Ajustando Q-Learning para generar jugadores automáticos: un ejemplo basado en Atari Breakout

Guillermo Fernández-Vizcaíno, Francisco J. Gallego-Durán

Cátedra Santander-UA de Transformación Digital
Universidad de Alicante
gfv4@alu.ua.es, fjgallego@ua.es

Abstract. El presente artículo explora la utilización de *Machine Learning* para la producción automatizada de agentes inteligentes en videojuegos. Concretamente, se muestra la creación de jugadores automáticos del videojuego clásico *Breakout* para la consola *Atari 2600*. La simplicidad de este videojuego permite centrarse en el problema principal tratado: el correcto ajuste de parámetros para la obtención de resultados deseados. En concreto, se analiza el ajuste de *Q-Learning* y su relación con los resultados de aprendizaje obtenidos. La metodología aplicada en este artículo es fácilmente aplicable a cualquier otro videojuego o entorno, permitiendo la utilización de *Q-Learning* para generar distintos agentes inteligentes de manera automática cuyo comportamiento sea ajustablemente imperfecto a la par que suficientemente variable pero predecible.

Keywords. *Machine Learning*, *Reinforcement Learning*, *Q-Learning*, *Atari 2600*, *Breakout*.

1 Introducción

Una de las áreas del videojuego que mayor interés y expansión están teniendo en los últimos años es la Inteligencia Artificial [3]. Las capacidades gráficas e interactivas de los videojuegos han alcanzado un nivel de realismo e inmediatez muy elevados, mostrando mundos enteros en tiempo real. En paralelo, la atención de los jugadores se ha focalizando en las capacidades de los personajes artificiales para ser creibles, retar al jugador y comportarse como humanos, tanto inteligente como erróneamente. Un personaje inteligente que comete errores o muestra cierta variabilidad en sus actos resulta más atractivo jugablemente y más creíble, mejorando la calidad del producto final.

El desarrollo de personajes artificiales en videojuegos es comunmente tratado como creación de contenido. Un ejemplo lo encontramos en *Half Life* [4]: el núcleo del videojuego implementa comportamientos básicos que son exportados a una capa de *scripting* [16]. Los diseñadores de niveles implementan todas las posibles acciones y respuestas de los personajes utilizando *scripting*. El resultado

final funciona de manera similar a un sistema experto, pudiendo resultar muy realista para el jugador si se diseñan un gran número de reglas para muchas posibles situaciones. Sin embargo, el coste de diseño es muy elevado en cantidad de horas y personas involucradas [4].

Una posible forma de reducir estos costes y ampliar los límites de los personajes diseñados es utilizar Machine Learning. Machine Learning permite generar personajes artificiales mediante entrenamiento. Este entrenamiento depende de parámetros que afectan directamente al resultado. Conociendo la influencia de los parámetros en el resultado se pueden generar distintos tipos de personajes artificiales. Este artículo explora esta influencia en los agentes entrenados con *Q-Learning* [21][22]. Utilizando el videojuego *Breakout* [1] de *Atari 2600* [11][3] como ejemplo, se muestra qué parámetros son importantes, cómo influyen y cómo ajustarlos para conseguir mejores resultados.

La sección 2 presenta algunas definiciones y el entorno en que se desarrolla el presente trabajo. La sección 3 repasa algunos trabajos previos, mientras que la sección 4 describe el desarrollo principal de este trabajo. La sección 5 muestra los resultados de los experimentos y, finalmente, la sección 6 enumera las conclusiones y trabajos futuros.

2 Conceptos

A continuación se definen los conceptos que se utilizan en el desarrollo de este trabajo:

- ***Atari 2600*** es una videoconsola desarrollada en 1977 y vendida durante más de una década. Posee una CPU de propósito general con una frecuencia de reloj de 1.19Mhz. Más de 500 juegos originales fueron publicados para este sistema y aún hoy siguen apareciendo nuevos títulos desarrollados por fans. Una de las partes más características de esta videoconsola es el *joystick*, que se puede apreciar junto a la videoconsola en la figura 1. La memoria RAM tiene muy poca capacidad, concretamente 128 bytes. Debido a las claras limitaciones de la videoconsola, los juegos de ésta son relativamente sencillos. Esto los convierte en un buen *benchmark* para técnicas de Machine Learning actuales.

Fig. 1. *Atari 2600* a la izquierda y captura del juego *Breakout* a la derecha.

– ***Breakout*** es un juego de tipo *arcade*, como todos los juegos de *Atari 2600*, que tiene como objetivo romper todos los bloques que cubren la mitad superior de la pantalla. El jugador es una pala o raqueta que puede moverse a la izquierda o a la derecha para golpear una pelota y romper con ella los bloques. Hay que impedir que la pelota caiga, contando con 5 oportunidades antes de perder el juego. Se trata de un juego de habilidad que requiere reflejos, cálculo y reacción rápida para poder dominarlo. En la figura 1 se muestra una captura del momento inicial del juego.
– ***Arcade Learning Environment*** (ALE, entorno de aprendizaje de *Arcade*) es un entorno de programación que incluye el emulador *Stella* [2], que emula una máquina *Atari 2600*. La interfaz de programación que proporciona ALE permite crear código para interactuar con el emulador de una forma muy sencilla. Así mismo, ALE permite ejecutar juegos sin interfaz gráfica, acelerando la ejecución y facilitando las tareas de entrenamiento para algoritmos de Machine Learning. En este trabajo se ejecuta el juego *Breakout* a través del entorno ALE.
– Un ***Agente*** es una entidad que puede desenvolverse de manera autónoma en un entorno determinado. Los agentes tienen la misión de, usando datos obtenidos del entorno, realizar una tarea lo mejor que puedan. Las acciones realizadas por el agente tienen repercusión en el entorno: producen un cambio de estado y proporcionan al agente una recompensa que le indica lo bien o mal que está realizando la tarea (*reward*). En la figura 2 se representa de manera gráfica la interacción de todo agente con su entorno.

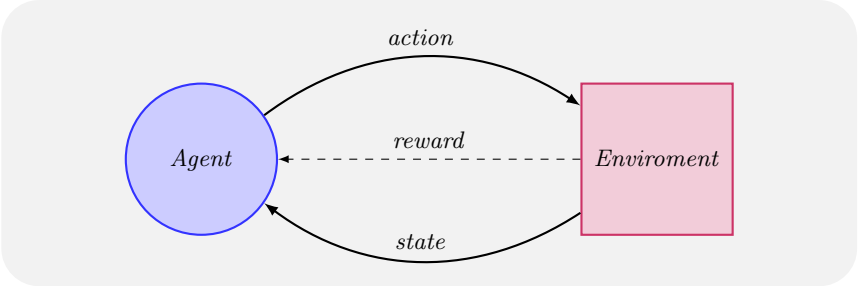

Fig. 2. Esquema de un agente interactuando con su entorno.

– ***Reinforcement Learning*** (aprendizaje por refuerzo) es un tipo específico de aprendizaje dentro del campo del Machine Learning muy utilizado para juegos y entornos interactivos [17]. Se caracteríza por realizar un entrenamiento mediante señales de refuerzo, por no poder determinarse a priori la mejor decisión absoluta ante un estado cualquiera. En este caso, las señales de refuerzo representan una evaluación de las acciones tomadas similar a una función de idoneidad. Los entornos interactivos y juegos tienen en común esta última parte: no es posible determinar la existencia de una mejor

acción absoluta ante un determinado estado del mundo, pero sí evaluar las acciones en función de los resultados.

3 Estado del arte

Desde los inicios del *Machine Learning* hasta la actualidad han habido numerosos avances importantes. En todo este tiempo, los juegos han estado siempre presentes. Ya en 1957, Arthur L. Samuel se interesó por el juego de las damas [13] introduciendo algunas técnicas de Machine Learning iniciales como *rote learning*. Samuel manifestó su pensamiento de que enseñar a los ordenadores a jugar era mucho más productivo que diseñar tácticas específicas para cada juego.

En 1992, Gerald Tesauro desarrolló una de las primeras aplicaciones de *Reinforcement Learning* conocidas en juegos: *TD-Gammon* [19][18]. *TD-Gammon* es el primer programa que consiguió jugar al *Backgammon* por encima del nivel de un humano profesional. *TD-Gammon* utiliza una red neuronal entrenada mediante *Temporal-Difference* jugando 1.5 millones de partidas contra sí misma. La red aproxima el valor de las jugadas, pudiendo obtener la mejor o una muy buena aproximación, incluso en situaciones que nunca antes ha visto.

Aunque el objetivo del *Machine Learning* es el aprendizaje autónomo a partir de datos, los algoritmos han sido aplicados tradicionalmente con un propósito concreto. Los ejemplos de Samuel y Tesauro [13][19][18] son una muestra: concebidos para jugar al *Backgammon* y las damas exclusivamente, similar al conocido *DeepBlue*[5] que ganó al campeón del mundo, pero sólo era capaz de jugar al ajedrez. Actualmente se investiga en superar estas limitaciones, creando algoritmos que aprendan ante varios problemas. Un ejemplo es el *General Game Playing* (juego generalizado) [7][9][14][2], que busca algoritmos que aprendan a jugar a múltiples juegos a la vez. En este área se enmarcan los últimos grandes éxitos del *Machine Learning*, con la aparición del *Deep Learning* [8][10][12][15]. El *Deep Learning* pretende automatizar la extracción de características, que es vital para un aprendizaje significativo. El *Deep Learning* deja que el propio algoritmo extraiga las características a partir de datos en crudo, evitando usar conocimiento experto sobre el problema. Un ejemplo clave es el algoritmo *DQN (Deep Q-Network)* de *Google DeepMind* [12]. DQN usa una *Convolutional Neural Network* para extraer las características de las imágenes obtenidas de ALE. DQN aprende a partir de imágenes de entrada en crudo, sin utilizar conocimiento experto, y aprende a jugar a muchos juegos de *Atari 600*. En ocasiones, DQN alcanza el nivel de humanos expertos o mejor. Por tanto, el *Reinforcement Learning* puede sernos útil para generar jugadores artificiales de calidad.

Pese a los grandes avances en *Machine Learning*, el sector profesional del desarrollo de videojuegos sigue diseñando los personajes artificiales manualmente [4][16]. Aquí el *Machine Learning* tiene un potencial doble: 1) Reducción de costes de diseño, al poder generar los personajes artificiales, y 2) capacidad de ajustar los resultados para generar personajes variables, pero suficientemente predecibles, que den una sensación más realista a los jugadores. En este trabajo se muestran varios pasos de ajuste de parámetros y elaboración de características

para un algoritmo de *Q-Learning*, analizando los resultados obtenidos. Estos pasos son fácilmente extrapolables a otros videojuegos y adaptables para conseguir los tipos de aprendizaje propuestos.

4 Metodología

Es importante matizar que la metodología propuesta incluye extracción manual de características, por lo que no es válida para la resolución de problemas generales como en [20][3][10][6]. El objetivo es ser útil en la industria actual del videojuego, por lo que el ajuste manual y el control sobre los resultados obtenidos es importante. De todas formas, esta metodología puede ser extendida y generalizada para otras necesidades futuras.

Los algoritmos de *Reinforcement Learning* desconocen el entorno en que actúan. El entrenamiento es el proceso por el cual un agente aprende de la experiencia, es decir, que ajusta sus patrones observación-acción a patir de un conjunto de observaciones

$$< s_t, a_t, r_{t+1}, s_{t+1} >^* \tag{1}$$

donde s_t es el estado actual, a_t la acción tomada, r_{t+1} el *reward* (recompensa) por tomar dicha acción y s_{t+1} el estado siguiente. El objetivo de los agentes es el de realizar las acciones que maximicen su futuro *reward*. En el juego *Breakout*, existen tres posibles acciones en cada instante de toma de decisión: ir a la izquierda, a la derecha o no moverse.

Para aprender de la experiencia se utilizará *Q-Learning* [21][22]. El funcionamiento de *Q-Learning* se basa en $Q(s, a)$, conocida como la función *action-value* (acción-valor). $Q(s, a)$ devuelve el valor medio del *reward* al aplicar la acción a estando en el estado s.

$$Q(s, a) = \mathbb{E}\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \;\middle|\; s_t = s, a_t = a\right] \tag{2}$$

El factor $\gamma \in [0, 1]$ es el llamado *discount factor* (factor de descuento). Cuanto mayor sea el valor de γ, más "visión" tendrá el algoritmo: será capaz de asociar acciones a futuros *rewards* (*long term reward*). A priori puede intuirse que los valores de γ cercanos a 1 serán más óptimos para el juego *Breakout*, ya que los *rewards* no se obtienen inmediatamente después de aplicar una acción: es tras la ejecución de varias acciones posteriores cuando se conoce si una acción concreta era positiva.

Sea $Q^*(s, a)$ la función acción-valor que maximiza el *reward*. Cuando el número de iteraciones se acerca a infinito, el algoritmo *Q-Learning* garantiza la convergencia. Puesto que esto es impracticable, su valor se aproximará actualizándola iterativamente mediante *Temporal-Difference*.

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)\right] \tag{3}$$

El parámetro $\alpha \in [0, 1]$ es el llamado *learning rate* (ratio de aprendizaje). Si $\alpha = 0$, $Q(s, a)$ nunca es actualizada y, por tanto, no se produce aprendizaje. Si $\alpha = 1$, $Q(s, a)$ se actualiza lo máximo posible en cada iteración. Aunque actualizar el máximo puede resultar atractivo, debe tenerse en cuenta que el resultado puede también olvidar lo aprendido anteriormente. Un equilibrado ajuste de este parámetro es fundamental para potenciar el rendimiento del *Q-Learning*.

Es vital que el algoritmo explore en profundidad el conjunto de estados, manteniendo el equilibrio entre exploración y explotación. Exploración es ejecutar acciones que no son las mejores, para actualizar estados no explorados de la función $Q(s, a)$. Explotación es reforzar y perfeccionar el uso de la mejor acción que hemos encontrado hasta el momento (Conocida como *greedy action*). La bondad de los resultados dependerá del equilibrio de estas dos aternativas. En [17] se discuten criterios de selección y la dificultad de saber cual de ellos funciona mejor a priori. En este trabajo se utiliza la estrategia ***ϵ-greedy policy*** por su sencillez y resultados. En cada turno se realiza una acción aleatoria con probabilidad ϵ, o la mejor acción encontrada con probabilidad $1 - \epsilon$.

En esta propuesta, cada estado s se describe con una 4-tupla: $s = (Ball_y, Ball_{vx}, Ball_{vy}, Diff_x)$ (ver figura 3). $Diff_x$ evita el uso de una 5-tupla para representar el estado, reduciendo la cantidad total de estados posibles a explorar. Como contrapartida, impide que el algoritmo detecte dónde están las paredes puesto que desconoce la posición horizontal absoluta de la pelota y el jugador (ver figura 4). Todos los valores se obtienen a partir de la RAM de la consola emulada durante la ejecución.

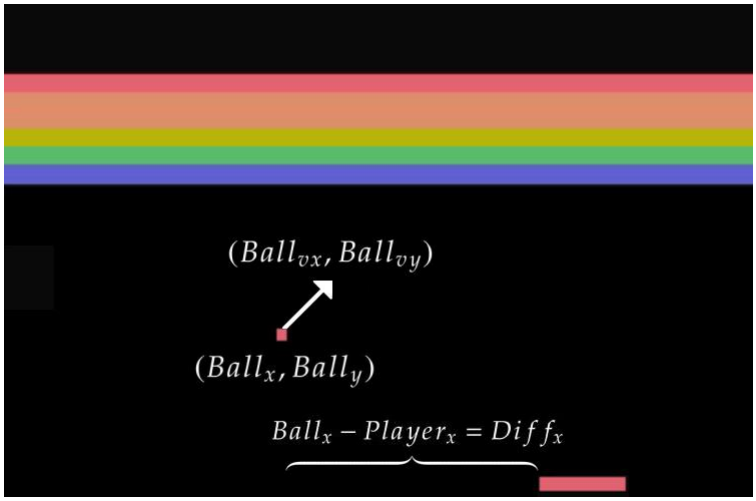

Fig. 3. Variables utilizadas para definir un estado.

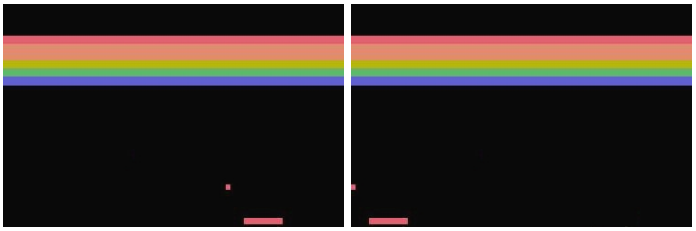

Fig. 4. Dos situaciones diferentes que producen el mismo estado ($Diff_x$ vale lo mismo). Suponiendo que la pelota cae hacia la izquierda, el la situación derecha no moverse y esperar que la pelota rebote sería una buena acción. No moverse en la situación izquierda supondría la pérdida de una vida.

La implementación de *Q-Learning* propuesta utiliza una tabla en la que cada combinación de estado s y acción a tiene un valor asociado $Q(s, a)$ (ver la tabla 1 como ejemplo). Inicialmente, todas las combinaciones tienen valor 0. La regla de actualización 3 modifica los valores en tiempo de ejecución.

Estado s	**Acción** a	**Valor** $Q(s, a)$
⋮	⋮	⋮

Table 1. Representación de la tabla que guarda el agente.

Es importante el número de posibles estados $|s|$ del entorno: cuanto menor sea $|s|$, más rápido se actualizará $Q(s, a)$, acelerando el aprendizaje. En un espacio continuo, $|s| = \inf$, por lo que una discretización previa sería imprescindible. *Breakout* no está en un espacio continuo, pero $|s|$ por sus dimensiones puede asimilarse. La estrategia propuesta será discretizar el espacio dividiendo $Diff_x$ y $Ball_y$ entre un *factor de discretización* d. Cuanto más grande sea d, menor será $|s|$, pero menos precisa será la percepción, lo que podrá influir en la calidad de las decisiones.

5 Experimentación y Resultados

El primer paso es seleccionar un factor de discretización d adecuado. Para ello, testeamos nuestro algoritmo con $d \in \{\mathbb{N} \cap [1, 30]\}$, visualizando los resultados en la figura 5. Atendiendo a la gráfica, $d = 10$ resulta apropiado por ser el valor más bajo con recompensas en la zona alta (12-15). Aunque otros $d > 10$ producen recompensas ligeramente superiores, $d = 10$ genera un entorno más preciso. Esto se muestra en la parte inferior de la figura 5: conforme d aumenta, el número de estados en la tabla $Q(s, a)$ se reduce drásticamente. Por tanto, menos estados, más actualizaciones, menos tiempo de aprendizaje, menos precisión. En estos experimentos, los valores de α y γ se mantienen constantes (se han utilizado $\{\alpha = 0.2, \gamma = 1\}$).

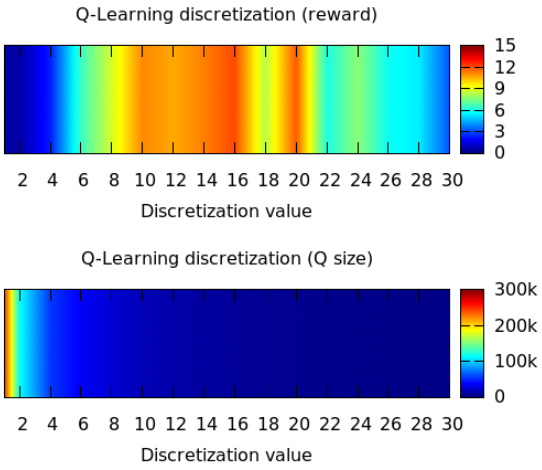

Fig. 5. Arriba se representa la tendencia del *reward* después de la ejecución de 2000 episodios con diferentes factores de discretización. Abajo se muestra cómo el número de entradas de la función $Q(s, a)$ es inversamente proporcional al factor de discretización al cuadrado (debido a que $Ball_y$ y $Diff_x$ se dividen por el factor).

Tras seleccionar d, se buscan valores óptimos para α y γ mediante *grid search* (búsqueda en rejilla). Se realizan nuevos entrenamientos con d fijo y $\alpha \in [0, 1]$, $\gamma \in [0, 1]$. La figura 6 la rejilla con los resultados. Los valores óptimos se encuentran $\alpha \in [0.1, 0.3]$, $\gamma \in [0.9, 1]$. Obsérvese que las representaciones gráficas de las figuras 5 y 6 nos permiten elegir parámetros subóptimos para la tarea u óptimos en distintos intervalos. Distintos valores darán lugar a distintos resultados de aprendizaje, pudiendo generar personajes inteligentes de distintos tipos.

Tras haber obtenido unos valores optimizados, el siguiente experimento consiste en alargar el periodo de entrenamiento y observar su evolución a más largo plazo. En la figura 7 se observan las curvas de aprendizaje de tres ejecuciones de 15000 episodios para $\alpha \in \{0.05, 0.1, 0.2\}$. Resulta interesante ver que $\alpha = 0.05$ obtiene el mejor *reward* final pese a no pertenecer al rango óptimo seleccionado anteriormente. En cambio, $\alpha = 0.2$ obtiene un aprendizaje mucho más rápido durante los primeros 2000 episodios pero pronto se produce un estancamiento. Este es el efecto comentado anteriormente del *learning rate* α: valores altos producen un aprendizaje rápido pero menos profundo, mientras que valores bajos tardan más en aprender pero pueden alcanzar un mejor rendimiento.

Para mejorar más los resultados de aprendizaje se necesita una combinación: α debe ser alto para aprender rápido y bajo para aprender profundo a largo plazo. Una forma de combinar ambos es usar la técnica *learning rate decay* (descenso del ratio de aprendizaje). Así pues, se utiliza un nuevo parámetro *decay* $D \in [0, 1]$. indicando el factor de reducción de α por cada 1000 episodios. Por tanto, siendo α_i el *learning rate* para el episodio i,

$$\alpha_i = \frac{\alpha_0 \times D \times i}{1000} \tag{4}$$

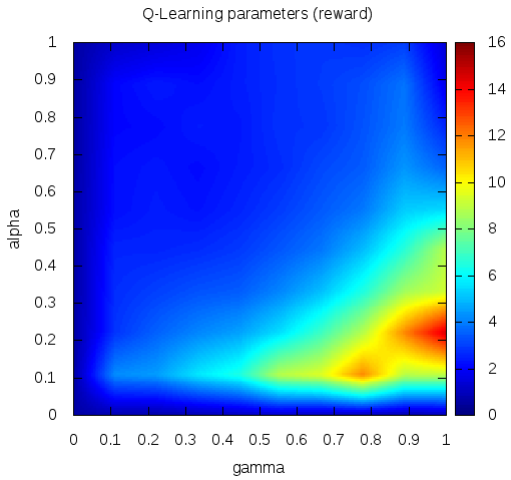

Fig. 6. *Grid search* realizado para encontrar la zona donde se encuentran los parámetros óptimos del *Q-Learning*. El color indica la tendencia del *reward* tras 2000 episodios o juegos.

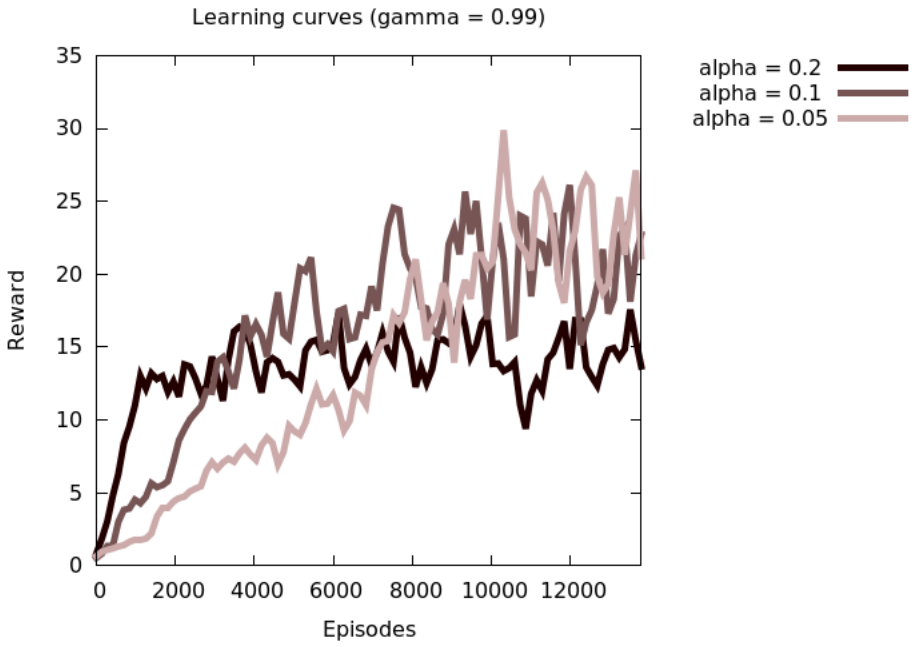

Fig. 7. Curva de aprendizaje de un agente con $\gamma = 0.99$, $d = 10$, $\alpha \in \{0.05, 0.1, 0.2\}$. Cuanto más pequeño α, más lento es el aprendizaje, pero mayor el alcance final.

Es decir, con $D = 0.2$ el *learning rate* se reduce un 20% cada 1000 episodios. Esto permite acelerar el aprendizaje al inicio y profundizar a largo plazo. Tras introducir *decay*, los resultados mejoran como puede verse en la figura 8.

La figura 8 muestra como valores altos como $D = 0.5$ hacen disminuir el *learning rate* demasiado rápido afectando al aprendizaje. Sin embargo, resulta interesante también apreciar que la curva se vuelve más estable (la varianza se reduce). Ajustando $D = 0.2$ el aprendizaje parece crecer linealmente incluso más allá de los 15000 episodios. Aún siendo un buen resultado, el efecto del *decay* hará tender α a 0, por lo que la curva terminará convergiendo igualmente.

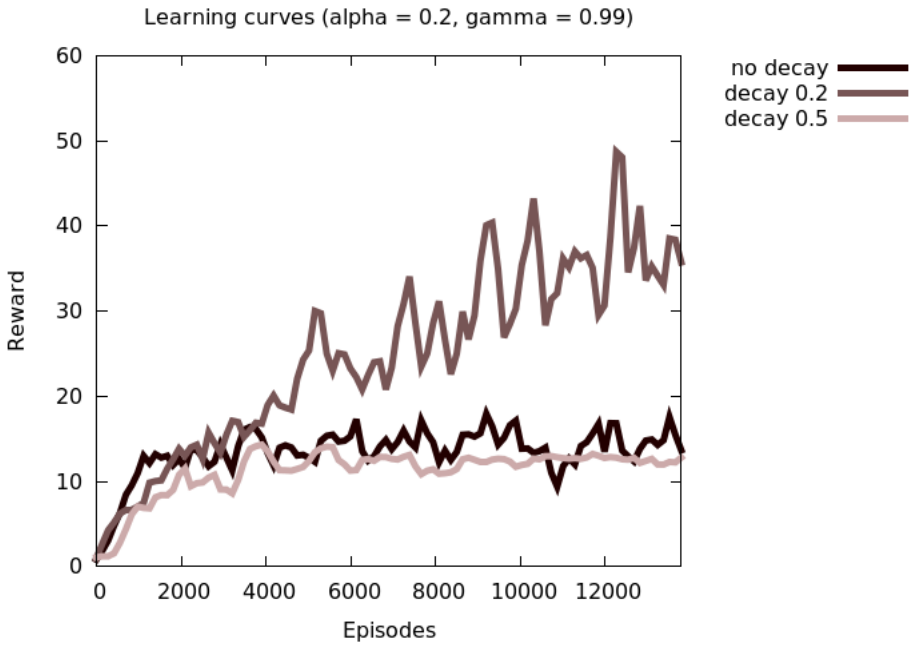

Fig. 8. Curvas de aprendizaje de un jugador automático de *Breakout* con $\{\alpha = 0.2, \gamma = 0.99, d = 10\}$.

6 Conclusiones

En este trabajo hemos hecho énfasis en las potenciales ventajas del *Machine Learning* a la hora de potenciar nuevas vías de producción de Inteligencia Artificial en la industria del videojuego. Utilizando *Reinforcement Learning* es posible generar distintos tipos de personajes artificiales con capacidades con inteligencias creíbles, variables y más humanas. Todo depende de una buena selección de parámetros de entrenamiento.

El principal problema con los algoritmos de *Reinforcement Learning* como *Q-Learning* es su delicado ajuste para conseguir convergencia en valores de rendimiento deseados. En este estudio hemos mostrado paso a paso como el adecuado ajuste de parámetros y la aplicación de técnicas como la discretización y el *learning rate decay* mejoran los resulados y nos permiten generar distintas variantes.

También hemos mostrado las limitaciones del algoritmo *Q-Learning* con respecto al tamaño de su tabla de estados. Sin utilizar discretización, incluso problemas con pocas variables como el presentado alcanzan cardinalidades muy elevadas en el conjunto de estados, requiriendo un tiempo de exploración exponencialmente mayor. En la práctica esto equivale a no conseguir aprendizaje, ya que el tiempo que se requeriría para hacerlo es generalmente inasumible.

Continuando con los métodos de optimización presentados, en futuros trabajos se explorará el uso de aproximaciones de la función $Q(s, a)$. Una adecuada técnica de *funtion approximation* permitiría mejorar los resultados y escalar el método a problemas de mayor tamaño. Además, podría ser otra ventaja el hecho de que el algoritmo resultante fuera más general. Quizá un solo algoritmo pudiera servir para la generación de muchos tipos de agente inteligente para distintos juegos. ¿Sería posible conseguir esto sin empeorar el rendimiento de los agentes resultantes?

References

1. Atari, Inc.: Breakout. Atari, Inc. (1978), manual de usuario original del juego *Breakout*, © 1978 Atari, Inc.
2. Bellemare, M.G., Naddaf, Y., Veness, J., Bowling, M.: The arcade learning environment: An evaluation platform for general agents. Journal of Artificial Intelligence Research 47, 253–279 (jun 2013)
3. Bellemare, M., Veness, J., Bowling, M.: Investigating contingency awareness using atari 2600 games (2012), https://www.aaai.org/ocs/index.php/AAAI/AAAI12/paper/view/5162/5493
4. Birdwell, K.: The cabal: Valve's design process for creating half-life. Game Developer Magazine (1999)
5. Campbell, M., Hoane, Jr., A.J., Hsu, F.h.: Deep blue. Artif. Intell. 134(1-2), 57–83 (Jan 2002), http://dx.doi.org/10.1016/S0004-3702(01)00129-1
6. Daswani, M., Sunehag, P., Hutter, M.: Feature reinforcement learning: State of the art. In: Proc. Workshops at the 28th AAAI Conference on Artificial Intelligence: Sequential Decision Making with Big Data. pp. 2–5. AAAI Press, Quebec City, Canada (2014)
7. Genesereth, M., Love, N.: General game playing: Overview of the aaai competition. AI Magazine 26, 62–72 (2005)
8. Guo, X., Singh, S., Lee, H., Lewis, R.L., Wang, X.: Deep learning for real-time atari game play using offline monte-carlo tree search planning. In: Ghahramani, Z., Welling, M., Cortes, C., Lawrence, N.D., Weinberger, K.Q. (eds.) Advances in Neural Information Processing Systems 27, pp. 3338–3346. Curran Associates, Inc. (2014), http://papers.nips.cc/paper/5421-deep-learning-for-real-time-atari-game-play-using-offline-monte-carlo-tree-search-planning.pdf

9. Günther, M., Schiffel, S., Thielscher, M.: Factoring general games. In: in Proc. of IJCAI Workshop on General Game Playing (GIGA (2009)
10. Hausknecht, M.J., Stone, P.: Deep recurrent q-learning for partially observable mdps. CoRR abs/1507.06527 (2015), http://arxiv.org/abs/1507.06527
11. Liang, Y., Machado, M.C., Talvitie, E., Bowling, M.H.: State of the art control of atari games using shallow reinforcement learning. CoRR abs/1512.01563 (2015), http://arxiv.org/abs/1512.01563
12. Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A.A., Veness, J., Bellemare, M.G., Graves, A., Riedmiller, M., Fidjeland, A.K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., Hassabis, D.: Human-level control through deep reinforcement learning. Nature 518(7540), 529–533 (02 2015), http://dx.doi.org/10.1038/nature14236
13. Samuel, A.L.: Some studies in machine learning using the game of checkers. IBM J. Res. Dev. 3(3), 210–229 (Jul 1959), http://dx.doi.org/10.1147/rd.33.0210
14. Schiffel, S., Thielscher, M.: Fluxplayer: A successful general game player. In: AAAI '07: Proc. 22nd AAAI Conf. Artificial Intelligence. pp. 1191–1196. AAAI Press (Jul 2007)
15. Silver, D., Huang, A., Maddison, C.J., Guez, A., Sifre, L., van den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel, T., Hassabis, D.: Mastering the game of go with deep neural networks and tree search. Nature 529, 484–503 (2016), http://www.nature.com/nature/journal/v529/n7587/full/nature16961.html
16. Spronck, P., Ponsen, M., Postma, E.: Adaptive game ai with dynamic scripting. In: Machine Learning. pp. 217–248. Kluwer (2006)
17. Sutton, R.S., Barto, A.G.: Reinforcement Learning, An Introduction. MIT Press Cambridge, MA, USA © 1998 (1998)
18. Tesauro, G.: Practical issues in temporal difference learning. In: Machine Learning. pp. 257–277 (1992)
19. Tesauro, G.: Td-gammon, a self-teaching backgammon program, achieves master-level play. Neural Comput. 6(2), 215–219 (Mar 1994), http://dx.doi.org/10.1162/neco.1994.6.2.215
20. Veness, J., Bellemare, M., Hutter, M., Chua, A., Desjardins, G.: Compress and control. In: Proc. 29th AAAI Conference on Artificial Intelligence (AAAI'15). pp. 3016–3023. AAAI Press, Austin, USA (2015), http://arxiv.org/abs/1411.5326
21. Watkins, C.J.C.H.: Learning from Delayed Rewards. Ph.D. thesis, King's Collage (1989)
22. Watkins, C.J.C.H., Dayan, P.: Q-learning. Machine Learning 8(3-4), 279–292 (1992)