UNIVERSIDAD DE CONCEPCIÓN
Departamento de Ingeniería Informática y
Ciencias de la Computación

Inteligencia Artificial

Proyecto Semestral: Haxball

Profesor: Julio Godoy Del Campo
Alumnos: Vicente Ríos Adasme 2020449783
Tomás Cárdenas Oyarce 2020449783
Pablo Sanhueza Yévenes 2021439005

12 de julio de 2024

1. Introducción

La *inteligencia artificial* (IA) ha avanzado significativamente en diversos campos, transformando la manera en que se abordan problemas complejos. Un área donde la IA ha mostrado un gran potencial es en los videojuegos. Al integrar IA, es posible crear experiencias de juego más dinámicas, desafiantes y envolventes.

Con el aprendizaje por refuerzo (RL), un agente aprende comportamientos óptimos a través de prueba y error, recibiendo retroalimentación de su entorno. Uno de los algoritmos fundamentales de RL es Q-learning, que permite a un agente maximizar las recompensas acumulativas al aprender el valor de diferentes acciones en varios estados. Sin embargo, Q-learning enfrenta dificultades cuando se trata de espacios de estados de alta dimensión, que son comunes en juegos complejos. En estos contextos, el agente debe lidiar con una gran cantidad de posibles estados y acciones, lo que hace que la tabla Q utilizada en Q-learning sea impracticable debido a su tamaño y complejidad.

Para abordar estos desafíos, se utiliza el algoritmo *Deep Q-Network* (DQN), que combina Q-learning con redes neuronales profundas. Las redes neuronales permiten al agente aproximar los valores de Q para estados de alta dimensión, superando así las limitaciones del Q-learning tradicional. DQN utiliza una red neuronal para generalizar y estimar los valores de acción-estado, lo que reduce significativamente el problema de la dimensionalidad y permite que el agente maneje de manera efectiva entornos complejos y continuos. Por lo tanto, en contextos donde el espacio de estados es vasto y continuo, como en muchos videojuegos modernos, DQN se presenta como una solución más efectiva y eficiente en comparación con el Q-learning tradicional.

2. Descripción del problema

Haxball es un videojuego multijugador basado en navegador web, donde equipos de diferente cantidad de jugadores compiten en una cancha 2D para anotar goles golpeando una pelota virtual con sus pequeños avatares.

El problema que se abarca consiste en:

- Hay dos equipos: Rojo y Azul, con un jugador cada uno.
- El jugador del equipo Azul será el 'agente' que será entrenado mediante DQN.
- El jugador del equipo Rojo puede ser controlado por una persona, un bot *común*[1], o bien, otro agente entrenado.
- **Se tiene el objetivo de entrenar al agente para que juegue eficazmente el juego Haxball.**

'Eficazmente' en el sentido de que, a diferencia de los bots comunes, este debe representar un comportamiento estratégico y adaptativo, tomando decisiones basadas en el análisis del estado del juego para maximizar la probabilidad de anotar goles y minimizar las oportunidades del oponente. El agente debe ser capaz de anticipar los movimientos del adversario, posicionarse adecuadamente en el campo, y tomar acciones que optimicen tanto la defensa como el ataque. Este enfoque debe demostrar un nivel de inteligencia y capacidad de aprendizaje que supere a los bots predefinidos, mostrando mejoras continuas a través de la experiencia adquirida durante los juegos.

[1]Se dice bot común o, *common bot* a un bot que ya fue creado con anterioridad en el gimnasio HaxBallGym, que no incluyen en su implementación, por sí solos, métodos de aprendizaje. Tal es el caso de un bot con movimientos aleatorios, o bien, de *ChaseBot*, por ejemplo.

3. Solución

3.1. Propuesta de Solución

En este proyecto, se propone utilizar el algoritmo DQN para entrenar al agente. Para implementar esta solución, se utilizará una versión adaptada llamada HaxballGym, obtenida de GitHub, que facilita el entrenamiento de un agente de RL en el juego HaxBall. Dado que Haxball es un juego con un estado continuo, donde las posiciones y velocidades de los jugadores y la pelota pueden tomar cualquier valor dentro de un rango continuo, el uso de Q-learning no es viable debido a la imposibilidad de manejar un espacio de estados tan grande y continuo, pues, en ese caso la Q-table tendría un tamaño infinito.

El algoritmo DQN se emplearía para abordar este desafío, combinando Q-learning con redes neuronales profundas para manejar los espacios de estados de alta dimensión presentes en Haxball. HaxballGym actúa como un puente entre el juego y los algoritmos de RL, permitiendo una experimentación fluida y el desarrollo de agentes de IA capaces de jugar de forma autónoma.

Se ajustaría el código de HaxballGym para mejorar la interacción del agente con el entorno del juego y optimizar su proceso de aprendizaje. El agente será entrenado utilizando DQN, discretizando las acciones posibles del jugador en tuplas que representan movimientos y acciones específicas, como moverse a la izquierda, derecha, arriba, abajo, o disparar. La discretización de estas acciones permite simplificar el espacio de acción, haciendo que el entrenamiento sea más rápido y sencillo. Esto facilita que el agente seleccione acciones óptimas basadas en el análisis del estado del juego, representado por las posiciones y velocidades de los jugadores y la pelota.

En DQN, una red neuronal se utiliza para aproximar la función Q, en lugar de utilizar una Q-table explícita. La red neuronal toma como entrada el estado del entorno y produce como salida los valores Q para todas las acciones posibles. La actualización de los pesos de la red neuronal se basa en minimizar la diferencia entre el valor Q predicho y el valor objetivo:

3.2. Sobre el DQN

En DQN, una red neuronal se utiliza para aproximar la función Q, en lugar de utilizar una Q-table explícita. La red neuronal toma como entrada el estado del entorno y produce como salida los valores Q para todas las acciones posibles. La actualización de los pesos de la red neuronal se basa en minimizar la diferencia entre el valor Q predicho y el valor objetivo:

$$\text{Pérdida} = \left(r + \gamma \max_{a'}(Q(s', a'; \theta^-)) - Q(s, a; \theta) \right)^2$$

donde:

- θ son los pesos de la red neuronal.
- θ^- son los pesos de la red neuronal objetivo, que se actualizan periódicamente para estabilizar el entrenamiento.

La red neuronal en DQN generalmente consiste en varias capas completamente conectadas (fully connected layers). En el ejemplo proporcionado, la red DQN tiene una estructura de capas que incluye:

1. Una capa de entrada que toma el estado del juego.
2. Dos capas ocultas con unidades ReLU (Rectified Linear Unit).
3. Una capa de salida que proporciona los valores Q para cada acción posible.

DQN introduce dos conceptos clave para mejorar la estabilidad y la eficiencia del entrenamiento:

1. **Memoria de Repetición (Replay Memory):** Una memoria que almacena las transiciones $(s, a, r, s', \text{done})$ y permite al agente entrenar en muestras aleatorias de esta memoria. Esto rompe la correlación temporal de las muestras y reduce la variabilidad de las actualizaciones de los pesos.
2. **Red Objetivo (Target Network):** Una copia de la red Q que se utiliza para calcular los valores objetivos $r + \gamma \max_{a'} Q(s', a'; \theta^-)$. Esta red se actualiza periódicamente copiando los pesos de la red Q principal. Esto ayuda a evitar oscilaciones y divergencias durante el entrenamiento.

La política de selección de acciones en DQN a menudo sigue una estrategia epsilon-greedy, donde:

- Con probabilidad ϵ, el agente selecciona una acción aleatoria para explorar el entorno.
- Con probabilidad $1 - \epsilon$, el agente selecciona la acción con el valor Q más alto para explotar el conocimiento actual.

El valor de ϵ se reduce gradualmente durante el entrenamiento (decay) para favorecer la explotación sobre la exploración a medida que el agente aprende.

Sin embargo, para este proyecto, dado que se quiere tener resultados más rápidos, se trabajará de manera diferente la exploración. En lugar de seleccionar acciones completamente aleatorias con probabilidad ϵ, se basa en los movimientos del Chase Bot. Chase Bot es un bot que simplemente se acerca a la pelota y le pega cuando está lo suficientemente cerca. Con el paso de los episodios, ϵ irá disminuyendo y el agente tomará cada vez más acciones basadas en lo que ha aprendido, en lugar de las acciones del Chase Bot.

El enfoque específico de este proyecto es:

- **Exploración:** Con probabilidad ϵ, el agente utiliza la lógica del Chase Bot para seleccionar una acción. Esto implica moverse hacia la pelota y golpearla cuando esté lo suficientemente cerca.
- **Explotación:** Con probabilidad $1 - \epsilon$, el agente utiliza la red DQN para seleccionar la acción que maximice el valor Q.

Este método permite una exploración inicial más guiada y eficiente, basada en un comportamiento predefinido que es razonable en el contexto del juego, y gradualmente transiciona hacia una explotación de las acciones aprendidas por el agente a medida que ϵ decrece.

3.3. Implementación

En la implementación del algoritmo DQN (Deep Q-Network) para entrenar un agente capaz de jugar Haxball, se han seguido una serie de pasos y estrategias clave que se describen a continuación. Este proyecto busca desarrollar un agente de inteligencia artificial que pueda jugar Haxball de manera eficiente, utilizando técnicas avanzadas de aprendizaje por refuerzo.

3.3.1. Definición de la Red Neuronal

La red neuronal utilizada en DQN está compuesta por varias capas completamente conectadas. En esta implementación, la red tiene una capa de entrada que toma el estado del juego, dos capas ocultas con unidades ReLU (Rectified Linear Unit), y una capa de salida que proporciona los valores Q para cada acción posible. La arquitectura de la red neuronal permite manejar el espacio de estados de alta dimensión de manera eficiente.

ReLU (Rectified Linear Unit): ReLU es una función de activación utilizada en las capas ocultas de la red neuronal. Se define como:

$$\text{ReLU}(x) = \max(0, x)$$

ReLU es popular en redes neuronales debido a su simplicidad computacional y su eficacia en la reducción del problema de gradientes desvanecientes. Esto permite entrenar redes profundas de manera más eficiente y con mejores resultados.

Algoritmo 1 Definición de la red neuronal DQN

```
function DQN(input_dim, output_dim)
    layers ← Sequential()
    layers.add_layer(Linear(input_dim, 128))
    layers.add_layer(ReLU())
    layers.add_layer(Linear(128, 128))
    layers.add_layer(ReLU())
    layers.add_layer(Linear(128, output_dim))
    return layers
end function
function FORWARD(x)
    return layers(x)
end function
```

3.3.2. Inicialización y Configuración

El proceso comienza con la inicialización del entorno de juego utilizando las bibliotecas ursinaxball y haxballgym. Estas bibliotecas permiten simular el juego Haxball en un entorno controlado y programable. Se configura el entorno de juego con los parámetros deseados, como el tipo de mapa, las reglas del juego y las condiciones del terminal.

Además, se inicializan las redes neuronales para el agente, incluyendo la red Q y la red objetivo. La red Q es la red principal que se entrena durante el proceso, mientras que la red objetivo se actualiza periódicamente para estabilizar el entrenamiento. También se inicializa una memoria de repetición, que almacena las transiciones del agente para ser usadas en el entrenamiento.

Algoritmo 2 Inicialización y Configuración

```
Inicializar las bibliotecas y módulos necesarios
Configurar el entorno de juego
Inicializar la red neuronal DQN y la red objetivo con pesos iniciales iguales
Inicializar el optimizador Adam con tasa de aprendizaje learning_rate
Establecer parámetros del agente:
    γ ← 0,99                                          ▷ Factor de descuento
    ϵ ← 1,0                    ▷ Valor inicial de epsilon para la política epsilon-greedy
    ϵ_decay ← 0,995                                   ▷ Decaimiento de epsilon por episodio
    ϵ_min ← 0,01                                      ▷ Valor mínimo de epsilon
    memory ← deque(maxlen=20000)                      ▷ Memoria de repetición
    batch_size ← 128                                  ▷ Tamaño del mini-lote
    learn_step ← 0                                    ▷ Paso de aprendizaje actual
    learn_frequency ← 4                               ▷ Frecuencia de actualización de la red
Configurar gráficos en tiempo real para recompensas promedio y acumulativas
```

3.3.3. Definición de la Red Neuronal

La red neuronal utilizada en DQN está compuesta por varias capas completamente conectadas. En esta implementación, la red tiene una capa de entrada que toma el estado del juego, dos capas ocultas con unidades ReLU (Rectified Linear Unit), y una capa de salida que proporciona los valores Q para cada acción posible. La arquitectura de la red neuronal permite manejar el espacio de estados de alta dimensión de manera eficiente.

3.3.4. Definición del Agente DQN

El agente DQN se define con varios parámetros importantes, como la tasa de aprendizaje, el factor de descuento (γ), y el valor inicial de ϵ para la política epsilon-greedy. El agente tiene métodos para recordar transiciones, actualizar la red mediante gradiente descendente, seleccionar acciones y aplicar la lógica del Chase Bot para la exploración.

- **Memoria de Repetición:** El agente almacena las transiciones (estado, acción, recompensa, estado siguiente, hecho) en una memoria de repetición. Esta memoria permite al agente entrenar en muestras aleatorias de sus experiencias, rompiendo la correlación temporal y mejorando la estabilidad del entrenamiento.
- **Red Objetivo:** La red objetivo es una copia de la red Q que se utiliza para calcular los valores objetivos. Esta red se actualiza periódicamente copiando los pesos de la red Q principal, lo que ayuda a evitar oscilaciones y divergencias durante el entrenamiento.
- **Política Epsilon-Greedy:** La política de selección de acciones sigue una estrategia epsilon-greedy. Con probabilidad ϵ, el agente utiliza la lógica del Chase Bot para seleccionar una acción, explorando el entorno de manera más eficiente. Con probabilidad $1 - \epsilon$, el agente selecciona la acción que maximice el valor Q, explotando el conocimiento actual.

3.3.5. Entrenamiento del Agente

El ciclo de entrenamiento del agente se lleva a cabo durante un número determinado de episodios. En cada episodio, se reinicia el entorno de juego y se obtiene el estado inicial. El agente interactúa con el entorno seleccionando acciones y recibiendo recompensas basadas en su desempeño. Las transiciones se almacenan en la memoria de repetición y se usan para entrenar la red Q mediante gradiente descendente.

- **Actualización de la Red:** Las actualizaciones de la red Q se realizan minimizando la pérdida entre el valor Q predicho y el valor objetivo. Cada cierto número de pasos, la red objetivo se actualiza copiando los pesos de la red Q principal.
- **Decaimiento de Epsilon:** El valor de ϵ se reduce gradualmente durante el entrenamiento, favoreciendo la explotación sobre la exploración a medida que el agente aprende.
- **Gráficos en Tiempo Real:** Se utilizan gráficos en tiempo real para mostrar la recompensa promedio y acumulativa durante el entrenamiento. Esto permite visualizar el progreso del agente y ajustar los parámetros de entrenamiento según sea necesario.