Deep Q Learning aplicado a Mario Kart: super circuit

Andrés Felipe Cruz Salinas[1] y Daniel Ricardo Castro Alvarado[1]

***Abstract*— El presente trabajo expone el entrenamiento de un agente para jugar Mario Kart: Super Circuit (de Game Boy Advance) utilizando técnicas de Deep Reinforcement Learning. Se muestra una visión general de Deep Q Learning, el modelo que se realizó para entrenar el agente y los resultados del entrenamiento.**

I. INTRODUCCIÓN

El aprendizaje por refuerzo (Reinforcement Learning) es un área de la inteligencia artificial inspirada en la psicología conductista, en donde se busca que un agente que puede percibir el estado de un ambiente y ejecutar determinadas acciones, escoja qué acción debe realizar para maximizar una recompensa acumulada definida en el ambiente. En el aprendizaje por refuerzo, se tiene un conjunto bien definido de estados en los cuales el agente puede estar, así como una función de transición que indica en qué momento se pasa de un estado a otro debido a la ejecución de alguna acción. Así mismo, se determinan una serie de reglas que indican la recompensa inmediata debido a la transición de un estado a otro.

Q Learning es una técnica de aprendizaje por refuerzo que no depende del ambiente en el cual el agente está aprendiendo. Se busca encontrar una política de comportamiento para cualquier proceso (finito) de decisión de Markov. Para encontrar la política óptima se aprende una función de acción-valor Q^* que determina la recompensa acumulada por una acción del agente desde el estado actual hasta el estado final. La política de comportamiento π óptima se construye escogiendo la acción que maximiza el valor de la función Q^*. Adicionalmente, Q Learning es una técnica que maneja ambientes estocásticos mediante un factor de descuento sobre la recompensa de los estados futuros. Dicho factor controla el nivel de incertidumbre al que el agente se enfrenta en el futuro. Recientemente ha surgido una técnica que combina el uso de Q learning con redes neuronales (Deep Reinforcement Learning) para aproximar la función Q^*, lo cual ha dado resultados notables debido a la flexibilidad que otorgan las redes neuronales dada su naturaleza no lineal.

El objetivo del presente trabajo es utilizar Deep Reinforcement Learning para entrenar un agente que juegue *Mario Kart: Super Circuit*, un videojuego por Nintendo lanzado en el 2001 para *Gameboy Advance*. Mario Kart es un juego de carreras, en donde los jugadores manejan un carro y su objetivo es completar 3 vueltas en una pista y llegar en el primer lugar. Existen trabajos previos de Deep Reinforcement Learning para aprender a manejar jugar videojuegos de vehículos, como en [3] y [6].

[1]Universidad Nacional de Colombia, Ingeniería de Sistemas y Computación

II. DEEP Q LEARNING

En escenarios de la vida real, la función Q suele no estar definida de manera explícita, ya sea porque existe una cantidad exponencial de estados del juego, o porque es compleja de calcular analíticamente. Deep Q Learning utiliza los avances en Deep Learning para aproximar funciones no lineales sin necesidad de realizar extracción de características. En el campo de procesamiento de imágenes, Deep Learning ha tenido un profundo impacto debido a la reducción dramática de el error en tareas de clasificación. Esto debido a que las imágenes suelen tener una representación implícita de características de alto a bajo nivel, las cuales son capturadas por las redes neuronales con varias capas de profundidad. Estas redes son entrenadas utilizando algoritmos que utilizan el gradiente de la función de pérdida asociada a la tarea de clasificación o regresión.

En la definición clásica de Q learning, se resuelven tareas en donde el agente interactúa con el ambiente mediante una secuencia de estados, estos se acceden a través acciones que proporcionan un reward o recompensa dependiendo del estado al que lleve dicha acción. El objetivo del agente es escoger la acción que maximice el reward futuro acumulado. Según [4] esta función se define formalmente como:

$$Q^*(s,a) = \max_{\pi} \mathbf{E}[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots | s_t = s, a_t = a, \pi]$$

La función Q^* es la función de acción-valor óptima la cuál es la suma máxima de recompensas r_t descontada por γ en cada paso de tiempo t. Todo esto alcanzado mediante una política de comportamiento $\pi = P(a|s)$ después de realizar una observación s y realizar una acción a.

En Deep Q learning se utiliza una red neuronal para aproximar la función Q^*. Sin embargo, cuando se utilizan aproximadores no lineales como una red neuronal, el entrenamiento es inestable o incluso puede diverger. La inestabilidad se puede dar por correlaciones presentes entre los estados consecutivos, grandes cambios en la política de comportamiento al realizar pequeñas actualizaciones en Q y la correlación entre las parejas de acciones-valores y los valores objetivo $r + \gamma \max_{a'} Q(s', a')$ [4]. Se define la función Q parametrizada con θ_i (los pesos de la red neuronal en la iteración i) como $Q(s, a; \theta_i)$.

Las experiencias del agente se definen como la tupla $e_t = (s_t, a_t, r_t, s_{t+1})$ en cada tiempo t. Estas experiencias se guardan en una memoria D de tamaño fijo. Durante el entrenamiento, se aplican actualizaciones con un minibatch conformado de experiencias extraídas de D muestreadas uniformemente. La regla para actualizar en la iteración i utiliza la siguiente función de pérdida [4]:

$$L_i(\theta_i) = \mathbf{E}_{(s,a,r,s')\sim U(D)}[(r+\gamma \max_{a'} Q(s', a'; \theta_i^-) - Q(s, a; \theta_i))^2]$$

En donde γ es el factor de descuento en el reward futuro, θ_i son los parámetros de la red neuronal en el iteración i y θ_i^- son los parametros usados para calcular el objetivo en la iteración i. Los parámetros θ_i^- se actualizan solamente con los parámetros θ_i cada C pasos y se mantienen fijos entre actualizaciones individuales. Esto con el objetivo de estabilizar el entrenamiento.

La función óptima de acción-valor obedece una identidad conocida como la ecuación de Bellman. Dicha identidad explota la subestructura óptima de los estados, en donde si se conoce el valor óptimo $Q^*(s', a')$ de la secuencia s' en el siguiente instante de tiempo, para todas las acciones a', entonces, la estrategia óptima es seleccionar la acción a' que maximiza el valor esperado de $r + \gamma Q^*(s', a')$:

$$Q^*(s', a') = \mathbf{E}_{s'}[r + \gamma \max_{a'} Q^*(s', a')|s, a]$$

La ecuación de Bellman se utiliza para estimar la función de acción-valor con actualizaciones iterativas. Esta clase de algoritmos iterativos convergen a la función óptima cuando la cantidad de iteraciones tiende al infinito. En la práctica, no es factible realizar la estimación para cada secuencia por separado, por lo que se utilizan aproximadores de funciones como las redes neuronales ya descritas. Con la función de pérdida definida y la regla de actualización dada por las ecuaciones de Bellman, el algoritmo para entrenar un agente en cualquier ambiente en donde se proporcionen los estados, acciones y recompensas se muestra en 1.

Se define ϕ como una función que realiza el preprocessing adecuado al ambiente que se está trabajando. Adicionalmente se tiene un parámetro ϵ que decrece en el tiempo y sirve para realizar exploración de los estados del ambiente mediante movimientos aleatorios al inicio del entrenamiento.

Algorithm 1 Deep Q Learning con experience replay

Inicializar la replay memory D con capacidad N.
Inicializar la función Q con pesos aleatorios θ.
Inicializar la función $\hat{Q}$ con pesos $\theta^- = \theta$.
for episode = 1, M **do**
  for $t = 1$, T **do**
    Con probabilidad ϵ escoger una acción aleatoria a_t en otro caso, seleccionar $a_t = argmax_a Q(\phi(s_t), a; \theta)$
    Ejecutar la acción a_t en el emulador y observar la recompensa r_t y la imagen x_{t+1}
    Asignar $s_{t+1} = s_t, a_t, x_{t+1}$ y preprocesar $\phi_{t+1} = \phi(s_{t+1})$
    Almacenar la transición $(\phi, a_t, r_t, \phi_{t+1})$ en D
    Generar un minibatch de transiciones $(\phi, a_t, r_t, \phi_{t+1})$ haciendo un muestreo en D
    if el episodio termina en el paso $j + 1$ **then**
      $y_j = r_j$
    else
      $y_j = r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-)$
    end if
    Realizar un paso de gradiente descendiente en $(y - Q(\phi_j, a_j; \theta))^2$ con respecto a los parámetros θ.
    Cada C pasos asignar $\hat{Q} = Q$
  end for
end for

III. INTERACCIÓN CON EL JUEGO

Para emular el juego se utilizó el emulador Bizhawk [5], que cuenta con un API en LUA para acceder al estado interno programáticamente y para poder jugar desde allí. El entrenamiento fue implementado en Python con Keras [1] para incluir las redes neuronales. La arquitectura general se puede apreciar en la figura 1.

Fig. 1: Arquitectura general de la interacción del emulador y el agente

Para realizar la comunicación entre LUA y python se implementó un servidor que recibe solicitudes HTTP con el estado del juego y devuelve la acción a ejecutar. Debido a que la pantalla del Game Boy Advance no tiene una resolución muy grande, no hay mayor retraso al enviar y capturar las imágenes del juego. El cuello de botella es el entrenamiento de la red neuronal, y para ello se utilizó una GPU Nvdia GTX 1070, que ayudó a lograr un entrenamiento en tiempo real.

IV. MODELO DEL JUEGO

Para llevar a cabo un entrenamiento exitoso, es necesario obtener cierta información del juego que permita, entre otros, determinar una función de recompensa adecuada para cada acción y marcar el final de cada episodio.

Entre la información que usamos en el modelo se encuentra:

- La vuelta actual
- La posición en la pista
- El tiempo total de carrera
- El estado de la carrera (finalizada o en curso)
- El número de fotogramas en los que el jugador ha estado por fuera de la pista

La vuelta actual y la posición en la pista permiten determinar el progreso del jugador en la carrera. Esto, combinado con el número de fotogramas en los que el jugador se ha salido de la pista, permite estimar un *reward* o recompensa para cada acción. El estado de la carrera permite marcar el final de cada episodio, y el tiempo total de carrera sirve como base para crear *checkpoints*, que explicaremos más adelante.

Fig. 2: El minimapa en Mario Kart: Super Circuit. El punto blanco representa la posición del jugador

Esta información, a excepción de la posición en la pista, se extrae directamente de la RAM de la consola emulada.

A. La posición en la pista y el progreso en la carrera

Debido a que no encontramos la dirección en memoria donde el juego almacena la posición del jugador, realizamos una estimación a partir de la ubicación del jugador en el minimapa (ver Figura 2), que sí es fácilmente accesible.

El minimapa es una imagen de 64×64 pixeles, que procesamos ejecutando una búsqueda en amplitud (BFS) para encontrar los caminos más cortos a todos los puntos de la pista. Para que esto nos permita hallar el progreso en la pista, es necesario que la búsqueda comience en la línea de meta. La búsqueda se realiza sobre todos los nodos que tienen el mismo color que el nodo fuente, para así excluir los caminos más cortos que se salen de la pista. Además, la fila que está justo detrás de la meta se pinta de otro color, con el fin de evitar que el algoritmo busque hacia atrás.

La distancia encontrada por la búsqueda en amplitud permite estimar un progreso en la pista: basta con dividir la distancia hasta la posición en la que está el jugador, por la distancia hasta el último nodo de la pista (que, por definición, es máxima).

Para los pixeles que no hacen parte de la pista, como el pasto, usamos el progreso del nodo dentro de la pista con menor distancia Euclidiana.

B. Checkpoints

Por la complejidad del juego, es muy poco probable que un agente pase una pista si sólo toma acciones aleatorias. Esto, a su vez, limita la capacidad de exploración, pues el agente tenderá a explorar solamente los estados a los que llega fácilmente. Por esta razón, nuestro modelo usa *checkpoints*. Cada vez que el jugador avanza cierto porcentaje de la carrera, se crea un nuevo checkpoint. Si el jugador pasa más de cierto tiempo sin llegar al siguiente checkpoint, el entorno restaurará el último checkpoint, dándole la oportunidad al agente de intentar avanzar nuevamente. Así, los checkpoints pueden evitar que el agente se quede estancado en una sección de la pista, o que se devuelva y deshaga cualquier progreso previo.

C. El puntaje en Mario Kart: Super Circuit

Mario Kart mantiene un puntaje interno, que es, en general, invisible para el jugador. El juego no muestra el puntaje durante la carrera. En su lugar, asigna una categoría a los jugadores destacados al final de cada carrera (e.g tres estrellas), teniendo en cuenta información como la posición final, el tiempo total, el uso de freno, la conducción por la pista, entre otros. La fórmula exacta está por fuera de los alcances de este artículo, pero ha sido descifrada (casi completamente), por otros autores [2].

Aunque la fórmula se calcula sólo al final de la carrera, es posible estimar un puntaje similar en cada instante, a partir de información extraída de la memoria RAM de la consola en el emulador. Inicialmente exploramos la posibilidad de usar este puntaje para calcular la recompensa, pero dada la cantidad de variables involucradas decidimos usar una función más simple.

D. Función de recompensa o reward

La función de recompensa que utilizamos es similar a la planteada por [6]. El agente es castigado cuando su velocidad es cero (o está retrocediendo) o cuando transita por el pasto, y premiado cuando avanza en la carrera siguiendo la pista.

Como no tenemos acceso a la velocidad actual en el juego, estimamos la velocidad del jugador como la diferencia entre progreso en la pista por unidad de tiempo.

La fórmula de la función de recompensa, entonces, está dada por:

$$R = \begin{cases} -1 & \text{timeout} \\ -1 & \text{speed} \leq 0 \\ -0.8 & \text{outside track} \\ \text{speed} & \text{otherwise} \end{cases}$$

E. Acciones

Limitamos las combinaciones de botones que puede realizar el agente a algunas particulares que tienen sentido para el juego:

- Ningún botón
- Acelerar (A)
- Acelerar y derecha (A, flecha derecha)
- Acelerar e izquierda (A, flecha izquierda)
- Frenar / Retroceder (B, flecha abajo)
- Derrapar hacia la derecha (A, flecha derecha, R)
- Derrapar hacia la izquierda (A, flecha izquierda, R)
- Acelerar y usar poder (A, L)
- Acelerar y derecha, poder (A, flecha derecha, L)
- Acelerar e izquierda, poder (A, flecha izquierda, L)
- Frenar / Retroceder, poder (B, flecha abajo, L)
- Derrapar hacia la derecha, poder (A, flecha derecha, R, L)
- Derrapar hacia la izquierda, poder (A, flecha izquierda, R, L)

V. ENTRENAMIENTO

La arquitectura de la red neuronal para Deep Q Learning fue la original planteada en [4], con la modificación de la

TABLE I: Arquitectura de la red neuronal para Deep Q Learning

Layer	Input	Filter size	Stride	Num filters	Activation
conv1	$84 \times 84 \times 4$	8×8	4	32	ReLU
conv2	$20 \times 20 \times 32$	4×4	2	64	ReLU
conv3	$9 \times 9 \times 64$	3×3	1	64	ReLU
dense4	$7 \times 7 \times 64$			512	ReLU
dense5	512			13	Linear

última capa para que tenga el número esperado de salidas para Mario Kart:

TABLE II: Hiperparámetros del modelo

Frames to stack together	4
Minibatch size	32
Replay memory size	80 000
Discount factor	0.95
Learning rate	0.00025
Gradient momentum	0.95
Squared gradient momentum	0.95
Minimum squared gradient	0.01
Initial exploration rate	1
Final exploration rate	0.1
Exploration frames	1 000 000
Replay memory start size	50 000
Target network update frequency	5 000

Entrenamos el modelo por 1387 episodios (2'037.590 pasos) utilizando Adam como optimizador. Los hiperparámetros del modelo se muestran en la tabla II.

Todo el entrenamiento lo realizamos en una sola pista. Una vez el modelo llegó al episodio 456, sospechamos que estaba explotando los checkpoints, y los desactivamos. En el episodio 878 encontramos un error relacionado con el tiempo límite, y lo arreglamos. En ambos casos, el entrenamiento se continuó restableciendo la tasa de exploración a 0.5.

VI. RESULTADOS

La evolución del entrenamiento se muestra a continuación. En la figura 3 se ve el promedio por episodio del error calculado en cada actualización de los pesos en la red neuronal, allí se puede ver como converge rápidamente la función de perdida y solo sufre algunas variaciones menores a lo largo del tiempo.

Por otro lado, en la figura 4 se ve cómo la función de recompensa evoluciona a través del tiempo. La función parece bastante ruidosa, pero es evidente el progreso que tiene desde el primer episodio. Es interesante observar que tanto en el episodio en que se retiran los checkpoints como en el que se arregla el timeout, la función de reward baja y posteriormente vuelve a subir hasta más o menos el mismo punto. Dicho comportamiento es esperado pues se realizan cambios importantes en el modelo. Por otro lado, la forma ruidosa que tiene en la función de recompensa es esperada, y es similar a los resultados reportados en [4].

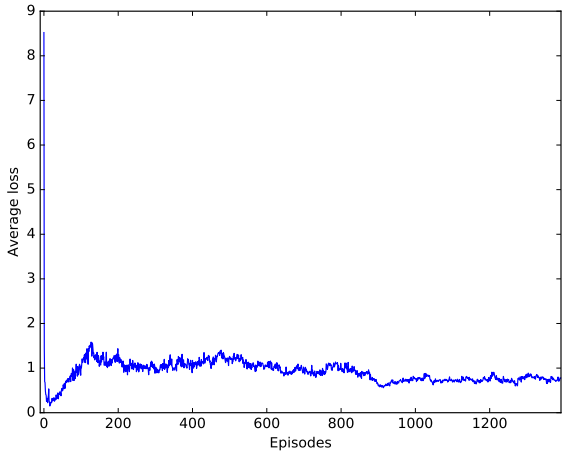


Fig. 3: Función de pérdida promedio por episodio

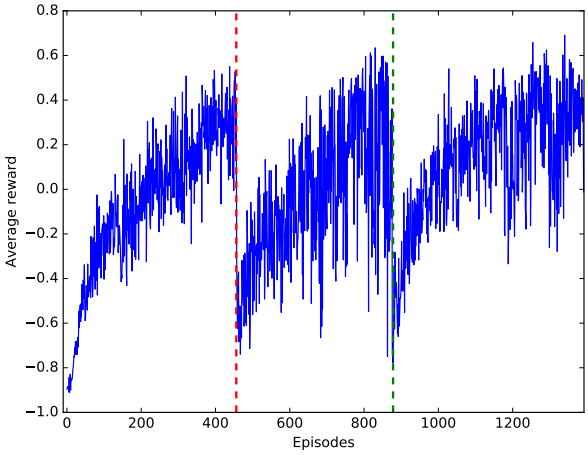
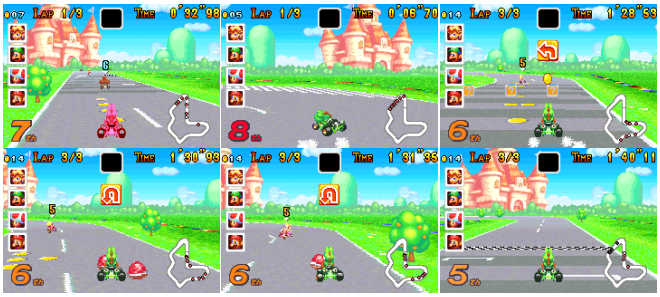


Fig. 4: Reward promedio por episodio obtenido por el agente. La línea verde indica el momento en que se retiraron los checkpoints y la roja el momento en que se realizó una correción en el timeout.



(a) El agente entrenado avanzando al quinto lugar y terminando el episodio

En la figura 5a se observan algunas capturas del agente entrenado jugando. En la primera captura, se observa cómo el agente (con un poder de estrella activado) va sobre la pista y recoge monedas en sobre la carretera. Este comportamiento

no fue premiado explícitamente, pero el juego aumenta la velocidad del jugador cuando alcanza cierta cantidad de monedas. En la segunda captura, se observa cómo el agente toma una curva, derrapando y girando hacia la izquierda. Es interesante observar que en algunas curvas el agente tiende a frenar, lo cual se deriva de que en el entrenamiento, probablemente ir con velocidad alta, se traduce en un reward negativo en estados futuros. Finalmente, en las últimas cuatro imágenes, aparece el agente usando los poderes que brinda el juego y los usa para retrasar a un contrincante y así avanzar del sexto al quinto lugar, para finalizar la carrera en dicha posición. Es de notar que la quinta posición fue el mejor lugar en el que el agente logró quedar con el modelo entrenado.

VII. CONCLUSIONES

Luego de realizar toda la implementación, se concluye que una de las dificultades más grandes es realizar la interacción del juego hacia el entrenamiento. Se tuvieron que hacer varias optimizaciones sobre la marcha para lograr que el todo el proceso fuera en tiempo real. Aún con eso, el entrenamiento toma bastante tiempo. Como se mencionó anteriormente, se utilizó una GPU NVIDIA GTX 1070 en un computador de 16GB de RAM y tomó un poco más de una semana.

Debido a que el entrenamiento se realizó sobre una pista, el agente hace *overfitting* y no es capaz de generalizar con otras pistas. Esto es esperado, pues los ejemplos de entrenamiento entre pistas son bastante distintos. Además de ello, nuevos estados aparecen (e.g el jugador puede salirse de la pista, caer en el agua o desde el cielo). Trabajo futuro incluye aplicar *transfer learning* para entrenar en un periodo de tiempo más corto el modelo con nuevas pistas.

A pesar de que se entrenó por una buena cantidad de tiempo, el agente no logró llegar en la primera posición. Creemos que esto se debe a que el reward que se le otorga al agente no premia la posición, aunque la velocidad implícitamente puede significar un aumento en la posición, debido al entrenamiento el agente castiga el uso de alta velocidad para evitar estrellarse.

References are important to the reader; therefore, each citation must be complete and correct. If at all possible, references should be commonly available publications.

REFERENCES

[1] François Chollet. Keras. `https://github.com/fchollet/keras`, 2015.

[2] GameFAQs. Mario kart: Super circuit rank calculation guide — gamefaqs, 2013. [En linea; Accedida el 30 de noviembre de 2016].

[3] Pedro Lopes. Deep reinforcement learning: Playing a racing game, 2016. [En linea; Accedida el 30 de noviembre de 2016].

[4] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[5] TAS Videos. Bizhawk emulator, 2016.

[6] April Yu, Raphael Palefsky-Smith, and Rishi Bedi. Deep reinforcement learning for simulated autonomous vehicle control.