Aprendizaje Mecánico
Proyecto: Deep Q Learning para IA que juega Tetris
Por: Daniel Ramírez, B45675
Celeste Balladares, B70873

A. Descripción del Juego Tetris

Tetris consiste en un juego cuyo espacio de juego consiste en un tablero (en nuestro caso de 20 bloques de altura y 10 bloques de ancho). En este juego se dejan caer piezas con distintas formas, generadas de forma aleatoria, desde la parte superior del tablero, estas al caer colapsan ya sea con la parte inferior del tablero o con otras piezas, tal y como podrá apreciarse en la figura 1.

El objetivo del juego es obtener el score más alto posible y esto se logra de dos formas:

- Logrando que una pieza caiga y colapse con el suelo u otra pieza sin que al pasar esto colapse también con el techo.
- Lograr eliminar líneas horizontales con bloques de colores, lo cual eliminará todos los bloques de esa fila y dejará caer los bloques arriba una fila por cada fila eliminada.

El jugador tiene en su control para lograr el score más alto posible el mover las piezas a la izquierda o derecha, puede también rotar la figura cuantas veces desee a favor o en contra de las manecillas del reloj y de estar seguro que tiene la posición de rotación de figura que desea y está en los ejes (x, y) que desea, este puede acelerar su caída.

Un jugador pierde en el instante que una pieza que cae, en el momento de colapsar con otra pieza, está también colapsa con la parte superior del tablero. En este caso, el jugador conocerá el score final obtenido en la partida y la cantidad de líneas que logró destruir.

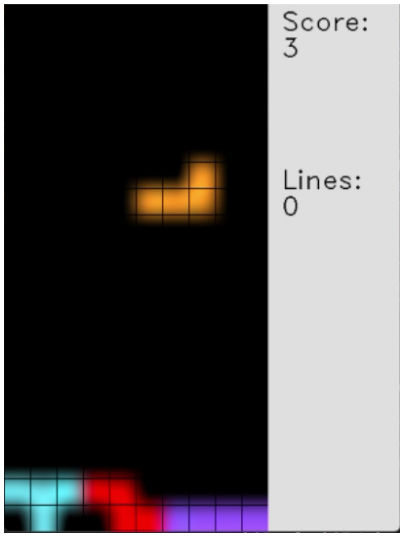

Figura 1. Ejemplo de cómo luce el juego tetris

B. Tipo de Algoritmo de Aprendizaje Mecánico

El algoritmo de aprendizaje mecánico utilizado para generar la inteligencia artificial que juega el tetris fue el de Deep Q Learning. Este es un algoritmo compuesto por dos algoritmos de aprendizaje mecánico. Por un lado un algoritmo de Q Learning que busca aprender qué acción tomar bajo qué circunstancias en un espacio explorado. Por otro lado tenemos a un algoritmo de Deep Learning que intenta identificar la mejor combinación de pesos para cada neurona en la red neuronal y así ayudar al algoritmo de Q Learning a tomar su decisión.

El género de juego en este caso consiste en uno de tipo IA vs Entorno, en el que el modelo intenta aprender del entorno con respecto a las decisiones que toma y la premiación o penalización de tal acción, lo que debería hacer para jugar tetris de la mejor forma posible.

C. Características del Agente

a. Entradas:

```
epochs = 10                    ## total de épocas por correr
batch_memory_size = 3000       ## Cantidad total de memoria
batch_size = 512               ##Tamaño de memoria  para entrenamiento
num_decay_epochs = 2000        ## decay_epochs
model_save_interval = 10       ## cada cuanto se salvará un modelo
lr = 1e-3                      ## rango tasa de aprendizaje
gamma = 0.99                   ##  gamma
epsilon = 1                    ##  epsilon
decay = 1e-3                   ##  decaimiento
```

b. Lógica agente:

```
Loop épocas:
    Loop hasta que se acabe el juego:
        Obtiene los siguientes estados del tablero
        Escoge acción
        Ejecuta acción
        Almacena en memoria
    Aprende
```

Figura 2. Descripción de pseudo código del agente.

Estado del juego:

El agente puede solicitarle al ambiente, el estado del juego, el cual consiste en un arreglo compuesto de los siguientes 4 valores:

- ➔ deleted lines: cantidad de filas eliminadas en el tablero
- ➔ holes: cantidad de huecos encontrados en el tablero
- ➔ bumpiness: suma de las diferencias de alturas de una columna y la siguiente
- ➔ heights: suma de todas las alturas

Cada figura en el tablero, va generar un estado dependiendo de su rotación y su posición en el eje x. Dependiendo de la forma de la figura, hay un número establecido de posibles rotaciones y una cantidad máxima de libertad para poder desplazarse en el eje *x*. Cada vez que el agente ejecuta un step de la simulación, obtiene todos los posibles estados de cada par (posición eje *x*, # rotación), como se puede visualizar en el siguiente ejemplo.

{(0, 0): tensor([0., 186., 3., 5.]),
(1, 0): tensor([0., 186., 4., 5.]),
(2, 0): tensor([0., 186., 4., 5.]),
(3, 0): tensor([0., 186., 4., 5.]),
((0, 1): tensor([0., 186., 3., 5.]),
(1, 1): tensor([0., 186., 6., 5.]),
(2, 1): tensor([0., 186., 6., 5.]),
(3, 1): tensor([0., 186., 6., 5.]),
(4, 1): tensor([0., 186., 6., 5.]), ...}

Escoger acción:

Una acción corresponde a la tupla (posición eje *x* , # rotación) de la figura actual, que se relaciona con cada posible estado del juego. Para saber cual es la mejor acción a elegir, se utiliza el parámetro épsilon, que disminuye conforme aumentan las iteraciones, de modo que el agente elige a veces una acción aleatoria en lugar de siempre escoger la mejor acción encontrada hasta el momento, lo que permite evitar quedarse en óptimos locales.

Ejecutar acción:

El agente puede solicitarle al ambiente ejecutar una acción. Este método recibe una acción a ejecutar con la posición en el eje x donde se debe colocar la figura y el número de rotaciones que se le tiene que realizar. Una vez que la figura haya rotado y se encuentre en la posición *x*, empieza a descender en el tablero, moviendo el eje *y* hasta que colisione con el borde o con alguna otra figura. Después de ejecutar la acción, devuelve una recompensa que se basa en lo siguiente:

- ➔ Cada figura colocada en el tablero obtiene 1 punto.
- ➔ Al borrar líneas, la puntuación dada es: 1 + (líneas borradas)^2 * ancho del tablero.
- ➔ Perder el juego resta 2 puntos. Se pierde el juego cuando la figura se desborda del techo del tablero.

Almacena en memoria:

Cada vez que el agente ejecuta un step de la simulación, utiliza un memoria para guardar los siguientes valores:

- ➔ estado actual
- ➔ Recompensa
- ➔ siguiente estado
- ➔ Is game_over

El objetivo es que el agente pueda entrenarse a sí mismo, usando una red neuronal, con una muestra aleatoria de la memoria. El parámetro batch_size especifica el tamaño de la muestra que se tomará de la memoria para cada entrenamiento. Algo importante es que el agente no va a empezar a entrenarse hasta que la memoria se llene con los primeros movimientos.

Aprende:

Una vez que se recopilarán los estados de todos los movimientos posibles, cada estado se insertará en la red neuronal, para predecir la puntuación obtenida. Se tomará la acción cuyo estado produzca el mayor valor.

Estructura red neuronal:

Nuestra solución propuesta es usando una Deep Q-Network, que recibe como entrada un conjunto de estados del tablero y da como salida la estimación de la recompensa para cada estado en la entrada. La red está compuesta por cuatro capas lineales, donde la primera recibe un arreglo de 4 elementos, que corresponde a los 4 valores de un estado. Posteriormente la segunda capa consta de 64 muestras que posteriormente se reduce a 32 neuronas en la tercera capa. Finalmente la última capa sólo tiene un neurona de salida con el valor estimado de la recompensa.

D. Desafíos Encontrados

Ninguno de los dos integrantes contábamos con GPU y nuestras computadoras no eran muy potentes. Cuando ya estaba todo ya casi listo y era cuestión de algunos cambios y ajustes de parámetros notamos que la computadora se calentó mucho y el ventilador se puso a trabajar, sin embargo debíamos continuar y el programa corrió por alrededor de 6 horas con 2000 iteraciones.

Nuestros archivos train.py, test.py y agente.py cuentan con la opción de usar gpu de existir está a disposición por medio de cuda. Sin embargo al ninguno contar con GPU, no estábamos seguros si esta implementación funcionaba, pero necesitábamos correr la versión final con 2500 iteraciones. Al solicitar a un amigo el poder usar su computadora para poder correr nuestro modelo en su equipo, nos dimos cuenta que había una línea de código que tenía conflicto con la asignación de cuda.

E. Solución a los Desafíos

En cuanto al primer desafío, nos tocó asegurarnos que teníamos todo el código listo, procurando no tener que hacer cambios para poder pedir prestado alguna computadora que contará con GPU.

En cuanto al segundo desafío, tuvimos que buscar documentación de cuda y de personas con errores similares. Después de varias páginas y de leer varias sugerencias de solución, por fin dimos con una que solucionaba el problema que se trataba de que a una instancia creada en la tupla de state_batch (para ser más específicos al state) se le debía también aplicar el to(device).

F. Resultados Obtenidos

a. Métricas obtenidas del entrenamiento:

Para la corrida del train.py se generó un dataframe que contiene el número de época, las líneas totales destruídas en la época y el score alcanzado en esta.

Dicho dataframe se usó para generar el archivo train_metrics.csv, como se observa en la figura 3, que se encuentra en la carpeta generated_metrics.

	epoch	lines destroyed	score
0	1	0	16
1	2	0	19
2	3	0	21
3	4	0	19
4	5	0	19
5	6	0	23
6	7	0	14
7	8	0	17
8	9	0	20
9	10	0	13
10	11	0	13
11	12	0	20
12	13	0	11
13	14	0	17
14	15	0	13
15	16	0	11
16	17	0	13
17	18	1	26
18	19	0	16

Figura 3. Archivo de métricas por época del train

Con el dataframe se realizó un gráfico del comportamiento del score con respecto a la época, como se puede apreciar en la figura 4.

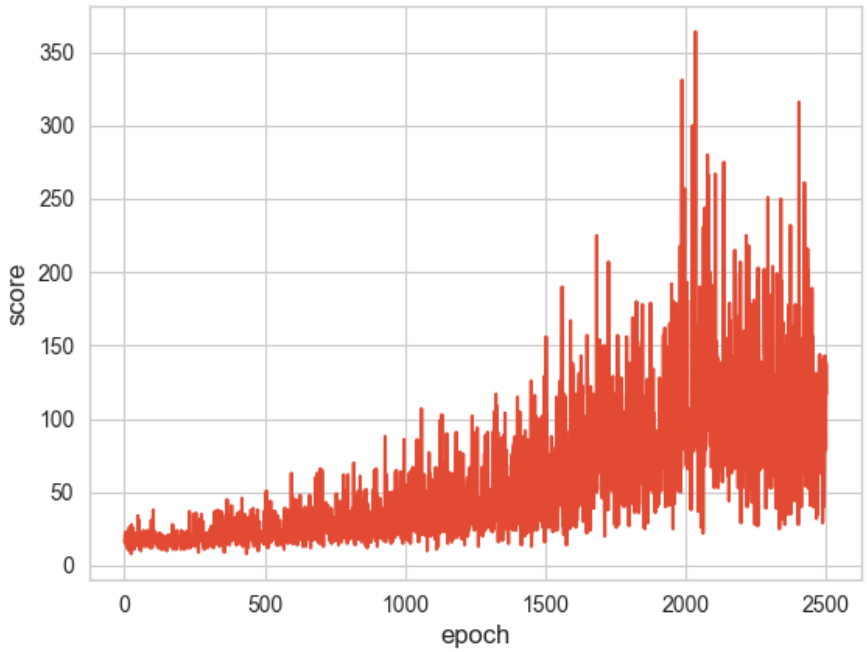

Figura 4. Scores con respecto a la época.

Dado que el batch de aprendizaje una vez lleno, hace un pop del primer valor de su lista ingresado (de comportamiento FIFO, con un tamaño de memoria máximo), constantemente se actualizan los estados, decisiones tomadas y la recompensa obtenida. Sin embargo parece que después de la época 2000, los scores obtenidos empezaron a bajar, esto probablemente se deba por la falta de una red neural Target, que podría aportar mayor estabilidad al modelo.

b. Métricas obtenidas del testeo:

Una vez creados los modelos de la etapa de entrenamiento, se generaron modelos entrenados, en este caso cada 500 épocas, siendo los modelo tetris_500, tetris_1000, tetris_1500, tetris_2000 y tetris_2500.

Para cada modelo se recolectaron los valores de las líneas destruídas y del score obtenido, con estos datos se creó un dataframe y con el que se creó el archivo test_metrics.csv.

En la figura 5 se puede apreciar los datos obtenidos para las primeras corridas.

	model	run	lines destroyed	score
0	tetris_500	0	5	93
1	tetris_500	1	7	115
2	tetris_500	2	9	144
3	tetris_500	3	8	130
4	tetris_500	4	11	167
5	tetris_500	5	11	167
6	tetris_500	6	6	108
7	tetris_500	7	8	131
8	tetris_500	8	25	343
9	tetris_500	9	8	130
10	tetris_500	10	12	181
11	tetris_500	11	18	257
12	tetris_500	12	10	155
13	tetris_500	13	5	94
14	tetris_500	14	7	120
15	tetris_1000	0	9	143
16	tetris_1000	1	12	181
17	tetris_1000	2	6	105

Figura 5. Archivo de métricas obtenidas en el test.

Finalmente se graficaron los resultados obtenidos por cada modelo en un gráfico de cajas, como se teien en la figura 6 donde se aprecia que el modelo entrenado que presentó mejor rendimiento en los scores fue el que fue creado en la época 1000. Por otro lado, puede observarse que hubo un sobre entrenamiento en los modelos de las épocas 1500, 2000 y 2500. De tenga la oportunidad de correr el train.py, podrá apreciar que las últimas épocas suelen escoger cierta posición en las piezas con una frecuencia muy alta, teniendo posiciones que se usan muy esporádicamente.

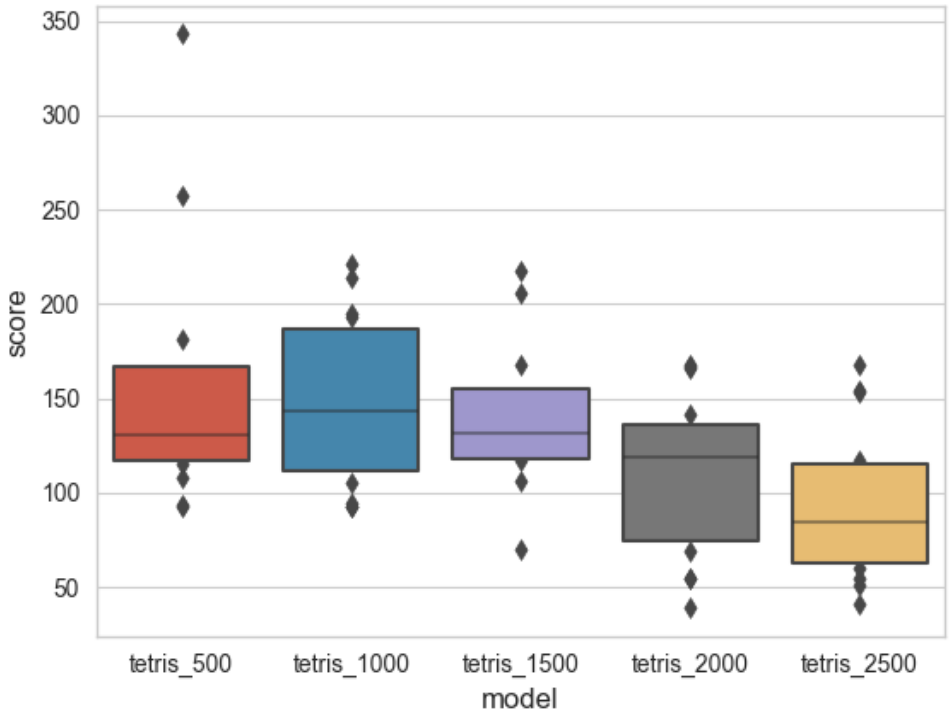

Figura 6. Scores en relación al modelo creado.

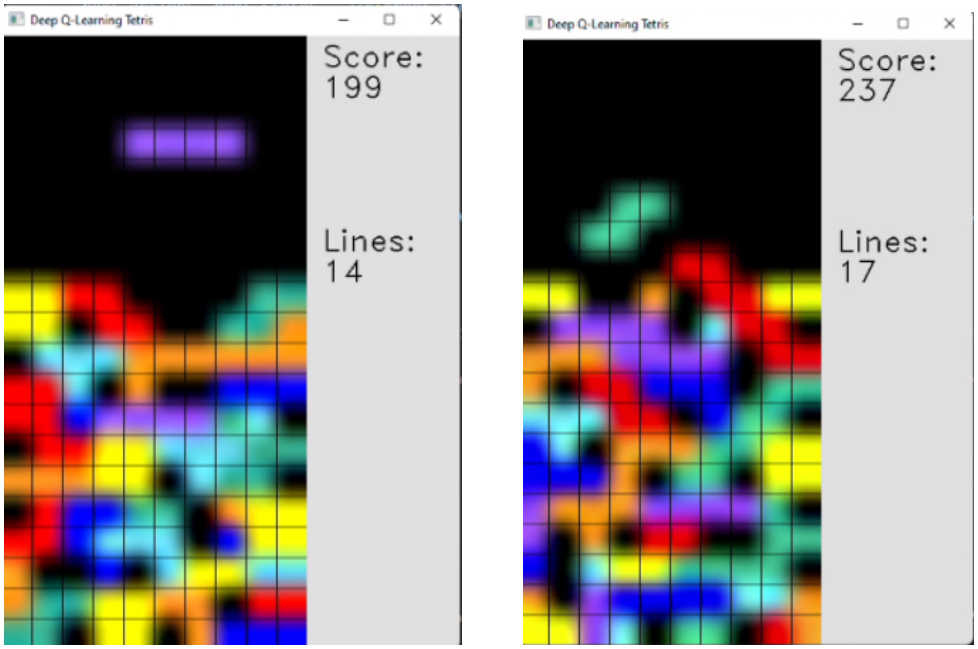

Figura 7. Ejemplos de scores altos en partidas.