UNIVERSIDAD DE CONCEPCIÓN

FACULTAD DE INGENIERÍA

DEPARTAMENTO DE INFORMÁTICA

Proyecto:
IA para Juegos

Informe final

Integrantes: - Pedro Palacios Rossi.

2021239140.

- Diego Venegas Anabalón.

2021433473.

Profesor: Julio Godoy Erasmo.

Fecha de Entrega: Domingo, 17 de diciembre del 2023.

1. Introducción

En este informe, se presentarán avances, discusiones y resultados que obtuvimos como grupo al implementar Double Deep Q-Learning en el clásico videojuego de Atari 2600, Breakout.

Primero, mencionaremos bastantes problemas que se nos presentaron a medida que íbamos avanzando en nuestro proyecto, cambios de planes y factores que no tuvimos en consideración. Seguido, explicaremos la implementación usada, mostraremos las especificaciones del equipo en las que se hizo el entrenamiento y también las herramientas y librerías utilizadas. Luego, se presentarán los parámetros utilizados para los diferentes entrenamientos junto a los resultados obtenidos en cada uno de ellos. Y, por último, pero aun importante, daremos nuestras conclusiones acerca de este proyecto.

Algo a tener muy en consideración es que, en este proyecto, no buscamos descubrir algo o lograr algún avance importante en el área, puesto que es nuestra introducción al mundo de las inteligencias artificiales, por lo que tomaremos esta instancia para familiarizarnos y aprender sobre cómo funcionan algunas cosas. Por lo mismo, nos planteamos los siguientes objetivos:

1.1. Objetivos:

- Mejorar nuestras habilidades de programación y comprensión sobre Python.
- Aprender sobre métodos de IA en la práctica.
- Conocer las dificultades que enfrentan algunos métodos.
- Conocer las dificultados que nos ofrece algún juego de Atari 2600.
- Resolver un juego de Atari 2600 mediante algún método de IA.
- Experimentar con los parámetros que utiliza Double Deep Q-Learning.
- Comprender el comportamiento de un modelo de Double Deep Q-Learning frente a nuestro juego seleccionado.

2. Contexto

En un principio, el objetivo de este proyecto era resolver algún juego de Atari 2600 de manera sencilla usando Q-learning, sin embargo, después de analizar múltiples opciones y percatarnos de que los juegos menos complejos cuentan con una cantidad de estados mayor a 2^{6200}, caímos en que no es posible utilizar este método, así que la alternativa por la que nos decidimos es utilizar una red neuronal para poder manejar tal cantidad de estados.

Deep Q learning consiste en utilizar aprendizaje reforzado para simular la función Q de Q-learning. El modelo de red neuronal puede recibir cualquier estado dentro del enorme espacio de estados posibles y aproximar el resultado de la función Q para este.

De entre los principales videojuegos de Atari 2600 que analizamos, se encuentra BankHeist, el cual nos habíamos decidido a resolver en un principio, sin embargo, después de comprender la complejidad de este y ver que en el paper de Deepmind sobre DQL, se mostraba como un caso de bajo desempeño, decidimos optar por otra alternativa.

El videojuego que finalmente vamos a resolver es Breakout, ya que es relativamente simple en cuanto a mecánicas, controles y visualización; Sin embargo, puede ser complejo de dominar, lo que se conoce como una “skill gap” y es interesante para el entrenamiento de una inteligencia artificial.

3. Desarrollo

Para la realización de este proyecto, utilizamos una implementación ya realizada, que fue encontrada en el siguiente repositorio de GitHub " https://github.com/hwcrane/atari-DQL ". En este repositorio, se encuentra un modelo ya entrenado, junto a algunos videos de sus avances, pero para efectos de nuestro proyecto, comenzaremos uno desde cero. Además, la implementación consta de dos funciones principales, "Training" que como su nombre lo dice, es para entrenar a nuestro modelo, y con ayuda de Tensorboard, realiza graficas para observar su comportamiento, y "Testing" que sirve para observar mediante un video los nuevos avances que se han logrado.

En cuanto a los parámetros utilizados, en principio fueron los planteados por el paper " https://arxiv.org/pdf/1509.06461.pdf ", pero por motivos de ir aprendiendo como repercutía cada uno, a medida que avanzamos en el proyecto los fuimos cambiando.

Para entrenar el modelo de DDQN utilizaremos el ambiente de gymnasium "ALE/ BreakoutDeterministic-v4".

2.1. Especificaciones del sistema:

La experimentación y el entrenamiento se hicieron en una notebook con sistema operativo Windows 10 y las siguientes especificaciones:

- Procesador: Intel(R) Core(TM) i5-10300H CPU @ 2.50GHz.
- Memoria RAM: 16 GB de 2833 MHz.
- GPU: NVIDIA GeForce GTX 1650.

2.2. Especificaciones de la implementación:

Se utilizo Python 3.8.2 y se aprovechó la tecnología CUDA y cuDNN en un entorno con las siguientes librerías:

- Numpy 1.24.3
- Gymnasium 0.29.1
- Pillow 10.1.0
- Torch 2.1.2 + cu121
- Torch-tb-profiler 0.4.3
- Torchaudio 2.1.2 + cu121
- Torchvision 0.16.2
- Tensorboard 2.13.0

2.3. Entrenamiento:

Para el entrenamiento de nuestra IA, fuimos variando los parámetros del archivo “BreakoutConfig.ini” y, además, acorde a nuestra disposición, variamos la cantidad de episodios por entrenamiento (episodios se refiere a partidas jugadas). Los siguientes entrenamientos no son acumulativos, ósea, contemplan el aprendizaje que obtuvo el modelo, pero no sumamos las horas y episodios que se hicieron anteriormente. Adicionalmente, hay que mencionar que cada entrenamiento tendrá un link a una carpeta de Google Drive con los parámetros utilizados y su video respectivamente. ¡Link a la carpeta madre! (en esta carpeta también se encuentra todo lo que respecta al código utilizado)

2.3.0. 0 episodios (0 minutos) ¡Link aquí!

La IA aún no sabe moverse, no hay más.

2.3.1. 500 episodios (13 minutos) ¡Link aquí!

La IA aprendió a moverse, mas no a evitar que la pelota caiga.

En la primera instancia de entrenamiento usamos los parámetros que estaban en la implementación que son similares a los que sugiere el paper de double DQN.

- eps_start = 1.
- eps_end = 0.02
- eps_decay = 1000000
- memory_size = 100000
- learning_rate = 0.00001
- initial_memory = 10000
- gamma = 0.99
- target_update = 40000
- batch_size = 32

2.3.2. 1000 episodios (37 minutos) ¡Link aquí!

La IA busca al menos romper un ladrillo, y si seguido pierde, no presiona el botón para que salga la nueva bola, puesto que como este es un evento aleatorio, aun no lo tiene incorporado.

En este caso, aumentamos épsilon de finalización a 0.2 para seguir obteniendo nuevos resultados con aleatoriedad y así que nuestro modelo siga aprendiendo.

2.3.3. 2000 episodios (1,2 horas) ¡Link aquí!

La IA aprendió la estrategia de romper los ladrillos que están más arriba, puesto que estos dan una mayor recompensa, pero aún no aprende a hacer rebotar la pelota de manera continua, además, si se queda con una vida, evita iniciar su última oportunidad para evitar penalizaciones.

Para este, mantenemos un épsilon moderado entre 0.9 y 0.7.

2.3.4. 500 episodios (25 minutos) ¡Link aquí!

No hay un aprendizaje substancial, pero el promedio de resultados en grafico mejoraron.

Este entrenamiento fue corto, con la razón de que probamos disminuir la cantidad de pasos de entrenamiento hasta llegar al mínimo épsilon a 100000 y también disminuir épsilon de finalización a 0.6

2.3.5. 1000 episodios (1,4 horas) ¡Link aquí!

La IA aprendió a hacer rebotar la pelota continuamente, pero, al momento en que la bola aumenta su velocidad, esta no sabe reaccionar y pierde la mayoría de las veces. Además, ahora juega todas sus vidas, sin dejar el juego en pausa indefinida, esto debe ser porque ve la última vida como una oportunidad de aumentar su puntuación de manera significativa.

Disminuimos épsilon de inicio a 0.8, aumentamos épsilon de finalización a 0.7 y aumentamos épsilon_decay a 500000.

2.3.6. 1000 episodios (1,7 horas) ¡Link aquí!

La IA aun no aprende una estrategia en específico, pero su capacidad de reacción al momento de cambios de velocidad ha mejorado.

Disminuimos épsilon de inicio a 0.7 y épsilon de finalización a 0.2.

2.3.7. 4200 episodios (8,9 horas) ¡Link aquí!

La IA, además de mejorar sus habilidades parece estar desarrollando la estrategia de crear un agujero hacia la zona más alta de los ladrillos, para así obtener una puntuación mucho más alta.

Aumentamos épsilon de inicio a 1 y disminuimos épsilon de finalización a 0.02, además, épsilon_decay ahora es 800000.

2.3.8. 2200 episodios (5 horas) ¡Link aquí!

Según el grafico de recompensa-episodio, hubo intentos en que la IA alcanzo puntajes de 300, por lo que suponemos que logro llevar a cabo la estrategia, pero en nuestro video, que considera el modelo sin acciones aleatoria, aun no lo consigue, por lo que suponemos que aún no adquiere la estrategia por completo.

Aumentamos épsilon de finalización a 0.2, reiniciamos épsilon_decay y reajustamos la taza de aprendizaje a 0.00008.

2.3.9. 500 episodios (2.5 horas) ¡Link aquí!

La IA aprendió la estrategia de hacer un túnel para llegar a la parte superior y dejar la bola obteniendo puntuación, pero igual sigue cometiendo errores graves, como lo es el hecho de que una vez pierde, no presiona el botón para comenzar una nueva partida.

Configuramos épsilon de inicio en 0.3 y épsilon de finalización en 0.05 además épsilon_decay en 500000.

2.3.10. 1500 episodios (3 horas) ¡Link aquí!

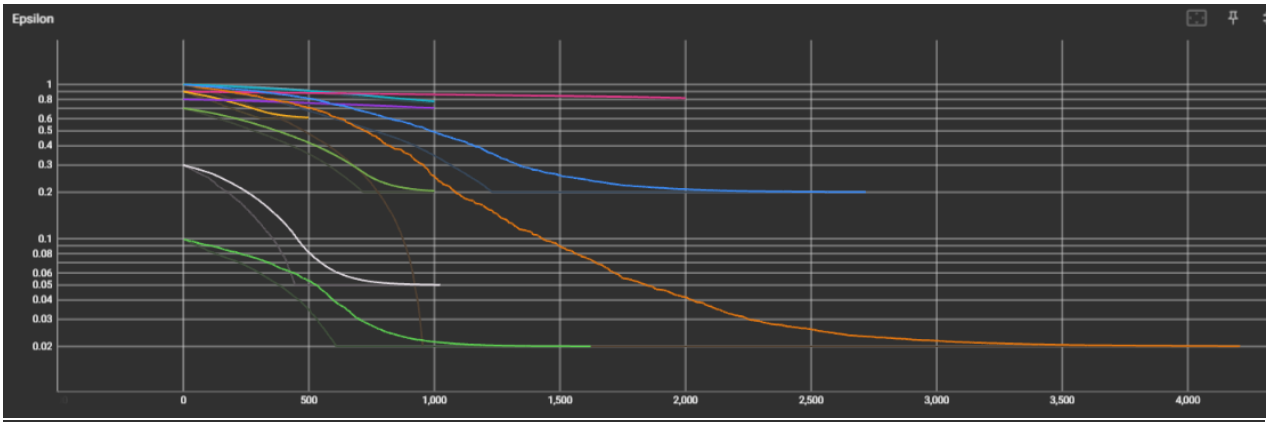
La IA mejoro la estrategia de hacer un túnel, pero, sigue cometiendo errores graves, como no presionar el botón para comenzar una nueva partida.

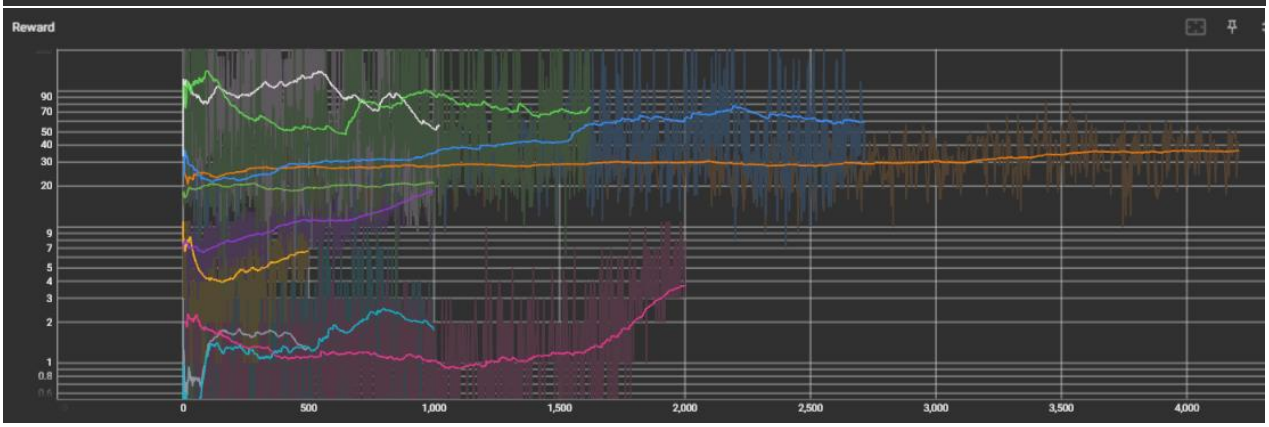
Configuramos épsilon de inicio en 0.1 y épsilon de finalización en 0.02 además épsilon_decay en 500000.

2.4. Resultados

Tras 13400 episodios y 24.95 horas de entrenamientos, tenemos los siguientes resultados:

Entrenamiento 1 al 10 toman los siguientes colores respectivamente:





Claramente, estos resultados pueden mejorar, lo que requerirá más tiempo y episodios, pero por temas de tiempo, no fue posible llegar a un mejor modelo.

4. ¡BankHeist!

El videojuego que originalmente queríamos resolver era BankHeist, llegar a un estado desempeño humano para este juego es un problema que escapa del alcance y enfoque de este proyecto, sin embargo, vamos a intentar de todas formas. Al utilizar Deep Q-Learning para resolver BankHeist surge un problema con el modelo y la sobreestimación de los valores Q, según el paper de double DQL, esto se genera por calcular la siguiente observación y su valor Q en el mismo modelo. Double Deep Q-Learning propone una solución, ya que utiliza una combinación de los modelos pre y post optimización para calcular ciertos valores utilizados en el ajuste del mismo modelo.

Para entrenar el modelo de double DQN utilizaremos el ambiente de gymnasium "ALE/BankHeist-v5" (en un principio se utilizó "BankHeistDeterministic-v4" por error), también y en vista de BankHeist es más complejo y es más difícil conseguir puntaje a largo plazo, se reajustaron los parámetros a lo siguiente:

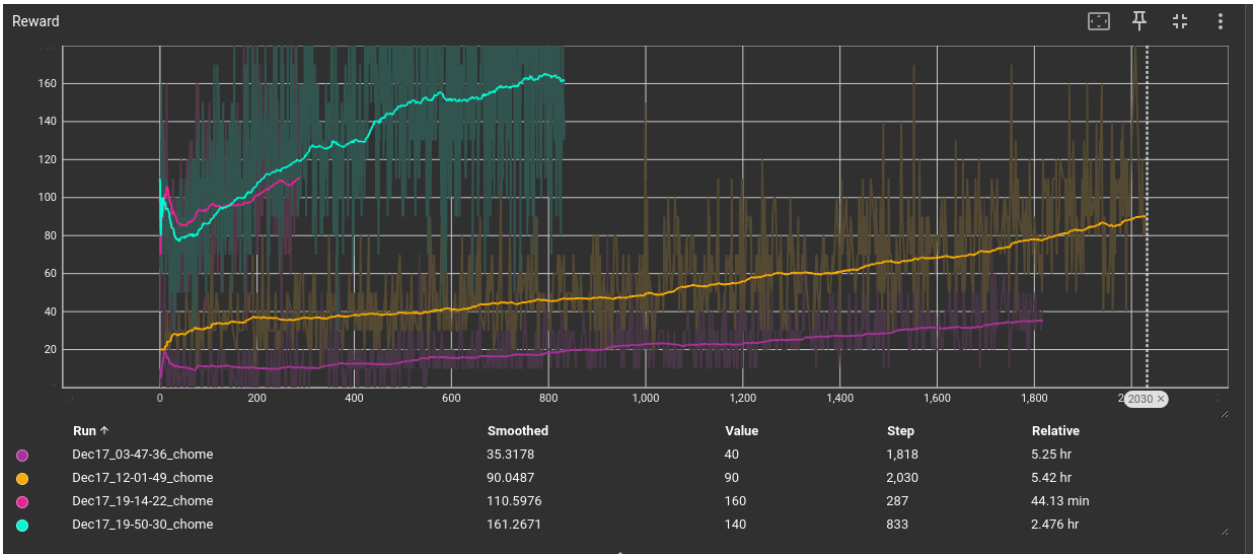
- eps_start = 1
- eps_end = 0.02
- eps_decay = 2.000.000
- memory_size = 100000
- learning_rate = 0.00003
- initial_memory = 10000
- gamma = 0.999
- target_update = 40.000
- batch_size = 32

El entrenamiento se realizó en un servidor dispuesto por la Universidad a través de una conexión ssh a @chome.inf.udec.cl, no es lo mejor para entrenamiento de IA, pero ya que este no es el punto central del proyecto, se decidió usarlo para no interferir con el resto del proyecto.

Especificaciones del servidor: 80 núcleos Intel Xeon Golde 5320T a 2.3 GHz y aproximadamente 263 GB de memoria RAM.

4.2. Resultados Parciales

Se ejecutaron múltiples instancias de entrenamiento estos son los resultados de cada una:



El puntaje promedio máximo obtenido fue de 160. Este se obtuvo con un coeficiente de aprendizaje y un valor de gamma altos, lo que permitió al agente entender que tiene que realizar varias acciones para poder conseguir puntaje.

En la última versión del modelo, el agente aprendió a abusar una mecánica de cambio de ciudad para reposicionar los bancos e ir a por el más cercano, probablemente tome muchas iteraciones para que descubra que puede buscar bancos más lejanos, evitar policías, destruir policías y cambiar de ciudad solo cuando sea más conveniente.

5. Conclusión

Esta instancia, ha sido una buena oportunidad para introducirnos en el aprendizaje por refuerzo y lo que lo rodea, a pesar de que nuestro proyecto no buscaba realizar un avance en el campo de la inteligencia artificial, cumplimos nuestros objetivos de aprender sobre métodos de IA en la práctica, comprender las dificultades involucradas, y experimentar con parámetros específicos de Double Deep Q-Learning.

Vimos como el modelo aprendía distintas estrategias y como exploraba nuevas acciones, en particular, cerca del episodio 12.000 aprendió a tirar la pelota detrás de los ladrillos para ganar mucho puntaje, pero luego siguió explorando con simplemente rebotar la pelota de frente y sin dejarla caer.

Un comportamiento interesante de ver es el de la acción de lanzar la pelota, en muchas ocasiones a nuestro agente le cuesta tomar la acción de lanzar la pelota, sucede que, debido a la política épsilon gready, el agente no necesita lanzar la pelota por su cuenta, ya que, estadísticamente una acción aleatoria lo va a hacer en un incluso para los valores más bajos de épsilon (0.02), de forma que si analizamos al agente jugar sin esta política, muchas veces se va a quedar parado sin lanzar la pelota después de perder una vida.

6. Referencias

- atari-DQL. Repositorio de GitHub: https://github.com/hwcrane/atari-DQL
- Deep Q Learning (DQN) Coding Tutorial - Reinforcement Learning. Videos de brthor: https://www.youtube.com/@brthor1117/playlists
- Gymnasium. API utilizado: https://gymnasium.farama.org
- Human-level control through deep reinforcement learning. Paper de múltiples autores: https://www.nature.com/articles/nature14236
- Deep Reinforcement Learning with Double Q-learning. Paper de Google DeepMind: https://arxiv.org/pdf/1509.06461.pdf
- Playing Atari with Deep Reinforcement Learning. Paper de DeepMind Technologies: https://arxiv.org/pdf/1312.5602.pdf