprender a jugar a juegos viendo directamente imágenes de las partidas o ganando a los humanos en juegos como el Go, considerados hasta hace poco algo fuera del alcance de una IA.

Es importante comprender qué es un problema de aprendizaje por refuerzo, cómo se diferencia de los problemas de aprendizaje supervisado vistos hasta ahora y cómo se define de manera formal mediante un proceso de decisión de Markov. Igualmente, es necesario aprender cómo podemos combinar esto con las redes neuronales profundas para dar lugar al *Deep Q-Learning*, así como entender el funcionamiento de este algoritmo.

8.2. Reinforcement Learning

Hasta ahora hemos estado trabajando principalmente en problemas de **aprendizaje supervisado**. En este tipo de *machine learning*, tenemos una serie de datos x y una salida y que queremos predecir. El objetivo es aprender una función capaz de llevarnos de x a y. En nuestro caso, esta función ha venido dada por redes neuronales. El proceso de entrenamiento ha consistido en utilizar los pares (x,y) conocidos para intentar aprender esa función. Un ejemplo de este tipo de problema es la clasificación de objetos en imágenes, con x siendo la imagen e y siendo el objeto a predecir.

Otro tipo de aprendizaje automático es el **aprendizaje no supervisado**; aquí tenemos solo los datos x, sin valor de salida a predecir. El objetivo es aprender una estructura existente en los datos. Problemas de este tipo incluyen *clustering*, *density estimation*, reducción de dimensionalidad, etc.

En este tema, vamos a tratar un tipo distinto de aprendizaje, el **aprendizaje por refuerzo** o *reinforcement learning*. En este tenemos:

- Un agente interactuando con un entorno (*environment*).
- El agente se encuentra en un estado s y lleva a cabo acciones a.
- Lo cual produce un nuevo estado y una recompensa (*reward*) del entorno.

El objetivo aquí es aprender las acciones a realizar según nuestro estado para maximizar la recompensa.

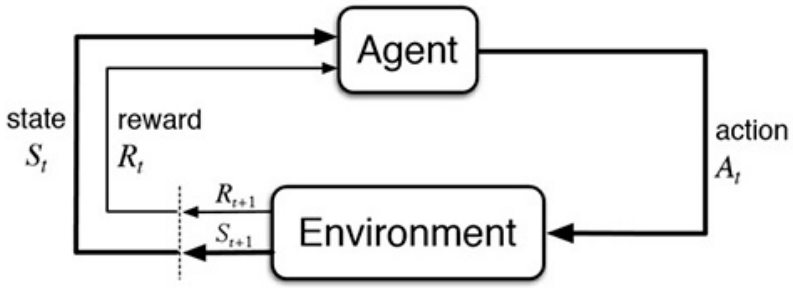


Figura 1. Aprendizaje por refuerzo o *reinforcement learning*.Fuente:
https://www.kdnuggets.com/2018/03/5-things-reinforcement-learning.html

Ejemplos

Ejemplo 1. El problema del *cart-pole* o péndulo invertido: tenemos un poste con la forma de un péndulo invertido encima de un carro en movimiento. El objetivo es mover el carro de modo que el poste no caiga y se mantenga en posición vertical.

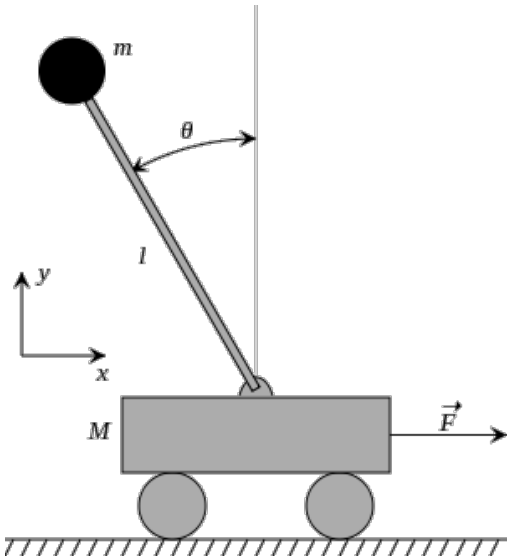


Figura 2. Dibujo esquemático de un péndulo invertido sobre un carro.Fuente:
https://en.wikipedia.org/wiki/Inverted_pendulum

En este problema:

- El **estado** en el entorno viene dado por:
  - El ángulo en el que se encuentra el poste.
  - La velocidad angular que lleva en ese momento.
  - La posición del carro.
  - La velocidad del carro.
- El **agente** puede realizar acciones como aplicar una fuerza al carro en dirección horizontal.
- La **recompensa** o *reward* es 1 por cada instante de tiempo en el que el péndulo esté en estado vertical. De este modo, el agente deberá aprender a, según un estado inicial, mover el carro hacia los lados de modo que el péndulo se estabilice.

Ejemplo 2. El movimiento de robots (*robot locomotion*). En este tipo de problemas, se define un entorno físico por medio de un simulador en el que unos robots-agentes con distintas formas (humanoides o arañas, por ejemplo) tienen que aprender a moverse hacia delante manteniéndose en equilibrio y esquivando ciertos obstáculos.

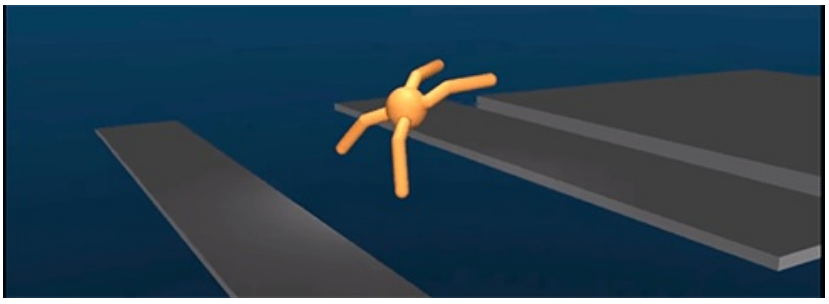


Figura 3. Robot araña aprendiendo a saltar.Fuente: https://en.wikipedia.org/wiki/Inverted_pendulum

En este caso:

- El estado del entorno viene dado por una posición en el plano físico y por la posición y movimiento de cada una de las articulaciones que definen al agente.
- Las acciones se corresponden con movimientos posibles de estas articulaciones para moverse.
- El agente es recompensado por moverse hacia delante. De este modo, este aprende a utilizar sus articulaciones para caminar y esquivar obstáculos.

Ejemplo 3. Otro ejemplo de *reinforcement learning* aplicado a tareas reales está en los videojuegos. Un agente inteligente puede aprender a jugar a videojuegos clásicos como Doom directamente a partir de las imágenes del juego. Esto es radicalmente diferente a las IA que normalmente vemos programadas en los videojuegos.

En este caso, podemos definir un problema de aprendizaje por refuerzo donde:

- El estado viene dado por lo que un jugador ve.
- Las acciones son el movimiento en todas las direcciones y disparar
- El agente es recompensado cuando elimina a un oponente y es castigado (recompensa negativa) cuando muere.

Figura 4. Imagen del videojuego Doom.Fuente: https://itunes.apple.com/us/app/doom-classic/id336347946?mt=8

Ejemplo 4. Finalmente, un ejemplo más orientado a la industria sería un robot para cadenas de montaje como podría ser un robot encargado de meter objetos en una caja.

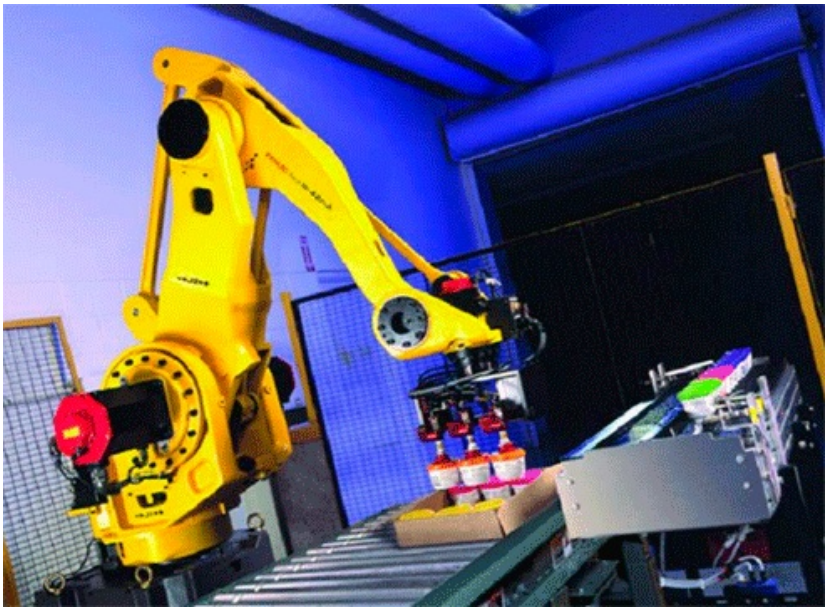


Figura 5. Robot de una cadena de montaje..Fuente: https://youtu.be/MQ6pP65o7QM

8.3. Procesos de decisión de Markov

Los procesos de decisión de Markov, *Markov Decision Processes* (MDP) en inglés, son la formulación matemática de los problemas de aprendizaje por refuerzo o *reinforcement learning*. Estos procesos se definen mediante una tupla:

$$(S,A,P_{sa},\gamma,R)$$

Donde:

- S es el conjunto de **estados**.
- A es el conjunto de **acciones**.
- P_{sa} son las probabilidades de **transiciones** entre estados. Para cada estado s y acción a, P_{sa} es una distribución de probabilidad sobre los posibles estados destino. Básicamente, nos da las probabilidades de a qué estado iremos si tomamos la acción a en el estado s.
- es el ***discount factor***. Es un número entre 0 y 1 y representa la importancia de obtener *rewards* lo más pronto posible.
- R es la ***reward function***, y modela las recompensas dadas a partir de un estado y acción.

Este modelo matemático es suficientemente genérico como para permitir transiciones probabilísticas entre estados, definidas por P_{sa}. En muchos problemas, sin embargo, las transiciones son deterministas y se corresponden con unas probabilidades de 1 por cada estado y acción. Podemos pensar que este es el caso si resulta complicado pensar en un mundo de probabilidades.

Ideas clave

Los procesos de Markov siguen la **propiedad de Markov**, que dice que el estado actual caracteriza completamente el estado del entorno. Es decir, a la hora de realizar una acción a_t en un estado s_t , la transición al nuevo estado al que vamos es independiente de los estados anteriores en los que podía encontrarse el mundo.

Un proceso de Markov funciona de la siguiente manera. En el *time step* inicial (t_0) empezamos en un estado inicial (s_0). A partir de ahí, mientras no lleguemos a un estado considerado terminal:

1. El agente selecciona una acción a_t .
2. El entorno otorga una recompensa r_t basada en el estado actual s_t y la acción a_t .
3. Según la distribución de probabilidad P_{sa} , el entorno obtiene un nuevo estado s_{t+1} para el agente.

El agente elegirá acciones con base en una política π ***(policy π)***. π es por tanto una función que va de los estados S a las acciones A y determina las acciones que el agente tomará en cada estado.

El objetivo es encontrar la política óptima π que maximice la suma cumulativa de recompensas a lo largo del tiempo.

- Estas recompensas vienen dadas por r_t .
- Hemos definido también el *discount factor* γ , que define un descuento del valor de las recompensas a lo largo del tiempo.
- Formalmente, queremos encontrar la política que maximice la suma cumulativa descontada de las recompensas:

$$\pi^* = \arg\max_{\pi} \mathbb{E}\left[\sum_{t\geq 0} \gamma^t r_t | \pi\right]$$

Figura 6. Fórmula de la política óptima.Fuente:
http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture14.pdf

Como tenemos ciertos valores aleatorios, ya que las transiciones P_{sa} han sido definidas de manera estocástica y lo mismo podría hacerse con el estado inicial y las *rewards*, la fórmula toma una esperanza matemática. Como vemos, el *discount factor* se multiplica por sí mismo en cada *time step*. Por tanto, tenemos que:

- Estas recompensas vienen dadas por r_t .
- Si = 1, no hay descuento.
- Cuanto más pequeño sea , menos sumarán las recompensas futuras en la búsqueda de la política óptima (*optimal policy*).

Veamos un ejemplo de *optimal policy* en un problema sencillo. Imaginemos que tenemos la siguiente cuadrícula, donde cada cuadro es un estado y las posibles acciones son moverse en las cuatro direcciones.

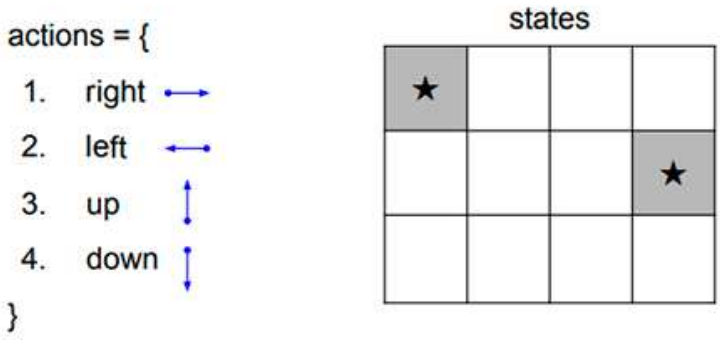


Figura 7. Ejemplo de MDP para encontrar la *optimal policy*.Fuente: http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture14.pdf

El objetivo es alcanzar uno de los estados marcados con una estrellita en el **menor número de acciones**. Por ello, a cada movimiento en la cuadrícula se le asigna una recompnesa negativa (por ejemplo, $r = -1$). De este modo, el agente está forzado a llegar a un estado terminal cuanto antes para evitar acumular valores negativos de recompensas.

Con esta información, podemos representar la *optimal policy* de la siguiente manera y compararla con una *policy* aleatoria:

Figura 8. Comparación entre *optimal policy* y una aleatoria.Fuente: http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture14.pdf

Value function y

Q-value function

Seguir una *policy* produce una trayectoria a lo largo del problema. Tal y como hemos visto, en esta estamos en un estado, realizamos una acción y recibimos una recompensa y un nuevo estado.

$$s_0, a_0, r_0, s_1, a_1, r_1, s_2, a_2, r_2 \ldots$$

Vamos a definir ahora dos conceptos fundamentales en *reinforcement learning* que nos permiten evaluar cómo de bueno es un estado y una acción a partir de la trayectoria potencial que podemos tomar a partir de estos.

La ***value function:*** *V* en el estado s para una *policy* π es la recompensa acumulada esperada a partir de seguir la *policy* desde el estado s :

$$V^{\pi}(s) = \mathbb{E}\left[\sum_{t \geq 0} \gamma^t r_t | s_0 = s, \pi\right]$$

Figura 9. *Value function*.Fuente: http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture14.pdf

L a ***Q-value function:*** Q en el estado s y acción a para una *policy* π es la recompensa acumulada esperada al elegir la acción a en el estado s y, después, seguir la *policy*:

$$Q^{\pi}(s, a) = \mathbb{E}\left[\sum_{t \geq 0} \gamma^t r_t | s_0 = s, a_0 = a, \pi\right]$$

Figura 10. *Q-value function*.Fuente: http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture14.pdf

Con esto, la ***optimal Q-value function*** Q es el máximo valor alcanzable con una *policy* π a partir de un par (estado, acción):

$$Q^*(s,a) = \max_{\pi} \mathbb{E}\left[\sum_{t\geq 0} \gamma^t r_t | s_0 = s, a_0 = a, \pi\right]$$

Figura 11. *Optimal Q-value function*.Fuente:
http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture14.pdf

Conociendo Q , la *optimal policy* π viene dada por realizar la acción que tiene un valor mayor de Q para el estado en el que estamos.

Por otro lado, la función Q satisface la **ecuación de Bellman**:

$$Q^*(s,a) = \mathbb{E}_{s'\sim\mathcal{E}}\left[r + \gamma \max_{a'} Q^*(s',a') | s, a\right]$$

Figura 12. Ecuación de Bellman.Fuente:
http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture14.pdf

De manera intuitiva, esta ecuación nos dice que si los valores de la función óptima Q son conocidos en $Q\,(s',a')$, entonces la estrategia óptima consiste en encontrar la acción que maximiza el valor de Q para cualquier de las acciones a' que podemos ejecutar después de (s,a). Esto es, partiendo de todas las acciones que podemos hacer a partir de ejecutar a en el estado s , si conocemos el valor óptimo para cada posible acción a' en el estado s' en el que hemos acabado, hemos de tomar la que maximice el valor. La esperanza aparece de nuevo porque en el marco de los MDP el estado destino tiene cierto componente aleatorio a partir de la distribución P_{sa} .

La ecuación de Bellman nos da un primer método para obtener Q y que es conocido como ***value iteration***, consiste en utilizar la ecuación de Bellman de forma iterativa:

$$Q_{i+1}(s,a) = \mathbb{E}\left[r + \gamma \max_{a'} Q_i(s',a')|s,a\right]$$

Figura 13. *Value iteration*.Fuente: http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture14.pdf

De manera que Q converge a Q cuando i tiende a infinito.

Este método funciona correctamente y ha sido muy utilizado en la época anterior a la llegada del *deep learning*. Si bien no profundizaremos en el algoritmo en clase, comentar que este tiene el problema de que necesita calcular $Q(s,a)$ para cada pareja (estado, acción). Esto lo hace solo tratable computacionalmente para problemas con un número de estados y acciones muy reducidos. Por ejemplo, en un problema donde un agente intente aprender a jugar a videojuegos, un estado viene dado por todos los píxeles de la pantalla, lo cual hace computacionalmente imposible mantener una tabla con valores para cada estado.

8.4. Deep Q-Learning

Hasta ahora hemos visto el marco tradicional de *reinforcement learning*. En este apartado veremos uno de los grandes avances en esta área obtenido a partir de los avances en *deep learning*. En particular, veremos cómo funciona el *Deep Q-Learning* y cómo fue utilizado por la empresa DeepMind para hacer que unos agentes aprendieran a jugar a juegos clásicos de Ataria directamente a partir de las imágenes del juego.

Anteriormente, hemos visto que el algoritmo de *value iteration* para *Q-learning* es computacionalmente intratable para problemas con un gran número de estados o acciones. La idea para superar este problema es utilizar una aproximación en forma de función para estimar la *Q-function*. De este modo, si obtenemos una función aproximada, no necesitamos tratar el problema de manera discreta por cada combinación de estado y acción.

$$Q(s,a;\theta) \approx Q^*(s,a)$$

Figura 14. Función aproximada para estimar la *Q-funciton*.Fuente: http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture14.pdf

La idea clave del Deep Q-learning (idea que seguro no nos sorprende a estas alturas) es utilizar una **red neuronal profunda como función de aproximación**.

De nuevo, partiremos de que la *optimal Q-function* que buscamos satisface la ecuación de Bellman:

$$Q^*(s,a) = \mathbb{E}_{s' \sim \mathcal{E}}\left[r + \gamma \max_{a'} Q^*(s',a') | s,a\right]$$

Figura 15. *Optimal Q-funciton*.Fuente: http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture14.pdf

Nuestra red neuronal calculará $Q(s,a;\theta)$ en donde:

- La red tendrá de *input* el estado s y en la última capa tendrá una salida por cada posible acción a.
- θ representa en este caso los parámetros o *weights* de la red.

Para aprender esta red neuronal, definiremos de nuevo una función de pérdida o *loss function*:

$$L_i(\theta_i) = \mathbb{E}_{s,a\sim\rho(\cdot)}\left[(y_i - Q(s,a;\theta_i))^2\right]$$

Figura 16. Función de pérdida.Fuente: http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture14.pdf

Esta función nos dice lo bien que aproxima la red $Q(s,a;\theta)$ al valor real y_i. Sin embargo, y_i indica el valor óptimo que buscamos, ¡y este valor es desconocido!

Así que utilizaremos la ecuación de Bellman de nuevo para obtener este valor (y_i) de manera iterativa a partir de la red neuronal actual:

$$y_i = \mathbb{E}_{s'\sim\mathcal{E}}\left[r + \gamma \max_{a'} Q(s',a';\theta_{i-1})|s,a\right]$$

Figura 17. Función de pérdida.Fuente: http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture14.pdf

Nótese que y_i es la parte de la derecha de la ecuación de Bellman, donde hemos sustituido Q por la aproximación $Q(s,a;\theta i-1)$, esto es, la aproximación que teníamos en un valor de iteración anterior. Así, de manera iterativa intentaremos hacer que el *Q-value* se acerque al valor que debería de tener (y_i) según la ecuación de Bellman si *Q* fuese óptimo.

Caso de estudio con Atari Breakout

Veamos un caso particular del algoritmo de *deep Q-learning* aplicado al juego de *Atari Breakout*. Este fue uno de los juegos de Atari utilizados para entrenar agentes en el *paper* original de DeepMind. En gran parte de los juegos, los agentes entrenados con aprendizaje por refuerzo consiguieron alcanzar un nivel mucho más elevado que el de los humanos.

En *Atari Breakout* el objetivo es romper todos los ladrillos de la pantalla usando una bolita que controlamos con una paleta inferior de manera que no caiga al vacío. El objetivo es acabar el juego con la mayor puntuación posible. El estado viene dado por todos los píxeles de la pantalla y las acciones son mover la paleta a la izquierda y a la derecha. Las recompensas son puntos según los ladrillos rotos, y una recompensa negativa a lo largo del tiempo para forzar al agente a acabar cuanto antes.

En la siguiente figura, vemos la arquitectura de la *Q-network* que define el algoritmo de *Q-learning*.

- El *input state* se corresponde con 4 imágenes 84x84 con 4 frames del juego.
- Arriba de este hay varias capas convolucionales que se encargan de encontrar una representación de la pantalla en forma de *features* de alto nivel, de modo que el algoritmo sea capaz de aprender conceptos como la bola, la paleta y los ladrillos.
- Más arriba, tenemos dos *fully connected layers*, la última de ellas con una salida por cada acción posible, que depende del juego en particular. La acción también podría pasarse como *input* a la red, pero si el número de acciones no es muy grande, es más eficiente obtener los *Q-values* de cada acción en el mismo *forward pass*.

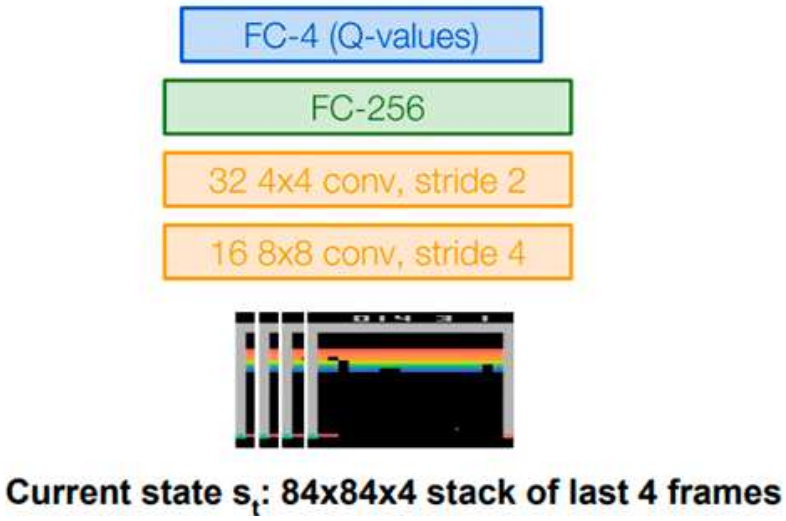


Figura 19. *Q-network* para *Breakout*.Fuente: http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture14.pdf

Para el entrenamiento de la red, se utilizó una serie de trucos explicados en el *paper*. Uno de ellos es ***Experience Replay***. En vez de simplemente jugar y alimentar a la red neuronal con *inputs* consecutivos, se crea una memoria de experiencias conocidas (acciones, transiciones y recompensas) y se crean mini *batches* de manera aleatoria a partir de ella. Esto evita tener *samples* del juego correlacionadas y hace el entrenamiento mucho más efectivo.

Por otro lado, hemos visto que la propia red $Q\left(s,a;\theta\right)$ se utiliza en el objetivo y_i , y por tanto cambia de manera continua. Para hacer el proceso de entrenamiento más estable, la red en el objetivo y_i solo se actualiza con los valores actuales de la *Q-network* cada 1000 *steps*.

Veamos el algoritmo completo de *Deep Q-learning* con *Experience Replay*:

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
  end for
end for

Figura 20. Algoritmo de *Deep Q-learning*.Fuente: https://www.cs.toronto.edu/~vmnih/docs/dqn.pdf

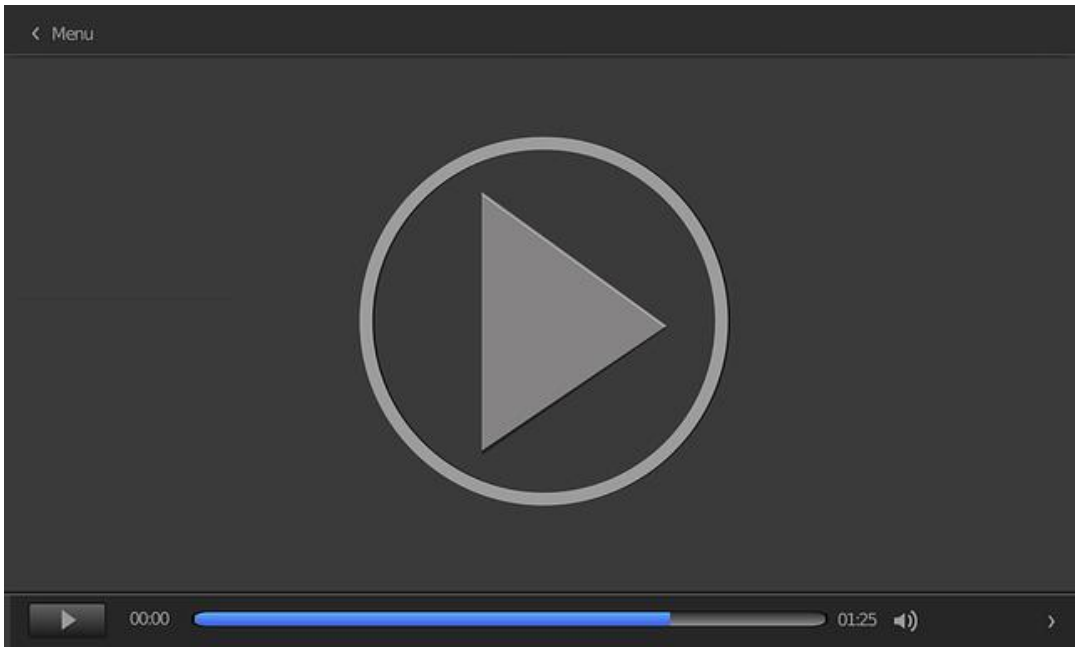
- El primer paso es inicializar la *replay memory* y la *Q-network* con pesos aleatorios.
- A partir de ahí, se juegan M episodios (partidas completas). Por cada episodio, se obtiene un estado inicial y una secuencia preprocesada, lo que quiere decir que los 4 frames de las imágenes son procesados mediante varias técnicas, como convertir a blanco y negro.

- Después, a lo largo de T instantes de tiempo en cada partida, se selecciona una acción al azar o se escoge la mejor acción posible a partir de la *Q-function* actual. La elección de una acción al azar se corresponde a que en ocasiones queremos explorar nuevos caminos para ver si encontramos alguna solución mejor a lo que tenemos hasta ahora, y es un problema clave en el mundo del aprendizaje por refuerzo (*exploration-exploitation tradeoff*).

- Una vez elegida la acción a realizar, se ejecuta esta acción en el emulador del juego, lo cual nos da el siguiente estado y la recompensa, que se almacenan en la *replay memory*. En este punto es cuando realmente entrenamos la Q-network, obteniendo una *batch* aleatoria de elementos a partir de la memoria y ejecutando *gradient descent*.

AlphaGo

Hablamos sobre el programa AlphaGo creado por DeepMind, compañía de Google; se trata de uno de los mayores hitos de la inteligencia artificial de los últimos años ante la victoria de este programa frente a un jugador profesional.



Accede al vídeo:

https://unir.cloud.panopto.eu/Panopto/Pages/Embed.aspx?id=579a4307-e0ca-4f39-ba3a-affc009a0e1c

Deep Reinforcement Learning: Pong from Pixels

Karpathy, A. (21 de mayo de 2015). Deep Reinforcement Learning: Pong from Pixels [Blog post].
http://karpathy.github.io/2016/05/31/rl/

Blog post de Andrej Karpathy acerca de aprendizaje por refuerzo o *Reinforcement Learning* en el que trata al juego de Pong como un caso especial de MDP.

Reinforcement Learning and Control

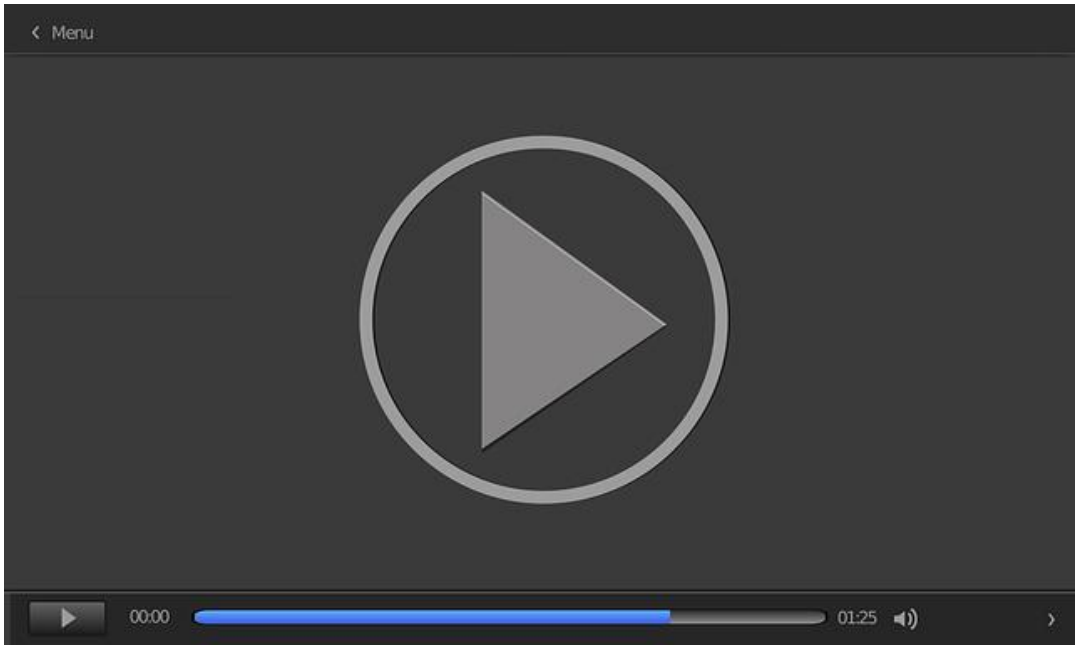
Ng, A. (2017). Lecture note 16: Reinforcement Learning and Control [Lecture notes]. http://cs229.stanford.edu/notes/cs229-notes12.pdf

Lecture Notes del CS229 de Stanford, donde se explica el algoritmo de *value iteration* en detalle.

Emergence of Locomotion Behaviours in Rich Environments

Google DeepMind (14 de julio de 2017). *Emergence of Locomotion Behaviours in Rich Environments [Vídeo].* Youtube. https://www.youtube.com/watch?v=hx_bgoTF7bs

Vídeo de DeepMind con ejemplos de agentes entrenados para andar hacia delante.



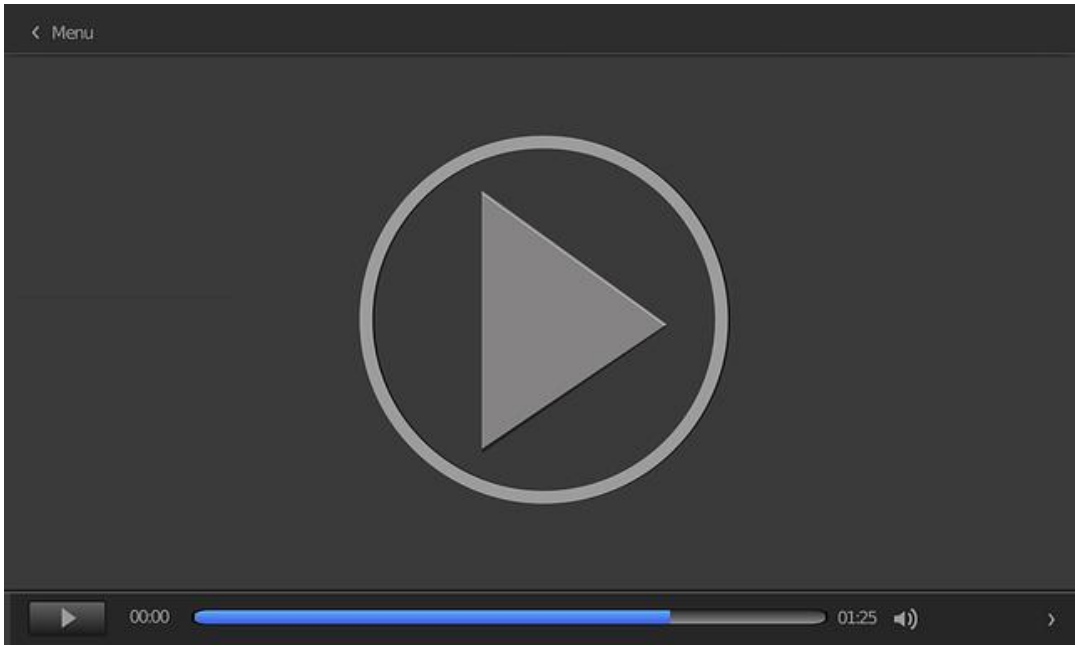
Accede al vídeo:

https://www.youtube.com/embed/hx_bgoTF7bs

Google DeepMind's Deep Q-learning playing Atari Breakout

Two Minute Papers (07 de marzo de 2015). *Google DeepMind's Deep Q-learning playing Atari Breakout! [Vídeo].* Youtube. https://www.youtube.com/watch?v=V1eYniJ0Rnk

Ejemplo de cómo el agente se hace mejor con el tiempo tras jugar a Breakout.



Accede al vídeo:

https://www.youtube.com/embed/V1eYniJ0Rnk

Playing Atari with Deep Reinforcement Learning

Mnih, V. et al. (2013). Playing Atari with Deep Reinforcement Learning. *NIPS'13 Deep Learning Workshop*. https://www.cs.toronto.edu/~vmnih/docs/dqn.pdf

Paper original de Deep Mind con el algoritmo *Deep Q-Learning* aplicado a juegos de Atari.

1. En *reinforcement learning* (marca la respuesta correcta):

A. Un agente está en un entorno y trata de aprender recompensas a partir del estado en el que se encuentra.

B. Un agente está en un entorno y trata de aprender los estados óptimos del entorno.

C. Un agente está en un entorno y trata de aprender las acciones que maximizan las recompensas en cada estado posible.

D. Un agente está en un entorno y trata de aprender las acciones que minimizan las recompensas en cada estado posible.

2. ¿Cuál de los siguientes problemas está definido como un problema de *reinforcement learning*? (marca todas las respuestas correctas):

A. Un helicóptero autónomo puede realizar acciones como mover las aspas y cambiar la dirección. Recibe *rewards* positivas si se mantiene en vuelo y negativas si cae al suelo.

B. Un robot industrial tiene que mover objetos de un sitio a otro y sus acciones son el movimiento de una mano mecánica. Recibe *rewards* positivas por mover el objeto con éxito.

C. Un clasificador de imágenes tiene una serie de imágenes y unas etiquetas por cada imagen. El clasificador aprende a catalogar las imágenes según las etiquetas.

D. Un algoritmo de recomendación asigna a cada usuario los vídeos más probables a ver en el futuro según el historial del usuario.

3. El entorno o *environment* en un problema de *reinforcement learning* nos da (marca todas las respuestas correctas):

A. Una acción

a_t a realizar para el agente.

B. Una recompensa

r_t a partir del estado y la acción realizada por el agente.

C. Un nuevo estado

s_{t+1} a partir del estado

s_t y la acción

a_t realizada por el agente.

D. Una *policy* a seguir por el agente.

4. La *value function* V (marca todas las respuestas correctas):

A. Está asociada a una *policy* π y la trayectoria que definen las acciones de esta *policy*.

B. Si el *discount factor* es 1, todas las recompensas valen lo mismo independientemente de en qué momento del tiempo se consiguen.

C. Nos da una medida de lo bueno que es un estado siguiendo una *policy* π a partir de las recompensas esperadas si seguimos la *policy* desde ese estado.

D. Nos da una medida de lo bueno que es un estado y una acción siguiendo una *policy* π a partir de las recompensas esperadas si seguimos la *policy* desde ese estado.

5. A partir de

Q^*, podemos obtener la *optimal policy*

π^*(marca la respuesta correcta):

A. Por cada estado

s, elegimos la acción que nos da un valor mayor de

$Q^*(s,a)$.

B. Por cada estado

s, aplicamos la ecuación de Bellman a partir de

Q^* para obtener el nuevo estado

s' y acción

a'.

C. Es imposible calcular

π^* a partir de

Q^*.

6. *Q-learning* a partir de *value iteration* (marca la respuesta correcta):

A. Se caracteriza por no utilizar la ecuación de Bellman.

B. Se hace computacionalmente intratable cuando hay muchos estados o acciones.

C. Converge mejor a mayor número de acciones.

7. En el algoritmo *Deep Q-Learning* visto en clase, la red neuronal tiene una salida por cada acción, en vez de representar la acción como *input*. Esto es así porque (marca la respuesta correcta):

A. El número de acciones puede llegar a ser muy grande.

B. El número de estados puede llegar a ser muy grande.

C. Podemos obtener todos los *Q-values* para un estado con un solo *forward pass*.

D. Ninguna de las anteriores.

8. *Experience Replay* (marca todas las respuestas correctas):

A. Define una memoria de jugadas guardadas con sus *rewards* de manera que podemos obtener jugadas aleatorias a partir de ella.

B. Es una memoria que guarda jugadas, *v-values* y *q-values* para cada jugada.

C. Hace el entrenamiento más eficiente al mostrar jugadas variadas en vez de jugadas consecutivas.

9. En el algoritmo *Deep Q-Learning* es importante dar cierta probabilidad a realizar acciones aleatorias porque (marca todas las respuestas correctas):

A. En un principio, la red neuronal es aleatoria y no puede ser considerada como una *policy* óptima.

B. Incluso cuando la red defina una buena aproximación a Q^*, es importante probar nuevas acciones que puedan llevar a mejores soluciones.

C. El algoritmo converge más rápido.

10. El estado en *Deep Q-Learning* aplicado a videojuegos Atari es (marca la respuesta correcta):

A. Una serie de valores simbólicos y formales que definen el estado del juego.

B. Un sistema de ecuaciones que define velocidad de la bola, la física respecto a la paleta, etc.

C. Las imágenes del juego que ve un jugador (dispuestas en conjuntos de 4 frames seguidos).

D. Ninguna de las anteriores.